Міністерство освіти і науки України
ДВНЗ «Прикарпатський національний університет імені Василя Стефаника»

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Лабораторний практикум

Івано-Франківськ – 2015

Міністерство освіти і науки України
ДВНЗ «Прикарпатський національний університет імені Василя Стефаника»

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Лабораторний практикум

Івано-Франківськ – 2015

УДК 519.683, 378.147.88

Рекомендовано до друку Вченою радою факультету математики та інформатики, протокол №9 від 2 червня 2015 року

Рецензенти:

Яремій І. П. – д.ф.-м.н., доцент, професор кафедри матеріалознавства і новітніх технологій

Горслов В. О. – к.т.н., доцент, доцент кафедри інформатики.

Власій О.О. Алгоритми та структури даних: Лабораторний практикум / О. О. Власій. – Івано-Франківськ: ДВНЗ «Прикарпатський національний університет імені Василя Стефаника», 2015. – 68 с.

Практикум містить цикл лабораторних робіт з дисципліни «Алгоритми та структури даних» та організаційно-методичні вказівки до проведення лабораторних занять. Кожна лабораторна робота містить коротко теоретичні відомості, приклад розв'язання типового завдання, завдання для самостійного виконання та питання для самоконтролю.

Практикум розрахований на студентів, які вивчають комп'ютерні науки, а також для студентів інших спеціальностей, які хочуть поглибити свої знання в галузі програмування.

Зміст

Вступ

Дисципліна «Алгоритми та структури даних» є базовою дисципліною професійного циклу підготовки спеціалістів з комп'ютерних наук. Ця дисципліна передбачає базові знання з дисциплін «Програмування», «Теорія алгоритмів», «Дискретна математика».

Метою вивчення даної дисципліни є формування у студентів знань, умінь і навичок в області методів представлення даних в пам'яті комп'ютера та основних алгоритмів їх опрацювання.

Поняття алгоритму невід'ємно пов'язане із структурою даних, обраною для практичної реалізації поставленої задачі. Алгоритми опрацювання даних в комп'ютерних науках використовуються для опису методів розв'язання задачі, які можна реалізувати в середовищі програмування. Ретельний підхід до вивчення класичних алгоритмів та пошуки шляхів розробки нових алгоритмів є фундаментом ефективного розв'язання поставленої задачі за допомогою комп'ютера майбутніми фахівцями з програмування.

В даному практикумі представлено 12 лабораторних робіт, кожна з яких містить необхідні короткі теоретичні відомості, приклад розв'язання типового завдання та приклад програмної реалізації в середовищі MS Visual Studio 2010 Express, зразки завдань для самостійного виконання, контрольні питання. В практикумі також наведені рекомендовані теми для самостійного вивчення та рекомендована література. З метою формування у студентів базових вмінь і навичок по обробці структур даних різних типів рекомендовано уникати використання стандартних алгоритмів та контейнерів C++ при програмній реалізації завдань.

Організаційно-методичні вказівки

Метою виконання лабораторних робіт є формування практичних вмінь та навичок створення та опрацювання статичних. напівстатичних та динамічних, лінійних та нелінійних структур даних, а також використання типових алгоритмів на цих структурах.

В результаті виконання лабораторних робіт студенти повинні знати основні статичні, напівстатичні та динамічні структури даних, лінійні та нелінійні структури даних, а саме масиви структур, списки різних видів, стеки, черги, дерева та графи; основні алгоритми опрацювання типових структур данихЖ алгоритми створення/видалення структури, вставки/видалення елемента структури, очищення структури і т.п.; алгоритми впорядкування та пошуку.

Порядок виконання лабораторних робіт

Для виконання лабораторних робіт необхідно:

1. використовуючи літературні джерела, конспект лекцій, методичні розробки з дисципліни, засвоїти теоретичний матеріал, пов'язаний з тематикою лабораторної роботи;
2. відповісти керівнику лабораторних занять на поставлені запитання за темою лабораторної роботи;
3. опрацювати наведений у лабораторній роботі приклад;
4. отримати індивідуальне завдання;
5. розробити схему алгоритму для розв'язування індивідуального завдання;
6. за потреби написати відповідну програмну реалізацію;
7. протестувати програму для тривіальних вхідних даних;
8. виконати програму та зафіксувати отримані результати згідно завдання;
9. перевірити правильність роботи програми; при необхідності внести зміни у програму та зафіксувати остаточні результати;
10. оформити та захистити звіт про виконання лабораторної роботи.

Вимоги до оформлення звіті про виконання лабораторних робіт

Звіти про виконання лабораторних робіт оформляються на аркушах формату А4 та подаються до захисту в електронному вигляді з доданими програмними реалізаціями завдань (вихідний код).

Структура звіту:

- титульний аркуш (зразок титульного аркуша наведено в додатку А);
- номер та тема роботи;
- мета виконання лабораторної роботи;
- індивідуальне завдання з детальним формулюванням задачі;
- графічна схема алгоритму розв'язування задачі з поясненням (за потреби);
- базові елементи тексту програми. Програма повинна контролювати правильність вводу вхідних даних та мати коментарі до її основних структурних конструкцій.
- результати тестування програми на тривіальних вхідних даних (за потреби);
- результати реалізації програми згідно з варіантом. Вказується формат і значення вхідних даних та отриманих для них результатів;
- висновки, в яких вказується призначення програми, обмеження на її застосування, можливі варіанти вдосконалення та які знання отримано в ході виконання роботи.

Звіт повинен бути написаний українською мовою, акуратно та грамотно, з дотриманням правил оформлення ділової документації. Назви розділів звіту повинні бути візуально виділені розміром, жирністю, курсивом шрифта або підкресленням. Лабораторна робота повинна бути здана у визначені викладачем терміни.

Лабораторна робота №1

Квадратичні алгоритми впорядкування масивів

Мета: Формування практичних вмінь та навичок при використанні різних типів квадратичних алгоритмів впорядкування.

1.1 Теоретичні відомості

Метод вибору (select)

Словесний опис

1) Знаходимо мінімальний елемент, переставляємо його (робимо взаємозаміну) з першим.

2) Знаходимо мінімальний елемент в залишеному невпорядкованому масиві a[2] .. a[n], переставляємо його із a[2].

. . .

n-2) Знаходимо мінімальний елемент в масиві (a[n-2], a[n-1], a[n]), переставляємо його із a[n-2].

n-1) Впорядковуємо залишену пару a[n] і a[n-1].

Алгоритм	**Ітерації**
procedure order_select(var a:BigArray);	Вихідний масив: (6,7,2,3,8,5,1,9,3,2)
var i,j,nun:integer;	1-й прохід: (1;6,7,2,3,8,5,9,3,2)
min:elements;	2-й прохід: (1,2;6,7,2,3,8,5,9,3)
t:byte;	3-й прохід: (1,2,2;6,7,3,8,5,9,3)
for i:=1 to n-1 do	4-й прохід: (1,2,2,3;6,7,3,8,5,9)
t:=0; num:=i; min:=a[i];	5-й прохід: (1,2,2,3,3;6,7,8,5,9)
for j:=i+1 to n do	6-й прохід: (1,2,2,3,3,5;6,7,8,9)
if a[j]<min then num:=j;	7-й прохід: (1,2,2,3,3,5,6;7,8,9)
min:=a[j];	8-й прохід: (1,2,2,3,3,5,6,7;8,9)
t:=1;	9-й прохід: (1,2,2,3,3,5,6,7,8;9)
if t=1 then a[num]:=a[i];	
a[i]:=min;	

Критерії

Порівнянь: $\frac{1}{2}n(n-1)$; n-квадратичний алгоритм.

Перестановок (дій): $\min = 3(n-1)$; $\max = \frac{1}{4}n^2 + 3(n-1)$.

Метод вставки (insert)

Словесний опис

1) Впорядковуємо два перші елементи: a[1] та a[2].

2) Вставляємо елемент a[3] "на своє місце" відносно перших двох, щоби перші три елементи масиву стали впорядкованими.

3) Вставляємо елемент a[4] "на своє місце" відносно попередніх чотирьох.

. . .

n-1) Вставляємо останній елемент a[n] "на своє місце" відносно усіх попередніх.

Вставлення елемента "на своє місце" можна здійснювати, зокрема, так: пересувати його наліво до тих пір, поки він не стане більшим за елемент, що стоїть зліва від нього.

Алгоритм	**Ітераці**
procedure order_insert(var a:BigArray);	Вихідний масив: (6,7,2,3,8,5,1,9,3,2)
var i,j:integer;	1-й прохід: (6,7;2,3,8,5,1,9,3,2)
x:elements;	2-й прохід: (2,6,7;3,8,5,1,9,3,2)
for i:=2 to n do	3-й прохід: (2,3,6,7;8,5,1,9,3,2)
x:=a[i];	4-й прохід: (2,3,6,7,8;5,1,9,3,2)
j:=i-1;	5-й прохід: (2,3,5,6,7,8;1,9,3,2)
while (x<a[j]) and (j>0) do	6-й прохід: (1,2,3,5,6,7,8;9,3,2)
a[j+1]:=a[j];	7-й прохід: (1,2,3,5,6,7,8,9;3,2)
j:=j-1;	8-й прохід: (1,2,3,3,5,6,7,8,9;2)
a[j+1]:=x;	9-й прохід: (1,2,2,3,3,5,6,7,8,9)

Критерії

Порівнянь: $\min = n-1$, $\max = \frac{1}{2}(n^2+n)$.

Перестановок (дій): $\min = 2(n-1)$, $\max = \frac{1}{2}(n^2+n)$.

Метод «бульбашки» (booble)

При цьому методі лічильник "біжить" від останнього елемента до першого і знаходячи по дорозі найменший елемент, "пересуває" його до початку масиву. Якщо масив записати зверху вниз, то лічильник "біжить" знизу вверх, підхоплюючи з собою найменший ("найлегший") елемент на верхню позицію, подібно до того, як бульбашка з рідини випливає на її поверхню. Звідси і назва методу.

Словесний опис

1) Порівнюємо a[n] із a[n-1]; якщо a[n]<a[n-1], то міняємо їх місцями;

порівнюємо a[n-1] із a[n-2]; якщо a[n-1]<a[n-2], то міняємо їх місцями;

. . .

порівнюємо a[2] із a[1]; якщо a[2]<a[1], то міняємо їх місцями.

В результаті цього a[1] стає найменшим.

2) Цю ж процедуру проробляємо для масиву a[2] .. a[n] (без елемента a[1]), "виштовхуючи" найменший серед залишених елементів на позицію a[2].

. . .

n-2) Порівнюємо a[n] із a[n-1], при потребі міняючи їх місцями;

порівнюємо a[n-1] із a[n-2], при потребі міняючи їх місцями.

n-1) Впорядковуємо залишену пару a[n] і a[n-1].

Алгоритм	**Ітерації**
procedure order_booble(var a:BigArray);	Вихідний масив: (6,7,2,3,8,5,1,9,3,2)
var i,j:integer;	1-й прохід: (1;6,7,2,3,8,5,2,9,3)
x:elements;	2-й прохід: (1,2;6,7,2,3,8,5,3,9)
for i:=1 to n-1 do	3-й прохід: (1,2,2;6,7,3,3,8,5,9)
for j:=n downto i+1 do	4-й прохід: (1,2,2,3;6,7,3,5,8,9)
if a[j]<a[j-1] then x:=a[j-1];	5-й прохід: (1,2,2,3,3;6,7,5,8,9)
a[j-1]:=a[j];	6-й прохід: (1,2,2,3,3,5;6,7,8,9)
a[j]:=x;	7-й прохід: (1,2,2,3,3,5,6;7,8,9)
	8-й прохід: (1,2,2,3,3,5,6,7;8,9)
	9-й прохід: (1,2,2,3,3,5,6,7,8;9)

Критерії

Порівнянь: $\frac{1}{2}n(n-1)$; n-квадратичний алгоритм.

Перестановок: $\min = 0, \max = \frac{3}{2}(n^2-n)$.

1.2 Приклад

Написати програму, яка сортує масив з 10 випадкових цілих чисел, методом вибору. Процес впорядкування оформити як функцію.

```
#include <iostream>
#include <cstdlib>
#include "windows.h"
#include <time.h>
#define N 100
using namespace std;
void selectSort(int a[], long size) {
  long i, j, k;
   int x;
   for( i=0; i < size; i++) {      // i - номер поточного кроку
    k=i; x=a[i];
    for( j=i+1; j < size; j++)// цикл вибору найменшого елемента
      if (  a[j] < x ) {
        k=j; x=a[j];           // k - індекс найменшого елемента
      }
    a[k] = a[i]; a[i] = x;  // міняємо місцями найменший з a[i]
      }
     }
 SetConsoleCP(1251);    SetConsoleOutputCP(1251);
  int s[N];
  srand(time(NULL));
  for (int i=0;i<N;i++){
       s[i]=rand();
       cout<<" s["<<i<<"]= "<<s[i]<<endl;
     }
  cout<<endl;
  cout<<"Sorted massiv"<<endl;
  for (int i=0;i<N;i++) {
       cout << " s["<<i<<"]= "<<s[i]<<endl;
       }
  return 0;
}
```

У програмі використано такі функції:

SetConsoleCP(), SetConsoleOutputCP() – функції, що встановлюють певну кодову сторінку (CodePage).

Сторінка 1251 дозволяє відображати кирилицю (а за замовчуванням встановлено сторінку 866).

Якщо після цього українські символи не друкуються, то у властивостях консольного вікна виберіть шрифт "**Lucida Console**".

