МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ОДЕСЬКА ПОЛІТЕХНІКА”
Інститут комп'ютерних систем
Кафедра інформаційних систем

Теорія алгоритмів

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт

(спеціальність 122 – «Комп'ютерні науки»)

Одеса НУОП 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ОДЕСЬКА ПОЛІТЕХНІКА”
Інститут комп'ютерних систем
Кафедра інформаційних систем

Теорія алгоритмів

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторних робіт
(спеціальність 122 – «Комп'ютерні науки»)

Затверджено
на засіданні кафедри
інформаційних систем
Протокол № 1 от 01.09.23р.

Одеса НУОП 2024

Методичні вказівки до лабораторних робіт з дисципліни «Теорія алгоритмів» для здобувачів всіх форм навчання інституту комп'ютерних систем (спеціальність 122 – «Комп'ютерні науки») / Укладач: О.О.Арсірій, С.Ю. Смик – Одеса: НУОП, 2024. – 89 с.

Укладач: Арсірій Олена Олександрівна,

к.т.н., доцент

Смик Сергій Юрійович,

к.т.н., доцент

Зміст

Вступ

Процес створення комп’ютерної програми для вирішення якого-небудь практичного завдання містить у собі формалізацію цього завдання, розробку обчислювального алгоритму її рішення, написання та налагодження програми, і, нарешті, вирішення поставленого завдання. Механізм реалізації алгоритмічного процесу зручно простежити на алгоритмічних моделях, що використовують кінцеві набори найпростіших об’єктів і елементарних дій.

Лабораторний практикум по дисципліні “Теорія алгоритмів” спрямований на вивчення основних типів алгоритмічних моделей, а також включає елементи розробки та програмування їх інтерпретації мовою високого рівня. В роботах розглянуті основні терміни та поняття теорії алгоритмів, засоби інтерпретації основних алгоритмічних моделей.

Лабораторна робота №1

Квадратичні алгоритми сортування

У роботі розглядаються алгоритми сортування вставки та вибором. Розглядається доцільність використання цих алгоритмів, їх переваги та недоліки.

Питання:

1. Загальні відомості про завдання сортування, поняття часової складності
2. Сортування вибором
3. Сортування вставками
4. Завдання
5. Вимоги до представлення звіту

1. Загальні відомості про завдання сортування, поняття часової складності алгоритмів сортування.

Нехай є послідовність a_0, a_1... a_n та функція порівняння, яка на будь-яких двох елементах послідовності набуває одного з трьох значень: менше, більше або дорівнює. Задача сортування полягає у перестановці членів послідовності у такий спосіб, щоб виконувалася умова: $a_i <= a_{i+1}$, для всіх і від 0 до n.

Можлива ситуація, коли елементи послідовності складаються з кількох полів:

Рис. 1

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

Якщо значення функції порівняння залежить тільки від поля x, то x називають ключем, по якому проводиться сортування. На практиці, у якості x часто виступає число, а поле y зберігає будь-які дані, що ніяк не впливають на роботу алгоритму.

Розглянемо параметри, за якими проводиться оцінка алгоритмів сортування

1. *Час* сортування – основний параметр, що характеризує швидкодію алгоритму.
2. *Пам'ять* – ряд алгоритмів вимагає виділення додаткової пам'яті під тимчасове зберігання даних.
3. *Стійкість* – стійке сортування не змінює взаємного розташування рівних елементів. Така властивість може бути дуже корисною, якщо вони складаються з декількох полів, а сортування відбувається по одному з них, наприклад, по x. (рис. 2 та 3)
4. *Природність поведінки* – ефективність методу при обробці вже відсортованих, або частково відсортованих даних. Алгоритм веде себе

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

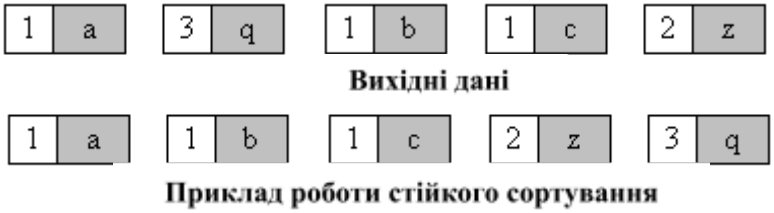


Рис. 2

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

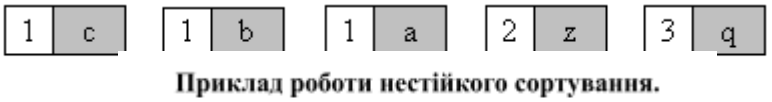


Рис. 3

Взаємне розташування рівних елементів із ключем 1 та додатковими полями "a", "b", "c" змінилося.

