Комп'ютинг

Алгоритми і структури даних

Підручник

Computing

Міністерство освіти та науки України

Н.Б. Шаховська, Р.О. Голощук

Алгоритми і структури даних

ПОСІБНИК

СЕРІЯ "КОМП'ЮТИНГ"

За загальною редакцією д.т.н., професора В. В. Пасічника

Затверджено Міністерством освіти і науки України

Видавництво «Магнолія 2006»

Львів – 2011

УДК 004.422.63(075)
ББК 32.973.2-018 я 7
Ш 32

Гриф надано Міністерством освіти і науки України як навчальний посібник для вищих навчальних закладів (лист № 1.4/18-Г-2635 від 04.12.08р.)

Ш 32

Рецензенти:

М.О. Медиковський – д.т.н., професор, професор кафедри автоматизованих систем управління Національного університету „Львівська політехніка";

Я.М. Матвійчук – д.т.н., професор, завідувач кафедри інформаційних систем та технологій інституту підприємництва та перспективних технологій при Національному університеті „Львівська політехніка";

Г.Г. Цегелик – завідувач кафедра математичного моделювання соціально-економічних процесів Львівського Національного університету.

ISBN 978-966-2025-95-8

У посібнику розглядаються статичні й динамічні структури даних і методи роботи з деревами та графами. Проаналізовано алгоритми пошуку та сортування. Уводиться поняття хеш-функції та подаються правила її вибирання.

Проаналізовано поняття обчислювальної складності, визначено класи алгоритмів та задач.

Буде корисним для студентів, що навчаються за напрямом підготовки фахівців «Комп'ютерні науки», «Системний аналіз».

ISBN 978-966-2025-95-8

УДК 004.422.63(075)
ББК 32.973.2-018 я 7
Ш 32

ЗМІСТ

ПЕРЕДМОВА НАУКОВОГО РЕДАКТОРА СЕРІЇ ПІДРУЧНИКІВ «КОМП'ЮТИНҐ»

Шановний читачу!

Започатковуючи масштабний освітньо-науковий проект підготовки і видання серії сучасних підручників під загальним гаслом «КОМП'ЮТИНҐ» і загальним методичним патронуванням його Інститутом інноваційних технологій та змісту освіти МОН України, мені, як ініціатору та науковому керівнику, неодноразово доводилося прискіпливо аналізувати загальну ситуацію в царині сучасного україномовного підручника комп'ютерно-інформатичного профілю. Загалом, позитивна тенденція останніх років ще не співмірна з надзвичайно динамічним розвитком як освітньо-наукової та виробничої сфери комп'ютингу, так і стрімким розширенням потенційної цільової читацької аудиторії цього профілю. Іншими словами, попередній аналіз засвідчує наявність значного соціального замовлення під реалізацію пропонованого Вашій увазі проекту.

Ще одним фактором формування освітньо-наукової ініціативи, пропонованої групою відомих вітчизняних науковців-педагогів та практиків, які організовують наукові дослідження, готують фахівців та провадять бізнес у галузі комп'ютингу, постало завдання широкомасштабного включення Української вищої школи до загальноєвропейських і всесвітніх об'єднань, структур і асоціацій. Виконуючи функцію науково-технічного локомотиву суспільства, галузь комп'ютингу невідворотно мусить зіграти роль активного творця загальної освітньо-наукової платформи, яка має бути методологічно об'єднавчою та професійно-інтеграційною основою для багатьох сфер людської діяльності.

Третім суттєвим фактором, який спонукав започаткувати пропоновану серію підручників, є те, що об'єктивно визріла ситуація, коли фахівцям та науковцям треба подати чіткий сигнал щодо науково-методологічного осмислення та викладення базових знань галузі комп'ютингу як освітньо-наукової, виробничо-економічної та сервісно-обслуговувальної сфери.

Читач, безсумнівно, зверне увагу на нашу послідовну промоцію нового терміну КОМП'ЮТИНҐ (computing, англ.), який є вдалим та комплексно узагальнювальним для означення галузі знань, науки, виробництва, надання відповідних послуг і сервісів. Видається доречним подати ретроспективу як самого терміну комп'ютинг, так і широкої освітньої, наукової, бізнесової та виробничої сфери діяльності, що іменується комп'ютингом.

Уперше термін комп'ютинг уведений 1998 року Яном Фостером з аргонської національної лабораторії Чікагського університету та Карлом Кесельманом з інституту інформатики штату Каліфорнія (США) і запропонований для означення комплексної галузі знань, яка включає проектування та побудову апаратних і програмних систем для широкого кола застосувань: вивчення процесів, структур і керування інформацією різних видів; виконання наукових досліджень із застосування комп'ютерів та їх інтелектуальності; створення і використання комунікаційних та демонстраційних засобів, пошуку й збирання інформації для конкретної мети і т. ін.

У подальшому сфера використання терміну суттєво розширилась, зокрема, в освітньо-науковій царині. Його почали використовувати для означення відповідної галузі знань, для якої періодично (орієнтовно щодесять років) провідними університетами та

професійними асоціаціями фахівців розробляються та імплементуються навчальні плани і програми, які в подальшому набувають статусу міжнародно визнаних освітньо-професійних стандартів. Зокрема, варто акцентувати увагу на версіях підсумкового документу "Computing CURRICULA" 2001 року. За окремими повідомленнями можна стверджувати, що черговий збірник стандартів "Computing CURRICULA" буде поданий професійному загалу до 2011 року. Перше організаційне засідання відповідних фахових робочих груп відбулося у Чікагському університеті влітку 2007 року.

Для формування цілісного однорідного подання суті КОМП'ЮТИНҐУ ми базуємось на сучасних наукових уявленнях з максимально можливим строгим покомпонентним викладенням основних базових означень та понять, які склались історично і є загально визнаними у професійних колах. Водночас для побудови цілісної зваженої картини ми використали певні узагальнення та загальносистемні класифікаційні підходи.

Безсумнівно, що базовим та фундаментальним поняттям було, є і залишається поняття ІНФОРМАТИКИ (informatique - франц.), як фундаментальної науки, котра вивчає найбільш загальні закони та закономірності процесів відбору, реєстрації, збереження, передавання, захисту, опрацювання та подання інформації. У такому сенсі як фундаментальна наука інформатика була подана в 70-х роках XX ст. При цьому хочу відразу ж застерегти від примітивного ототожнення, яке часто є наївно вживаним щодо еквівалентності понять «інформатика» (informatique - франц.) та «комп'ютерні науки» (computer science - англ.). Такі ототожнення з певною мірою наближення можливі щодо розширеного сучасного трактування інформатики як, загалом, прикладної науки про обчислення, збереження, опрацювання інформації та побудову прикладних інформаційних технологій і систем на їх базі. Таке трактування є характерним в ряді європейських країн. Строге ж означення та подання предмету досліджень інформатики, а саме – інформації, має справу з фундаментальним не редукованим поняттям і фіксується у словниках як «informatio» (лат.) – відомості, повідомлення. Вивченням та всестороннім аналізом сутності інформації опікується наука, що називається „теорія інформації". На нашу думку, основною принциповою відмінністю між інформатикою та комп'ютерними науками є те, що перша в своєму первинному поданні відноситься до категорії фундаментальних наук, як то фізика, математика, хімія і т. ін. У той же час комп'ютерні науки загалом за своєю сутнісною природою та всіма наявними ознаками належать до категорії прикладних наук, які базуються на фундаментальних законах та закономірностях інформаційних процесів, котрі вивчаються в рамках фундаментальної науки інформатики.

Особливо наголосимо на тому, що фундаментальна наука та її результати не призначені для безпосереднього промислового використання.

Для комп'ютерних наук характерною ознакою виділення їх у спектрі прикладних наук є об'єкт прикладення знань, умінь та навичок у контексті конкретного об'єкту – обчислювача (комп'ютера). Іншою відокремленою прикладною науковою галуззю, що базується на підвалинах інформатики, є розділ прикладних наук, основним об'єктом яких є сам процес обчислень. Це науки, які іменуються обчислювальними науками – «computationally science» - (англ.). Традиційно сюди відносять обчислювальну та комп'ютерну математику.

Третьою прикладною науковою галуззю, яка ґрунтується на фундаментальних законах інформатики, є розділ прикладних наук, основним об'єктом яких є інформаційний

ресурс (у сучасній літературі часто вживається поняття «контент» (content-англ.) у розумінні інформаційного наповнення. Ці прикладні науки одержали назву „інформаційні науки" (information science - англ.).

У галузі прикладних інформаційних наук базовий об'єкт досліджень, а саме інформаційний ресурс, подається, як правило, у формі даних та знань. За спрощеною формулою позначатимемо дані як матеріалізовану інформацію, тобто інформацію, яку подано на матеріальних носіях, знання як суб'єктивізовану інформацію, тобто інформацію, яка природно належить суб'єкту і, в традиційному розумінні, перебуває в людській пам'яті.

Узагальнюючи класифікаційно-ознакову схему, стверджуємо, що на базі фундаментальної науки ІНФОРМАТИКИ формуються три прикладні наукові галузі, а саме: комп'ютерні науки, обчислювальні науки та інформаційні науки з відповідними об'єктами досліджень у своїх сферах.

Ще раз підкреслимо, що результати фундаментальних наукових досліджень не призначені для безпосереднього промислового використання, у той же час результати прикладних наукових досліджень, як правило, призначені для створення й удосконалення нових технологій.

Гносеологічний аналіз подальшого формування інженерного рівня сфери КОМП'ЮТИНҐУ невідворотно веде до структурного подання базових типів інженерій, які трактуються у класичному розумінні. ІНЖЕНЕРІЯ (майстерний - від лат. ingeniosus) – це наука про проектування та побудову (чит. створення) об'єктів певної природи. У цьому контексті природними для сфери КОМП'ЮТИНҐУ є декілька видів інженерії. Мова йтиме про:

- КОМП'ЮТЕРНУ ІНЖЕНЕРІЮ (computer engineering, англ.), яка охоплює проблематику проектування та створення об'єктів комп'ютерної техніки;
- ПРОГРАМНУ ІНЖЕНЕРІЮ (software engineering, англ.), яка опікується проблематикою проектування та створення об'єктів, що іменуються програмними продуктами;
- ІНЖЕНЕРІЮ ДАНИХ ТА ЗНАНЬ (data & knowledge engineering, англ.), інженерія, яка опікується проектуванням та створенням інформаційних продуктів;
- інженерію, яка опікується проектуванням та створенням міжкомпонентних (інтерфейсних) взаємозв'язків і формуванням цілісних системних об'єктів і які усе частіше іменують СИСТЕМНОЮ ІНЖЕНЕРІЄЮ (systems engeneering, англ.).

У разі такого структурно-класифікаційного подання видів інженерій сфери комп'ютингу зазначимо, що кожен з них у цьому трактуванні є „відповідальним" за певний тип забезпечення, а саме апаратного (hardware, англ.), програмного (software, англ.), інформаційного (dataware, англ.) та міжкомпонентного (middleware, англ.). Інформаційну технологію (ІТ) можна трактувати як певну точку в чотиривимірному просторі зазначених інженерій. При цьому слід обов'язково зважити на певну частку наближення та інтерпретації цього простору як дискретного та неметричного.

У зв'язку з поширеним різночитанням і трактуванням поняття інформаційної технології (ІТ) видається необхідним детальніше подати сутнісну структуру цього терміну, використовуючи при цьому термінологічні статті популярного інформаційного ресурсу, яким є Wikipedia - (http://www.wikipedia.org/).

Технологія (від грецького téchne – мистецтво, майстерність, вміння та грецького logos – знання) – сукупність методів та інструментів для досягнення бажаного результату, спосіб перетворення чогось заданого в необхідне. Технологія – це наукова дисципліна, в рамках якої розробляються та удосконалюються способи й інструменти виробництва.

У широкому розумінні – це знання, які можна використати для виробництва продуктів (товарів та послуг) з економічних ресурсів. У вузькому розумінні технологія подається як спосіб перетворення речовини, енергії, інформації в процесі виготовлення продукції, опрацювання та переробки матеріалів, складання готових виробів, контроль якості та керування.

Технологія включає в себе методи, прийоми, режими роботи, послідовність операцій та процедур, вона тісно взаємопов'язана із засобами, що застосовуються, обладнанням, інструментами, використовуваними матеріалами. За методологією ООН технологія в чистому вигляді охоплює методи та техніку виробництва товарів і послуг (dissembled technology, англ.); втілена технологія охоплює машини, обладнання, споруди, виробничі системи та продукцію з високими техніко-економічними параметрами (embodied technology, англ.). Матеріальна технологія (МТ) створює матеріальний продукт. Інформаційна технологія (ІТ) створює інформаційний продукт на основі інформаційних ресурсів.

Інформаційні технології (ІТ) використовують комп'ютерні та програмні засоби для реалізації процесів відбору, реєстрації, подання, збереження, опрацювання, захисту та передавання інформації - інформаційного ресурсу у формі даних та знань - з метою створення інформаційних продуктів.

Аналітична картина видаватиметься незавершеною, якщо не позначити ще одну базову сутність сфери комп'ютингу, якою є інформаційна система. Не претендуючи на абсолютну точність пропонованого твердження, розглядатимемо інформаційну систему як множину координат у чотиривимірному просторі інженерій сфери комп'ютингу. Тобто інформаційну систему (ІС) подаємо як певний набір інформаційних технологій, що в комплексі зорієнтовані на досягнення певної системної мети, виконуючи задані функції та пропонуючи при цьому споживачам якісні інформаційні продукти та сервіси.

У свою чергу, для всіх штучних інформаційних систем притаманними є чотири життєвих фази їхнього формування та функціонування. Йдеться про фази системного аналізу, системного проектування, системної інтеграції та системного адміністрування, які генерують відповідні вимоги до професійної підготовки та практичної орієнтації фахівців у царині інформаційних систем. Ринок потребує системних аналітиків, системних проектувальників, системних інтеграторів та системних адміністраторів.

Комплексний виклад структурованого подання галузі КОМП'ЮТИНҐУ дозволяє, загалом, чіткіше уявити проблематику та тематику підручників, які будуть виходити у світ в однойменній освітньо-науковій серії. Для кращого розуміння в майбутньому ще раз наведемо означення сфери КОМП'ЮТИНҐУ як галузі знань (науки, виробництва, бізнесу та надання послуг), предметом якої є комплексні дослідження, розроблення, впровадження та використання інформаційних систем, складовими елементами яких є інформаційні технології, що реалізовані на основі сучасних інженерних досягнень комп'ютерної інженерії, інженерії програмного забезпечення, інженерії даних та знань, системної інженерії, які базуються на фундаментальних законах і закономірностях інформатики.

Автори підручників серії «КОМП'ЮТИНГ» пропонують значний перелік навчальних дисциплін, які, з одного боку, включаються до сфери комп'ютингу за означенням, а, з іншого боку, їх предмет ще не знайшов якісного висвітлення у вітчизняній навчальній літературі для вищої школи. Перший крок ми робимо у 2008 році, виданням принаймні десяти книг серії з подальшим її п'ятикратним розширенням до 2011 року. Структурно серія подається узагальненими профілями як то:

- ✓ *фундаментальні проблеми комп'ютингу;*
- ✓ *комп'ютерні науки;*
- ✓ *комп'ютерна інженерія;*
- ✓ *програмна інженерія;*
- ✓ *інженерія даних та знань;*
- ✓ *системна інженерія;*
- ✓ *інформаційні технології та системи.*

При цьому зауважу, що наведені укрупнені профілі серії підручників загалом співпадають з профілями бакалаврів, зафіксованих у підсумковому звіті "Computing CURRICULA" редакції 2006 року. Ми розуміємо, що чітка завершена будівля комп'ютингу з'явиться лише в перспективі, а наша праця буде подаватись як активний труд будівничих з якнайшвидшого втілення в життя проекту цієї, без перебільшення, грандіозної будівлі сучасного інформаційного суспільства. Я запрошую потенційних авторів долучитися до цього освітньо-наукового проекту, а шановних читачів виступити в ролі творчих критиків та опонентів. Буду вдячний за Ваші побажання, зауваження та пропозиції.

З глибокою повагою науковий редактор серії підручників «КОМП'ЮТИНГ», д.т.н., професор Володимир Пасічник.

Вступ

Шановний читачу, подаючи на Твій критичний огляд результати нашої колективної праці сподіваюсь на взаєморозуміння та активну співпрацю. Безсумнівним є той факт, що частина структурно-логічної схеми освітньо-наукового напряму «Інженерія даних та знань», по якій ми презентуємо посібник, а саме – алгоритми та структури даних, є напрямком, який динамічно розвивається.

Хочемо спочатку зафіксувати базові позиції та мотиви, які слугували нам основними засадними принципами під час роботи над рукописом пропонованого посібника.

По-перше, започатковуючи проект, автори в повній мірі усвідомлювали велику складність та надвеликі обсяги інформаційних матеріалів, які видруковуються та з'являються на книжкових полицях маркетів, як то електронних так і традиційно - звиклих книгарень та книгозбірень.

По-друге, автори з великою уважністю знайомились з набутим досвідом, який зафіксований як результат аналогічних спроб інших авторських колективів та висвітлення результатів досліджень інших наукових шкіл та напрямків, а також спробували викласти своє бачення принципів побудови алгоритмів.

По-третє, автори, під час підготовки рукопису, сповідували ідею прикладання теоретичних знань та набутків до вирішення конкретних, прорахованих практикою, завдань. У посібнику є фрагменти програм, що подають прикладне вирішення тих чи інших задач. Програми написані на мовах Паскаль та Сі, як базових з основ програмування.

Начальний посібник складається з вступу, 10-ти розділів та додатків.

***У розділі 1** подано поняття алгоритму, визначено виконавців алгоритму та прокласифіковано алгоритми за типами і складністю. Приклади пояснюють, як розрізнити алгоритми за класами та як визначити їх властивості. Подано основні визначення теорії алгоритмів: машина Тьюринга, рекурсія, теза Чорча.*

***Розділ 2** присвячений розгляду різних типів структур даних. Описано рівні структур даних та здійснено їх класифікацію. Подано особливості відображення структур даних у пам'яті комп'ютера.*

***У розділі 3** описано одну з найпростіших та водночас найпоширеніших структур даних – масив. Описано операції, що можуть виконуватися над масивом. Показано способи подання масивів. Також здійснено опис структури даних типу «запис» і показано, що записи найчастіше подаються за допомогою масивів. Подано характеристики динамічних структур даних типу «стек», «черга», «дек» та показано відмінності між ними.*

***У розділі 4** описано одну з найпростіших та водночас найпоширеніших структур даних – масив. Описано операції, що можуть виконуватися над масивом. Показано способи подання масивів. Також здійснено опис структури даних типу «запис» і показано, що записи найчастіше подаються за допомогою масивів. Подано характеристики динамічних структур даних типу «стек», «черга», «дек» та показано відмінності між ними.*

***Розділ 5** присвячений описанню методів прискорення доступу до даних в таблицях. Одним з таких методів є використання хеш-функцій. Проте, під час*

хешування можуть виникати колізії даних та потреба у рехешуванні.

***Розділ 6** присвячений опису дерев та визначенню операцій над ними. Подано приклади роботи з деревами різної арності.*

***У наступному розділі** подано визначення графа та охарактеризовано його основні властивості. Показано способи подання графа та варіанти реалізації операцій над ним.*

***У розділі 8** розглянуто алгоритми пошуку стрічок: прямий пошук, Ахо-Корасика, Кнута-Моріса-Прата, Рабіна-Карпа, Боуєра-Мура. Подано алгоритми пошуку у масивах та списках.*

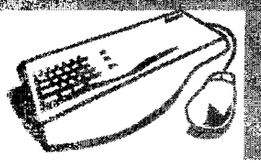
***Розділ 9** присвячений методам внутрішнього та зовнішнього сортування.*

***У розділі 10** подано опис жадібних алгоритмів.*

У додатках наведено завдання до лабораторних робіт.

З глибокою повагою
Автори.

РОЗДІЛ 1



БАЗОВІ ПОНЯТТЯ ТЕОРІЇ АЛГОРИТМІВ

- *Визначення інформації.*
- *Визначення алгоритму.*
- *Виконавці алгоритмів.*
- *Способи описування алгоритмів.*
- *Властивості алгоритмів.*
- *Типи алгоритмів.*
- *Машина Тьюринга.*
- *Поняття рекурсії.*
- *Алгоритми з поверненням назад.*
- *Теза Чорча.*

У розділі подано поняття алгоритму, визначено виконавців алгоритму та прокласифіковано алгоритми за типами і складністю. Приклади пояснюють, як розрізнити алгоритми за класами та як визначити їх властивості. Подано основні визначення теорії алгоритмів: машина Тьюринга, рекурсія, теза Чорча.

1.1. ВИЗНАЧЕННЯ ІНФОРМАЦІЇ

Термін «*інформація*» походить від латинського слова «*informatio*» – виклад, роз'яснення фактів, подій. Тому, в найширшому розумінні інформація трактується наступним чином.

✱ ***Інформація** – це відомості, пояснення, знання про всілякі об'єкти, явища, процеси реального світу.*

Доволі розповсюдженим є погляд на інформацію як на ресурс, аналогічний енергетичним, матеріальним, трудовим і грошовим ресурсам. Ця точка зору відображається у наступному трактуванні інформації.

Інформація – нові відомості, які дозволяють поліпшити процеси, пов'язані з перетворенням речовин, енергії та самої інформації.

Інформацію не можна відокремити від процесу інформування. В основі цього процесу лежить взаємозалежність пари об'єктів – джерела і споживача інформації.

Джерелами інформації, насамперед, є природні об'єкти: люди, тварини, рослини, планети тощо. Разом із цим, у міру розвитку науки і техніки, джерелами інформації стають наукові експерименти, машини, апарати, технологічні процеси. Значний також перелік об'єктів, що є споживачами інформації: люди, тварини, рослини, різноманітні технічні пристрої тощо.

Погляд на інформацію з точки зору її споживачів окреслюється у наступному понятті інформації.

Інформація – це нові відомості, прийняті, зрозумілі і оцінені її користувачем як корисні. Іншими словами, інформація – це нові знання, які отримує споживач (суб'єкт) в результаті сприйняття і опрацювання певних відомостей, тобто у результаті застосування до цих відомостей адекватних методів оперування ними (термін «адекватний» означає цілком відповідний; прирівняний; той, що влаштовує).

Адекватність є надзвичайно важливою вимогою до методів, які застосовуються до наявної інформації з метою отримання нової інформації. Так, для сприйняття відомостей, викладених на аркуші паперу незнайомою іноземною мовою, адекватним може бути метод перекладу зі словником. У той же час, для сприйняття усної іноземної мови цей метод є явно не адекватним і має бути застосований метод синхронного перекладу.

Інформатика використовує своє власне тлумачення терміну «інформація», яке є значно вужчим, ніж розглянуті вище. Це зумовлене тим, що при визначенні цієї категорії, інформатика не може базуватися на таких поняттях як знання, відомості, усунення невизначеності. Засоби комп'ютерної техніки володіють здатністю опрацьовувати інформацію автоматично, без участі людини, і про жодні знання або незнання тут мова йти не може. Ці засоби можуть працювати зі штучною, абстрактною і навіть із хибною інформацією, яка не має об'єктивного відображення ні у природі, ні у суспільстві.

Враховуючи зазначену специфіку комп'ютерної техніки, інформатика вилучає змістовні аспекти поняття інформації і використовує погляд на інформацію як на предмет певної діяльності. Найконцентрованіше це виражається у наступному понятті інформації.

✱ ***Інформація*** *– це відомості, які є об'єктом збирання, реєстрації, зберігання, передавання і перетворення.*

Основна маса інформації збирається, передається і опрацьовується у знаковій формі – числовій, текстовій, табличній, графічній і т.ін. Знакове подання інформації інформатика пов'язує з поняттям *«дані»*.

✱ ***Дані*** *– це відомості, подані у певній знаковій формі, придатній для передавання, інтерпретації та опрацювання людиною або автоматичним пристроєм.*

Взаємозв'язок між інформацією, даними і методами оперування ними характеризується низкою властивостей.

Інформація має динамічний характер. Вона не є статичним об'єктом, а динамічно змінюється її існує тільки у момент взаємодії даних і методів, тобто в момент протікання інформаційного процесу. Весь інший час вона перебуває у стані даних.

Зв'язок даних і методів носить діалектичний характер. Дані є об'єктивними, коли вони виникають у результаті реєстрації об'єктивно існуючих сигналів, викликаних змінами в матеріальних тілах або полях. У той же час, методи оперування даними в інформаційних процесах є суб'єктивними. Дійсно, інформаційний процес здійснюється за допомогою штучних або природних методів, в основі штучних методів лежать алгоритми, складені і підготовлені людьми (суб'єктами), в основі природних – біологічні властивості суб'єктів інформаційного процесу.

Дані надають інформацію лише в момент їх використання. Це означає, що збережувані дані залишаються лише даними доти, поки їх не використано в тих чи інших методах деяким суб'єктом інформаційного процесу (людиною або автоматичним пристроєм). Якщо дані зовсім не використовуються, то говорять, що вони мають нульову

інформативність, і вважають такі дані інформаційним шумом. Дані, що використовуються, називають інформативними.

Рівень інформативності даних залежить від ступеня адекватності методів, що застосовуються в інформаційних процесах.

Дані сприймаються їх отримувачем як певна змістовна інформація лише в тому випадку, коли в його «пам'яті» закладені поняття і моделі, що дозволяють зрозуміти зміст отриманих відомостей.

Коли дані опрацьовуються певними методами або евристиками, то на їх основі можна встановити залежності, послідовності чи висновки, які дозволяють здійснити підтримку процесу прийняття рішення. Такі дані називають знаннями. Рішення може прийматися як експертом предметної області (такі знання називають суб'єктизованими), так і комп'ютером (такі знання називають комп'ютеризованими).

1.2. ВИЗНАЧЕННЯ АЛГОРИТМУ

За визначенням А.П.Єршова, інформатика – це наука про методи подання, накопичення, передавання та опрацювання інформації за допомогою комп'ютера. Що таке інформація? Вважається, що інформація – це поняття, яке передбачає наявність матеріального носія інформації, джерела і передавача інформації, приймача і каналу зв'язку між джерелом і приймачем інформації.

Основними в загальній інформатиці є три поняття: задача, алгоритм, програма. Відповідно, маємо три етапи в розв'язуванні задач (зазначимо, що, з точки зору інформатики, розв'язати задачу – це отримати програму, тобто, забезпечити можливість отримати рішення за допомогою комп'ютера): постановка задачі, побудова і обґрунтування алгоритму, складання і налагодження програми. Оскільки програма – об'єкт гранично формальний, а тому точний (можливо не завжди прозорий, навантажений неістотними із змістовної точки зору деталями, але недвозначний) то, пов'язані з нею об'єкти також мають бути точними. Алгоритм – точний, чіткий і скінченний спосіб побудови результатів за точно вказаною в постановці задачі залежності від наявних аргументів.

Відповідно до етапів маємо три групи засобів інформатики: специфікація, алгоритмізація і програмування.

Для специфікації задач в курсі застосовані засоби типу рекурсивних співвідношень, рекурсивних визначень, а також прості інваріантні співвідношення, початкові й кінцеві умови.

Побудова алгоритму за точною постановкою задачі дає можливість його обґрунтування математичними методами. Більше того, існують класи задач, які дозволяють формальне перетворення специфікації в алгоритм. Вивченню деяких таких класів і відповідних методів відводиться важливе місце в нашому курсі.

Істотно, що, порівняно з програмою, алгоритм може перебувати на вищому рівні абстракції, бути вільним від тих або інших деталей реалізації, пов'язаних з особливостями мови програмування та конкретної обчислювальної системи. Засоби, прийняті для зображення алгоритмів, за традицією називають алгоритмічною мовою. До речі, так називалися також перші мови програмування високого рівня, наприклад, Алгол – це просто скорочення ALGOrithmic Language – алгоритмічна мова. Але, загалом, жодна мова програмування не може цілком замінити алгоритмічну мову, оскільки консервативна

за своєю суттю. Стабільність – один з необхідних компонентів якості мови програмування. Повинні існувати гарантії, що всі програми, складені вчора, в минулому році, десять років тому, не втратять значення ні сьогодні, ні завтра. Модифікація мови програмування призводить до небажаних наслідків: вимагає перероблення системи програмування, знецінює напрацьоване програмне забезпечення. У той же час алгоритмічна мова може створюватися спеціально для певної предметної області, певного класу задач або навіть окремої задачі. Вона може розвиватися навіть при створенні алгоритму, вбираючи в себе новітні результати.

Алгоритм – точне формальне розпорядження, яке однозначно трактує зміст і послідовність операцій, що переводять задану сукупність початкових даних в шуканий результат, або можна також сказати, що алгоритм – це кінцева послідовність загальнозрозумілих розпоряджень, формальне виконання яких дозволяє за скінченний час отримати рішення деякої задачі або будь-якої задачі з деякого класу задач.

✱ ***Алгоритм*** *– це скінченна послідовність команд, які треба виконати над вхідними даними для отримання результату.*

Приклад 1.1. Обчислити $(x+y)/(a-b)$

$A=(A_1, A_2, A_3)$

A_1: $x+y$

A_2: $a-b$

A_3: A_1/A_2

Слово алгоритм походить від algorithmi – латинської форми написання імені великого математика IX ст. Аль-Хорезмі, який сформулював правила виконання арифметичних дій.

Приклад 1.2. Розглянемо відому задачу про людину з човном (Л), вовком (В), козою (Кз) і капустою (Кп). Алгоритм її розв'язання можна подати так:

{ Л, В, Кз, Кп → } – початковий стан,
пливуть Л, Кз, Кп
{ В → Л, Кз, Кп } – 1-ий крок,
пливуть Л, Кз
{ Л, В, Кз → Кп } – 2-ий крок,
пливуть Л, В
{ Кз → Л, В, Кп } – 3-ій крок,
пливуть Л
{ Л, Кз → В, Кп } – 4-ий крок,
пливуть Л, Кз
{→Л, В, Кз, Кп } – кінцевий стан.

1.3. ВИКОНАВЦІ АЛГОРИТМІВ

Отже, алгоритм – це абстрактний математичний об'єкт, тому він вимагає абстрактного виконавця. Визначення цього виконавця по суті дає операційну семантику алгоритмічної мови. Модифікація алгоритмічної мови, очевидно, вимагає відповідної зміни іншого математичного об'єкту – виконавця.

Кожен виконавець може виконати певну кількість команд. Ці команди називаються **допустимими командами виконавця.**

Виконавцем ми будемо називати пристрій, здатний виконувати дії із заданого набору дій. Команду на виконання окремої дії називають *оператором* або *інструкцією*.

Приклади виконавців: пральна машина, телефон, мікрокалькулятор, магнітофон, комп'ютер тощо. Приклади інструкцій: перемотати плівку, встановити з'єднання із заданим номером, виконати прання бавовняної білизни, зіграти партію у реверсі і т.ін.

Зазначимо особливості виконавців. По-перше, людина далеко не єдиний виконавець алгоритмів. По-друге, будь-який виконавець складається з пристрою керування й «робочого інструменту». По-третє, кожний виконавець алгоритмів має обмежений набір допустимих дій (описати виконавця означає вказати його допустимі дії). Кожний виконавець може розуміти і виконувати якусь порівняно невелику кількість різних елементарних команд. Із цих команд складаються алгоритми. Як із 33 літер українського алфавіту можна написати і шкільний твір, і поему «Мойсей», так і з цієї невеликої кількості команд можна скласти невелику навчальну програму, а можна – і складну програму з тисячами команд.

По-четверте, для розв'язування одних і тих самих задач виконавці з «біднішим» набором допустимих дій вимагають складніших і докладніших алгоритмів. По-п'яте, різні класи задач вимагають різних наборів допустимих дій, різних виконавців.

Розглянемо приклади найпростіших виконавців.

Виконавець «Обчислювач». Він вміє: множити число на 2 (*2); збільшувати число на 1 (+1). Треба скласти алгоритм отримання числа 100 з одиниці. Скільки дій у найкоротшому з таких алгоритмів? Розглянемо простішу задачу отримання з одиниці числа 4. Для її розв'язання можна використати алгоритм вигляду 1 (+1) (+1) (+1) або 1 (*2) (*2). Очевидно, що другий алгоритм коротший. Повернемося до попередньої задачі. Для отримання найкоротшого алгоритму перетворюємо число 100 у 1, використовуючи ділення на 2 і віднімання 1. Маємо

$100 \to 50 \to 25 \to 24 \to 12 \to 6 \to 3 \to 2 \to 1$.

Отже, для нашого виконавця алгоритм
1 (*2) (+1) (*2) (*2) (*2) (+1) (*2) (*2)
буде найкоротшим і містить 8 дій.

Виконавець «Маляр». «Маляр» може переміщуватися плоским полем, що розбитий на клітинки однакового розміру. Між клітинками поля можуть бути розташовані стіни, деякі клітинки можуть бути зафарбовані. Під час переміщення «Маляр» може зафарбовувати клітинки. Отже, виконавець може виконувати команди:

вгору
вниз
ліворуч
праворуч
зафарбувати.

Треба скласти алгоритм, при виконанні якого «Маляр» переміститься з клітинки А у клітку В і зафарбує клітинки, позначені крапками в полі:

A		.	B
	.	.	

Алгоритм розв'язування поставленої задачі такий:

вниз → праворуч → зафарбувати → праворуч → зафарбувати → вгору → зафарбувати → праворуч .

І, нарешті, **складання програми для комп'ютера** – це кінцевий етап розв'язування задачі. Якщо перший в більшій мірі абстрактно-математичний, то в

останньому переважають моменти конкретно-виробничого характеру. Як влучно зазначив А.П.Єршов, «програміст мусить мати здатність першокласного математика до абстракції і логічного мислення в поєднанні з едісонівським талантом до створення чогось із нуля й одиниці» [21].

Якщо алгоритмічну мову вибирають, виходячи з власних бажань, то вибір мови програмування може диктувати певна реальність: замовник, комп'ютер, попередній досвід. У цьому посібнику розглядається одна з найбільш розповсюджених мов програмування – мова Сі. Створена 1972 року як мова для системного програмування, мова Сі дістала поширення у всьому світі. Сьогодні Сі має багатьох наступників серед виробничих мов програмування: C++, Java, C#. Приклади у курсі розглядаються з використанням системи Borland C++ v.3.1.

Іншою мовою, використаною у нашому посібнику, буде мова Паскаль, названа на честь великого французького вченого, який першим в світі 1642 року виготовив автоматичний пристрій для додавання чисел. Мова Паскаль створена у 1969 році Ніклаусом Віртом у Швейцарському технічному інституті у Цюріху спеціально для вивчення програмування. Сьогодні існує декілька варіантів мови Паскаль. Ми будемо дотримуватися версії системи Borland Pascal v.7.0.

1.4. СПОСОБИ ОПИСАННЯ АЛГОРИТМІВ

Існують такі способи описання алгоритмів:

- *словесний,*
- *формульний,*
- *графічний,*
- *алгоритмічною мовою.*

Перший спосіб *– це словесний опис алгоритму. Словесний опис потребує подальшої формалізації.*

Другий спосіб *– це подавання алгоритму у вигляді таблиць, формул, схем, малюнків тощо. Він є найбільш формалізованим та дозволяє описати алгоритм за допомогою системи умовних позначень.*

Третій спосіб *– запис алгоритмів за допомогою блок-схеми. Цей метод був запропонований в інформатиці для наочності подання алгоритму за допомогою набору* спеціальних блоків. Основні з цих блоків подані на рис. 1.1.

Четвертий спосіб *– це мови програмування. Справа в тому, що найчастіше в практиці виконавцем створеного людиною алгоритму являється машина і тому він має бути написаний мовою, зрозумілою для комп'ютера, тобто мовою програмування.*

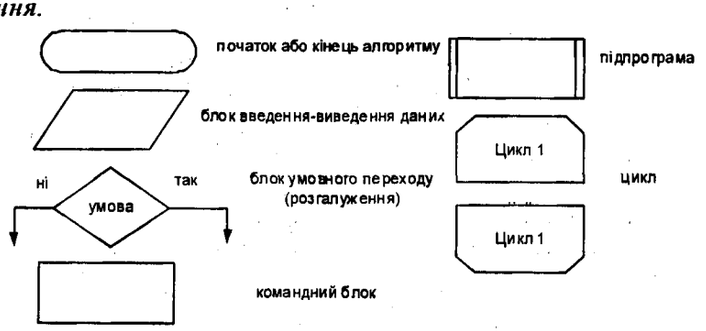


Рис. 1.1. Блоки для подання блок-схем.

1.5. ВЛАСТИВОСТІ АЛГОРИТМІВ

Розглянемо такі властивості алгоритмів: *визначеність, скінченність, результативність, правильність, формальність, масовість.*

Визначеність **алгоритму. Алгоритм визначений, якщо він складається з допустимих команд виконавця, які можна виконати для деяких вхідних даних.**

Приклад 1.3. Невизначеність виникне в алгоритмі (x+y)/(a-b), якщо в знамен-нику буде записано, наприклад, 92 – 92 (ділення на нуль неприпустиме).

Невизначеність виникне, якщо деяка команда буде записана не-правильно, бо така команда не належатиме до набору допустимих команд виконавця, (x+y)/(a-b).

Скінченність алгоритму. Алгоритм повинен бути скінченним – послідовність команд, які треба виконати, мусить бути скінченною. Кожна команда починає виконуватися після закінчення виконання попередньої. Цю властивість ще називають **дискретністю** алгоритму.

Приклад 1.4. Алгоритм (x+y)/(a-b) — скінченний. Він складається з трьох дій. Кожна дія, у свою чергу, реалізується скінченною кількістю елементар-них арифметичних операцій. Нескінченну кількість дій передбачає математичне правило перетворення деяких звичайних дробів, таких як 5/3, у нескінченні десяткові дроби.

Результативність алгоритму. Алгоритм результативний, якщо він дає результати, які можуть виявитися і невірнимим.

Наведені вище алгоритми є результативними. Прикладом нерезультативного алгоритму буде алгоритм для виконання обчислень, в якому пропущена команда виведення результатів на екран тощо.

Правильність алгоритму. Алгоритм правильний, якщо його вико-нання забезпечує досягнення мети.

Приклад 1.5. Наведені вище алгоритми є правильними. Помінявши місцями в алгоритмі з прикладу 1.2 будь-які дві команди, отримаємо неправильний алгоритм.

Формальність алгоритму. Алгоритм формальний, якщо його можуть виконати не один, а декілька виконавців з однаковими резуль-татами. Ця властивість означає, що коли алгоритм *A* застосовують до двох однакових наборів вхідних даних, то й результати мають бути однакові.

Приклад 1.6. Наведені алгоритми задовольняють цю умову, їх можуть виконати багато виконавців.

Масовість алгоритму. Алгоритм масовий, якщо він придатний для розв'язування не однієї задачі, а задач певного класу.

Приклад 1.6. Алгоритм з прикладу 1.1 не є масовим. Алгоритм Маляр є масовим, оскільки може застосовуватись не тільки для зафарбовування якихось елементів, але й для певних задач на графах. Прикладами масових алгоритмів є загальні правила, якими користуються для множення, додавання, ділення двох багатозначних чисел, бо вони застосовні для будь-яких пар чисел. Масовими є алгоритми розв'язування математичних задач, описаних у загальному вигляді за допомогою формул, їх можна виконати для різних вхідних даних.

1.6. ПОНЯТТЯ ОБЧИСЛЮВАЛЬНОЇ СКЛАДНОСТІ

Основними мірами обчислювальної складності алгоритмів є:

- *часова складність*, яка характеризує час, необхідний для виконання алгоритму на певній машині; цей час, як правило, визначається кількістю операцій, які треба виконати для реалізації алгоритму;
- *ємнісна складність*, яка характеризує об'єм пам'яті, необхідний для виконання алгоритму.

Часова та ємнісна складність тісно пов'язані між собою. Обидві є функціями від розміру вхідних даних. Надалі обмежимося тільки аналізом часової складності.

Складність алгоритму описуванняється функцією $f(n)$, де n – розмір вхідних даних. Важливе теоретичне і практичне значення має класифікація алгоритмів, яка бере до увагу швидкість зростання цієї функції.

1.7. КЛАСИ АЛГОРИТМІВ

Основною оцінкою функції складності алгоритму $f(n)$ є оцінка Θ.

Кажуть, що $f(n) = \Theta(g(n))$, якщо при $g > 0$ при $n > 0$ існують додатні c_1, c_2, n_0, такі, що

$$c_1 g(n) \le f(n) \le c_2 g(n)$$

при $n > n_0$. Тобто, можна знайти такі c_1 та c_2, що при достатньо великих n функція $f(n)$ знаходитиметься між $c_1 g(n)$ та $c_2 g(n)$.

У такому випадку функція $g(n)$ є асимптотично точною оцінкою функції $f(n)$, оскільки за визначенням функція $f(n)$ не відрізняється від функції $g(n)$ з точністю до постійного множника.

Виділяють такі основні класи алгоритмів:

- ✓ *логарифмічні*: $f(n) = \Theta(\log_2 n)$;
- ✓ *лінійні*: $f(n) = \Theta(n)$; якщо n=1, то отримуємо константні алгоритми;
- ✓ *поліноміальні*: $f(n) = \Theta(n^m)$; тут m – натуральне число, більше від одиниці; при m=1 алгоритм є лінійним;
- ✓ *експоненційні*: $f(n) = \Theta(a^n)$; a – натуральне число, більше від одиниці. Експоненційні алгоритми часто пов'язані з перебором різних варіантів розв'язку.

Для однієї й тієї ж задачі можуть існувати алгоритми різної складності. Часто буває і так, що повільніший алгоритм працює завжди, а швидший – лише за певних умов.

Будемо називати **часовою складністю задачі** часову складність найефективнішого алгоритму для її розв'язання.

Алгоритми без циклів і рекурсивних викликів мають константну складність. Якщо немає рекурсії та циклів, всі керуючі структури можуть бути зведені до структур константної складності. Отже, і весь алгоритм також характеризується константною складністю.

Визначення складності алгоритму в основному зводиться до аналізу циклів і рекурсивних викликів.

Приклад 1.7. Розглянемо алгоритм опрацювання елементів масиву. Нехай таким опрацюванням буде пошук заданого елемента.


```
For i:=1 to N do
Begin
If a[i]=k then
...
End;
```

Складність цього алгоритму (N), оскільки тіло циклу виконується N разів, і складність тіла циклу рівна (1).

Якщо один цикл вкладений у інший і обидва цикли залежать від величини однієї і тієї ж змінної, то вся конструкція характеризується квадратичною складністю.

```
For i:=1 to N do
For j:=1 toN do
Begin
...
End;
```

Складність цієї програми (N^2).

Але, якщо заздалегідь відомо, що послідовність упорядкована за зростанням або за спаданням, можна застосувати інший алгоритм – алгоритм половинного ділення. Послідовність ділиться на дві рівні частини. Оскільки послідовність упорядкована, можна визначити, в якій частині міститься потрібний елемент. Після цього процедура повторюється: потрібна частина знову ділиться навпіл і т.д. Цей алгоритм є логарифмічним.

Дамо тепер визначення **складності класів задач P і NP**. **Клас P** складається зs задач, для яких існують поліноміальні алгоритми розв'язання. **Клас NP** складають задачі, для яких існують поліноміальні алгоритми перевірки правильності рішення (точніше, якщо є розв'язок задачі, то існує деяка підказка, яка дозволяє за поліноміальний час отримати цю відповідь). Неформально кажучи, клас P складається зі задач, які можна швидко розв'язати, а класс NP – зі задач, розв'язок яких можна швидко перевірити.

Наведемо приклад задачі класу P. Треба визначити, чи є у масиві дійсних чисел $A[1..n]$ елемент зі значенням не меншим, ніж k. Очевидний у цьому випадку алгоритм перебирає всі елементи масиву за час (n).

Прикладом задачі класу NP є задача комівояжера (є множина міст та відділей між ними, мандрівний торговець має відвідати усі міста, не заходячи у жодне двічі, з мінімальними витратами на дорору). Дійсно, якщо задано деякий маршрут завдовжки не більше ніж k, то за час (n) можна перевірити, що він дійсно має саме таку довжину, і тим самим переконатися у його існуванні. Для цього треба перебрати всі n переходів між містами, що містяться у маршруті, і додати їх довжини.

Очевидним є також включення P NP (для перевірки розв'язання задачі класу P досить розв'язати її поліноміальним алгоритмом).

Задача називається **NP-повною**, якщо вона належить класу NP і до неї за поліноміальний час можна звести будь-яку іншу задачу цього класу. Якщо якась NP-повна задача має поліноміальний алгоритм розв'язання, то всі NP-повні задачі можуть бути поліноміально розв'язані і, як наслідок, P=NP.

NP-повні задачі є найважчими у класі NP.

1.7.1. Експоненційні алгоритми та перебір

Експоненційні алгоритми часто пов'язані з перебором різних варіантів розв'язання. Наведемо типовий приклад.

Приклад 1.8. Розглянемо задачу про виконуваність булевого виразу, яка формулюється так: для будь-якого булевого виразу від n змінних знайти хоча б один набір значень змінних $x_1 \dots x_n$, при якому цей вираз приймає значення 1.

Типова схема розв'язування цієї задачі може мати такий вигляд: спочатку надаємо одне з двох можливих значень (0 або 1) першій змінній x_1, потім другій і т.ін. Коли будуть розставлені значення всіх змінних, ми можемо визначити значення булевого виразу. Якщо він дорівнює 1, задача розв'язана. Якщо ні – треба повернутися назад і змінити значення деяких змінних.

Можна інтерпретувати цю задачу як задачу пошуку на певному дереві перебору. Кожна вершина цього дерева відповідає певному набору встановлених значень $x_1 \dots, x_k$. Ми можемо встановити змінну x_{k+1} в 0 або 1, тобто маємо вибір з двох дій. Тоді кожній із цих дій відповідає одна з двох дуг, які йдуть від цієї вершини. Вершина $x_1 \dots, x_k$ має двох синів $x_1 \dots, x_k\ 0$ і $x_1 \dots, x_k\ 1$.

При n=3 дерево можливостей матиме вигляд, показаний на рис. 1.2. (вершини позначені наборами значень змінних, а дуги – рішеннями, які приймаються на черговому кроці). Вершини, що відповідають розв'язкам задачі для виразу $(x_1 x_2) \in x_3$, зафарбовані.

З рис.1.2 видно, що розв'язування задачі зводиться до перебору листків дерева з метою виявлення, які з них відповідають наборам, що перетворюють заданий булевий вираз на 1. Для виразу $(x_1 x_2) \in x_3$ такими наборами будуть 000, 010, 111.

Якщо n=3, дерево має 2^3= 8 листів. У загальному ж випадку кількість можливих варіантів дорівнює 2^n. Цей вираз експоненційно залежить від n, і перебірний алгоритм має експоненційний характер.

Описана вище схема алгоритму розв'язування цієї задачі має фундаментальний характер. Вона має назву бектрекінгу (інша назва – перебір з поверненням) і використовується для розв'язування найрізноманітніших перебірних задач (задача про вісім ферзів, задача про розфарбовування карти, автоматизація дедуктивних побудов тощо).

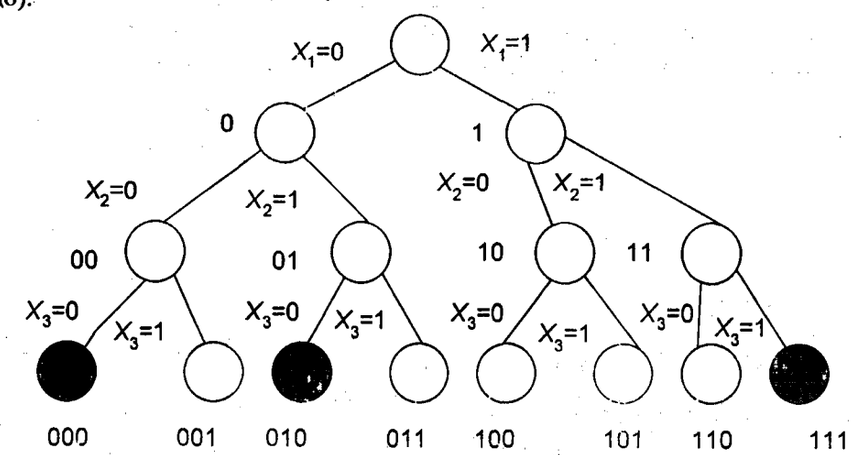


Рис. 1.2. Пошук на дереві.

Звертаємо увагу на те, що не кожен перебірний алгоритм є експоненційним. (Наприклад, алгоритм пошуку в масиві. Незважаючи на його перебірний характер, він є лінійним, а не експоненційним).

Зі зростанням розмірності будь-який поліноміальний алгоритм стає ефективнішим, ніж будь-який експоненційний. Для лінійного алгоритму зростання швидкодії комп'ютера в 10 разів дозволяє за той самий час розв'язати задачу, розмір якої в 10 разів більший. Для експоненційного алгоритму з основою 2 цей самий розмір можна збільшити лише на 3 одиниці.

Як правило, якщо для розв'язування якоїсь задачі є деякий поліноміальний алгоритм, то часова оцінка цього алгоритму значно покращується.

1.7.2. Алгоритм із поверненнями назад

Метод перебору із поверненнями дозволяє розв'язувати практично незліченну множину задач, для багатьох з яких не відомі інші алгоритми. Незважаючи на таке велике різноманіття перебірних задач, в основі їх розв'язування є щось спільне, що дозволяє застосувати цей метод. Таким чином, перебір можна вважати практично універсальним методом розв'язування перебірних завдань. Наведемо загальну схему цього методу.

Розв'язування задачі методом перебору з поверненням будується конструктивно послідовним розширенням часткового розв'язування. Якщо на конкретному кроці таке розширення провести не вдається, то відбувається повернення до коротшого часткового розв'язування, і спроби його розширити продовжуються.

```
{ пошук одного вирішення }
procedure backtracking(k: integer); { k – номер ходу }
  begin
  { запис варіанту }
  if { рішення знайдене } then
        { виведення рішення }
  else
        { перебір всіх варіантів }
  if { варіант задовольняє умови задачі } then
        backtracking(k+1); { рекурсивний виклик }
              { стирання варіанту }
  end
begin
backtracking(1);
end.
```

1.7.3. Машини Тьюринґа

Машина Тьюринґа, що була описана А.Тьюрингом 1936 року, є теоретичною моделлю обчислювальної машини. **Машину Тьюринґа** (МТ) (рис. 1.3) слід розглядати як одну з можливих формалізацій поняття алгоритму. Її робота може бути описана таким чином.

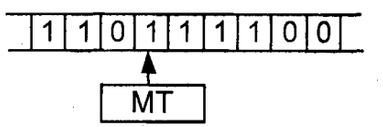


Рис. 1.3. Схема машини Тьюринґа.

Розглянемо стрічку, розділену на окремі комірки; ця стрічка є потенційно нескінченною в обидва боки. У кожній комірці може бути записаний певний символ з деякого заданого алфавіту A. Машина Тьюринга в будь-який момент часу може перебувати в певному стані (множина станів S є скінченною) і вказувати на певну комірку.

Машина Тьюринга в залежності від поточного символа, на який вона вказує, може записати на його місце будь-який інший символ (він може співпадати зі старим), зсунутися на один символ вліво або вправо, змінити свій вміст чи зупинитися (часто вважається, що машина зупиняється автоматично, якщо немає жодної інструкції, яку вона могла б виконати). Робота машини Тьюринга визначається її програмою.

Програма машини Тьюринга є послідовністю інструкцій, кожна з яких має вигляд

$$a_i s_j \to a_k s_l I,$$

де $a_i, a_k \in A$; $s_l, s_m S$, $I \in \{R, L, H\}$.

Цей запис читається так: якщо машина перебуває в стані s_j і зчитує символ a_i, вона повинна записати в поточну позицію символ a_k, перейти до стану s_m і зсунутися вправо (відповідає літері R), вліво (відповідає літері L) або зупинитися (відповідає літері H).

Вважається, що на початку роботи машина перебуває на лівому кінці стрічки в початковому стані s_0. Вона виконує операції, що визначаються її програмою. Якщо вона в деякий момент зупиняється, результатом роботи алгоритму вважається послідовність символів, яка записана на стрічці в момент зупинки.

Приклад 1.9. Наведемо програму для машини Тьюринга, яка обчислює функцію $x+y$, де x,y – натуральні числа. Необхідно домовитися про відображення цих чисел. Стандартним для машини Тьюринга є подання натурального числа n послідовністю з $n+1$ одиниць.

Ідея реалізації такої програми могла б полягати у тому, щоб замінити крайній зліва та крайній справа символи «1» на «0», а роздільник «0» – на «1». Якби були доступні відповідні команди, це можна було б зробити просто і швидко, але ми обмежені жорсткими рамками машини Тьюринга. За таких умов схема алгоритму полягає в такому: рухатися вправо до виявлення першої одиниці (початок першого числа); як тільки вона буде виявлена, замінити її на нуль і перейти в інший стан s_1. Потім рухатися вправо, поки 0, який розділяє два числа, не буде замінений на 1; при цьому знову змінити стан. Далі рухатися вправо до появи першого нуля (кінця другого доданку). Після цього зсунутися вліво, замінити останню 1 на 0 та зупинитися.

Програма може мати вигляд:

$0 s_0 \to 0\ s_0 R$
$1 s_0 \to 0\ s_1 R$
$1 s_1 \to 1\ s_1 R$
$0 s_1 \to 1\ s_2 R$
$1 s_2 \to 1\ s_2 R$
$0 s_2 \to 0\ s_3 L$
$1 s_3 \to 0\ s_4 H.$

Навсдений приклад показує, що машина Тьюринга є дуже незручною для програмування. Ці незручності пов'язані з тим, що:

- немає довільного доступу до пам'яті; якщо, наприклад, машина вказує на першу комірку, а треба перейти до десятої, машина повинна послідовно переглянути другу, третю і т.ін. комірки;

- неструктурованість записів на стрічці; заздалегідь невідомо, де закінчується одне число і починається інше;
- дуже обмежений набір команд; відсутні, наприклад, основні арифметичні операції.

Універсальну машину Тьюринга можна неформально визначити як машину, яка може сприймати програму для обчислення будь-якої функції, яку, в принципі, можна обчислити за допомогою спеціалізованої машини M1 і надалі працювати, як машина M1. Можна довести, що таку машину можна побудувати.

Багатство можливостей конструкції Тьюринга полягають у тому, що якщо якісь алгоритми A та B реалізуються машинами Тьюринга, то можна будувати програми машин Тьюринга, які реалізують композиції алгоритмів A та B, наприклад, виконати A, потім виконати B або виконати A знову. Якщо в результаті утворилося слово «так», то виконати B. У протилежному випадку не виконувати B або виконувати по черзі A, B, поки B не дасть відповідь «ні».

У інтуїтивному сенсі такі композиції є алгоритмами. Тому їхня реалізація за допомогою машини Тьюринга служить одним із засобів обґрунтування універсальності конструкції Тьюринга.

Реалізованість таких композицій доводиться у загальному вигляді, незалежно від особливостей конкретних алгоритмів A та B. Доведення полягає в тому, що вказується засіб побудови з програм A та B програми требаї композиції. Нехай, наприклад, треба побудувати машину A \cdot B, еквівалентну послідовному виконанню алгоритмів A та B. Поки виконується алгоритм A, у програмі A\cdot B працює частина A без урахування частини B. Коли алгоритм A дійде до кінця, то замість зупинки відбудеться перехід у перший стан частини B, і потім частина B буде працювати звичайним чином, наче частини A й не було.

Аналогічно конструюють й інші композиції машин Тьюринга; щораз будуються загальні правила, які визначають, що на що змінювати у вихідних програмах.

Описуванняючи різноманітні алгоритми для машин Тьюринга і стверджуючи реалізованість усіляких композицій алгоритмів, Тьюринг переконливо показав розмаїтість можливостей запропонованої ним конструкції, що дозволило йому виступити з такою тезою:

будь-який алгоритм може бути реалізований відповідною машиною Тьюринга.

Це основна гіпотеза теорії алгоритмів у формі Тьюринга. Одночасно ця теза є формальним визначенням алгоритму. Завдяки їй можна доводити існування або неіснування алгоритмів, створюючи відповідні машини Тьюринга або доводячи неможливість їхньої побудови. Завдяки цьому з'являється загальний підхід до пошуку алгоритмічних розв'язків.

Якщо пошук розв'язку наштовхується на перешкоду, то можна використовувати цю перешкоду для доведення неможливості розв'язування, спираючись на основну гіпотезу теорії алгоритмів. Якщо ж при доказі неможливості виникає своя перешкода, то вона може допомогти просунутися в пошуку розв'язку, хоча б частково усунувши стару перешкоду. Так, по черзі намагаючись довести то існування, то відсутність розв'язку, можна поступово наблизитися до розуміння суті поставленної задачі.

Довести тезу Тьюринга не можна, тому що в його формулюванні не визначене поняття будь-який алгоритм, тобто ліва частина тотожності. Його можна тільки обґрунтувати, подаючи різноманітні відомі алгоритми у вигляді машин Тьюринга. Додаткове обґрунтування цієї тези полягає в тому, що пізніше було запропоновано ще декілька загальних визначень поняття алгоритму і щораз вдавалося довести, що, хоча нові алгоритмічні схеми і виглядають інакше, вони насправді еквівалентні машинам Тьюринга:

усе, що реалізовано в одній з цих конструкцій, можна зробити і в інших.