Функція **time()** (описана в бібліотеці <time.h>) з параметром **NULL** повертає поточний час.

Функція **srand()**, для якої потрібно задати аргумент, задає функції **rand()** початковий параметр для генерації.

Тобто функція **srand(time(NULL))** задає можливість при кожному запуску програми генерувати РІЗНІ послідовності випадкових чисел функцією **rand().**

1.3 Завдання для самостійного виконання

1. Написати 3 програми, кожна з яких демонструє окремий метод впорядкування.

Структура кожної програми:

– кількість елементів масиву задати в директиві **#define N 10** ;

– масиви – цілочисельні (довжиною N елементів);

– ініціалізація масивів – не з клавіатури, а за допомогою функції генерування псевдовипадкових цілих чисел **rand()**, причому із заданням початкового параметра.

– програма, окрім впорядкування, повинна виводити на екран проміжні результати, як це зображено на рисунках в правих таблицях.

2. Посортувати рядки двовимірного масиву цілих чисел у порядку спадання.

1.4 Питання для самоконтролю.

1. Як класифікують алгоритми за часом їх виконання?
2. Що означає швидкість виконання алгоритму $O(n^2)$?
3. Які є квадратичні алгоритми впорядкування?
4. Яку дію виконує алгоритм впорядкування вибором при *i*–тій ітерації?
5. Яку дію виконує алгоритм впорядкування бульбашкою при *i*–тій ітерації?
6. Яку дію виконує алгоритм впорядкування вставками при *i*–тій ітерації?
7. Які критерії алгоритму впорядкування вибором?
8. Які критерії алгоритму впорядкування вставками?
9. Які критерії алгоритму впорядкування бульбашкою?
10. Який із вивчених алгоритмів, на вашу думку, найефективніший?

Лабораторна робота №2

Аналіз алгоритмів впорядкування

Мета: навчитися оцінювати алгоритми впорядкування за різними критеріями: часом, стійкістю, природністю, додатковою пам'яттю.

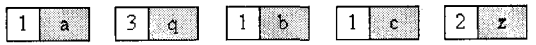
2.1Теоретичні відомості

Параметри, за якими проводиться оцінка алгоритмів:

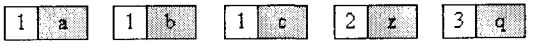
1. *Час впорядкування* – основний параметр, що характеризує швидкодію алгоритму.

2. *Пам'ять* – ряд алгоритмів вимагає виділення додаткової пам'яті під тимчасове зберігання даних. При оцінці використовуваної пам'яті не враховуватиметься місце, яке займає вихідний масив і незалежні від вхідної послідовності витрати, наприклад, на зберігання коду програми.

3. *Стійкість* – стійка впорядкування не змінює взаємного розташування рівних елементів. Така властивість може бути дуже корисним, якщо вони складаються з декількох полів, як на рис. 1, а впорядкування відбувається по одному з них, наприклад, по **x**.

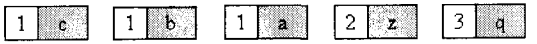


Вихідні дані



Приклад роботи стійкого впорядкування

Взаємне розташування рівних елементів з ключем 1 і додатковими полями " a ", " b ", " c " залишилося колишнім: елемент з полем " a ", потім – з " b ", потім – з " c ".



Приклад роботи нестійкого впорядкування

Взаємне розташування рівних елементів з ключем 1і додатковими полями " a ", " b ", " c " змінилося.

4. *Природність поведінки* – ефективність методу при обробці вже відсортованих, або частково відсортованих даних. Алгоритм поводиться

природно, якщо враховує цю характеристику вхідної послідовності і працює краще.

Обчислення часу виконання програм

Для того, щоб знайти час виконання програми, потрібно використати функцію clock(). Прототип функції clock() знаходиться в заголовковому файлі <ctime>, який потрібно підключити на початку програми. Функція clock() повертає значення часу в мілісекундах (1с = 1000млс). Причому відлік часу починається з моменту запуску програми. Якщо потрібно виміряти час виконання всієї програми, то в кінці програми, перед оператором return 0; потрібно запустити функцію clock(), яка покаже робочий час. Для пошуку часу виконання фрагмента коду потрібно знайти різницю між початковим та кінцевим часом.

Приклад 1

```
// Як знайти час виконання фрагмента програми?
 // заголовковий файл з прототипом потрібної функції clock()
#include <ctime>
//          ...
    unsigned int start_time =  clock(); // початковий час
// тут повинен бути фрагмент програми, час виконання якого потрібно обчислити
    unsigned int end_time = clock(); // кінцевий час
    unsigned int search_time = end_time - start_time; // шуканий час
```

Приклад 2

```
// Як знайти час роботи програми?
  #include <ctime>
 //          ...
 // тут повинен бути код програми, час роботи якого потрібно виміряти
 unsigned int end_time = clock(); // час роботи програми
```

Вдосконалення методу бульбашки

По-перше, розглянемо ситуацію, коли на якому-небудь з проходів не відбулося жодного обміну. Що це означає? Це означає, що всі пари розташовані в правильному порядку, так що масив вже відсортований. І продовжувати процес не має сенсу (особливо, якщо масив був відсортований із самого початку!). Отже,

перше покращення алгоритму полягає в запам'ятовуванні, чи проводився на даному проході який-небудь обмін. Якщо ні - алгоритм закінчує роботу.

Процес покращення можна продовжити, якщо запам'ятовувати не тільки сам факт обміну, а й індекс останнього обміну K. Дійсно: всі пари сусідніх елементів з індексами, *меншими за K, вже розташовані в потрібному порядку*. Подальші проходи можна закінчувати на індексі K, замість того, щоб рухатися до встановленої заздалегідь верхньої межі.

Якісно інше покрщення алгоритму можна отримати з наступного спостереження. Хоча легка бульбашка знизу підніметься наверх за один прохід, важкі бульбашки опускаються зі мінімальною швидкістю: один крок за ітерацію. Так що масив 2 3 4 5 6 січня буде відсортований за 1 прохід, а впорядкування послідовності 6 1 2 3 4 5 зажадає 5 проходів. Щоб уникнути подібного ефекту, можна змінювати напрямок проходів, що слідують один за одним. Одержаний алгоритм іноді називають **"шейкер-впорядкуванням"**.

2.2 Приклад функції, яка реалізує шейкер-алгоритм

```
void shakerSort(int a[], long size) {
  long j, k = size-1;
  long lb=1, ub = size-1; // межі невідсортованої частини масиву
  int x;
  do
   {
         for( j=ub; j>0; j-- ) {  // прохід знизу вгору
         {
           if ( a[j-1] > a[j] ) {
              x=a[j-1]; a[j-1]=a[j]; a[j]=x;
              k=j;
             }
         }
       lb = k+1;
        for (j=1; j<=ub; j++){ // прохід згори вниз
           if ( a[j-1] > a[j] )              {
              x=a[j-1]; a[j-1]=a[j]; a[j]=x;
              k=j;
             }
         }
     ub = k-1;
  } while ( lb < ub );
}
```

Які властивості змінилися для даного алгоритму в порівнянні з бульбашкою – дослідити самостійно.

2.3Завдання для самостійного виконання

1. На прикладі масиву з 10 елементів показати покрокове виконання шейкер-впорядкування. Масив заповнити випадковими числами від 1 до 20*N (N-номер по журналу).

2. Провести порівняльний аналіз квадратичних алгоритмів. Час виконання виміряти для різної кількості елементів масиву – 10*N, 100*N, 10000*N. Дані записати в таблицю

Алгоритм	Час (в мілісекундах)			Додаткова пам'ять	Стійкість	Природність
	110N	1100N	10000N			
Впорядкування вибором						
Впорядкування бульбашкою						
Шейкер-впорядкування (модифікація бульбашки)						
Впорядкування вставками						

2.4 Питання для самоконтролю

1. За якими параметрами проводиться оцінювання алгоритмів впорядкування?
2. Які алгоритми називають: лінійними, квадратичними, логарифмічними?
3. Які алгоритми називають стійкими?
4. Які алгоритми називають природними?
5. Як класифікують алгоритми за потребою додатковою пам'яті?
6. Яку дію виконує алгоритм шейкер-впорядкування при *i*–тій ітерації?
7. Наведіть порівняльний аналіз методів впорядкування бульбашкою та шейкер-впорядкування.
8. Наведіть порівняльний аналіз методів впорядкування вибором та вставками.
9. Наведіть порівняльний аналіз методів впорядкування бульбашкою та вибором.
10. Які особливості методу впорядкування вставками зі сторожовим елементом?

Лабораторна робота №3

Впорядкування структурованих даних

Мета: Закріпити навички роботи зі структурованими даними типу «структура» та навички використання алгоритмів сортування.

3.1 Теоретичні відомості

Нехай є послідовність a0, a1... an і функція порівняння, яка на будь-яких двох елементах послідовності приймає одне з трьох значень : менше, більше або дорівнює. Завдання впорядкування полягає в перестановці членів послідовності таким чином, щоб виконувалася умова: ai <= ai +1, для всіх i від 0 до n.

Можлива ситуація, коли елементи складаються з декількох полів :

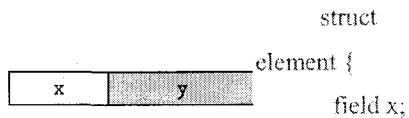


Рис. 1

```
struct
element {
    field x;
    field y;
}
```

Якщо значення функції порівняння залежить тільки від поля **x**, то **x** називають ***ключем***, за яким відбувається впорядкування (або ще кажуть *впорядкування*). На практиці, як **x** часто виступає число, а поле **y** зберігає будь-які дані, ніяк що не впливають на роботу алгоритму.

3.2 Приклад.

Розглянемо частину програми впорядкування масиву записів, що містять наступну інформацію студентів: код, прізвище, рейтинг.

```
const int SurLen=30;
struct stud{
     unsigned int id;
     char surname[SurLen];
     double rating;
     void stud_out()//метод для виведення об'єкта структури на екран
     {
          cout<<id<<surname<<rating<<endl;     //потрібно зробити
форматоване  виведення!
     }
};
```



```
void swap_char(char a[],char b[], int size) //процедура обміну
символьних масивів
{
     char temp;
     for (int i=0;i<size;i++)  {
          temp=a[i]; a[i]=b[i]; b[i]=temp;
     }
}
void swap(stud &S1, stud &S2) //процедура обміну даних типу «студент»
{
          swap(S1.rating,S2.rating);
          swap_char(S1.surname,S2.surname, SurLen);
}
int main() {
          int n = 3;
     stud M1[3];
     for (int i=0;i<n;i++) //введення масиву структур
     {//потрібно зробити зручний інтерфейс введення !
          M1[i].id=i;
          cout<<"surname?";cin>>M1[i].surname;
          cout<<"rating?";cin>>M1[i].rating;
     }
     //тут потрібно самостійно здійснити виведення даних на екран!
     for (int i = 0; i < n; i++) //впорядкування бульбашкою списку
студентів за рейтингом
     {
     for(int j = i + 1; j < n; j++)  {
          if (M1[i].rating > M1[j].rating) { swap(M1[i],M1[j]);}
          }
     }
//тут потрібно здійснити виведення відсортованих даних на екран!!
          return 0;
}
```

Ой! У відсортованому масиві не всі поля вірні. Самостійно знайти помилку і відлагодити поданий фрагмент програми.

3.3 Завдання для самостійного виконання.

1. Скласти програму для організації роботи зі структурою згідно зі своїм варіантом. Варіант V розраховується за формулою $V = N \bmod 6$. Організувати впорядкування K даних за вказаними ключами для демонстрації стійких і нестійких алгоритмів впорядкування (алгоритми впорядкування вибирати на свій смак).

2. Додати до своєї структури ще одне поле – дату (народження, приїзду, покупки і т.п.), яку оформити як структуру з трьома полями: день, місяць, рік.

3. Додати до своєї програми впорядкування за датою.

4. Реалізувати програмно можливість вибору користувачем ключа впорядкування. Тобто в результаті програма повинна працювати за схемою:,

- введення інформації;
- виведення даних в порядку створення;
- запит, в якому порядку хочемо їх бачити, наприклад,

0 – в попередньому порядку; 1 – впорядкованих за прізвищем,

2 – впорядкованих за роком народження, 3 – за рейтингом.

очевидно, виводить дані, впорядковані за вибраним пунктом.

Варіанти завдань

1. Інформація про K працівників підприємства задається рядком такого вигляду: номер паспорта, прізвище, спеціальність, стаж, середній оклад.

Ключі: а) номер паспорта, б) середній оклад. K=8.

2. Інформацію про T подорожніх містить такі поля: код білета (ціле число), прізвище пасажира,інформація про багаж – кількість речей і загальна вага.

Ключі: а) код білета, б) вага багажу. T=10.

3. Є інформація за підсумками іспитів в інституті, всього в списку M студентів. По кожному зі студентів є такі відомості: номер заліковки, прізвище, оцінка з математики, оцінка з інформатики та оцінка з фізики, рейтинг (обчислюється автоматично після введення оцінок як їх середнє арифметичне).

Ключі: а) номер заліковки, б) рейтинг. M=6.

4. Відомість про P працівників компанії містить таку інформацію: особистий номер (який виражається цілим числом), прізвище, вік, стаж, оклад.

Ключі: а) особистий номер, б) стаж. P=9.

5. Картотека відеотеки організована містить інформацію про S фільмів: код фільму (виражається цілим числом), назва фільму, вартість, режисер.