Часова складність алгоритму визначає час роботи, який використовує алгоритм, як функції від довжини рядка, що представляє вхідні дані. Часова складність алгоритму звичайно виражається із використанням нотації «*O*» *велике*, яка виключає коефіцієнти та члени меншого порядку. Якщо складність виражена у такий спосіб, говорять про *асимптотичний* опис часової складності, тобто у разі прагнення розміру входу до нескінченності. Наприклад, якщо час, який потрібний алгоритму для виконання роботи, для всіх входів довжини n не перевищує $5n^3 + 3n$ для деякого n (більшого деякого n_0), асимптотична часова складність дорівнює O (n^3).

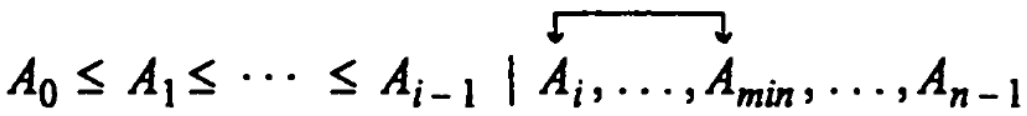
Часова складність найчастіше оцінюється шляхом підрахунку числа елементарних операцій, здійснюваних алгоритмом, де елементарна операція займає для виконання фіксований час. Тоді повний час виконання та кількість елементарних операцій, виконаних алгоритмом, відрізняються максимум на постійний множник.

Алгоритми сортування бульбашкою, вставками та вибором відносяться до сортувань із квадратичним часом складності $O(n^2)$

2. Сортування вибором

Ми починаємо *сортування вибором* з пошуку найменшого елемента у списку та обміну його з першим елементом (отже найменший елемент поміщається в остаточну позицію у відсортованому списку). Потім ми скануємо список, починаючи з другого елемента, у пошуках найменшого серед n-1 елементів, що залишилися, і обмінюємо знайдений найменший елемент з другим, тобто поміщаємо другий найменший елемент у остаточну позицію у відсортованому списку. У загальному випадку, при i-ому проході по списку ($0 \le i \le n$-2) алгоритм знаходить найменший елемент серед останніх n-i

елементів і обмінює його з A_i. (рис. 4). Після виконання n-1 проходів список виявляється відсортованим.



Елементи в остаточних позиціях **Останні $n - i$ елементів**

Рис.4 Алгоритм сортування вибором

Алгоритм *Selectionsort* (*A* [0..*n* - 1])
// Сортування масиву методом вибору
// Вхідні дані: Масив *A* [0.. *n* - 1] елементів, що впорядковуються
// Вихідні дані: Масив *A* [0.. *n* - 1], відсортований у неспадному
//порядку
for *i*← 0 to *n*-2 do
    min ← *i*
        for *j* ← *i*+1 to *n* -1 do
            if *A*[j] < *A*[*min*]
                min← *j*
    Обмін *A*[*i*] та *A* [*min*]

Як приклад на рис. 5 наведено сортування вибором наступного списку: 89, 45, 68, 90, 29, 34, 17.

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Рис. 5 Приклад сортування вибором

На рисунку наведено наступні позначення. Кожен рядок відповідає одній ітерації алгоритму, тобто сканування частини списку праворуч від вертикальної межі. Напівжирним шрифтом виділено найменші елементи, що виявляються під час сканування. Елементи праворуч від вертикальної межі знаходяться в остаточних позиціях і не скануються.

Результат чисельного моделювання наведеного псевдокоду.

Вихідні дані A=[89_0, 45_1, 68_2, 90_3, 29_4, 34_5, 17_6] n=7

1. i=0; *min*=0; *j*=0+1=1; A[1]<A[0] (45<89) min=1
    a. j=2 A[2]<A[1] (68<45) min=1
    b. j=3 A[3]<A[1] (90<45) min=1

c. j=4 A[4]<A[1] (29<45) min=4
d. j=5 A[5]<A[4] (34<29) min=4
e. j=6 A[6]<A[4] (17<29) min=6 (end for *j*)

b=A[0] A[0]=A[6] A[6]=b A[0]=17 A[6]=89

A=[17_0, 45_1, 68_2, 90_3, 29_4, 34_5, 89_6]