Ці твердження доводяться строго, тому що в них мова йде вже про тотожність формальних схем.

1.7.4. Рекурсія та її використання

Означення називається **рекурсивним**, якщо воно задає елементи множини за допомогою інших елементів цієї ж множини. Об'єкти, задані рекурсивним означенням, також називаються **рекурсивними**. Нарешті, **рекурсія** – це використання в алгоритмі рекурсивних означень.

Рекурсивною функцією називається функція, яка може бути отримана з базових функцій за допомогою скінченної кількості застосувань підстановок, та примітивних рекурсій.

До базових примітивних рекурсивних функцій відносяться три функції:

1) нуль-функція $Z(x)=0$ при будь-якому x;

2) додавання одиниці: $N(x) = x+1$;

3) проектуючі функції: $U_i(x_1,\ldots, x_n)= x_i$ для всіх $x_1, \ldots, x_n$ (визначає натуральне число з цієї множини).

Функція $f(x_1,\ldots, x_n)$ отримана з функцій g, $h_1,\ldots, h_m$, за допомогою підстановки, якщо $f(x_1,\ldots, x_n) = g\,(h_1(x_1,\ldots, x_n), \ldots, h_1(x_1,\ldots, x_n))$.

Функція $f(x_1,\ldots, x_n, y)$ отримана за допомогою рекурсії, якщо:

$f(x_1,\ldots, x_n, y+1) = h\,(x_1,\ldots, x_n, y, f(x_1,\ldots, x_n, y))$;

$f(x_1,\ldots, x_n, 0) = g\,(x_1,\ldots, x_n)$ при $n \neq 0$;

$f(y+1) = h\,(y, f(y))$; $f(0) = k$, k – ціле невід'ємне число при $n= 0$.

Функція $f(x_1,\ldots, x_n)$ отримана за допомогою μ - оператора, якщо її значення дорівнює мінімальному значенню y, при якому $g((x_1,\ldots, x_n, y)=0$.

Частково рекурсивною називається функція, яка конструюється за допомогою скінченного числа підстановок, рекурсій та μ -операторів, але визначена не при всіх значеннях аргументів.

Якщо деяка функція може бути сконструйована з базових функцій за допомогою скінченного числа тільки підстановок та рекурсій, вона називається **примітивно рекурсивною**.

Приклад 1.10. Визначення функції «факторіал» задаються виразом: 0!=1, n!=nx (n-1)!. Вони утворюють множину {1,2,6,...}: 0!=1, 1!=1, 2!=2, 3!=6, ... Усі її елементи, крім першого, визначаються рекурсивно.

Отже, функція «факторіал» задається рекурентним співвідношенням порядку 1 і початковим відрізком 0!=1. Загалом, будь-яке рекурентне співвідношення порядку k разом із завданням перших k елементів послідовності є прикладом рекурсивного означення.

Приклад 1.11. Арифметичні вирази зі сталими та знаком операції '+' у повному дужковому записі (ПДЗ) задаються таким означенням:

1) стала є виразом у ПДЗ;

2) якщо E і F є виразами у ПДЗ, то (E)+(F) також є виразом у ПДЗ.

Такими виразами є, наприклад, 1, 2, (1)+(2), ((1)+(2))+(1). Всі вони, крім сталих, означаються рекурсивно.

Для коректного виходу з рекурсії повинні виконуватися умови:

- ✓ *множина означуваних об'єктів є частково упорядкованою;*
- ✓ *кожна спадна за цим впорядкуванням послідовність елементів закінчується деяким мінімальним елементом;*
- ✓ *мінімальні елементи означаються нерекурсивно;*
- ✓ *немінімальні елементи означаються за допомогою менших від них елементів.*

Глибина рекурсії – кількість викликів підпрограми, що реалізують рекурсію.

Вживання рекурсивних підпрограм вимагає обережності та вміння оцінити можливу глибину рекурсії та загальну кількість викликів. Не завжди треба писати рекурсивні підпрограми безпосередньо за рекурсивним означенням. Справа в тому, що виконання кожного виклику підпрограми потребує додаткових дій комп'ютера. Тому «циклічний» варіант описання обчислень виконується, як правило, швидше від рекурсивного. Також не треба вживати рекурсію для обчислення елементів рекурентних послідовностей. За великої глибини рекурсії це взагалі може призвести до вичерпання автоматичної пам'яті та аварійного завершення програми.

Наведемо приклад нерекурсивного та рекурсивного визначення значення функції.

Приклад 1.12. Маємо функцію, що розкладається у ряд:

$$s = \prod_{x=1}^{n} \frac{x}{e^x - x^2}.$$

```
#include <stdio.h>
#include <math.h>
double func1(int n); // прототип нерекурсивної функції ряду
double func2(int n); // прототип рекурсивної функції ряду
void main()
{
    int  n; // n – кількість елементів ряду
    double s=1; // значення функції
    scanf("%n", &n);  // ввели кількість елементів ряду
    printf("Nonrecursiv result is %6.2f", func1(n));
    printf("Recursiv result is %6.2f", func2(n));
}
double func1(int n)
{
    double s=1;
    int x; // x – поточний член ряду
    for(x=1;x<=n;x++)
        s*=x/(exp(x) – x*x)
```

```
    return (s);
}
    double func2(int n)
    {
        double s=1;
        if (n<1) // умова виходу з рекурсії
                return (s);
        else
                s*=n/(exp(n) – n*n)*func2(n-1);// виклик рекурсивної функції з n-1
    }
```

Найкраще рекурсію використовувати тоді, коли розв'язання задачі з допомогою рекурсії загалом зводиться до розв'язання подібної задачі, але меншої розмірності, а, отже, легшої для розв'язання.

Найяскравіший приклад задачі такого плану – задача про ханойські вежі.

Легенда свідчить, що десь в Ханої знаходиться храм, в якому розміщена така конструкція: на підставці знаходяться 3 діамантових стержня, на яких при створенні світу Брахма нанизав 64 золотих диска із отвором посередині, причому внизу опинився найбільший диск, на ньому трохи менший і так далі, поки на верхівці піраміди не опинився найменший диск. Жерці храму зобов'язані перекладати диски за такими правилами:

1) за один хід можна перенести лише один диск,

2) не можна класти більший диск на менший.

Керуючись цими правилами, жерці повинні перенести початкову піраміду з 1-го стержня на 3-й. Як тільки вони впораються з цим завданням, настане кінець світу.

Для розв'язання задачі спочатку спробуємо побудувати абстрактну модель процесу перенесення частини піраміди.

Отже, вирішуємо узагальнену задачу: як перенести піраміду з *n* кілець із стержня *i* на стержень *j*, користуючись стержнем *k* як допоміжним? Ця задача розв'язується таким чином.

1.Перенести (n-1) кільце *i* на *k*.

2.Перенести 1 кільце з *i* на *j*.

3.Перенести (n-1) кільце на *j*.

Отже, задача перенесення *n* кілець розв'язується через перенесення (n-1) кільця. Залишилося лише додати тривіальну граничну умову для виродженого випадку перенесення порожньої піраміди з 0 кілець, щоб наш алгоритм завершився.

Реалізація алгоритму на мові С:

```
void Hanoj (int n, short i, short j, short k)
// перенесення піраміди з n дисків з стержня i
// на стержень j,  використовуючи стержень k як допоміжний
{
// перевірка граничної умови – порожню піраміду не переміщувати
if (!n) return;
Hanoj (n-1, i, k, j);
// переносимо одне кільцо – фактично це Hanoj (1, i, j, k);
printf(«%2d - %2d\n», i, j);
Hanoj (n-1, k, j, i);
}
```


1.7.5. Теза Чорча. Алгоритмічно нерозв'язні проблеми

Теза Чорча часто формулюється в еквівалентній формі, а саме: будь-який алгоритм в інтуїтивному розумінні цього слова може бути реалізований за допомогою деякої машини Тьюринга. Іншими словами, за допомогою машини Тьюринга можна розв'язати будь-яку задачу, для якої існує алгоритм розв'язування в інтуїтивному розумінні.

Довести тезу Чорча неможливо. Її можна було б спростувати, якби були запропоновані формалізації поняття алгоритму, здатні обчислювати нерекурсивні функції. Але таких формалізмів досі запропоновано не було.

Якщо ми визнаємо тезу Чорча, ми можемо прийняти машину Тьюринга як основу для загального визначення алгоритму: алгоритм є послідовністю інструкцій, яка може бути виконана за допомогою машини Тьюринга або еквівалентної їй обчислювальної моделі.

Тепер ми можемо дати більш чітке визначення універсального комп'ютера: універсальним називається комп'ютер, за допомогою якого можна промоделювати роботу машини Тьюринга.

Якщо теза Чорча є справедливою, ми маємо визнати наступне. Якщо вдається довести, що не існує машини Тьюринга, яка могла б вирішити певну проблему, то ця проблема є алгоритмічно нерозв'язною, тобто для неї не існує загального алгоритму розв'язування. Такі проблеми насправді існують.

Наприклад, це знаменита проблема зупинки, яка формулюється так: потрібно для будь-яких початкових даних визначити, зупиниться машина Тьюринга чи ні. Оскільки будь-яка програма, яка працює на будь-якому комп'ютері, може бути реалізована і на машині Тьюринга, це твердження можна переформулювати так: не існує загального алгоритму, який для будь-якої програми заздалегідь визначав би, зупиниться вона чи ні.

Але можна навести формалізми, які полегшують програмування і дозволяють здійснювати обчислення швидше, ніж машини Тьюринга.

Резюме

1. Інформація – це відомості, пояснення, знання про всілякі об'єкти, явища, процеси реального світу

2. Інформатика – це наука про методи подання, накопичення, передавання та опрацювання інформації за допомогою комп'ютера.

3. Алгоритм – це скінченна послідовність команд, які треба виконати над вхідними даними для отримання результату.

4. Виконавцем називають пристрій, здатний виконувати дії із заданого набору дій. Він складається з пристрою керування і «робочого інструмента».

5. Є такі способи описування/навання алгоритмів: словесний, формульний, графічний, алгоритмічною мовою.

6. Алгоритм має такі властивості: визначеність, скінченність, результативність, правильність, формальність, масовість.

7. Основними мірами обчислювальної складності є часова складність та ємнісна складність.

8. Є такі класи алгоритмів: логарифмічні, лінійні, поліміальні, експоненційні. Експоненційні алгоритми часто пов'язані з перебором різних варіантів розв'язування.

9. Клас P складається із задач, для яких існують поліноміальні алгоритми рішення. Клас NP складають задачі, для яких існують поліноміальні алгоритми перевірки правильності рішення (точніше, якщо є розв'язок задачі, то існує деяка підказка, яка дозволяє за поліноміальний час отримати цю відповідь).

10. Машина Тьюринга є однією з можливих формалізацій поняття алгоритму.

11. Означення називається рекурсивним, якщо воно задає елементи множини за допомогою інших елементів цієї ж множини. Об'єкти, задані рекурсивним означенням, також називаються рекурсивними.

12. Рекурсивною називається функція, яка може бути отримана з базових функцій за допомогою скінченної кількості застосувань підстановок, примітивних рекурсій. Базові примітивні рекурсивні функції: нуль-функція, додавання одиниці, проектуюча функція.

13. Глибина рекурсії – кількість викликів підпрограми, що реалізують рекурсію.

14. Теза Чорча: будь-який алгоритм в інтуїтивному розумінні цього слова може бути реалізований за допомогою деякої машини Тьюринга. Іншими словами, за допомогою машини Тьюринга можна вирішити будь-яку задачу, для якої існує алгоритм розв'язування в інтуїтивному розумінні.

Контрольні запитання

1. *Яке походження терміну «алгоритм»?*
2. *Що ми розуміємо під поняттям «алгоритм»?*
3. *Що таке допустимі команди виконавця?*
4. *Які є способи описуваннявання алгоритмів?*
5. *Які властивості повинен мати алгоритм?*
6. *Що означає скінченність (дискретність) алгоритму?*
7. *Що таке формальність алгоритму?*
8. *Що означає масовість алгоритму?*
9. *Яка різниця між поліноміальними та експоненційними класами алгоритмів?*
10. *Дайте визначення часової складності.*
11. *Дайте визначення класу задач P та NP.*
12. *Поясніть принцип роботи машини Тьюринґа.*
13. *Дайте визначення рекурсивної функції.*
14. *Наведіть приклади частково рекурсивної функції.*
15. *Наведіть приклад примітивної рекурсії.*
16. *Дайте визначення терміну «глибина рекурсії».*
17. *Наведіть приклади рекурсивних функцій.*
18. *Сформулюйте тезу Чорча.*
19. *Вкажіть відмінність між інформацією та даними.*

Тести для закріплення матеріалу

1. Оберіть визначення інформатики:

а) це наука про методи подання, накопичення, передавання та опрацювання інформації за допомогою електронно-обчислювальних машин;

б) це поняття, що передбачає наявність матеріального носія інформації, джерела і передавача інформації, приймача і каналу зв'язку між джерелом і приймачем інформації;

в) це наука про відображення реальних об'єктів, процесів, подій тощо на комп'ютерних носіях;

г) це одних із видів програмування та проектування.

2. Визначте поняття загальної інформатики:

а) блок-схема;

б) задача;

в) функція мети;

г) алгоритм.

3. Визначте поняття алгоритму:

а) точне формальне розпорядження, яке однозначно визначає зміст і послідовність операцій, що переводять задану сукупність початкових даних у потрібний результат;

б) скінчена послідовність машинних команд для розв'язедання конкретної задачі;

в) записана на певній мові програмування послідовність операторів, що подає конкретну задачу;

г) кінцева послідовність загальнозрозумілих розпоряджень, формальне виконання яких дозволяє за скінчений час отримати розв'язок деякої задачі або будь-якої задачі з деякого класу задач;

д) скінченна послідовність команд, які треба виконати над вхідними даними для отримання результату.

4. Дати поняття виконавця:

а) пристрій, здатний виконувати дії із заданого набору дій;

б) команда на виконання окремої дії;

в) складається з пристрою керування і «робочого інструмента»;

г) може розуміти і виконувати якусь порівняно невелику кількість різних елементарних команд;

д) алгоритмічна мова високого рівня.

5. Способи описування алгоритмів:

а) словесний;

б) схематичний;

в) табличний;

г) формульний;

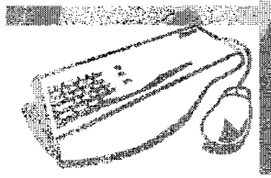
д) графічний;
е) алгоритмічною мовою.

6. Властивість алгоритму «скінченність» інакше ще називають:
а) результативність;
б) формальність;
в) масовість;
г) дискретність;
д) правильність.

7. Перерахувати класи алгоритмів:
а) логарифмічні;
б) табличні;
в) лінійні;
г) прогресійні;
д) поліноміальні;
е) експоненційні.

8. Дати визначення рекурсії:
а) формує наступне у прогресії значення;
б) підпрограма викликає саму себе;
в) задає елементи множини за допомогою інших елементів цієї ж множини;
г) використовується для розрахунків над матрицями.

РОЗДІЛ 2



ПОНЯТТЯ СТРУКТУРИ ДАНИХ. РІВНІ ПОДАННЯ СТРУКТУР ДАНИХ

- *Поняття структури даних.*
- *Рівні описуваннявання даних.*
- *Класифікація СД у програмах користувача й у пам'яті комп'ютера.*
- *Основні види складених типів даних.*
- *Структури даних у пам'яті комп'ютера.*

Розділ присвячений розгляду різних типів структур даних. Описано рівні структур даних та здійснено їх класифікацію. Подано особливості відображення структур даних у пам'яті комп'ютера.

2.1. ПОНЯТТЯ СТРУКТУРИ ДАНИХ

✱ ***Структура даних (СД)*** *– загальна властивість інформаційного об'єкта, з яким взаємодіє та або інша програма. Ця загальна властивість характеризується:*

- ✓ *множиною допустимих значень цієї структури;*
- ✓ *набором допустимих операцій;*
- ✓ *характером організованості.*

Найпростіші структури даних називаються також *типами даних*.

У програмуванні та комп'ютерних науках структури даних — це способи організації даних у комп'ютерах. Часто разом зі структурою даних пов'язується і специфічний перелік операцій, які можуть бути виконаними над даними, організованими в таку структуру.

Правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів їх опрацювання. Добре побудовані структури даних дозволяють оптимізувати використання машинного часу та пам'яті комп'ютера для виконання найбільш критичних операцій.

Відома формула «**Програма = Алгоритми + Структури даних**» дуже точно виражає необхідність відповідального ставлення до такого підбору. Тому іноді навіть не обраний алгоритм для опрацювання масиву даних визначає вибір тієї чи іншої структури даних для їх збереження, а навпаки.

2.2. РІВНІ ОПИСУВАННЯВАННЯ ДАНИХ

Розрізняють наступні рівні описуваннявання даних:

- *абстрактний (математичний) рівень;*
- *логічний рівень;*
- *фізичний рівень.*

Логічний рівень (ЛСД) *– подання структури даного на тій чи іншій мові програмування.* ***Фізичний рівень (ФСД)*** *– відображення у пам'ять комп'ютера*

інформаційного об'єкту відповідно до логічного описування. Оскільки пам'ять комп'ютера обмежена, то виникають питання розподілу пам'яті й керування нею.

Логічний і фізичний рівні відрізняються один від одного, тому в обчислювальних системах здійснюється відображення фізичного рівня на логічний і навпаки (рис. 2.1).

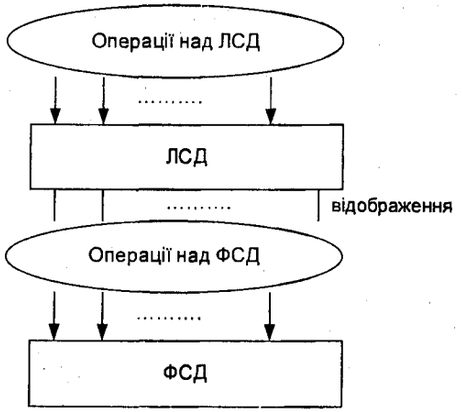


Рис. 2.1. Звязок між логічним та фізичним рівнями подання СД.

Будь-яка структура на абстрактному рівні може бути подана у вигляді двійки $<D,R>$, де D – скінчена множина елементів, які можуть бути типами даних або структурами даних, а R – множина відношень, властивості якої визначають різні типи структур даних на абстрактному рівні.

2.3. КЛАСИФІКАЦІЯ СТРУКТУР ДАНИХ У ПРОГРАМАХ КОРИСТУВАЧА Й У ПАМ'ЯТІ КОМП'ЮТЕРА

Структури даних поділяються на вбудовані (реалізовані в мовах програмування) та похідні (утворюються користувачами). Класифікація СД у програмах користувача та пам'яті комп'ютера подана на рис. 2.2.

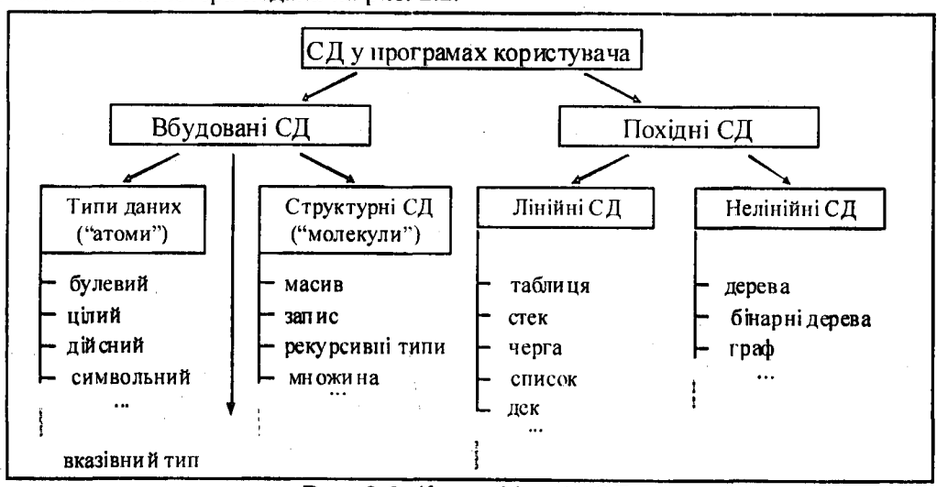


Рис. 2.2. Класифікація СД.

Важливою ознакою для класифікації є зміна структур даних під час виконання програми. Наприклад, якщо змінюється кількість елементів і/або відношення між ними, то такі структури даних називаються *динамічними*, інакше – *статичними*.

2.4. ОСНОВНІ ВИДИ СКЛАДЕНИХ ТИПІВ ДАНИХ

Складеним типом даних назвемо тип даних, що складається із скінченної та наперед заданої множини елементів певного типу, які не обов'язково є атомарними.

Перерахуємо складені типи даних та дамо їм коротку характеристику.

Множина – скінчена сукупність елементів, в якої $R = \varnothing$.

Послідовність – абстрактна структура, у якої множина R складається з одного відношення лінійного порядку (тобто для кожного елемента, крім першого і останнього, є попередній і наступний елементи).

Матриця – структура, в якої множина R складається із двох відношень лінійного порядку.

Дерево – множина R складається з одного відношення ієрархічного порядку.

Граф – множина R складається з одного відношення бінарного порядку.

Гіперграф – множина R складається із двох і більше відношень різного порядку.

Приклади СД подано на рис. 2.3.

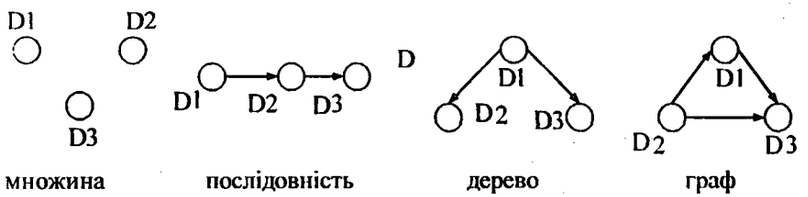


Рис. 2.3. Приклади подання структур даних.

Прикладом гіперграфа є мережа Петрі – дводольний граф, який складається з двох типів вершин: станів, які мають певну кількість фішок, та переходів, що перерозподіляють фішки у станах залежно від кількості вхідних та вихідних дуг.

2.5. СТРУКТУРИ ДАНИХ У ПАМ'ЯТІ КОМП'ЮТЕРА

2.5.1. Структури даних в оперативній пам'яті

В оперативній пам'яті структури даних можна подати як на рис. 2.4.

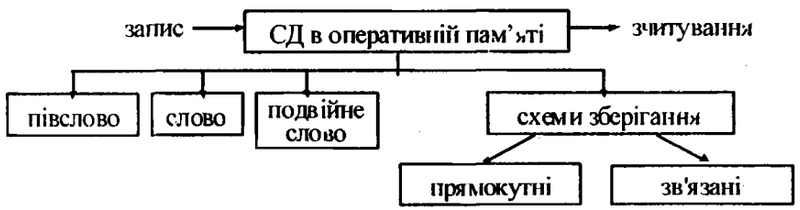


Рис. 2.4. Подання СД в оперативній памяті

Слід зазначити, що оперативна пам'ять є масивом. Слово – мінімальна кількість біт, яка може опрацьовуватися одночасно.

2.5.2. СД у зовнішній пам'яті

На рис 2.5. подано структури даних у зовнішній пам'яті.

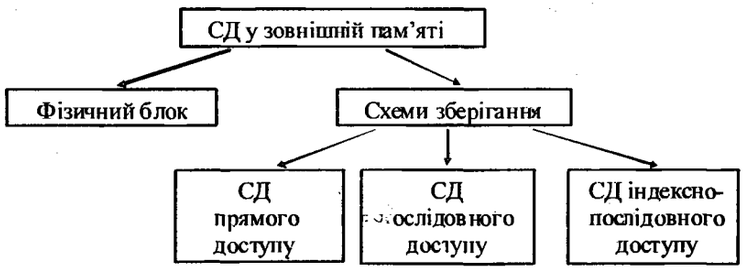


Рис. 2.5. Подання СД у зовнішній памяті.

СД послідовного доступу передбачає опрацювання даних у порядку їх розміщення на носії. Така СД є простою для реалізації, але унеможливлює безпосередній доступ до заданого елемента. Приклад структури даних послідовного доступу: стек, черга, дек.

На мові Сі для послідовного зчитування з файла можна скористатися таким фрагментом програми:

```
f=fopen("my.txt", "rt");//відкрили послідовно для читання
for(i=0;i<k;i++)
  {читаємо k елементів без перевірки правильності зчитування}
fscanf(f, "%d",&x);
```

СД прямого доступу дозволяє опрацьовувати елементи у необхідному для користувача порядку шляхом зміщення певної кількості бітів. Переваги: швидкість, простота. Недоліки: відірваність процедур доступу від самих даних.

Прикладом прямого доступу є запис у файл fd змінної типу long, починаючи з 20-ої позиції:

```
#define SEEK_SET 0 // позиція на початку файла
long a=0 x1256;
fseek (fd, 20L, SEEK_SET);
fwrite (&a, sizeof(long),1,fd);
```

Індексно-послідовна модель передбачає, що для кожного опрацьованого елемента даних вводиться ідентифікатор даних – ключ. Звертання до елементів даних здійснюється через значення ключа. Прикладом такого ключа є номер елемента масиву.

Резюме

1. Структура даних – загальна властивість інформаційного об'єкту, з яким взаємодіє та чи інша програма. Ця загальна властивість характеризується: множиною допустимих значень цієї структури; набором допустимих операцій; характером організованості.

2. Рівні описування даних: абстрактний, логічний, фізичний.

3. Структури даних поділяються на вбудовані (реалізовані в мовах програмування) та похідні (утворюються користувачами).

4. Складеним типом даних назвемо тип даних, що складається із скінченної та наперед заданої множини елементів певного типу, які не обов'язково є

атомарними. Перерахуємо складені типи даних: множина, послідовність, матриця, дерево, граф, гіперграф.

5. В зовнішній пам'яті структури даних можна подати таким чином: фізичний блок або схема зберігання. Схема зберігання формується через структури даних прямого доступу, послідовного доступу, індексно-послідовного доступу.

Контрольні запитання

1. Поняття структури даних. Рівні подання структур даних.
2. Рівні описування даних.
3. Основні види (типи) структур даних.
4. Класифікація структур даних у програмах користувача й у пам'яті комп'ютера.
5. Приклади структур даних.
6. Структури даних в оперативній пам'яті.
7. Структури даних у зовнішній пам'яті.
8. Порівняти різні схеми зберігання даних.
9. Навести приклади прямого доступу до даних.

Тести для закріплення матеріалу

1. Структури даних характеризуються:
а) множиною допустимих значень певної структури;
б) описом правил переходу від одного значення до іншого;
в) набором допустимих операцій;
г) набором помилок опрацювання даних;
д) характером організованості.

2. Перерахувати структурні типи даних:
а) дерево;
б) масив;
в) таблиця,
г) запис;
д) множина;
е) рекурсивний тип.

3. Перерахувати лінійні структури даних:
а) масив;
б) таблиця;
в) стек;
г) множина;
д) черга;
е) напрямлений граф.

4. Обрати тип даних, що відповідає визначенню: множина R складається з одного відношення ієрархічного порядку:
а) множина;
б) матриця;

в) послідовність;
г) дерево;
д) граф.

5. Перерахувати схеми зберігання структур даних в зовнішній пам'яті:
а) подвійне слово;
б) напівслово;
в) фізичний блок;
г) прямого доступу;
д) послідовного доступу;
е) індексно-послідовного доступу.

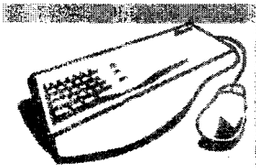
6. Обрати тип даних, що відповідає визначенню: абстрактна структура, у якої множина R складається з одного відношення лінійного порядку:
а) множина;
б) матриця;
в) послідовність;
г) дерево;
д) граф.

7. Обрати тип даних, що відповідає визначенню: множина R складається з одного відношення бінарного порядку:
а) множина;
б) матриця;
в) послідовність;
г) дерево;
д) граф.

8. Обрати тип даних, що відповідає визначенню: множина R складається із двох і більше відношень різного порядку
а) множина;
б) матриця;
в) послідовність;
г) дерево;
д) мультиграф.

9. Обрати можливі варіанти схеми зберігання даних:
а) прямий доступ;
б) обернений доступ;
в) індексний доступ;
г) паралельний доступ;
д) перпендикулярний доступ.

РОЗДІЛ 3



СТРУКТУРНІ ТА ЛІНІЙНІ ТИПИ ДАНИХ

- *Поняття структури даних типу «масив».*
- *Набір допустимих операцій для СД типу «масив».*
- *Дескриптор СД типу «масив».*
- *Ефективність масивів.*
- *Зберігання багатовимірних масивів.*
- *Структура даних типу «множина».*
- *СД типу «запис».*
- *СД типу «таблиця».*
- *СД типу «стек».*
- *СД типу «черга».*
- *СД типу «дек».*

У розділі описано одну з найпростіших та водночас найпоширеніших структур даних – масив. Описано операції, що можуть виконуватися над масивом. Показано способи подання масивів. Також здійснено опис структури даних типу «запис» і показано, що записи найчастіше подаються за допомогою масивів. Подано характеристики динамічних структур даних типу «стек», «черга», «дек» та показано відмінності між ними. Показано відображення стека та черги на масив та на список.

3.1. ПОНЯТТЯ СТРУКТУРИ ДАНИХ ТИПУ «МАСИВ»

✱ ***Масив*** *– послідовність елементів одного типу, який називається базовим. Математичною мовою масив – це функція з обмеженою областю визначення. Структура масивів однорідна. Для виділення окремого компонента масиву використається індекс. Індекс – це значення спеціального типу, визначеного як тип індексу певного масиву. Тому на логічному рівні СД типу «масив» можна записати так:*

type A = array [T1] of T2,

де $T1$ – базовий тип масиву, $T2$ – тип індексу.

Якщо D_{T1} – множина значень елементів типу $T1$, D_{T2} – множина значень елементів типу $T2$, то A: D_{T1} ® D_{T2} (відображення).

Кардинальне число Car(T) структури типу T – це множина значень, які може приймати задана структура типу Т. Кардинальне число характеризує об'єм пам'яті, необхідний такій структурі.

Для масиву A: Car(A) = [Car($T2$)] Car($T1$).

Масив може бути *одновимірним* (вектором), та *багатовимірним* (наприклад, двовимірною таблицею), тобто таким, де індексом є не одне число, а кортеж (сукупність) із декількох чисел, кількість яких співпадає з розмірністю масиву.

У переважній більшості мов програмування масив є стандартною вбудованою структурою даних.

Отже, з вищенаведеного сформулюємо такі властивості масиву:

- ✓ *усі елементи масиву мають той самий тип;*
- ✓ *кожний компонент має свій номер у послідовності (індекс) і відрізняється ним від інших елементів (ідентифікується);*
- ✓ *множина індексів (індексова множина) скінченна й зафіксована в означенні масиву і ід час виконання програми не змінюється;*
- ✓ *можливість опрацювання компонента, або його доступність, не залежить від його місця в послідовності (елементи рівнодоступні).*

3.2. НАБІР ДОПУСТИМИХ ОПЕРАЦІЙ ДЛЯ СД ТИПУ «МАСИВ»

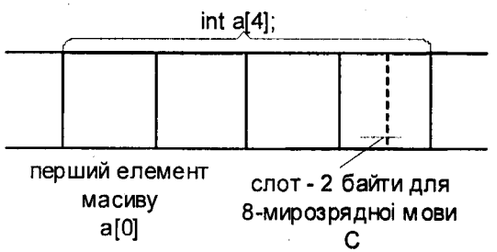
Над масивом можна виконувати такі операції:

1) Операція доступу (доступ до елементів масиву – прямий; від розміру структури операція не залежить).

2) Операція присвоювання.

3) Операція ініціалізації (визначення початкових умов).

На фізичному рівні СД типу «масив» є неперервною ділянкою пам'яті елементів однакового об'єму. Ділянка пам'яті, необхідна для одного елемента, називається ***слотом.***



Var B: A {визначаємо змінну *B* як змінну типу «масив *A*»};

$p \le i \le g$, де p – індекс першого елемента масиву, g – індекс останнього елемента масиву, i – індекс елемента.

3.3. ДЕСКРИПТОР СД ТИПУ «МАСИВ»

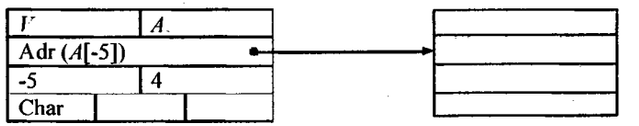
Нерідко фізичній структурі ставиться у відповідність дескриптор (заголовок), що містить загальні відомості про задану фізичну структуру. Дескриптор також зберігається, як і структура, в пам'яті. Загалом дескриптор являє собою структуру типу «запис».

Стосовно до СД типу «масив», дескриптор містить такі компоненти: ім'я масиву, умовне позначення заданої структури, адресу першого елемента масиву, індекси нижньої й верхньої границь масиву, тип елемента масиву, розмір слота.

Наприклад, для наступного описування масиву:

```
var A: array [-5 .. 4] of Char
```

дескриптор буде виглядати так:



Для СД типу «масив» розмір дескриптора не залежить від розмірності масиву. При кожній операції доступу використовується вся інформація дескриптора. Наприклад, поля границі зміни індексу використовуються при обробці виняткових операцій.

3.4. ЕФЕКТИВНІСТЬ МАСИВІВ

Масиви ефективні при звертанні до довільного елемента, яке відбувається за постійний час ($\Theta(1)$), однак такі операції як додавання та видалення елемента, потребують часу $\Theta(n)$, де n – розмір масиву. Тому масиви переважно використовуються для зберігання даних, до елементів яких відбувається довільний доступ без додавання або видалення нових елементів, тоді як для алгоритмів з інтенсивними операціями додавання та видалення, ефективнішими є зв'язані списки.

Інша перевага масивів, яка є досить важливою – це можливість компактного збереження послідовності їх елементів в локальній області пам'яті (що не завжди вдається, наприклад, для зв'язаних списків), що дозволяє ефективно виконувати операції з послідовного обходу елементів таких масивів.

Масиви є дуже економною щодо пам'яті структурою даних. Для збереження 100 цілих чисел у масиві треба рівно у 100 разів більше пам'яті, ніж для збереження одного числа (плюс, можливо, ще декілька байтів). У той же час, усі структури даних, які базуються на вказівниках, потребують додаткової пам'яті для збереження самих вказівників разом із даними. Однак, операції з фіксованими масивами ускладнюються тоді, коли виникає необхідність додавання нових елементів у вже заповнений масив. Тоді його слід розширювати, що не завжди можливо і для таких задач слід використовувати зв'язані списки, або динамічні масиви.

У випадках, коли розмір масиву є досить великий і використання звичайного звертання за індексом стає проблематичним, або великий відсоток його комірок не використовується, треба звертатися до асоціативних масивів, де проблема індексування великих об'ємів інформації вирішується оптимальніше. В асоціативному масиві замість числових індексів використовуються ключі будь-яких типів. Дані в асоціативному масиві так само можуть бути різнотипними. Така структура також відома як «хеш» або як «словник». Це лише відображення «назва-значення», як показано нижче:

```
h = {1 => 2, «2» => «4»}
print hash,»\n»
print hash[1],»\n»
print hash[«2»],»\n»
print hash[5],»\n»
```

З тої причини, що масиви мають фіксовану довжину, треба дуже обережно ставитися до процедури звертання до елементів за їхнім індексом, тому що намагання звернутися до елемента, індекс якого перевищує розмір такого масиву (наприклад, до

елемента з індексом 6 у масиві з 5 елементів), може призвести до непередбачуваних наслідків.

Треба також бути уважним щодо принципів нумерації елементів масиву, яка в одних мовах програмування може починатись з 0, а в інших – з 1.

3.5. ЗБЕРІГАННЯ БАГАТОВИМІРНИХ МАСИВІВ

Збереження одновимірного масиву в пам'яті є тривіальним, тому що сама пам'ять комп'ютера є одновимірним масивом. Для збереження багатовимірного масиву ситуація ускладнюється. Припустимо, що ми хочемо зберігати двовимірний масив такого вигляду:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Найпоширеніші способи його організації в пам'яті наведено нижче.

Розташування «рядок за рядком». Це найбільш уживаний на сьогодні спосіб, який зустрічається у більшості мов програмування:

1 2 3 4 5 6 7 8 9

Розташування «стовпчик за стовпчиком». Такий метод розташування масивів використовується, зокрема, в мові програмування Fortran:

1 4 7 2 5 8 3 6 9

Масив із масивів. Багатовимірні масиви репрезентуються одновимірними масивами вказівників на одновимірні масиви. Розташування може бути як «рядок за рядком», так і «стовпчик за стовпчиком».

Перші два способи дозволяють розміщувати дані компактніше (мають більшу локальність), однак це одночасно і обмеження: такі масиви мають бути «прямокутними», тобто кожний рядок має містити однакову кількість елементів. Розташування «масив із масивів», з іншого боку, не дуже ефективне щодо використання пам'яті (необхідно зберігати додатково інформацію про вказівники), але знімає обмеження на «прямокутність» масиву.

Особливим видом масивів є **розріджений масив** (матриця), в якому дані подані не незперервно, а фрагментарно. Алгоритми опрацювання розріджених матриць передбачають дії тільки з ненульовими елементами і, отже, кількість операцій буде пропорційна кількості ненульових елементів.

Існують різні методи зберігання елементів матриці в пам'яті, наприклад, лінійний зв'язний список. Можна зберігати матрицю, використовуючи кільцевий зв'язний список, двонапрямлені стеки і черги (детальніше розглянуто далі). Існує діагональна схема зберігання симетричних матриць, а також зв'язні схеми розрідженого зберігання. Зв'язна схема зберігання матриць, запропонована Кнутом [7, 8], пропонує зберігати в масиві (наприклад, в *AN*) в довільному порядку самі елементи, індекси рядків і стовпців відповідних елементів (наприклад, в масивах *I* і *J*), номер (з масиву *AN*) наступного ненульового елемента, розташованого в матриці у рядку (*NR*) і у стовпці (*NC*), а також номера елементів, з яких починається рядок (вказівники для входу в терміні – *JR*) і номери елементів, з яких починається стовпець (вказівники для входу в стовпець – *JC*). Така схема зберігання надлишкова, але дозволяє легко здійснювати будь-які операції з елементами матриці.

Найширше вживана схема зберігання розріджених матриць – це схема, запропонована Чангом і Густавсоном: «розріджений рядковий формат» [22]. Ця схема висуває мінімальні вимоги до пам'яті і дуже зручна при виконанні операцій додавння, множення матриць, транспонування, розв'язування систем лінійних рівнянь, при зберіганні коефіцієнтів у розріджених матрицях і т.ін. У цьому випадку значення ненульових елементів зберігаються в масиві *AN*, відповідні їм індекси стовпців – у масиві *JA*. Крім того, використовується масив вказівників, наприклад *IA*, що відзначають позиції *AN* і *JA*, з яких починається опис чергової стрічки. Додатковий компонент в *IA* містить вказівник першої вільної позиції в *JA* і *AN*.

На практиці для роботи з розрідженим масивом розробляються функції:

а) для перетворення індексів масиву в індекс вектора;

б) для визначення значення елементу масиву з його запакованого подання за двома індексами (рядок, стовпчик);

в) для запису значення елементу масиву в його запакованому поданні за двома індексами.

При такому підході звернення до елементів початкового масиву виконується за допомогою вказаних функцій. Наприклад, нехай є двовимірна розріджена матриця, у якій усі ненульові елементи розташовані у шаховому порядку, починаючи з другого елемента. Для такої матриці формула обчислення індексу елементу в лінійному поданні буде такою:

$$L=((y-1)*XM+x)\ /2),$$

де L – індекс у лінійному поданні; x, y – відповідно рядок та стовпчик у двовимірному поданні; XM – кількість елементів у рядку початкової матриці.

Програмна реалізація цього методу така (значення *XM* відоме):

```
Function PutTab(y,x,value: integer): boolean;
Function GetTab(x,y: integer): integer;
Var arrp: array[1..XM*XM div 2] of integer;
Function NewIndex(y, x : integer): integer;
   var i: integer;
   begin  NewIndex:=((y-1)*XM+x) div 2);  end;
Function PutTab(y,x,value : integer): boolean;
   begin
     if NOT ((x mod 2<>0) and (y mod 2<>0)) or
       NOT ((x mod 2=0) and (y mod 2=0))  then begin
         arrp[NewIndex(y,x)]:=value;  PutTab:=true; end
                                 else PutTab:=false;
  end;
Function GetTab(x,y: integer): integer;
   begin
     if ((x mod 2<>0) and (y mod 2<>0)) or
       ((x mod 2=0) and (y mod 2=0)) then GetTab:=0
    else GetTab:=arrp[NewIndex(y,x)];
   end;
end.
```

Стисле подання матриці зберігається у векторі *arrp*.

Функція *NewIndex* виконує перерахунок індексів за наведеною вище формулою і повертає індекс елемента у векторі arrp.

Функція *PutTab* виконує зберігання у стислому поданні одного елемента з індексами x, y і значенням value. Зберігання виконується тільки у тому випадку, якщо індекси x, y адресують не нульовий елемент. Якщо зберігання виконане, функція повертає true, інакше – false.

Для доступу до елемента за індексами двовимірної матриці використовується функція *GetTab*, яка за індексах x, y повертає вибране значення. Якщо індекси адресують нульовий елемент матриці, функція повертає 0.

Розглянемо дещо інший спосіб ефективного зберігання розрідженої матриці: для матриці створюється дескриптор – масив *desc*, який заповнюється при ініціалізації матриці таким чином, що *i*-ий елемент масиву *desc* містить індекс першого елементу *i*-ого рядка матриці у її лінійному поданні. Такий метод спрощує доступ до кожного елемента матриці (функція *NewIndex*): за номером рядка з дескриптора відразу вибирається індекс початку рядка і до нього додається зсув елемента зі стовпчика *x*. Процедури *PutTab* і *GetTab* – такі ж, як і в попередній реалізації, тому тут не наводяться.

```
Function PutTab(y,x,value : integer): boolean;
Function GetTab(x,y: integer): integer;
Procedure InitTab;
Var arrp: array[1..XM*XM div 2] of integer;
  desc: array[1..XM] of integer;
Procedure InitTab;
var i : integer;
 begin
   desc[1]:=0;  for i:=1 to XM do desc[i]:=desc[i-1]+XM;
 end;
Function NewIndex(y, x : integer): integer;
var  i: integer;
begin  NewIndex:=desc[y]+x div 2;  end;
end.
```

3.6. СД ТИПУ «МНОЖИНА»

** **Множина** – скінчений набір елементів одного типу, для яких не важливий порядок слідування і жоден з елементів не може бути два рази включений. Така СД визначається конструкцією type* T = *set of* T0, *де* T0 – *вбудований або раніше визначений тип даних (базовий тип). Значеннями змінних типу* T *є множини елементів типу* T0 *(зокрема, порожні множини).*

Кардинальне число множини (потужність) рівне кількості її елементів.

Набір допустимих операцій для СД типу «множина»: «*» – перетин множин, «+» – об'єднання множин, «-» – різниця множин, «in» – перевірка належності до множини елемента базового типу.

Дескриптор СД типу «множина» не відрізняється від дескриптора СД типу «масив».

Подамо програму, яка виводить усі цифри, що не входять у десятковий запис числа.

Для цього скористаємося множиною s, що міститиме усі цифри.

```
Var s:set of 0..9;
  n,ost,i:integer; {n – число, яке треба проаналізувати}
begin write('Input number');
   readln(n);
  s:=[0,1,2,3,4,5,6,7,8,9]; {включили у множину всі цифри}
   while n>0 do
    {виділяємо цифри числа методом ділення його на 10}
    begin
       {визначаємо остачу від ділення}
        ost:=n mod 10;
        n:=n div 10;
        if (ost in s) then s:=s-[ost] {здійснюємо операцію різниці}
     end;
{виведемо всі цифри, що не належать до запису числа, за зростанням}
   for i:=0 to 9 do
    if i in s then write(i,',')
end.
```

Найчастіше множини використовуються для формування набору елементів, що зустрічаються у масивах лише один раз.

3.7. СД ТИПУ «ЗАПИС (ПРЯМИЙ ДЕКАРТОВИЙ ДОБУТОК)»

** **Запис** – послідовність елементів, які, в загальному випадку, можуть бути одного типу. На логічному рівні СД типу «запис» можна записати так:*

```
type T = Record
S1: T1;
S2: T2;
.....
Sn: Tn;
End;
```

Тут: T_i змінюється при i = 1, 2, ..., n; S_1, ..., S_n – ідентифікатори полів; T_1, ..., T_n – типи даних. Якщо T_i також є у свою чергу записом, то S_i – ієрархічний запис.

Якщо DT_1 – множина значень елементів типу T_1, DT_2 – множина значень елементів типу T_2, ... , DT_n – множина значень елементів типу T_n, де DT – множина значень елементів типу T буде визначатися за допомогою прямого декартового добутку: $DT = DT_1 \times DT_2 \times \ldots \times DT_n$. Іншими словами, множина допустимих значень СД типу

$$\text{«запис»: } Car(T) = \prod_{i=1}^{n} Car(Ti).$$

Допустимі операції для СД типу «запис» аналогічні до операцій для СД типу «масив».

Дескриптор СД типу «запис» містить у собі: умовне позначення, назва структури, кількість полів, вказівник на перший елемент (у випадку прямокутної СД), характеристики кожного елемента, умовні позначення типу кожного елемента, розмір слота, а також зсув, необхідний для обчислення адреси.

Загалом, зсув – це адреса компоненту (поля) r_i щодо початкової адреси запису r. Зсув обчислюється так: $k_i = S_1 + S_2 + \ldots + S_{i-1}$, $i=1,2, \ldots,n$, де S_i – розмір слота кожного елемента запису.

Дескриптор СД типу «запис», на відміну від дескриптора СД типу «масив», залежить від кількості елементів запису.

Приклад структури на мові Сі:

```
typedef struct { char name[20];
          char city[30];
          char dcount[10];
        } school;
```

Тут оголошена структура (запис), що складається із 3 стрічкових компонентів різної довжини і визначено назву цієї структури – school.

Покажемо, яким чином можна поєднати особливості СД типу «множина» та СД типу «запис». Нехай виникла задача формування списку міст, у яких розміщені задані школи. Оскільки у кожному місті є, як мінімум, одна школа, то назви міст у записах можуть повторюватись. Для цього сформуємо множину міст. Оскільки мова С не підтримує СД «множина», то сформуємо множину за допомогою масиву.

```
#include<stdio.h>
#include<string.h>
struct school
{
 char name[20];
 char city[20];
 char dcount[10];
} a[10];
void main()
{
// множина міст
 char set[10][20];
//на початку роботи програми множина порожня
 int k=0,i,j;
 for(i=0;i<10;i++)
   scanf("%s%s%s",&a[i].name,&a[i].city,&a[i].dcount);
 strcpy(set[0],a[0].adress);
 for(i=1;i<10;i++)
  {
   int l=0;
   for(j=0;j<=k;j++)
//перевірка, чи є елемент у множині
   if(strcmp(a[i].city,set[j])==0) l++;
//якщо немає, то включаємо
   if(!l) strcpy(set[++k],a[i].city);
  }
 for(j=0;j<=k;j++)
  printf("%s\n",set[j]);
}
```


3.8. СД ТИПУ «ТАБЛИЦЯ»

✱ ***Таблиця*** *– послідовність записів, які мають ту саму організацію. Такий окремий запис називається* ***елементом таблиці****. Найчастіше використовується простий запис. Отже, таблиця – це агрегація елементів. Якщо послідовність записів впорядкована щодо певної ознаки, то така таблиця називається* ***впорядкованою****, інакше – таблиця* ***невпорядкована****.*

Класифікацію СД типу таблиця подано на рис. 3.1.

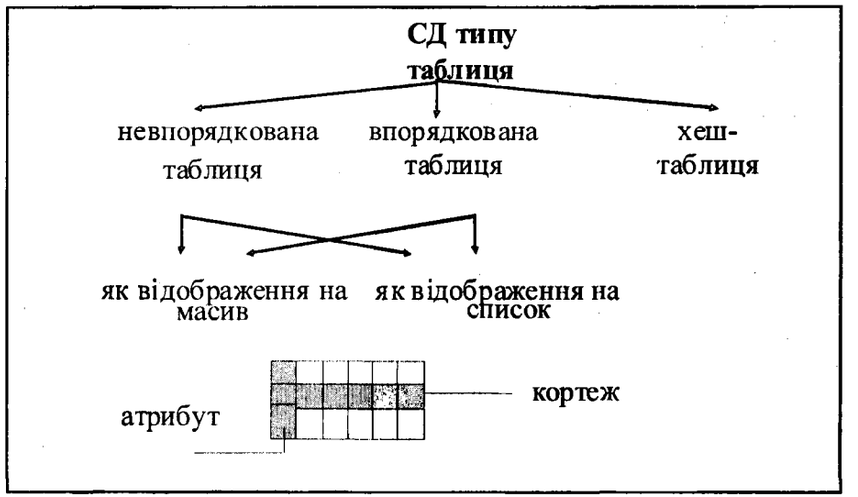


Рис. 3.1. Класифікція СД типу «таблиця».

Якщо один елемент d_i, то *кортеж* – це $<d_1, d_2, \ldots, d_n>$, причому DT_i O d_i. Множина значень елементів типу T (множина допустимих значень СД типу «таблиця») буде визначатися за допомогою прямого декартового добутку:

$DT = DT_1$ r DT_2 r … DT_n, причому DT_i O $<d_1, d_2, \ldots, d_n>$...

Сам елемент таблиці можна подати у вигляді двійки $<K, V>$, де K – ключ, а V – тіло елемента. Ключем може бути різна кількість полів, які визначають цей елемент. Ключ використовується для операції доступу до елемента, тому що кожний із ключів унікальний для заданого елемента. Отже, таблиця є сукупністю двійок $<K, V>$.

На логічному рівні елемент СД типу «таблиця» описуванняється так (приклад на мові Паскаль):

```
Type Element = record
Key: integer;
{опис інших полів}
end;
```

При реалізації таблиці як відображення на масив її опис виглядає так:

```
Tabl = array [0 .. N] of Element.
```

Під час виконання програми кількість елементів може змінюватися. Структура, у якій змінюється кількість елементів під час виконання програми, називається *динамічною*. Якщо розглядати динамічну структуру як відображення на масив, то така структура називається *напівстатичною*.

Перед тим як визначити операції, які можна виконувати над таблицею, розглянемо **класифікацію операцій**.

Конструктори – операції, які створюють об'єкти розглянутої структури.

Деструктори – операції, які руйнують об'єкти розглянутої структури. Ознакою цієї операції є звільнення пам'яті.

Модифікатори – операції, які модифікують відповідні структури об'єктів. До них належать динамічні й напівстатичні модифікатори.

Спостерігачі – операції, у яких елементом (вхідним параметром) є об'єкти відповідної структури, а повертають ці операції результати іншого типу. Отже, операції-спостерігачі не змінюють структуру, а лише подають інформацію про неї.

Ітератори – оператори доступу до вмісту частини об'єкта у певному порядку.

Набір допустимих операцій для СД типу «таблиця» подаємо нижче

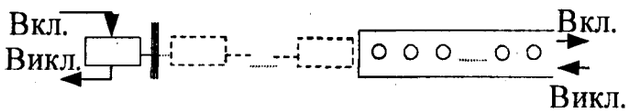
- *Операція ініціалізації (конструктор).*
- *Операція включення елемента в таблицю (модифікатор).*
- *Операція виключення елемента з таблиці (модифікатор).*

Операції-предикати:

- ✓ *таблиця порожня / таблиця не порожня (спостерігач),*
- ✓ *таблиця переповнена / таблиця не переповнена (спостерігач).*
- ✓ *Читання елемента за ключем (спостерігач).*

3.9. СД ТИПУ «СТЕК»

✸ ***Стек*** *– це послідовність, у якій включення й виключення елемента здійснюється з однієї сторони послідовності (вершини стека). Так само здійснюється й операція доступу. Структура функціонує за принципом LIFO (останній, що прийшов, обслуговується першим). Умовні позначення стека зображені на рис 3.2.*



а) відображеня на масив *б) відображення на список*

Рис. 3.2. Організація стека.

При реалізації стека розглядаються стек як відображення на масив і стек як відображення на список.

Відображення на масив передбачає оголошення звичайного масиву та змінної, значення якої дорівнюватиме значенню індексу елемента, що відіграватиме роль «голови» (елемента, на який вказуватиме вказівник):

```
int a[100]; // оголошення стеку
int n=0; // дійсна кількість елементів у стека
int current=99; // індекс «голови», діє принцип LIFO
void pop () // функція видобування (вилучення) елемента зі стека
{
    if (n!=0)
    {
        current++; // зсунули «голову» на один елемент вперед
```



```
        printf("%d%, a[current]);
        n—;                    // зменшили кількість елементів
    }
}
void push () // функція додавання елемента до стеку
{
    if (n<99)
    {
        current—;
        //додали «голову», зсунувши індекс на один елемент вперед
        scanf("%d%, &a[current]);
        n++;                   // збільшили кількість елементів
    }
}
```

Відображення на список передбачає оголошення динамічної структури. Перевагою такого відображення є відсутність обмежень на максимальну кількість елементів у стеку, але, водночас, передбачає підтримку складнішої вказівникової структури:

```
struct stack {
        int el;                // значення елемента
        struct stack *next;} st   // адреса наступного елемента
st *head, *p1, *p2;            //вказівник на «голову», допоміжні вказівники
st* push(int a; st *cur)
// функція додавання елемента,
// a – значення, яке треба внести у стек
// cur – вершина («голова») стека
{ st *p;
    // якщо стек порожній
    if(!cur)
    {
        // створюємо вершину
        cur=(st*)malloc(sizeof(st));
        cur->el=a;
        cur->next=NULL;
        return(cur);
    }
    else
    {
        // створюємо новий елемент, який стане вершиною стека
        p=(st*)malloc(sizeof(st));
        p->el=a;
        p->next=cur;
        return(p);
    }
}
st* pop()                      // видалення вершини стека
{ st *p;
    if(head) // якщо в стеку є елементи
```

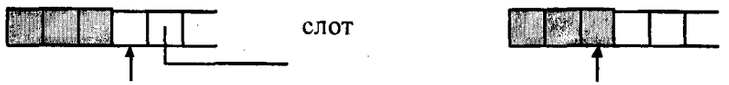
```
    {    .// вершиною стає наступний елемент
         p=head->next;
         // знищуємо «голову»
         free(head);
         return(p);
    }
else return (NULL);
}
```

Сукупність операцій, що визначають структуру типу «стек», подана нижче.

- ✓ *Операція ініціалізації.*
- ✓ *Операція включення елемента в стек.*
- ✓ *Операція виключення елемента зі стека.*
- ✓ *Операція перевірки: стек порожній / стек не порожній.*
- ✓ *Операція перевірки: стек переповнений / стек не переповнений (ця операція характерна для стека як відображення на масив).*
- ✓ *Операція читання елемента (доступ до елемента).*

Є дві модифікації стека:

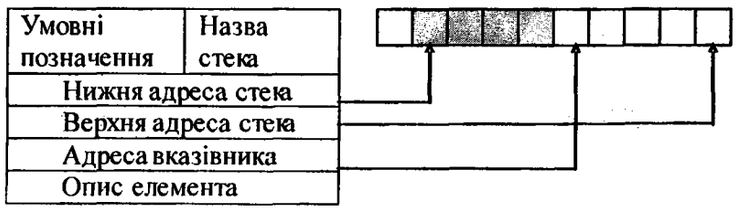
- *вказівник перебуває на вершині стека, показуючи на перший порожній елемент;*
- *слот*
- *вказівник вказує на перший заповнений елемент.*



3.9.1. Дескриптор СД типу «стек»

Дескриптор СД типу «стек» містить:

- ✓ *адреси початку та кінця стека,*
- ✓ *адресу вказівника,*
- ✓ *опис елементів*



3.9.2. Області застосування СД типу «стек»

Стек використовується при перетворенні рекурсивних алгоритмів у нерекурсивні. Зокрема, за допомогою стека можна модифікувати алгоритм сортування Хоора (описаний у наступних розділах) [4, 7, 8].

Стек використовується при розробленні компіляторів.

Стеки вплинули й на архітектуру комп'ютера, послужили основою для стекових машин. У такого комп'ютера акумулятор виконаний у вигляді стека, що дозволяє

розширити спектр безадресних команд, тобто команд, що не вимагають явного задання адрес операндів. Наслідком використання стека є збільшення швидкості опрацювання.

3.10. СД ТИПУ «ЧЕРГА»

✱ ***Черга** – послідовність, у яку включають елементи з одного боку, а виключають – з іншого. Структура функціонує за принципом FIFO (надійшовший першим, обслуговується першим). Умовне позначення черги подане на рис 3.3.*

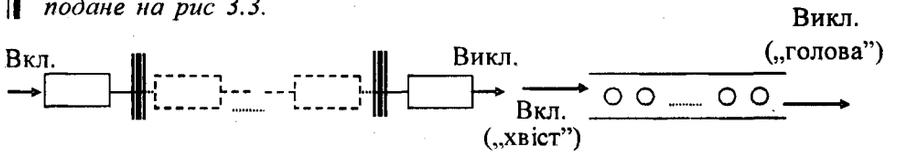


Рис. 3.3. СД типу «черга».