Ключі: а) код фільму, б) вартість. S=10.

6. Є інформація про учасників олімпіади з інформатики: серія паспорта, прізвище, назв навчального закладу, вік.

Ключі: а) серія паспорта. б) вік.

3.4 Питання для самоконтролю

1. В чому особливість впорядкування структурованих даних?
2. Як здійснюється звертання до полів структури?
3. Як здійснюється звертання до полів структури, яка є елементом масиву?
4. Який тип даних використовується для збереження даних різних базових типів?
5. Що таке «ключ сортування»?
6. Чи відрізнятимуться в загальному випадку результати впорядкування масиву за різними ключами? Відповідь обгрунтуйте?
7. Чи може бути ключовим
   - текстове поле?
   - числове поле?
   - логічне поле?
8. Які із вказаних полів можна вибирати ключовими:
   - номер по порядку,
   - прізвище,
   - середня оцінка,
   - номер паспорта,
   - адреса.
9. Порівняйте значення ключового поля для баз даних та ключового випадку для алгоритмів сортування.
10. В чому особливість процедури обміну двома елементами масиву структур?
11. Наведіть приклади масивів структурованих даних та можливі ключі для їх впорядкування.

Лабораторна робота №4

Вдосконалені алгоритми впорядкування

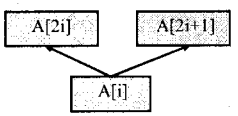
Мета: Набуття практичних навичок та вмінь роботи з логарифмічними алгоритмами впорядкування: пірамідальним, швидким, злиттям.

4.1 Теоретичні відомості

Пірамідальне сортування

Алгоритм пірамідального сортування використовує представлення масиву у вигляді дерева. Нехай A – деякий масив. Зіставимо йому дерево, використовуючи наступні правила:

1. A[1] – корінь дерева;



2. Якщо A[i] – вузол дерева та $2i \le n$, то A[2i] – «лівий син» вузла A[i]

3. Якщо A[i] – вузол дерева та $2i + 1 \le n$, то A[2i+1] – «правий син» вузла A[i]

Правила 1-3 визначають у масиві структуру дерева, причому глибина дерева не перевершує $\log_2 n + 1$. Вони ж задають спосіб руху по дереву від кореня до листків. Рух вгору задається правилом 4:

4. Якщо A[i] - вузол дерева та $i > 1$,то A[i mod 2] – «батько» вузла A[i] ;

Дерево, для елементів якого виконуються вказан умови, називається сортуючим.

Алгоритм HeapSort працює в два етапи:

I. Побудова сортуючого дерева;

II. Просівання елементів по сортуючому дереву.

Швидке сортування

Метод заснований на підході «розділяй і володарюй». Загальна схема така:

1) із масиву вибирається деякий опорний елемент a[i];
2) запускається процедура розділення масиву, яка переміщує всі ключі, менші, або рівні a[i], вліво від нього, а всі ключі, більші, або рівніа[i] – вправо;

3) тепер масив складається із двох підмножин, причому ліва менша, або рівна правій;
4) для обох підмасивів: якщо в підмасиві більше двух елементів, рекурсивно запускаємо для нього ту же процедуру.

В кінці отримаємо повністю відсортовану послідовність.

Впорядкування злиттям

Інколи виникає потреба працювати із сукупностями, які неможливо завантажувати у пам'ять в масиви. Такі сукупності знаходяться у файлах на зовнішніх носіях. Надалі будемо розглядати послідовні файли, тобто такі, для яких в кожний момент часу доступна лише одна компонента. Означену сукупність елементів будемо називати *послідовністю*.

Оскільки в послідовностях ми не маємо можливості стрибати по їх елементах, то попередні методи сортування застосовувати неможливо. Замінимо їх злиттям двох або більше упорядкованих підпослідовностей в одну упорядковану.

Ідея методу

1) Послідовність a розіб'ємо на дві частини b і c.
2) Частини b і c зливатимемо, при цьому одиночні елементи утворюватимуть упорядковані пари.
3) Одержана послідовність під іменем a знову оброблятьсяся, але упорядковані пари заміняються четвірками.
4) Повторюючи попепердні кроки, зливаємо четвірки у вісьмірки і т.д., кожний раз подвоюючи довжину злитих підпослідовностей до того часу, доки не буде упорядкована вся послідовність.

4.2 Приклад

Розглянемо функції, які реалізують вказані вище алгоритми. *Увага!* У прикладах пропущено деякі частини.

1. *Пірамідальне впорядкування*

а) Процедура просіювання k-го елемента через піраміду – масив розміру size

```
void downHeap(int a[], int k, int size)//
{
     int child; int tmp; tmp=a[k];
```

```
      while (k<size/2)//поки в даного елемента є нащадки
      {
      child=2*k+1; if((child<size-1)&&(a[child]<a[child+1])){child++;}
      if (tmp>=a[child]) {break;}//новий елемент більший за всі дані
піраміди
            a[k]=a[child];k=child; a[k]=tmp;
      }
}
```

б) Процес впорядкування масиву (N-розмір масиву)

```
//побудова з вхідного масиву піраміди
for (i=N/2-1;i>=0;i--) downHeap(a,i,N);
//власне сама процедура впорядкування
for (i=N-1; i>0;i--)
{
last=a[i];a[i]=a[0];a[0]=last;
???????//просіювання нульового елемента
}
```

2. *Швидке впорядкування*

```
void quickSort(int a[], long N) {
   // На вході - масив a[], N - номер останнього елемента.
   long i = 0, j = N; // поставити вказівники а початкові місця
   int temp, p;
    p = a[ N>>1 ]; // центральний елемент
   // процедура розділення
   do {
        while ( a[i] < p ) i++;
        while ( a[j] > p ) j--;

          if (i<= j) ?????здійснити обмін елементами

     } while ( i<=j );
    //     рекурсивні виклики, якщо є, що сортувати
    if ( j > 0 ) quickSort(a, j);
    if ( N >i ) quickSort(a+i, N-i);

    }
```

3. *Впорядкування злиттям*

а) Процедура злиття впорядкованих частин масиву у буфер-проміжний масив temp з подальшим перенесенням вмісту temp в a[left]...a[right]

```
 void merge(int a[], long left, long split, long right) {
      long pos1=left; // поточна позиція читання з першої
послідовності a[left]...a[split]
      long pos2=split+1; // поточна позиція читання з другої
послідовності a[split+1]...a[right]
      long pos3=0;   // поточна позиція запису в temp

   int *temp;

   temp = new int[right-left+1];
```



```
   // продовжується злиття, поки є хоча б один елемент в кожній
послідовності while (pos1 <= split && pos2 <= right)

    {
        if (a[pos1] < a[pos2]) { temp[pos3] = a[pos1]; pos3++;pos1++;}
        else {temp[pos3] = a[pos2]; pos3=pos3++;pos2++;}

        }
        //  одна послідовність закінчилась- копіювати залишок іншої в
кінець буфера
        while (pos1 <= split) // поки перша послідовність непорожня

        { temp[pos3] = a[pos1]; pos3++;pos1++;}

        while (pos2 <= right) // пока друга послідовність непорожня

        { temp[pos3] = a[pos2]; pos3++; pos2++; }
        // скопіювати буфер temp в a[left]...a[right]
        for (pos3 = 0; pos3 < right-left+1; pos3++)
        a[left+pos3] = temp[pos3];
        delete [] temp;
        }
```

б) Процедура безпосереднього впорядкування масиву

```
      void mergeSort(int a[], long left, long right) {

      long split;         // індекс, за яким ділимо масив

           if (left<right)// якщо є більше, ніж 1 елемент

          {

               split = (left + right)/2;
               mergeSort(a, left, split); // сортувати ліву половину

      ?????????????????????????? // сортувати праву половину

               merge(a, left, split, right);    // об'єднати масиви в один

      }
      }
```

4.3 Завдання для самостійного виконання

1. Проаналізувати приклади і доповнити частини, яких не вистачає.

2. Проілюструвати графічно роботу пірамідального сортування на прикладі впорядкування масиву з 10 цілих елементів.

3. Написати програму, яка впорядковує масив з 1000 випадкових елементів на вибір одним із методів: пірамідальним, швидким, злиттям.

4. Заповнити таблицю

Алгоритм	Час (в мілісекундах)			Додаткова пам'ять	Стійкість	Природність
	100N	1000N	10000N			
Пірамідальне впорядкування						
Швидке впорядкування						
Впорядкування злиттям						

4.4 Питання для самоконтролю

1. Що таке «піраміда» для пірамідального впорядкування масивів?
2. З яких етапів складається пірамідальне впорядкування?
3. Оцініть стійкість та природність пірамідального впорядкування.
4. Які переваги швидкого впорядкування масивів перед пірамідальним?
5. З яких етапів складається швидке впорядкування?
6. Оцініть стійкість та природність швидкого впорядкування.
7. У яких випадках доцільно застосовувати впорядкування злиттям?
8. Що відбувається при *i*-тій ітерації впорядкування злиттям?
9. Що відбувається при *i*-тій ітерації пірамідального впорядкування?
10. Що відбувається при *i*-тій ітерації швидкого впорядкування?
11. Оцініть стійкість та природність алгоритму пірамідального впорядкування.
12. Оцініть стійкість та природність алгоритму швидкого впорядкування.
13. Оцініть стійкість та природність алгоритму впорядкування злиттям.
14. Наведіть порівняльну характеристику алгоритмів пірамідального та швидкого впорядкування.
15. У яких випадках краще використовувати впорядкування злиттям?

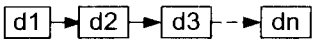
Лабораторна робота №5

Способи організації списків

Мета: ознайомитися із оснвними способами організації спискві та особливостями їх програмної реалізації; набути практичних навичок по роботі з простими списками двох типів: на основі масивів та на основі вказівників.

5.1 Теоретичні відомості

Лінійний список – це скінчена послідовність однотипних елементів (вузлів), можливо, з повторенням. Кількість елементів у послідовності називається довжиною списку. Вона в процесі роботи програми може змінюватися. Лінійний список L, що складається з елементів $d_1, d_2, \ldots, d_n$, які мають однаковий тип, записують у вигляді $L = \langle d_1, d_2, \ldots, d_n \rangle$, або зображають графічно.



Важливою властивістю лінійного списку є те, що його елементи можна лінійно впорядкувати у відповідності з їх позицією в списку.

Над лінійним списком допустимі наступні операції.

Операція вставки – вставляє елемент в конкретну позицію в списку, переміщуючи елементи від цієї позиції і далі в наступну, більш вищу позицію.

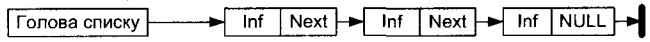
Операція локалізації – повертає позицію об'єкта в списку. Якщо в списку об'єкт зустрічається декілька разів, то повертається позиція першого від початку списку об'єкта. Якщо об'єкта немає в списку, то повертається значення, яке рівне довжині списку, збільшене на одиницю.

Операція вибірки елемента з списку – повертає елемент, який знаходиться в конкретній позиції списку. Результат не визначений, якщо в списку немає такої позиції.

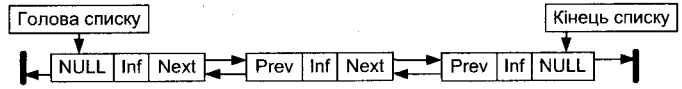
Операція вилучення – вилучає елемент в конкретній позиції зі списку. Результат невизначений, якщо в списку немає вказаної позиції.

Операції вибірки попереднього і наступного елемента – повертають відповідно наступній і попередній елемент списку відносно конкретної позиці в списку.

На наступному рисунку приведена структура однозв'язного списку. Кожний список повинен мати особливий елемент, який називається покажчиком на початок списку, або головою списку.



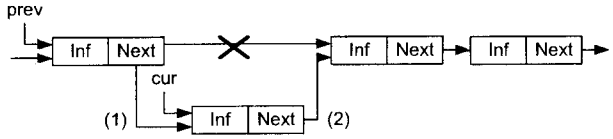
Проте, обробка однозв'язного списку не завжди зручна, оскільки відсутня можливість просування в протилежну сторону. Таку можливість забезпечує двохзв'язний список, кожний елемент якого містить два покажчики: на наступний і попередній елементи списку.



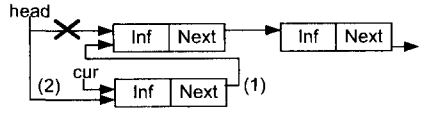
Для зручності обробки списку додають ще один особливий елемент – покажчик кінця списку. Наявність двох покажчиків в кожному елементі ускладнює список і приводить до додаткових витрат пам'яті, але в той же час забезпечує більш ефективне виконання деяких операцій над списком.