2. i=1; *min=1; j=1+1=2;* A[2]<A[1] (68<45) min=1
   1. j=3 A[3]<A[1] (90<45) min=1
   2. j=4 A[4]<A[1] (29<45) min=4
   3. j=5 A[5]<A[4] (34<29) min=4
   4. j=6 A[6]<A[4] (89<29) min=4 (end for *j*)

   b=A[1] A[1]=A[4] A[4]=b A[1]=29 A[4]=45

   A=[17_0, 29_1, 68_2, 90_3, 45_4, 34_5, 89_6]
3. i=2; *min=2; j=2+1=3;* A[3]<A[2] (68<90) min=2
   1. j=4 A[4]<A[1] (45<68) min=4
   2. j=5 A[5]<A[4] (34<45) min=5
   3. j=6 A[6]<A[4] (89<34) min=5 (end for *j*)

   b=A[2] A[2]=A[5] A[5]=b A[2]=34 A[5]=68

   A=[17_0, 29_1, 34_2, 90_3, 45_4, 68_5, 89_6]

4. i=3; *min=3; j=3+1=4;* A[4]<A[3] (45<90) min=4
   1. j=5 A[5]<A[4] (68<45) min=4
   2. j=6 A[6]<A[4] (89<45) min=4 (end for *j*)

   b=A[3] A[3]=A[4] A[3]=b A[3]=45 A[4]=90

   A=[17_0, 29_1, 34_2, 45_3, 90_4, 68_5, 89_6]

5. i=4; *min=4; j=4+1=5;* A[5]<A[4] (68<90) min=5
   1. j=6 A[6]<A[5] (89<68) min=5 (end for *j*)

   b=A[4] A[4]=A[5] A[4]=b A[4]=68 A[5]=90

   A=[17_0, 29_1, 34_2, 45_3, 68_4, 90_5, 89_6]

6. **i=5**; *min=5;* ***j=5+1=6;*** A[6]<A[5] (89<90) min=6 (end for *j*)

   b=A[5] A[5]=A[6] A[5]=b A[5]=89 A[6]=90

   A=[17_0, 29_1, 34_2, 45_3, 68_4, 89_5, 90_6]

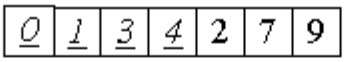
   (end for *i*)

3. Сортування вставками

Будемо розбирати алгоритм вставками, розглядаючи його дії на і-му кроці. Послідовність А на цей момент розділена на дві частини: готову A[0]...A[i] та невпорядковану A[i+1]...A[n-1]. На наступному, (i+1)-му кожному кроці алгоритму беремо a[i+1] і вставляємо на потрібне місце у готову частину масиву.

Пошук відповідного місця для чергового елемента вхідної послідовності здійснюється шляхом послідовних порівнянь з елементом, що стоїть перед ним.

Залежно від результату порівняння елемент або залишається на поточному місці (вставка завершена), або вони змінюються місцями і процес повторюється (Рис. 6).



Послідовність на поточний момент. Частина a[0]..a[2] вже впорядкована.

вставка завершена

Таким чином, у процесі вставки ми "просіюємо" елемент A до початку масиву, зупиняючись у разі, коли: знайдений елемент, менший за *A* або досягнуто початку послідовності.

Алгоритм сортування вставками складається з n-1-го проходів (n – розмірність масиву). Кожен із проходів включає чотири дії:

1. взяття чергового *i*-го невідсортованого елемента та збереження його у додатковій змінній;
2. пошук позиції *j* у відсортованій частині масиву, у якій присутність взятого елемента не порушить упорядкованості елементів;
3. зсув елементів масиву від *i*-1-го до *j*-1-го вправо, щоб звільнити знайдену позицію вставки;
4. вставка взятого елементу в знайдену j-ю позицію.