При реалізації черги розглядаються черга як відображення на масив (напівстатична реалізація) і черга як відображення на список.

Відображення на масив:

```
int a[100]; // оголошення черги
int n=0; // індекс останнього елемента у черзі
int current=0; // індекс «голови», діє принцип FIFO
void pop () // функція видобування (вилучення) елемента з черги
{
    if (n!=0)
    {
        current++; // зсунули «голову» на один елемент вперед
        printf("%d%, a[current]);
    }
}
void push () // функція додавання елемента до черги
{
    if (!current) // черга порожня
    {
        scanf("%d%, &a[current]);
    }
    else if (n<99)
    {                       // збільшили кількість елементів
        n++;                // додали елемент у кінець черги
        scanf("%d%, &a[n]);
    }
}
```

Відображення на список:

```
struct cherga {
        int el;                     // значення елемента
        struct charga *next;} ch  // адреса наступного елемента
ch *head, *p1, *p2;                 //вказівник на «голову», допоміжні вказівники
```

```
ch* push(int a; ch *cur)
// функція додавання елемента,
// a – значення, яке треба внести у чергу
// cur – останній елемент черги
{ ch *p;
    // якщо черга порожня
    if(!cur)
    {     // створюємо вершину
          cur=(ch*)malloc(sizeof(ch));
          cur->el=a;
          cur->next=NULL;
          return(cur);
    }
    else
    {     // створюємо новий елемент та додаємо його у кінець
          p=(ch*)malloc(sizeof(ch));
          p->el=a;
          cur->next=p;
          return(p);
    }
}
ch* pop() // видалення вершини черги
{ ch *p;
    if(head) // якщо в черзі є елементи
    {     .// вершиною стає наступний елемент
          p=head->next;
          // знищуємо «голову»
          free(head);
          return(p);
    }
    else return (NULL);
}
```

Сукупність операцій, що визначають структуру типу «черга» подана нижче.

- ✓ *Операція ініціалізації.*
- ✓ *Операція включення елемента в чергу.*
- ✓ *Операція виключення елемента із черги.*
- ✓ *Операція перевірки: черга порожня / черга не порожня.*
- ✓ *Операція перевірки: черга переповнена / черга не переповнена.*

У послідовній пам'яті черга має такий вигляд, як на рис. 3.4.

Тут: Uk_1 – вказівник на «голову» (початок) черги, Uk_2 – вказівник на «хвіст» (кінець) черги.

Щоб уникнути переповнення, раціонально використати кільцеву чергу. При цьому $Uk_1 > Uk_2$ або $Uk_2 > Uk_1$. У випадку ж якщо черга не кільцева, то завжди $Uk_2 > Uk_1$. Якщо черга порожня або переповнена, то $Uk_1 = Uk_2$.

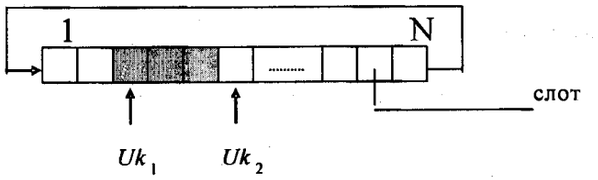


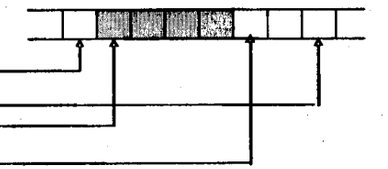
Рис. 3.4. Структура даних типу «черга» у послідовній пам'яті.

Нехай n – поточний стан черги, а N – кількість елементів у черзі, тоді маємо умову $0 \le n \le N$, і значення n може бути умовою перевірки стану черги. Операція включення можлива, якщо $n < N$, а операція виключення – якщо $n > 0$. У випадку кільцевої черги при операції модуля вказівники будуть змінюватися так: $Uk_1 = Uk_1 \bmod N + 1$, $Uk_2 = Uk_2 \bmod N + 1$.

Схема на фізичному рівні СД типу «черга» виглядає так:

Дескриптор:

Умовні позначення	Назва черги
Нижня адреса черги	
Верхня адреса черги	
Вказівник початку черги	
Вказівник кінця черги	
Властивості елементів черги	



Області застосування СД типу «черга»

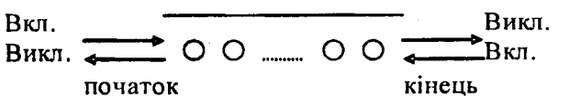
Черга використається при передаванні даних з оперативної у вторинну пам'ять (при цьому відбувається процедура буферизації: накопичується блок і передається у вторинну пам'ять). Наявність буфера забезпечує незалежність взаємодії процесів між виробником і споживачем (рангування задач користувача). Задачі розділяються за пріоритетами:

- *задачі, розв'язувані в режимі реального часу (вищий пріоритет) (черга 1);*
- *задачі, розв'язувані в режимі розділення часу (черга 2);*
- *задачі, розв'язувані в пакетному режимі (фонові задачі) (черга 3).*

Доступ до елементів черги здійснюється послідовно.

3.11. СД ТИПУ «ДЕК»

✸ ***Дек** – особливий вид черги. Дек (з англ. deq – double ended queue, черга з двома кінцями) – це такий послідовний список, у якому як включення, так і виключення елементів може здійснюватися з обох кінців списку. Окремий випадок дека – дек з обмеженим входом і дек з обмеженим виходом. Логічна і фізична структури дека аналогічні логічній і фізичній структурі кільцевої FIFO-черги. Проте, стосовно дека доцільно говорити не про початок і кінець, а про лівий і правий кінець.*



Набір допустимих операцій для СД типу «дек»:

- *ініціалізація;*
- *включення елемента справа;*
- *включення елемента зліва;*
- *виключення елемента справа;*
- *виключення елемента зліва;*
- *визначення розміру;*
- *очищення.*

Така структура даних є однією з найлегших для опрацювання серед зв'язних списків та поєднує позитивні сторонни реалізації списку у вигляді стеку та черги.

Задачіадачі, що вимагають структури дека, зустрічаються в обчислювальній техніці і програмуванні набагато рідше, ніж задачіадачі, що реалізовуються на структурі стека або черги. Як правило, вся організація дека виконується програмістом без якихось спеціальних засобів системної підтримки.

Прикладом дека може бути, наприклад, якийсь термінал, у який вводяться команди, кожна з яких виконується заданий час. Якщо ввести наступну команду і не дочекатися закінчення виконання попередньої, то вона встане у чергу і почне виконуватися, як тільки звільниться термінал. Це FIFO-черга. Якщо ж додатково ввести операцію відміни останньої введеної команди, то виходить дек.

```
#include "stdio.h"
#include "conio.h"
struct node
 {
 int data; // описуємо структуру
 struct node *next;
 struct node *prev;
 };
node *first(int data); // визначаємо перший елемент
node *add(node *p, int data); // додаємо елемент на початок
node *addAfter(node *head, int data); //додаємо елемент у кінець
node *del(node *del, int st); //знищуємо елемент
void view(node *head); //відображення дека
void delDek(node *head); //знищуємо дек
// головна програма
main()
{
node *head, *p;
head=NULL;
int m_gl, m_dob, m_ud, m_vi;
int data;
    while(m_gl!=4)
    {
            clrscr();
           printf("1. Додати елеметн.\n");
```



```
 printf("2. Знищити елемент.\n");
printf("3. Вивести дек.\n");
printf("4. Вихід.\n");
gotoxy(1,6);
scanf("%i", &m_gl);
if(m_gl==1)
{
     while(1)
     {
          clrscr();
          if (head==NULL)
          {
               printf("Дек порожній. Введіть елемент: ");
               scanf("%i", &data);
               head = p = first(data);
          }
          clrscr();
          printf(".....Додати елемент....\n");
          printf("1. На початок.\n ");
          printf("2. У кінець.\n");
          printf("3. Повернутись назад.\n");
          gotoxy(1,6);
          scanf("%i", &m_dob);
          if (m_dob==1)
          {
               clrscr();
               printf("Додати елемент на початок.\n");
               scanf("%i", &data);
               head=addAfter(head, data);
          }
          if (m_dob==2)
          {
               clrscr();
               printf("Додати елемент у кінець.\n");
               scanf("%i", &data);
               p=add(p, data);
                }
          if (m_dob==3) break;
     }
}
if(m_gl==2)
{
     while(1)
     {              // виводимо меню програми
```

```
                clrscr();
                printf("      Знищити елемент....\n");
                printf("1. На початку.\n");
                printf("2. У кінці.\n");
                printf("3. Очистити дек.\n");
                printf("4. Повернутись назад.\n");
                gotoxy(1,6);
                scanf("%i", &m_ud);
                if (m_ud == 2) p = del(p, m_ud);
                else if (m_ud==1) head=del(head, m_ud);
                else if (m_ud==3)
                {delDek(head); head=NULL; printf("Дек очистили!");}
                    if (m_ud==4) break;
            }
        }
        if (m_gl==3)
        {
            while(1)
            {
                clrscr();
                if (head!=NULL) view(head);
                else   printf("Дек порожній.\n");
                printf("1. Повернутись назад.\n");
                scanf("%i", &m_vi);
                if (m_vi==1) break;
            }
        }
    }
}
node *first(int data)
{  //створюємо перший елемент
    node *temp = new node;
    temp->data=data;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}
node *add(node *p, int data)
{   // елемент зі значенням data  розміщуємо після p
    node *temp=new node;
    temp->data=data;
    temp->prev=p;
    temp->next=NULL;
    p->next=temp;
    return temp;
```



```
}
node *addAfter(node *head, int data)
{   // елемент зі значенням data  розміщуємо перед головою
    node *temp = new node;
    temp->data=data;
    temp->next=head;
    head->prev=temp;
    temp->prev=NULL;
    return temp;
}
void view(node *head)
{   // виводимо на екран значення елементів дека
    node *t=head;
    printf(".....DEK.....\n");
        while(t)
         {
            printf("%i\t", t->data);
            t=t->next;
         }
    printf("\n\n");
}
node *del(node *del, int st)
{                  // знищуємо елемент зі значення st
    node *temp;
    if (st==2)
     {
        temp=del->prev;
        temp->next=NULL;
        delete del;
     }
    else if (st==1)
    {
        temp=del->next;
        temp->prev=NULL;
        delete del;
    }
    return temp;
}
void delDek(node *head)
{   // знищуємо усі елементи дека
    node *t;
    for(t=head; t!=NULL; t=t->next)
        delete t->prev;
}
```

Резюме

1. Масив – послідовність елементів одного типу, який називається базовим. Математичною мовою масив – це функція з обмеженою областю визначення. Структура масивів однорідна. Для виділення окремого компонента масиву використовується індекс.

2. Індекс – це значення спеціального типу, визначеного як тип індексу заданого масиву. Над масивом виконують операції: доступу, присвоювання, ініціалізації.

3. Способи організації масивів: рядок за рядком, стовпчик за стовпчиком, масив із масивів.

4. Множина – скінченний набір елементів одного типу, для яких не важливий порядок слідування і жоден з елементів не може бути два рази включений. Набір допустимих операцій над множиною: перетин, об'єднання, різниця, перевірка на належність.

5. Запис – послідовність елементів, які, в загальному випадку, можуть бути одного типу. Дескриптор СД типу «запис», на відміну від дескриптора СД типу «масив», залежить від кількості елементів запису.

6. Таблиця – послідовність записів, які мають однакову організацію. Такий окремий запис називається елементом таблиці. Найчастіше використовується простий запис. Отже, таблиця – це агрегація елементів. Якщо послідовність записів впорядкована щодо певної ознаки, то така таблиця називається впорядкованою, інакше – таблиця невпорядкована. Таблиця може бути реалізована як відображення на масив чи як відображення на список.

7. Є такі класи операцій: конструктори, деструктори, модифікатори, спостерігачі, ітератори.

8. Стек – це послідовність, у якій включення й виключення елемента здійснюється з однієї сторони послідовності (вершини стека). Так само здійснюється й операція доступу. Структура функціонує за принципом LIFO (останній, що прийшов, обслуговується першим).

9. Черга – послідовність, у яку включають елементи з одного боку, а виключають – з іншого. Структура функціонує за принципом FIFO (надійшов першим, обслуговується першим).

10. Дек – це такий послідовний список, у якому як включення, так і виключення елементів може здійснюватися з обох кінців списку.

Контрольні запитання

1. Опишіть структури даних типу «масив».
2. Перерахуйте набір допустимих операцій для структури даних типу «масив».
3. Поняття дескриптора. Приклад.
4. Дескриптор структури даних типу «масив».
5. Структури даних типу «запис» (прямий декартовий добуток).
6. Структури даних типу «таблиця».
7. Класифікація структур даних типу «таблиця».
8. Класифікація операцій над структурами даних типу «таблиця».

9. Набір допустимих операцій для структури даних типу таблиця».
10. Структури даних типу «стек».
11. Сукупність операцій, що визначають структуру типу «стек».
12. Дескриптор структури даних типу «стек».
13. Області застосування структури даних типу «стек».
14. Структури даних типу «черга».
15. Сукупність операцій, що визначають структуру типу «черга».
16. Дескриптор структури даних типу «черга».
17. Області застосування структури даних типу «черга».
18. Області застосування СД типу «дек».

Тести для закріплення матеріалу

1. Перерахувати допустимі операції над масивами:

а) операція доступу;
б) операція розіменування;
в) операція присвоєння;
г) операція індексування;
д) операція ініціалізації.

2. Перерахувати дані, що містить дескриптор масиву:

а) ім'я;
б) умовне позначення;
в) адреса першого елемента;
г) адреса останнього елемента;
д) індекс першого елемента;
е) індекс останнього елемента.

3. Перерахувати операції над множинами:

а) перетин;
б) транспонування;
в) об'єднання;
г) різниця;
д) сума;
е) перевірка належності.

4. Дати визначення структур даних типу «запис»:

а) послідовність елементів, які, в загальному випадку, можуть бути одного типу;
б) послідовність елементів, які, в загальному випадку, можуть бути різного типу;
в) послідовність декількох множин елементів;
г) послідовність декількох масивів.

5. Перерахувати допустимі операції над записами:

а) операція доступу;

б) операція розіменування;
в) операція присвоєння;
г) операція індексування;
д) операція ініціалізації.

6. Типи таблиць:
а) невпорядкована таблиця;
б) впорядкована таблиця;
в) умовно-впорядкована таблиця;
г) хеш-таблиця;
д) відображення на множину;
е) відображення на масив;
є) відображення на список.

7. За визначенням вибрати операцію над таблицею: операція, у якої вхідним параметром є об'єкти відповідної структури, вона повертає результати іншого типу:
а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

8. За визначенням вибрати операцію над таблицею: опера тор доступу до вмісту об'єкту частинами у певному порядку:
а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

9. За визначенням вибрати операцію над таблицею: операція, яка руйнує об'єкти розглянутої структури:
а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

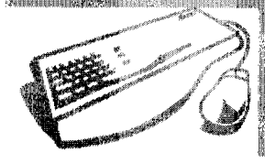
10. За визначенням вибрати операцію над таблицею: операція, яка створює об'єкти розглянутої структури:
а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

11. Перерахувати допустимі операції над таблицями:
а) операція ініціалізації;
б) операція присвоєння;
в) операція включення елемента в таблицю;
г) операція виключення елемента з таблиці;
д) операції-предикати;
е) операція порівняння;
є) читання елемента за ключем.

12. Принцип LIFO діє для:
а) черги;
б) стека;
в) списку;
г) слота.

13. Принцип FIFO діє для:
а) черги;
б) стека;
в) списку;
г) слота.

РОЗДІЛ 4



ЗВ'ЯЗНИЙ РОЗПОДІЛ ПАМ'ЯТІ

- *СД типу «вказівник».*
- *Статичні та динамічні змінні.*
- *Структура даних типу «однозв'язний список».*
- *СД типу «двозв'язний лінійний список».*
- *Багатозв'язний список. Приклади.*

Розділ присвячений огляду структур даних, які не передбачають статичного збереження та відображення на статичні структури. Для подання таких структур даних використовується спеціальний тип даних – вказівник. Відмінності між динамічними та статичними змінними показано на прикладах.

4.1. СД ТИПУ ВКАЗІВНИК

Вказівний тип займає проміжне положення між скалярними й структурними типами: з одного боку значення вказівного типу є атомарним (неподільним), а з іншого, ці типи визначаються через інші (у тому числі й структурні) типи.

Type <тип вказівника> = ^ <тип об'єкту, що вказується>
(цей тип дозволяє використати базовий тип перед описом)

```
Type PtrType = ^BaseType;
BaseType = record
x, y: real;
end;
Var A: PtrType; {A – змінна статичного типу, значенням якої є адреси розташування в пам'яті конкретних значень заданого типу}
B: BaseType;
C: ^PtrType;
```

Змінній *A* можна присвоїти адресу якоїсь змінної, для чого використаємо унарну операцію взяття вказівника: A = @ B.

На фізичному рівні вказівник займає два слоти: у першому слоті перебуває адреса сегмента, у другому – адреса зсуву.

Операції над вказівним типом:

1) операція порівняння на рівність: = (рівність, якщо співпадають адреси сегмента й зсуву);

2) операція порівняння на нерівність: <>;

3) операція доступу: B.X = B.X + C.

Серед всіх можливих вказівників виділяється один спеціальний вказівник, що нікуди не вказує. Тобто у пам'яті виділяється одна адреса, у яку не записується жодна змінна.

На це місце в пам'яті й посилається такий порожній або "нульовий" вказівник, що позначається nil на мові Паскаль або NULL на мові С. Вказівник nil вважається константою, сумісною з будь-яким вказівним типом, тобто це значення можна присвоювати будь-якому вказівному типу.

4.2. СТАТИЧНІ Й ДИНАМІЧНІ ЗМІННІ

4.2.1. Відмінності між статичними та динамічними змінними

Дотепер ми розглядали змінні, які розміщаються в пам'яті відповідно до цілком певних правил, а саме, для локальних змінних, описаних у підпрограмах, пам'ять приділяється при виклику підпрограми; при виході з неї ця пам'ять звільняється, а самі змінні припиняють існування. Глобальним змінним програми пам'ять приділяється на початку її виконання; ці змінні існують протягом усього періоду роботи програми. Іншими словами, розподіл пам'яті у всіх цих випадках відбувається повністю автоматично. Змінні, пам'ять під які розподіляється подібним чином, називаються **статичними**.

Змінні, створенням і знищенням яких може управляти програміст, називаються **динамічними** змінними. Вони можуть перебувати в будь-якій частині динамічної пам'яті, і звертання до них може здійснюватися тільки через вказівник. Засобами доступу до статичних змінних є ідентифікатори цих змінних. Динамічні змінні, кількість яких і місце розташування в пам'яті заздалегідь не відомо, неможливо позначити ідентифікаторами. Тому єдиним засобом доступу до динамічних змінних є вказівник на місце їхнього поточного розташування в пам'яті.

Розглянемо розподіл пам'яті (рис. 4.1).

Під локальні змінні виділяється спеціальний сегмент оперативної пам'яті (сегмент стека). Утворення динамічних змінних реалізується в іншій області пам'яті, що існує окремо від стекового сегмента й називається **купою** (heap) або динамічною областю пам'яті.

Рис. 4.1. Розподіл памяті для динамічних змінних

4.2.2. Створення та знищення динамічних змінних

Основні дії над динамічними змінними – створення та знищення – реалізуються стандартними процедурами *New* і *Dispose* (в мові програмування Паскаль).

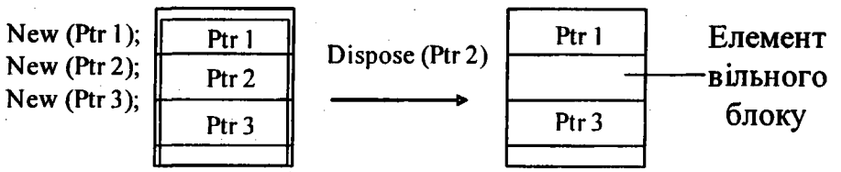
New (<ім'я посилання>): point; – процедура призначена для створення динамічних змінних певного типу (інакше кажучи, для відведення пам'яті в купі для зберігання значень динамічної змінної).

Dispose (<ім'я посилання>): point; – процедура використовується для звільнення пам'яті, відведеної за допомогою процедури *New*.

MaxAvail: longint; - функція повертає максимальний розмір (у байтах) безперервної вільної ділянки купи. Застосування цієї процедури необхідне для контролю динамічної пам'яті при реалізації операції включення.

Наприклад: if MaxAvail > SizeOf (BaseType) then {генеруємо об'єкт}

Створимо три об'єкти, які розташуються в пам'яті послідовно (без фрагментарності), а потім знищимо другий об'єкт. У результаті виникне фрагментарність, якої треба уникати.



MemAvail: longint; – функція повертає загальну кількість вільної пам'яті.

Щоб перевірити, є чи фрагментарність, треба порівняти результати застосування функцій *MaxAvail* і *MemAvail* – вони повинні збігатися.

4.3. КЛАСИФІКАЦІЯ СД ТИПУ «ЗВ'ЯЗНИЙ СПИСОК»

Усі вищерозглянуті структури, що реалізувалися як відображення на масив, мають певні недоліки, тобто вони малоефективні при розв'язуванні деяких задач. До таких недоліків можна віднести наступне.

1. Точно невідомо, скільки елементів буде мати певна структура, тобто не можна зробити оцінку.

2. Якщо ми розглядаємо якусь послідовність елементів у послідовній пам'яті $x_1, x_2, \ldots, x_n$ і необхідно включити який-небудь новий елемент x у цю послідовність, то ми повинні здійснити масову операцію зсуву всіх елементів, що перебувають за тим елементом послідовності, після якого ми хочемо включити новий елемент. Після цього вставимо цей новий елемент x на місце, що звільнилося. Отже, тут проявляється властивість фізичної суміжності (рис. 4.2).

Рис. 4.2. Додавання нового елемента в список.

Позбутися фізичної суміжності можна, якщо елементи будуть мати не тільки дані, але й вказівники. У цьому випадку елементи можуть бути хаотично розкидані по оперативній пам'яті, а логічна послідовність буде забезпечуватися одним або декількома вказівниками.

Класифікація зв'язних списків подана на рис. 4.3.

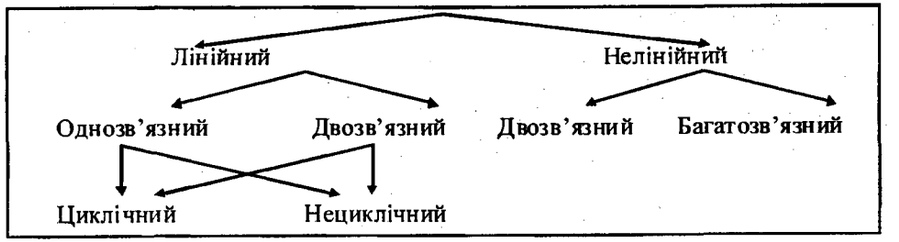


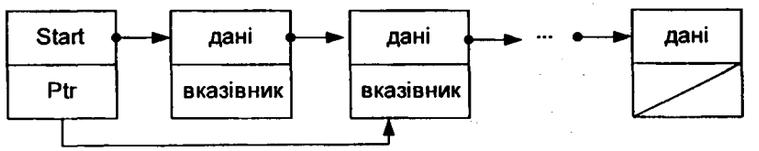
Рис. 4.3. Класифікація зв'язних списків.

Розглянемо кожен із класів зв'язних списків детальніше.

4.4. СД ТИПУ «ЛІНІЙНИЙ ОДНОЗВ'ЯЗНИЙ СПИСОК»

❋ ***Зв'язний список*** *– структура даних, елементами якої є записи, зв'язані один з одним за допомогою вказівників, що зберігаються в самих елементах.*

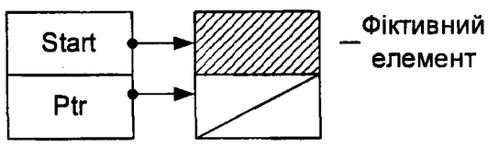
Найпростішим зв'язним списком є *лінійний однозв'язний список*. Кожний елемент у цьому списку складається з різних за призначенням полів: змістовного й поля-вказівника. У типі-вказівнику перебуває адреса наступного елемента. Поле останнього вказівника має вказівку на кінець списку – ознака nil (NULL).



Операції, що визначають структуру типу «лінійний однозв'язний список»:

- ✓ *ініціалізація;*
- ✓ *включити елемент як робочий вказівник списку;*
- ✓ *виключити елемент, що перебуває за робочим вказівником списку;*
- ✓ *пересунути робочий вказівник на один крок;*
- ✓ *перевірка: список порожній / список не порожній;*
- ✓ *помістити робочий вказівник у початок списку (похідна операція);*
- ✓ *помістити робочий вказівник у кінець списку (похідна операція);*
- ✓ *зробити список порожнім.*

При ініціалізації списку ефективно використати реалізацію такої структури:



Реалізацію списку можна здійснити за допомогою відображення на масив.

Операції включення й виключення елементів списку

Розглянемо операції включення й виключення елементів у загальному випадку. Уведемо наступні позначення полів: Data – поле, куди записуються дані; Link – поле,

де перебуває вказівник на наступний елемент списку; Start – вказівник на початок списку; Free – вказівник, який вказує на перший елемент вільного (робочого) списку.

Для реалізації необхідно мати функціональний список, у якому перебуває поле даних, а також вільний список, що є джерелом даних для функціонального. Поле Data у вільному списку не має змістовної інформації. Data (Ptr) указує на поле Data того елемента, на який указує Ptr. Link (Ptr) – вказівник того елемента, на який указує Ptr.

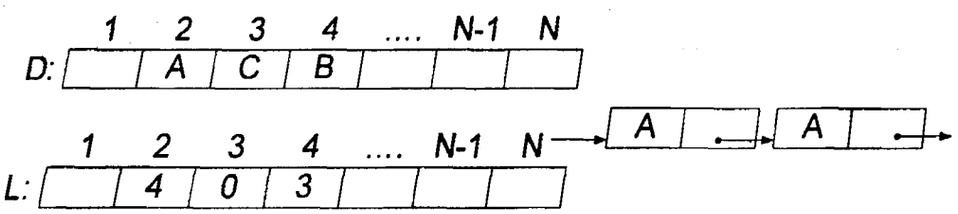
Операції включення й виключення від розмірності списку не залежать.

Включення елемента в список:

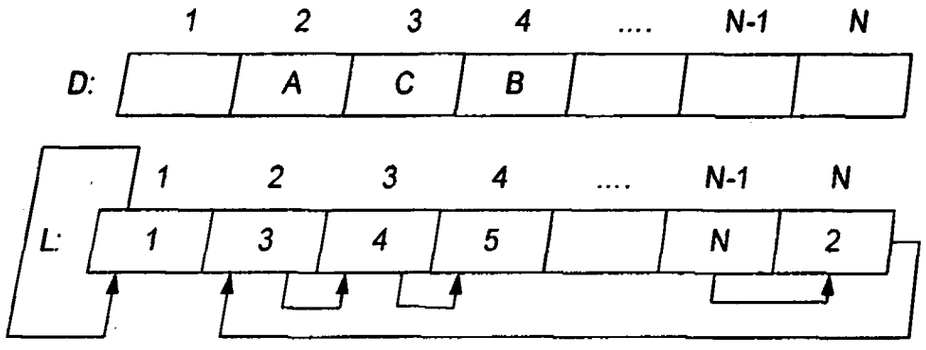
Pntr = Free;
Free – Link(Pntr);
Data(Pntr) = { };
Link(Pntr) – Link(Ptr) {формуємо вказівник у новому елементі};
Link(Pntr) = Pntr {модифікація вказівника}.

Реалізація списку в послідовній пам'яті

У послідовній пам'яті список реалізується за допомогою двох погоджених масивів, перший з яких використовується для запису елементів, а другий – для запису вказівників. Наприклад:



D – масив, для запису елементів, L – масив, для запису вказівників, 0 – вказівник кінця списку.



Будемо вважати, що: L[1] буде вказувати на перший елемент функціонального ску, L[2] – на перший елемент вільного списку.

Після ініціалізації (приведення структури в початковий стан) функціональний список – жній, і ми робимо його циклічним, а вільний список будуть складати всі інші нти (його також робимо циклічним).

ім реалізації списку в послідовній пам'яті, зв'язний список можна реалізувати в динамічній пам'яті (з використанням вказівного типу).

4.5. СД ТИПУ «ЦИКЛІЧНИЙ ЛІНІЙНИЙ СПИСОК»

Структура даних типу «циклічний лінійний список» реалізується за схемою, поданною на рис. 4.4.

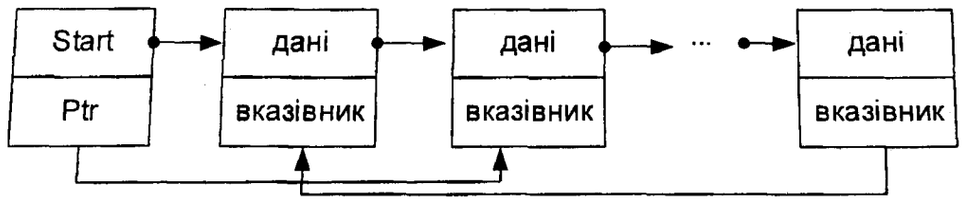


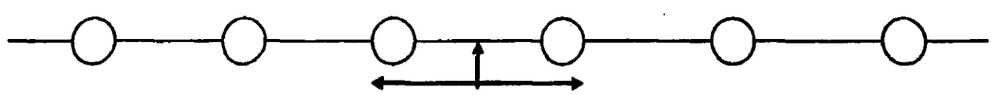
Рис. 4.4. Схема структури даних лінійний циклічний список

Така модифікація лінійного списку дозволяє спростити алгоритми для роботи зі списком, будь-який елемент доступний, але необхідно відслідковувати зациклення, тобто має бути зовнішній вказівник.

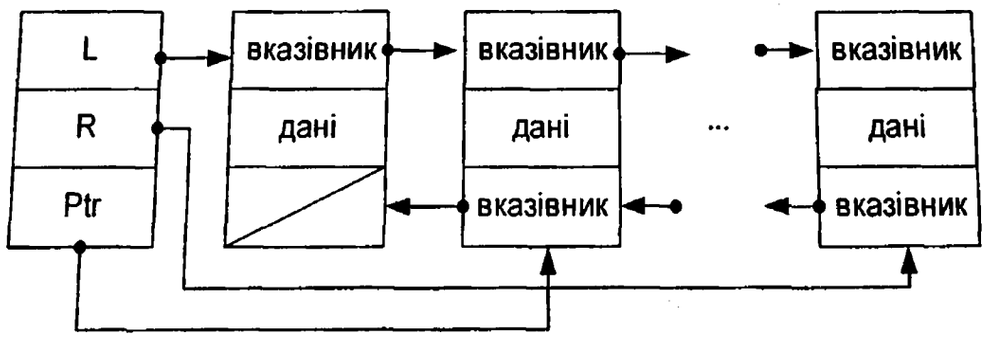
Операції для роботи із циклічним лінійним списком ті самі, що й для однозв'язного лінійного списку.

4.6. СД ТИПУ «ДВОЗВ'ЯЗНИЙ ЛІНІЙНИЙ СПИСОК»

Схему структури даних типу «двозв'язний лінійний список» можна відобразити наступним чином:



У списку такого типу вказівник можна переміщати як в один, так і в другий бік. Дескриптор такої структури складається із трьох полів вказівного типу:



Набір допустимих операцій для СД типу «двозв'язний лінійний список»:

- ✓ *ініціалізація;*
- ✓ *зробити список порожнім;*
- ✓ *перевірка: список порожній / список не порожній;*
- ✓ *встановити вказівник у кінець списку / початок списку (дві окремі процедури);*
- ✓ *пересунути вказівник уперед / назад на один крок;*
- ✓ *включити елемент у список до вказівника / після вказівника;*
- ✓ *виключити елемент зі списку до вказівника / після вказівника.*

Наведемо фрагмент програми створення двозв'язного лінійного списку:

```
typedef  struct dlist
   {
   char *info;
   struct dlist *next;
      struct dlist *prev;
   }list;
char *s;
list *head,*p1,*p2,*p3;
head=malloc(sizeof(list));
gets(s);
strcpy(head->info,s);
head->next=NULL;
head->prev=NULL;
p1=head;
gets(s);
    while(strcmp(s,"exit")!=0)
    {
          p2=malloc(sizeof(list));
          strcpy(p2->info,s);
          p2->next=NULL;
          p2->prev=p1;
          p1->next=p2;
          p2=p1;
          gets(s);
    }
p1=head;
    while(p1!=NULL)
    {
          printf("%s",p1->info);
          p1=p1->next;
    }
```

При цьому для включення елемента в список необхідно розпізнати наступні три ситуації:

- ✓ *список порожній;*
- ✓ *включення елемента в середину списку;*
- ✓ *включення елемента в список як першого.*

Для спрощення операції включення / виключення реалізують *циклічні двозв'язні списки.* При реалізації такого списку формують вказівники (замість маркерів кінця й початку списку), що вказують на перший і останній елементи списку. Допустимі операції для СД типу «циклічний двозв'язний список» аналогічні операціям для лінійного двозв'язного списку.

4.7. БАГАТОЗВ'ЯЗНИЙ СПИСОК. ПРИКЛАДИ

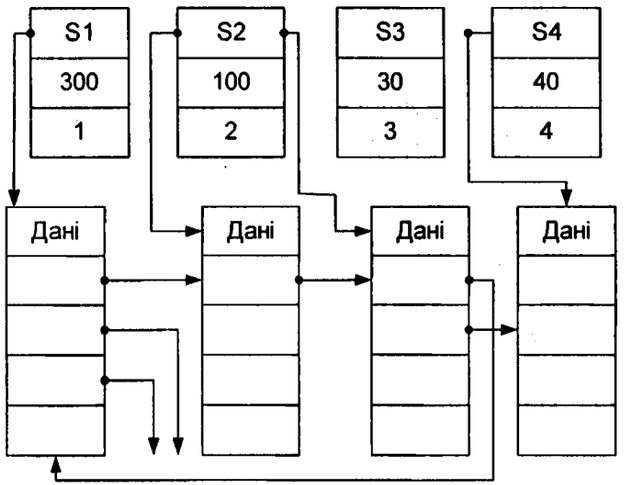
У загальному випадку кількість вказівників може бути довільною. У результаті такого узагальнення ми одержуємо *багатозв'язний список*. Розглянемо кілька прикладів таких списків.

Приклад 4.1. Кожний елемент багатозв'язного списку може входити одночасно в кілька однозв'язних списків. Тобто багатозв'язний список «прошитий» у декількох напрямках.

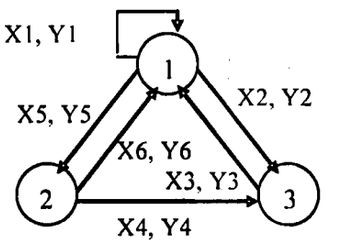
Наприклад, необхідно організувати дані про абітурієнтів, але крім організації всіх відомостей, необхідно із загального списку вибрати абітурієнтів, що характеризуються деякими загальними даними (абітурієнти-медалісти, абітурієнти, що проживають в одному місті; абітурієнти, що здали іспит на «відмінно»).

У такому випадку дані можна організувати у вигляді чотиризв'язного списку, кожний елемент якого містить всі необхідні відомості, а також чотири вказівники: перший зв'язує всіх студентів; другий – тих, що мешкають в одному місті; і т.ін.

Для кожного із чотирьох однозв'язних списків є свій дескриптор, у який входить кількість елементів, умови їхнього розміщення в списку.

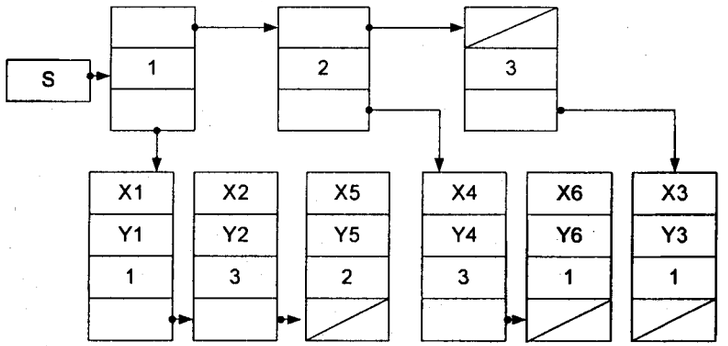


Приклад 4.2. Розглянемо нелінійний зв'язний список, що зручний для подання діаграм стану дискретних систем.



Маємо діаграму із трьох станів. Дуги показують зміну поточного стану. Зміна стану здійснюється під дією вхідного сигналу *X*, при цьому система переходить у новий стан і видає вихідний сигнал *Y*. Подібна діаграма може бути використана в діалоговій

обчислювальній системі, де вхідні сигнали являють собою команди, що вводять у систему з термінала користувача, а вихідні є іменами підпрограм. Отже, одержавши команду X і відшукавши у списку за поточним станом відповідний цій команді елемент описування вихідної дуги системи, виконується підпрограма, зазначена в знайденому елементі. Тим самим реалізується необхідна реакція Y. Після цього система переходить в інший стан, і із цього моменту цей стан стає поточним.



Резюме

1. Структури, що реалізовані як відображення на масив, у разі додавання чи знищення елементів потребують здійснення масової операції зсуву, що є малоефективним.

2. Зв'язний список – структура даних, елементами якої є записи, зв'язані один з одним за допомогою вказівників, що зберігаються в самих елементах.

3. За способом організації зв'язні списки бувають лінійні та нелінійні.

4. За кількістю використаних вказівників зв'язні списки поділяються на однозв'язні, двозв'язні та багатозв'язні.

5. Вказівний тип займає проміжне положення між скалярними й структурними типами: з одного боку значення вказівного типу є атомарним (неподільним), а з іншого, ці типи визначаються через інші (у тому числі й структурні) типи.

6. Змінні, пам'ять під які розподіляється подібним чином, називаються статичними. Змінні, створенням і знищенням яких може керувати програміст, називаються динамічними змінними.

7. Лінійний циклічний список дозволяє спростити алгоритми для роботи зі списком, будь-який елемент доступний, але необхідно відслідковувати зациклення, тобто повинен бути зовнішній вказівник.

8. У двозв'язному лінійному списку вказівник можна переміщати як в один, так і в інший бік. Для спрощення операції включення / виключення реалізують циклічні двозв'язні списки.

Контрольні запитання

1. Зв'язний розподіл пам'яті.

2. Класифікація структури даних типу «зв'язний список».

3. Структури даних типу «лінійний однозв'язний список».

4. Операції, що визначають структуру типу «лінійний однозв'язний список».

5. Операції включення й виключення елементів списку.

6. Реалізація списку в послідовній пам'яті.

7. Структури даних типу «вказівник».

8. Операції над вказівним типом.

9. Статичні й динамічні змінні.

10. Створення й знищення динамічних змінних.

11. Структури даних типу «циклічний лінійний список».

12. Структури даних типу «двозв'язний лінійний список».

13. Набір допустимих операцій для структури даних типу «двозв'язний лінійний список».

14. Багатозв'язний список. Приклади.

Тести для закріплення матеріалу

1. Перерахуйте лінійні списки:

а) циклічний;
б) нециклічний;
в) синхронний;
г) асинхронний;
д) двозв'язний;
е) багатозв'язний.

2. Перерахуйте нелінійні списки:

а) циклічний;
б) нециклічний;
в) синхронний;
г) асинхронний;
д) двозв'язний;
е багатозв'язний.

3. Перерахуйте операції над вказівниками:

а) операція порівняння на рівність;
б) операція порівняння на нерівність;
в) операція доступу;
г) операція зменшення;
д) операція анулювання.

4. Нехай дано опис структури даних:

struct mmm { int x; mmm *next;} *p1, *p2;

За фраґментом програми визначте структуру, що утворюється зf його допомогою:

```
p1=(mmm*)malloc(sizeof(mmm));
scanf ("%d", &xx);
p1->x=xx; p1->next=null;
scanf ("%d", &xx);
    while (xx!=0)
    {
        p2=(mmm*)malloc(sizeof(mmm));
        p2->x=xx; p2->next=null;
        p1->next=p2; p1=p2;
        scanf ("%d", &xx);
    }
```

а) черга,
б) стек,
в) список,
г) слот.

5. Нехай дано опис структури даних:

struct mmm { int x; mmm *next;} *head, *p1, *p2;

За фраґментом програми визначте структуру, що утворюється з його допомогою:

```
head=(mmm*)malloc(sizeof(mmm));
scanf ("%d", &xx);
head->x=xx;
head->next=null; p1=head;
scanf ("%d", &xx);
    while (xx!=0)
    {
        p2=(mmm*)malloc(sizeof(mmm));
        p2->x=xx; p2->next=null;
        p2->next=p1; p1=p2;
        scanf ("%d", &xx);
    }
```

а) черга;
б) стек;
в) список;
г) слот.

6. За фраґментом програми визначте її призначення

```
struct mmm { int x; mmm *next;} *head, *p1, *p2;
...
    while(head)
    {
        p1=head->next;
        dispose (head);
        head=p1;
    }
```

а) пошук елемента;
б) додавання елемента;
в) знищення елемента;
г) знищення списку;
д) знищення множини.

7. За фраґментом програми визначте її призначення

```
struct mmm { int x; mmm *next;} *head, *p1, *p2;
...
    while(head->next)
    {
        head=head->next;
    }
dispose (head);
```

а) пошук елемента;
б) додавання елемента;
в) знищення першого елемента;
г) знищення останнього елемента;
д) знищення множини.

8. За фраґментом програми визначте її призначення

```
struct mmm { int x; mmm *next;} *head, *p1, *p2;
...
p1=(mmm*)malloc(sizeof(mmm));
p1->x=xx;
p1->next=null;
    while(head->next)
    {
        head=head->next;
    }
head->next=p1;
```

а) пошук елемента;
б) додавання елемента в кінець;
в) додавання елемента на початок;
г) знищення першого елемента;
д) знищення останнього елемента.

9. За фраґментом програми визначте його призначення

```
struct mmm { int x; mmm *next;} *head, *p1, *p2;
...
p1=(mmm*)malloc(sizeof(mmm));
p1->x=xx;
p1->next=head;
head=p1;
```

```
while(head->next)
{
    head=head->next;
}
```

а) пошук елемента;
б) додавання елемента в кінець;
в) додавання елемента на початок;
г) знищення першого елемента;
д) знищення останнього елемента.

10. За фраґментом програми визначте її призначення

```
struct mmm { int x; mmm *next;} *head, *p1, *p2;
...
p1=head; dispose(head); head=p1;
```

а) пошук елемента;
б) додавання елемента в кінець;
в) додавання елемента на початок;
г) знищення першого елемента;
д) знищення останнього елемента.

11. Нехай дано опис структури даних:

```
struct mmm { int x; mmm *next;} *head, *p1, *p2;
```

За фраґментом програми визначте структуру, що утворюється за її допомогою:

```
head=(mmm*)malloc(sizeof(mmm));
scanf ("%d", &xx);
head->x=xx;
head->next=null;
p1=head;scanf ("%d", &xx);
while (xx!=0)
    {
        p2=(mmm*)malloc(sizeof(mmm));
        p2->x=xx;
        p2->next=null;
        p1->next=p2; p1=p2;
        scanf ("%d", &xx);
    }
p1->next=head;
```

а) черга;
б) стек;
в) циклічний список;
г) ациклічний список.

12. Нехай дано опис структури даних:

```
struct mmm { int x; mmm *next, *prev;} *head, *p1, *p2;
```

За фраґментом програми визначте структуру, що утворюється за її допомогою:


```
p1=(mmm*)malloc(sizeof(mmm));
scanf ("%d", &xx);
p1->x=xx; p1->prev=null;
scanf ("%d", &xx);
    while (xx!=0)
    {
        p2=(mmm*)malloc(sizeof(mmm));
        p2->x=xx; p2->next=null;
        p2->prev=p1; p1->next=p2;
        p1=p2; scanf ("%d", &xx);
    }
```

а) черга;
б) стек;
в) циклічний список;
г) двозв'язний список.

13. Нехай дано опис структури даних:

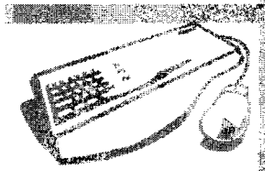
```
struct mmm { int x; mmm *next, *prev;} *head, *p1, *p2, *p3, *p4;
```

За фраґментом програми визначте структуру, що утворюється за її допомогою:

```
p1=(mmm*)malloc(sizeof(mmm)); scanf ("%d", &xx);
p1->x=xx; p1->prev=null; scanf ("%d", &xx);
    while (xx!=0)
    {
        p2=(mmm*)malloc(sizeof(mmm));
        p2->x=xx; p1->next=p2; p2=p4;
            while (xxx!=0)
            {
                p3=(mmm*)malloc(sizeof(mmm));
                p3->x=xxx;
                p3->prev=p2;
                p3->next=null;
                p4->next=p3;
                p4=p3;
                scanf ("%d", &xx);
            }
        p1=p2;
        scanf ("%d", &xx);
    }
```

а) черга;
б) стек;
в) циклічний список;
в) багатозв'язний список,
г) лінійний список,
д) нелінійний список.

РОЗДІЛ 5



ХЕШУВАННЯ ДАНИХ

- *Поняття хеш-функції.*
- *Алгоритми хешування.*
- *Динамічне хешування.*
- *Методи розв'язування колізій.*
- *Переповнення таблиці та рехешування.*
- *Оцінка якості хеш-функції.*

Розділ присвячений описанню методів прискорення доступу до даних у таблицях. Одним із таких методів є використання хеш-функцій. Проте, під час хешування можуть виникати колізії даних та потреба у рехешуванні.

5.1. ПОНЯТТЯ ХЕШ-ФУНКЦІЇ

Для прискорення доступу до даних у таблицях можна використовувати попереднє впорядковування таблиці відповідно до значень ключів.

При цьому можуть бути використані методи пошуку у впорядкованих структурах даних, наприклад, метод половинного розподілу, що істотно скорочує час пошуку даних за значенням ключа. Проте при додаванні нового запису вимагається перевпорядкувати таблицю. Втрати часу на повторне впорядковування таблиці можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних у таблицях використовується так зване **випадкове впорядковування** або **хешування**. При цьому дані організовуються у вигляді таблиці за допомогою хеш-функції h, яка використовується для обчислення адреси за значенням ключа (рис. 5.1).

Ідеальною хеш-функцією є така хеш-функція, яка для будь-яких двох неоднакових ключів повертає неоднакові адреси.

Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів наперед відомі. Така організація даних має назву «досконале хешування». У разі наперед невизначеної множини значень ключів і обмеженої довжини таблиці підбір досконалої функції важко здійснити. Тому часто використовують хеш-функції, які не гарантують виконання умови.

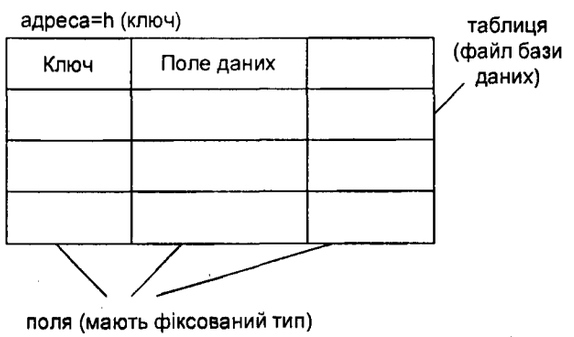


Рис. 5.1. Структура хаш-таблиці.

Приклад 5.1. Розглянемо приклад реалізації недосконалої хеш-функції на мові TurboPascal. Припустимо, що ключ складається із чотирьох символів. При цьому таблиця має діапазон адрес від 0 до 10000.

```
Function hash (key : string[4]): integer;
    var f: longint;
    begin
        f:=ord (key[1]) – ord (key[2]) + ord (key[3]) –ord (key[4]);
        { обчислення функції за значенням ключа}
        f:=f+255*2;
        { поєднання початку області значень функції з початковою
        адресою хеш-таблиці (a=1)}
        f:=(f*10000) div (255*4);
        { поєднання кінця області значень функції з кінцевою адресою
        хеш-таблиці (a=10 000)}
        hash:=f
    end;
```

При заповненні таблиці виникають ситуації, коли для двох неоднакових ключів функція обчислює одну і ту саму адресу. Даний випадок має назву «**колізія**», а такі ключі називаються **ключами-синонімами**.

5.2. АЛГОРИТМИ ХЕШУВАННЯ

Для визначеності вважатимемо, що хеш-функція $h(K)$ має не більше як M різних значень і, що ці значення задовольняють умову

$$0 \le h(K) < M$$

для всіх ключів K.

Теоретично неможливо так визначити хеш-функцію, щоб вона створювала випадкові дані з невипадкових реальних файлів. Але на практиці неважко зробити достатньо хорошу імітацію випадковості, використовуючи прості арифметичні діїРозглянемо, наприклад, випадок десятизначних ключів на десятковому комп'ютері. Сам собою напрошується наступний спосіб вибору хеш-функції: встановити M рівним, скажімо, 1000, а як $h(K)$ узяти три цифри, вибрані приблизно з середини 20-значного добутку $K*K$. Здавалося б, це повинно давати досить рівномірний розподіл значень між 000 і 999 з незначною ймовірністю часткою колізій. Насправді, експерименти з реальними даними показали, що такий метод «**серединних квадратів**» непоганий за умови, що ключі не містять багато лівих або правих нулів підряд.

З'ясувалося, проте, що існують надійніші й простіші способи хеш-функцій.

Метод ділення особливо простий: використовується остача частка від ділення на M

$$h(K)=K \bmod M.$$

У цьому випадку, очевидно, що деякі значення M будуть кращі за інших.Наприклад, якщо M – парне число, то значення $h(K)$ буде парним при парному K, інакше – непарним; часто це приводить до значних зсувів даних. Зовсім погано братп M рівним розрядності машинного слова, оскільки тоді $h(K)$ дає нам праві значущі цифри K (K mod M не залежить від інших цифр). Аналогічно, M не має бути кратним 3, бо буквенні ключі, що

відрізняються один від одного лише регістром, могли б дати значення функції, різниця між якими кратна 3. (Причина криється в тому, що $10n \mod 3 = 4n \mod 3 = 1$.) Взагалі ми хотіли б уникнути значень M, які діляться на $rk \pm a$, де k і a –невеликі числа, а r – «основа системи числення» для множини літер, що використовуються (зазвичай r =64, 256 і 100), оскільки остача від ділення на такі значення M виявляється суперпозицією цифр ключа.

Мультиплікативна схема хешування полягає в заданні послідовність випадкових цілих чисел за формулою

$$X_{i+1} \quad \lambda X_i (\mathrm{mod}\ M).$$

Для машинної реалізації найзручнішим є $M = 2^g$, де g – розрядність машинного слова. Алгоритм подаємо нижче.

1. Вибрати X_0 – довільне непарне число.
2. Визначити коефіцієнт $\lambda = 8t \pm 3$, де t – довільне ціле додатнє число.
3. Знайти добуток λX_0 що містить не більше $2g$ значущих розрядів.
4. Взяти g молодших розрядів в якості X_1.
5. Знайти дріб $x_1 = X_1/2^g$ в інтервалі (0,1).
6. Присвоїти $X_0 = X_1$.
7. Повторити з п. 3.

Хороша хеш-функція повинна задовольняти дві вимоги:

a) її обчислення має бути дуже швидким;

b) вона повинна мінімізувати число колізій.

Властивість (a) частково залежить від особливостей машини, а властивість (b) – від характеру даних. Якби ключі були дійсно випадковими, можна було б просто виділити декілька бітів і використовувати їх для хеш-функції, але на практиці, щоб задовольнити (b), майже завжди потрібна функція, залежна від усіх бітів.

5.3. ДИНАМІЧНЕ ХЕШУВАННЯ

5.3.1. Означення динамічного хешування

Описані вище методи хешування є статичними, тобто спочатку виділяється деяка хеш-таблиця, під її розмір підбираються константи для хеш-функції. На жаль, це не надається для завдань, у яких розмір бази даних часто змінюється. У міру зростання бази даних можна

- ✓ *користуватися початковою хеш-функцією, втрачаючи продуктивність через зростання колізій;*
- ✓ *вибрати хеш-функцію «із запасом», що спричинить невиправдані втрати дискового простору;*
- ✓ *періодично змінювати функцію, перераховувати всі адреси; це забирає дуже багато ресурсів і виводить з ладу базу на деякий час.*

Існує техніка, що дозволяє динамічно змінювати розмір хеш-структури. Це – динамічне хешування. Хеш-функція генерує так званий псевдоключ, який використовується лише частково для доступу до елементу. Іншими словами, генерується досить довга бітова послідовність, яка має бути достатня для адресації всіх потенційно можливих елементів. У той час, як при статичному хешуванні було б треба дуже велику таблицю (яка зазвичай зберігається в оперативній пам'яті для прискорення доступу), тут розмір зайнятої пам'яті прямо пропорційний кількості

елементів в базі даних. Кожен запис в таблиці зберігається не окремо, а в якомусь блоці ("bucket"). Ці блоки збігаються з фізичними блоками на пристрої зберігання даних. Якщо в блоці немає більше місця, щоб вміщати запис, то блок ділиться на два, а на його місце ставиться вказівник на два нові блоки.

Завдання полягає в тому, щоб побудувати бінарне дерево, на кінцях гілок якого були б вказівники на блоки, а навігація здійснювалася б на основі псевдоключа. Вузли дерева можуть бути двох видів: вузли, які показують на інші вузли або вузли, які показують на блоки. Наприклад, нехай вузол має такий вигляд, якщо він показує на блок:

Zero	Null
Bucket	Вказівник
One	Null

Якщо ж він вказуватиме на два інші вузли, то він матиме такий вигляд:

Zero	Адреса a
Bucket	Null
One	Адреса b

Спочатку є тільки вказівник на динамічно виділений порожній блок. При додаванні елемента обчислюється псевдоключ, і його біти по черзі використовуються для визначення місця розташування блоку.

5.3.2. Розширюване хешування

Розширюване хешування близьке до динамічного. Цей метод також передбачає зміну розмірів блоків у міру зростання бази даних, але це компенсується оптимальним використанням місця. Оскільки за один раз розбивається не більш як один блок, накладні витрати досить малі.

Замість бінарного дерева розширюване хешування передбачає список, елементи якого посилаються на блоки. Самі ж елементи адресуються за деякою кількістю i бітів псевдоключа. При пошуку береться i бітів псевдоключа і через список (directory) знаходиться адреса шуканого блоку. Додавання елементів відбувається складніше. Спочатку виконується процедура, аналогічна до пошуку. Якщо блок неповний, додається запис у нього і в базу даних. Якщо блок заповнений, він розбивається на два, записи перерозподіляються за описаним вище алгоритмом. У цьому випадку можливе збільшення числа бітів, необхідних для адресації. Тоді розмір списку подвоюється і кожному новому створеному елементу присвоюється вказівник, який містить його батько. Отже, можлива ситуація, коли декілька елементів показують на один і той самий блок. Зауважимо, що за одну операцію додавання перераховуються значення не більш, ніж одного блоку. Видалення здійснюється за таким самим алгоритмом, тільки навпаки. Блоки, відповідно, можуть бути склеєні, а список – зменшений у два рази.

Отже, основною перевагою розширюваного хешування є висока ефективність, яка не знижується при збільшенні розміру бази даних. Окрім цього, розумно витрачається місце на пристрої зберігання даних, оскільки блоки виділяються тільки під реальні дані, а список вказівників на блоки має розміри, мінімально необхідні для адресації заданої кількості блоків. За ці переваги розробник розплачується додатковим ускладненням програмного коду.

5.3.3. Функції, що зберігають порядок ключі

Існує клас хеш-функцій, які зберігають порядок ключі. Іншими словами, виконується

$$K_1 < K_2 \rightarrow h(K_1) < h(K_2).$$

Ці функції корисні для сортування, яке не потребує додаткової роботи. Іншими словами, ми уникнемо безлічі порівнянь, оскільки для того, щоб відсортувати об'єкти за зростанням, досить просто лінійно просканувати хеш-таблицю.

У принципі, завжди можна створити таку функцію, за умови, щодо хеш-таблиця більша, ніж простір ключів. Проте, завдання пошуку правильної хеш-функції нетривіальне. Зрозуміло, що вона дуже сильно залежить від конкретного завдання. Крім того, подібне обмеження на хеш-функцію може згубно позначитися на її продуктивності. Тому часто вдаються до індексування замість пошуку подібної хеш-функції, якщо тільки заздалегідь не відомо, що операція послідовної вибірки буде частою.

5.3. МЕТОДИ РОЗВ'ЯЗУВАННЯ КОЛІЗІЙ

Для розв'язування колізій використовуються різні методи, які, в основному, зводяться до методів ланцюжків і відкритої адресації (рис. 5.2).

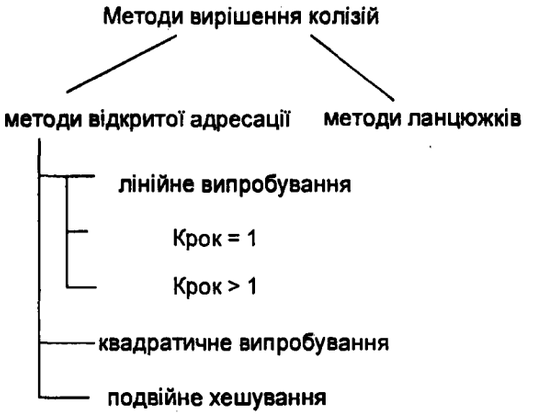


Рис. 5.2. Методи розвязування колізій.