Вставка елемента в середину однозв'язного списку

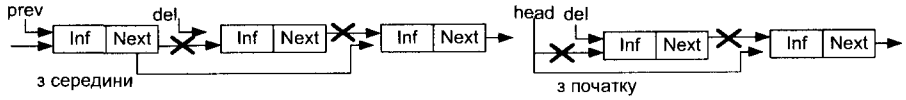
а) всередину списку



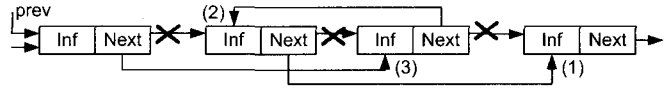
б) на початок списку на початок списку. При такій операції повинен модифікуватися покажчик на початок списку:



Видалення елемента з однозв'язного списку для двох варіантів – з середини і з голови



Перестановка елементів списку (без врахування початкового елемента)



5.2 Приклади програмної реалізації списків

1) За допомогою масивів

```
typedef int position;
class List {
      int EL[N];
      int last;
public:
      List(){last=-1;}
      void MakeNULL();//робить список порожнім
      void del(int p);//видаляє елемент з позиції p
      void insert(int x, position p);//вставляє елемент в позицію p
      void add(int x);//додає елемент до списку (в кінець)
      void print();//виводить список на екран
};
void List::add(int x)
{
      if (last==-1){EL[0]=x;last=1;}
      else {EL[last]=x;last++;}
}
```

2) За допомогою вказівників

```
struct element  //елемент списку із вказівником на наступний елемент
{
          int element_list;
          element *next;
};
class List      {
          element *header;
public:
          List();
          void add(int X); //аналогічно, як у попередньому випадку
          void print();//аналогічно, як у попередньому рядку :)
};
List::List() {  header=NULL;}
void List::add(int x) //функція. яка додає елемент x до голови списку
{
      element *p;
      p = new element;
      p->element_list=x;
      p->next=header;
      //if (header==NULL) {last=p;}
      header=p;
}
void List::print() //функція. яка виводить список на екран
{
      element *p;
```

```
        p=header;
        int k=0;
        while (p!=NULL)// поки не досягнуто кінець списку
        {
              k=k+1;
              cout<<p->element_list<<endl;
              p=p->next;
        }
        if (k==0) {cout<<"empty List"<<endl;} // повідомлення у випадку
порожнього списку
        else cout<<"end_of_list"<<endl;
    }
```

5.4 Завдання для самостійного виконання

1. Дописати функції для роботи зі списками. Зробити програми дієвими і продемонструвати роботу кожної із запропонованих функцій. Написати програму для об'єднання двох списків в один.

2. Організувати список з елементами, відповідними до завдань з лаб.роб. №3 (список структур). Видозмінити (де є необхідність) функції по роботі зі списком.

3. Написати програму впорядкування списків (ключ вибрати самостійно, метод впорядкування теж). Написати програму для об'єднання двох відсортованих за однаковим ключем списків в один відсортований список.

5.4 Питання для самоконтролю

1. Що представляє собою «список» як структура даних?
2. Які особливості організації даних у структурі «список»?
3. Які основні функції по роботі зі списками?
4. Які структурні елементи обов'язково містить список, а які – можуть бути додатковими?
5. В чому переваги та недоліки організації списків за допомогою масивів?
6. В чому переваги та недоліки організації списків за допомогою вказівників?
7. Наведіть порівняльну характеристику способів організації списків.
8. Наведіть приклади доцільності використання масивів для організації списку.
9. Наведіть приклади доцільності використання списку на основі вазівників.
10. У чому особливість кільцевих списків?

Лабораторна робота №6

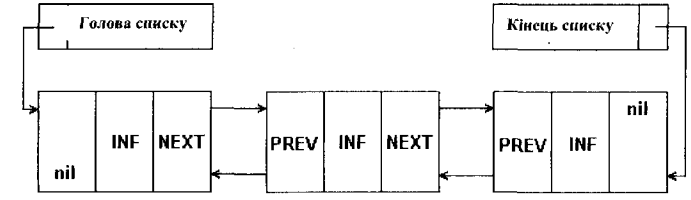
Робота з однозв'язними та двозв'язними списками

Мета: закріплення вмінь та навичок по опрацюванню списків різних видів.

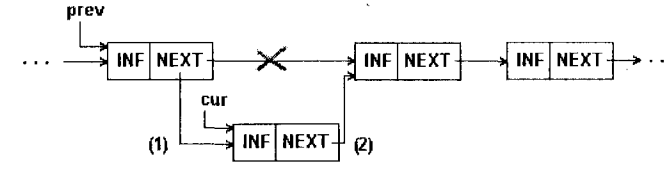
6.1 Теоретичні відомості

Представлення однозв'язного списку в пам'яті

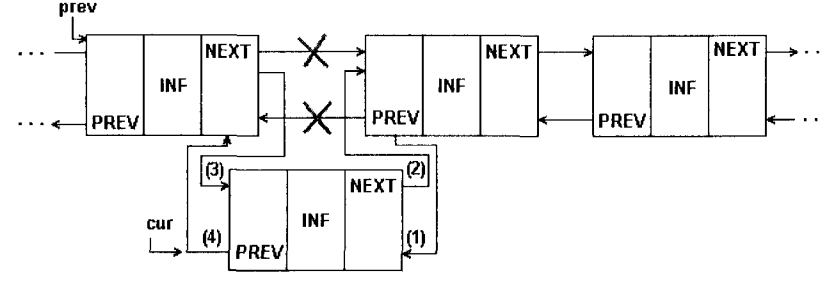
Представлення двозв'язного списку в пам'яті



Вставка елемента в однозв'язний список

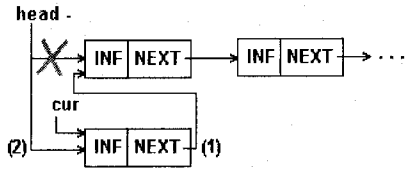


Вставка елемента в двозв'язний список



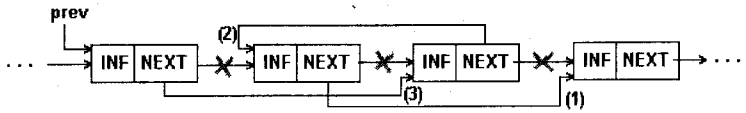
Однак такі алгоритми не працюють для вставки елемента на початок списку.

Вставка елемента в початок однозв'язного списку

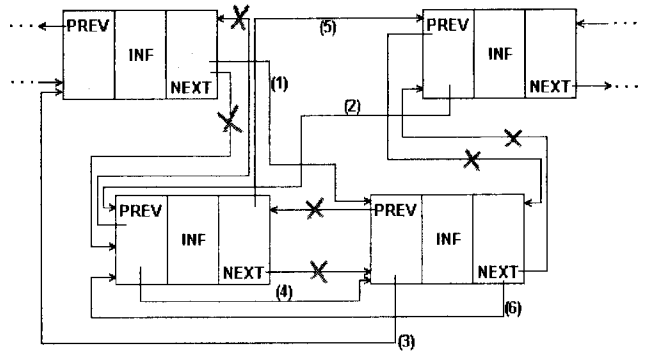


Перестановка сусідніх елементів списку

а) однозв'язний список



б) двозв'язний список



6.2 Приклад

Наведений фрагмент програми реалізує додавання елемента в кінець двозв'язного списку. В списку міститься наступна інформація про студентів – прізвище та рейтинг.

```
struct stud
{
    char surname[20];
    double rating;
};
struct cell
{
    stud data;
    cell *previous;
    cell *next;
};
```



```
class list
{
public:
    cell *header;//вказівник на початок списку
    cell *tail;//вказівник на кінець списку
public:
    list() {header=NULL;tail=NULL;}
    void add();
    void print();
};
void list::add()
{
    cell *p;
    p = new cell;
    cout<<"input surname"<<endl;
    cin >>p->data.surname;
    cout<<"input rating"<<endl;
    cin >>p->data.rating;
    if (header==NULL)
    {header=p; tail=p;tail->next=NULL;header->previous=NULL;}
    else {
         p->next=NULL;
         p->previous=tail;
         tail->next=p;
         tail=p;
         }
}
```

6.3 Завдання для самостійного виконання

1. Написати програму, яка демонструє основні дії по роботі із двозв'язним списком цілих чисел і пропонує користувачу вибір дії:

- очистити список
- додати елемент до списку (в кінець)
- вилучити вказаний елемент зі списку
- визначити кількість елементів у списку
- поміняти два сусідні елементи місцями, які слідують за позицією *p*
- об'єднати два списки в один
- з утворенням нового списку
- шляхом дописування другого списку в кінець першого списку

При виконанні всіх дій передбачити відображення поточного стану списку. Спосіб реалізації списку – на вибір.

2. *Приклад реалізації утворення нового списку шляхом злиття двох списків*

```
list merge_lists(list LL1,list LL2)
{
   list LL;
   cell *ptr;
   if(LL2.header!=NULL)
   {if (LL1.header==NULL){LL.header=LL2.header;}
        else
        {
             ptr=LL1.header;
             while(ptr->next!=NULL){ptr=ptr->next;}
             LL2.header->previous=ptr;
             ptr->next=LL2.header;
        }
   }
   LL.header=LL1.header;
   return LL;
}
```

Завдання. Реалізувати аналогічну програму для роботи зі своїм варіантом структури з реалізацією наступних функцій:

- Одержати доступ до k-го вузла списку.
- Включити новий вузол безпосередньо перед k-им вузлом.
- Об'єднати два (або більше) лінійних списки в один список.
- Розбити лінійний список на два (або більше) списки.
- Зробити копію лінійного списку.
- Визначити кількість вузлів у списку.
- Знайти в списку вузол із заданим значенням у деякім полі.

6.4 Питання для самоконтролю

1. Що таке список як структура даних?
2. Які є способи організації списків в пам'яті комп'ютера?
3. Які основні функції по роботі зі списками?
4. Які структурні елементи обов'язково містить список?
4. Які структурні елементи додатково може містити список?
6. Переваги і недоліки організації списків на основі масивів.
7. Переваги і недоліки організації списків за допомогою вказівників.
8. Для яких задач доцільно зберігати списки у вигляді масивів?
9. Для яких задач доцільно організовувати списки за допомогою вказівників?
10. Наведіть порівняльний аналіз вивчених способів організації списків.

Лабораторна робота №7

Стеки та черги

Мета: набуття практичних вмінь та навичок по роботі з стеками та чергами.

7.1 Теоретичні відомості.

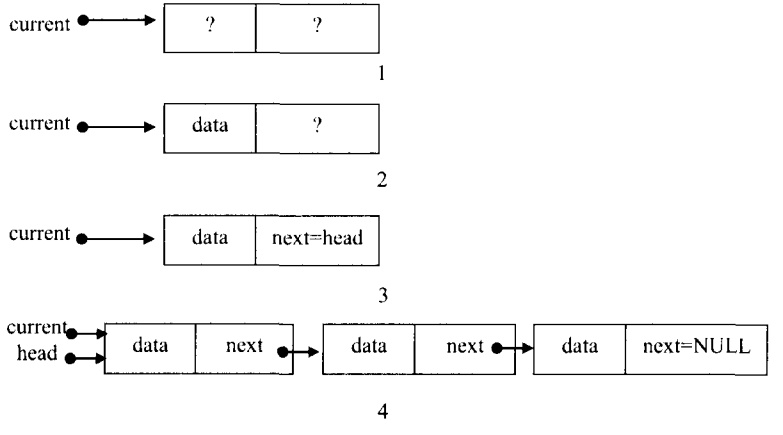
Стек (stack) – лінійний список, у якому всі видалення й доповнення (і звичайно всякий доступ) робляться з одного кінця списку.

У стеці елемент додаються й віддаляються тільки з одного кінця. На малюнку це елемент N. Тобто якщо він додався, то видалятися може спочатку тільки він, а вже потім всі інші.

Для стеку характерні такі властивості:

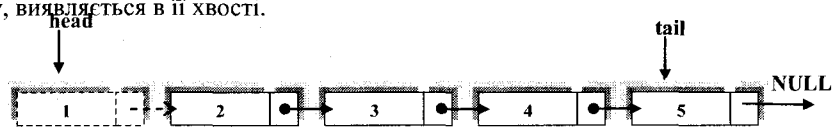
- елементи додаються у вершину (голову) стеку;
- елементи видаляються з вершини (голови) стеку;
- покажчик в останньому елементі стеку дорівнює NULL;
- неможливо вилучити елементи стеку, не вилучивши всі елементи, що йдуть попереду.

Графічне представлення алгоритму вставки елемента до стеку



Черга (queue) – лінійний список, у якому всі видалення відбуваються на одному кінці списку, а всі включення (і звичайно всякий доступ) робляться на іншому його кінці.

Інакше кажучи, у черги є голова (head) і хвіст (tail). Елемент, що додається в чергу, виявляється в її хвості.



У черзі новий елемент додається тільки з одного кінця. Видалення елемента відбувається на іншому кінці. Черга, це по суті однонаправлений список, тільки додавання й видалення елементів відбувається на кінцях списку.

Черга характеризується такими властивостями:

елементи додаються в кінець черги;

елементи зчитуються та видаляються з початку (вершини) черги;

покажчик в останньому елементі черги дорівнює NULL;

неможливо отримати елемент із середини черги, не вилучивши все елементи що ідуть попереду.