Псевдокод сортування методом вставок представлений нижче під назвою *InsertionSort*. На вхід подається масив A [1..n], що містить послідовність з n чисел, що сортуються (кількість елементів масиву A позначено в цьому коді як A.length). Вхідні числа сортуються без використання додаткової пам'яті: їх перестановка проводиться в межах масиву, і обсяг використовуваної при цьому додаткової пам'яті не перевищує постійну величину. По закінченні роботи алгоритму InsertionSort вхідний масив містить відсортовану послідовність:

```
InsertionSort
for j = 2 to A.length do
   key = A[j]
   i = j-1
   while (int i > 0 and A[i] > key) do
      A[i + 1] = A[i]
      i = i - 1
   end while
   A[i+1] = key
end
```

Перевірте наведений псевдокод на чисельному прикладі із сортування вибором

Вихідні дані $A=[89_1, 45_2, 68_3, 90_4, 29_5, 34_6, 17_7]$ n=7

A.length=7

1. j = 2; key = A[2] = 45; i = 2 – 1 = 1;
   i>0 (True) A[1] > key (89 > 45 – True) → A[2] = A[1] = 89;
   $A=[89_1, 89_2, 68_3, 90_4, 29_5, 34_6, 17_7]$
   i = 1 – 1 = 0;
   i>0 (False)
   A[1] = key = 45

$A=[45_1, 89_2, 68_3, 90_4, 29_5, 34_6, 17_7]$

j = 3; key = A[3] = 68; i = 3 – 1 =2;
i>0 (True) A[2] > key (89 > 68 – True) → A[3] = A[2] = 89;
$A=[45_1, 89_2, 89_3, 90_4, 29_5, 34_6, 17_7]$
i = 2 – 1 = 1;
i>0 (True) A[1] > key (45 > 68 – False);
A[2] = key = 68

$A=[45_1, 68_2, 89_3, 90_4, 29_5, 34_6, 17_7]$

j = 4; key = A[4] = 90; i = 4 – 1 =3;
i>0 (True) A[3] > key (89 > 90 – False)
A[4] = key = 90

$A=[45_1, 68_2, 89_3, 90_4, 29_5, 34_6, 17_7]$

j = 5; key = A[5] = 29; i = 5 – 1 =4;
i>0 (True) A[4] > key (90 > 29 –True) → A[5] = A[4] = 90;
$A=[45_1, 68_2, 89_3, 90_4, 90_5, 34_6, 17_7]$
i = 4 – 1 = 3;
i>0 (True) A[3] > key (89 > 29 –True) → A[4] = A[3] = 89;
$A=[45_1, 68_2, 89_3, 89_4, 90_5, 34_6, 17_7]$
i = 3 – 1 = 2;
i>0 (True) A[2] > key (68 > 29 –True) → A[3] = A[2] = 68;
$A=[45_1, 68_2, 68_3, 89_4, 90_5, 34_6, 17_7]$
i = 2 – 1 = 1;
i>0 (True) A[1] > key (45 > 29 –True) → A[2] = A[1] = 45;
$A=[45_1, 45_2, 68_3, 89_4, 90_5, 34_6, 17_7]$
i = 1 – 1 = 0;
i>0 (False)
A[1] = key = 29

$A=[29_1, 45_2, 68_3, 89_4, 90_5, 34_6, 17_7]$

j = 6; key = A[6] = 34; i = 6 – 1 =5;
i>0 (True) A[5] > key (90 > 34 –True) → A[6] = A[5] = 90;
$A=[29_1, 45_2, 68_3, 89_4, 90_5, 90_6, 17_7]$
i = 5 – 1 = 4;

i>0 (True) A[4] > key (89 > 34 –True) → A[5] = A[4] = 89;
A=[29_1, 45 $_2$, 68_3, 89_4, 89_5, 90_6, 17_7]
i = 4 – 1 = 3;
i>0 (True) A[3] > key (68 > 34 –True) → A[4] = A[3] = 68;
A=[29_1, 45 $_2$, 68_3, 68_4, 89_5, 90_6, 17_7]
i = 3 – 1 = 2;
i>0 (True) A[2] > key (45 > 34 –True) → A[3] = A[2] = 45;
A=[29_1, 45 $_2$, 45_3, 68_4, 89_5,90_6, 17_7]
i = 2 – 1 = 1;
i>0 (True) A[1] > key (29 > 34 –False) → A[3] = A[2] = 45;
A[2] = key = 34
A=[29_1, 34 $_2$, 45_3, 68_4, 89_5,90_6, 17_7]