Методом ланцюжків називається метод, в якому для дозволу колізій у всі записи вводяться вказівники, використовувані для організації списків у ланцюжків переповнення. У разі виникнення колізії при заповненні таблиці в список для необхідної адреси хеш-таблиці додається ще один елемент.

Пошук у хеш-таблиці з ланцюжками переповнення здійснюється таким чином. Спочатку обчислюється адреса за значенням ключа. Потім здійснюється послідовний пошук у списку, пов'язаному з обчисленою адресою.

Процедура видалення з таблиці зводиться до пошуку елемента і його видалення з ланцюжка переповнення.

Метод відкритої адресації полягає в тому, щоб, користуючись якимсь алгоритмом, що забезпечує перебір елементів таблиці, проглядати їх у пошуках вільного місця для нового запису.

Лінійне випробування зводиться до послідовного перебору елементів таблиці з деяким фіксованим кроком

a=h(key)+ c*i,

де і – номер спроби дозволити колізію. При кроці, рівному одиниці, відбувається послідовний перебір всіх елементів після поточного.

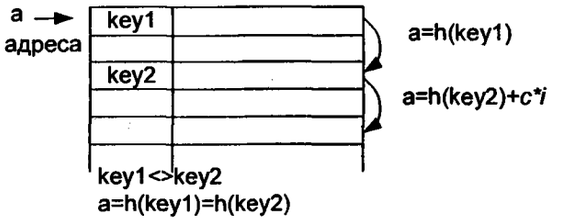


Рис. 5.3. Лінійне випробування.

Квадратичне випробування відрізняється від лінійного тим, що крок перебору елементів не лінійно залежить від номера спроби знайти вільний елемент a = h(key2)+ +c*i + d*i 2.

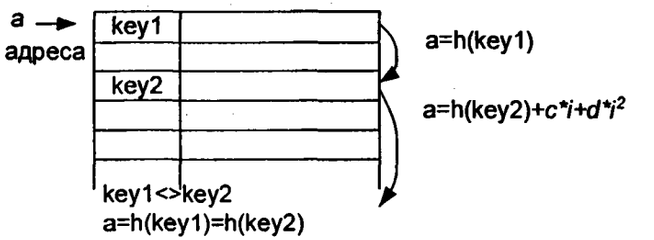


Рис. 5.4. Квадратичне випробування.

Завдяки не лінійності такої адресації зменшується кількість спроб при великій кількості ключів-синонімів.

Проте, навіть відносно невелика кількість спроб може швидко привести до виходу за адресний простір невеликої таблиці внаслідок квадратичної залежності адреси від номера спроби.

Ще один різновид методу відкритої адресації, який називається **подвійним хешуванням,** заснований на нелінійній адресації, що досягається за рахунок підсумовування значень основної і додаткової хеш-функцій a=h1(key)+ i*h2(key).

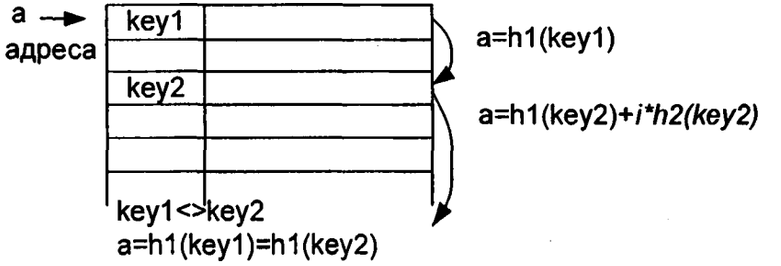


Рис. 5.5. Нелінійне випробування.

Опишемо алгоритми додавання і пошуку для методу лінійне випробування.

Додавання

i = 0
a = h(key)+ i*c
Якщо t(a)= вільно, то t(a)= key, записати елемент, стоп елемент доданий
i = i + 1, перейти до кроку 2

Пошук

i = 0
a = h(key)+ i*c
Якщо t(a)= key, то стоп, елемент знайдений
Якщо t(a)= вільно, то стоп, елемент не знайдений
i = i + 1, перейти до кроку 2

Аналогічним чином можна було б сформулювати алгоритми додавання і пошуку елементів для будь-якої схеми відкритої адресації. Відмінності будуть тільки у виразі, використовуваному для обчислення адреси (крок 2). Із процедурою видалення все відбувається не так просто, оскільки вона у цьому випадку не буде зворотною до процедури додавання.

Річ у тому, що елементи таблиці перебувають у двох станах: "вільно" і "зайнято". Якщо видалити елемент, перевівши його у стан "вільно", то після такого видалення алгоритм пошуку працюватиме некоректно. Припустимо, що ключ елемента, який видаляється, має в таблиці ключі синоніми. У тому випадку, якщо за елементом, що видаляється, в результаті дозволу колізій були розміщені елементи з іншими ключами, то пошук цих елементів після видалення завжди даватиме негативний результат, оскільки алгоритм пошуку зупиняється на першому елементі, що перебуває у стані "вільно".

Скоректувати цю ситуацію можна різними способами. Найпростіший з них полягає в тому, щоб виконувати пошук елемента не до першого вільного місця, а до кінця таблиці. Проте така модифікація алгоритму зведе нанівець весь виграш від прискорення доступу до даних, який досягається в результаті хешування.

Інший спосіб зводиться до того, щоб відслідковувати адреси всіх ключів-синонімів для ключа елементу, що видаляється, і при необхідності розмістити відповідні записи в таблиці. Швидкість пошуку після такої операції не зменшиться, але витрати часу на саме розміщення елементів можуть виявитися дуже значними.

Існує підхід, який не має перерахованих недоліків. Його суть полягає у тому, що для елементів хеш-таблиці додається стан "видалено". Такий стан в процесі пошуку інтерпретується, як "зайнято", а в процесі запису як "вільно".

Сформулюємо алгоритми додавання, пошуку і видалення для хеш-таблиці, що має три стани елементів.

Додавання

i = 0
a = h(key)+ i*c
Якщо t(a)= вільно або t(a)= видалено, то
t(a)= key, записати елемент, стоп елемент доданий
i = i + 1, перейти до кроку 2

Видалення

i = 0
a = h(key)+ i*c
Якщо t(a)= key, то t(a)=знищений стоп, елемент видалений
Якщо t(a)= вільно, то стоп елемент не знайдений
i = i + 1, перейти до кроку 2

Пошук

i = 0
a = h(key)+ i*c
Якщо t(a)= key, то стоп, елемент знайдений
Якщо t(a)= вільно, то стоп елемент не знайдений
i = i + 1, перейти до кроку 2

Алгоритм пошуку для хеш-таблиці, що має три стани, практично не відрізняється від алгоритму пошуку без врахування видалень. Різниця полягає у тому, що при організації самої таблиці необхідно відзначати вільні і видалені елементи. Це можна зробити, зарезервувавши два значення ключового поля. Інший варіант реалізації може передбачати введення додаткового поля, в якому фіксується стан елементf. Довжина такого поля може складати всього два бітb, що цілком достатньо для фіксації одного з трьох станів. На мові TurboPascal таке поле зручно описати типом Byte або Char.

5.4. ПЕРЕПОВНЕННЯ ТАБЛИЦІ І РЕХЕШУВАННЯ

Очевидно, що у міру заповнення хеш-таблиці, виникатимуть колізії і, як результат, їх дозволи методами відкритої адресації, чергова адреса може вийти за межі адресного простору таблиці. Щоб це явище відбувалося рідше, можна піти на збільшення довжини таблиці в порівнянні з діапазоном адрес, який визначається хеш-функцією.

З одного боку це приведе до скорочення кількості колізій і прискорення роботи з хеш-таблицею, а з іншого – до нераціонального витрачання адресного простору. Навіть при збільшенні довжини таблиці в два рази в порівнянні з областю значень хеш-функції, немає гарантії того, що в результаті колізій адреса не перевищить довжину таблиці. При цьому в початковій частині таблиці може залишатися досить багато вільних елементів. Тому на практиці використовують циклічний перехід до початку таблиці.

Розглянемо такий спосіб на прикладі методу лінійного випробування. При обчисленні адреси чергового елемента можна обмежити адресу, взявши як таку залишок від цілочисельного розподілу адреси на довжину таблиці n.

У цьому алгоритмі ми не враховуємо можливість багатократного перевищення адресного простору. Коректнішим буде алгоритм, який використовує зсув адреси на 1 елемент у разі кожного повторного перевищення адресного простору. Це підвищує вірогідність знаходження вільних елементів у разі повторних циклічних переходів до початку таблиці.

Розглянувши можливість виходу за межі адресного простору таблиці, ми не враховували чинники заповненої таблиці і вдалого вибору хеш-функції. При великій заповненій таблиці виникають часті колізії і циклічні переходи в початок таблиці. При невдалому виборі хеш-функції відбуваються аналогічні явища. У якнайгіршому варіанті

при повному заповненні таблиці алгоритми циклічного пошуку вільного місця приведуть до зациклення. Тому при використанні хеш-таблиць необхідно прагнути уникати дуже щільного заповнення таблиць. Зазвичай, довжину таблиці вибирають із розрахунку двократного перевищення передбачуваної максимальної кількості записів. Не завжди при організації хешування можна правильно оцінити необхідну довжину таблиці, тому у разі великої заповненої таблиці може знадобитися рехешування. У цьому випадку збільшують довжину таблиці, змінюють хеш-функцію і переупорядковують дані.

Здійснювати окреме оцінювання густини заповнення таблиці після кожної операції додавання недоцільно, тому можна виконувати таке оцінювання непрямим чином v за кількістю колізій під час одного додавання. Досить визначити деякий поріг кількості колізій, при перевищенні якого треба здійснити рехешування. Крім того, така перевірка унеможливлює зациклення алгоритму в разі повторного перегляду елементів таблиці.

Розглянемо алгоритм додавання, що реалізує пропонований підхід.

Додавання

i = 0
a = ((h(key)+ c*i) div n + (h(key)+ c*i) mod n) mod n
Якщо t(a)= вільно або t(a)= видалено, то
t(a)= key, записати елемент, стоп елемент доданий
Якщо i > m, то стоп потрібен рехешування
i = i + 1, перейти до кроку 2

У цьому алгоритмі номер ітерації порівнюється з пороговим числом *m*. Зазначимо, що алгоритми додавання, пошуку і видалення повинні використовувати ідентичне утворення адреси чергового запису.

Видалення

i = 0
a = ((h(key)+ c*i) div n + (h(key)+ c*i) mod n) mod n
Якщо t(a)= key, то t(a)=знищено, стоп, елемент видалений
Якщо t(a)= вільно або i>m, то стоп, елемент не знайдений
i = i + 1, перейти до кроку 2

Пошук

i = 0
a = ((h(key)+ c*i) div n + (h(key)+ c*i) mod n) mod n
Якщо t(a)= key, то стоп, елемент знайдений
Якщо t(a)= вільно або i>m, то стоп, елемент не знайдений
i = i + 1, перейти до кроку 2

5.5. ОЦІНЮВАННЯ ЯКОСТІ ХЕШ-ФУНКЦІЇ

Як зазначалося, правильний вибір хеш-функції дуже важливий. При вдалій побудові хеш-функції таблиця заповнюється рівномірніше, зменшується кількість колізій і зменшується час виконання операцій пошуку, додавання і видалення. Для того, щоб заздалегідь оцінити якість хеш-функції, можна виконати імітаційне моделювання. Моделювання здійснюється таким чином. Формується цілочисельний масив, довжина

якого співпадає з довжиною хеш-таблиці. Випадково генерується достатньо велика кількість ключів, для кожного ключа обчислюється хеш-функція. В елементах масиву обраховується кількість генерацій заданої адреси. За наслідками такого моделювання можна побудувати графік розподілу значень хеш-функції. Для отримання коректних оцінок кількість ключів, що генеруються, повинна у декілька разів перевищувати довжину таблиці.

Різні підходи до побудови генераторів випадкових чисел дають різні за якістю послідовності. Основні критерії якості – рівномірність, стохастичність, незалежність. Ми розглядатимемо лише перевірку рівномірності.

Перевірка рівномірності ключів $\{x_i\}$ хеш-функції за гістограмою. Відрізок (0,1) розбивається на m рівних частин. Знаходиться відносна частота потрапляння чисел у кожен з інтервалів. Отримуємо гістограму розподілу.

Перевірка рівномірності за непрямими ознаками здійснюється таким чином. Послідовність $\{x_i\}$ розбивається на дві послідовності: парних членів $\{x_{2i}\}$ і непарних членів $\{x_{2i-1}\}$. У загальному випадку, точка (x_{2i-1}, x_{2i}) потрапляє всередину одиничного квадрата, а якщо виконується умова

$$x_{22i-1} + x_{22i} < 1,\ i = 1 \div N,$$

то така точка (x_{2i-1}, x_{2i}) потрапляє всередину четвертини одиничного кола. Після N/2 дослідів, буде згенеровано N чисел. Позначимо кількість пар, що відповідають умові (7) $k \le N/2$. Якщо числа розподілені рівномірно, то відношення кількості попадань в четвертину кола до загальної кількості експериментів прямує до відношення площ цих фігур, тобто $2k/N \rightarrow \pi/4$.

Функція перевірки рівномірності за прямими та непрямими ознаками може бути реалізована так:

```
void rivnom(float m[]) // передаємо масив ключів
{
float p[5000],np[5000]; // парні та непарні елементи
float pp,ll;
long k,r,l,l=0;
double rivno,rz_np,rz_p;
int pop[10]; // масив відрізків для гістограми
// усі значення ключів ділимо на парні та непарні
// за номерами розміщення
for (r=1;r<=count;r++)
    if (r%2==0)
        p[l]=m[r];
    else
    {
        np[l]=m[r];
        l++;
    }
// будуємо гістограму
for (i=0;i<10;i++)
{ // у поточний відрізок ще нема потраплянь
    pop[i]=0;
```

```
    for (r=1;r<=count;r++)
    {
        // розраховуємо початок інтервалу
        ll=(i)/10;
        // якщо є потрапляння, то фіксуємо його
        if ((m[r]>=ll)&&(m[r]<=(ll+0.1))) pop[i]++;
    }
}
// рахуємо кількість потраплянь у коло
k=0; // нема жодного потрапляння
for (r=1;r<=count/2;r++)
{
    rz_np=np[r-1]*np[r-1];
    rz_p=p[r-1]*p[r-1];
    // розраховуємо потрапляння у коло
    rivno=rz_p+rz_np;
    if (rivno<1) k++;
}
// перевірка рівномірності
if ((abs(2*k)/(count)-pi/4)<0.01)
    puts("rivnomirno");
else
    puts("nerivnomirno");
}
```

Резюме

1. Для прискорення доступу до даних у таблицях можна використовувати попереднє впорядковування таблиці відповідно до значень ключів. При цьому дані організовуються у вигляді таблиці за допомогою хеш-функції, б використовується для обчислення адреси за значенням ключа.

2. Є такі алгоритми породження ключів хеш-функцій: серединних квадратів, ділення, мультиплікативний. Ці методи відносяться до статичних методів. У динамічному хешуванні хеш-функція генерує так званий псевдоключ, який використовується лише частково для доступу до елемента.

3. Колізія – ситуація, яка виникає при заповненні таблиці, коли для двох неоднакових ключів функція обчислює одну і ту саму адресу. Ключі з однаковими значеннями називаються ключами-синонімами.

4. Найпоширенішими методами розв'язування колізії є метод ланцюжків та відкритої адресації.

5. Методом ланцюжків називається метод, в якому для дозволу колізій у всі записи вводяться вказівники, використовувані для організації списків у ланцюжках переповнення. У разі виникнення колізії при заповненні таблиці в список для необхідної адреси хеш-таблиці додається ще один елемент.

6. Метод відкритої адресації полягає в тому, щоб, користуючись якимсь алгоритмом, що забезпечує перебір елементів таблиці, проглядати їх у пошуках

вільного місця для нового запису. Бувають такі методи відкритої адресації: лінійне випробування, квадратичне випробування, подвійне хешування.

7. Оцінювання якості хеш-функції здійснюється з метою рівномірнішого заповнення хеш-таблиці та реалізується за допомогою імітаційного моделювання.

Контрольні запитання

1. Визначення хешування та його призначення.

2. Визначити статичні методи породження ключів хеш-функ-цією.

3. Обгрунтувати необхідність використання динамічних мето-дів породження ключів.

4. Перерахувати причини колізій та методи їх вирішення.

5. Перерахувати методи відкритої адресації та порівняти їх між собою.

6. Описати метод ланцюжків.

7. Перерахуйте методи оцінювання якості хеш-функції.

Тести для закріплення матеріалу

1. Хешування – це:

а) випадкове впорядкування ключів;

б) випадкове впорядкування індексів;

в) впорядкування індексів за зростанням;

г) впорядкування індексів за спаданням.

2. За фраґментом функції визначте її призначення.

```
Function mmm (key : string[4]): integer;
var f: longint;
begin
f:=ord (key[1]) – ord (key[2]) + ord (key[3]) –ord (key[4]);
f:=f+255*2;
f:=(f*10000) div (255*4);
mmm:=f end;
```

а) сортування;

б) пошук за ключем;

в) хешування;

г) розіменування.

3. Перерахуйте методи розв'язування колізій:

а) відкритої адресації;

б) закритої адресації;

в) ланцюжків;

г) подвійного хешування;

д) квадратичного випробування;

е) лінійного випробування.

4. За фраґментом програми визначити використаний алгоритм хешування

```
for i:=1 to n do begin
    x:=x*x;
    x:=x*10000;
    x:=x-trunc(x);
    x:=x*100000000;
    x:=trunc(x)/100000000;
    vector[i-1]:=x;
    end;
```

а) серединних квадратів;
б) ділення;
в) мультиплікативний.

5. За фраґментом програми визначити використаний алгоритм хешування

```
g=8;
for  (j=1;j<=n;j++)
{
    dob=x0*y;
    gcvt(dob,20,str); //перетворили число у стрічку
    strncpy(st,str,2*g); // y*x0<2g
    i=strlen(st);
    i=strlen(st); if(i>=g)
    temp=i-g;else temp=0;
    for (int q=temp;q<=temp+g;q++)
        strr[q-temp]=st[q];
    strr[q]='\0';
    x1=atof(strr)/256;
    x0=x1;
    usefl[j-1]=x0;
}
```

а) серединних квадратів;
б) ділення;
в) мультиплікативний.

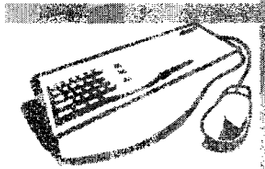
6. Перерахуйте статичні методи ґенерації хеш-ключів:

а) ділення;
б) лінійний;
в) мультиплікативний;
г) закритої адресації;
д) ланцюжків;
е) подвійного хешування;
є) квадратичного випробування.

7. Перевірка якості хеш-фунції здійснюється за допомогою критеріїв:

а) рівномірність;
б) розподіленість;
в) стохастичність;
г) подвійна лінійність;
д) незалежність.

РОЗДІЛ 6



НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ: ДЕРЕВА

- *Поняття дерева.*
- *Поняття бінарного дерева.*
- *Подання дерев у зв'язній пам'яті комп'ютера.*
- *Алгоритми проходження дерев углиб і вшир.*
- *Застосування бінарних дерев в алгоритмах пошуку.*
- *Операція включення в СД типу «бінарне дерево».*
- *Операція виключення з бінарного дерева.*

Розділ присвячений опису дерев та визначенню операцій над ними. Подано приклади роботи з деревами різної арності. Особливу увагу закцентовано на бінарних деревах та організації роботи з ними. Наведено задачі, що розв'язуються з допомогою бінарних дерев.

6.1. ДЕРЕВО

6.1.1. Визначення дерева

✱ ***Дерево*** *– скінченна непорожняТ, що складається з одного й більше вузлів таких, що виконуються наступні умови:*

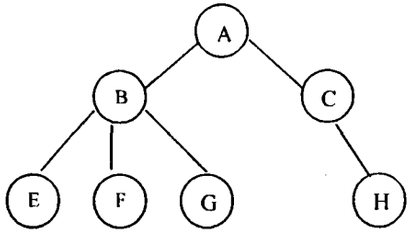
✓ *є один спеціально позначений вузол, який називається коренем дерева;*

✓ *інші вузли (крім кореня) містяться в m>=0 попарно не пересічних множинах* T_1, T_2, ..., T_m, *кожна з яких у свою чергу, є деревом. Дерева* T_1, T_2, ..., T_m *називаються піддеревами такого кореня.*

Дерева зображаються такими способами:

- *графічно,*
- *за допомогою множин,*
- *як модифікація багатозв'язних списків.*

Продемонструємо зображення структури дерева (рис. 6.1):



A: T_1 = {B, E, F, G}
T_2 = {C, H}
B: T_{11} = {E}
T_{12} = {F}
T_{13} = {G}
C: T_{21} = {H}

а) графічний б) за допомогою множин

Рис. 6.1. Способи зображення дерева.

Якщо підмножини T_1, T_2, ..., T_m упорядковані, то дерево називають **упорядкованим**. Якщо два дерева вважаються рівними й тоді, коли вони відрізняються порядком, то такі дерева називаються **орієнтованими деревами**. Кінцева множина непересічних дерев називається **лісом** (рис. 6.2).

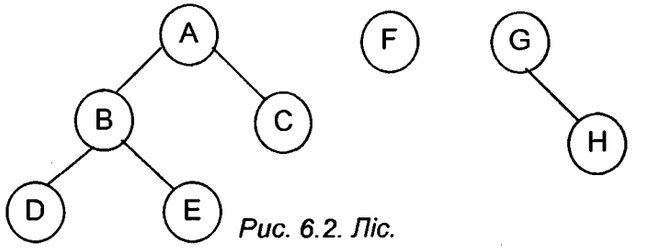
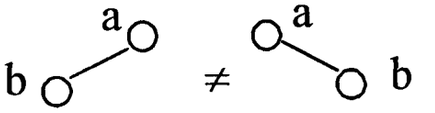


Рис. 6.2. Ліс.

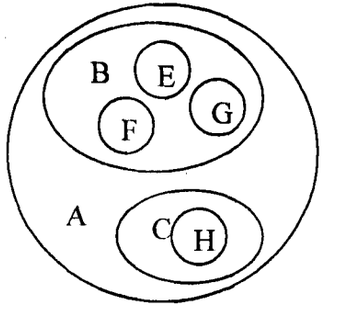
6.1.2. Бінарне дерево

Бінарне дерево – скінченна множина елементів, що може бути порожньою, яка складається з кореня й двох непересічних бінарних дерев, причому піддерева впорядковані: ліве піддерево й праве піддерево.



Кількість підмножин для заданого вузла називається **ступенем вузла**. Якщо така кількість дорівнює нулю, то вузол є **листом**. Максимальний ступінь вузла в дереві – **ступінь дерева**. **Рівень вузла** – довжина шляху від кореня до розглянутого вузла. Максимальний рівень дерева – ***висота дерева***.

Структуру дерева можна зображати й за допомогою способів, поданих на рис. 6.3.



Вкладені множини

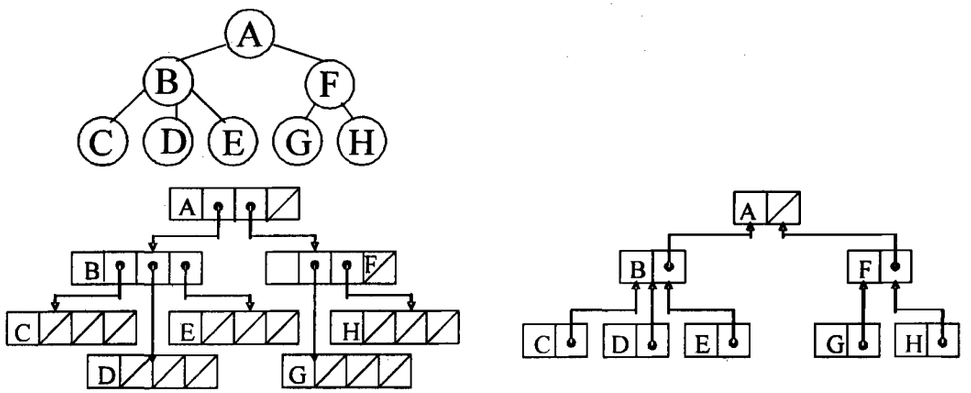
Дужкова форма: (A (B (E) (F) (G)) (C (H)))

Десяткова форма Дьюі: A~1;
B~1.1; C~1.2;
E~1.1.1; F~1.1.2; G~1.1.3; H~1.2.1

Рис. 6.3. Способи подання дерев.

6.1.3. Подання дерев у зв'язній пам'яті комп'ютера

Розрізняють три основні способи подання дерев у зв'язній пам'яті: стандартний, інверсний, змішаний. Розглянемо ці способи для дерева ,зображеного на рис. 6.4.



При стандартному способі вузли, що перебувають на одному рівні, є *братами*. Якщо ж вузол перебуває на нижхому рівні, то він вважається *сином*.

При інверсному способі кожен вузол дерева має вказівник, що вказує на батька.

Рис. 6.4. Стандартний та інверсний способи подання дерев.

Якщо ж говорити про змішаний спосіб подання дерева у зв'язній пам'яті, то тут, як видно з назви, кожний вузол включає вказівники, що вказують як на синів, так і на батька.

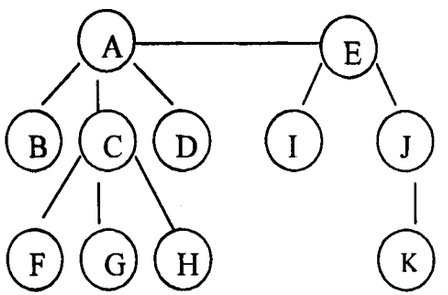
Функція побудови дерева стандартним способом:

```
struct  btree // структура дерева
{
    int el;
    struct btree *l,*r;
}; // вказівники на лівого та правого сина
//вказівники на корінь дерева та поточний елемент
struct btree *root, *cur;
//функція додавання елемента у дерево
struct btree * insert (struct btree *c, int k, struct btree * new1)
{
struct btree *p,*l;  int a;
    if(!c) //поточний елемент є порожнім (відсутнім)
    {
        c=(struct btree*) malloc (sizeof(struct btree));
        c->el=k;
        c->l =NULL;
        c->r =NULL;
        //якщо задано значення a, то дерево вже містить вузли
        // і ми здійснюємо прив'язку створенного елемента
        // як лівого (a=1) або правого (a=2) сина
        if (a==1) new1->l=c;
        if (a==2) new1->r=c;
```



```
        return (c);
    }
    else
    //спускаємося далі по дереву
    {
    new1=c;
        if(c->el>k)
        //переходимо до лівого сина
        {
            c=c->l; a=1;
            p=insert (c,k,new1);
        }
        else
        //переходимо до правого сина
        {
            c=c->r; a=2;
            p=insert (c,k,new1);
        }
    }
}
void main()
//виклик функції побудови дерева
{
int k,n=5,l;
scanf("%d",&k);
root=insert(NULL,k,cur);
    for(i=2;i<=n;i++)
    {
        scanf("%d",&k);
        cur=insert(root,k,cur);
    }
}
```

6.1.4. Алгоритми проходження дерев углиб і вшир



При проходженні вглиб зображеного дерева, список його вершин, записаних у порядку їхнього відвідування, буде виглядати так

A, B, C, F, G, H, D, E, I, J, K.

Алгоритм проходження дерева вглиб

```
<порожній стек S>;
<пройти корінь і включити його в стек S>;
while <стек S не порожній> do
begin
    {нехай P – вузол, що перебуває у вершині стека S}
    if <не всі сини вузла P пройдені>
        then <пройти старшого сина й включити його в стек S>
        else begin
            <виключити з вершини стека вузол P>;
            if <не всі брати вершини P пройдені>
                then <пройти старшого брата й включити його в стек S>
        end;
end;
```

При проходженні зображеного дерева вшир (по рівнях), список його вершин, записаних у порядку їхнього відвідування, буде таким:

A, B, C, D, E, F, G, H, I, J, K.

Алгоритм проходження дерева вширину:

```
<взяти дві черги O1 і O2>;
<помістити корінь у чергу O1>;
while <O1 або O2 не порожня> do
begin
    if <O1 не порожня> then
        {P – вузол, що перебуває в голові черги O1}
    begin
        <виключити вузол із черги O1 і пройти його>;
        <помістити всі вузли, що відносяться до братів цього вузла P, у чергу O2>;
    end
    else <O1=O2; O2=∅
end;
```

6.1.5. Подання дерев у вигляді бінарних

Між деревами загального виду (вузол дерева може мати більше двох синів) і бінарними деревами існує взаємно однозначна відповідність, тому бінарні дерева часто використовують для подання дерев загального виду.

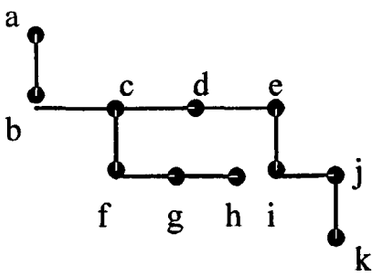
Для такого подання використовують наступний алгоритм:

1) зображуємо корінь дерева;

2) по вертикалі зображуємо старшого сина цього кореня;

3) по горизонталі вправо від цього вузла подаємо всіх його братів;

4) пп. 1, 2, 3 повторюємо для всіх його вузлів.



Отже, вертикальні ребра зображають ліві піддерева, а горизонтальні – праві.

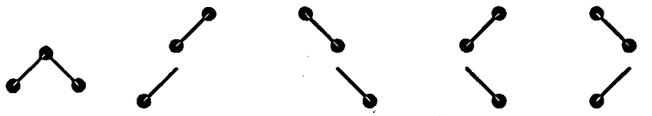
Такий алгоритм можна використати й для лісу.

Теорема. Кількість бінарних дерев з n вершинами можна визначити за формулою:

$$\frac{(2n)!}{(n+1)(n!)^2}.$$

Наприклад, для n=3 маємо $\frac{(2\cdot 3)!}{(3+1)(3!)^2} = \frac{1\cdot 2\cdot 3\cdot 4\cdot 5\cdot 6}{4\cdot 1\cdot 2\cdot 3\cdot 1\cdot 2\cdot 3} = 5.$

Проілюструємо цей приклад графічно:



Подання бінарних дерев у зв'язній пам'яті

При поданні бінарних дерев у зв'язній пам'яті, розрізняють три основних способи подання:

- ✓ *стандартний,*
- ✓ *інверсний,*
- ✓ *змішаний.*

Вузли дерева при кожному такому поданні виглядають так:

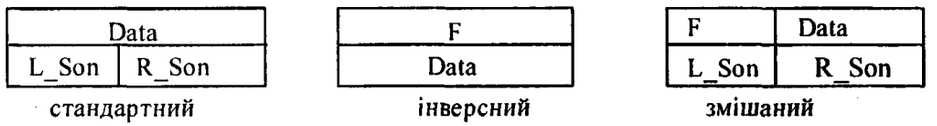
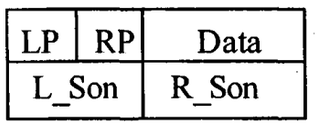


Рис. 6.5. Подання бінарних дерев.

Якщо ми скористаємося прошитим бінарним деревом, то вигляд вузла для нього буде таким:



Один із можливих способів **прошивання** бінарного дерева полягає в тому, що для кожного вузла, що не має лівого піддерева, запам'ятовується посилання на безпосереднього попередника, а для кожного вузла, що не має правого піддерева, запам'ятовується посилання на його безпосереднього спадкоємця. Попередників і спадкоємців для вузла визначають виходячи зі списку, який отримуємо при проходженні розглянутого бінарного дерева в симетричному порядку. Щоб відрізняти посилання на лівого або правого сина (такі посилання називають *структурними зв'язками*) від посилання на попередника або спадкоємця вузла (такі посилання називають "нитками"), використовуються індикаторні поля *LP* і *RP*.

Використання «прошивань» істотно пришвидшує проходження дерева, дозволяє не використовувати стек, але при цьому ускладнюються алгоритми включення / виключення вузла, тому що в прошитому дереві необхідно керувати не тільки структурними зв'язками, але й "нитками".

Прошите бінарне дерево на Паскалі можна подати як:

```
type   TreePtr:=^S_Tree;
       S_Tree = record
            key: KeyType;    { ключ }
       left,right: TreePtr;    { лівий і правий сини }
           LP,RP: boolean;    { характеристики зв'язків }
           end;
```

Якщо *LP* і *RP* рівні FALSE, то зв'язок є ниткою.

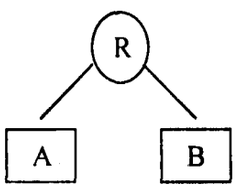
Функція для визначення сина у прошитому дереві змішаним способом:

```
Funtion IsSon (x: TreePtr):TreePtr;
begin
  IsSon:=x^.right;
  if not (x^.rf)  then exit;          {чи це ліва гілка}
  while IsSon^.LP do                  { доки зв'язок не нитка }
      IsSon:=IsSon^.left;     { перейти по лівій гілці }
end;
```

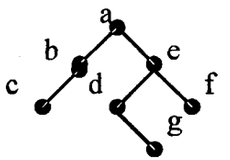
Функція для визначення батька змішаним способом:

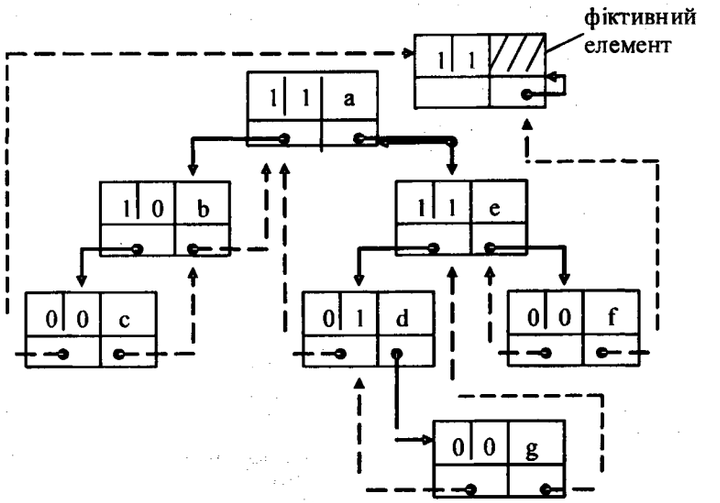
```
Function Parent (x:TreePtr):TreePtr;
begin
  Parent:=x^.left;
  if not (x^.LP)  then exit;
{ доки зв'язок не нитка }
  while Parent ^.rf do  Parent:= Parent ^.right;
end;
```

Симетричним проходженням бінарного дерева є таке проходження, коли ми проходимо спочатку ліве піддерево, потім відвідуємо корінь і останнім відвідуємо праве піддерево: *A R B*.



Приклад 6.1. Розглянемо бінарне дерево. Е пам'яті це дерево буде виглядати так:

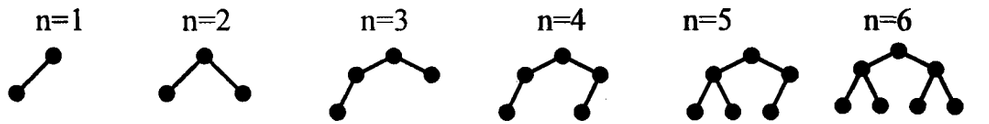




При його симетричному проходженні одержуємо такий список вершин: c, b, a, d, g, e, f.

Формування бінарного дерева

При формуванні бінарного дерева треба задати деякі обмеження. Розглянемо приклад, коли необхідно побудувати дерево з *n* вершин, що має мінімальну висоту. Для досягнення мінімальної висоти необхідно, щоб всі рівні дерева, крім, можливо, останнього, були заповнені. Щоб зробити такий розподіл, всі вершини треба розподіляти порівно: ліворуч і праворуч.



Отже, правила для рівномірного розподілу при відомій кількості вузлів *n* можна сформулювати за допомогою подальшого рекурсивного алгоритму:

- ✓ взяти один вузол як корінь;
- ✓ побудувати ліве піддерево з кількістю вузлів $n_l = [n / 2]$ тим самим способом;
- ✓ побудувати праве піддерево з кількістю вузлів $n_r = n - n_l - 1$ тим самим способом.

Опис вузла дерева мовою Turbo Pascal та додавання вузла виглядає так:

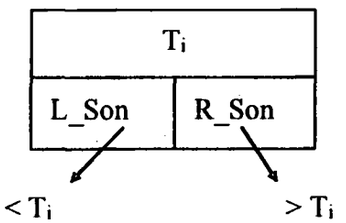
```
Type ElPtr = ^Element
Element = record
Data: integer;
L_Son, R_Son: ElPtr;
end;
Function Tree (n: integer): ElPtr;
var nl, nr, x: integer;
NewElement: ElPtr;
begin
```

```
if n=0 then Tree:=nil else begin
nl:=n div 2;
nr:=n – nl – 1;
read (x); New (NewElement);
with NewElement do begin
Data:=x;
L_Son:=Tree (nl);
R_Son:=Tree (nr);
end;
Tree:=NewElement;
end;
end;
```

Ефективність рекурсивного визначення полягає в тому, що воно дозволяє за допомогою кінцевого висловлення визначити нескінченну множину об'єктів.

6.1.5. Застосування бінарних дерев в алгоритмах пошуку

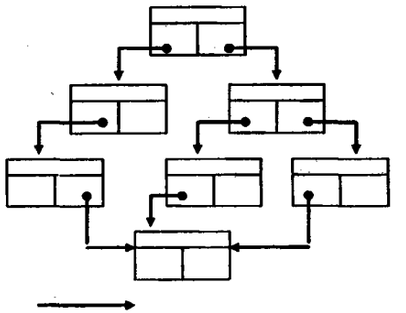
В однозв'язному списку неможливо використати бінарні методи, вони можуть використовуватися тільки в послідовній пам'яті. Однак, якщо використати бінарні дерева, то в такій зв'язній структурі можна одержати алгоритм пошуку зі складністю $O(\log_2 N)$. Таке дерево реалізується в такий спосіб: для будь-якого вузла дерева із ключем T_i всі ключі в лівому піддереві повинні бути менші від T_i, а в правому – більше T_i. У дереві пошуку можна знайти місце кожного ключа, рухаючись, починаючи від кореня й переходячи на ліве або праве піддерево, залежно від значення його ключа. З n елементів можна організувати бінарне дерево (ідеально збалансоване) з висотою не більшою ніж $\log_2 N$, що визначає кількість операцій порівняння при пошуку.




```
Function Search (x: integer; t: ElPtr): ElPtr;
{поле Data замінимо на поле Key}
var f: boolean;
begin
    f:=false;
    while (t<>nil) and not f do
        if x=t^. Key then f:=true
        else
            if x>t^. Key then
                t:=t^. R_Son
            else
                t:=t^. L_Son;
    Search:=t;
end;
```


Якщо одержимо, що значення функції = nil, то ключа зі значенням x у дереві не знайдено.

```
Function Search (x: integer; t: ElPtr): ElPtr;
begin
    S^.key:=x;
    while t^.key<>x do
        if x > t^.key then
            t:=t^. R_Son else t:=t^. L_Son;
    Search:=t;
end;
```



6.1.6. Операція включення в СД типу «бінарне дерево»

Побудова частотного словника

Розглянемо випадок дерева, що постійно зростає. Гарним прикладом цього є побудова частотного словника. У цій задачі задана послідовність слів і треба встановити кількість появи кожного слова. Це означає, що, починаючи з порожнього дерева, кожне слово, що зустрічається в тексті, шукається в ньому. Якщо це слово знайдене, то лічильник появи цього слова збільшується; якщо ж слово не знайдене, то в дерево включається нове слово зі значенням лічильника = 1.

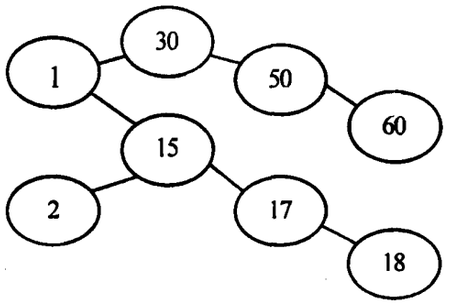
```
Type ElPtr = ^Element
Element = record
Key: integer;
n: integer;
L_Son, R_Son: ElPtr;
end;

Procedure Put (x: integer; var t: ElPtr);
    begin
        if t=nil then
            begin
                New(t);
                with t^ do begin
                key:=x; n:=1;
                L_Son:=nil;
                R_Son:=nil;
            end;
```

```
end
    else
        if x> t^.key then Put(x, t^.R_Son)
        else
            if x<t^.key then Put(x, t^.L_Son)
            else t^.n:=t^.n+1;
end;
```

Хоча задача цього алгоритму – **пошук із включенням,** але його можна використати й для сортування, тому що він нагадує сортування із включенням. А тому, що замість масиву використається дерево, то пропадає необхідність переміщення елементів вище місця включення. Щоб сортування цього алгоритму було стійким, алгоритм повинен виконуватися однаково як при *t^.key=x*, так і при *t^.key>x*.

Нехай задана послідовність: 30, 1, 15, 17, 18, 50, 2, 60. Після побудови дерева здійснюється проходження дерева в симетричному порядку: 1, 2, 15, 17, 18, 30, 50, 60.



Аналіз алгоритму пошуку

Якщо дерево вироджується в послідовність, то складність у такому випадку $O(N)$ (у найгіршому разі). У кращому ж випадку, якщо дерево збалансоване, то складність буде $O(\log_2 N)$. Отже, маємо:

$$O(N) \le П_{ФВС} \le O(\log_2 N).$$

Якщо вважати, що ймовірність появи будь-якої структури дерева однакова, і ключі з'являються з однаковою ймовірністю, то $\lim\limits_{n\to\infty} \dfrac{П_{ФВС}}{O(\log_2 N)} = 1{,}38$.

Алгоритм сортування є стійким, якщо елементи з однаковими ключами з'являються в тій же послідовності при симетричному обході дерева, що й у процесі їхнього включення в дерево.

6.1.7. Операція виключення з бінарного дерева

При видаленні вузла дерево має залишатися впорядкованим щодо ключа:

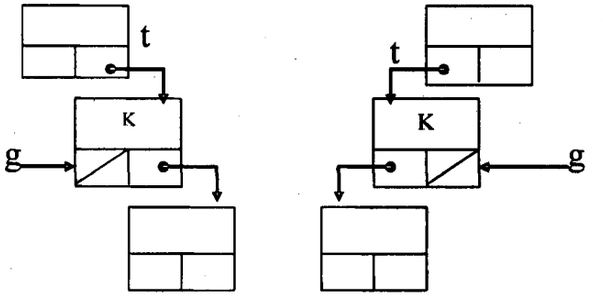
а) видаляється аркуш;

б) видаляється вузол з одним нащадком (нащадок ліворуч або праворуч);

в) видаляється вузол із двома нащадками, але в лівому нащадку немає правого піддерева;

г)загальний випадок.

Розглянемо випадок б):

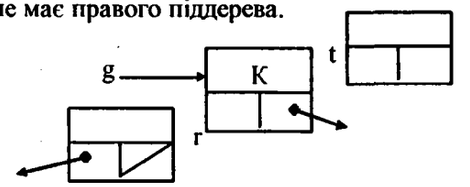



```
if g^.L_Son=nil then begin
g:=t;
t:=t^.R_Son;dispose(g);
end;if g^.R_Son=nil then begin
g:=t;
t:=t^.L_Son;
dispose(g);
end;
```

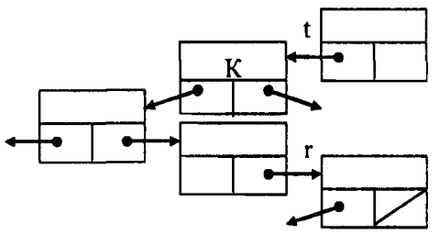
Випадок а) є частковим випадком цього алгоритму.

Випадок в): *r* вказує на елемент, що не має правого піддерева.

```
g:=t;
g^.key:=r^.key;
g:=r;
r:=r^.L_Son;
dispose(g);
```



Випадок г):



Ми шукаємо такий елемент, у якого відсутнє праве піддерево (на такий елемент вказує *r*). Послідовність операторів та сама, тобто видаляє елемент, який треба замінити найбільш правим елементом його лівого піддерева. Такі елементи не можуть мати більше одного нащадка.

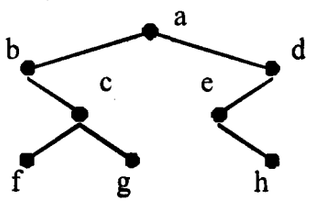
```
Procedure Get (x: integer; var t: ElPtr);
var g: ElPtr;
begin
if t=nil then
{опрацювання виняткової ситуації, коли елемента із заданим ключем у дереві
немає}
else if x>t^.key then Get(x, t^.R_Son)
else if x<t^.key then Get(x, t^.L_Son)
else begin
g:=t;
```

```
if g^.L_Son=nil then t:=g^.R_Son
else if g^.R_Son=nil then t:=g^.R_Son
else D(g^.L_Son); {знайти r)
dispose(g);
end;
end;
```

Подання бінарного дерева в прямокутній пам'яті

При поданні дерева в прямокутній пам'яті одне з полів R_Son або L_Son видаляється, тому що необхідність у ньому відпадає. Нащадок може розміщуватися поруч, тобто в безпосередній близькості. Щоб довідатися, чи є нащадок, уводиться додаткове поле L або R. Нехай, наприклад, відсутній лівий нащадок. Уведемо наступні змінні:

```
index = 0..max;
type Element = record
R_Son: index;
Data: BaseType;
L: boolean;
end;
var Tree = array [index] of Element;
```



a	b	c	f	g	D	e	h
T	F	T	F	F			
1	2	3	4	5	6	7	8

R_Son
Data
L

Порядок називається **прямим**, якщо відсутнє ліве піддерево. Якщо ж відсутнє праве піддерево, то такий порядок називається **фамільним**.

6.1.8. Застосування бінарних дерев

1. **Дерево Хафмана** – двійкове дерево, листям якого є символи алфавіту, в кожній вершині якого зберігається частота символу (для листа) або сума частот його двох нащадків (називатимемо це число вагою вершини). При цьому виконується властивість: якщо відстань від кореня до вершини *X* більша, ніж до вершини *Y*, то вага *X* не перевищує вагу *Y*. Один із способів застосування дерев Хафмана – *алгоритми кодування (архівування) інформації* (детальніше див. розділ 10.3.3).

Код змінної довжини символа (змінний код у дереві Хафмана)– послідовність бітів, яку отримуємо при проходженні по ребрах від кореня до вершини із цим символом, якщо зіставити кожному ребру значення 1 або 0. У дереві Хафмана ребрам, що виходять із вершини до її нащадків присвоюються різні значення (вважатимемо далі, що ребро з 1 – ліве, а з 0 – праве).

Статичний метод Хафмана передбачає, що частоти символів алфавіту наперед відомі. Для цього, перш ніж почати стискання файла, програма проходить файлом і підраховує, який символ скільки разів зустрічається. Потім, відповідно до ймовірності

появи символів, будується двійкове дерево Хафмана, з нього отримуються відповідні кожному символу коди змінної довжини. Нарешті, знову здійснюється проходження початковим файлом, при цьому кожен символ замінюється на свій код у дереві. Отже, статичному алгоритму потрібні два проходи по файлу-джерелу, щоб закодувати дані.

Статичний алгоритм:

```
{ Ініціалізуємо двійкове дерево}
InitTree;
For i:=0 to Length(CharCount)
{ Ініціалізуємо масив частот елементів алфавіту}
    CharCount[i]:=0;
{Запускаємо перше проходження файлу}
While not EOF(Файл-джерело)
Begin
    {Читаємо активний символ}
    Read(Файл-джерело, C);
    {Збільшуємо частоту його появи}
    CharCount[C]= CharCount[C]+1;
End;
{За отриманими частотами будуємо двійкове дерево}
ReBuildTree;
WriteTree (Файл-приймач)     {Записуємо у файл двійкове дерево}
{Запускаємо друге проходження файлу}
While not EOF(Файл-джерело)
Begin
    Read(Файл-джерело, C);  {Читаємо поточний символ}
    {Запускаємо другий прохід файлу}
    code=FindCodeInTree(C);
    write( Файл-приймач, code); {Записуємо послідовність бітів у файл}
End
```

Процедура InitTree ініціалізує двійкове дерево, онулюючи значення, ваги, і вказівники на батька і нащадків активного листка. Далі виконується перший прохід по джерелу даних з метою перевірки частотності символів, результати якого запам'ятовуються у масив CharCount. Розмірність цього масиву повинна дорівнювати кількості елементів алфавіту джерела. У загальному випадку, для стискання заданого комп'ютерного файлу його довжина повинна становити 256 елементів. Потім, відповідно до отриманого набору частот ,будуємо двійкове дерево ReBuildTree. Під час другого проходу перекодуємо джерело відповідно до дерева і записуємо одержані послідовності бітів у стиснутий файл. Для того, щоб мати можливість відновити стиснуті дані, необхідно в отриманий файл зберегти копію двійкового дерева WriteTree.

Динамічний алгоритм:

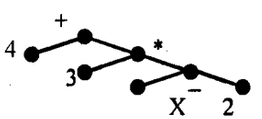
```
{ Ініціалізуємо двійкове дерево}
InitTree;
For i:=0 to Length(CharCount)
{ Ініціалізуємо масив частот елементів алфавіту}
    CharCount[i]:=0;
```

```
{ Запускаємо проходження файлу}
While not EOF(Файл-джерело)
Begin
    Read(Файл-джерело, C);  {Читаємо активний символ}
    If C немає в дереві then  {Перевіряємо чи зустрічався символ раніше}
        Code:=Код "порожнього" символу & Asc(C)
    {Якщо ні, то запам'ятовуємо код порожнього листка і ASCII код активного
символу}
    { Інакше запам'ятовуємо код активного символу}
    else
        code:=Код C;
    {Записуємо послідовність бітів у файл}
    write( Файл-приймач, code);
    { Оновлюємо двійкове дерево символом}
    ReBuildTree(C);
End;
```

Динамічний алгоритм дозволяє реалізувати однопрохідну модель стискання. Не знаючи реальної ймовірності появи символів у початковому файлі, програма поступово змінює двійкове дерево, з кожним символом, що зустрічається, збільшуючи частоту його появи е дереві та перебудовуючи зв'язки у самому дереві. Проте, стає очевидним, що, вигравши в кількості проходжень початковим файлом, ми втрачаємо у стисканні, оскільки у статичному алгоритмі реальні частоти символів були відомі із самого початку і довжини кодів цих символів ближчі до оптимальних, тоді як динамічний метод, вивчаючи джерело, поступово доходить до його реальних частотних характеристик. Але, оскільки динамічне двійкове дерево постійно модифікується новими символами, немає необхідності запам'ятовувати їх частоти заздалегідь – при розархівуванні програма, отримавши з архіву код символа, так само відновить дерево, як вона це робила при стисканні, і збільшить на одиницю частоту символа.

2. Будь-який *алгебраїчний* вираз містить змінні, числа, знаки операцій і дужки можна подавати *у вигляді бінарного дерева* (виходячи з бінарності цих операцій). При цьому знак операції міститься в корені, перший операнд – у лівому піддереві, а другий операнд – у правому. Дужки при цьому опускаються. У результаті всі числа й змінні виявляться в листках, а знаки операцій – у внутрішніх вузлах.

Розглянемо приклад: 3 (x – 2) + 4.



Звичною формою виразів є **інфіксна**, коли знак бінарної операції записується між позначеннями операндів цієї операції, наприклад, x-2. Розглянемо запис знаків операцій після позначень операндів, тобто **постфіксний** запис, наприклад, x 2 -. Такий запис має також назву **зворотного польського,** оскільки його запропонував польський логік Ян Лукасевич.

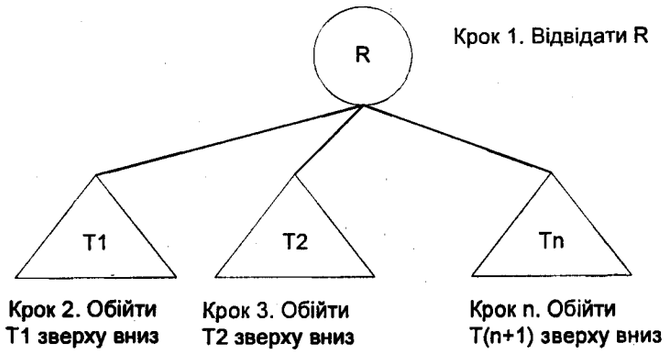
Сформулюємо правило обчислення значень виразу у інфіксному записі: вираз проглядається справа наліво, виділяються перші 2 операнди перед операцією, ця операція виконується, і її результат – це новий операнд.

Правило обчислення значення виразу в зворотному польському записі: вираз проглядається зліва направо, виділяються перші 2 операнди перед операцією, ця операція виконується, і її результат – це новий операнд.

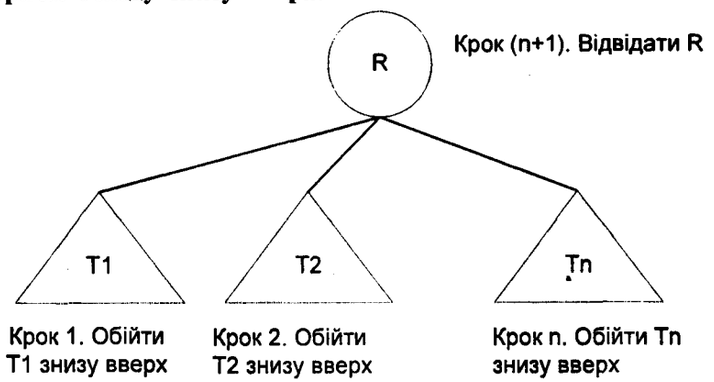
Запис виразу в інфіксній формі: 4 + 3 * x – 2.

Запис виразу в зворотному польському записі: 4 3 x 2 - * +.

Алгоритм обходу зверху вниз:



Алгоритм обходу знизу вверх:



3. Класична програма із класу інтелектуальних: **побудова дерев рішень.** У цьому випадку не листкові (внутрішні) вузли містять предикати – запитання, відповіді на які можуть приймати значення "так" або "ні". На рівні листків перебувають об'єкти (альтернативи, за допомогою яких розпізнаються програми). Користувач одержує запитання, починаючи з кореня, і, залежно від відповіді, спускається або на ліве піддерево, або на праве. Таким чином він виходить на той об'єкт, що відповідає сукупності відповідей на зппитання.

6.2. ВИДИ БІНАРНИХ ДЕРЕВ

6.2.1. Збалансоване дерево

У програмуванні **збалансоване дерево** в загальному розумінні цього слова – це такий різновид бінарного дерева пошуку, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем, мінімальною. Ця властивість є важливою тому, що час виконання більшості алгоритмів на бінарних деревах пошуку пропорційний до їхньої висоти, і звичайні бінарні дерева пошуку можуть мати досить велику висоту в тривіальних ситуаціях. Процедура зменшення (балансування) висоти дерева виконується за допомогою трансформацій, відомих як обернення дерева, в певні моменти часу (переважно при видаленні або додаванні нових елементів).

Більш строге визначення збалансованих дерев було дане Г.Адельсон-Вельським та Є.Ландісом. *Ідеально збалансованим деревом* за Адельсон-Вельським та Ландісом є таке, у якого для кожної вершини різниця між висотами лівого та правого піддерев не перевищує одиниці. Однак, така умова доволі складна для виконання на практиці і може вимагати значної перебудови дерева при додаванні або видаленні елементів.

Тому було запропоноване менш строге визначення, яке отримало назву умови *АВЛ(AVL)-збалансованості* і говорить, що бінарне дерево є збалансованим, якщо висоти лівого та правого піддерев різняться не більше ніж на одиницю. Дерева, що задовольняють такі умови, називаються AVL-деревами. Зрозуміло, що кожне ідеально збалансоване дерево є також АВЛ-збалансованим, але не навпаки.