7.2 Приклад організації стеків

```
struct cellOfStack {
     char liter;
     cellOfStack *next;
};
class Stack {
     cellOfStack *top; // вказівник на вершину стеку
public:
     Stack(){top=NULL;}
     void add(char x); // Push
     void del(); // Pop;
     char Top();
     void print();
};
void Stack::add(char x){
     cellOfStack *p;
     p= new cellOfStack;
     p->liter=x;
     if (top==NULL){top=p;top->next=NULL;} else {p->next=top;top=p;}
}
int main()
{
     Stack S;
```



```
     for (int i=0;i<=10;i++)//дайте собі відповідь на питання: що робить цей цикл?
     {    S.add(10*i);}
          S.print();//переконайтеся у правильності своєї відповіді
          S.del(); // Ooooooops!Щось у програмі бракує
          S.print();//що сталося зі стеком?
          return 0;
}
```

А ось і черги

```
struct CellOfQueue {
     int  number;
     CellOfQueue *next;
   };
class Queue {
     public:
     CellOfQueue *header;
     CellOfQueue *tail;
     Queue() {header=NULL; tail=NULL;}
     void add (int x); //EnQueue
     void print();
     int front();    //Front
     void del();//DeQueue
};

void Queue::add (int x) {
     CellOfQueue *p;
     p=new CellOfQueue;
     p->number=x;
     p->next=NULL;
     cout<<"added"<<p->number<<endl;
     if (header==NULL){header=p;tail=p;tail->next=NULL;}
      else if (header==tail){header->next=p;tail=p;}
               else{
                    tail->next=p;
                    tail=p;
                    }
}
void Queue::del() { header=header->next; }
```

7.3 Завдання для самостійного виконання

1. Написати програму, яка передбачає введення в стек цілих чисел з клавіатури. Вивести стек на екран. Який порядок слідування чисел?

2. Створити і роздрукувати два стеки – з парними та непарними числами серед введених.

3. Будемо розглядати послідовності круглих і квадратних дужок, що відкриваються і і закриваються ( ) [ ]. Серед усіх таких послідовностей виділимо правильні – ті, котрі можуть бути отримані за такими правилами:

– визначення розміру черги;

– очищення черги.

При роботі з пріоритетними чергами частіше говорять не про елементи, а про запити, що записують у чергу. При цьому можлива організація черг із пріоритетним включенням елемента в чергу і з пріоритетним виключенням запиту з черги. Ми розглянемо перший випадок.

Для черги з пріоритетним включенням послідовність запитів у черзі увесь час підтримується упорядкованою, тобто кожен новий елемент включається на те місце в послідовності, що визначається його пріоритетом.

Алгоритм пріоритетного включення передбачає так дії:

– якщо черга неповна, знайти згідно з пріоритетом таке місце в черзі, де треба поставити запит та внести запит до черги.

При виключенні завжди вибирається запит із початку черги.

Зауважимо, при виборі елемента з пріоритетної черги кожного разу вибирається елемент із найбільшим (найменшим) пріоритетом.

8.2.Приклад. Спробуємо без прикладу, бо основа є з попередньої лабораторної роботи :)))

8.3. Завдання для самостійного виконання

1. Реалізувати основні функції по роботі з деками, в яких записані цілочисельні дані. Ввести функцію контролю (виводу) поточного стану дека. Передбачити недопустимість помилок при некоректних діях (видалення елемента з порожнього деку).

2. Перевірити, чи введений рядок є паліндромом.

3. Написати програму для роботи з чергою із пріоритетним включенням. Елементом черги зробити свій варіант структури (див. лаб. роб. №3). Пріоритетними вважати:

- Стаж працівника фірми (у роках)
- Кількість речей у багажі.
- Оцінка з інформатики
- Вік (у роках)

- Вартість
- Вік (у роках)

4. Організувати демонстрацію програми для створення черги із 5 елементів.

Зауваження 1. Додавання елементу до такої черги відрізняється від додавання елементу до звичайної черги необхідністю пошуку відповідного місця в черзі

Зауваження 2. Чергу можна реалізувати або на основі двозв'язного списку, або на основі кільцевого однозв'язного списку.

5. Використання стеків для обчислення виразів

Існують три способи запису складних виразів. Для двомісних операцій (тут - □) існують три способи запису:

префіксний (функціональний): **□ x y;**

інфіксний (шкільний): **x □ y;**

постфіксний (польський): **x y □;**

Всі три означають, що треба взяти операнди **x** та **y** (які можуть бути числами, тобто константами, змінними або іншими виразами) та виконати з ними операцію □.

Приклад: вираз

(((A - B) * C) + (D / (E + F)))

Звичайний (шкільний) спосіб - це інфіксний.

Для даного прикладу її префіксна та постфіксна форми мають вигляд:

префіксна:

+ * - ABC / D + EF

постфіксна:

AB - C * DEF + / +

Обидві форми однозначно визначають порядок обчислення та не вимагають дужок. Префіксна структура дуже зручно реалізується за допомогою рекурсивних процедур. Постфіксна структура краще за все реалізується за допомогою стеку.

Розглянемо реалізацію постфіксної форми за допомогою стеку. Вираз 1 + 2 у постфіксній формі записується як 1 2 +, а вираз (1 + 2)*4 — як 1 2 + 4 *. В

постфіксній формі порядок виконання операцій визначається виключно їх розташуванням в рядку; відпадає необхідність у використанні дужок і у такому понятті, як пріоритет операцій.

Для обчислення представлених у постфіксній формі виразів достатньо скористатися наступним простим алгоритмом: операнди, які зустрічаються у вхідному рядку, поміщаються в стек, а операції, які зустрічаються, виконуються над двома верхніми значеннями зі стеку з розміщенням результату в стек. Таким чином, при коректному записі виразу в результаті введення рядка на вершині стеку міститиметься результат обчислень.

Вхідний рядок		2	2 +	1 2 + 4	1 2 + 4 *
Стек				4 3	12

Завдання. Перетворити наведені нижче вирази у постфіксну форму і реалізувати програму за допомогою стеку:

а) (A-B-C)/D-E*F

б) (A+B)*C-(D+E)/F

в) A/(B-C)+D*(E-F)

г) (A*B+C)/D-F/E

8.4 Питання для самоконтролю

1. Що таке «черга»?
2. Що таке «дек»?
3. На основі яких структур даних доцільно організовувати деки?
4. Які основні функції по роботі з деками.
5. Що таке «черга з пріоритетом»?
6. Які основні функції по роботі з чергами з пріоритетом?
7. Який пріоритет у класичних черг?
8. Наведіть приклади організації черг та можливих пріоритетів для них.

Лабораторна робота № 9

Бінарні дерева

Мета: Набуття практичних вмінь та навичок опрацювання нелінійних структур даних, представлених у вигляді бінарних дерев.

9.1 Теоретичні відомості

Дерево – це нелінійна структура даних, яка характеризується наступними властивостями:

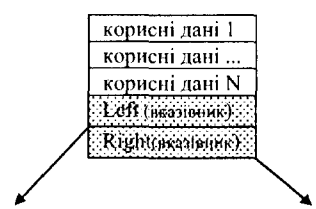
- Існує єдиний елемент (вузол або вершина), на який не посилається ніякий інший елемент, – він називається коренем.
- Починаючи з кореня і слідуючи по певному ланцюжку покажчиків, що містяться в елементах, можна здійснити доступ до будь-якого елемента структури.
- На кожний елемент, крім кореня, є єдине посилання, тобто кожний елемент адресується єдиним покажчиком.

Правило побудови бінарного дерева з будь-якого дерева

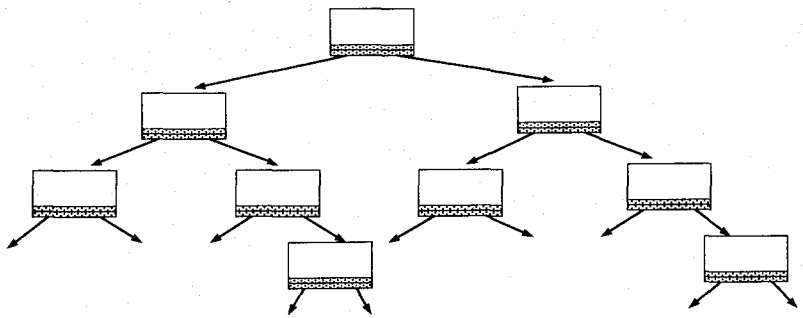
1. В кожному вузлі залишити тільки гілку до старшого сина;
2. З'єднати горизонтальними ребрами всіх братів одного батька;
3. Таким чином перебудувати дерево за правилом:
   - лівий син – вершина, розташована під даною;
   - правий син – вершина, розташована праворуч від даної (тобто на одному ярусі з нею).
4. Розвернути дерево так, щоб усі вертикальні гілки відображали лівих синів, а горизонтальні – правих.

Структурна одиниця дерева – *вузол*:

```
struct DATA {
        int x;
        int y;
        int z;          };
struct NODE {
        DATA data;
        ELEMENT *Left;
        ELEMENT *Right; };
```



в програмі ми можемо будувати бінарне дерево так:



Якщо гілка не має на собі вузлів, то обов'язково її значення приймається рівним NULL.

Порівняння ефективності використання різних структур даних

	Впорядкований масив	Лінійний список	Бінарне дерево
	Оцінка часу	Оцінка часу	Оцінка часу
пошук елемента за значенням	$O(\log N)$	$O(N)$	$O(\log N)$
додавання елемента: – знаходження місця – розміщення в знайдене місце	$O(\log N)$ $O(N)$	$O(N)$ $O(1)$	$O(\log N)$ $O(1)$
видалення елемента	$O(N)$	$O(1)$	$O(1)$

9.2 Приклад

Написати програму, в якій дані записуються в дерева. Вузол дерева містить таку корисну інформацію: ціле число a та дійсне число b. Дані вводити до тих пір, поки потрібно користувачеві. Роздрукувати отримане дерево. Зауважити, що дані виводяться вже впорядкованими за полем a.

Сьогодні спробуємо використати заголовковий файл, в якому організуємо всю роботу, пов'язану з бінарним деревом.

Файл Tree.h

```
struct DATA {
      int a;
      double b;
};
struct NODE {
      DATA data;
      NODE *Left;
      NODE *Right;
      };
void Add (DATA t, NODE* &c);
void ShowElement(NODE* c);
void ShowTree(NODE* c);
int CountElements(NODE* c);
void Add(DATA d, NODE* &c){
      if (c==NULL)
      {     c = new NODE;
            c->data = d;
            c->Left = NULL;
            c->Right = NULL;
            return;
      }
      // -- якщо с - непорожній вказівник ---
      if (d.a < c->data.a)
            Add(d, c->Left);
      else
            Add(d, c->Right);
}
void ShowElement(NODE* c){ if (!c) return;
      cout.width(10); cout << c->data.a << ", ";
      cout.width(10); cout << c->data.b << " ";
      cout << endl;
}
void ShowTree(NODE* c){
      if (c->Left) ShowTree(c->Left);
      ShowElement(c);
      if (c->Right) ShowTree(c->Right);
}
int CountElements(NODE* c){
      int q = 0;
      if (!q) return 0;
      if (c->Left) q+=CountElements(c->Left);
      q++;
      if (c->Right) q+=CountElements(c->Right);
      return q;
}
```

Файл основної програми

```
//.....
#include "Tree.h"
int main(){
      DATA t;
      NODE *T = NULL;
   int _;
      while (true){
            cout << "ВВОДИТЕ ДАНІ (1/0)? ";
            cin >> _;
            if (!_) break;
            cout << "Enter a:";   cin >> t.a;
            cout << "Enter b:";   cin >> t.b;
```

```
            Add(t, T);
        }
        ShowTree(T);
//.....
}
```

9.3 Завдання для самостійного виконання

1. Написати програму, в якій дані вашого варіанту структури записуються в дерева (використати три поля для вузла – текстове дане та два числові. Наприклад, вузол дерева містить таку корисну інформацію: прізвище студента, рік народження, оцінка). Ввести з клавіатури декілька "студентів" у двійкове дерево, організоване за порядком текстового поля. Роздрукувати отримане дерево.

2. Знайти середню значення одного з числових полів, зчитуючи дані з дерева.

3. Дописати функцію видалення з пам'яті всього дерева

Вказівка: рекурсивна функція, яка

1) видаляє ліве піддерево, і ліву гілку занулює;

2) видаляє праве піддерево, і праву гілку занулює;

3) видаляє сам вузол, потім зануливши вказівник на нього

Наприкінці програми видалити з пам'яті дерево.

4. "Пересипати" дані з першого дерева у друге дерево того ж типу, тільки організованого за першим числовим ключем (напр., роком народження) та роздрукувати його (а перше дерево стерти).

9.4 Питання для самоконтролю

1. Що таке «дерево» як структура даних?
2. Що таке «корінь дерева»? глибина дерева?
3. Які основні властивості дерева?
4. Яке дерево називають *n*-арним? бінарним?
5. Чи можна з *n*-арного дерева побудувати бінарне? Якщо можна, то як?
6. Які основні функції по роботі з деревами?
7. Яким чином при роботі зі структурою «дерево» використовуються рекурсивні алгоритми?
8. Які особливості організації вузла дерева?
9. Яким чином дерева використовують для впорядкування даних?

Лабораторна робота № 10

Способи задання графів

Мета: Набуття практичних вмінь і навичок при представленні заданих графів різними способами та можливістю їх комп'ютерної реалізації.

10.1 Теоретичні відомості.

Нехай граф $G=(V,E)$ містить вершини v_i та ребра e_j, $i=1,...,n, j=1,...,m$.

Визначимо відповідність Γ, яка вказує, як зв'язані між собою вершини.

Відповідність Γ є багатозначним відображенням множини V в множину. Тоді граф можна позначати $G=(V,\Gamma)$.

Для простих графів відповідність Γ визначає для кожної вершини v_i суміжні з нею.

Для псевдографів відповідність Γ визначає для кожної вершини v_i суміжні з нею та додає v_i, якщо в цій вершині існує петля.

Для орієнтованих графів відповідність Γ визначає для кожної вершини v_i вершини v_j, якщо існує ребро e_{ij}, v_j та додає вершину v_i, якщо в цій вершині існує петля.