j = 7; key = A[7] = 17; i = 7 – 1 =6;
i>0 (True) A[6] > key (90 > 17 –True) → A[7] = A[6] = 90;
A=[29_1, 34 $_2$, 45_3, 68_4, 89_5,90_6, 90_7]
i = 6 – 1 = 5;
i>0 (True) A[5] > key (89 > 17 –True) → A[6] = A[5] = 89;
A=[29_1, 34 $_2$, 45_3, 68_4, 89_5,89_6, 90_7]
i = 5 – 1 = 4;
i>0 (True) A[4] > key (68 > 17 –True) → A[5] = A[4] = 68;
A=[29_1, 34 $_2$, 45_3, 68_4,68_5,89_6, 90_7]
i = 4 – 1 = 3;
i>0 (True) A[3] > key (45 > 17 –True) → A[4] = A[3] = 45
A=[29_1, 34 $_2$, 45_3, 45_4,68_5,89_6, 90_7]
i = 3 – 1 = 2;
i>0 (True) A[2] > key (34 > 17 –True) → A[3] = A[2] = 34
A=[29_1, 34 $_2$, 34_3, 45_4,68_5,89_6, 90_7]
i = 2 – 1 = 1;
i>0 (True) A[1] > key (29 > 17 –True) → A[2] = A[1] = 29
A=[29_1, 29 $_2$, 34_3, 45_4,68_5,89_6, 90_7]
i = 1 – 1 = 0;
i>0 (False)
A[1] = key = 17
A=[17_1, 29 $_2$, 34_3, 45_4,68_5,89_6, 90_7]
End for j

4.Завдання

1. Варіанти завдань – співпадають із порядковим номером студента в групі.

Варіант	Послідовність
1	53, 12, 68, 63, 3, 47, 50, 61, 41
2	98, 40, 42, 88, 61, 87, 79, 97, 82
3	70, 80, 61, 92, 12, 23, 67, 65, 50
4	29, 1, 24, 69, 52, 97, 27, 10, 88
5	95, 43, 98, 41, 68, 67, 10, 7, 69

Варіант	Послідовність
6	61, 32, 27, 45, 75, 58, 5, 50, 99
7	78, 77, 4, 62, 8, 69, 46, 11, 49
8	28, 76, 27, 10, 5, 35, 95, 16, 33
9	31, 40, 22, 50, 53, 68, 97, 12, 15
10	18, 20, 2, 88, 61, 17, 79, 97, 82
11	41, 52, 47, 65, 95, 38, 15, 50, 99
12	11, 42, 67, 55, 65, 78, 25, 50, 69
13	53, 5, 44, 47, 35, 83, 82, 85, 28
14	77, 89, 74, 68, 70, 49, 5, 62, 51
15	19, 75, 43, 31, 5, 66, 62, 34, 76
16	50, 57, 78, 34, 41, 68, 47, 61, 38
17	86, 36, 14, 50, 64, 21, 2, 83, 82
18	80, 27, 37, 36, 91, 53, 86, 66, 98
19	21, 44, 22, 50, 63, 68, 97, 12, 15
20	7, 89, 4, 68, 70, 49, 10, 62, 51
21	54, 65, 7, 33, 86, 29, 11, 91, 12
22	50, 80, 19, 86, 35, 7, 60, 48, 51
23	10, 90, 95, 30, 45, 60, 57, 28, 5
24	90, 10, 15, 80, 100, 6, 57, 5, 29
25	53, 100, 44, 74, 53, 38, 82, 65, 28
26	47, 50, 61, 41,53, 12, 68, 63, 3
27	87, 79, 97, 82,98, 40, 42, 88, 61
28	12, 23, 67, 65, 50, 70, 80, 61, 92
29	69, 52, 97, 27, 10, 88,29, 1, 24
30	41, 68, 67, 10, 7, 69, 95, 43, 98
31	58, 5, 50, 99, 61, 32, 27, 45, 75
32	46, 11, 49, 78, 77, 4, 62, 8, 69
33	35, 95, 16, 33, 28, 76, 27, 10, 5
34	68, 97, 12, 15, 31, 40, 22, 50, 53
35	79, 97, 82, 18, 20, 2, 88, 61, 17
36	38, 15, 50, 99, 41, 52, 47, 65, 95

2. Відсортувати за допомогою сортування вибором
3. Відсортувати за допомогою сортування вставками
4. Порівняти між собою алгоритми сортування за кількістю операцій порівняння та присвоювання

5. Вимоги до представлення звіту

Звіт повинен містити:

1. Титульну частину (лист) з вказівкою на назву лабораторної роботи, ПІБ студента, групу, номер варіанту та послідовність, яку необхідно отсортувати.
2. Псевдокоди сортування вибором та вставками, результат їх чисельного моделювання за варіантами завдань
3. Результати порівнянь алгоритмів сортування вибором та вставками між собою за кількістю операцій порівняння та присвоювання
4. Висновки

Лабораторна робота №2

Логарифмічні алгоритми сортування

У роботі розглядаються алгоритми сортування злиттям та швидкого сортування. Розглядається доцільність використання цих алгоритмів, їх переваги та недоліки.