Наведемо програму додавання та знищення елемента у збалансованому дереві.

```
Type node = record      {запис для визначення дерева}
    Key: integer;
    Left, right: ref;
    Bal: -1..1;         {показує різницю висоти гілок дерева}
End;
procedure search(x: integer; var p: ref; var h: boolean);
var p1, p2: ref; {h = false}
begin
if p = nil then
    begin               {вузла немає у дереві; включити його}
        new(p);
        h := true;
        with p^ do
        begin
            key := x;
            count := 1;
            left := nil;
            right := nil;
            bal := 0;
        end
    end
else
    if x < p^.key then
    begin
        search(x, p^.left, h);
        if h then           {виросла ліва гілка}
            case p^.bal of
                1: begin p^.bal := 0; h := false end ;
                0: p^.bal := -1;
                -1: begin {балансування}
                p1:= p^.left;
        if p1^.bal = -1 then
        {однократний правий поворот}
            begin
```



```
                p^.left := p1^.right;
                p1^.right := p;
                p^.bal := 0;
                p := p1
            end
        else
        {двократний правий – лівий поворот}
            begin
                p2 := p1^.right;
                p1^.right := p2^.left;
                p2^.left :=p1;
                p^.left := p2^.right;
                p2^. Right := p;
                if p2^.bal =-1 then p^.bal:=1 else p^.bal := 0;
                it p2^.bal = 1 then p1^.bal := -1 else p1.^bal :=0;
                p :=p2;
            end;
        p^.bal := 0;
        h := false;
    end
end
end
    else
        if x > p^.key then
        begin
            search(x, p^.right, h);
            if h then       {виросла права гілка}
                case p^.bal of
                -1: begin p^.bal := 0; h :=false; end ;
                0: p^.bal := 1;
                1: begin                    {балансування}
                    p1 := p^.right;
                    if p1^.bal =1 then
                        begin               {однократний лівий поворот}
                            p^.right := p1^.left;
                            p1^.left := p;
                            p^.bal := 0;
                            p := p1;
                        end
            else
                {двократний лівий – правий поворот}
                begin
                    p2 := p1^.left;
                    p1^.left := p2^.right;
                    p2^.right := p1;
```

```
                p^.right := p2^.left;
                p2^.left := p;
                if p2^.bal = 1 then p^.bal := -1
                else p^.bal:=0;
                if p2^.bal = -1 then p1^.bal := 1
                else p1^.bal: = 0;
                p :=p2;
            end ;
        p^.bal := 0; h:= false;
    end
end
    else
        begin
            p^.count := p^.count + 1; h :=false;
        end
end.
```

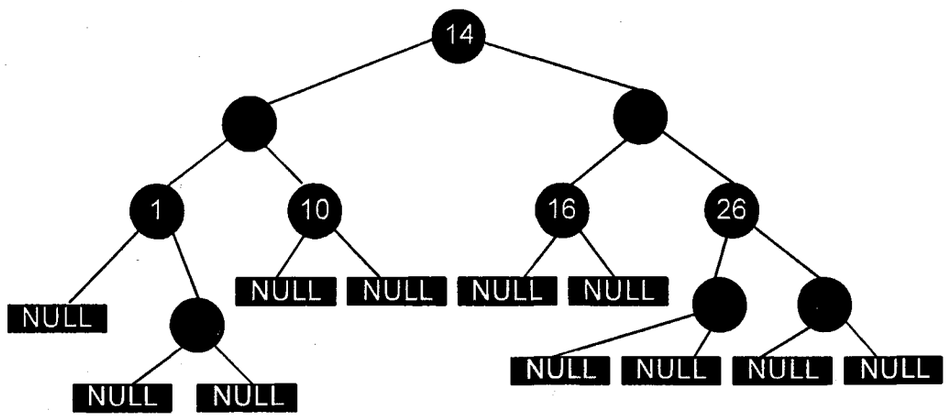
6.2.2. Червоно-чорне дерево

Червоно-чорне дерево (англ. Red-black tree, RB tree) –різновид бінарного дерева пошуку, вершини якого мають додаткові властивості (RB-властивості), зокрема «колір» (червоний або чорний). Точнішим є таке визначення: червоно-чорні дерева – різновид збалансованих дерев, в яких за допомогою спеціальних трансформацій гарантується, що висота дерева h не буде перевищувати $\Theta(\log n)$. Зважаючи на те, що час виконання основних операцій на бінарних деревах (пошук, видалення, додавання елементу) є $\Theta(h)$, ці структури даних на практиці є набагато ефективнішими, аніж звичайні бінарні дерева пошуку.

RB-властивості

Бінарне дерево називається червоно-чорним, якщо воно має наступні властивості:

1) кожна вершина або червона, або чорна,
2) корінь дерева – чорний,
3) кожний листок (NULL) – чорний,
4) якщо вершина червона, обидві його нащадки чорні,
5) усі шляхи від кореня до листків, мають однакову кількість чорних вершин.



Такі властивості надають червоно-чорному дереву додаткового обмеження: найдовший шлях із кореня до будь-якого листка перевищує найкоротший шлях не більше ніж вдвічі. У цьому сенсі таке дерево можна назвати збалансованим. Зважаючи на те, що час виконання основних операцій з бінарними деревами пошуку залежить від висоти, таке обмеження гарантує їхню ефективність в найгіршому випадку, чого звичайні бінарні дерева гарантувати не можуть.

Для того, щоби зрозуміти, чому перелічені властивості забезпечують існування такого обмеження, зазначимо, що в червоно-чорному дереві, відповідно до властивості 4, не існує такого шляху, на якому б зустрілися дві червоні вершини підряд. Найкоротший шлях складається з усіх чорних вершин, а в найдовшому червоні та чорні вершини чергуються. З врахуванням властивості 5, отримуємо, що глибина будь-яких двох листів відрізняється не більше ніж в два рази.

У деяких зображеннях червоно-чорних дерев, NULL-листки не наводяться, тому що вони не містять корисної інформації, але їхнє існування необхідне для забезпечення усіх властивостей.

Основні операції

Операції, які не пов'язані з модифікацією RB-дерева, не потребують коректив і повністю аналогічні до відповідних операцій для звичайних бінарних дерев пошуку. Разом з тим, додавання або видалення елемента з червоно-чорного дерева може призвести до порушення RB-властивостей. Відновлення цих властивостей після модифікації дерева потребує порівняно невеликої кількості ($\Theta(\log n)$) операцій зі зміни кольору вершин та не більше як трьох операцій повороту (дві при доданні елемента). Це залишає часові параметри операцій додавання та видалення в межах ($\log n$), але ускладнює відповідні алгоритми.

Додавання елемента

Процедура починається аналогічно до додавання елемента в бінарне дерево, пошуку та фарбування її у червоний колір. Подальші дії залежать від кольорів сусідніх вершин. Зазначимо, що:

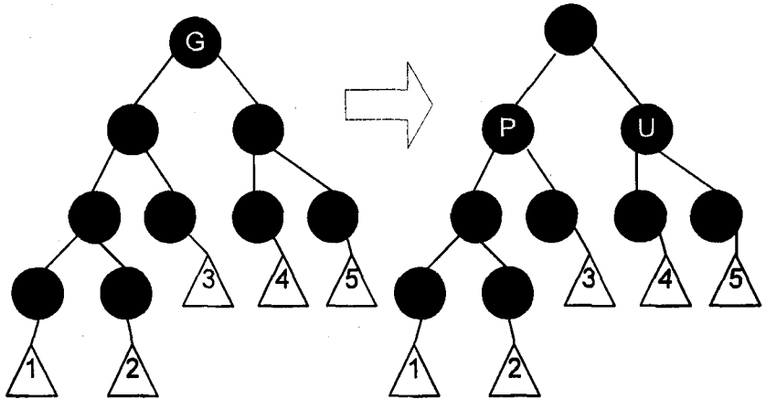
- ✓ властивість 2 завжди зберігається,
- ✓ властивість 3 порушується тільки при додаванні червоної вершини, перефарбуванні чорної вершини в червону або обертанні,
- ✓ властивість 4 порушується тільки при додаванні чорної вершини, перефарбуванні червоної вершини в чорну або обертанні.

На допоміжних діаграмах, вершина, яка додається, позначена *N*, первісний батько цієї вершини позначений *P*, батько вершини *P* («дідусь» *N*) позначений *G*. «Дядько» N (тобто вершина, яка має спільного з *P* батька – *G*) позначений як *U*. Розглянемо подані випадки.

Випадок 1. Нова вершина розташована в корені дерева. У такому випадку необхідно пофарбувати її в чорний колір для забезпечення властивості 1. Очевидно, що властивість 5 при цьому залишається справедливою.

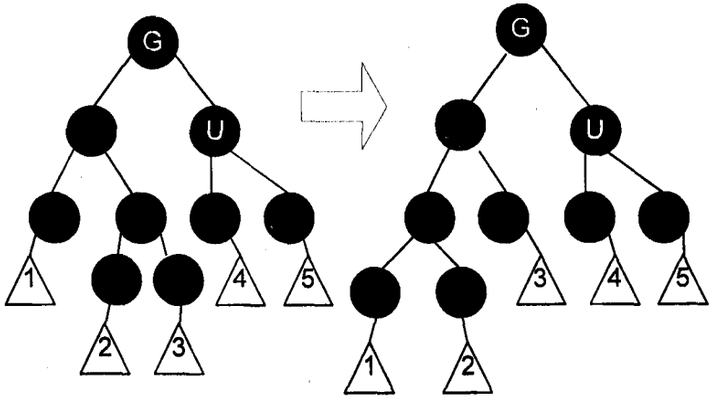
Випадок 2. Батько нової вершини є чорним. Властивість 3 не порушена. У цьому випадку дерево є коректним. Властивість 5 також зберігається, адже червона вершина додається на місце чорного листка і це не змінює кількості чорних вершин на цьому шляху.

Випадок 3. Батько та дядько доданої вершини є червоними. Тоді ми можемо перефарбувати їх обох в чорний колір, а також перефарбувати дідуся в червоний. Тепер наша червона вершина має чорного батька. Завдяки тому, що будь-який шлях через батька чи дядька повинен проходити і через дідуся, кількість чорних вершин на шляху залишається незмінною. Однак батько дідуся (тобто прадідусь) може бути червоною вершиною, як тепер і дідусь. Якщо це так, слід повторити цю операцію рекурсивно.



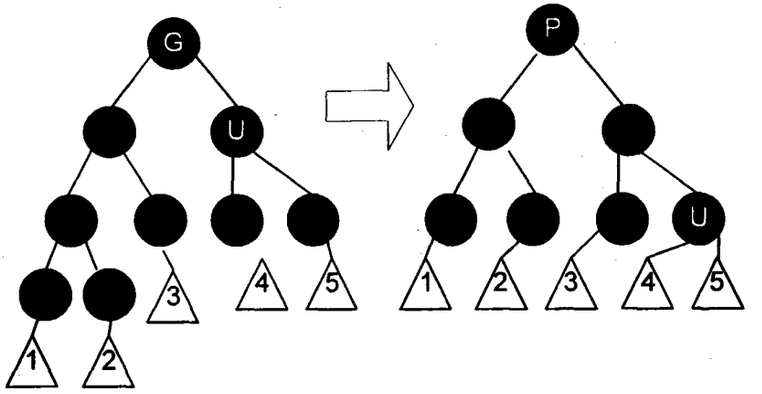
Зауваження: У наступних випадках ми припускаємо, що вершина P є лівим нащадком G. Якщо P – правий нащадок, ліву та праву вершини в аналізі поданих далі випадків треба поміняти місцями.

Випадок 4. Батько є червоним, але дядько – чорний. До того ж, нова вершина – правий нащадок свого батька, і батько, в свою чергу, лівий нащадок свого батька. У такому випадку, ми можемо виконати лівий поворот, внаслідок якого нова вершина та її батько поміняються ролями. Подальші дії з колишнім батьком виконуються відповідно до випадку 5. Зважаючи на те, що нова вершина та її батько є червоними, операція повороту не змінює умови 4.



Випадок 5. Батько є червоним, але дядько чорний. До того ж, нова вершина – лівий нащадок свого батька, і батько є лівим нащадком свого батька. У такому випадку ми виконуємо правий поворот навколо дідуся. Як результат колишній батько тепер є батьком і нової вершини, і свого бувшого дідуся. Ми знаємо, що бувший дідусь є чорним,

інакше батько не був би червоним. Тепер треба поміняти місцями кольори бувшого батька та дідуся, і тепер для дерева виконується властивість 3. Як бачимо, властивість 4 також залишається незмінною.



Функція додавання вершини у червоно-чорне дерево:

```
Type link=^l;
l=record {структура для описування червоно-чорного дерева}
    key:integer; {значення ключа}
    red:boolean; {якщо вершина червона, true, інакше false}
    l ,r:link; {лівий та правий сини}
end;
Var head, w: link;
k: integer;
st1: string;

Function rotate(v:integer;y:link):link; {функція повороту}
    Var c,gs:link;
    begin
        {обираємо напрямок руху}
        if v<y^.key then c:=y^.l else c:=y^.r;
        if v<c^.key then
        begin
            {здійснюємо обмін вершинами }
            gs:=c^.l; c^.l:=gs^.r; gs^.r:=c;
        end else
        begin
            gs:=c^.r; c^.r:=gs^.l; gs^.l:=c;
        end;
        if v<y^.key then y^.l:=gs else y^.r:=gs;
        rotate:=gs;
    end;
```

```
Function split(v : integer; m,p,g,gg : link):link;
{перефарбування вершини}
   begin
      m^.red:=true; m^.l^.red:=false; m^.r^.red:=false;
      if p^.red then
         begin
            g^.red:=true;
            if (v < g^.key) <> (v < p^.key) then
            p:=rotate(v,g);
            m:=rotate(v,gg); m^.red:=false;
         end;
      head^.r^.red:=false;
      split:=m;
   end;

Function insert(v : rec; n : link):link;
   {Ідодавання вершини}
   Var q1,q2,p : link;
   begin
      p:=n; q1:=p;
      repeat
      q2:=q1; q1:=p;
      p:=n;
      if v<n^.key then n:=n^.l Else n:=n^.r;
      if n^.l^.red And n^.r^.red then n:=split(v,n,p,q1,q2);
      until n^.l=n;
      New(n);
      n^.key.fio:=v.fio;
      n^.l:=n;
      n^.r:=n;
      if v<p^.key then p^.l:=n else p^.r:=n;
      insert:=n;
      n:=split(v,n,p,q1,q2);
   End;
```

Резюме

1. Дерево – скінчена непорожня множина T, що складається з одного й більше вузлів таких, що виконуються наступні умови: є один спеціально позначений вузол, який називається коренем дерева; інші вузли (крім кореня) містяться в m>=0 попарно не пересічних множинах T_1, T_2, ..., T_m, кожне з яких, у свою чергу, є деревом. Дерева T_1, T_2, ..., T_m називаються піддеревами цього кореня.

2. Кінцева множина непересічних дерев називається лісом.

3. Бінарне дерево – кінцева множина елементів, які можуть бути порожніми, що складається з кореня й двох бінарних дерев, які не перетинаються,

причому піддерева впорядковані: ліве піддерево й праве піддерево.

4. Розрізняють три основні способи подання дерев у зв'язній пам'яті: стандартний, інверсний, змішаний.

5. При проходженні дерева вшир для занесення відвіданих використовують чергу, в довжину – стек.

6. Прошивання бінарного дерева полягає в тому, що для кожного вузла, що не має лівого піддерева, запам'ятовується посилання на безпосереднього попередника, а для кожного вузла, що не має правого піддерева, запам'ятовується посилання на його безпосереднього спадкоємця.

7. Найяскравішими областями використання дерев є архівування даних за допомогою дерев Хафмана, розбір виразу та задачі прийняття рішень.

8. Збалансоване дерево – це такий різновид бінарного дерева пошуку, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем, мінімальною.

9. Червоно-чорні дерева – різновид збалансованих дерев, в яких за допомогою спеціальних трансформацій гарантується, що висота дерева h не буде перевищувати Θ (log n).

Контрольні запитання

1. Перерахуйте складові дерева.
2. Дайте визначення бінарного дерева.
3. Операції над деревами.
4. Використання бінарних дерев для пошуку даних.
5. Червоно-чорні дерева. Основні властивості.
6. Поняття збалансованого дерева. Види збалансованих дерев.
7. Побудувати бінарне дерево для виразу 12-2(3+4).

Тести для закріплення матеріалу

1. Перерахуйте складові означення дерева.

а) Є один спеціально позначений вузол, який називається коренем заданого дерева.

б) Є декілька спеціально позначених вузлів, які називаються коренями заданого дерева.

в) Інші вузли (крім кореня) містяться в m>=0 попарно не пересічних множинах T_1, T_2, ..., T_m, кожна з яких, у свою чергу, є деревом. Дерева T_1, T_2, ..., T_m називаються піддеревами заданого кореня.

г) Інші вузли (і корінь також) містяться в m>=0 попарно не пересічних множинах T_1, T_2, ..., T_m, кожна з яких, у свою чергу, є деревом. Дерева T_1, T_2, ..., T_m називаються піддеревами заданого кореня.

2. Перерахуйте алгоритми проходження дерева:

а) вшир;

б) по діагоналі;

в) вглиб;

г) у висоту.

3. Корінь дерева – це:
а) вузол, з якого немаж жодної дуги;
б) вузол, який має нащадків і не має предків;
в) вузол, який має пердків і не має нащадків;
г) вузол, рівень якого рівний одиниці;
д) вузол, ступінь якого рівний нулеві.

4. Листок дерева – це:
а) вузол, з якого немає жодної дуги;
б) вузол, який має нащадків і не має предків;
в) вузол, який має пердків і не має нащадків;
г) вузол, рівень якого рівний одиниці;
д) вузол, ступінь якого рівний нулеві.

5. Види подання бінарних дерев:
а) прошиті дерева;
б) перелічені дерева;
в) прямокутне подання;
г) чотирикутне подання.

6. За визначенням знайдіть означення: вузли, що перебувають на одному рівні є братами. Якщо ж вузол перебуває на нижхому рівні, то він вважається сином:
а) стандартний спосіб подання дерев у зв'язній пам'яті;
б) інверсний спосіб подання дерев у зв'язній пам'яті;
в) обхід графа вшир;
г) обхід графа вглиб.

7. Обхід дерева вглиб використовує:
а) чергу;
б) стек;
в) список;
г) таблицю.

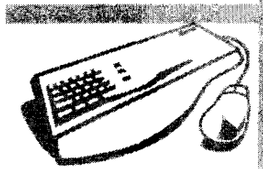
8. Обхід дерева вшир використовує:
а) чергу;
б) стек;
в) список;
г) таблицю.

9. За визначенням знайдіть означення: кожен вузол дерева має вказівник, що вказує на батька:
а) стандартний спосіб подання дерев у зв'язній пам'яті;
б) інверсний спосіб подання дерев у зв'язній пам'яті;
в) обхід графа вшир;
г) обхід графа вглиб.

10. Види збалансованих дерев:
а) чорно-білі дерева;
б) червоно-чорне дерево;
в) AVL-дерево;
г) ВС-дерево.

11. Перерахувати властивості RB-дерев:
а) кожна вершина або червона, або чорна;
б) корінь дерева – чорний;
в) корінь дерева – червоний;
г) кожний листок– чорний;
д) кожний листок червоний;
е) якщо вершина червона, обидваї нащадки чорні;
є) усі шляхи від кореня до листів, мають однакову кількість чорних вершин.

РОЗДІЛ 7



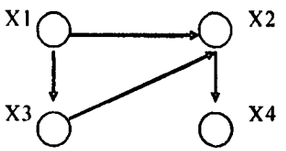
НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ: ГРАФ

- *Поняття графу.*
- *Подання графу в пам'яті комп'ютера.*
- *Алгоритми проходження графу.*
- *Алгоритми аналізу графу.*
- *Приклади задач на графах.*

Подано визначення графу та охарактеризовано його основні властивості. Показано способи подання графу та варіанти реалізації операцій над ним.

7.1. ПОНЯТТЯ ГРАФУ

✱ ***Граф*** *– двійка* G = *(*X, U*), де* X *– множина елементів (вершин, вузлів), а* U *– бінарне відношення на множині* X *(*U ⊆ X x X*). Якщо* |X| = n, *то граф є* скінченим. *Елементи* U *називають або* дугами, *або* ребрами.



X = {X1, X2, X3, X4}
X×X = {(X1,X1), …,(X4,X4)}
U = {(X1,X2), (X1,X3), (X2,X3), (X2,X4)}⇒ U ⊆ X×X = X^2

Якщо (X_i, X_j) – впорядкована пара, то такий граф називається **орієнтованим (орграфом)**, а елементи *U* називаються **дугами**.

Якщо (X_i, X_j) = (X_j, X_i), то граф – **неорієнтований (неорграф),** а елементи *U* називаються **ребрами**.

F: *U* ⟶ *R* – якщо задано таку функцію, то граф є **зваженим**, де *R* – множина дійсних чисел, тобто відображення дуги на число.

Загалом: ⌀ *U* ⊆ *X* x *X*.

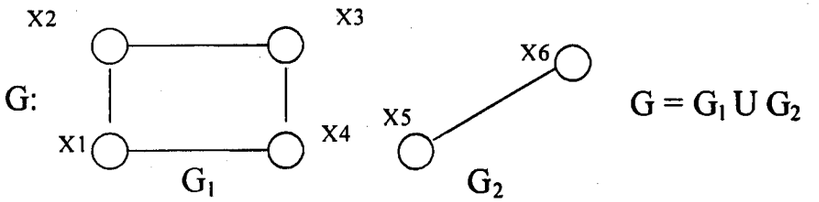
Для орієнтованого графу кількість ребер, що входять у вузол, називається **напівступенем вузла**, кількість ребер, що виходять з вузла – **напівступенем результату**. Кількість вхідних та вихідних ребер може бути довільною, у тому числі і нульовою. Граф без ребер називається **нуль-графом.**

Мультиграфом називається граф, що має паралельні (що сполучають одні і ті ж вершини) ребра, інакше граф називається **простим**.

Компонентами орграфу є: дуга, шлях, контур. **Шлях** – така послідовність дуг, у

якій кінець кожної попередньої дуги збігається з початком наступної. **Контур** – кінцевий шлях, у якому початкова вершина збігається з кінцевої. Граф, у якому є контур, називається **циклічним**. Контур одиничної довжини називають **петлею**.

Компонентами неорграфу є, відповідно: ребро, ланцюг, цикл. **Ланцюг** – безперервна послідовність ребер між парою вершин неорієнтованого графу. Неорієнтований граф називають **зв'язним**, якщо будь-які дві його вершини можна з'єднати ланцюгом. Якщо ж граф – незв'язний, то його можна розбити на підграфи. Наприклад:



Слабка зв'язність – орграф замінюється неорієнтованим графом, який, у свою чергу, є зв'язним. **Однобічна зв'язність** – це така зв'язність, коли між двома вершинами існує шлях в одну або в іншу сторону. **Сильна зв'язність** – це зв'язність, коли між будь-якими двома вершинами існує шлях в одну й в іншу сторону.

Отже, багатозв'язна структура має такі властивості:

1) на кожен елемент (вузол, вершину) може бути довільна кількість посилань;

2) кожен елемент може мати зв'язок з будь-якою кількістю інших елементів;

3) кожна зв'язка (ребро, дуга) може мати напрямок і вагу.

Дерево – зв'язний граф без циклів.

Ліс (або ациклічний граф) – неограф без циклів (може бути і незв'язним).

Контур (каркас) зв'язного графу – дерево, що містить всі вершини графу. Визначається неоднозначно.

Редукція графу – контур з найбільшим числом ребер.

Цикломатичне (або циклічний ранг) число графу $S=m-n+c$, де n – кількість вершин, m – кількість ребер, c – кількість компонент зв'язності графу.

Коциклічний ранг (або коранг) $S^*=n-c$.

Неограф G є лісом тоді і тільки тоді, коли $S(G)=0$.

Неограф G має єдиний цикл тоді і тільки тоді, коли $S(G)=1$.

Контур неографу має S^* ребер.

Ребра графа, що не входять в контур, називаються **хордами.**

Цикл, що виходить при додаванні до контуру графу його хорди, називається **фундаментальним** щодо цієї хорди.

7.2. ПОДАННЯ ГРАФУ В ПАМ'ЯТІ КОМП'ЮТЕРА

Графічний спосіб подання (якщо граф невеликий).

Використання матриць. Матриця легко описуванняється, й при аналізі характеристик графу можна використати алгоритми лінійної алгебри. Також використається подання графа у зв'язній пам'яті, у тому випадку, якщо значна кількість елементів у матриці дорівнює нулю (матриця не заповнена).

Одним із матричних способів подання графу є **матриця суміжності**. Нехай задано граф $G = (X, U)$, $|X| = n$. Маємо матрицю A розмірності n x n, що називається *матрицею суміжності*, якщо елементи її визначаються так:

$$a_{ij} = \begin{cases} 1, (X_i, X_j) \in U \\ 0, (X_i, X_j) \notin U \end{cases}$$

Приклад 7.1. Нехай маємо граф, поданий рис. 7.1

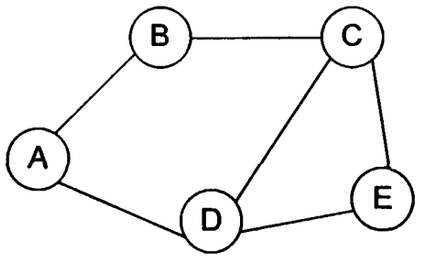


Рис. 7.1. Приклад графу.

Йому відповідає така матриця суміжності:

	A	B	C	D	E
A	0	1	0	1	0
B	0	0	1	1	0
C	0	0	0	1	1
D	1	0	1	0	1
E	0	0	1	1	0

Розглянемо застосування матричної алгебри для визначення характеристик графу. Вираз $a_{ik} L a_{kj}$ означає, що між вузлами i і j є дві дуги, що проходять через вузол k, якщо значення виразу дорівнює True.

Вираз $\bigvee_{k=1}^{n} a_{ik} \wedge a_{kj}$ означає, що завжди є шлях між цими вузлами довжиною 2, якщо вираз є істинним.

A L $A = A^{(2)}$ – логічні операції заміняються арифметичними. Тоді кожний елемент a_{ij} буде подавати інформацію про те, є чи шлях з i в j ($i, j = 1, 2, \ldots, n$)...

Вираз $A^{(n)} = A^{(n-1)}$ L A означає, чи є шлях довжиною n між різними вузлами і. По діагоналі буде характеристика, чи є цикли (контури) у матриці.

$P = A \vee A^{(2)} \vee \ldots \vee A^{(n)} = \bigvee_{k=1}^{n} A^{(k)}$ – **матриця зв'язності**. Визначає, чи існує який-небудь шлях між вузлами i та j. Алгоритм обчислення заданого виразу:

1) $P = A$;
2) повторити 3, 4 (k=1, 2,..., n);
3) повторити 4 для i=1, 2, ...,n;
4) повторити $P_{ij} = P_{ij} \vee (P_{ik} L P_{kj})$, j=1, 2,..., n...

У зв'язній пам'яті найчастіше подання графу здійснюється за допомогою **структур**

суміжності. Для кожної вершини множини X задається множина $M(X_i)$ відповідно до дуг її послідовників (якщо це орграф) або сусідів (для неорграфу). Отже, структура суміжності графу G буде списком таких множин: $< M(X_1), M(X_2), \ldots, M(X_n) >$ для всіх його вершин.

Приклад 7.2. Нехай маємо граф, поданий рис. 7.2 (вузли позначаємо у вигляді цифр: 1, 2,..., n):

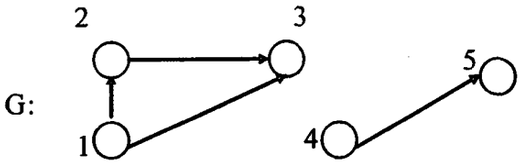
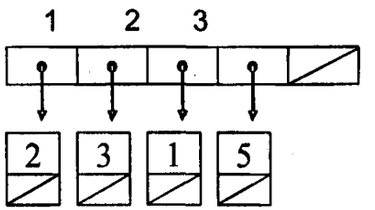


Рис. 7.2. Подання графу.

Для неорграфу:
1: 2, 3;
2: 1, 3;
3: 1, 2;
4: 5;
5: 4.

Для орграфу:
1: 2;
2: 3;
3: 1;
4: 5;
5: - .

Структуру суміжності можна реалізувати масивом з n лінійно зв'язаних списків:



Подання графу може вплинути на ефективність алгоритму. Часто запис алгоритмів на графах задається в термінах вершин і дуг, незалежно від подання графу. Наприклад, алгоритм визначення кількості послідовників вершин: $C(X)$=0, S – кількість дуг.

```
S = 0;
∀ x ∈ X виконати:
початок
C(x)=0;
∀ t ∈ M(x) виконати: C(x) = C(x) + 1;
S = S + C(x);
кінець;
```

7.3. АЛГОРИТМИ ПРОХОДЖЕННЯ ГРАФУ

Алгоритм проходження може бути використаний як алгоритм пошуку, якщо вузлами графу є елементи таблиці.

Маємо граф $G = (X, U)$, $X = \{x_1, x_2, \ldots, x_n\}$… Кожне проходження можна розглядати як певну послідовність. Максимальна кількість проходжень (перестановок) – $n!$.

7.3.1. Алгоритм проходження графу вглиб

Для пояснення принципу проходження графу вглиб скористаємося графом, поданим на рис. 7.3.

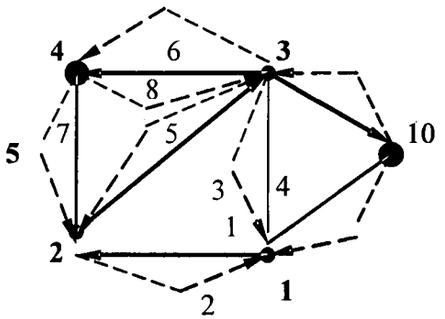


Рис. 7.3. Демонстрація проходження графу вглиб.

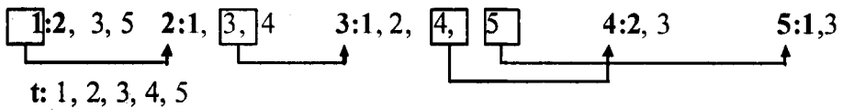
Цифри на ребрах графу позначають кроки відвідування. Будемо розрізняти відвідувані (¬) і не відвідувані (¬ -) вершини. Кожна вершина обходиться два рази. Якщо всі суміжні вершини пройдені, то повертаємося у попередню вершину.

Проходження графу вглиб здійснюється за такими правилами:

1) перебуваючи у вершині x, треба рухатися до будь-якої іншої, раніше не відвіданої вершини (якщо така знайдеться), одночасно запам'ятовуючи дугу, по якій ми вперше потрапили до цієї вершини;

2) якщо з вершини x ми не можемо потрапити до раніше не відвіданої вершини або такої взагалі немає, то ми повертаємося у вершину z, з якої вперше потрапили до x, і продовжуємо обхід вглиб з вершини z.

Визначимо списки суміжності для кожної вершини графу G:



Якщо граф G зв'язний, то описаний процес визначає проходження (обхід) графу G. Якщо ж граф G не зв'язний, то проходимо тільки одну з компонентів графу G, що містить початкову вершину. Якщо граф G є незв'язним, то для одержання повного обходу необхідно досягати такого результату у кожному зв'язному компоненті. За допомогою цього методу можна визначити кількість компонентів.

Для кожного вибору початкової вершини у зв'язному графі може бути отримане єдине проходження. Якщо граф являє собою об'єднання компонентів: $G = \bigcup_{i=1}^{P} G_i$ і $|x_i| = n_i$, то кількість проходжень такого незв'язного графа: $n_1 \cdot n_2 \cdot \ldots \cdot n_p$ ($i = 1, 2, \ldots, p$)…

Алгоритм проходження графу вглиб буде виглядати так:

```
procedure ОБХІД-ВГЛИБ(p: вершина);
begin
    відвідати вершину p;
    for all q from множини вершин, суміжних до p, do
        if q ще не відвідана then ОБХІД-ВГЛИБ(q) end
    end
end;
    begin
        for all p from множини вершин G do
            if p ще не відвідувалась then ОБХІД-ВГЛИБ(p) end
        end
    end.
```

На мові Паскаль рекурсивна процедура проходження вглиб виглядає так:

```
procedure dfs(v:integer);
var
i:integer;
begin
    used[v]:=true; {відзначити вершину як відвідану}
    for i:=1 to n do
        {якщо між вершинами є зв'язок та вершина не відвідана}
        if (a[v,i]=1)and(not used[i]) then
            dfs(i); {викликаємо процедуру з цією вершиною}
end;
```

Нерекурсивна функція проходження графу вглиб виглядає так:

```
procedure dfs(v: integer);
var
i: integer;
found: boolean;
begin
    used[v]:=true; {відзначили вершину як відвідану}
    inc(c); {збільшуємо кількість занесених у стек вершин}
    st[c] := v; {занесли у стек}
    {доки стек не порожній}
        while c > 0 do
            begin
                v := st[c]; {беремо вершину з голови стеку}
                found := false; { шляху не знайдено}
                {проходимо по усіх вершинах з метою пошуку шляху з обраної
                вершини}
                for i := 1 to n do
                    if a[v, i] and not used [i] then
                        begin
                            found := true; {знайшли шлях}
                            break;
```

```
                end;
            {якщо шлях знайдений}
            if found then
                begin
                    used [i] := true;
                    inc(c); {додається у стек}
                    st[c] := i;
                    p[i] := v;
                end
            else {вилучаємо вершину зі стека}
            dec(c);
        end;
end;
```

7.3.2. Алгоритм проходження графу вшир

При проходженні вшир замість стека рекурсивних викликів зберігається черга, в яку записуються вершини в порядку віддалення від початкової.

Визначимо списки суміжності для кожної вершини графа G (рис. 7.3):

1:2, 3, 52:1, 3, 4 3:1, 2, 4, 5 4:2, 3 5:1,3

t: **1, 2, 3, 4, 5 (для обраної вершини переписуємо весь список суміжності).**

Виберемо початкову вершину x_u. Перші елементи шуканої перестановки t є елементами суміжного списку вершини x_u, тобто $M(x_u)$. Позначимо список суміжності в такий спосіб: $M(x_u) = \{(w_1, x_u), (w_2, x_u),\ldots,(w_k, x_u)\}$. Наступними елементами перестановки будуть ті елементи $M(w_1)$, які відсутні у формованій перестановці t. Потім, беремо всі елементи з $M(w_2)$, і т.ін. Проходження припиняється, коли всі вершини, досяжні з x_u, будуть утримуватися в t (w_i O X, i=1, 2,..., k)...

Знову ж таки, введемо масив used, а також створимо чергу для зберігання вершин (реалізуємо її у вигляді масиву queue; природно, можливі інші варіанти). У початок черги запишемо початкову вершину.

```
Star:=1; {початкова вершина – перша}
queue[1] = start;
used[start] = 1;
r = 1, w = 2;
{r - позиція черги, з якої читаємо дані, змінна w — позиція, куди дані будемо записувати}
while (r < w) do
    begin
        curr:= queue[r]; {беремо перший елемент із черги}
        incl;
        for i: = 1 to N do
            begin
                if (used [i]:= true) and (a[curr][i]=1) then
                    begin
                        used[i] := 1;
                        queue[w] := I;
                        inc(w);
```



```
                end;
            end;
    end;
```

7.4. ІНШІ ЗАДАЧІ НА ГРАФАХ

7.4.1. Топологічне сортування

** Топологічним сортуванням називають порядок нумерації вершин орієнтованого графу, при якому будь-яке ребро йде з вершини з меншим номером у вершину з більшим. Для цього алгоритму використовується модифікований метод проходження вглиб.*

Очевидно, що не будь-який граф можна відсортувати топологічно. Можна довести, що топологічне сортування існує для ациклічних графів і не існує для циклічних.

```
Procedure dfs(v:integer);
var
i, j:integer;
begin
    {позначили вершину як відвідану}
    used[v]:=true;
    for i:=1 to n do
        {якщо між вершинами є зв'язок та вершина не відвідана}
        if (a[v,i]=1)and(not used[i]) then
            begin
                {занесли відвідану вершину у масив порядку відвідувань}
                p[j] := v;
                inc(j);
                {викликаємо процедуру з цією вершиною}
                dfs(i);
            end;
end;
```

7.4.2. Пошук мостів

Це завдання вирішується за допомогою пошуку вглиб. Алгоритм працює за $\Theta(N \log N + M)$. Спочатку перетворимо граф в орієнтований, для цього виконуємо серію пошуків вглиб і кожне ребро орієнтуємо в тому напрямку, в якому ми намагалися по ньому обійти в процесі пошуку. Тепер знаходимо в отриманому графі всі компоненти сильної зв'язності. Мостами будуть ті ребра, кінці яких належать різним компонентам.

```
{pre_c – перша вершина, на початку вона дорівнює 1}
procedure dfs(v: integer);
var
i: integer;
begin
    pre[v] := pre_c; {початок дуги}
    low[v] := pre_c; {кінець дуги}
    inc(pre_c);
    for I := 1 to n do
```

```
        if a[v, i]=1 then
            if pre[i] = 0 then
                begin
                    p[i] := v;
                    dfs(i);
                    low[v] := min(low[v], low[i]);
                    {якщо є сильна зв'язність}
                    if low[i] = pre[i] then
                        write(v, '-', I, ' ');
                end
            else if p[v] <> I then
                low[v] := min(low[v], pre[i]);
    { min – функція, що повертає мінімальне значення}
end;
```

7.4.3. Задача про максимальний потік

Часто в завданнях зустрічається наступна конструкція – є будинки й дороги, що їх з'єднують; для кожної дороги є довжина (будинки – вершини, дороги – ребра, довжина дороги – вага ребра). Виникає задача знайти мінімальну вагу шляху між вершинами s і k у графі. Пропускна здатність дуги (i,j) означає, наприклад, скільки вантажу можна перевезти дорогою (i,j) за одиницю часу; потік дугою (i,j) – це скільки перевозиться зараз насправді.

Використаємо такі позначення: $G(x(i))$ – множина вершин, у які є дуга з вершини i; $D(x(i))$ – множина вершин, з яких є дуга у вершину i.

Нехай у графі є N вершин.

Довжини дуг звичайно заносяться у матрицю суміжності C розміру N x N:

```
var C: array [1..N,1..N] of integer;
```

Елемент $C[i,j]$ цієї матриці дорівнює довжині дуги, що з'єднує вершини i і j, і дорівнює (наприклад) 0 або -1, якщо такої дуги немає. Якщо дорога двонаправлена (дуга неорієнтована), то очевидно, що $C[i,j]=C[j,i]$.

Алгоритм розміщення позначок для задачі про максимальний (від s до t) потік.

А. Розміщення позначок. Вершина може перебувати в одному із трьох станів: вершині присвоєна позначка й вершина переглянута (тобто вона має позначку і всі суміжні з нею вершини "опрацьовані"), позначка присвоєна, але вершина не переглянута (тобто вона має позначку, але не всі суміжні з нею вершини опрацьовані), вершина не має позначки. Позначка вершини $x(i)$ складається з двох частин і має один із двох типів: $(+x(j),m)$ або $(-x(j),m)$. Частина $+x(j)$ позначки першого типу означає, що потік допускає збільшення вздовж дуги $(x(j),x(i))$. Частина $-x(j)$ позначки іншого типу означає, що потік може бути зменшений вздовж дуги $(x(i),x(j))$. В обох випадках m задає максимальну величину додаткового потоку, що може протікати від s до $x(i)$ уздовж побудованого ланцюга потоку, що збільшується. Присвоєння позначки вершині $x(i)$ відповідає знаходженню ланцюга потоку, що збільшується, від s до $x(i)$. Спочатку всі вершини не мають позначок.

Крок 1. Присвоїти вершині s позначку $(+s,m(s)=$нескінченність$)$. Вершині s присвоїти позначку іввважати її переглянутою, всі інші вершини без позначок.

Крок 2. Взяти деяку непереглянуту вершину з позначкою; нехай її позначка буде $(+\text{-}x(k),m(x(i)))$ (+- позначає, що перед $x(k)$ може стояти як плюс, так і мінус).

(I) Кожній позначеній вершині $x(j)$, що належить $G(x(\text{I}))$, для якої $c(i,j)<q(i,j)$, присвоїти позначку $(\text{-}x(i),m(x(j)))$, де

$$m(x(j))=\min[m(x(i)),q(i,j)\text{-}c(i,j)].$$

(II) Кожній непозначеній вершині $x(j)$, що належить $D(x)$, для якої $c(i,j)>0$, присвоїти позначку $(\text{-}x(i),m(x(j)))$, де

$$m(x(j))=\min[m(x(i)),c(j,i)].$$

(Тепер вершина $x(i)$ і позначена, і переглянута, а вершини $x(j)$, яким присвоєні позначки у (I) і (II), є непереглянутими.) Позначити, що вершина $x(i)$ переглянута.

Крок 3. Повторювати крок 2 доти, поки або вершина t буде позначена, і тоді перейти до кроку 4, або t буде не позначена й ніяких інших позначок не можна буде розставити; у цьому випадку алгоритм закінчує роботу з максимальним вектором потоку c. Тут слід зазначити, що якщо $X(0)$ – множина позначених вершин, а $X'(0)$ – множина не позначених, то $(X(0)\ !\ X'(0))$ є мінімальним розрізом.

Б. Збільшення потоку.

Крок 4. Присвоїти $x=t$ і перейти до кроку 5.

Крок 5.

(I) Якщо позначка у вершині x має вигляд $(+z,m(x))$, то змінити потік уздовж дуги (z,x) з $c(z,x)$ на $c(z,x)+m(t)$.

(II) Якщо позначка у вершині x має вигляд $(\text{-}x,z)$, то змінити з $c(x,z)$ на $c(x,z)\text{-}m(t)$.

Крок 6. Якщо $z=s$, то стерти усі позначки і повернутися до кроку 1, щоб почати розставляти позначки, але використовуючи вже покращений потік, знайдений на кроці 5. Якщо $z<>s$, то $x=z$ і повернути до кроку 5.

7.4.4. Найкоротша відстань між вершинами (алгоритм Дейкстри)

Алгоритм Дейкстри використовується для визначення найкоротшої відстані між двома вершинами.

Позначення:

$D[i]$ – найкоротша у цей момент відстань від вершини *поч* до вершини i.

flag[i]- інформація про перегляд вершини i: 0 – якщо вершина не переглянута, 1 – якщо переглянута. Якщо вершина переглянута, то для неї $D[i]$ є найкоротшою відстанню від вершини *поч* до вершини i.

предок[i] – інформація про номер вершини, що передує вершині i у найкоротшому шляху від вершини *поч*.

мінв – це мінімальна відстань.

```
для і від 1 до N виконувати
    предок[i]:=нач;
    flag [i]:=0;
    D[i]:=C[нач,i]
flag [поч]:=1; {поки ми знаємо тільки відстань}
```

```
предок[поч]:=0 {від вершини нач до її ж, рівне 0}
для і від 1 до N-1 виконувати
    мінв:=нескінченність;
    для j від 1 до N виконувати
    якщо (flag [j]=0 і (мінв > D[j]) {знаходимо мінімальне}
    то мінв:=D[j]; {відстань}
    k:=j; {до непомічених вершин}
flag [k]:=1; {вершина k позначається переглянутою}
для j від 1 до N виконувати {виконуємо перегляд}
    якщо flag [j]=0 і D[j]>D[k]+C[k,j]
    { якщо для вершини j ще не знайдена найкоротша відстань
    від поч, і з вершини k по дузі C[k,j] шлях у j коротший,
    ніж знайдений раніше}
    то D[j]:=D[k]+C[k,j] {то запам'ятовуємо його}
    предок[j]:=k;
```

Резюме

1. Граф складається з множини вершин та множини ребер(дуг). Якщо вершини. Що формують дугу, впорядковуються, то граф називається впорядкованим (орграфом).

2. Якщо дуги мають значення, то граф є зваженим.

3. Для орієнтованого графа кількість ребер, що входять у вузол, називається напівступенем вузла, кількість ребер, що виходять з вузла – напівступенем результату. Граф без ребер називається нуль-графом.

4. Мультиграфом називається граф, що має паралельні (що сполучають одні і ті ж вершини) ребра, інакше граф називається простим.

5. Компонентами орграфа є: дуга, шлях, контур. Контур одиничної довжини називають петлею.

6. Компонентами неорграфа є, відповідно: ребро, ланцюг, цикл. Ланцюг – безперервна послідовність ребер між парою вершин неорієнтованого графу. Неорієнтований граф називають зв'язним, якщо будь-які дві його вершини можна з'єднати ланцюгом. Якщо ж граф – незв'язний, то його можна розбити на підграфи.

7. Кількісними характеристиками графу є кількість вершин, кількість дуг, циклічний ранг, коранг.

8. Подати граф можна графічно, матрицею суміжності, матрицею зв'язності, структур суміжності.

9. Прохоження графу може відбуватися вглиб і вшир.

10. Топологічне сортування та пошук мостів базуються на алгоритмі проходження графу вглиб.

Контрольні запитання

1. Дайте визначення графу.

2. Дайте визначення дерева за допомогою графу.

3. Поясніть призначення та принципи роботи алгоритмів проходження по графу.

4. Наведіть типи графів та порівняйте їх.

5. Назвіть структури даних, що використовуються в алгоритмах проходження вглиб та вшир.

6. Способи подання графу.

7. Поясніть основні кроки алгоритму розташування позначок.

8. наведіть приклад застосування алгоритму Дейкстри.

Тести для закріплення матеріалу

1. Назвати компоненти неорграфу:

а) дуги;
б) ребра;
в) шлях;
г) контур;
д) ланцюг;
е) цикл.

2. Назвати компоненти орграфу:

а) дуги;
б) ребра;
в) шлях;
г) контур;
д) ланцюг;
е) цикл.

3. За визначенням знайти термін: орграф заміняється неорієнтованим графом, що у свою чергу є зв'язним:

а) слабка зв'язність;
б) однобічна зв'язність;
в) сильна зв'язність.

4. За визначенням знайти термін: між двома вершинами існує шлях в одну або в іншу сторону:

а) слабка зв'язність;
б) однобічна зв'язність;
в) сильна зв'язність.

5. За визначенням знайти термін: між будь-якими двома вершинами існує шлях в одну й в іншу сторону:

а) слабка зв'язність;
б) однобічна зв'язність;

в) сильна зв'язність.

6. Способи подання графу:

а) графічний;
б) матричний;
в) описовий;
г) перелічуваний.

7. Знайти правильний вислів:

а) будь-яке дерево є графом;
б) будь-який граф є деревом;
в) будь-яке двійкове дерево є графом;
г) для того, щоб граф був деревом, в ньому не має бути циклів.

8. Визначити суміжні вершини до вершини B у графі, заданому матрицею суміжності:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	1	0	0	0
E	0	0	0	1	0

а) ACD;
б) CD;
в) AC;
г) немає правильної відповіді.

9. Визначити суміжні вершини до вершини A у графі, заданому матрицею суміжності:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	1	0	0	0
E	0	0	0	1	0

а) BCD;
б) CD;
в) B;
г) немає правильної відповіді.

10. Визначити суміжні вершини до вершини B у графі, заданому матрицею суміжності:

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	1	1	0
C	0	0	0	0	1
D	1	1	0	0	0
E	0	0	0	1	0

а) ACD;
б) CD;
в) AC;
г) немає правильної відповіді.

11. Визначити суміжні вершини до вершини C у графі, заданому матрицею суміжності:

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	1	1	0
C	0	0	0	0	1
D	1	1	0	0	0
E	0	0	0	1	0

а) ACD;
б) CD;
в) BE;
г) немає правильної відповіді.

12. Визначити суміжні вершини до вершини D у графі, заданому матрицею суміжності:

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	1	1	0
C	0	0	0	0	1
D	1	1	0	1	0
E	0	0	1	1	0

а) ABD;
б) CD;
в) BE;
г) немає правильної відповіді.

13. Визначити суміжні вершини до вершини E у графі, заданому матрицею суміжності:

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	1	1	0
C	0	0	0	0	1
D	1	1	0	1	0
E	0	0	0	1	0

а) ABD;
б) CD;
в) BE;
г) немає правильної відповіді.

РОЗДІЛ 8

АЛГОРИТМИ ПОШУКУ

- *Загальна класифікація алгоритмів пошуку.*
- *Лінійний пошук.*
- *Двійковий (бінарний) пошук елемента в масиві.*
- *Пошук методом Фібоначчі.*
- *М-блоковий пошук.*
- *Методи обчислення адреси.*
- *Інтерполяційний пошук елемента в масиві.*
- *Бінарний пошук із визначенням найближчих вузлів.*
- *Пошук у таблиці.*
- *Прямий пошук рядка.*
- *Алгоритм Ахо-Корасик.*
- *Алгоритм Кнута, Моріса і Пратта.*
- *Алгоритм Рабіна-Карпа.*
- *Алгоритм Боуєра і Мура.*

Розглянуто алгоритми пошуку стрічок: прямий пошук, Ахо-Корасика, Кнута-Моріса-Прата, Рабіна-Карпа, Боуєра-Мура. Подано алгоритми пошуку у масивах та списках.

8.1. ЗАГАЛЬНА КЛАСИФІКАЦІЯ АЛГОРИТМІВ ПОШУКУ

Усі методи можна розділити на статичні й динамічні. При статичному пошуку масив значень не змінюється під час роботи алгоритму. Під час динамічного пошуку масив може перебудовуватися або змінювати розмірність.

Так само методи пошуку можна розділити на методи, що використовують дійсні ключі, і на методи, що працюють за перетвореними ключами. У цьому випадку «ключем» називають те значення, яке ми шукаємо.

Інший варіант класифікації – методи, засновані на порівнянні самих значень, і методи, засновані на їх цифрових властивостях. Це так звані методи хешування.

8.2. ЛІНІЙНИЙ ПОШУК

Якщо нема додаткових вказівок про розташування необхідного елемента, то природнім є послідовний перегляд масиву із збільшенням тієї його частини, де бажаного елемента не знайдено. Такий метод називається лінійним пошуком. Умови закінчення пошуку наступні.

1. *Елемент знайдений, тобто* $a_i = x$.
2. *Весь масив проглянутий і збігу не знайдено.*
3. *Це дає нам лінійний алгоритм:*

Звертаємо увагу, що якщо елемент знайдений, то він знайдений разом із мінімально можливим індексом, тобто це перший з таких елементів. Очевидно, що закінчення циклу здійсниться, оскільки на кожному кроці значення *i* збільшується, і, отже, воно досягне за скінченну кількість кроків межі *N*; фактично ж, якщо збігу не було, це відбудеться після *N* кроків.

```
Int LinearSearch (int[] a, int N, int L, int R, int Key)
{
    for (int I = L;i<= R; i++)
        if (a[i] = = Key)
            return (i);
    return (-1); // елемент не знайдений
}
```

8.3. ДВІЙКОВИЙ (БІНАРНИЙ) ПОШУК ЕЛЕМЕНТА В МАСИВІ

Якщо у нас є масив, що містить впорядковану послідовність даних, то дуже ефективний двійковий пошук.

Змінні *Lb* і *Ub* містять, відповідно, ліву і праву межі відрізка масиву, де міститься потрібний елемент. Починаємо завжди з дослідження середнього елементу відрізка. Якщо шукане значення менше від середнього елемента, ми переходимо до пошуку у верхній половині відрізка, де всі елементи менші від тільки що перевіреного. Іншими словами, значенням *Ub* стає $(M-1)$ і на наступній ітерації ми працюємо з половиною масиву. Отже, в результаті кожної перевірки ми удвічі звужуємо область пошуку.

Блок-схема бінарного пошуку подана на рис. 8.1.

Функція, що реалізує бінарний пошук, має такий вигляд:

```
int BinarySearch (int a, int
Lb, int Ub, int Key)
{
int M;
do
    M = Lb + (Ub-Lb)/2;
    //шукаємо середину
    //відрізку
        if (Key < a[M])
```

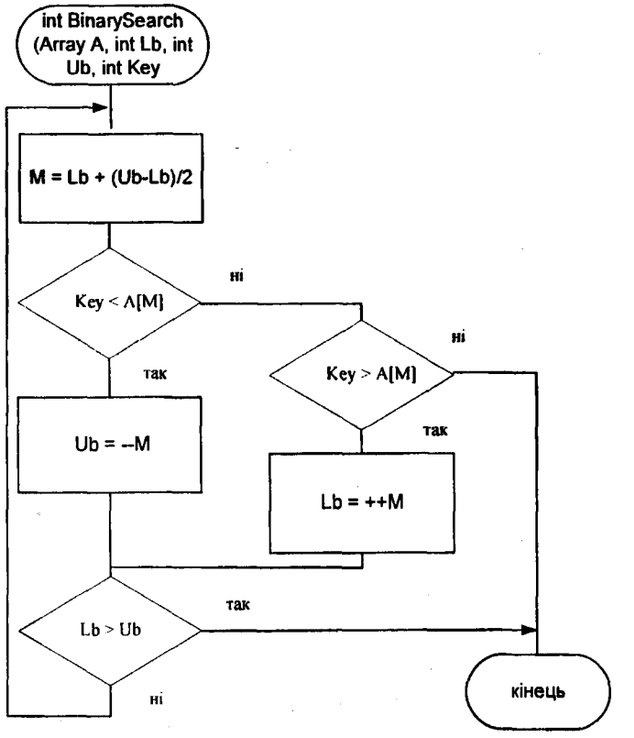


Рис. 8.1. Блок-схема функції бінарного пошуку за параметром.

```
            Ub = —M; //переходимо у ліву частину
        else if (Key > a[M])
            Lb = ++M; //переходимо у парву частину
        else
            return (M);
    if (Lb > Ub)
        return (-1); //не знайдено
while (1);
}
```

8.4. ПОШУК МЕТОДОМ ФІБОНАЧЧІ

Він працює швидше, ніж бінарний пошук, оскільки замість операцій ділення, що застосовуються у попередньому методі, використовує операції додавання та віднімання. Суть методу полягає у визначенні наступного елемента для порівняння з числами Фібоначчі (звідси і назва). Зменшення індекса означає перехід по попереднього числа, збільшення – перехід до наступного.

Числа Фібоначчі формуються на основі додавання двох попередніх чисел, де перше і друге число дорівнюють 1. Тобто, можна скласти таку послідовність чисел:

1, 1, 2, 3, 5, 8, 13, 21, 34…

Блок-схема пошуку методом Фібоначчі подана на рис. 8.2.

```
int find_fibo(int strPar)
{
    int resFind, a[10];
    int q=Fibonachi(1), p=Fibonachi(2), i=Fibonachi(3);
    //функція, що визначає число Фібоначчі за номером
    int FindResult = -1;
    do
    {
        if(a[i]= strPar)
        {
            FindResult = i;
            curSelRow = i+1;
            resFind = a[FindResult];
            return (resFind);
        }
    else
        if (if(a[i]> strPar)
            if(q==0)
            {
                resFind = NULL;
                return (resFind);
            }
            else
                {i = i-q; p=q; q=p-q;} //перерахунок наступного числа
```



```
        else
            if(p==1)
            {
                resFind = NULL;
                return (resFind);
            }
            else
            {
              i = i+q; p=p-q; q=q-p;
            }
    }
    while(resFind);
        if(FindResult == -1)
            resFind = NULL;
    return (resFind);
}
```

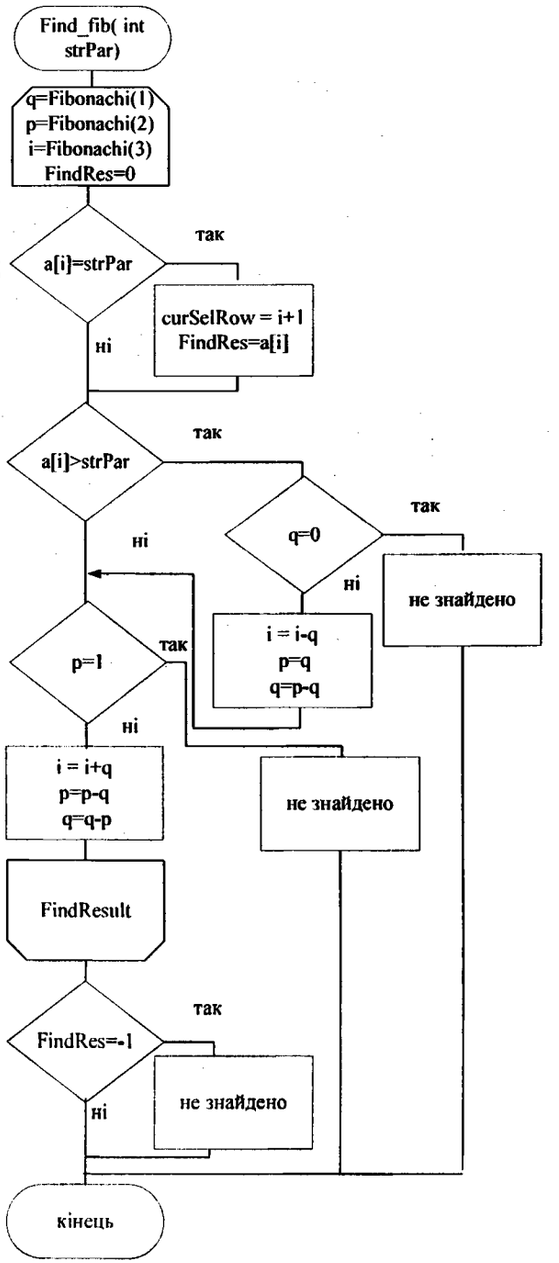


Рис. 8.2. Блок-схема функції пошуку Фібоначчі за параметром.

8.5. М-БЛОКОВИЙ ПОШУК

Цей спосіб зручний при індексному збереженні списку. Передбачається, що вихідний упорядкований список B довжини N розбитий на M підсписків $B=B_1, B_2,..,B_m$ довжини $N_1, N_2,\ldots,N_m$, таким чином, що $B=B_1, B_2,..,B_m$.

Для знаходження ключа V, треба спочатку визначити перший зі списків B_i, i=1,...,M, останній елемент якого більше V, а потім застосувати послідовний пошук до списку Bi.

Збереження списків Bi може бути зв'язним чи послідовним. Якщо довжини всіх підсписків приблизно рівні і M=N, то Max M-блокового пошуку дорівнює 2N. При однаковій частоті використання елементів Avg M - блокового пошуку дорівнює N.

Описаний алгоритм ускладнюється, якщо не відомо, чи дійсно в списку є елемент, що збігається з ключем V. При цьому можливі випадки: або такого елемента в списку немає, або їх кілька.

Якщо замість ключа V є упорядкований список ключів, то послідовний чи M-блоковий пошук може виявитися зручнішим, ніж бінарний, оскільки не треба повторної ініціалізації для кожного нового ключа зі списку V.