Способи задання графів

Графічний

Вершини графа представляються точками чи кругами, а ребра – відрізками, які їх з'єднують.

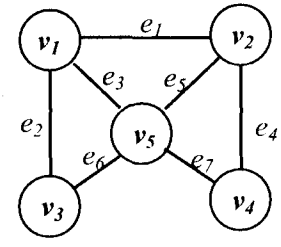


Рисунок 10.1 – Простий граф

Графічний спосіб є дуже наглядним, однак непридатним для представлення графів у пам'яті комп'ютера.

Матриця інцидентності $M = \|m_{ij}\|$, $i = 1,...,n, j = 1,...,m$.

Для простих графів:

$$m_{ij} = \begin{cases} 1, \text{ якщо вершина } v_i \text{ та дуга } e_j \text{ інцидентні} \\ 0 \text{ у протилежному випадку} \end{cases}$$

Для простого графа нема однакових стовпців і в кожному стовпці є точно по дві 1.

Для мультографа – будуть однакові стовпці.

Для псевдографів для позначення петлі використовують 2.

Для орієнтованих графів:

$$m_{ij} = \begin{cases} 1, \text{ якщо дуга } e_j \text{ виходить з вершини } v_i \\ -1, \text{ якщо дуга } e_j \text{ входить у вершину } v_i \\ 2, \text{ якщо дуга } e_j - \text{петля у вершині } v_i \\ 0 \text{ в інших випадках} \end{cases}$$

Для графа, зображеного на рис.10.1:

$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Примітка: Серед наведених прикладів один містить помилку. Хто перший її вивить – додатковий бал. :)

Матриця суміжності $A = \|a_{ij}\|$, $i = 1,...,n$

Для простого графа

$$a_{ij} = \begin{cases} 1, \text{ якщо } v_i, v_j \in E \\ 0 \text{ у протилежному випадку} \end{cases}$$

Для неорієнтованого графа – симетрична і $a_{ij} = 0, i = j$.

Для псевдографа елемент a_{ij} дорівнює кількості ребер, що з'єднують відповідні вершини. Петля – $a_{ij} = 1, i = j$.

Для орієнтованого графа – аналогічно (тільки матриця, взагалі кажучи, несиметрична).

Для орієнтованого мультографа елементи матриці рівні кількості ребер, що з'єднують відповідні вершини.

Для графа, зображеного на рис.10.1:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

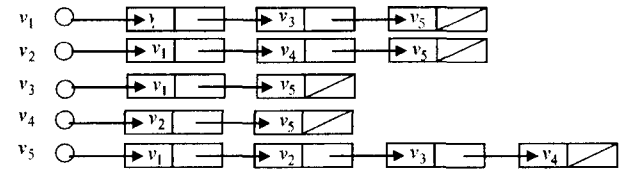
Список ребер (найекономніший спосіб щодо пам'яті) – список пар, що відповідають ребрам графа.

До нашого прикладу:

v_1	v_2
v_1	v_3
v_1	v_5
v_2	v_4
v_2	v_5
v_3	v_5
v_4	v_5

Списки суміжності – задання графа відповідністю Γ. Для цього використовується масив списків (по одному списку на кожну вершину), який для кожної вершини v_i містить у довільному порядку (вказівники на) вершини з множини $\Gamma(v_i)$.

До нашого прикладу:



Списки суміжності доцільно використовувати у тих випадках, коли матриця суміжності є сильно розрідженою.

10.2 Завдання для самостійного виконання.

Для неорієнтованих та орієнтованих графів зі своїх варіантів (варіант вибирається згідно номера по журналу, варіанти завдань наведено у додатку Б) виконати наступні завдання:

1. Створити програму, яка буде зберігати в комп'ютері заданий граф наступними способами:

- а) матрицею суміжності
- б) матрицею інцидентності
- в) списком ребер
- г) списком суміжності.

Передбачити вивід кожного представлення на екран.

2. Написати програму, яка виконуватиме наступні завдання:

- за заданою матрицею суміжності побудувати матрицю інцидентності;
- за заданою матрицею інцидентності побудувати список ребер;
- за заданою матрицею суміжності побудувати список суміжності.
- за заданою матрицею інцидентності побудувати матрицю суміжності;
- за заданою матрицею суміжності побудувати список ребер;
- за заданою матрицею інцидентності побудувати список суміжності.

10.4 Питання для самоконтролю

1. Що таке «граф»?
2. Які є способи представлення графів?
3. Які особливості комп'ютерної реалізації графів?
4. Чим відрізняються матриця суміжності від матриці інцидентності?
5. В яких випадках задання графа матрицею суміжності не є доцільним?
6. Які графи називають простими? мультиграфами? псевдографами? орієнтованими? навантаженими?
7. Оцініть об'єм пам'яті, необхідний для збереження графів у вигляді: матриці суміжності, матриці інцидентності, списку ребер, списку суміжності.

Лабораторна робота № 11

Способи обходу графів: DFS та BFS алгоритми

Мета: Набуття практичних вмінь та навичок по використанню алгоритмів обходу графів: алгоритму в глибину та алгоритму в ширину.

11.1Теоретичні відомості.

Вільним деревом називають дерево, в якому невизначений корінь або ж (що те саме) ациклічний зв'язний граф.

Каркасаним деревом графа *G* називається вільне дерево, яке містить всі вершини даного графа.

В процесі обходу графа методом пошуку в глибину будується каркасне дерево, яке називають *глибинним каркасним деревом (лісом).*

Для простих графів:

При цьому ребра, які ведуть до вершин, що раніше не відвідувалися, називають *ребрами глибинного дерева.*

Зворотні ребра – ребра («штрихові»), які ведуть до вершин, що вже відвідувалися.

Для орграфів:

Дуги глибинного дерева – дуги, які в каркасному дереві ведуть від предків до нащадків.

Прямі дуги – дуги, які йдуть від предків до істинних нащадків, але не є дугами каркасного дерева («штрихові»)

Зворотні дуги – дуги, ведуть від нащадків до предків глибинного каркасного дерева («штрихові»). Дуга, яка веде із вершини в саму себе, теж називається зворотньою.

Поперечні дуги – дуги, які пов'язують вершини, що не є ні предками, ні нащадками одна одної («штрихові»).

Пошук в глибину у простому зв'язному графі

Пошук вглиб або DFS-метод (Dipth First Search)

Нехай $G=(V, E)$ – простий зв'язний граф, усі вершини якого позначені попарно різними символами. У процесі пошуку вглиб вершинам надають DFS-номери та певним чином помічають ребра.

Крок 1. Почати з довільної вершини v_s. Покласти DFS(v_s)=1. Включити цю вершину у стек.

Крок 2. Розглянути вершину, яка знаходиться у верхівці стека:. Нехай це буде вершина *x*. Якщо всі ребра, інцидентні цій вершині, позначені, то перейти до кроку 4, інакше – до кроку 3.

Крок 3. Нехай {*x*, *y*} – непозначене ребро. Якщо DFS(*y*) уже визначений, то це ребро позначити штриховою лінією і перейти до кроку 2. Якщо DFS(*y*) не визначений, то дане ребро позначити суцільною лінією, визначити DFS(*y*) як черговий DFS-номер, включити цю вершину до стека і перейти до кроку 2.

Крок 4. Виключити вершину *x* зі стека. Якщо стек порожній, то зупинитись, інакше – перейти до кроку 2.

Для однозначності вибору номерів доцільно домовитись, що аналіз вершин, суміжних з вершиною, яка вже отримала DFS-номер, здійснюють за зростанням їх порядкового номера (або ж у алфавітному порядку) .

Динаміку DFS-обходу відображають за допомогою таблиці з трьома стовпцями: вершина, DFS-номер, вміст стека. Цю таблицю називають *DFS-протоколом.*

Пошук в ширину або BFS-метод (Breadth First Search)

Нехай $G=(V, E)$ – простий зв'язний граф, усі вершини якого позначені попарно різними символами. У процесі пошуку вглиб вершинам надають DFS-номери та певним чином помічають ребра.

Крок 1. Почати з довільної вершини v_s. Покласти BFS(v_s)=1. Включити цю вершину у чергу.

Крок 2. Розглянути вершину, яка знаходиться на початку черги: нехай це буде вершина *x*. Якщо для всіх вершин, мімічних з нею , вже визначені BFS-номери, то перейти до кроку 4, інакше – до кроку 3.

Крок 3. Нехай *y* –вершина, для якої ще не визначений DFS-номер. Ребро {*x*, *y*} позначити суцільною лінією, визначити BFS(*y*) як черговий BFS-номер, включити цю вершину до черги і перейти до кроку 2.

Крок 4. Виключити вершину *x* з черги. Якщо черга порожня, то зупинитись, інакше – перейти до кроку 2.

Аналогічно, як і для DFS-обходу, для однозначності вибору номерів доцільно домовитись, що аналіз вершин, суміжних з вершиною, яка вже отримала BFS-номер, здійснюють за зростанням їх порядкового номера (або ж у алфавітному порядку) .

Динаміку BFS-обходу nt; відображають за допомогою таблиці з трьома стовпцями: вершина, BFS-номер, вміст черги. Цю таблицю називають *BFS-протоколом.*

11.2 Приклади

1) Пошук в глибину – аналог обходу дерев у прямому порядку.

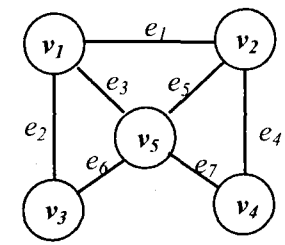


Рисунок 11.1 – Простий граф

Результат DFS-обходу, починаючи з вершини v_1 : v_1-> v_2-> v_4-> v_5-> v_3

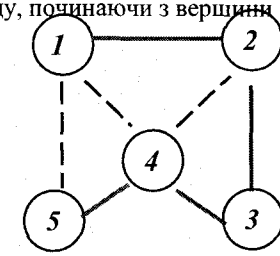


Рисунок 11.2 – DFS-обходу, починаючи з вершини v_1

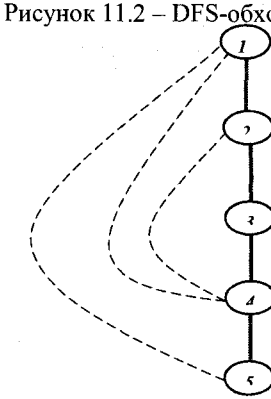


Рисунок 11.3 – Результат DFS-обходу, починаючи з вершини v_1 у вигляді дерева

В даному випадку, DFS(v_1)=1, DFS(v_2)=2, DFS(v_3)=5, DFS(v_4)=3, DFS(v_5)=4.

2) **Пошук в ширину.**

Результат BFS-обходу, починаючи з вершини v_1: v_1 -> v_2 -> v_5 -> v_3 -> v_4

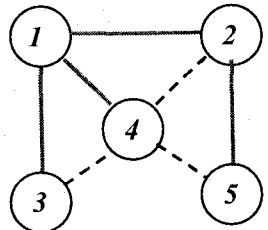


Рисунок 11.4 – Результат BFS-обходу, починаючи з вершини v_1

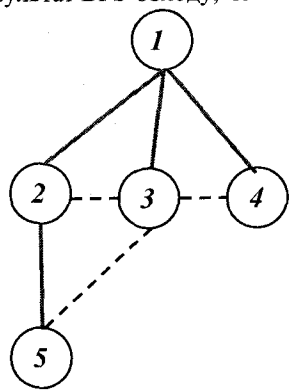


Рисунок 11.5 – Результат BFS-обходу, починаючи з вершини v_1, у вигляді дерева

В даному випадку: BFS(v_1)=1, BFS(v_2)=2, BFS(v_3)=3, BFS(v_4)=5, BFS(v_5)=4.

3) Для орієнтованих графів

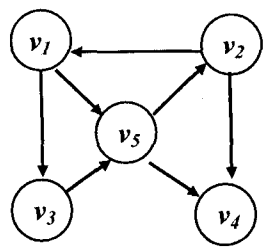


Рисунок 11.6 – Оргграф

а) DFS: v_1 -> v_3 -> v_5 -> v_4 -> v_2

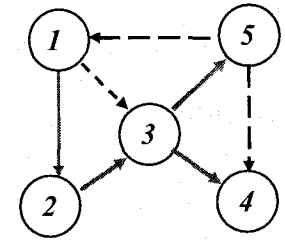


Рисунок 11.7 – Результат DFS-обходу, починаючи з вершини v_1

DFS(v_1)=1, DFS(v_2)=5, DFS(v_3)=2, DFS(v_4)=4, DFS(v_5)=3.

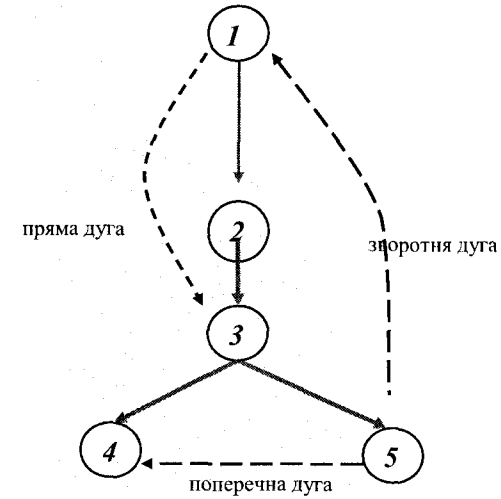


Рисунок 11.8 – Результат DFS-обходу, починаючи з вершини v_1, у вигляді дерева

б) BFS: v_1 -> v_3 -> v_5 -> v_2 -> v_4

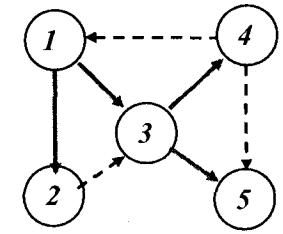


Рисунок 11.9 – Результат BFS-обходу, починаючи з вершини v_1

BFS(v_1)=1, BFS(v_2)=4, BFS(v_3)=2, BFS(v_4)=5, BFS(v_5)=3.

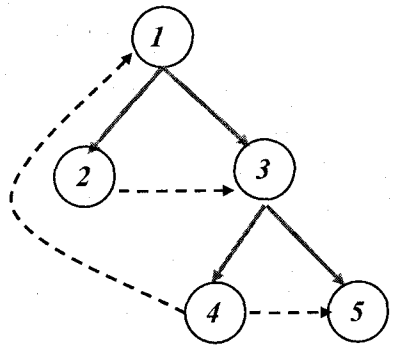


Рисунок 11.10 – Результат BFS-обходу, починаючи з вершини v_1

11.3 Завдання для самостійного виконання

1. Виконати обходи в глибину та в ширину графів зі своїх варіантів, оформивши результати, як показано вище. Номер варіанту вибирати згідно номера по журналу. Варіанти завдань наведено у додатку Б.
2. Написати програмну реалізацію даних алгоритмів.

11.4 Питання для самоконтролю

1. Навіщо використовувати алгоритми обходу графів?
2. Яку структуру отримуємо внаслідок здійснення обходу графа в глибину або в ширину?
3. Розкрийте особливості DFS-алгоритму.
4. Розкрийте особливості BFS-алгоритму.
5. Наведіть порівняльну характеристику DFS та BFS алгоритмів.
6. При використанні якого алгоритму обходу графа використовується стек?
7. При використанні якого алгоритму обходу графа використовується черга?
8. Який функції по роботі з графом доцільно передбачити для реалізації DFS- та BFS- алгоритмів?
9. Що таке «каркасне дерево» і яке його призначення?
10. Наведіть приклади доцільності застосування DFS та BFS алгоритмів.

Лабораторна робота № 12

Пошук найкоротших шляхів на графах

Мета: набуття практичних вмінь і навичок з використання алгоритмів Дейкстри та Флойда.

12.1 Теоретичні відомості.

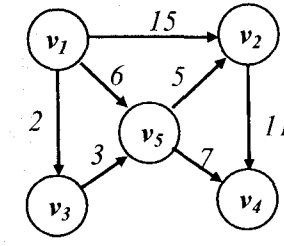
Пошук найкоротших шляхів від джерела – алгоритм Дейкстри

Задача знаходження найкоротшого шляху з одним джерелом полягає у знаходженні найкоротших (мається на увазі найоптимальніших за вагою) шляхів від деякої вершини (джерела) до всіх вершин графа G. Для розв'язку цієї задачі використовується «жадібний» алгоритм, який називається алгоритмом Дейкстри.

«Жадібними» називають алгоритми, які на кожному кроці вибирають оптимальний із можливих варіантів.

Алгоритм Дейкстри будує множину U вершин, для яких уже відомі найкоротші шляхи. На кожному кроці до цієї множини додається та із вершин, які залишилися, відстань до якої від джерела менша, аніж до інших вершин.

Нехай граф G заданий матрицею суміжності, в якій визначені вартості його дуг (тобто задана матриця ваг).



Матриця суміжності (ваг): $C = \begin{pmatrix} 0 & 15 & 2 & 0 & 6 \\ 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 7 & 0 \end{pmatrix}$

Будемо в таблицю D з $n-1$ елементів записувати на кожному кроці найкоротші шляхи до кожної вершини від джерела, які проходять тільки через

вершини з множини U. Відстань до вершини v_j позначатимемо $D[v_j]$ або просто $D[j]$. Якщо шляху між вершинами не існує, то відповідний елемент матриці C покладемо рівним ∞.

$$C=\begin{pmatrix} 0 & 15 & 2 & \infty & 6 \\ \infty & 0 & \infty & 11 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 7 & 0 \end{pmatrix}$$

Псевдокод алгоритму Дейкстри

$U = v_1$
для $i = 2$ до n
{
$D[i] = C[1,i]$
}
для $i = 1$ до $n-1$
{
вибір з множини $V \setminus U$ такої вершини w, для якої $D[w]$ мінімальне;
приєднання w до множини U;
для кожної вершини v з множини $V \setminus U$
{
$D[v] = \min\ D[v], D[w] + C[w,v]$
}
}
Позначатимемо вершини v_j тільки їх номером j.

Крок 0 (ініціалізація)

Записуємо в множину U джерело 1 та в таблицю D – відстані від джерела до всіх інших вершин

$D[2]$	$D[3]$	$D[4]$	$D[5]$
15	2	∞	6

Крок 1. Вибираємо з усіх вершин множини $V \setminus U = 2,3,4,5$ вершину, для якої відстань від джерела найменша. Тобто вибираємо мінімальний елемент таблиці D:

$D[2]$	$D[3]$	$D[4]$	$D[5]$
15	2	∞	6

Це є значення 2, що відповідає вершині v_3, то додаємо цю вершину до множини U: $U = 1,3$

Для всіх вершин з множини $V \setminus U$ вибираємо «кращий» шлях – або поточний (від джерела) або ж той, який проходить через вершину v_3:

$D[2] = \min\ D[2], D[3] + C[3,2] = \min\ 15, 2+\infty = 15$

$D[4] = \min\ D[4], D[3] + C[3,4] = \min\ \infty, 2+\infty = \infty$

$D[5] = \min\ D[5], D[3] + C[3,5] = \min\ 6, 2+3 = 5$

$D[2]$	$D[3]$	$D[4]$	$D[5]$
15	2	∞	5

Крок 2. Вибираємо з усіх вершин множини $V \setminus U = 2,4,5$ вершину, для якої поточна відстань від джерела найменша. Тобто вибираємо мінімальний елемент таблиці D, не враховуючи третій:

$D[2]$	$D[3]$	$D[4]$	$D[5]$
15	2	∞	6

Це є значення 6, що відповідає вершині v_5, то додаємо цю вершину до множини U: $U = 1,3,5$

Для всіх вершин з множини $V \setminus U = 2,4$ вибираємо «кращий» шлях – або поточний (від джерела) або ж той, який проходить через вершину v_5:

$D[2] = \min\ D[2], D[5] + C[5,2] = \min\ 15, 5+5 = 10$

$D[4] = \min\ D[4], D[5] + C[5,4] = \min\ \infty, 6+7 = 13$

$D[2]$	$D[3]$	$D[4]$	$D[5]$
10	2	13	5

Крок 3. Вибираємо з усіх вершин множини $V \setminus U = 2,4$ вершину, для якої поточна відстань від джерела найменша. Тобто вибираємо мінімальний елемент таблиці D, не враховуючи третій і п'ятий:

$D[2]$	$D[3]$	$D[4]$	$D[5]$
10	2	13	5

Це є значення 10, що відповідає вершині v_2, то додаємо цю вершину до множини U: $U = 1,3,5,2$

Для всіх вершин з множини $V \setminus U = 4$ (тобто для єдиної вершини, що залишилася) вибираємо «кращий» шлях – або поточний (від джерела) або ж той, який проходить через вершину v_2:

$D[4] = \min\ D[4], D[2] + C[2,4] = \min\ 13, 10+11 = 13$

$D[2]$	$D[3]$	$D[4]$	$D[5]$
10	2	13	5

Крок 4. Вибираємо з усіх вершин множини $V \setminus U = 4$ вершину, для якої поточна відстань від джерела найменша. Оскільки ця множина містить тільки одну точку, то вибираємо її і приєднуємо до множини U: $U = 1,3,5,2,4$

$D[2]$	$D[3]$	$D[4]$	$D[5]$
10	2	13	5

Вершин у множині не залишилося, тому процес зупиняється.

Таким чином, процес знаходження найкоротших шляхів від вершини v_1 до всіх інших вершин ми могли б записати через наступну таблицю:

Кроки	w	**Множина** U	$D[2]$	$D[3]$	$D[4]$	$D[5]$
Ініціалізація	–	1	15	2	∞	6
1	3	1,3	15	2	∞	5
2	5	$U = 1,3,5$	10	2	13	5
3	2	$U = 1,3,5,2$	10	2	13	5
4	4	$U = 1,3,5,2,4$	10	2	13	5

Для того, щоб вибрати оптимальний шлях до кожної з вершин, потрібно виписати вершини, які їй передують при записі у множину U.

Найкоротший шлях з вершини v_1:

до вершини v_2: $v_1 \to v_3 \to v_5 \to v_2$, довжина шляху 10

до вершини v_3 – $v_1 \to v_3$, довжина шляху 2

до вершини v_4 – $v_1 \to v_3 \to v_5 \to v_2 \to v_4$, довжина шляху 13

до вершини v_5 – $v_1 \to v_3 \to v_5$, довжина шляху 5

Пошук найкоротших шляхів між парами вершин – алгоритм Флойда

Для того, щоб знайти найкоротші відстані між усіма парами вершин, можна n разів застосувати алгоритм Дейкстри, вибираючи в якості джерела послідовно кожну з вершин. Проте, є ще один алгоритм – алгоритм Флойда.

Алгоритм Флойда використовує матрицю A розміром $n \times n$, яка містить довжини найкоротших шляхів. Для початку матриця A співпадає з матрицею ваг, тільки якщо шляху між вершинами не існує – відповідний елемент ставиться ∞.

Над матрицею A проводиться n ітерацій. Після k-тої ітерації кожен елемент $A[i,j]$ містить найкоротші шляхи від вершини i до вершини j, які проходять через вершини з номерами, не більшими за k. На k-тій ітерації елементи матриці A обчислюються за формулою

$A_k[i,j] = \min\ A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]$.

$$A_0 = \begin{pmatrix} 0 & 15 & 2 & \infty & 6 \\ \infty & 0 & \infty & 11 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 7 & 0 \end{pmatrix}; \quad A_1 = \begin{pmatrix} 0 & 15 & 2 & \infty & 6 \\ \infty & 0 & \infty & 11 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 7 & 0 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} 0 & 15 & 2 & 26 & 6 \\ \infty & 0 & \infty & 11 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 7 & 0 \end{pmatrix}; \quad A_3 = \begin{pmatrix} 0 & 15 & 2 & 26 & 5 \\ \infty & 0 & \infty & 11 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 7 & 0 \end{pmatrix};$$

$$A_4 = \begin{pmatrix} 0 & 15 & 2 & 26 & 5 \\ \infty & 0 & \infty & 11 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 7 & 0 \end{pmatrix}; \quad A_5 = \begin{pmatrix} 0 & 10 & 2 & 12 & 5 \\ \infty & 0 & \infty & 11 & \infty \\ \infty & 8 & 0 & 10 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 7 & 0 \end{pmatrix}$$

Увага! Проаналізувати на наведеному вище прикладі застосування алгоритму Флойда, чому не змінився вигляд матриці при $k = 1$ та $k = 4$.

12.3 Завдання для самостійного виконання

1. Для свого варіанту графа (варіанти завдань наведено у додатку В) знайти найкоротші шляхи до всіх вершин від вершини *a* за допомогою алгоритму Дейкстри.

2. Для свого варіанту графа знайти найкоротші шляхи між усіма парами вершин за допомогою алгоритму Флойда.

Примітка: Відповіді до завдань 1 та 2 оформити, як у запропонованих іище прикладах.

3. Написати програмну реалізацію поданих алгоритмів, вибравши на свій розсід спосіб представлення графів в пам'яті комп'ютера.

12.4 Питання для самоконтролю

1. Для чого використовуються алгоритми пошуку шляхів на графах?
2. В чому особливість алгоритму Дейкстри.
3. Опишіть *i*-ту ітерацію алгоритму Дейкстри.
4. В чому особливість алгоритму Флойда.
5. Опишіть *i*-ту ітерацію алгоритму Флойда.
6. Наведіть порівняльну характеристику алгоритмів Дейкстри та Флойда.
7. Оцініть час, потрібний для виконання алгоритму Дейкстри.
8. Оцініть час, потрібний для виконання алгоритму Флойда.
9. Наведіть приклади доцільності використання алгоритму Дейкстри.
10. Наведіть приклади доцільності використання алгоритму Флойда.

Рекомендовані теми для самостійного вивчення

1. Обчислення часової складності алгоритмів.
2. Алгоритм впорядкування вставками зі сторожовим елементом.
3. Алгоритм впорядкування методом підрахунку.
4. Порозрядний алгоритм впорядкування.
5. Реалізація стеків за допомогою масивів.
6. Реалізація черг за допомогою масивів.
7. Робота з мультисписками.
8. Словники як структура даних.
9. Взаємозв'язок стеків і рекурсивних алгоритмів.
10. Алгоритм розмалювання графа.
11. Алгоритм Прима.
12. Алгоритм Крускала.

Рекомендована література

1. Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман.– М.: Изд. Дом «Вильямс», 2001. – 384с.
2. Нікольський Ю. В. Дискретна математика: Підручник / Ю. В Нікольський , В. В. Пасічник, Ю. М. Щербина. – Львів: «Магнолія Плюс», 2005. – 608 с.
3. Страуструп Б. Язык программирования C++. Специальное издание. Пер. с англ. / Б. Страуструп – М.: Издательство Бином, 2011 г. — 1136 с: ил.