8.6. МЕТОДИ ОБЧИСЛЕННЯ АДРЕСИ

Нехай у кожному з M елементів масиву T міститься елемент списку (наприклад, ціле позитивне число). Якщо є деяка функція $H(V)$, що обчислює однозначно по елементі V його адресу – ціле позитивне число з інтервалу [0,M-1],то V можна зберігати в масиві T з номером $H(V)$ тобто V=$T(H(V))$. При такому збереженні пошук будь-якого елемента відбувається за постійний час, не залежний від M.

Масив T називається масивом хешування, а функція H – функцією хешування (див. розділ 5).

При конкретному застосуванні хешування зазичай є визначена область можливих значень елементів списку V і деяка інформація про них. На основі цього вибирається розмір масиву хешування M і будується функція хешування. Критерієм для вибору M і H є можливість їхнього ефективного використання.

Нехай треба зберігати лінійний список з елементів K_1 ,K_2,..,K_n, таких, що при K_i=K_j mod(K_i,26)= mod(K_j,26). Для збереження списку виберемо масив хешування T(26) із простором адрес 0-25 і функцію хешування $H(V)$= =mod(V,26). Масив T заповнюється елементами $T(H(K_i))$=K_i і $T(j)$=0, якщо j є у ($H(K_1)$, $H(K_2)$,..,$H(K_n)$).

Пошук елемента V у масиві T із присвоюванням Z його індексу, якщо V міститься в T, чи -1, якщо V не міститься в T, здійснюється так:

```
int t[26],v,z,i;
i=(int)fmod((double)v,26.0);
if(t[i]==v) z=i;
else z=-1;
```

Додавання нового елемента V у список з поверненням у Z індексу елемента, де він буде зберігатися, реалізується фрагментом

```
z=(int)fmod((doule)v,26.0);
t[z]=v;
```

а виключення елемента V зі списку присвоєнням

```
t[(int)fmod((double)v,26)]=0;
```

Тепер розглянемо складніший випадок, коли умова K_i=Kj $H(K_i)$=$H(K_j)$ не виконується. Нехай V – множина можливих елементів списку (цілі позитивні числа), у якому максимальна кількість елементів дорівнює 6. Візьмемо M=8 і як функцію хешування виберемо функцію $H(V)$=mod(V,8).

Припустимо, що B=8, причому $H(K_1)$=5, $H(K_2)$=3, $H(K_3)$=6, $H(K_4)$=3, $H(K_5)$=1, тобто $H(K_2)$=$H(K_4)$ хоча K_2=K_4. Така ситуація, як показано у попередніх розділах, називається колізією, і в цьому випадку при заповненні масиву хешування треба метод для її дозволу. Зазвичай вибирається перша вільна комірка за власною адресою. Для нашого випадку масив T[8] може мати вигляд

$$T=<0,K_5,0,K_2,K_4,K_1,K_3,0>.$$

При наявності колізій ускладнюються всі алгоритми роботи з масивом хешування. Розглянемо роботу з масивом T[100], тобто з простором адрес від 0 до 99. Нехай кількість елементів N не більш 99, тоді в T завжди буде хоча б один вільний елемент дорівнює нулю. Для оголошення масиву використовуємо оператор

```
int static t[100];
```

Додавання в масив T нового елемента Z із занесенням його адреси в I і числа елементів у N виконується так:

```
i=h(z);
while (t[i]!=0 && t[i]!=z)
    if (i==99) i=0;
    else i++;
    if (t[i]!=z)  t[i]=z,  n++;
```

Пошук у масиві T елемента Z із присвоєнням I індекса Z, якщо Z є в T, чи -=-1, якщо такого елемента немає, реалізується в такий спосіб:

```
i=h(z);
while (t[i]!=0 && t[i]!=z)
    if (i==99) i=0;
    else i++;
    if (t[i]==0) i=-1;
```

При наявності колізій виключення елемента зі списку шляхом позначення його як порожнього, тобто $t[i]$=0, може привести до помилки. Наприклад, якщо зі списку B виключити елемент K_2, то одержимо масив хешування T=<0,K_5,0,K_2,K_4,K_1,K_3,0>, у якому неможливо знайти елемент K_4, оскільки $H(K_4)$=3, а T(3)=0. У таких випадках при виключенні елемента зі списку можна записувати в масив хешування деяке значення, що не належить області значень елементів списку і не рівне нулю. При роботі з таким масивом це значення буде вказувати на те, що треба переглядати із середини комірок.

Перевага методів обчислення адреси полягає в тому, що вони найшвидші, а недолік у тому, що порядок елементів у масиві T не збігається з їх порядком у списку, крім того, досить складно здійснити динамічне розширення масиву T.

8.7. ІНТЕРПОЛЯЦІЙНИЙ ПОШУК ЕЛЕМЕНТА В МАСИВІ

Якщо відомо, що ключ лежить між K_1 і K_u, то наступний пошук доцільно здійснювати не всередині впорядкованого масиву, а на відстані (u-l)(K-K_1)/(K_u-K_1) від l, припускаючи, що ключі є числами, що зростають приблизно в арифметичній прогресії.

Інтерполяційний пошук працює за log log N операцій, якщо дані розподілені рівномірно. Як правило, він використовується лише на дуже великих таблицях, причому робиться декілька кроків інтерполяційного пошуку, а потім на малому відрізку використовується бінарний або послідовний варіант.

```
Int interpolationSearch(int[] a, int toFind, int high)
{
 // повертає індекс елемента зі значенням toFind або -1, якщо такого // елемента нема
int low = 0;
//high – кількість елементів масиву
int mid;
while (a[low] < toFind && a[high] >= toFind)
{
    mid = low + ((toFind – a[low]) * (high – low)) / (a[high] – a[low]);
    if (a[mid] < toFind)
        low = mid + 1;
    else if (a[mid] > toFind)
        high = mid – 1;
    else
        return mid;
}
if (a[low] == toFind)
    return low;
else
    return -1; // Not found
}
```

8.8. БІНАРНИЙ ПОШУК ІЗ ВИЗНАЧЕННЯМ НАЙБЛИЖЧИХ ВУЗЛІВ

У ряді випадків (зокрема, в завданнях інтерполяції) доводиться з'ясовувати, де по відношенню до заданого впорядкованого масиву дійсних чисел розташовується задане дійсне число. На відміну від пошуку в масиві цілих чисел, задане число в цьому випадку найчастіше не співпадає ні з одним із чисел масиву, і вимагається знайти номери елементів, між якими це число може бути розміщене.

Одним з найшвидших способів цього є бінарний пошук, характерна кількість операцій якого має порядок $\log_2(n)$, де n – кількість елементів масиву. При численних зверненнях до цієї процедури кількість операцій буде рівна $m \log_2(n)$ (m – кількість обходів). Прискорення цієї процедури можна добитися за рахунок збереження попереднього результату операції і спроб пошуку при новому обігу в найближчих вузлах масиву з подальшим розширенням області пошуку у разі неуспіху.

При цьому в найгірших випадках кількість операцій буде більшою (приблизно у 2 рази) в порівнянні з бінарним пошуком, але, зазвичай, при m, значно більшою, ніж $\log_2(n)$, вдається довести порядок кількості операцій до m, тобто зробити її майже незалежною від розміру масиву.

Завдання ставиться так. Заданий впорядкований масив дійсних чисел *array* розмірності *n*, значення *value*, що перевіряється, і початкове наближення вузла *old*. Вимагається знайти номер вузла *res* масиву *array*, такий, що *array*[*res*]<=*value*<*array*[*res*+1].

Алгоритм працює таким чином.

1. Визначається, чи лежить значення value за межами масиву array. У разі *value*<*array*[0] повертається -1, у разі *value*>*array*[n-1] повертається n-1.
2. Інакше перевіряється: якщо значення old лежить за межами індексів масиву (тобто *old*<0 або *old*>=n, то переходимо до звичного бінарного пошуку, встановивши ліву межу *left*=0, праву *right*=n-1.
3. Інакше переходимо до з'ясування меж пошуку. Встановлюється *left=right=old*, *inc*=1 – інкремент пошуку.
4. Перевіряється нерівність *value*>=*array*[*old*]. При його виконанні переходимо до наступного пункту (5), інакше до пункту (7).
5. Права межа пошуку відводиться далі: *right=right+inc*. Якщо *right*>=n-1, то встановлюється *right*=n-1 і переходимо до бінарного пошуку.
6. Перевіряється *value*>=*array*[*right*]. Якщо ця нерівність виконується, то ліва межа переміщається на місце правої: *left=right*, *inc* множиться на 2, і переходимо назад на (5). Інакше переходимо до бінарного пошуку.
7. Ліва межа відводиться: *left=left-inc*. Якщо *left*<=0, то встановлюємо *left*=0 і переходимо до бінарного пошуку.
8. Перевіряється *value*<*array*[*left*]. При виконанні права межа переміщується на місце лівої: *right=left*, inc множиться на 2, переходимо до пункту (7). Інакше до бінарного пошуку.
9. Проводиться бінарний пошук у масиві з обмеженням індексів *left* і *right*. При цьому кожного разу інтервал скорочується приблизно в 2 рази (у залежності від парності різниці), поки різниця між *left* і *right* не досягне 1. Після цього повертаємо *left* як результат, одночасно присвоюючи *old=left*.

```
int fbin_search(float value,int *old,float *array,int n)
{
    register int left,right;
    /* перевірка позиції за межами масиву */
    if(value < array[0]) return(-1);
        if(value >= array[n-1]) return(n-1);
        /* процес розширення області пошуку. Перевіряємо валідність
        початкового наближення */
            if(*old>=0 && *old<n-1)
            {
                register int inc=1;
                left = right = *old;
                if(value < array[*old])
                {
                    /* область розширюють вліво */
                    while(1)
                    {
                        left -= inc;
                        if(left <= 0)
                        {
                            left=0;break;
```

```
                }
            if(array[left] <= value) break;
            right=left; inc<<=1;
            }
        }
        else
        {
            /* область розширюють вправо */
            while(1)
            {
                right += inc;
                if(right >= n-1)
                {
                    right=n-1;break;
                }
            }
            if(array[right] > value) break;
            left=right; inc<<=1;
        }
    /* початкове наближення погане–
    за область пошуку приймається весь масив */
    else
    {
        left=0;right=n-1;
    }
    /* це алгоритм бінарного пошуку необхідного інтервалу */
    while(left<right-1)
    {
        register int node=(left+right)>>1;
        if(value>=array[node]) left=node;
        else right=node;
    }
/* повертаємо знайдену ліву межу,
обновивши старе значення результату */
return(*old=left);
}
```

8.9. ПОШУК У ТАБЛИЦІ

Пошук у масиві іноді називають пошуком у таблиці, особливо, якщо ключ сам є складовим об'єктом, таким, як масив чисел або символів. Часто зустрічається саме останній випадок, коли масиви символів називають рядками або словами. Рядковий тип визначається так:

```
String = array[0..Mv1] of char
```

відповідно визначається і відношення порядку для рядків *x* і *y*:

x = y, якщо xj = yj для 0 face=«Symbol» =< j < M
x < y, якщо xi < yi для 0 face=«Symbol» =< i < M і xj = yj
для 0 face=«Symbol» =< j < i

Щоб встановити факт збігу, необхідно встановити, що всі символи порівнюваних рядків відповідно рівні один іншому. Тому порівняння складових операндів зводиться до пошуку частин, що неспівпадають, тобто до пошуку на нерівність. Якщо нерівних частин не існує, то можна говорити про рівність. Припустимо, що розмір слів достатньо малий, скажімо, менше ніж 30. В цьому випадку можна використовувати лінійний пошук і діяти так.

Для більшості практичних застосувань бажано виходити з того, що рядки мають змінний розмір. Це припускає, що розмір вказується в кожному окремому рядку. Якщо виходити з раніше описаного типу, то розмір не повинен перевершувати максимального розміру *M*. Така схема достатньо гнучка і придатна для багатьох випадків, в той самий час вона дозволяє уникнути складнощів динамічного розподілу пам'яті. Найчастіше використовуються два такі подання розміру рядків.

Розмір неявно вказується шляхом додавання кінцевого символа, більше цей символ ніде не вживається. Зазвичай, для цієї мети використовується недрукований символ із значенням 00h. (Для подальшого важливо, що це мінімальний символ зі всієї множини символів.)

Розмір *явно зберігається як перший елемент масиву*, тобто рядок *s* має такий вигляд: $s = s_0, s_1, s_2, \ldots, s_{M-1}$. Тут $s_1, \ldots, s_{M-1}$ – фактичні символи рядка, а $s_0 = \mathrm{Chr}(M)$. Такий прийом має ту перевагу, що розмір явно доступний, а недолік – тому, що цей розмір обмежений розміром множини символів (256).

У подальшому алгоритмі пошуку віддається перевага першій схемі. У цьому випадку порівняння рядків виконується так:

```
i:=0;
while (x[i]=y[i]) and (x[i]<>00h) do i:=i+1;
```

Кінцевий символ працює тут як бар'єр.

Тепер повернемося до завдання пошуку в таблиці. Він вимагає вкладених пошуків, а саме: пошуку по рядках таблиці, а для кожного рядка послідовних порівнянь між компонентами.

8.10. ПРЯМИЙ ПОШУК РЯДКА

Нехай заданий масив *s* з *N* елементів і масив *p* з *M* елементів, причому 0 < M face=«Symbol» =< N. Описані вони так:

```
s: array[0..Nv1] of Item;
p: array[0..Mv1] of Item;
```

Пошук рядка знаходить перше входження *p* в *s*. Зазвичай, Item *v* – це символи, тобто s можна вважати деяким текстом, а *pv* словом, і необхідно знайти перше входження цього слова у вказаному тексті. Ця дія типова для будь-яких систем опрацювання текстів, звідси і очевидна зацікавленість у пошуку ефективного алгоритму для цього завдання. Розглянемо алгоритм пошуку, який називатимемо прямим пошуком рядка.

```
I:=-1;
repeat
```

```
        i:=i+1; j:=0;
        while (j<M) and (s[i+j]=p[j]) do j:=j+1;
    until (j=M) or (i=N-M)
```

Вкладений цикл з передумовою починає виконуватися тоді, коли перший символ слова *p* співпадає із черговим, *i*-м символом тексту *s*. Цей цикл повторюється стільки разів, скільки співпадає символів тексту *s*, починаючи з *i*-го символа, з символами слова *p* (максимальна кількість повторень рівна *M*). Цикл завершується при вичерпанні символів слова p (перестає виконуватися умова *j<M*) або при неспівпаданні чергових символів *s* і *p* (перестає виконуватися умова *s[i+j]=p[j]*). Кількість співпадінь підраховується з використанням j. Якщо порівняння відбулося зі всіма символами слова *p* (тобто слово *p* знайдене), то виконується умова *j=M*, і алгоритм завершується. Інакше пошук продовжується доти, поки не переглянутою залишиться частина тексту s, яка містить символів менше, ніж є в слові *p* (тобто цей залишок вже не може співпасти із словом *p*). У цьому випадку виконується умова *i=N-M*, що теж приводить до завершення алгоритму. Це доводить гарантованість закінчення алгоритму.

Цей алгоритм працює достатньо ефективно, якщо припустити, що неспівпадання пари символів відбувається після незначної кількості порівнянь у внутрішньому циклі. При великій потужності типу Item це достатньо частий випадок. Можна припускати, що для текстів, складених зі 128 символів, неспівпадіння виявлятиметься після одної або двох перевірок. Проте, у гіршому разі, продуктивність може виявитися набагато гіршою.

8.11. АЛГОРИТМ АХО-КОРАСИК

Алгоритм Ахо-Корасик – алгоритм пошуку підрядків в рядку, створений Альфредом Ахо і Маргарет Корасик. Алгоритм реалізує пошук множини підрядків із словника в даному рядку. Час роботи пропорційний $\Theta(M+N+K)$, де N – довжина рядка-зразка, M – сумарна довжина рядків словника, а K – довжина відповіді, тобто сумарна довжина входжень слів із словника в рядок-зразок. Тому сумарний час роботи може бути квадратичним (наприклад, якщо в рядку 'аааааа' ми шукаємо слова 'а', 'аа', 'ааа' ...).

```
q := 0;
for i := 1 to m do
    begin
        while g (q, T [i]) = -1 do
            q := f (q);
        q := g (q, T [i]);
        if out (q) ≠ 0 then write( i ), out (q);
    end;
```

8.12. АЛГОРИТМ МОРІСА-ПРАТА

Цей алгоритм модифікує прямий пошук, збільшуючи розмір зсуву, запам'ятовуючи одночасно частини тексту, що співпадають зі зразком.це дозволяє уникнути непотрібних порівнянь і збільшує швидкість пошуку.

Розглянемо порівняння на позиції *i*, де зразок *x*[0, *m* - 1] порівнюється зі частиною тексту *y*[*i*, *i* + *m* - 1]. Припустимо, що перша розбіжність відбулася між *y*[*i* + *j*] і *x*[*j*], де $1 < j < m$.

Тоді *y*[*i*, *i* + *j* - 1] = *x*[0, *j* - 1] = *u* і *a* = *y*[*i*+*j*] =/= *x*[*j*] = *b*.

Введемо поняття перфікс-функції.

> ✱ ***Префікс-функцією*** *називається функція, що повертає найбільший префікс рядка. Префіксом рядка називається підрядок, який одночасно є і суфіксом (закінченням рядка). Так, для рядка «аавпа» префік-функція поверне символ «а», а для рядка «ааввсваа» - підрядок «аав».*

При зсуві можна очікувати, що префікс зразка *u* співпаде з якимсь суфіксом підслова тексту *u*. Найдовший такий префікс – межа *u* (він зустрічається на обох кінцях *u*). Це приводить нас до наступного алгоритму: нехай *mp_next*[*j*] – довжина межі *x*[0, *j* - 1]. Тоді після зсуву ми можемо відновити порівняння з місця *y*[*i* + *j*] і *x*[*j-mp_next*[*j*]] без втрати можливого місцезнаходження зразка. Таблиця *mp_next* може бути обчислена за (m) перед самим пошуком. Максимальна кількість порівнянь на один символ – *m*.

```
void PRE_MP( char *x, int m, int mp_next[] )
{
    int i, j;
    i=0;
    j=mp_next[0]=-1;
    while ( i < m )
    {
        while ( j > - 1 && x[i] != x[j] )
            j=mp_next[j];
        mp_next[++i]=++j;
    }
}
void MP( char *x, char *y, int n, int m )
{
    int i, j, mp_next[XSIZE];
    PRE_MP(x, m, mp_next );
    i=j=0;
    while ( i < n )
    {
        while ( j > -1 && x[j] != y[i] )
            j=mp_next[j];
        i++;
        j++;
        if ( j >= m )
        {
            OUTPUT( i - j );
            j = mp_next[ j ];
        }
    }
}
```

8.13. АЛГОРИТМ КНУТА, МОРІСА І ПРАТТА

Приблизно в 1970 р. Д. Кнут, Д. Моріс і В. Пратт винайшли алгоритм (КМП), що фактично вимагає тільки N порівнянь навіть в найгіршому випадку. Новий алгоритм ґрунтується на тому міркуванні, що після часткового співпадіння початкової частини слова з відповідними символами тексту фактично відома пройдена частина тексту і можна визначити деякі відомості (на основі самого слова), за допомогою яких потім можна швидко просунутися по тексту. Приведений приклад пошуку слова ABCABD показує принцип роботи такого алгоритму. Символи, що порівнювалися, тут підкреслені. Зверніть увагу: при кожному неспівпадінні пари символів слово зсувається на усю пройдену відстань, оскільки менші зсуви не можуть привести до повного співпадіння.

```
ABCABCABAABCABD
ABCABD
  ABCABD
    ABCABD
      ABCABD
        ABCABD
```

Основною відмінністю КМП-алгоритму від алгоритму прямого пошуку є здійснення зсуву слова не на один символ на кожному кроці алгоритму, а на деяку змінну кількість символів. Отже, перш ніж здійснювати черговий зсуву, необхідно визначити величину зсуву. Для підвищення ефективності алгоритму необхідно, щоб зсув на кожному кроці був якомога більшим.

Якщо j визначає позицію в слові, що містить перший символ, що не співпада (як в алгоритмі прямого пошуку), то величина зсуву визначається як j-D. Значення D визначається як розмір найдовшої послідовності символів слова, які безпосередньо передують позиції j, і яка повністю співпадає з початком слова. D залежить тільки від слова і не залежить від тексту. Для кожного j буде своя величина D, яку позначимо d_j.

Оскільки величини d_j залежать тільки від слова, то перед початком фактичного пошуку можна обчислити допоміжну таблицю d. Відповідні зусилля будуть виправданими, якщо розмір тексту значно перевищує розмір слова ($M<<N$). Якщо треба шукати багато входжень одного і того ж слова, то можна користуватися одними і тими самим d. Точний аналіз КМП-пошуку, як і сам його алгоритм, вельми складний. Його винахідники доводять, що треба порядку $M+N$ порівнянь символів, що значно краще, ніж $M*N$ порівнянь із прямого пошуку. Вони так само відзначають таку позитивну властивість, як показник сканування, який ніколи не повертається назад, тоді як при прямому пошуку після неспівпадіння перегляд завжди починається з першого символа слова і тому може включати символи, які раніше вже були видимими. Це може привести до негативних наслідків, якщо текст читається з вторинної пам'яті, адже в цьому випадку повернення обходиться дорого. Навіть при введенні буфера може зустрітися таке велике слово, що повернення перевищить місткість буфера.

```
Var n : longint;
T : array[1..40000] of char;
S  : array[1..10000] of char;
P: array[1..10000] of word; {масив, в якому зберігається значення префікс-
```



```
функції}
    i,k : longint;
    m  : longint;
    Procedure Prefix; {процедура, яка визначає префікс-функцію}
    Begin
        P[1]:=0; {префікс рівний нулеві}
        k:=0;
        for i:=2 to m do
            begin
                while (k>0) and (S[k+1]<>S[i]) do
                    k:=P[k]; {отримаємо значення функції з попередніх розрахунків}
                if S[k+1]=S[i] then
                    k:=k+1;
                P[i]:=k; {присвоєння префікс-функції}
            end;
    End;
```

8.14. АЛГОРИТМ РАБІНА-КАРПА

Ідея, запропонована *Рабіном* і *Карпом*, має на увазі поставити у відповідність кожному рядку деяке унікальне число і замість того, щоб порівнювати самі рядки, порівнювати числа, що набагато швидше. Проблема в тому, що шуканий рядок може бути довгим, рядків у тексті теж вистачає. А оскільки кожному рядку треба поставити у відповідність число, то і чисел має бути багато, а отже, числа будуть великими (порядку D_m, де D – кількість різних символів), і працювати з ними буде так само незручно.

```
    Var T  : array[1..40000] of 0..9;
        S  : array[1..8] of 0..9;
        i,j  : longint;
        n,m : longint;
        v,w : longint; {v – число, яке характеризує рядок, що шукається,
    w характеризує рядок довжини m у тексті}
        k  : longint;
    const D : longint = 10; {кількість різних символів}
    Begin
        v:=0;
        w:=0;
        for i:=1 to m do
            begin
                v:=v*D+S[i]; {визначення v, рядок поданий як число}
                w:=w*D+T[i]; { визначення початкового значення w}
            end;
        k:=1;
        for i:=1 to m-1 do
            {k необхідне для багатократного визначення w і рівне Dm-1}
```

```
        k:=k*D;
    for i:=m+1 to n+1 do
        begin
            if w=v then {якщо числа рівні, то рядки співпадають}
                writeln('УРА');
            if i<=n then
                w:=d*(w-k*T[i-m])+T[i]; { визначення нового значення w}
        end;
End.
```

Цей алгоритм виконує лінійне проходження по рядку (m кроків) і лінійне проходження по всьому тексту (n кроків), отже, спільний час роботи є $\Theta(n+m)$. Цей час лінійно залежить від розміру рядка і тексту, отже, програма працює швидко. Але який інтерес працювати тільки з короткими рядками і цифрами? Розробники алгоритму придумали, як поліпшити цей алгоритм без особливих втрат у швидкості роботи. Як ви помітили, ми ставили у відповідність рядку його числове подання, але виникала проблема великих чисел. Її можна уникнути, якщо проводити всі арифметичні дії з модулями якогось простого числа (постійно брати залишок від ділення на це число). Таким чином знаходиться не само число, що характеризує рядок, а його залишок від ділення на якесь просте число. Тепер ми ставимо число у відповідність не одному рядку, а цілому класу, але оскільки класів буде досить багато (стільки, скільки різних залишків від ділення на це просте число), то додаткову перевірку доведеться виконувати рідко.

```
V:=0;
w:=0;
for i:=1 to m do { визначення v та w}
    begin
        v:=(v*D+ord(S[i])) mod P; {ord перетворює символ у число}
        w:=(w*D+ord(T[i])) mod P;
    end;
k:=1;
for i:=1 to m-1 do
    k:=k*D mod P; {k маємо значення Dm-1 mod P}
for i:=m+1 to n+1 do
    begin
        if w=v then {якщо числа рівні, то рядки належать до одного класу,
                     перевірка
                на співпадіння}
            begin
                j:=0;
                while (j<m) and (S[j+1]=T[i-m+j]) do
                    j:=j+1;
                if j=m then {кінцева перевірка}
                    writeln('УРА');
            end;
        if i<=n then
            w:=(d*(w+P-(ord(T[i-m])*k mod P))+ord(T[i])) mod P;
    end.
```


Час роботи алгоритму ($m+n+mn/P$), mn/P досить невеликий, так що складність роботи майже лінійна. Зрозуміло, що просте число слід вибирати більшим, чим більше це число, тим швидше працюватиме програма. Цей алгоритм значно швидший від попереднього і цілком придатний для роботи з дуже довгими рядками.

8.15. АЛГОРИТМ БОУЕРА І МУРА

КМП-пошук дає справжній виграш тільки тоді, коли невдачі передувала деяка кількість співпадінь. Лише в цьому випадку слово зсувається більше ніж на одиницю. Але це швидше виняток, ніж правило: співпадіння зустрічаються значно рідше, ніж неспівпадіння. Тому виграш від використання КМП-стратегії в більшості випадків пошуку в звичних текстах вельми незначний. Метод, запропонований Р. Боуером і Д. Муром 1975 р., не тільки покращує опрацювання найгіршого випадку, але дає виграш у проміжних ситуаціях.

БМ-пошук ґрунтується на незвичайному міркуванні, v порівняння символів починається з кінця слова, а не з початку. Як і у випадку КМП-пошуку, слово перед фактичним пошуком трансформується в деяку таблицю. Нехай для кожного символа x з алфавіту величина $dx\ v$ – відстань від найправішого у слові входження x до правого кінця слова. Уявимо собі, що знайдено розбіжність між словом і текстом. У цьому випадку слово відразу ж можна зсунути вправо на $dx\ M$-1 позицій, тобто на кількість позицій, швидше за все більшу за одиницю. Якщо символ тексту, що не співпадає, в слові взагалі не зустрічається, то зсув стає навіть більшим, а саме зсувати можна на довжину всього слова.

Майже завжди, окрім спеціально побудованих прикладів, поданий алгоритм вимагає значно менше від N порівнянь. За найсприятливіших обставин, коли останній символ слова завжди потрапляє на символ тексту, що не співпадає, кількість порівнянь буде рівна N/M.

```
/* готуємося для зсуву поганих символів */
PRE_BC( char *x, int m, int bm_bc[] )
{
    int a, j;
    for ( a=0; a < ASIZE; a++ ) bm_bc[ a ] = m;
        for ( j=0; j < m-1;  j++ ) bm_bc[ (unsigned char)x[ j ] ] = m - j - 1;
}
/* готуємося до зсуву добрих символів вправо  */
PRE_GS( char *x, int m,  int bm_gs[] )
{
    int i, j, p, f[XSIZE];
    memset( bm_gs, 0, ( m + 1 ) * sizeof( int ) );
    f[ m ] = j = m + 1;
    for (i=m; i > 0; i— )
    {
        while ( j <= m && x[ i - 1 ] != x[ j - 1 ] )
        {
            if ( bm_gs[ j ] == 0 ) bm_gs[ j ] = j - i;
```

```
            j = f[ j ];
        }
        f[ i - 1 ] = —j;
    }
    p=f[ 0 ];
    for ( j=0; j <= m; ++j )
    {
        if ( bm_gs[ j ] == 0 )
            bm_gs[ j ] = p;
        if ( j == p )
            p = f[ p ];
    }
}
/* власне сам алгоритм*/
void BM( char *x, char *y, int n, int m )
{
    int i, j, bm_gs[XSIZE], bm_bc[ASIZE];
    PRE_GS( x, m, bm_gs );
    PRE_BC( x, m, bm_bc );
    i=0;
    while ( i <= n-m )
    {
        for ( j=m-1; j >= 0 && x[j] == y[ i+j ];  —j );
            if ( j < 0 )
            {
                OUTPUT(i);
                i += bm_gs[ j+1 ];
            }
            else i += MAX(( bm_gs[ j+1 ]), (bm_bc[ (unsigned char)y[ i+j ] ] - m + j+1));
    }
}
```

8.16. АЛГОРИТМ ХОРСПУЛА

Цей алгоритм – спрощення стандартного алгоритму Боуера-Мура.

1980 року Хорспул (Horspool) запропонував використовувати лише один зсув по найправішому символу для визначення зсуву в алгоритмі Боуера-Мура.

Отриманий алгоритм має квадратичну швидкість у гіршому випадку, але доведено, що середня кількість порівнянь на символ текста знаходиться між $1/|s|$ і $2/(|s|+1)$.

```
void HORSPOOL( char *y , char *x ,int n , int m )
{
    int a, i, j, bm_bc[ ASIZE ];
    char ch, lastch;
    /* попередні присвоєння */
    for ( a=0; a <ASIZE; a++ )
```



```
        bm_bc[ a ] = m;
    for ( j=0; j < m-1; j++ )
        bm_bc[ x[ j ] ] = m - j - 1;
    /* пошук */
    lastch = x[ m-1 ];
    i = 0;
    while ( i <= n-m )
    {
        ch = y[ i + m - 1 ];
        if ( ch == lastch )
            if ( memcmp( &y[ i ], x, m-1 ) == 0 )
                OUTPUT( i );
        i += bm_bc[ ch ];
    }
}
```

8.17. ПОРІВНЯННЯ МЕТОДІВ ПОШУКУ

Порівняння методів пошуку можна звести у таблицю 8.1.

Таблиця 8.1. Порівняння методів пошуку.

Алгоритм	Час на пребуд.	Середній час пошуку	Гіршний час пошуку	Витрати пам'яті
Алгоритм прямого пошуку рядка	Немає	2*n	Θ (n*m)	Немає
Алгоритм Рабіна-Карпа	Немає	Θ (n+m)	Θ (n*m)	Немає
Алгоритм Кнута-Моріса-Прата	Θ (m)	Θ (n+m)	Θ (n+m)	Θ (m)
Алгоритм Боуера-Мура	Θ (m+s)	Θ (n+m)	Θ (n*m)	Θ (m+s)
Алгоритм	Час на пребуд.	Середній час пошуку	Гіршний час пошуку	Витрати пам'яті
Алгоритм Боуера-Мура-Хорспула	Θ (m+s)	Θ (n+m)	Θ (n*m)	Θ (m+s)
Швидкий пошук	Θ (m+s)	Θ (n+m)	Θ (n*m)	Θ (m+s)
Алгоритм оптимальної розбіжності	Θ (m+s)	Θ (n+m)	Θ (n*m)	Θ (m+s)
Алгоритм максимального зрушення	Θ (m+s)	Θ (n+m)	Θ (n*m)	Θ (m+s)

Резюме

1. Усі методи можна розділити на статичні і динамічні. При статичному пошуку масив значень не змінюється під час роботи алгоритму. Під час динамічного пошуку масив може перебудовуватися або змінювати розмірність.

2. Для пошуку рядків існують спеціальні алгоритми: прямого пошуку, Ахо-Корасика, Кнута-Моріса-Прата, Рабіна-Карпа, Боуєра-Мура.

3. Серед алгоритмів пошуку чисел є: лінійний, бінарний, інтерполяційний пошук, пошук Фібоначчі, М-блоковий пошук тощо.

4. Методи обчислення адреси відносяться до методів хешування.

Контрольні запитання

1. Наведіть приклади використання алгоритмів пошуку.
2. Поясність особливості алгоритмів пошуку рядків.
3. Порівняйте алгоритми лінійного та бінарного пошуку.
4. Навіть алгоритми пошуку рядків.
5. Назвіть найефективніші методи пошуку рядків.

Тести для закріплення матеріалу

1. Перерахувати алгоритми пошуку
а) бульбашки;
б) лінійний;
в) ортогональний;
г) бінарний;
д) інтерполяційний.

2. Перерахувати алгоритми пошуку
а) бульбашки;
б) лінійний;
в) Боуера і Мура;
г) бінарний;
д) Кнута-Моріса-Прата.

3. Перерахувати алгоритми пошуку в стрічках
а) Боуера і Мура;
б) бінарний;
в) Кнута-Моріса-Прата;
г) прямий пошук;
д) паралельний пошук.

4. За фраґментом програми визначте метод пошуку:

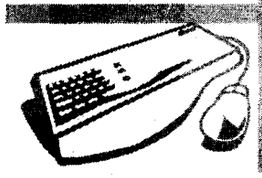
```
L:=0; R:=N;
while L<R do begin
    m:=(L+R) div 2; i:=0;
    while (T[m,i]=x[i]) and (x[i]<>00h) do i:=i+1;
```



```
    if T[m,i]<x[i] then L:=m+1 else R:=m
end;
if R<N then begin
    i:=0;
    while (T[R,i]=x[i]) and (x[i]<>00h) do i:=i+1
end
```

а) Боуера і Мура;
б) бінарний;
в) Кнута-Моріса-Прата;
г) прямий пошук;
д) паралельний пошук.

РОЗДІЛ 9



АЛГОРИТМИ СОРТУВАННЯ

- *Метод простого включення.*
- *Метод Шелла.*
- *Сортування шляхом підрахунку.*
- *Обмінне сортування.*
- *Сортування вибором.*
- *Сортування поділом (Хоара).*
- *Сортування за допомогою дерева.*
- *Пірамідальне сортування.*
- *Побудова піраміди методом Флойда.*
- *Сортування злиттям.*
- *Методи порозрядного сортування.*
- *Пряме злиття.*
- *Природне злиття.*
- *Збалансоване багатошляхове злиття.*
- *Багатофазне сортування*

Сортування застосовується в усіх без винятку областях програмування, хоч то бази даних чи математичні програми.

Практично кожен алгоритм сортування можна розбити на три частини: порівняння, що визначає впорядкованість пар елементів; перестановку, що змінює місцями пари елементів; власне алгоритм сортування, що здійснює порівняння і перестановку елементів доти, поки всі елементи множини не будуть упорядковані.

9.1. МЕТОДИ ВНУТРІШНЬОГО СОРТУВАННЯ

У загальній постановці завдання сортування ставиться таким чином. Є послідовність однотипних записів, одне з полів яких вибране як ключове (далі ми називатимемо його ключем сортування). Тип даних ключа повинен включати операції порівняння («=», «>», «<», «>=» і «<=»). Завданням сортування є перетворення початкової послідовності в послідовність, що містить ті самі записи, але у порядку зростання (або спадання) значень ключа. Метод сортування називається стійким, якщо при його застосуванні не змінюється відносне положення записів із рівними значеннями ключа.

Розрізняють сортування масивів записів, розташованих в основній пам'яті (внутрішнє сортування), і сортування файлів, що зберігаються в зовнішній пам'яті і не вміщуються повністю в основній пам'яті (зовнішнє сортування). Для внутрішнього і зовнішнього сортування потрібні істотно різні методи.

Природною умовою, що висувається до будь-якого методу внутрішнього сортування є те, що ці методи не повинні вимагати додаткової пам'яті: всі перестановки

з метою впорядкування елементів масиву мають вироблятися в межах того ж масиву. Мірою ефективності алгоритму внутрішнього сортування є кількість необхідних порівнянь значень ключа (C) і кількість перестановок елементів (M).

Зазначимо, що оскільки сортування засноване тільки на значеннях ключа і ніяк не зачіпає поля записів, що залишилися, можна говорити про сортування масивів ключів.

9.1.1. Метод простого включення

Нехай є масив ключів $a[1], a[2], \ldots, a[n]$. Для кожного елемента масиву, починаючи з другого, здійснюється порівняння з елементами з меншим індексом (елемент $a[i]$ послідовно порівнюється з елементами $a[i-1], a[i-2] \ldots$) і до тих пір, поки для чергового елемента $a[j]$ виконується співвідношення $a[j] > a[i]$, $a[i]$ і $a[j]$ міняються місцями. Якщо вдається зустріти такий елемент $a[j]$, що $a[j] <= a[i]$, або якщо досягнута нижня межа масиву, здійснюється перехід до опрацювання елемента $a[i+1]$ (поки не буде досягнута верхня межа масиву) (рис. 9.1).

Можна побачити, що в кращому разі (коли масив вже впорядкований) для виконання алгоритму з масивом із n елементів буде треба n-1 порівняння і 0 пересилань. У гіршому разі (коли масив впорядкований у зворотному порядку) буде треба $n*(n-1)/2$ порівнянь і стільки ж пересилань. Отже, можна оцінювати складність методу простих включень як $\Theta(n^2)$.

Можна скоротити кількість порівнянь, що використовуються в методі простих включень, якщо скористатися тим фактом, що при опрацюванні елемента масиву $a[i]$ елементи $a[1], a[2], \ldots, a[i-1]$ вже впорядковані, і скористатися для пошуку елементу, з яким має виконуватись перестановка, методом двійкового розподілу. У цьому випадку оцінка кількості необхідних порівнянь стає $\Theta(n \log n)$. Зазначимо, що оскільки при виконанні перестановки потрібне зсування на один елемент декількох елементів, то оцінка кількості пересилань залишається $\Theta(n^2)$.

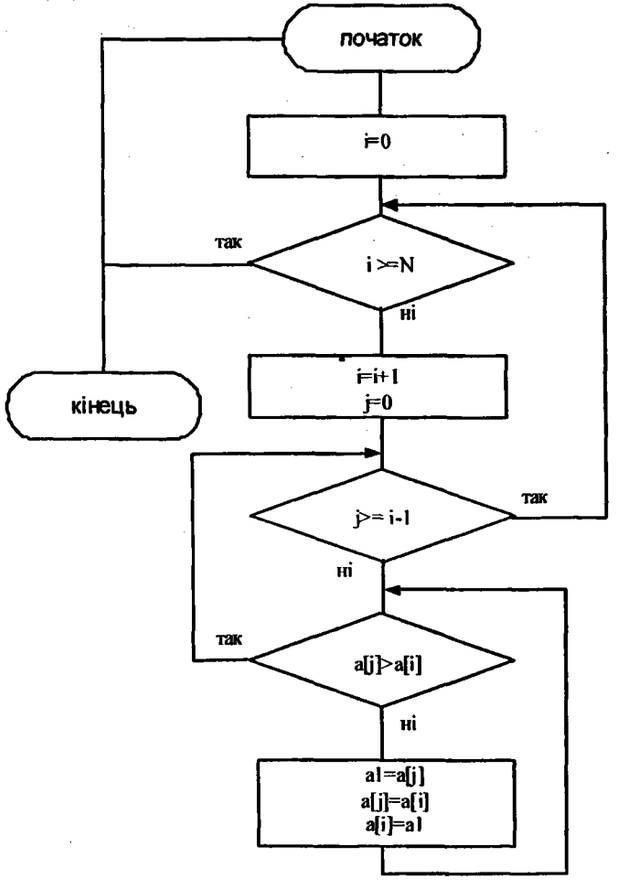


Рис. 9.1. Блок-схема сортування методом включення.

```
Void sort_vstavka()
{
    int i,j;   int a[50], a1;
    int k[50];
    i=0;
    while(i<50)
    {
        for (j=i-1; j>=0; j—)
        {
            if(a[j]>a[i])
            {
                a1=a[j];
                a[j]=a[i];
                a[i]=a1; i—;
            }
        }
        i++;
    }
}
```

Таблиця 9.1. Приклад сортування методом простого включення.

Початковий стан масиву	8 23 5 65 44 33 1 6
Крок 1	8 23 5 65 44 33 1 6
Крок 2	8 5 23 65 44 33 1 6
	5 8 23 65 44 33 1 6
Крок 3	5 8 23 65 44 33 1 6
Крок 4	5 8 23 44 65 33 1 6
Крок 5	5 8 23 44 33 65 1 6
	5 8 23 33 44 65 1 6
Крок 6	5 8 23 33 44 1 65 6
	5 8 23 33 1 44 65 6
	5 8 23 1 33 44 65 6
	5 8 1 23 33 44 65 6
	5 1 8 23 33 44 65 6
	1 5 8 23 33 44 65 6
Крок 7	1 5 8 23 33 44 6 65
	1 5 8 23 33 6 44 65
	1 5 8 23 6 33 44 65
	1 5 8 6 23 33 44 65
	1 5 6 8 23 33 44 65

9.1.3. Сортування шляхом підрахунку

Кожний елемент порівнюється зі всіма іншими; остаточно положення елементів визначаються після підрахунку кількості менших ключів.

Треба знайти перестановку $p(1)\ p(2)\ldots p(n)$, таку, що

$$Kp(1)<=Kp(2)<=\ldots<=Kp(n).$$

Метод заснований на тому що j-ключ впорядкованої послідовності перевищує рівно j-1 інших ключів. Ідея полягає в тому, щоб зрівняти попарно всі ключі і підрахувати, скільки з них менші від кожного окремого ключа.

Спосіб виконання поставленого завдання такий:

((порівняти Kj з Ki) при $1<=j<i$) при $1<i<=N$.

```
Void sort_pidr()
{
    int i,j;   int a[50], a1[50];
    int k[50];
    for (i=0; i<50; i++)
        k[i]=0;
    for (i=0; i<50-1; i++)
        for (j=i+1; j<50; j++)
            if(a[i]<a[j])
                k[i]++;
            else k[j]++;
    for (i=0; i<50; i++)
        a1[k[i]]=a[i];
    for (i=0; i<50; i++)
        a[i]=a1[i];
}
```

9.1.2. Метод Шелла

Подальшим розвитком методу сортування з включеннями є сортування методом Шелла, яке інакше називається сортуванням включеннями з відстанню, що зменшується.

Метод полягає в тому, що таблиця, яка впорядковується, розділяється на групи елементів, кожна з яких упорядковується методом простих включень. У процесі впорядкування розміри таких груп збільшуються доти, поки всі елементи таблиці не ввійдуть у впорядковану групу. Вибір чергової групи для сортування і її розташування всередині таблиці здійснюється так, щоб можна було використовувати попередню впорядкованість. Групою таблиці називають послідовність елементів, номери яких утворять арифметичну прогресію з різницею h (h називають кроком групи). На початку процесу впорядкування вибирається перший крок групи h_1, що залежить від розміру таблиці. Шелл запропонував брати

$$h_1=[n/2],\ \text{а}\ h_i=\text{h}((i\text{-}1)/2).$$

У більш пізніх роботах Хіббард показав, що для прискорення процесу доцільно визначити крок h_1 за формулою:

$$h_1=2^{**}k+1\ ,\ \text{де}\ 2^{**}k < n <= 2^{**}(k+1).$$

Після вибору h_1 методом простих включень впорядковуються групи, що містять елементи з номерами позицій $i, i+h_1, i+2*h_1, \ldots, i+m_i*h_1$.

При цьому $i=1,2,\ldots,h_1$; $m[i]$ – найбільше ціле, що задовольняє нерівність $i+m[i]*h_i <= n$.

Потім вибирається крок $h_2<h_1$ і впорядковуються групи, що містять елементи з номерами позицій $i, i+h_2, \ldots, i+m[i]*h_2$. Ця процедура з усе зменшуваними кроками продовжується доти, поки черговий крок $h[l]$ стане рівним одиниці ($h_1>h_2>\ldots>h_l$). Цей останній етап являє собою впорядкування всієї таблиці методом включень. Але оскільки вихідна таблиця впорядковувалася окремими групами з послідовним об'єднанням цих груп, то загальна кількість порівнянь значно менша, ніж n /4, необхідна при методі включень. Кількість операцій порівняння пропорційна $n*(\log_2(n))**2$.

Застосування методу Шелла до масиву, використаного в наших прикладах, показано в таблиці 9.2.

Таблиця 9.2. Приклад сортування методом Шелла.

Початковий стан масиву	8 23 5 65 44 33 1 6
Фаза 1 (сортуються елементи, відстань між якими чотирьом)	8 23 5 65 44 33 1 6
	8 23 5 65 44 33 1 6
	8 23 1 65 44 33 5 6
	8 23 1 6 44 33 5 65
Фаза 2 (сортуються елементи, відстань між якими двом)	1 23 8 6 44 33 5 65
	1 23 8 6 44 33 5 65
	1 23 8 6 5 33 44 65
	1 23 5 6 8 33 44 65
	1 6 5 23 8 33 44 65
	1 6 5 23 8 33 44 65
	1 6 5 23 8 33 44 65
Фаза 3 (сортуються елементи, відстань між якими одному)	1 6 5 23 8 33 44 65
	1 5 6 23 8 33 44 65
	1 5 6 23 8 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65

У загальному випадку алгоритм Шелла природно переформулюється для заданої послідовності з t відстаней між елементами $h_1, h_2, \ldots, h_t$, для яких виконуються умови $h_1 = 1$ і $h(i+1)< h_i$. При правильно підібраних t і h складність алгоритму Шелла є $\Theta(n(1.2))$, що істотно менша за складність простих алгоритмів сортування.

Блок-схема методу Шелла подана на рис. 9.2. У ньому використано змінні:

a[] – массив, який необхідно відсортувати,

t – допоміжна змінна того ж типу, що і масив,

n – розмірність масиву.

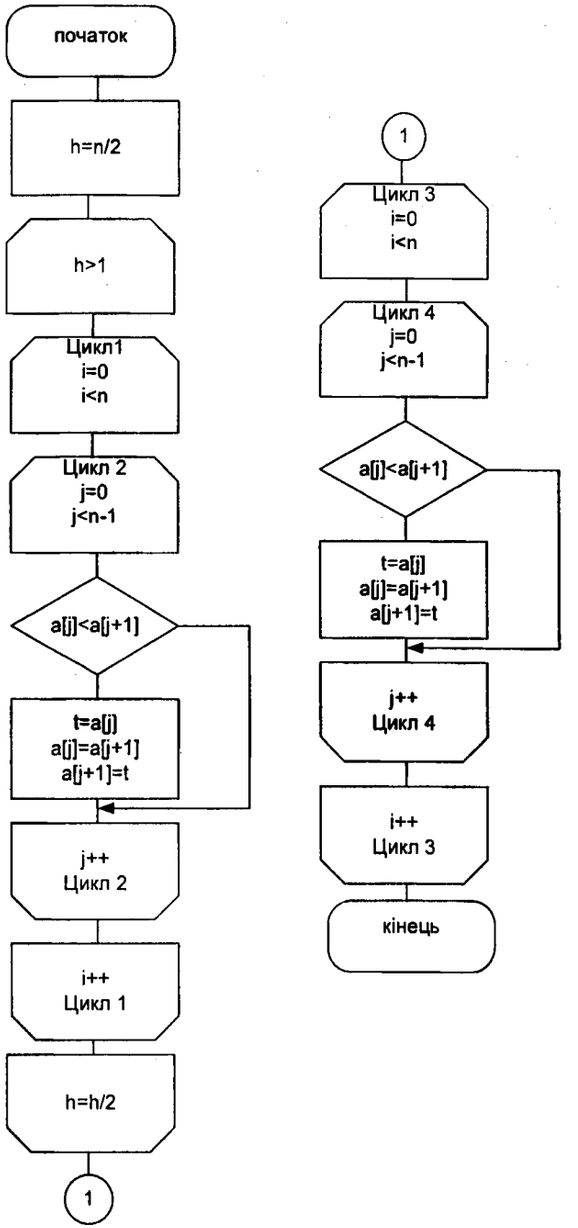


Рис. 9.2 Блок-схема методу Шелла.

9.1.4. Обмінне сортування

Просте обмінне сортування («методом бульбашки») для масиву a $a[1], a[2], \ldots, a[n]$ працює таким чином. Починаючи з кінця масиву порівнюються два сусідні елементи ($a[n]$ і $a[n-1]$). Якщо виконується умова $a[n-1]>a[n]$, то значення елементів міняються місцями. Процес продовжується для $a[n-1]$ і $a[n-2]$ і т.ін., поки не буде здійснено

порівняння $a[2]$ і $a[1]$. Зрозуміло, що після цього на місці $a[1]$ виявиться елемент масиву з якнайменшим значенням. На другому кроці процес повторюється, але останніми порівнюються $a[3]$ і $a[2]$. І так далі. На останньому кроці порівнюватимуться тільки поточні значення $a[n]$ і $a[n-1]$. Зрозуміла аналогія з бульбашкою, оскільки найменші елементи («найлегші») поступово «спливають» до верхньої межі масиву. Приклад сортування методом бульбашки показаний в таблиці 9.3.

Для методу простого обмінного сортування потрібна кількість порівнянь $nx(n-1)/2$, мінімальна кількість пересилань 0, а середня і максимальна кількість пересилань – $\Theta(n^2)$.

Блок-схема методу бульбашки поданана рис. 9.3.

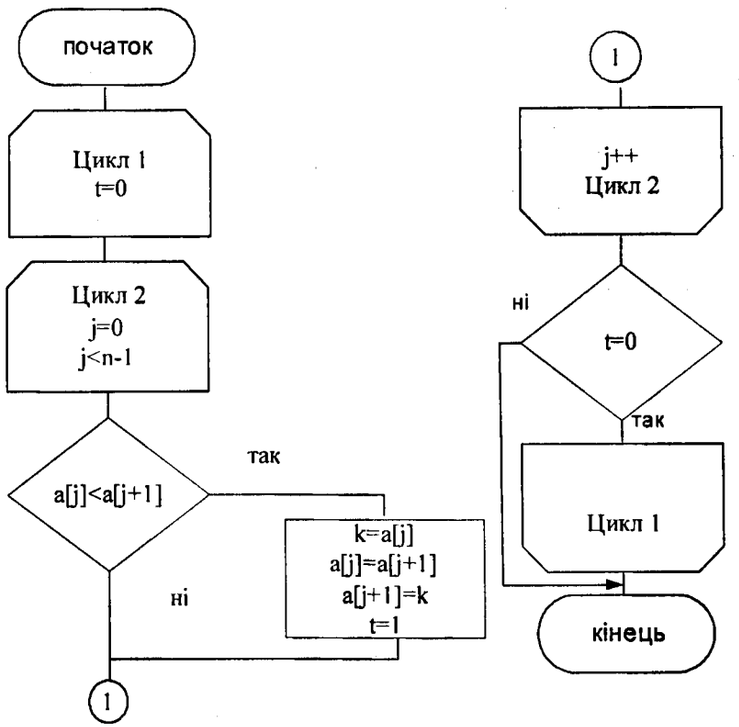


Рис. 9.3. Алгоритм сортування методом бульбашки.

Таблиця 9.3. Приклад сортування методом бульбашки.

Початковий стан масиву	8 23 5 65 44 33 1 6
Крок 1	8 23 5 65 44 33 1 6
	8 23 5 65 44 1 33 6
	8 23 5 65 1 44 33 6
	8 23 5 1 65 44 33 6
	8 23 1 5 65 44 33 6
	8 1 23 5 65 44 33 6
	1 8 23 5 65 44 33 6

Продовження таблиці 9.3. Приклад сортування методом бульбашки.

Крок 2	1 8 23 5 65 44 6 33
	1 8 23 5 65 6 44 33
	1 8 23 5 6 65 44 33
	1 8 23 5 6 65 44 33
	1 8 5 23 6 65 44 33
	1 5 8 23 6 65 44 33
Крок 3	1 5 8 23 6 65 33 44
	1 5 8 23 6 33 65 44
	1 5 8 23 6 33 65 44
	1 5 8 6 23 33 65 44
	1 5 6 8 23 33 65 44
Крок 4	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
Крок 5	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
Крок 6	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
Крок 7	1 5 6 8 23 33 44 65

Метод бульбашки допускає три прості вдосконалення. По-перше, як показує таблиця 9.3, на чотирьох останніх кроках розташування значень елементів не змінювалося (масив виявився вже впорядкованим). Тому, якщо на деякому кроці не було вироблено жодного обміну, то виконання алгоритму можна припиняти. По-друге, можна запам'ятовувати якнайменше значення індекса масиву, для якого на поточному кроці виконувалися перестановки.

Очевидно, що верхня частина масиву до елемента із цим індексом вже відсортована, і на наступному кроці можна припиняти порівняння значень сусідніх елементів, досягши такого значення індекса. По-третє, метод бульбашки працює нерівноправно для «легких» і «важких» значень. Легке значення потрапляє на потрібне місце за один крок, а важке на кожному кроці опускається у напрямку до требаго місця на одну позицію.

На цих спостереженнях заснований метод **шейкерного сортування** (ShakerSort). При його застосуванні на кожному наступному кроці змінюється напрямок послідовного перегляду. У результаті, на одному кроці «спливає» черговий найлегший елемент, а на іншому «тоне» черговий найважчий. Приклад шейкерного сортування наведений у таблиці 9.4.

Шейкерне сортування дозволяє скоротити кількість порівнянь (за оцінкою Кнута середньою кількістю порівнянь є $(n2 - n(\text{const} + \ln n))$, хоча порядком оцінки як і раніше

залишається $n2$. Кількість посилань загалом не ...ісці $a[$...ється. Шейкерне сортування рекомендується використовувати в тих випадк...цес пс...и відомо, що масив «майже впорядкований».

...у кроці ...з більє...

Таблиця 9.4. Приклад шейкер...о верх...ортування.

Початковий стан масиву	8 2... ...4 33 1 6
Крок 1	8 2... ...44 33 1 6
	8 2... ...44 1 33 6
	8 2... ...1 44 33 6
	8 2... ...5 44 33 6
	8 2... ...5 44 33 6
	8 1... ...5 44 33 6
	1 8... ...5 44 33 6
Крок 2	1 8... ...5 44 33 6
	1 8... ...5 44 33 6
	1 8... ...5 44 33 6
	1 8... ...4 65 33 6
	1 8... ...4 33 65 6
	1 8... ...4 33 6 65
Крок 3	1 8... ...ні 6 33 65
	1 8... ...44 33 65
	1 8... ...44 33 65
	1 8... ...44 33 65
	1 5... ...44 33 65
Крок 4	1 5... ...4 33 65
	1 5... ...4 33 65
	1 5... ...4 33 65
	1 5... ...3 44 65
Крок 5	1 5... ...3 44 65
	1 5... ...3 44 65
	1 5... ...3 44 65

...9.3.

...рібна к...ксимал...

9.1.5. Сортування ...етодом

При сортуванні масиву $a[1]$, $a[2]$, ..., $a[n]$ м... простого вибору серед всіх елементів знаходиться елемент з якнайменшим зна...метод... $a[i]$, і $a[1]$ і $a[i]$ обмінюються значеннями. Потім цей процес повторюєть...44 33 1 одержуваних підмасивів $a[2]$, $a[3]$, ..., $a[n]$ ... $a[j]$, $a[j+1]$, ..., $a[n]$ до тих п...44 33 ...ми не дійдемо до підмасиву $a[n]$, що містить до цього моменту найбільше значе...44 1 ...бота алгоритму ілюструється прикладом в таблиці 9.5.

...1 44 3...

Для методу сортування простим вибором необ...65 44 ...кількість порівнянь – $n(n-1)/2$. Порядок необхідної кількості посилань (включ...65 44 ...ті, які потрібні для вибору мінімального елемента) у гіршому разі складає...65 44 ...). Проте порядок середньої кількості пересилань є ($n \ln n$), що у ряді випадків ро...65 44 ...ей метод одним із найкращих.

Таблиця 9.5. Приклад сортування простим вибором.

Початковий стан масиву	8 23 5 65 44 33 1 6
Крок 1	1 23 5 65 44 33 8 6
Крок 2	1 5 23 65 44 33 8 6
Крок 3	1 5 6 65 44 33 8 23
Крок 4	1 5 6 8 44 33 65 23
Крок 5	1 5 6 8 33 44 65 23
Крок 6	1 5 6 8 23 44 65 33
Крок 7	1 5 6 8 23 33 65 44
Крок 8	1 5 6 8 23 33 44 65

Функція сортування вибором виглядає так:

```
void sort_vyb()
{
    int i,j a[10],t;
    for (i=0; i<10-1; i++)
        for (j=i+1; j<10; j++)
            if(a[i]<a[j]);
            {
                t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
}
```

9.1.6. Сортування поділом (Хоара)

Метод сортування поділом був запропонований Чарльзом Хоаром 1962 року. Цей метод є розвитком методу простого обміну і настільки ефективний, що його почали називати «методом швидкого сортування – Quicksort».

Основна ідея алгоритму полягає в тому, що випадковим чином вибирається деякий елемент масиву x, після чого масив є видимим зліва, поки не зустрінеться елемент $a[i]$ такий, що $a[i]>x$, а потім масив є видимим справа, поки не зустрінеться елемент $a[j]$ такий, що $a[j]<x$. Ці два елементи міняються місцями, і процес перегляду, порівняння і обміну продовжується, поки ми не дійдемо до елемента x. У результаті масив виявиться розбитим на дві частини – ліву, в якій значення ключів будуть менші від x, і праву із значеннями ключів, більшими від x. Далі процес рекурсивно продовжується для лівої і правої частин масиву до тих пір, поки кожна частина не міститиме лише один елемент (рис. 9.4). Зрозуміло, що, як завжди, рекурсію можна замінити ітераціями, якщо запам'ятовувати відповідні індекси масиву. Прослідкуємо цей процес на прикладі нашого стандартного масиву (таблиця 9.6).

Таблиця 9.6. Приклад швидкого сортування.

Початковий стан масиву	8 23 5 65 |44| 33 1 6
Крок 1 (як x вибирається a[5])	8 23 5 6 44 33 1 65
	8 23 5 6 1 33 44 65
Крок 2 (у підмасиві a[1], a[5] як x вибирається a[3])	8 23 |5| 6 1 33 44 65
	1 23 5 6 8 33 44 65
	1 5 23 6 8 33 44 65
Крок 3 (у підмасиві a[3], a[5] як x вибирається a[4])	1 5 23 |6| 8 33 44 65
	1 5 8 6 23 33 44 65
Крок 4 (у підмасивах a[3], a[4] вибирається a[4])	1 5 8 |6| 23 33 44 65
	1 5 6 8 23 33 44 65

Алгоритм недаремно називається швидким сортуванням, оскільки для нього оцінкою кількості порівнянь і обмінів є $\Theta(n \log n)$. Насправді, в більшості утиліт, що виконують сортування масивів, використовується саме цей алгоритм.

9.1.7. Сортування за допомогою дерева

Почнемо з простого методу сортування за допомогою дерева, при використанні якого будується двійкове дерево порівняння ключів. Побудова дерева починається з листя, яке містить всі елементи масиву. З кожної сусідньої пари вибирається найменший елемент, і ці елементи

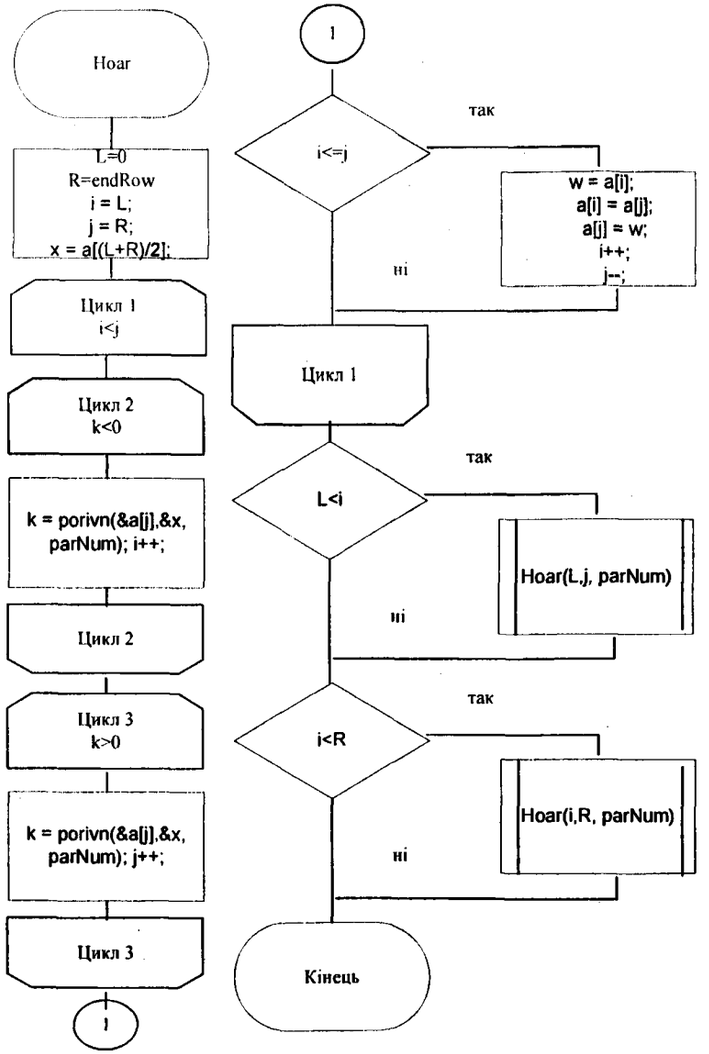


Рис. 9.4. Блок-схема рекурсивного методу Хоара

утворюють наступний (ближчий до кореня рівень дерева). Із кожної сусідньої пари вибирається найменший елемент і т.ін., поки не буде побудований корінь, тобто найменший елемент масиву.

Приклад двійкового дерева показано на рис. 9.5.

Отже, ми вже маємо якнайменше значення елементів масиву. Щоб одержати наступний по величині елемент, спустимося від коріння по шляху, що веде до листка з якнайменшим значенням.

У цій листковій вершині ставиться фіктивний ключ з «нескінченно великим» значенням, а у всі проміжні вузли, що займалися якнайменшим елементом, вноситься якнайменше значення з вузлів – безпосередніх нащадків (рис. 9.6). Процес продовжується доти, поки всі вузли дерева не будуть заповнені фіктивними ключами (рисунки 9.8 – 9.12).

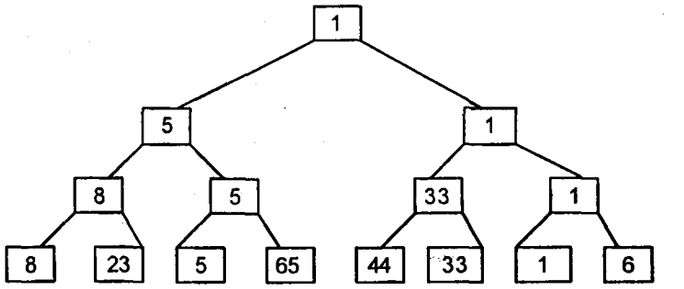


Рис. 9.5. Перший крок.

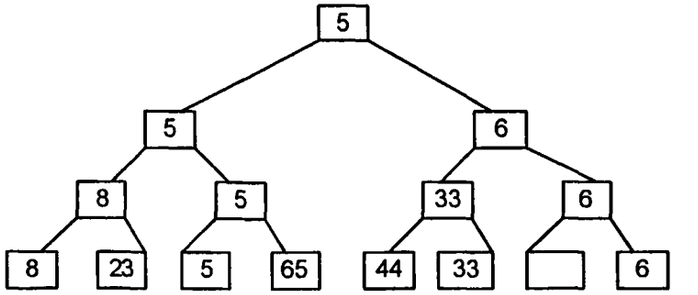


Рис. 9.6. Другий крок.

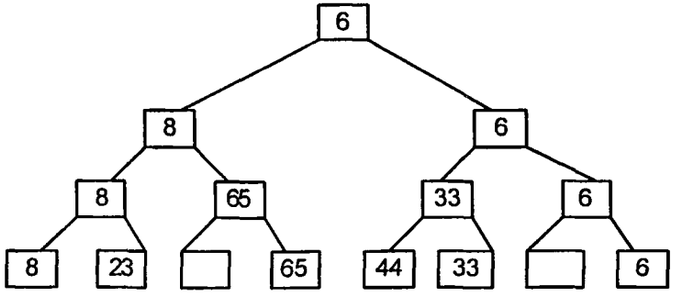


Рис. 9.7. Третій крок.

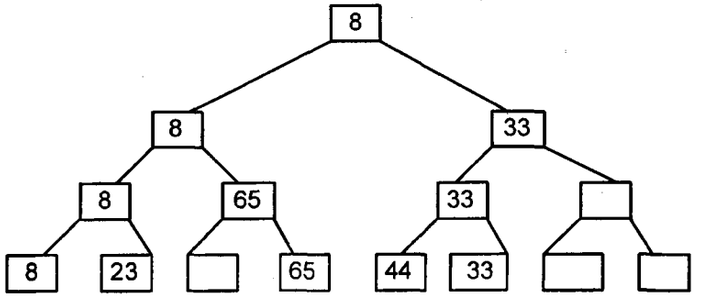


Рис. 9.8. Четвертий крок.

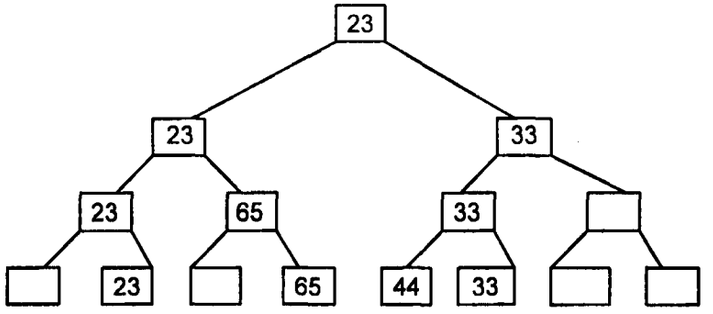


Рис. 9.9. Пятий крок.

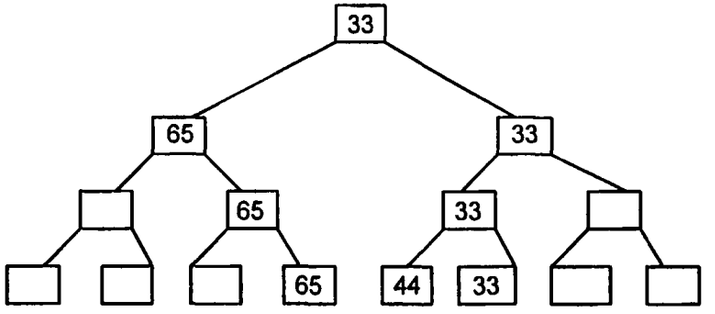


Рис. 9.10. Шостий крок.

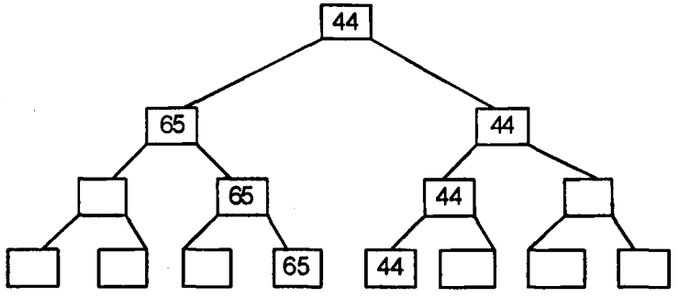


Рис. 9.11. Сьомий крок.

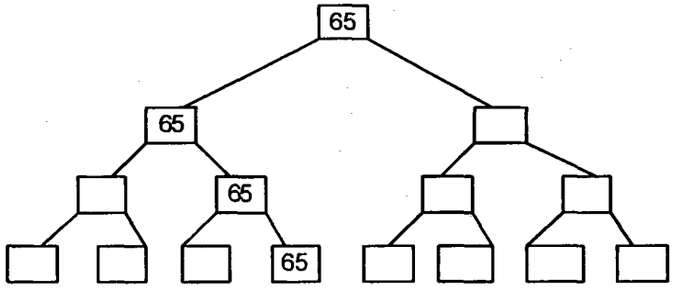


Рис. 9.12. Восьмий крок.

На кожному з n кроків, що необхідні для сортування масиву, треба виконати $\log_2 n$ порівнянь. Отже, всього буде треба $n*\log_2 n$ порівнянь, але для подання дерева знадобиться $2n - 1$ додаткова одиниця пам'яті.

9.1.8. Пірамідальне сортування

Є досконаліший алгоритм, який прийнято називати пірамідальним сортуванням (Heapsort). Його ідея полягає у тому, що замість повного дерева порівняння початковий масив $a[1], a[2], \ldots, a[n]$ перетвориться на піраміду, що має таку властивість, що для кожного $a[i]$ виконуються умови $a[i] <= a[2i]$ і a[i]<= a[2i+1]. Потім піраміда використовується для сортування.

Найнаочніше метод побудови піраміди виглядає при деревовидному зображенні масиву, показаному на рис. 9.13. Масив подається у вигляді двійкового дерева, корінь якого відповідає елементу масиву $a[1]$. На другому ярусі знаходяться елементи $a[2]$ і $a[3]$. На третьому – $a[4], a[5], a[6], a[7]$ і т.ін. Як видно, для масиву з непарною кількістю елементів відповідне дерево буде збалансованим, а для масиву з парною кількістю елементів n елемент $a[n]$ буде єдиним (найлівішим) листком «майже» збалансованого дерева.

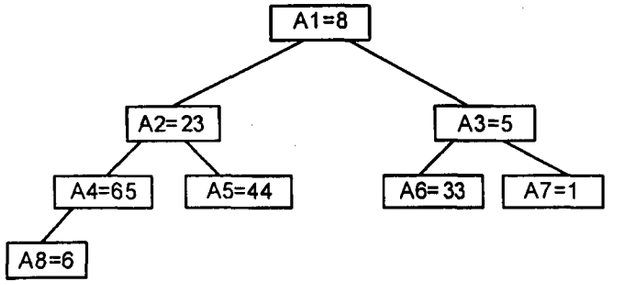


Рис. 9.13. Приклад пірамідального сортування.

Очевидно, що при побудові піраміди нас цікавитимуть елементи $a[n/2], a[n/2-1], \ldots, a[1]$ для масивів з парною кількістю елементів і елементи $a[(n-1)/2], a[(n-1)/2-1], \ldots, a[1]$ для масивів з непарною кількістю елементів (оскільки тільки для таких елементів істотні обмеження піраміди). Нехай i – найбільший індекс

з індексів елементів, для яких існують обмеження піраміди. Тоді береться елемент $a[i]$ у побудованому дереві і для нього виконується процедура просівання, що полягає у тому, що вибирається гілка дерева, яка відповідає $\min(a[2i], a[2i+1])$, і значення $a[i]$ міняється місцями із значенням відповідного елементу. Якщо цей елемент не є листком дерева, для нього виконується аналогічна процедура і т.ін. Такі дії виконуються послідовно для $a[i]$, $a[i-1]$, …, $a[1]$. Як бачимо, в результаті ми одержимо деревовидне подання піраміди для початкового масиву (послідовність кроків для використовуваного в наших прикладах масиву показана на рис. 9.14 – 9.17).

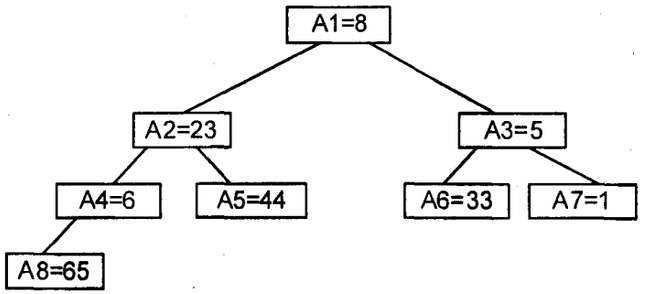


Рис. 9.14. Пірамідальне сортування. Крок 1.

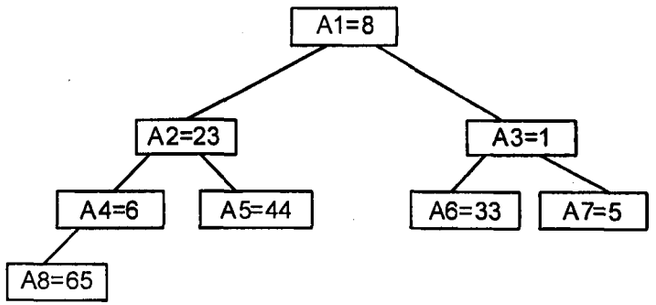


Рис. 9.15. Пірамідальне сортування. Крок 2.

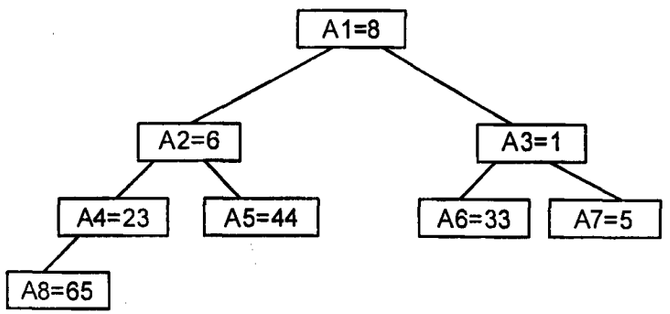


Рис. 9.16. Пірамідальне сортування. Крок 3.

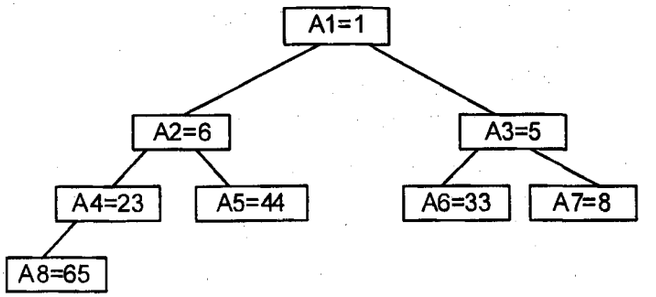


Рис. 9.17. Пірамідальне сортування. Крок 4.

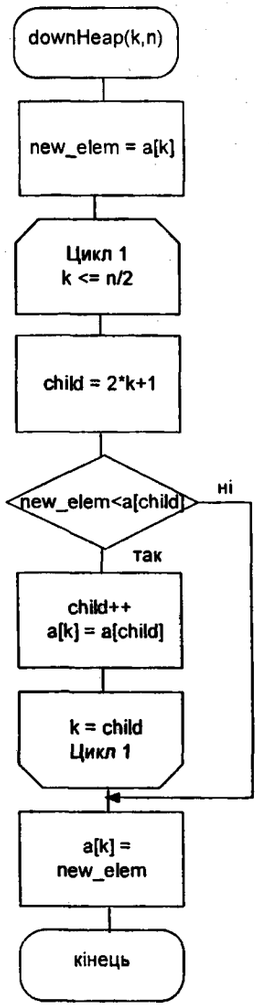


Рис. 9.18. Блок-схема методу впорядкування піраміди.

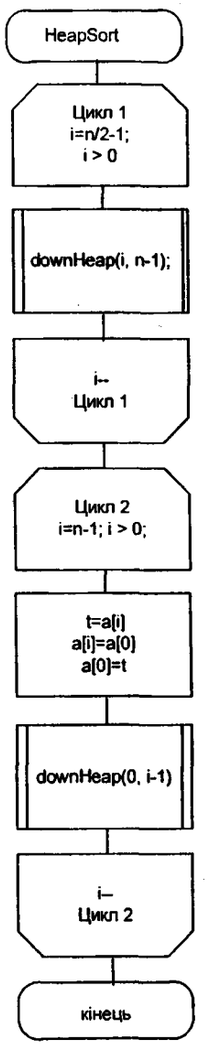


Рис. 9.19. Блок-схема побудови піраміди та її перевірки.

9.1.9. Побудова піраміди методом Флойда

1964 року Флойд запропонував метод побудови піраміди без явної побудови дерева (хоча метод заснований на тих самих ідеях). Побудова піраміди методом Флойда для нашого стандартного масиву показана в таблиці 9.7.

Таблиця 9.7 Приклад побудови піраміди.

Початковий стан масиву	8 23 5 |65| 44 33 1 6
Крок 1	8 23 |5| 6 44 33 1 65
Крок 2	8 |23| 1 6 44 33 5 65
Крок 3	|8| 6 1 23 44 33 5 65
Крок 4	1 6 8 23 44 33 5 65
	1 6 5 23 44 33 8 65

У таблиці 9.8 показано, як здійснюється сортування з використанням побудованої піраміди. Суть алгоритму полягає в подальшому. Нехай i – найбільший індекс масиву, для якого вказані умови піраміди. Тоді, починаючи з $a[1]$ до $a[i]$ виконуються такі дії.

Таблиця 9.8 Сортування за допомогою піраміди.

Початкова піраміда	1 6 5 23 44 33 8 65
Крок 1	65 6 5 23 44 33 8 1
	5 6 65 23 44 33 8 1
	5 6 8 23 44 33 65 1
Крок 2	65 6 8 23 44 33 5 1
	6 65 8 23 44 33 5 1
	6 23 8 65 44 33 5 1
Крок 3	33 23 8 65 44 6 5 1
	8 23 33 65 44 6 5 1
Крок 4	44 23 33 65 8 6 5 1
	23 44 33 65 8 6 5 1
Крок 5	65 44 33 23 8 6 5 1
	33 44 65 23 8 6 5 1
Крок 6	65 44 33 23 8 6 5 1
	44 65 33 23 8 6 5 1
Крок 7	65 44 33 23 8 6 5 1

На кожному кроці вибирається останній елемент піраміди (у нашому випадку першим буде вибраний елемент $a[8]$). Його значення міняється зі значенням $a[1]$, після чого для $a[1]$ виконується просіювання. При цьому на кожному кроці кількість елементів в піраміді зменшується на 1 (після першого кроку як елементи піраміди розглядаються $a[1]$, $a[2]$, …, $a[n\text{-}1]$; після другого – $a[1]$, $a[2]$, …, $a[n\text{-}2]$ і т.ін., поки в піраміді не залишиться один елемент). Легко побачити (це ілюструється в таблиці 9.8), що як результат ми одержимо масив, впорядкований у порядку спадання. Можна модифікувати метод побудови піраміди і сортування, щоб одержати впорядкування у порядку зростання, якщо змінити умову піраміди $a[i]>=a[2i]$ і $a[1]>=a[2i+1]$ для всіх значень індекса i.

Процедура сортування з використанням піраміди вимагає виконання порядку $n\log_2$

n кроків у гіршому разі, що робить її особливо привабливою для сортування великих масивів.

9.1.10. Сортування злиттям

Сортування злиттям, як правило, застосовуються в тих випадках, коли вимагається відсортувати послідовний файл, що не вміщаються в основній пам'яті.

Нехай є два відсортованих у порядку зростання масиви $p[1]$, $p[2]$, …, $p[n]$ і $q[1]$, $q[2]$, …, $q[n]$ і є порожній масив $r[1]$, $r[2]$, …, $r[2*n]$, який ми хочемо заповнити значеннями масивів p і q у порядку зростання. Для злиття виконуються такі дії: порівнюються $p[1]$ і $q[1]$, і менше зі значень записується в $r[1]$. Припустимо, що це значення p[1]. Тоді $p[2]$ порівнюється з $q[1]$, і менше із значень записується в $r[2]$. Припустимо, що це значення $q[1]$. Тоді на наступному кроці порівнюються значення $p[2]$ і $q[2]$ і т.ін., поки ми не досягнемо меж одного з масивів. Тоді залишок іншого масиву просто дописуванняється в «хвіст» масиву r.

Алгоритм сортуванням злиттям подано на рис. 9.20.

Для випадку масиву, що використовується в наших прикладах, послідовність кроків показана в таблиці 9.9.

При застосуванні сортування зі злиттям кількість порівнянь ключів і кількість пересилань оцінюється як $\Theta(n*\log n)$. Але слід врахо-вувати, що для виконання алгоритму для сортування масиву розміру n потрібно $2n$ елементів пам'яті.

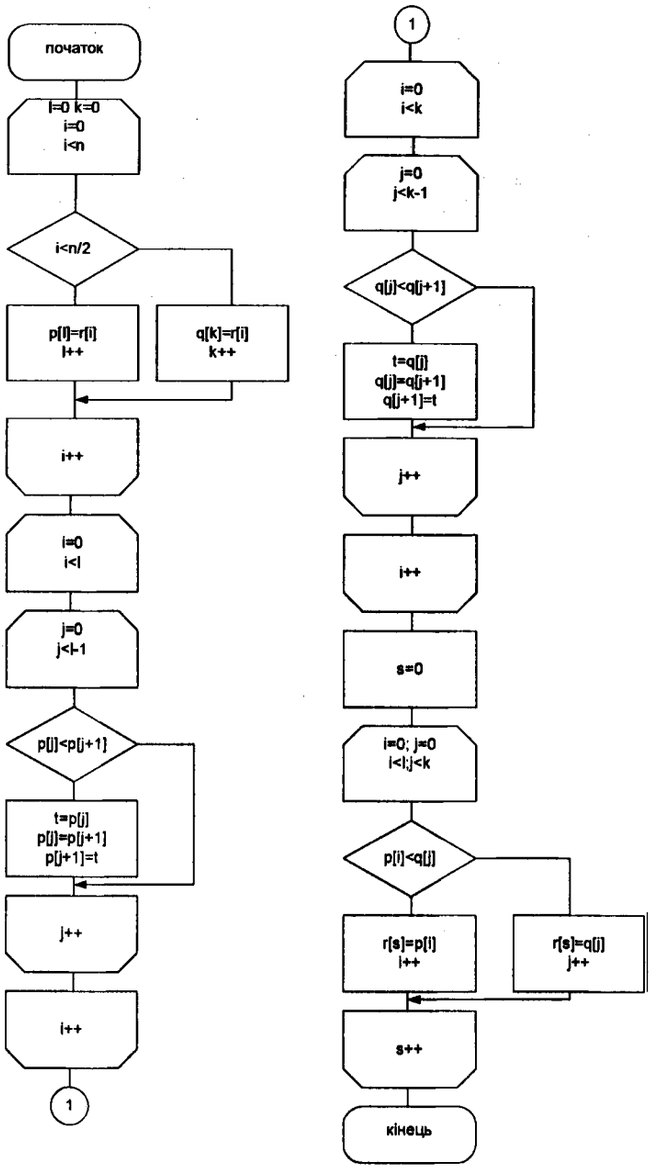


Рис. 9.20. Блок-схема методу сортування злиттям.

Таблиця 9.9. Приклад сортування злиттям.

Початковий стан масиву	8 23 5 65 44 33 1 6
Крок 1	6 8 1 23 5 33 44 65
Крок 2	6 8 44 65 1 5 23 33
Крок 3	1 5 6 8 23 33 44 65

```
Void sort_bin()
{
    int i,j,k=0,l=0;int t=(int)n/2+n%2;
    int resCmp=0;
    student a1[30],a2[30],tmp;
    for(i=0;i<n;i++)
    {
        if(i<(int)n/2)
            a1[i]=a[i];
        else
            a2[i-(int)n/2]=a[i];
    }
    for(i=0;i<(int)n/2;i++)
        for(j=0;j<(int)n/2-1;j++)
        {
            if(a1[j]>a1[j+1])
            {
                tmp=a1[j];
                a1[j]=a1[j+1];
                a1[j+1]=tmp;
            }
        }
    for(i=0;i<(int)n/2+n%2;i++)
        for(j=0;j<(int)n/2+n%2-1;j++)
        {
            if(a2[j]>a2[j+1])
            {
                tmp=a2[j];
                a2[j]=a2[j+1];
                a2[j+1]=tmp;
            }
    }
    i=0;
    while(i<n)
    {
        if ((k==(int)n/2)&&(l!=t))
        {
            a[i]=a2[l];
            i++;l++;continue;
```



```
        }
        if ((k!=(int)n/2)&&(l==t))
        {
            a[i]=a1[k];
            i++;k++;continue;
        }
        if(a2[l]>a1[k])
        {
            a[i]=a1[k];
            i++;k++;continue;
        }
        if(a1[k]>a2[l])
        {
            a[i]=a2[l];
            i++;l++; continue;
        }
        if(a1[k]==a2[l])
        {
            a[i]=a1[k];
            i++;k++;
            a[i]=a2[l];
            i++;l++; continue;
        }
    }
```

9.1.11. Методи порозрядного сортування

Розглянутий нижче алгоритм істотно відрізняється від описаних раніше.

По-перше, він зовсім не використовує порівнянь сортованих елементів.

По-друге, ключ, за яким відбувається сортування, необхідно розділити на частини, *розряди* ключа. Наприклад, слово можна розділити на літери, число – на цифри.

До сортування необхідно знати два параметри: k і m, де k – кількість розрядів у найдовшому ключі, m – розрядність даних: кількість можливих значень розряду ключа.

При сортуванні українських слів $m = 33$, тому що літера може приймати не більше як 33 значення. Якщо в найдовшому слові 10 літер, $k = 10$.

Аналогічно, для шістнадцяткових чисел m=16, якщо як розряд брати цифру, і m=256, якщо використовувати побайтовий поділ.

Ці параметри не можна змінювати в процесі роботи алгоритму. У цьому – ще одна відмінність методу від вищеописаних.

Порозрядне сортування для списків

Припустимо, що елементами лінійного списку L є k-розрядні десяткові числа, розрядність максимального числа відома заздалегідь. Позначимо $d(j,n)$ – j-а права цифра числа n, яку можна виразити як $d(j,n) = [\, n / 10^{j-1} \,] \bmod 10$.

Нехай $L_0, L_1, \ldots, L_9$ – допоміжні списки (кишені), спочатку порожні. Поразрядне сортування складається з двох процесів, які називаються *розподіл* і *збирання,* і виконуються для j=1,2,…,k.

Фаза розподілу розносить елементи L по кишенях: елементи l_i списку L послідовно додаються в списки L_m, де $m = d(j, l_i)$. Таким чином одержуємо десять списків, у кожному з який j-ті розряди чисел однакові і рівні m.

Фаза збирання полягає в об'єднанні списків $L_0, L_1, \ldots, L_9$ у загальний список

$$L = L_0 => L_1 => L_2 => \ldots => L_9.$$

Розглянемо приклад роботи алгоритму на вхідному списку

0 => 8 => 12 => 56 => 7 => 26 => 44 => 97 => 2 => 37 => 4 => 3 => 3 => 45 => 10.

Максимальне число містить дві цифри, отже, розрядність даних k=2.

Перше проходження, j=1.

Розподіл за першою праворуч цифрою:

L_0:0 => 10

// усі числа з першою правою

//цифрою 0

L_1: порожньо

L_2:12 => 2

L_3:3 => 3

L_4:44 => 4

L_5:45

L_6:56 => 26

L_7:7 => 97 => 37

L_8:8

L_9: порожньо

// усі числа с першою правою

//цифрою 9

Збирання:

з'єднуємо списки L_i один за одним

L: 0 => 10 => 12 => 2 => 3 => 3 => 44 => 4 => 45 => 56 => 26 => 7 => 97 => 37 => 8

Друге проходження, j=2.

Розподіл за другою цифрою справа:

L_0:0 => 2 => 3 => 3 => 4 => 7 => 8

L_1:10 => 12

L_2:26

L_3:37

L_4:44 => 45

L_5:56

L_6: порожньо

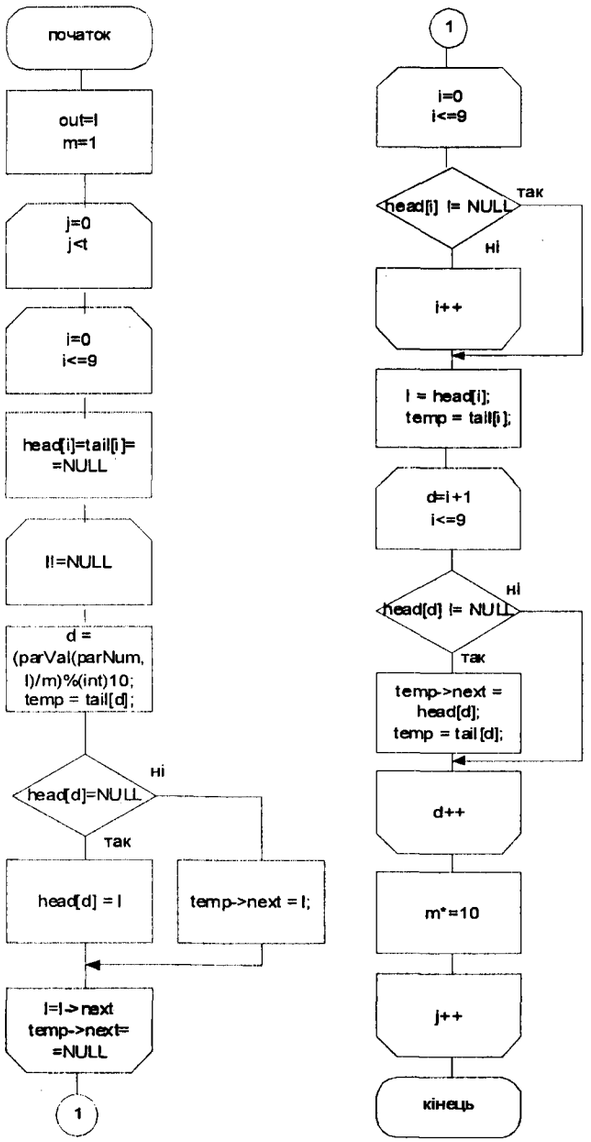


Рис. 9.21. Алгоритм порозрядного сортування.

L_7: порожньо

L_8: порожньо

L_9:97

Збирання:

з'єднуємо списки L_i один за одним

L: 0 => 2 => 3 => 3 => 4 => 7 => 8 => 10 => 12 => 26 => 37 => 44 => 45 => 56 => 97

Кількість використаних кишень – кількість можливих значень елемента списку.

```
//структура для кишень, подана у вигляді списку
struct spys   { student val;
    struct  spys *next;} *head,*tail;
void FirstMinMax()
//пошук першого елемента масиву з метою пришвидшення пошуку
{
    int i,j;    int tmp;
    for(i=1;i<n;i++)
        if(a[0]<a[i])
        {
            tmp=a[0];
            a[0]=a[i];
            a[i]=tmp;
        }
}
void toSpys()
//перетворюємо масив у список
{
    struct spys *q;int i;
    for(i=0; i<n; i++)
    {
        q=(struct spys*) malloc(sizeof(struct spys)); //формуємо список
        q->val=a[i];
        if(tail!=NULL)
        tail->next=q;
        else head=q;
        tail=q;
    }
    tail->next=NULL;
}
void toArray(struct spys *head)
//перетворюємо список у масив
{
    int i=0;
    while (head)
    {
        a[i]=head->val;head=head->next;i++;
    }
```

```
}
struct spys *bin_s_as(struct spys *l, int t)
//порозрядне сортування
{
    int i, j, d, m=1;
    //список кишень
    struct spys *h[10]={NULL}, *tail[10]={NULL}, *out,*temp,*mm;
    out=l;
    for (j=1; j<=t; j++)
    {
        for (i=0; i<=9; i++)
        //онулили кишеньки
        h[i] = (tail[i]=NULL);
        while ( l != NULL )
        {
            //розраховуємо,який розряд опрацьовується зараз
            d = (l->val.oz/m)%(int)10;
            temp = tail[d];
            //додаємо у кишеню
            if ( h[d]==NULL ) h[d] = l;
            else temp->next = l;
            temp = tail[d] = l;
            l = l->next;
            temp->next = NULL;
        }
        for (i=0; i<=9; i++)
            if ( h[i] != NULL ) break;
        l = h[i];
        temp = tail[i];
        for (d=i+1; d<=9; d++)
        {
            if ( h[d] != NULL)
            {
                temp->next = h[d];
                temp = tail[d];
            }
        }
        //переходимо до аналізу наступного розряду чисел масиву
        m*=10;
        mm=out;
        //список перетворюємо у масив
        toArray(mm);
    }
    return (out);
}
```


Сортування можна організувати таким чином, щоб не використовувати додаткової пам'яті для кишень, тобто елементи списку не переміщати, а за допомогою перестановки вказівників приєднувати їх до тієї чи іншої кишені.

Порозрядне сортування для масивів

Списки досить зручні тим, що їх легко реорганізовувати, поєднувати і т.ін. Як застосувати таку ідею для масивів?

Нехай у нас є масив *source* з *n* десяткових цифр ($m = 10$).

Наприклад, *source*[7] = { 7, 9, 8, 5, 4, 7, 7 }, *n*=7. Тут покладемо const *k*=1.

1.Створити масив *count* з *m* елементів(лічильників).

2.Присвоїти кількість елементів *source*, рівних *i*. Для цього:

проініціалізувати *count*[] нулями,

пройти по *source* від початку до кінця, для кожного числа збільшуючи елемент *count* з відповідним номером.

```
for( i=0; i<n; i++)   count [ source[i] ]++
```

У нашому прикладі count[] = { 0, 0, 0, 0, 1, 1, 0, 3, 1, 1 }.

3.Присвоїти *count*[*i*] значення, рівні сумі всіх елементів, розміщених до нього:

count[i] = count[0]+count[1]+…count[i-1].

У нашому прикладі count[] = { 0, 0, 0, 0, 1, 2, 2, 2, 5, 6 }.

Ця сума є кількістю чисел вихідного масиву, менших від *i*.

4.Зробити остаточне розміщення.

Для кожного числа *source*[*i*] ми знаємо, скільки чисел є менші від нього – це значення зберігається в *count*[*source*[*i*]]. Отже, нам відомо остаточне місце числа в упорядкованому масиві: якщо є *K* чисел менших від заданого, то *source*[*i*] повинне стояти на позиції *K*+1.

Здійснюємо проходження по масиву *source* зліва направо, одночасно заповнюючи вихідний масив *dest*:

```
for ( i=0; i<n; i++ ) {
  c = source[i];   dest[ count[c] ] = c;
  count[c]++;  // для повторюваних чисел
}
```

Отже, число *c*=*source*[*i*] ставиться на місце *count*[*c*]. На той випадок, якщо числа повторюються в масиві, передбачений оператор *count*[*c*]++, що збільшує значення позиції для наступного числа *c*, якщо таке буде.

Цикли займають ($n + m$) часу. Стільки ж треба пам'яті.

Отже, ми навчилися за ($n+m$) сортувати цифри. Аналогічно продемонструємо сортування для рядків (чисел). Нехай у нас в кожнім ключі *k* цифр ($m = 10$). Аналогічно до випадку зі списками відсортуємо їх у кілька проходів від молодшого розряду до старшого.

Вихідна k=3 послідовність	Перше проходження по 3-му розряду	Друге проходження по 2-му розряду	Третє проходження по 1-му розряду
523	523	523	088
153	153	235	153
088	554	153	235
554	235	554	523
235	088	088	554

Отже, загальна кількість операцій рівна $k(n+m)$, при використовуваній додатково пам'яті $(n+m)$. Ця схема допускає невелику оптимізацію. Зазначимо, що сортування по кожоіму байту складається з 2 проходів по всьому масиві: на першому кроці і на четвертому. Однак, можна створити відразу всі масиви *count*[] (по одному на кожну позицію) за одне проходженя.

Отже, перший крок буде виконуватися один раз за все сортування, а тому, загальна кількість проходжень зміниться з $2k$ на $k+1$.

Ефективність порозрядного сортування

Ріст швидкого сортування: $\Theta(n\log n)$. Судячи з оцінки, порозрядне сортування росте лінійним чином по n, оскільки k, m – константи.

Також воно росте лінійним чином по k – при збільшенні довжини типу (кількості розрядів) відповідно зростає кількість проходжень. Однак, в останній пункт реальні умови вносять свої корективи.

Справа в тому, що основний цикл сортування по i-му байту відбувається по вказівнику uchar* bp з кроком sizeof(T) у масиві для одержання чергового розряду. Коли T=char, крок дорівнює 1, T=short – крок дорівнює 2, T=long – крок дорівнює 4... Чим більший розмір типу, тим менш послідовний доступ до пам'яті здійснюється, а це дуже істотно для швидкості роботи. Тому реальний час зростає набагато швидше, ніж теоретичний.

Крім того, порозрядне сортування вже за своєю суттю нецікаве для малих масивів. Кожне проходження містить мінімум 256 ітерацій, тому при невеликих n загальний час виконання $(n+m)$ визначається константою m=256.

9.2. МЕТОДИ ЗОВНІШНЬОГО СОРТУВАННЯ

Прийнято називати «зовнішнім» сортування послідовних файлів, розташованих у зовнішній пам'яті і дуже великих, щоб можна було повністю перемістити їх в основну пам'ять і застосувати один з розглянутих у попередньому розділі методів внутрішнього сортування. Найчастіше зовнішнє сортування застосовується в системах керування базами даних при виконанні запитів, і від ефективності вживаних методів істотно залежить продуктивність СУБД.

Мусимо пояснити, чому йдеться саме про послідовні файли, тобто про файли, які можна читати запис за записом в послідовному режимі, а писати можна тільки після останнього запису. Методи зовнішнього сортування з'явилися, коли найбільш поширеними пристроями зовнішньої пам'яті були магнітні стрічки. Для стрічок послідовний доступ був абсолютно природним. Коли відбувся перехід до пристроїв, що запам'ятовують, з магнітними дисками, що забезпечують «прямий» доступ до будь-якого блоку інформації, здавалося, що послідовні файли втратили свою актуальність. Проте це припущення було помилковим.

Вся річ у тому, що практично всі використовувані на сьогодні дискові пристрої забезпечені рухомими магнітними головками. При виконанні обміну з дисковим накопичувачем виконується підведення головок до потрібного циліндра, вибір потрібної головки (доріжки), прокручування дискового пакету до початку необхідного блоку і, нарешті, читання або запис блоку. Серед всіх цих дій найбільший час займає підведення головок. Саме цей час визначає загальний час виконання операції. Єдиним доступним прийомом оптимізації доступу до магнітних дисків є якомога «ближче» розташування файла на накопичувачі блоків, що послідовно адресуються. Але і в цьому випадку рух

головок буде мінімізованим тільки у тому випадку, коли файл читається або пишеться в послідовному режимі. Саме з такими файлами при потребі сортування працюють сучасні СУБД.

Зазначимо, що насправді швидкість виконання зовнішнього сортування залежить від розміру буфера (або буферів) основної пам'яті, яка може бути використана для цих цілей.

9.2.1. Пряме злиття

Припустимо, що є послідовний файл A, що складається із записів $a1, a2, \ldots, an$ (знову для простоти припустимо, що n є ступенем числа 2). Вважатимемо, що кожен запис складається лише з одного елементу, що є ключем сортування. Для сортування використовуються два допоміжні файли B і C (розмір кожного з них буде $n/2$).

Сортування складається з послідовності кроків, в кожному з яких виконується розподіл стану файлу A у файли B і C, а потім злиття файлів B і C у файл A. (Помітимо, що процедура злиття для файлів повністю ілюструється рисунком 9.20.) На першому кроці для розподілу послідовно читається файл A, і записи $a1, a3, \ldots, a(n-1)$ пишуться у файл B, а записи $a2, a4, \ldots, an$ – у файл C (початковий розподіл). Початкове злиття здійснюється над парами $(a1, a2), (a3, a4), \ldots, (a(n-1), an)$, і результат записується у файл A. На другому кроці знову послідовно читається файл A, і у файл B записуються послідовні пари з непарними номерами, а у файл C – з парними. При злитті утворюються і пишуться у файл A впорядковані четвірки записів. І так далі. Перед виконанням останнього кроку файл A міститиме дві впорядковані підпослідовності розміром $n/2$ кожна. При розподілі перша з них потрапить у файл B, а друга – у файл C. Після злиття файл A міститиме повністю впорядковану послідовність записів. У таблиці 9.10 показаний приклад зовнішнього сортування простим злиттям. Зазначимо, що для виконання зовнішнього сортування методом прямого злиття в основній пам'яті вимагається розташувати всього дві змінні – для розміщення чергових записів з файлів B і C.

Таблиця 9.10. Приклад зовнішнього сортування прямим злиттям.

Початковий стан файла A	8 23 5 65 44 33 1 6
Перший крок Розподіл	
Файл B	8 5 44 1
Файл C	23 65 33 6
Злиття: файл A	8 23 5 65 33 44 1 6
Другий крок Розподіл	
Файл B	8 23 33 44
Файл C	5 65 1 6
Злиття: файл A	5 8 23 65 1 6 33 44
Третій крок Розподіл	
Файл B	5 8 23 65
Файл C	1 6 33 44
Злиття: файл A	1 5 6 8 23 33 44 65

9.2.2. Природне злиття

При використанні методу прямого злиття не береться до уваги те, що початковий файл може бути частково відсортованим, тобто містити впорядковані підпослідовності записів. *Серією* називається підпослідовність записів *ai*, *a*(*i*+1), …, *aj* така, що *ak* <= *a*(*k*+1) для всіх *i* <= *k* < *j*, *ai* < *a*(*i*-1) і *aj*>*a*(*j*+1). Метод природного злиття ґрунтується на розпізнаванні серій при розподілі і їх використанні при подальшому злитті.

Як і у разі прямого злиття, сортування виконується за декілька кроків, в кожному з яких спочатку виконується розподіл файла *A* по файлам *B* і *C*, а потім злиття *B* і *C* у файл *A*. При розподілі розпізнається перша серія записів і переписується у файл *B*, друга – у файл *C* і т.ін. При злитті перша серія записів файлу *B* зливається з першою серією файлу *C*, друга серія *B* з другою серією *C* і т.ін. Якщо перегляд одного файлу закінчується раніше, ніж перегляд іншого (внаслідок різної кількості серій), то залишок не до кінця переглянутого файла повністю копіюється в кінець файла *A*. Процес завершується, коли у файлі *A* залишається тільки одна серія. Приклад сортування файла показаний на рис. 9.22 і рис. 9.23.

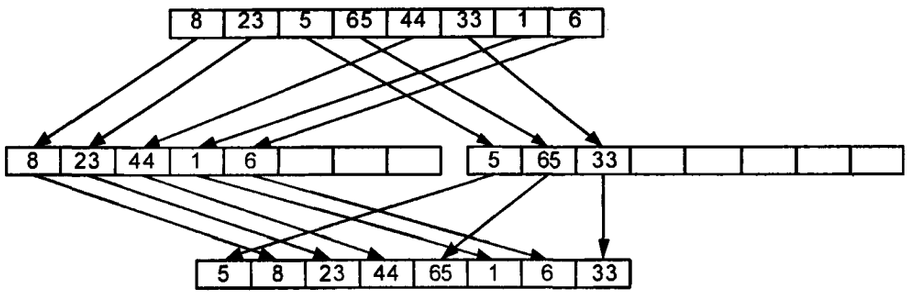


Рис. 9.22. Зовнішнє пряме сортування злиттям. Перший крок.

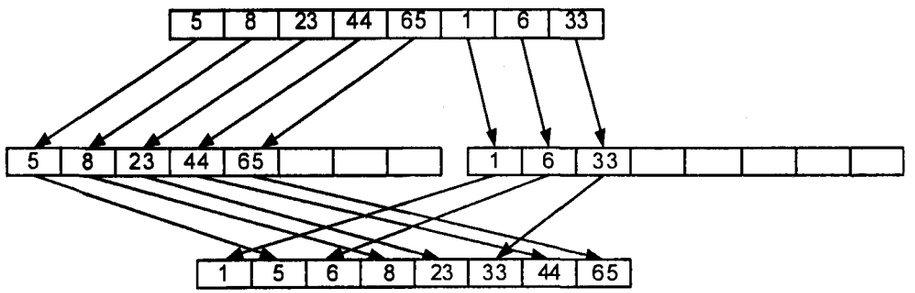


Рис. 9.23. Зовнішнє пряме сортування злиттям. Другий крок.

Очевидно, що кількість зчитувань/перезаписів файлів при використанні цього методу буде не більша, ніж при застосуванні методу прямого злиття, а в середньому – менша. З другого боку, збільшується кількість порівнянь за рахунок тих, які потрібні для розпізнавання кінців серій. Крім того, оскільки довжина серій може бути довільною, то максимальний розмір файлів *B* і *C* може бути близький до розміру файла *A*.

9.2.3. Збалансоване багатошляхове злиття

В основі методу зовнішнього сортування збалансованим багатошляховим злиттям

є розподіл серій початкового файлу по *m* допоміжних файлів *B*1, *B*2, …, *Bm* і їх злиття в *m* допоміжних файлів *C*1, *C*2, …, *Cm*. На наступному кроці здійснюється злиття файлів *C*1, *C*2, …, *Cm* у файли *B*1, *B*2, …, *Bm* і т.ін., поки в *B*1 або *C*1 не утворюється одна серія.

Багатошляхове злиття є природним розвитком ідеї звичного (двошляхового) злиття, ілюстрованого рис. 9.23. Приклад тришляхового злиття показаний на рис. 9.24.

На рис. 9.25 показаний простий приклад застосування сортування багато шляховим (багатофазним) злиттям.

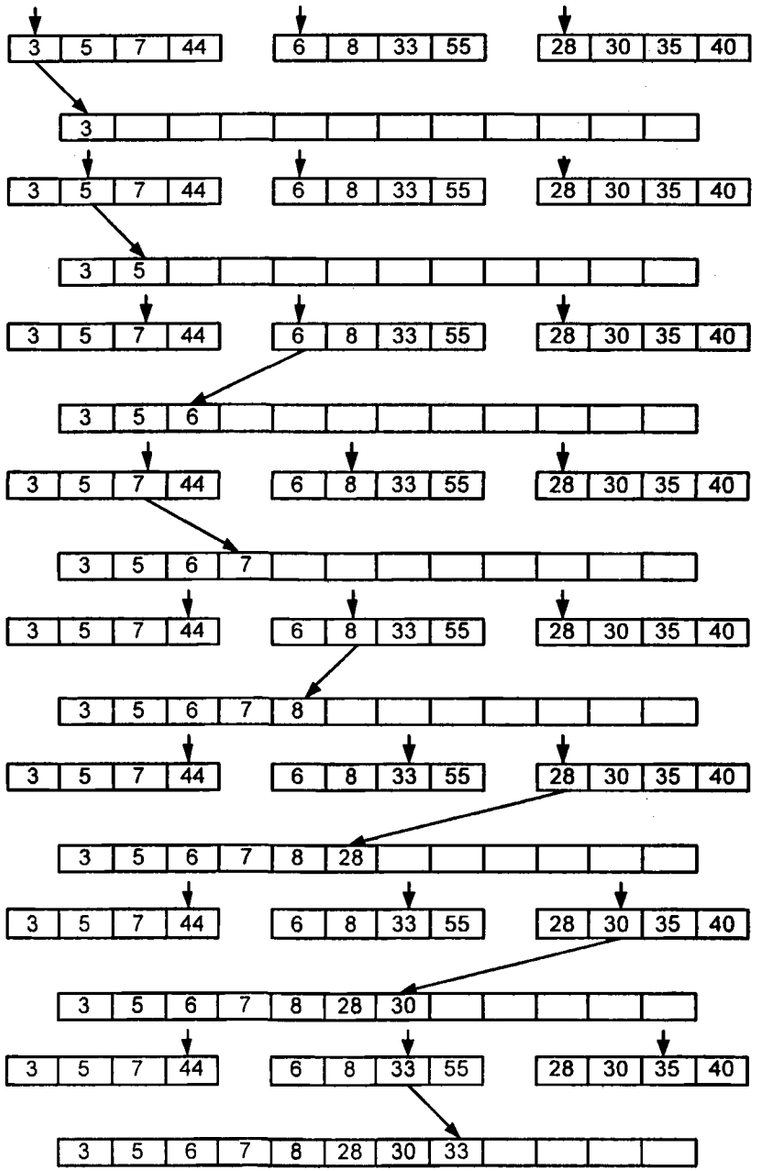


Рис. 9.24. а) Тришляхове злиття. Початок.

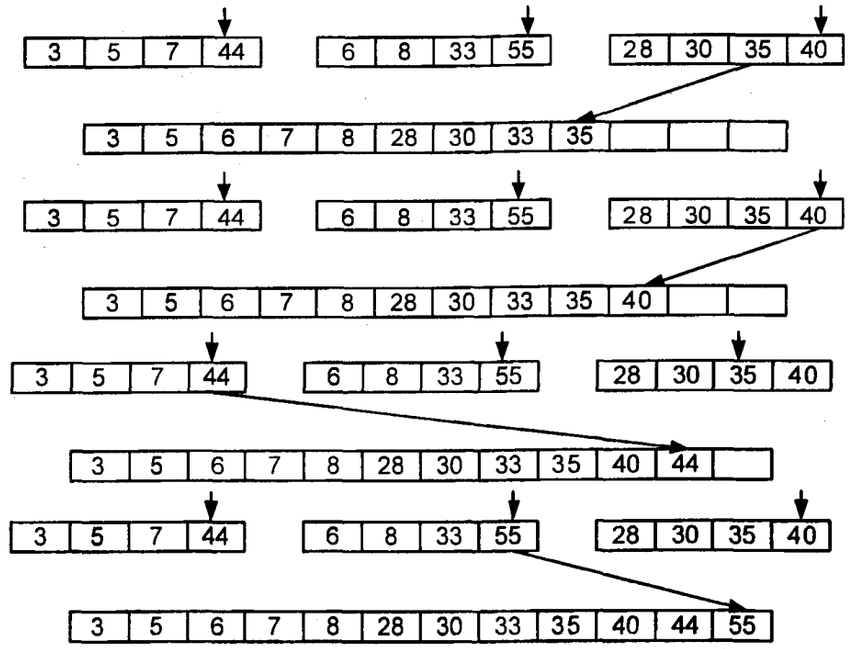


Рис. 9.24. б) Тришляхове злиття. Закінчення.

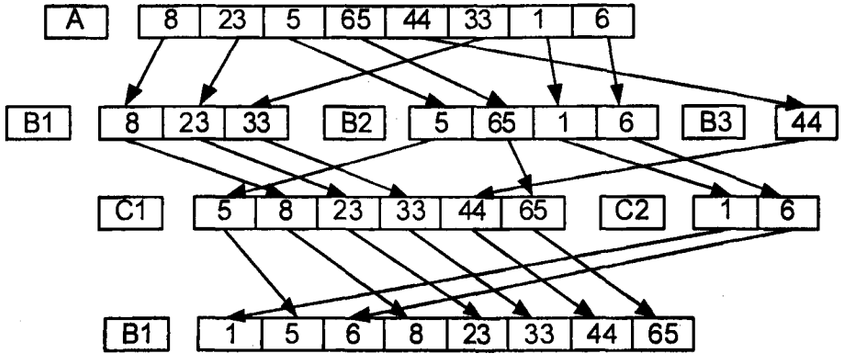


Рис. 9.25. Багатофазне злиття.

Він, зазвичай, дуже тривіальний, щоб продемонструвати декілька кроків виконання алгоритму, проте достатній як ілюстрація загальної ідеї методу. Як показує цей приклад, у міру збільшення довжини серій допоміжні файли з великими номерами (починаючи з номера n) перестають використовуватися, оскільки їм «не дістається» жодної серії. Перевагою сортування збалансованим багатофазним злиттям є те, що кількість проходжень алгоритму оцінюється як $\Theta(\log n)$ (n – кількість записів у початковому файлі), де логарифм береться по n. Порядок кількості копіювань записів – $(\log n)$. Зазвичай, кількість порівнянь не буде меншою, ніж при застосуванні методу простого злиття.

9.2.4. Багатофазне сортування

При використанні розглянутого вище методу збалансованого багатошляхового зовнішнього сортування на кожному кроці приблизно половина допоміжних файлів використовується для введення даних і приблизно стільки ж для виведення злитих серій. Ідея багатофазного сортування полягає у тому, що з наявних m допоміжних файлів (m-1) файл служить для введення злитих послідовностей, а один – для виведення утворюваних серій. Як тільки один з файлів введення стає порожнім, його починають використовувати для виведення серій, одержуваних при злитті серій нового набору (m-1) файлів. Отже, є перший крок, при якому серії початкового файлу розподіляються по m-1 допоміжному файлу, а потім виконується багатошляхове злиття серій з (m-1) файлу, поки в одному з них не утворюється одна серія.

Очевидно, що при довільному початковому розподілі серій по допоміжним файлам алгоритм може не зійтися, оскільки в єдиному непорожньому файлі існуватиме більше, ніж одна серія. Припустимо, наприклад, що використовується три файли $B1$, $B2$ і $B3$, і при початковому розподілі у файл $B1$ вміщені 10 серій, а у файл $B2$ – 6. При злитті $B1$ і $B2$ до моменту, коли ми дійдемо до кінця $B2$, в $B1$ залишаться 4 серії, а у $B3$ потраплять 6 серій. Продовжиться злиття $B1$ і $B3$, і при завершенні перегляду $B1$ у $B2$ міститимуться 4 серії, а у $B3$ залишаться 2 серії. Після злиття $B2$ і $B3$ в кожному із файлів $B1$ і $B2$ міститиметься по 2 серії, яка злитиметься і утворюють 2 серії в $B3$ при тому, що $B1$ і $B2$ – порожні. Отже, алгоритм не зійшовся (таблиця 9.11).

Таблиця 9.11. Приклад початкового розподілу серій, при якому трифазне зовнішнє сортування не приводить до требаго результату.

К-сть серій у файлі B1	К-сть серій у файлі B2	К-сть серій у файлі B3
10	6	0
4	0	6
0	4	2
2	2	0
0	0	2

Оскільки кількість серій у початковому файлі може не забезпечувати можливість такого розподілу серій, застосовується метод додавання порожніх серій, які надалі якомога рівномірніше розподіляються між проміжними файлами і пізнаються при подальшому злитті. Зрозуміло, що чим менше таких порожніх серій, тобто чим ближча кількість початкових серій за вимогами Фібоначчі, тим ефективніше працює алгоритм.

Резюме

1.Розрізняють сортування масивів записів, розташованих в основній пам'яті (внутрішнє сортування), і сортування файлів, що зберігаються в зовнішній пам'яті і не вміщуються повністю в основній пам'яті (зовнішнє сортування). Для внутрішнього і зовнішнього сортування потрібні істотно різні методи.

2.Методами внутрішнього сортування є: метод простого включення, метод

Шелла, сортування вибором, сортування поділом, метод Хоара, сортування деревом, сортування пірамідою, порозрядне сортування, сортування природним злиттям.

3.До зовнішніх методів сортування відносять: пряме злиття, збалансоване багатошляхове злиття, багатофазне злиття.