Використані джерела

1. Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман.– М.: Изд. Дом «Вильямс», 2001. – 384с.
2. Браунси К. Основные концепции структур данных и реализация в C++ / К. Браунси. – М.: Изд. Дом «Вильямс», 2002. – 320с.
3. Вирт Н. Алгоритмы и структуры данных. / Н. Вирт. – М., ДМК_Пресс, 2011. – 272 с.
4. Кнут Д. Искусство программирования, т.3. Сортировка и поиск. / Д. Кнут – М.: Вильямс, 2011. – 824 с.
5. Кормен Т Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М.: Вильямс, 2011. – 1296 с.
6. Нікольський Ю. В. Дискретна математика: Підручник / Ю. В Нікольський , В. В. Пасічник, Ю. М. Щербина. – Львів: «Магнолія Плюс», 2005. – 608 с.
7. Страуструп Б. Язык программирования C++. Специальное издание. Пер. с англ. / Б. Страуструп – М.: Издательство Бином, 2011 г. — 1136 с: ил.

Додаток А

Зразок оформлення титульного аркуша лабораторної роботи

Державний вищий навчальний заклад
«Прикарпатський національний університет імені Василя Стефаника»
Кафедра інформатики

ЛАБОРАТОРНА РОБОТА №__

з дисципліни «Інформатика»

Тема: «_____»

Виконав(ла)
студент (ка)
групи _____
(назва групи)
_____
(Прізвище, ініціали)
Перевірив
викладач
Власій О. О.

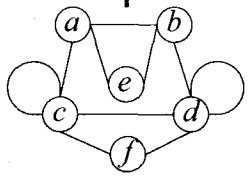
Івано-Франківськ – 2015

Додаток Б

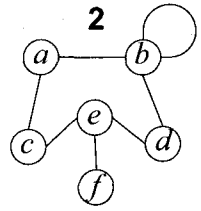
Варіанти завдань до лабораторних 10-11

Неорієнтовані графи

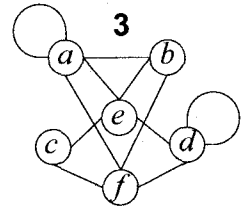
1



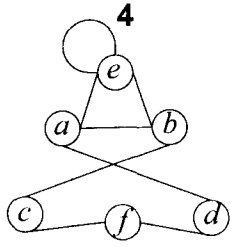
2



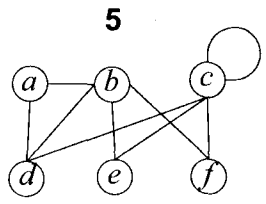
3



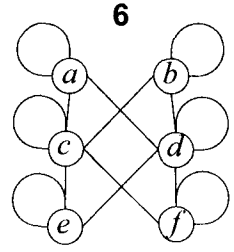
4



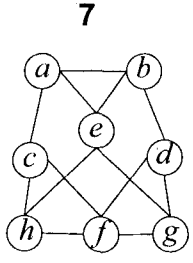
5



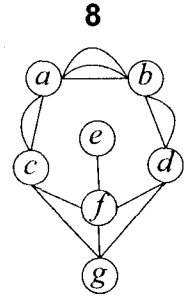
6



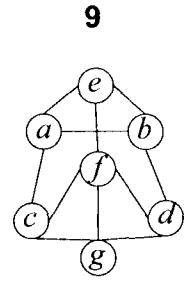
7



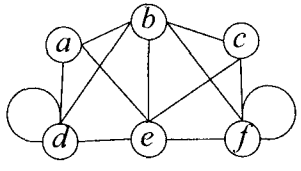
8



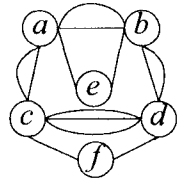
9



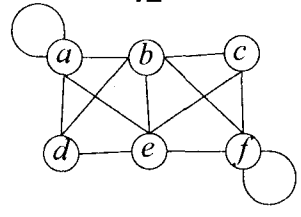
10



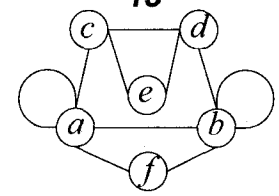
11



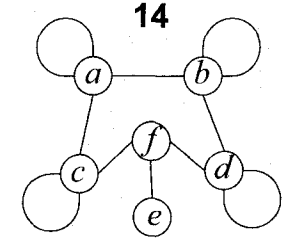
12



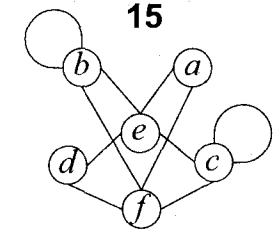
13



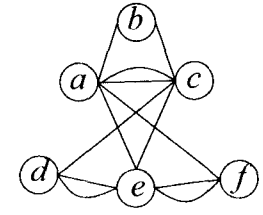
14



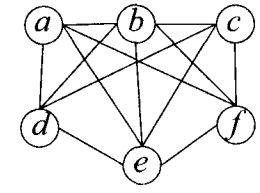
15



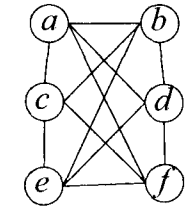
16



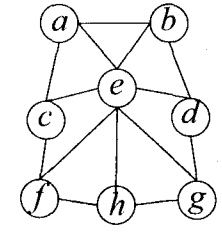
17



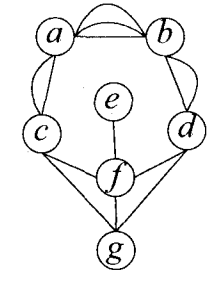
18



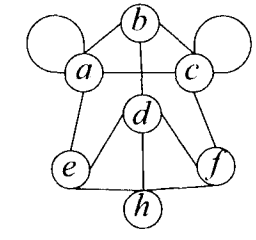
19



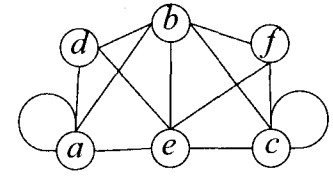
20



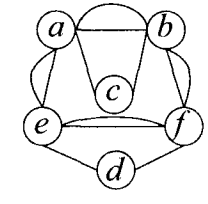
21



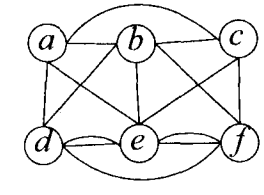
22



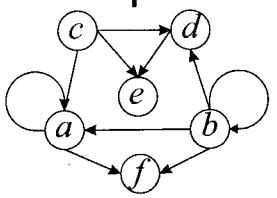
23



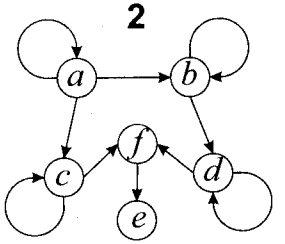
24



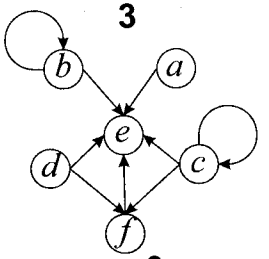
1



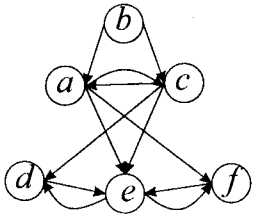
2



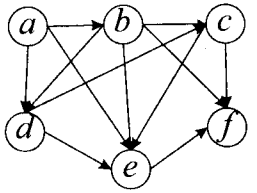
3



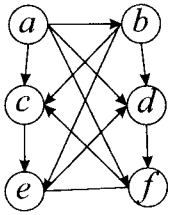
4



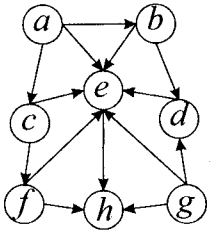
5



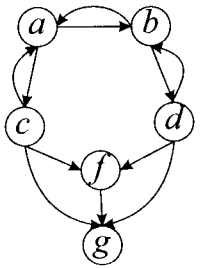
6



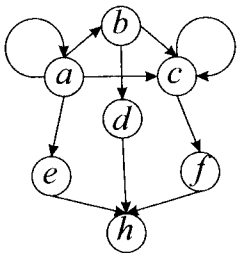
7



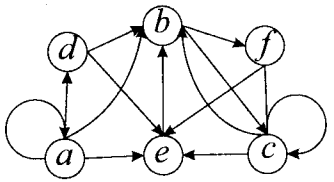
8



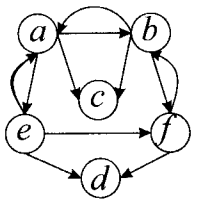
9



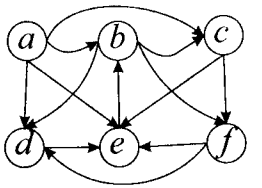
10



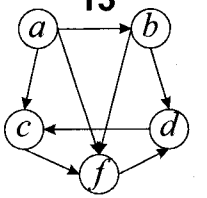
11



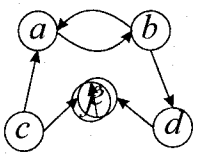
12



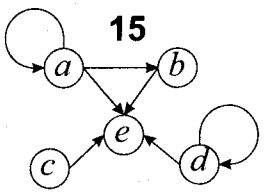
13



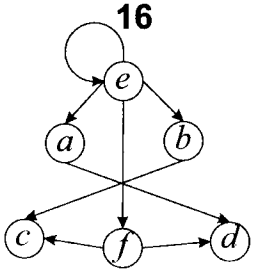
14



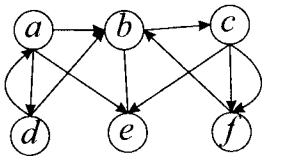
15



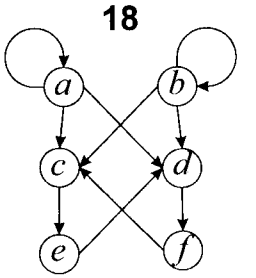
16



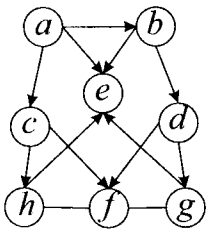
17



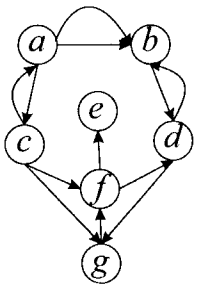
18



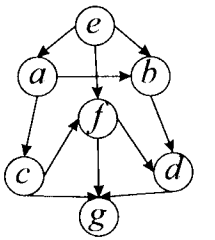
19



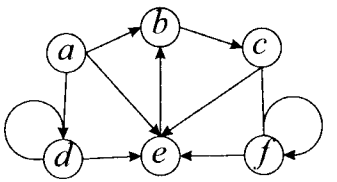
20



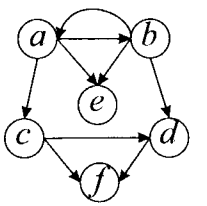
21



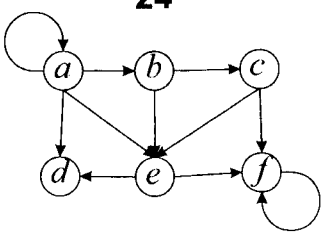
22



23



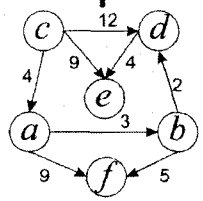
24



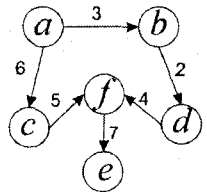
Додаток В

Варіанти завдань до лабораторної роботи №12

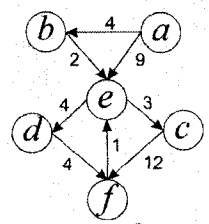
1



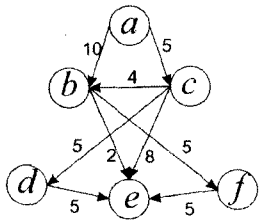
2



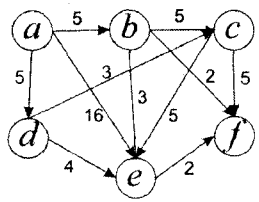
3



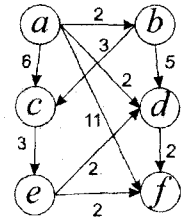
4



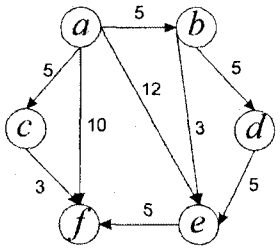
5



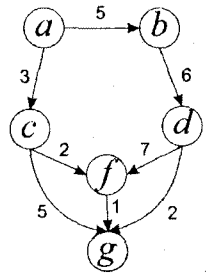
6



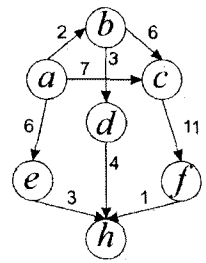
7



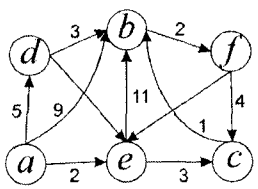
8



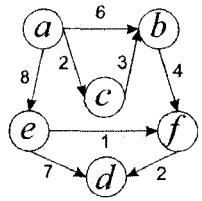
9



10



11



12