Контрольні запитання

1. Поясніть особливості внутрішніх та зовнішніх методів сортування.
2. Назвіть методи внутрішнього сортування
3. Назвіть методи зовнішнього сортування.
4. Визначте алгоритмічну складність сортування методом Хоара.
5. Визначте алгоритмічну складність пірамідального сортування.

Тести для закріплення матеріалу

1. Назвати складові частини алгоритмів сортування

а) порівняння, що визначає впорядкованість пари елементів;

б) перестановка, що змінює місцями пари елементів;

в) пошук, який ставить найбільший елемент на початок;

г) власне алгоритм сортування, що здійснює порівняння і перестановку елементів доти, поки всі елементи множини не будуть впорядковані.

2. Перерахуйте методи внутрішнього сортування

а) включенням;
б) Шелла;
в) обмінне;
г) природне злиття;
д) багатофазне злиття;
е) пірамідальне;
є) вибором;
ж) поділом;
з) сортування деревом.

3. Перерахуйте методи зовнішнього сортування

а) включенням;
б) Шелла;
в) обмінне;
г) природне злиття;
д) багатофазне злиття;
е) пірамідальне;
є) вибором;
ж) поділом;
з) сортування деревом.

4. За фраґментом програми визначте метод сортування

```
for (i=1,i<=n-1;i++)
    for(j=i+1;<=n;j++)
        if( a[i]<a[j])
        {
            b=a[i];
            a[i]=a[j];
            a[j]=b;
        }
```

а) бульбашки;
б) включення;
в) Шелла;
г) поділу.

5. За фраґментом програми визначте метод сортування

```
for (i=1,i<=n;i++)
    for(j=i;<=n-1;j++)
        if( a[i]<a[i+1])
        {
            b=a[i];
            a[i]=a[i+1];
            a[i+1]=b;
        }
```

а) бульбашки;
б) включення;
в) Шелла;
г) поділу.

6. За фраґментом програми визначте її призначення

```
Type tree=^tr;
tr=record
k:integer;s:string;  l,r:tree;  end;
function mmm(k:integer;var d,res:tree):boolean;
    var p,q:tree;  b:boolean;
begin b:=false;  p:=d;
    if d<>nil then repeat q:=p; if p^.k=k then b:=true
    else begin  q:=p;  if k<p^.k then p:=p^.l  else p:=p^.r
end

until b or (p=nil); mmm:=b; res:=q;end;
```

а) видалення елемента з списку;
б) видалення елемента в дереві;

в) пошук елемента в списку;
г) пошук елемента в дереві.

7. За фраґментом програми визначте метод сортування:

```
for(i=0;i<n;i++)
    if(i<(int)n/2)
        a1[i]=a[i];
    else
        a2[i-(int)n/2]=a[i];
for(i=0;i<(int)n/2;i++)
    for(j=0;j<(int)n/2-1;j++)
        if(a1[j]<a1[j+1])
        {
            tmp=a1[j]; a1[j]=a1[j+1]; a1[j+1]=tmp;
        }
for(i=0;i<(int)n/2+n%2;i++)
    for(j=0;j<(int)n/2+n%2-1;j++)
        if(a2[j]<a2[j+1])
        {
            tmp=a2[j]; a2[j]=a2[j+1]; a2[j+1]=tmp;
        }
i=0;
while(i<n)
{
    if ((k==(int)n/2)&&(l!=t))
        { a[i]=a2[l]; i++;l++;continue; }
    if ((k!=(int)n/2)&&(l==t))
        { a[i]=a1[k];  i++;k++;continue; }
    if(a2[l]<a1[k])
        { a[i]=a1[k]; i++;k++;continue; }
    if(a1[k]<a2[l])
        { a[i]=a2[l]; i++;l++; continue; }
    if(a2[l]==a1[k])
        { a[i]=a1[k]; i++;k++; a[i]=a2[l]; i++; l++; continue; }
}
```

а) бінарний;
б) включення;
в) Шелла;
г) пірамідальний.

8. За фраґментом програми визначте метод сортування:

```
while (j<znach-1)
{
    for (i=0;i<znach-1;i++)
        if (strcmp(mas[i],mas[i+1])>0)
        {
            pidmin=mas[i];
            mas[i]=mas[i+1];
            mas[i+1]=pidmin;
        }
    j++;
}
```

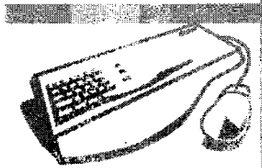
а) бінарний;
б) включення;
в) Шелла;
г) поділу.

9. За фраґментом програми визначте метод сортування:

```
for (i=0; i<rowArr; i++)
    k[i]=0;
for (i=0; i<rowArr-1; i++)
{
    for (j=i+1; j<rowArr; j++)
    {
        if (ar_s[i]<ar_s[j])
        {
            k[i]++;
        }
        else k[j]++;
        }
    }
for (i=0; i<rowArr; i++)
    ar_s1[k[i]]=ar_s[i];
for (i=0; i<rowArr; i++)
    ar_s[i]=ar_s1[i];
```

а) включення;
б) Шелла;
в) поділу;
г) підрахунку.

РОЗДІЛ 10



ЖАДІБНІ АЛГОРИТМИ

- *Поняття жадібного алгоритму.*
- *Принципи жадібного алгоритму.*
- *Відмінність між динамічним програмуванням і жадібним алгоритмом.*

У розділі розглянуто принципово новий вид алгоритмів – жадібний, який застосовується розв'язування для багатьох евристичних задач.

10.1. ПОНЯТТЯ ЖАДІБНОГО АЛГОРИТМУ

✸ ***Жадібний алгоритм*** *– метод оптимізації задач, заснований на тому, що процес ухвалення рішення можна розбити на елементарні кроки, на кожному з яких приймається окреме рішення.*

Рішення, яке приймається на кожному кроці, має бути оптимальним тільки на поточному кроці і прийматися без урахування попередніх або подальших рішень.

Ознаки того, що задачу можна розв'язати за допомогою жадібного алгоритму:

1) задачу можна розбити на підзадачі;

2) величини, що розглядаються в задачі, можна дробити так само, як і задачу, на елементарніші;

3) сума оптимальних рішень для двох підзадач дасть оптимальне рішення для всієї задачі.

Приклад жадібного алгоритму:

```
k:=1;
v[k]:=1;
kol_ver[1]:=kol_ver[1]+1;
s:=0;
while (k<n) do
begin
    min:=maxint;
    for i:=1 to k do
        for j:=1 to n do
            if a[v[i],j]<min then
                begin
                    min:=a[v[i],j];
                    vi:=v[i];
                    vj:=j;
                end;
```



```
    f:=true;
    for i:=1 to k do
        if vj=v[i] then
            f:=false;
        if f then
            begin
                k:=k+1;
                v[k]:=vj;
                kol_ver[vj]:=kol_ver[vj]+1;
                kol_ver[vi]:=kol_ver[vi]+1;
                s:=s+a[vi,vj];
                writeln(v[k],' ',vi,' ',vj);
                readln;
            end;
    a[vi,vj]:=maxint;
    a[vj,vi]:=maxint;
end;
```

Загалом неможливо сказати, чи можна отримати оптимальне рішення за допомогою жадібного алгоритму стосовно конкретного завдання. Але є дві особливості, характерні для завдань, які вирішуються за допомогою жадібних алгоритмів: принцип жадібного вибору і властивість оптимальності для підзадач.

Говорять, що до завдання оптимізації застосуємо принцип **жадібного вибору,** якщо послідовність локально оптимальних виборів дає глобально оптимальне рішення. У цьому полягає головна відмінність жадібних алгоритмів від динамічного програмування: у другому прораховуються відразу наслідки всіх варіантів.

Щоб довести, що жадібний алгоритм дає оптимум, треба спробувати виконати доведення, яке аналогічне до доведення алгоритму задачі про вибір заявок. Спочатку ми показуємо, що жадібний вибір на першому кроці не закриває шлях до оптимального розв'язання: для будь-якого розв'язку є інший варіант, узгоджений з жадібним вибором і не гірший від першого. Потім ми показуємо, що розв'язок, який виник після жадібного вибору на першому кроці, аналогічний до початкового. За індукцією витікатиме, що така послідовність жадібних виборів дає оптимальний розв'язок.

Оптимальність для підзадач

Ця властивість говорить про те, що оптимальний розв'язок всієї задачі містить у собі оптимальні розв`язки підзадач. Наприклад, у задачі про вибір заявок можна помітити, що, якщо A — оптимальний набір заявок, що містить заявку номер 1, то $A1 = A \setminus \{1\}$ — оптимальний набір заявок для меншої множини заявок $S1$, що складається з тих заявок, для яких $si \in fl$.

Оптимальне розв'язування будується покроково. На кожному кроці часткове розв'язування доповнюється новим елементом, який обирається з допустимої підмножини. Для цього необхідні три процедури: процедура вибору кандидата для розширення часткового розв'язку, процедура визначення допустимості цього кандидата, процедура визначення, чи є цей розширений частковий розв'язок повним розв'язком задачі.

10.2. ВІДМІННІСТЬ МІЖ ДИНАМІЧНИМ ПРОГРАМУВАННЯМ І ЖАДІБНИМ АЛГОРИТМОМ

Відмінність між динамічним програмуванням і жадібним алгоритмом можна проілюструвати на прикладі завдання про рюкзак, а точніше, на його дискретному і неперервному формулюванні. Далі ми покажемо, що неперервне завдання вирішується жадібним методом, дискретне ж вимагає тоншого, динамічнішого рішення.

Дискретне завдання про рюкзак. Злодій заліз на склад, на якому зберігається n речей. Кожна річ коштує vi доларів і важить wi кілограмів. Злодій хоче понести товару на максимальну суму, проте він не може підняти більш W кілограмів (всі числа цілі). Що він може покласти в рюкзак?

Неперервне завдання про рюкзак. Тепер злодій уміє дробити товари і укладати в рюкзак тільки їх частки, а не обов'язково ціле. Зазвичай в дискретному завданні йде мова про золоті злитки різної проби, а в неперервному — про золотий пісок.

10.3. ПРИКЛАДИ ЖАДІБНИХ АЛГОРИТМІВ

10.3.1. Алгоритм Краскала

Алгоритм Краскала знаходить контурний ліс мінімальної ваги у заданому графі.

Спочатку опрацьована множина ребер встановлюється порожньою. Потім, доки це можливо, виконується така операція: зі всіх ребер, додавання яких до вже наявної множини не викличе появу в ньому циклу, вибирається ребро мінімальної ваги і додається до множини опрацьованих ребер. Коли таких ребер більше немає, алгоритм завершений. Підграф графу, що містить усі його вершини і знайдену множину ребер, є його контурним лісом мінімальної ваги.

До початку роботи алгоритму необхідно відсортувати ребра за вагою, це вимагає $\Theta(E \log(E))$ часу. Після цього компоненти зв'язності зручно зберігати у вигляді системи непересічної множини. Всі операції у такому разі триватимуть $(E(E, V))$.

10.3.2. Алгоритм Шеннона-Фано

Алгоритм Шеннона-Фано — один з перших алгоритмів стискання, який вперше сформулювали американські вчені Шеннон і Фано. Алгоритм використовує коди змінної довжини: символ, що часто зустрічається, позначається кодом меншої довжини, а символ, що рідко зустрічається — кодом більшої довжини. Коди Шеннона-Фано префіксні, тобто, ніяке кодове слово не є префіксом будь-якого іншого. Ця властивість дозволяє однозначно декодувати будь-яку послідовність кодових слів.

Алгоритм обчислення кодів Шеннона-Фано

Код Шеннона-Фано будується за допомогою дерева. Побудова цього дерева починається з кореня. Вся множина кодованих елементів відповідає кореню дерева (вершині першого рівня). Вона розбивається на дві підмножини з приблизно однаковими сумарними ймовірностями. Ці підмножини відповідають двом вершинам другого рівня, які з'єднуються з коренем. Далі кожна з цих підмножин розбивається на дві підмножини з приблизно однаковою сумарною ймовірністю. Їм відповідають вершини третього рівня. Якщо підмножина містить єдиний елемент, то йому відповідає кінцева вершина кодового

дерева; така підмножина розбиттю не підлягає. Так само чинимо доти, поки не отримаємо всі кінцеві вершини. Гілки кодового дерева позначаємо символами 1 і 0. При побудові коду Шеннона-Фано розбиття множини елементів може бути виконане декількома способами. Вибір розбиття на рівні n може погіршити варіанти розбиття на наступному рівні $(n+1)$ і привести до погіршення коду загалом. Іншими словами, оптимальна поведінка на кожному кроці шляху ще не гарантує оптимальності всієї сукупності дій. Тому код Шеннона-Фано не є оптимальним у загальному сенсі, хоч і дає оптимальні результати при деяких розподілах ймовірності. Для одного і того ж розподілу ймовірності можна побудувати, взагалі кажучи, декілька кодів Шеннона-Фано, і всі вони можуть дати різні результати.

Якщо побудувати всі можливі коди Шеннона-Фано для заданого розподілу ймовірності, то серед них будуть і всі оптимальні коди.

10.3.3. Алгоритм Хафмана

Алгоритм Хафмана (англ. Huffman) — адаптивний жадібний алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю. Був розроблений 1952 року доктором Массачусетського технологічного інституту Девідом Хафманом. На сьогодні використовується в багатьох програмах стискання даних.

На відміну від алгоритму Шеннона-Фано, алгоритм Хафмана залишається завжди оптимальним і для вторинних алфавітів m_2 з більше ніж двома символами.

Цей метод кодування складається з двох основних етапів:

1) побудова оптимального кодового дерева.

2) побудова відображення код->символ на основі побудованого дерева.

Алгоритм

1.Визначається ймовірність появи символів первинного алфавіту в початковому тексті (якщо вони не задані заздалегідь).

2.Символи первинного алфавіту $m1$ виписують у порядку зменшення ймовірності.

3.Останні n_0 символів об'єднують у новий символ, ймовірність якого дорівнює сумарній ймовірності цих символів, видаляють ці символи і вставляють новий символ у список останніх на відповідне місце (за ймовірністю). N_0 визначається із системи:

$$\begin{cases} 2 \le n_0 \le m_2 \\ n_0 = m_1 - a(m_2 - 1) \end{cases},$$

де a — ціле число, m_1 і m_2 — потужність первинного і вторинного алфавіту відповідно.

4. Останні m_2 символів знову об'єднують в один і вставляють його на відповідну позицію, заздалегідь видаливши символи, що увійшли до об'єднання.

5. Попередній крок повторюють доти, доки сума всіх m_2 символів не стане рівною 1.

Цей процес можна подати як побудову дерева, корінь якого — символ із ймовірністю 1, який отримано при об'єднанні символів з останнього кроку, його m_2 нащадків — символи з попереднього кроку і так далі.

Кожні m_2 елементів, що розташовані на одному рівні, нумеруються від 0 до m_2 -1. Коди отримують зі шляхів (від першого нащадка кореня і до листка). При декодуванні можна використовувати те ж саме дерево, прочитується по одній цифрі і робиться крок по дереві, доки не досягається листка — тоді виводиться символ, що стоїть у листку і відбувається повернення в корінь.

Кодування Шеннона-Фано є досить старим методом стискання, і на сьогодні воно не має особливого практичного застосування. У більшості випадків довжина стисненої послідовності за заданим методом дорівнює довжині стисненої послідовності з використанням кодування Хафмана. Але на деяких послідовностях все ж формуються неоптимальні коди Шеннона-Фано, тому стискання методом Хафмана прийнято вважати ефективнішим.

Приклад реалізації

Приклад реалізації алгоритму Хафмана на мові C++ (замість впорядковування піддерев кожного разу шукаємо в масиві дерево з мінімальною вагою) (текст програми взято з Вікіпедії).

```
Class Tree {
    public Tree child0;     // нащадки «0» і «1»
    public Tree child1;
    public boolean leaf;    // ознака листка дерева
    public int character;   // вхідний символ
    public int weight;      // вага цього символу
    public Tree() {}
    public Tree(int character, int weight, boolean leaf)
    {
        this.leaf = leaf;
        this.character = character;
        this.weight = weight;
    }

/*  Обхід дерева з ґенерацією кодів
   1. «Роздрукувати» листове дерево і записати код Хафмана в масив
   2. Рекурсивно обійти ліве піддерево (з ґенеруванням коду).
   3. Рекурсивно обійти праве піддерево.
*/
    public void traverse(String code, Huffman h)
    {
        if (leaf)
        {
            System.out.println((char)character +»  «+ weight +»  «+ code);
            h.code[character] = code;
        }
        if ( child0 != null) child0.traverse(code + «0», h);
        if ( child1 != null) child1.traverse(code + «1», h);
    }
}
class Huffman
{
    public static final int ALPHABETSIZE = 256;
    Tree[] tree   = new Tree[ALPHABETSIZE];      // робочий масив дерев
    int weights[] = new int[ALPHABETSIZE];       // ваги символів
```



```
public String[] code = new String[ALPHABETSIZE]; // коди Хафмана
private int getLowestTree(int used)
{  // шукаємо «найлегше» дерево
    int min=0;
    for (int i=1; i<used; i++)
    if (tree[i].weight < tree[min].weight ) min = i;
    return min;
}
public void growTree( int[] data )
{  // нарощуємо дерево
    for (int i=0; i<data.length; i++)   // шукаємо ваги символів
        weights[data[i]]++;
    //  заповнюємо масив з «листкових» дерев
    int used = 0;                                  // з використаними символами
    for (int c=0; з < ALPHABETSIZE; c++)
    {
        int w = weights[c];
        if (w != 0) tree[used++]= new Tree(з, w, true);
    }
    while (used > 1)
    {                                              // парами зливаємо легкі гілки
        int min = getLowestTree( used );           // шукаємо першу гілку
        int weight0 = tree[min].weight;
        Tree temp = new Tree();                    // створюємо нове дерево
        temp.child0 = tree[min];                   // і прищеплюємо першу гілку
        tree[min]= tree[—used];                    // на місце першої гілки поміщаємо
        // останнє дерево в списку
        min = getLowestTree( used );               // шукаємо 2 гілку і
        temp.child1 = tree[min];                   // прищеплюємо її до нового дерева
        temp.weight = weight0 + tree[min].weight;  // рахуємо вагу нового
                                                   //дерева
        tree[min]= temp;                           // нове дерево поміщаємо на місце 2 гілки
    }                                              // отримали 1 дерево Хафмана
}
public void makeCode()
{  // запускаємо обчислення кодів Хафмана
    tree[0].traverse( «», this);
}
public String coder( int[] data )
{ // кодує дані рядка з 1 і 0
    String str = «»;
    for (int i=0; i<data.length; i++) str += code[data[i]];
    return str;
}
```

```
    public String decoder(String data)
    {
        String str=»»;          // перевіряємо в циклі дані на входження
        int l = 0;              // коду, якщо так, то відкидаємо його ...
        while(data.length() > 0)
        {
            for (int c=0; c < ALPHABETSIZE; c++)
            {
                if (weights[c]>0 && data.startsWith(code[c]))
                {
                    data = data.substring(code[c].length(), data.length());
                    str += (char)c;
                }
            }
        }
        return str;
    }
}
public class HuffmanTest
{ // тест і демонстрація
    public static void main(String[] args)
    {
        Huffman h = new Huffman();
        String str = «to be or not to be?»;   // вхідні дані
        int data[] = new int[str.length()];   // перетворений в масив
        for (int i=0; i<str.length(); i++)
        data[i]= str.charAt(i);
        h.growTree( data );                   // нарощуємо дерево
        h.makeCode();                         // знаходимо коди
        str = h.coder(data);
        System.out.println(str);
        System.out.println(h.decoder(str));
    }
}
```

Результат роботи для рядка «to be or not to be?». Виводяться: символ, його вага і двійковий код. Далі закодований рядок і результат декодування.

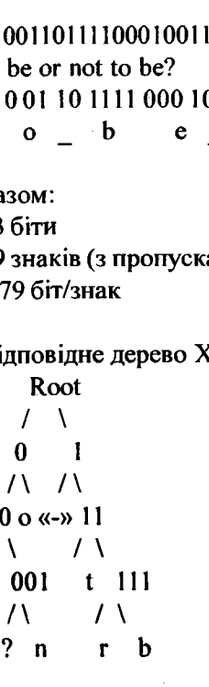
```
E 2 000
? 1 0010
n 1 0011
про 4 01
  5 10
t 3 110
r 1 1110
b 2 1111
```



```
11001101111000100111101000110111010110011011110000010
to be or not to be?
110 01 10 1111 000 10 01 1110 10 0011 01 110 10 110 01 10 1111 000 0010
t  o  _  b    e  _  o  r  _  n  o  t  _  t  o  _  b  e  ?
```

Разом:
53 біти
19 знаків (з пропусками)
2,79 біт/знак

Відповідне дерево Хаффмана.



10.3.4. Алгоритм Пріма

Алгоритм Пріма знаходить контурне дерево мінімальної ваги у зв'язному графі.

Алгоритм Пріма дуже схожий на алгоритм Дейкстри для пошуку найкоротшого шляху в графі (див. розділ 8). Алгоритм Пріма має таку властивість, що ребра завжди утворюють єдине дерево. Дерево починається з довільної вершини *r* і зростає доти, доки не охопить всі вершини *V*. На кожному кроці до дерева *A* додається легке дерево, що сполучає дерево і окрему вершину з частини графа, що залишилася. Ця стратегія є жадібною, оскільки на кожному кроці до дерева додається ребро, котре вносить мінімально можливий вклад до спільної ваги.

Спочатку поточна множина ребер встановлюється порожньою. Потім, доки це можливо, виконується наступна операція: зі всіх ребер, додавання яких до вже наявної множини не викличе появу в утвореному підграфі циклу і розбивання ребер на дві незв'язних групи, обирається ребро мінімальної ваги і додається до вже наявної множини. Коли таких ребер більше немає, алгоритм завершений. Підграф заданого графа, що містить всі його вершини і знайдену множину ребер, є його контурним лісом мінімальної ваги.

Резюме

1. Жадібний алгоритм – метод оптимізації задач, заснований на тому, що процес ухвалення рішення можна розбити на елементарні кроки, на кожному з яких ухвалюється окреме рішення.

2. Ознаки того, що задачу можливо розв'язати за допомогою жадібного алгоритму: задачу можна розбити на підзадачі; величини, що розглядаються в задачі, можна дробити так само як і задачу на піделементи; сума оптимальних розв'язків для двох підзадач дасть оптимальний розв'язок для всієї задачі.

3. Є дві особливості, характерні для завдань, які вирішуються за допомогою жадібних алгоритмів: принцип жадібного вибору і властивість оптимальності для підзадач.

4. Оптимальне рішення будується покроково. На кожному кроці часткове рішення доповнюється новим елементом, який обирається з допустимої підмножини. Для цього необхідні три процедури: процедура вибору кандидата для розширення часткового рішення, процедура визначення допустимості цього кандидата, процедура визначення, чи є це розширене часткове рішення повним рішенням завдання.

5. Приклади жадібних алгоритмів: алгоритм Краскала, алгоритм Шеннона-Фано, алгоритм Хаффмана, алгоритм Пріма.

Контрольні запитання

1. Дайте визначення жадібного алгоритму.

2. Назвіть ознаки задачі, які вказують на те, що її можна розв'язати за допомогою жадібного алгоритму.

3. Поясніть, чому алгоритм Хафмана ефективніший за алгоритм Шеннона-Фано.

4. Поясніть, чому стратегія алгоритму Пріма є жадібною.

5. Вкажіть відмінність між жадібною стратегією та динамічним програмуванням.

Тести для закріплення матеріалу

1. За алгоритмом Хафмана знайти двійкові коди букв за таблицею частот:

а	б	л	т	у	ф	ц	я
0,1	0,02	0,03	0,04	0,008	0,005	0,003	0,01

2. Перерахуйте ознаки жадібного алгоритму:

а) задачу можна розбити на підзадачі;

б) величини, що розглядаються в задачі, можна дробити так само, як і задачу, на елементарніші;

в) кожну підзадачу можна розв'язувати окремим методом;

в) сума оптимальних рішень для двох підзадач дасть оптимальне рішення для всієї задачі.

СПИСОК ТЕРМІНІВ

ЛІТЕРАТУРА ДО ТЕОРЕТИЧНОГО КУРСУ

1. *Анисимов А. В.* Информатика. Творчество. Рекурсия. - Киев: Наук. думка, 1988. – 234 с.

2. *Ахо Альфред, Хопкрофт Джон, Ульман Джеффри.* Структуры данных и алгоритмы. : Пер. с англ. : Уч. пос. – М. : Издательский дом "Вильяме", 2000. – 384 с.: ил.

3. *Боглаев Ю.П.* Вычислительная математика и программирование. – М.: Высшая школа, 1990. – 245 с.

4. *Вирт Н.* Алгоритмы и структуры данных. – М: Мир, 1989. -360с.

5. *Вирт Н.* Алгоритмы + структуры данных = программы. – М., Мир, 1985. – 308 с.

6. *Вирт Н.* Системное программирование: Введение. – М., Мир, 1977. – 403 с.

7. *Д. Кнут.* Искусство программирования, т. I. Основные алгоритмы, 3-е изд. – М.: "Вильяме", 2000. – 328 с.

8. *Д. Кнут.* Искусство программирования, т.2. Получисленные алгоритмы, 3-е изд. - М.: "Вильяме", 2000. – 390 с.

9. *Д. Кнут.* Искусство программирования, т.3. Сортировка и поиск, 2-е изд. – М.: "Вильяме", 2000. – 367 с.

10. Информатика: Энциклопедический словарь для начинающих / Сост. Д. А. Поспелов. М.: Педагогика-Пресс, 1994. – 289 с.

11. *Кормен Томас Х., Лейзерсон Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд.* Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. – М. : Издательский дом "Вильямс", 2005. – 1296 с.

12. *Кузьмичев Д. А.* и др. Автоматизация экспериментальных исследований. – М.: Наука, 1983. – 506 с.

13. *Марков А. А., Нагорный Н. М.* Теория алгорифмов. - М.: Наука, 1984. – 302 с.

14. *Миков А. И.* Информатика. Введение в компьютерные науки: Учебное пособие для университетов по направлению «Прикладная математика и информатика». - Пермь: Перм. ун-т, 1998. – 345 с.

15. *Сибуя М., Ямамото Т.* Алгоритмы обработки данных. – М: Мир, 1986 - 218с.

16. *Абрамов С.А.* и др. Задачи по программированию. – М.: Наука, 1988. – 321 с.

17. *Ван Тассел Д.* Стиль, разработка, эффективность, отладка и испытание программ. – М.: Мир, 1981. – 345 с.

18. *Дейкстра Э.* Дисциплина программирования. - М.: Мир, 1978. – 404 с.

19. *Лэгсам Й, Огенстайн М.* Структуры данных для персональных ЭВМ – М: Мир, 1989 -586с.

20. Турбо Паскаль 7.0 — Киев: BHV, 1998 — 448с.

ДОДАТКИ

ЗАВДАННЯ ДО ЛАБОРАТОРНИХ РОБІТ

Лабораторна робота №1

Тема: моделювання подання в пам'яті векторів і таблиць.

Мета роботи: набуття навичок розміщення в пам'яті векторів і таблиць

Завдання до роботи

Розробити спосіб економного зберігання в пам'яті розріджених матриць (таблиць). Розробити процедури і функції для забезпечення доступу (читання-запис) до елементів матриці.

У контрольному прикладі забезпечити читання і запис всіх елементів матриці. Оцінити час виконання операцій.

Варіанти індивідуальних завдань

№	Завдання
1	Зберегти всі нульові елементи, розміщені в лівій частині матриці.
2	Зберегти всі нульові елементи, розміщені в правій частині матриці.
3	Зберегти всі нульові елементи, розміщені вище головної діагоналі.
4	Зберегти всі нульові елементи, розміщені у верхній частині матриці.
5	Зберегти всі нульові елементи, розміщені в нижній частині матриці.
6	Зберегти всі нульові елементи непарних рядків.
7	Зберегти всі нульові елементи парних рядків.
8	Зберегти всі нульові елементи непарних стовпців.
9	Зберегти всі нульові елементи парних стовпців.
10	Зберегти всі нульові елементи, розміщені у шаховому порядку, починаючи з 1-го елементу 1-ого рядка.
11	Зберегти всі нульові елементи, розміщені у шаховому порядку, починаючи з 2-го елементу 1-ого рядка.
12	Зберегти всі нульові елементи, розміщені на місцях з парними індексами рядків і стовпців.
13	Зберегти всі нульові елементи, розміщені на місцях з непарними індексами рядків і стовпців.
14	Зберегти всі нульові елементи, розміщені вище від головної діагоналі у непарних рядках і нижче від головної діагоналі – у парних.
15	Зберегти всі нульові елементи, розміщені нижче від головної діагоналі на непарних рядках і вище від головної діагоналі – у парних.
16	Зберегти всі нульові елементи, розміщені на головній діагоналі, в перших 3 рядках вище від діагоналі і в останніх 2 рядках нижче від діагоналі.
17	Зберегти всі нульові елементи, розміщені на головній діагоналі і у верхній половині області вище від діагоналі.
18	Зберегти всі нульові елементи, розміщені на головній діагоналі і в нижній половині області нижче від діагоналі.
19	Зберегти всі нульові елементи, розміщені у рядках, індекси яких кратні 3.
20	Матриця розділена діагоналями на 4 трикутники. Зберегти нульові елементи верхнього і нижнього трикутників.
21	Зберегти нульові елементи, розміщені у верхній і нижній четвертинах матриці.
22	Зберегти нульові елементи, розміщені у лівій і правій четвертинах матриці.

№	Завдання
23	Зберегти нульові елементи, розміщені у лівій і верхній четвертинах матриці.
24	Зберегти нульові елементи, розміщені у рядках, індекси яких кратні 3.
25	Зберегти нульові елементи, розміщені нуа стовпцях, індекси яких кратні 3.
26	Зберегти нульові елементи, розміщені у верхній третині рядків і середній третині стовпців.
27	Зберегти нульові елементи, розміщені у верхній третині рядків, першій і третій третині стовпціів.
28	Зберегти нульові елементи, розміщені у верхньому і нижньому трикутнику, за умови розділення матриці діагоналями на 4 трикутники.
29	Зберегти нульові елементи, розміщені у лівому і правому трикутнику, за умови розділення матриці діагоналями на 4 трикутники.
30	Зберегти нульові елементи, розміщені на головній діагоналі і в нижній половині матриці нижче від діагоналі, індекси яких кратні 3.

Лабораторна робота №2

Тема: операції над рядками.

Мета роботи: набуття практичних навичок застосування операцій над рядками.

Завдання до роботи

Розробити процедури та функції, які забезпечують виконання операцій, вказаних у завданні.

У контрольному прикладі передбачити всі можливі комбінації вхідних параметрів (нульова довжина рядка, вихід за межі рядка і т.ін.), у тому числі і неправильні.

Варіанти індивідуальних завдань.

№		Завдання
1	Copies(s,s1,n)	Копіювання рядка **s** у рядок **s1 n** разів.
2	Words(s)	Підрахунок кількості слів у рядку **s.**
3	Parse(s,c)	Розбиття рядка **s** на дві частини: до першого входження символу **c** і після нього.
4	Center(s1,s2)	Центрування – розміщення рядка **s1** в середині рядка **s2.**
5	Left(s,m)	Вирівнювання рядка **s** зліва до довжини **m.**
6	Right(s,m)	Вирівнювання рядка **s** справа до довжини **m.**
7	Reverse(s)	Реверсування рядка **s.**
8	LastPos(s,s1)	Пошук останнього входження підрядка **s1** у рядка **s.**
9	WordIndex(s,n)	Визначення позиції початку в рядку **s** слова з номером **n.**
10	WordLength(s,n)	Визначення довжини слова з номером **n.**
11	WordCmp(s1,s2)	Порівняння рядків (з ігноруванням множинних пробілів).
12	StrSpn(s,s1)	Знаходження довжини тієї частини рядка **s**, яка містить тільки символи з рядка **s1.**
13	Overlay(s,s1,n)	Перекриття частини рядка **s**, починаючи з позиції **n** рядком **s1.**
14	StrLength(s)	Визначити кількість символів у рядку **s**, не враховуючи пробіли.
15	StrCChar(s,c1,s2, n)	Замінити всі символи **c1** у рядку **s**, починаючи з позиції **n**, на рядок **s2.**
16	StrLB(s,n)	Замінити у рядку **s**, починаючи з позиції **n**, всі малі букви на великі.
17	StrDel(s,n,k)	Видалити з рядка **s** підрядок, починаючи з позиції **n** довжиною **к.**

№		Завдання
18	StrAdd(s,s1,n)	Вставити у рядок **s** підрядок **s1**, починаючи з позиції **n.**
19	StrLWord(s,k)	Визначити кількість слів довжиною **к** символів ву рядку **s.**
20	DelBlank(s)	Видалити у рядку **s** головні, хвостові і множинні пробіли.
21	Split(s,s1,s2, c)	Розбити рядок **s** на два рядка **s1** і **s2**, в одній всі символи менші від **c**, в іншій відповідно більші.
22	StrBL(s,n)	Замінити у рядку **s**, починаючи з позиції **n**, всі великі букви на малі.
23	NumCount(s)	Порахувати кількість цифр у рядку **s.**
24	NumCut(s)	Вирізати всі цифри з рядка **s.**

Лабораторна робота №3

Тема: Інтегровані структури даних, запису.

Мета роботи: придбання і закріплення навичок у роботі із записами, в інтеграції даних, в модульному програмуванні.

Завдання до роботи

Для заданої прикладної області розробити опис об'єктів цієї області. Розробити процедури, що реалізують базові операції над цими об'єктами, зокрема:

- ✓ *текстове введення-виведення (консольне і файлове);*
- ✓ *присвоєння;*
- ✓ *задання константних значень;*
- ✓ *порівняння (не менше як 2 типи).*

Підготувати файл початкових даних, що містять не менше як 10 значень конкретних об'єктів.

Використовуючи процедури і описи модуля типу даних, розробити програму, що забезпечує введення початкових даних з першого файла даних в пам'ять і зберігання їх в масиві, сортування масиву за алфавітним і зао числовим параметром.

Варіанти індивідуальних завдань

Для кожної області перераховані параметри об'єкту. Серед параметрів обов'язково є ключове алфавітне поле (наприклад, прізвище), яке ідентифікує об'єкт, у кожного об'єкту є також одне або декілька числових полів, за якими вірогідні звертання до об'єкту.

Набір характеристик може бути розширений і ускладнений на розсуд виконавця.

Nзп	Прикладна область	Атрибути інформації
1	Відділ кадрів	прізвище співробітника, ім'я, по батькові, посада, стаж роботи, оклад
2	Червона книга	вид тварини, рід, сімейство, популяція, проживання, чисельність популяції
3	Виробництво	позначення виробу, група до якої цей виріб відноситься, рік випуску, обсяг випуску, витрату металу

Nзп	Прикладна область	Атрибути інформації
4	Персональні комп'ютери	фірма-виробник, тип процесора, тактова частота, місткість ОЗП, місткість жорсткого диска
5	Бібліотека	автор книги, назва, рік видання, код УДК, ціна, кількість у бібліотеці
6	Супутники планет	назва, назва планети-господаря, рік відкриття, діаметр, період обігання
7	Радіодеталі	позначення, тип, номінал, кількість у схемі, позначення можливого замінника
8	Текстові редактори	найменування, фірма-виробник, кількість вікон, кількість шрифтів, вартість
9	Телефонна станція	номер абонента, прізвище, адреса, наявність блокування, заборгованість
10	Побут студентів	прізвище студента, ім'я, по батькові, факультет, розмір стипендії, кількість членів сім'ї
11	Спортивні змагання	прізвище спортсмена, ім'я, команда, вид спорту, заліковий результат, штрафні очки
12	Змагання факультетів за успішність	факультет, кількість студентів, середній бал по факультету, кількість відмінників, кількість двієчників
13	Роботи, виконані студентами	прізвище студента, ім'я, по батькові, факультет, вид робіт, заробіток
14	Сільське господарство	найменування с/г підприємства, вид власності, кількість працівників, основний вид продукції, прибуток
15	Відомості про сім'ю студента	прізвище студента, ім'я, по батькові, факультет, спеціальність батька, спеціальність матері, кількість братів і сестер
16	Скотарство	вид тварин, кількість особин у стаді у віці до 1 року, кількість особин віком 1 - 3 років, понад 3 роки, смертність у кожній групі, народжуваність
17	Мікросхеми пам'яті	позначення, розрядність, місткість, час доступу, кількість у схемі, вартість
18	Опис зображення	тип фігури (квадрат, коло і т.ін.), координати на площині, числові характеристики, периметр, площа
19	Лісове господарство	найменування зеленого масиву, площа, основна порода, середній вік, густина дерев на один кв.км
20	Міський транспорт	вид транспорту, номер маршруту, початкова зупинка, кінцева зупинка, час у дорозі

Лабораторна робота №4

Тема: стек і черга; хеш-таблиця.

Мета роботи: набуття навичок моделювання зв'язаних динамічних структур даних і роботи з ними

Завдання до роботи

Розробити підпрограми, які забезпечують запити на запис або читання даних із черги, стека або дека. Для організації вказаних структур використовувати масиви або списки. Перевірити працездатність розроблених підпрограм. Послідовність виконання операцій запису або читання вибираються випадково. Порівняти результати роботи, зробити висновки.

Варіанти індивідуальних завдань.

№	Завдання
1	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується за пріоритетом, зняття – з молодших адрес (засади черги). Черга організована у масиві зі зсувом після кожного читання, і на масиві зі зсувом після досягнення межі пам'яті, яка виділена для черги. Пріоритет: мінімальне значення числового параметра, при збігу параметрів – LIFO.
2	Розробити підпрограми роботи з деком. Дек організований у масиві з циклічним заповненням і з використанням двонапрямленого списку. Операції виконуються з обох кінців дека.
3	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується підряд у кінець черги, зняття - по пріоритету. Черга організована на масиві або списку. Пріоритет: мінімальне значення числового параметра, при збігу параметрів – LIFO.
4	Розробити підпрограми роботи зі стеком. Стек організований у масиві з використанням двонапрямленого списку.
5	Розробити підпрограми роботи з деком. Дек організований у масиві з циклічним заповненням і з використанням двонаправленого списку. Операції виконуються з різних кінців дека.
6	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується за пріоритетом, зняття – з молодших адрес (початок черги). Черга організована на масиві з циклічним заповненням і зі зсувом. Пріоритет: *max* значення числового параметра, при збігу параметрів – FIFO.
7	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується за пріоритетом, зняття – зі старших адрес (кінець черги). Черга організована у масиві або у списку. Пріоритет: максимальне значення числового параметра, при збігу параметрів – FIFO.
8	Розробити підпрограми роботи з деком. Дек організований у масиві з циклічним заповненням і зі зсувом. Операції виконуються з обох кінців дека.

№	Завдання
9	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується за пріоритетом, зняття – з молодших адрес (початок черги). Черга організована на масиві з циклічним заповненням і у списку. Пріоритет: максимальне значення числового параметра, при збігу параметрів – FIFO.
10	Розробити підпрограми роботи з деком. Дек організований у масиві з циклічним заповненням і зі зсувом. Операції виконуються з різних кінців дека.
11	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується за пріоритетом, зняття – з молодших адрес (початок черги). Черга організована на масиві зі зсувом після кожного читання і у масиві зі зсувом після досягнення межі пам'яті, яка виділена для черги. Пріоритет: максимальне значення числового параметра, при збігу параметрів – FIFO.
12	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів в чергу виконується по пріоритету, зняття – з молодших адрес (початок черги). Черга організована на масиві з циклічним заповненням і зі зсувом. Пріоритет: мінімальне значення числового параметра, при збігу параметрів – LIFO.
13	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів в чергу виконується за пріоритетом, зняття – зі старших адрес (кінець черги). Черга організована у масиві і у списку. Пріоритет: *min* значення числового параметра, при збігу параметрів – LIFO.
14	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується за пріоритетом, зняття – з молодших адрес (початок черги). Черга організована у масиві з циклічним заповненням і у списку. Пріоритет: мінімальне значення числового параметра, при збігу параметрів – LIFO.
15	Розробити підпрограми роботи з пріоритетною чергою. Встановлення запитів у чергу виконується підряд у кінець черги, зняття – за пріоритетом. Черга організована у масиві і у списку. Пріоритет: максимальне значення числового параметра, при збігу параметрів – FIFO.
16	Розробити процедуру хешування масиву записів, в який передбачається часте додавання даних.

Лабораторна робота №5

Тема: робота з динамічними структурами.

Мета роботи: набуття практичних навичок опрацювання таких динамічних структур, як зв'язні списки і дерева.

Завдання до роботи

Розробити програми які виконують операції, вказані в індивідуальному завданні.

Розробити програму для роботи з двонапрямленими зв'язними списками. Кожен елемент списку містить посилання на наступний і попередній елементи у списку. Програма має забезпечувати введення і побудову списку.

Розробити програму для роботи з деревами. Кожен елемент дерева містить посилання на батьківський елемент і посилання на елементи-нащадки (необмежена кількість). Програма має забезпечувати введення і побудову дерева.

Кожен елемент списку містить інформаційне поле (атрибут) деякого простого типу: символ, стрічка, число.

Всі операції над динамічними структурами мають супроводжуватися відповідним виведенням на екран.

У контрольних прикладах забезпечити опрацювання структур з 10-20 елементами.

Варіанти індивідуальних завдань.

№ зп	Двонапрямлений зв'язний список	Дерево
1	Видалення елемента зі списку за вказаним значенням інформаційного атрибута.	Знайти значення максимуму і мінімуму арифметичних чисел у дереві, кожен елемент якого містить деяке число як значення інформаційного показника.
2	Додавання нового елемента у список після вказаного елемента за значенням інформаційного атрибута.	Визначення кількості нащадків у кожного елемента дерева.
3	Додавання нового елемента у список перед вказаним елементом за значенням інформаційного атрибута.	Визначення елемента дерева, який має найменшу кількість безпосередніх нащадків.
4	Додавання елемента у список у вказану позицію.	Визначення найкоротшої стрічки у дереві, кожен елемент якого містить деяку стрічку як значення інформаційного показника.
5	Видалення зі списку повторень.	Додавання нового елемента в дерево як предка елемента зі вказаним значенням інформаційного атрибута.
6	Видалення другого і передостаннього елемента списку.	Розбиття дерева на два за вказаним значенням інформаційного атрибута елемента для розбиття.
7	Видалення елемента зі списку за вказаним порядковим номером.	Видалення елемента з дерева за вказаним значенням інформаційного атрибуту. (Нащадки видаленого елемента стають нащадками батьківського елемента видаленого)

№ зп	Двонапрямлений зв'язний список	Дерево
8	Доповнення списку з обох кінців.	Видалення елемента з дерева за вказаним значенням інформаційного атрибуту. (Нащадки видаляються)
9	Розбиття списку на два за вказаним порядковим номером елемента для розбиття.	Визначення середнього арифметикного чисел у дереві, кожен елемент якого містить деяке число як значення інформаційного показника.
10	Розбиття списку на два списки за вказаним значенням інформаційного атрибута елемента для розбиття.	Визначення елемента дерева, який розміщений найдалі від кореня дерева.
11	Розбиття списку на два, один з яких містить усі елементи зі значенням інформаційного атрибута меншим від вказаного значення інформаційного атрибут, а інший, відповідно, зі значеннями більшими від вказаного значення.	Визначення елемента дерева, який має найбільшу кількість безпосередніх нащадків.
12	Сортування списку методом бульбашки.	Визначення найдовшої стрічки в дереві, кожен елемент якого містить деяку стрічку як значення інформаційного показника.
13	Видалення всіх елементів із парним порядковим номером.	Додавання нового елемента в дерево як нащадка елемента із вказаним значенням інформаційного атрибута.
14	Видалення всіх елементів з непарним порядковим номером.	Розбиття дерева на два дерева за вказаним значенням інформаційного атрибута елемента для розбиття.
15	Додавання нових елементів у непарні позиції в списку.	Видалення елемента з дерева за вказаним значенням інформаційного атрибута. (Нащадки видаленого елемента стають нащадками батьківського елемента видаленого)
16	Додавання нових елементів у прані позиції в списку.	Видалення елемента з дерева за вказаним значенням інформаційного атрибута. (Нащадки видаляються)
17	Розбиття списку на два за вказаним порядковим номером елемента для розбиття.	Додавання нового елемента в дерево як предка елемента із вказаним значенням інформаційного атрибута.
18	Розбиття списку на два за вказаним значенням інформаційного атрибуту елементу для розбиття.	Розбиття дерева на два за вказаним значенням інформаційного атрибута елемента для розбиття.

№ зп	Двонапрямлений зв'язний список	Дерево
19	Розбиття списку на два, один з яких містить всі елементи зі значенням інформаційного атрибута	Видалення елемента з дерева за вказаним значенням інформаційного атрибуту.
20	Видалення всіх елементів із парним порядковим номером.	Визначення середнього арифметичного чисел в дереві кожен елемент якого містить деяке число як значення інформаційного показника.
21	Видалення всіх елементів із непарним порядковим номером.	Визначення елемента дерева, який розміщений найдалі від кореня дерева.
22	Додавання нових елементів на непарні позиції в списку.	Визначення елемента дерева, який має найбільшу кількість безпосередніх нащадків.
23	Додавання нових елементів на прані позиції в списку.	Визначення всіх листків дерева.
24	Видалення елемента зі списку за вказаним значенням інформаційного атрибуту.	Додавання нового елемента в дерево як нащадка елемента із вказаним значенням інформаційного атрибута.
25	Додавання нового елемента в список після вказаного елемента за значенням інформаційного атрибута.	Розбиття дерева на два дерева за вказаним значенням інформаційного атрибута елементу для розбиття.
26	Додавання нового елемента у список перед вказаним елементом за значенням інформаційного атрибуту.	Видалення елемента з дерева за вказаним значенням інформаційного атрибута.
27	Додавання елемента в список у вказану позицію.	Видалення елемента з дерева за вказаним значенням інформаційного атрибуту. (Нащадки видаляються)
28	Видалення зі списку повторень.	Визначення всіх листків дерева.

Лабораторна робота №6

Тема: рекурсивні алгоритми опрацювання структур даних.

Мета роботи: набуття практичних навичок роботи з рекурсивними функціями.

Завдання до роботи

Розробити програми згідно з алгоритмом з використанням рекурсивної функції та без використання рекурсивної функції. Оцінити час виконання та складність алгоритму.

Варіанти індивідуальних завдань.

№ зп	Завдання	№	Завдання
1	$s = \prod_{x=1}^{n} \frac{x}{e^x - x^2}$	2	$r = \sum_{p=3}^{n} \frac{\cos^2 p}{3p - 3}$

№ зп	Завдання	№	Завдання
3	$p=\prod_{a=1}^{n}\frac{1}{a}$	17	$x=\prod_{j=1}^{n}2j$
4	$q=\sum_{l=1}^{k}(2l-1)$	18	$f=\sum_{t=2}^{n}\frac{\sin^3 t}{t^2-1}$
5	$p=\prod_{b=1}^{n}\frac{\cos b}{2b-1}$	19	$p=\prod_{x=1}^{n}\frac{\sin x}{x^2+1}$
6	$s=\sum_{j=1}^{n}\frac{j^2}{e^j}$	20	$d=\sum_{k=1}^{r}\frac{1}{2k}$
7	$y=\prod_{i=1}^{m}\frac{3i-2}{3i}$	21	$t=\prod_{x=1}^{n}\frac{\cos x}{3x+x^2}$
8	$k=\sum_{a=1}^{n}\frac{a^2}{e^a-e^{-a}}$	22	$y=\sum_{p=1}^{m}\frac{1}{\sin(e^p-1)}$
9	$r=\prod_{l=1}^{n}\frac{\sin l}{(l-1)^2}$	23	$f=\prod_{l=1}^{n}\frac{2i+1}{i^3}$
10	$u=\sum_{i=1}^{m}\frac{2i+1}{i^2}$	24	$q=\sum_{p=2}^{r}\frac{p-1}{p^2}$
11	$p=\prod_{k=1}^{n}\frac{\sin k}{k}$	25	$a=\prod_{k=1}^{p}-(\frac{2k}{k+1}-k^2)$
12	$n=\sum_{l=2}^{p}\frac{l}{l^2+1}$	26	$c=\sum_{j=1}^{p}\frac{\ln j}{j^2}$
13	$q=\prod_{b=1}^{m}\frac{1}{2b-1}$	27	$p=\prod_{t=1}^{n}\frac{\sqrt{t}}{t^2+1}$
14	$a=\sum_{k=1}^{r}\frac{2k-1}{2k+1}$	28	$f=\sum_{x=1}^{t}\frac{2x}{x^3-\sin x}$

Лабораторна робота №7

Тема: дерева, бінарні дерева, пошук.

Мета роботи: набуття навичок програмування дерев.

Завдання до роботи

Розробити засоби динамічного збереження дерев та виконання дій над ними відповідно до варіанту.

Варіанти індивідуальних завдань.

№ зп	Завдання
1	Перевірити, чи двійкове дерево є збалансованим.
2	Знайти вершину в дереві.
3	Розробити програму для роботи з червоно-чорним деревом та процедуру пошуку в ньому.
4	Розробити програму побудови бінарного дерева за арифметичним виразом (наприклад, 2+3-5+7).
5	Знищити заданий елемент у бінарному дереві.
6	Додати вершину у впорядковане бінарне дерево.
7	Додати вершину у невпорядковане дерево.
8	Здійснити заміну значення заданої вершини.
9	Вивести на друк листки дерева.
10	Вивести на друк ліві вершини дерева.
11	Вивести на друк всі вершини, значення яких більше за корінь на одиницю.
12	Вивести на друк праві вершини дерева.
13	Вивести на друк всі вершини, значення яких більше за корінь на задану величину.
14	Перевірити за допомогою дерева, чи стрічка є паліндромом (чинається зліва направо та справа наліво однаково).
15	Побудувати дерево синтаксичного розбору речення.
16	Вивести на друк всі ліві вершини збалансованого дерева.
17	Вивести на друк всі червоні вершини червоно-чорного дерева.
18	Вивести на друк всі чорні вершини червоно-чорного дерева.
19	Знайти в бінарному дереві вершину, сума значень прямих нащадків якої є максимальною.
20	Знайти в бінарному дереві вершину, сума значень прямих нащадків якої є мінімальною

Лабораторна робота №8

Тема: графи, обхід графу, пошук.

Мета роботи: набуття навичок програмування графів.

Завдання до роботи

Забезпечити зберігання графа у вигляді матриці суміжності.

Варіанти індивідуальних завдань.

№ зп	Завдання
1	Розробити алгоритм знаходження зв'язних підграфів заданого графу.
2	Розробити програму обходу графу вшир, поданого матрицею суміжності.
3	Розробити програму обходу графа вглиб, поданого матрицею суміжності.
4	Розробити програму пошуку вершини у графі.
5	Розробити програму пошуку вершини у графі за її значенням.
6	Розробити програму, результатом якої є стек, сформований на основі послідовності вершин, отриманих при обході вглиб.
7	Розробити програму, результатом якої є черга, сформований на основі послідовності вершин, отриманих при обході вшир.
8	Дерево — це зв'язний ациклічний (що не має циклів) граф. Розробити алгоритм, що визначає, чи є граф деревом.
9	Розробити програму, яка за матрицею суміжності формує множину ребер.
10	Знайти у графі двонапрямлені ребра.
11	Сформувати множину вершин, з яких виходять ребра заданої вартості.
12	Сформувати множину ребер, які є ациклічними.
13	Знайти у графі петлі.
14	Знайти вартість шляху між заданими вершинами. Якщо прямого шляху немає, то вивести повідомлення.
15	Знайти суму всіх дуг, що виходять з певної вершини.
16	За матрицею суміжності вивести на друк множину дуг графу.

Лабораторна робота №9

Тема: алгоритми пошуку та сортування для одновимірних масив.

Мета роботи: набуття практичних навичок застосування алгоритмів пошуку та сортування.

Завдання до роботи

Розробити процедури та функції для пошуку в одновимірних масивах та для їх сортування. В контрольному прикладі забезпечити пошук потрібних елементів в непосортованих масивах. Здійснити їх сортування. Здійснити пошук в посортованих масивах. Оцінити час виконання операцій.

Варіанти індивідуальних завдань.

Знайти:

№зп	Завдання
1	Елементи, які наявні в обох масивах **A** і **B**.
2	Елементи, які є тільки в масиві **A** або тільки у масиві **B** по одному разу.
3	Елементи, які наявні в масиві **A**, але відсутні у масиві **B**.
4	Елементи, які наявні в обох масивах **A** і **B** у декількох екземплярах.
5	Елементи, які наявні в декількох екземплярах у масиві **A**, але відсутні у масиві **B**.

№зп	Завдання
6	Елементи, які наявні в декількох екземплярах або тільки у масиві **A**, або тільки у масиві **B**.
7	Елементи, які наявні в декількох екземплярах або в масиві **A**, або в масиві **B** (або в обох масивах).
8	Елементи масиву **A**, які повторюються в масиві **B** декілька раз.
9	Елементи, наявні в обох масивах **A** і **B** в одному екземплярі.
10	Елементи, наявні в одному екземплярі або тільки в масиві **A**, або тільки в масиві **B**.
11	Елементи масиву **A**, які повторюються і одночасно є в масиві **B**.
12	Елементи масиву **A**, які повторюються і одночасно є в масиві **B** в одному екземплярі.
13	Елементи масиву **A**, які не повторюються і одночасно є в масиві **B** у декількох екземплярах.
14	Елементи масиву **A**, які повторюються і одночасно відсутні у масиві **B**.
15	Елементи масиву **A** в одному екземплярі, які є в масиві **B** тільки в одному екземплярі.
16	Елементи масиву **A** в одному екземплярі, які є в масиві **B** у декількох екземплярах.
17	Елементи, які наявні в декількох екземплярах або тільки в масиві **A**, або тільки в масиві **B**.
18	Непарні елементи масиву **A**, які парні в масиві **B**.
19	Елементи, наявні в обох масивах **A** і **B** і більші від числа **K**.
20	Елементи, які є тільки в масиві **A** або в масиві **B** декілька разів.
21	Парні елементи масиву **A**, наявні в масиві **B**.
22	Неповторювані елементи масиву **A**, котрих немає у масиві **B**.
23	Елементи масиву **A** в одному екземплярі, які наявні в масиві **B**.
24	Елементи масиву **A**, наявні в одному екземплярі в масиві **B**.
25	Елементи масиву **B**, які повторюються в масиві **A** декілька разів.
26	Елементи масивів, які наявні в масиві **B**, але відсутні в масиві **A**.
27	Елементи масивів, які наявні непарну кількість разів в обох масивах **A** і **B**.
28	Елементи масивів, які наявні в декількох екземплярах у масиві **B**, але відсутні в масиві **A**.
29	Елементи масивів, які наявні в декількох екземплярах або тільки в масиві **A**, або тільки в масиві **B**.
30	Медіани масивів **A** і **B**.

1. Алгоритми пошуку:

а) лінійний пошук;

б) лінійний пошук з бар'єром;

в) бінарний пошук;

г) пошук Фібоначчі;

д) пошук з переставленням у початок;

е) пошук с транспозицією;

2. Алгоритми сортування:

а) сортування обміном;
б) сортування вибором;
в) бульбашкове сортування;
г) ортування включеням;
д) бульбашкове сортування;
е) швидке сортування;
є) сортування Шелла;
ж) пірамідальне сортування;
з) сортування Хоара.

Лабораторна робота №10

Тема: робота зі записами і файлами.

Мета роботи: набуття практичних навичок опрацювання структур та роботи з файлами.

Завдання до роботи

Розробити програму, яка забезпечує опрацювання структур даних і їх збереження у файлі.

Опис деякого об'єкту здійснюється за допомогою структури даних типу запис. Необхідно забезпечити опрацювання 3-5 атрибутів об'єкту з використанням різних простих типів даних (стрічки, символи, числа, логічний тип). Забезпечити виконання таких операцій:

- *введення даних;*
- *пошук за значенням атрибут;*
- *послідовний перегляд;*
- *модифікацію значень атрибутів об'єктів (структури, що його описує);*
- *видалення об'єкту (запису, що його описуванняс);*
- *сортування за значеннями атрибутів;*
- *результати всіх операцій мають зберігатися у файлі.*

У контрольному прикладі продемонструвати виконання основних операцій з файлом, який містить 10-20 збережених описів об'єктів.

Варіанти індивідуальних завдань.

Кожен студент обирає довільний об'єкт для описування у вигляді запису.

НАВЧАЛЬНЕ ВИДАННЯ

Шаховська Наталія Богданівна
Голощук Роман Олегович

з курсу

"Алгоритми і структури даних"

НАВЧАЛЬНИЙ ПОСІБНИК

для студентів інституту комп'ютерних наук та технологій стаціонарної та заочної форми навчання

Керівник видавничого проекту *В. М. Піча*

Підписано до друку з оригінал-макета 20.08.2009 р.
Формат 70 × 100/16. Умовн. друк. арк. 38,8. Гарнітура Таймс Нью Роман

ІПП "Магнолія 2006"
а/с 2623, м. Львів-60, 79060, Україна, тел./факс 240-54-84; 245-63-70
e-mail: alexmagnolia@org.lviv.net
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції: серія
ДК № 2534 від 21.06.2006 року,
видане Державним комітетом інформаційної політики,
телебачення та радіомовлення України

Надруковано у друкарні видавництва "Магнолія 2006"
м. Львів, вул. Луганська, 238 Д.