Питання:

6. Сортування злиттям
7. Швидке сортування
8. Завдання
9. Оформлення звіту

1. Сортування злиттям.

Сортування злиттям (*Merge sort*) – алгоритм сортування, що використовує O(n) додаткової пам'яті та працює за O($n$log(n)) часу.

Алгоритм використовує метод декомпозиції: задача розбивається на під задачі меншого розміру, які вирішуються окремо, після чого їх рішення зливаються (комбінуються) для одержання вирішення вихідного завдання. Конкретно процедуру сортування злиттям можна описати наступним чином:

1. Якщо в аналізованому масиві один елемент, то він вже відсортований і алгоритм завершує роботу.
2. Інакше масив розбивається на дві частини, які сортуються рекурсивно.
3. Після сортування двох частин масиву до них застосовується процедура злиття, яка за двома відсортованими частинами отримує вихідний відсортований масив.

Псевдокод алгоритму

Ітеративний алгоритм

```
function mergeSortIterative(a : int[n]):
    for i = 1 to n, i *= 2
        for j = 0 to n - i, j += 2 * i
            merge(a, j, j + i, min(j + 2 * i, n))
```

Рекурсивний алгоритм

```
function mergeSortRecursive(a : int[n]; left, right : int):
    if left + 1 >= right
        return
    mid = (left + right) / 2
    mergeSortRecursive(a, left, mid)
    mergeSortRecursive(a, mid, right)
    merge(a, left, mid, right)
```

```
function merge(a : int[n]; left, mid, right : int):
    it1 = 0
    it2 = 0
    result : int[right - left]

    while left + it1 < mid and mid + it2 < right
        if a[left + it1] < a[mid + it2]
            result[it1 + it2] = a[left + it1]
            it1 += 1
        else
```

```
        result[it1 + it2] = a[mid + it2]
        it2 += 1

    while left + it1 < mid
        result[it1 + it2] = a[left + it1]
        it1 += 1

    while mid + it2 < right
        result[it1 + it2] = a[mid + it2]
        it2 += 1

    for i = 0 to it1 + it2
        a[left + i] = result[i]
```

Приклад чисельного моделювання наведеного псевдокоду ітеративного алгоритму:

Вихідні дані $A=[89_0, 45_1, 68_2, 90_3, 29_4, 34_5, 17_6]$ n=7

i=1;

j=0

1. merge(a, j, j+i, min(j+2*i,n)) merge(a, 0, 1, min(2,7))

left=0; mid=1;right=2

it1 = 0 it2 = 0

result[2(left-right)];

1.while (left + it1) 0 < 1 (mid) and (mid + it2) 1 < 2 (right) – True

if a[0(left + it1)]89 < 45a[1(mid + it2)] – False

result[0(it1 + it2)] = a[1(mid + it2)] result[0]=45

it2 += 1 it2=1

while (left + it1) 0 < 1(mid) and (mid + it2) 2 < 2 (right) – False

while (left + it1) 0< 1mid – True

result[1(it1 + it2)] = a[0(left + it1)] result[1]=89

it1 += 1 it1=1

while (left + it1) 1< 1mid – False

while mid + it2 2 < 2 right – False

for i = 0 to it1 + it2 (2)

a[left + i] = result[i] a[0]= result[0]=45 a[1]= result[1]=89

A=$[45_0, 89_1, 68_2, 90_3, 29_4, 34_5, 17_6]$

i= 1 j=2 (j=j+2*i)

2. merge(a, j, j+i, min(j+2*i,n)) merge(a, 2, 3, min(4,7))

left=2; mid=3;right=4

it1 = 0 it2 = 0

result[2(left-right)];

1.while (left + it1) 2 < 3(mid) and (mid + it2) 3 < 4 (right) – True

if a[2(left + it1)]68 < 90a[3(mid + it2)] – True

result[0(it1 + it2)] = a[2(left + it1)] result[0]=68

it1 += 1 it1=1

while (left + it1) 3 < 3(mid) and (mid + it2) 3 < 4 (right) – False

while (left + it1)3 < 3 mid – False

while (mid + it2) 3 < 4 right
result[it1 + it2] = a[mid + it2] result[1] = a[3] result[1]=90
it2 += 1 it2=1

for i = 0 to it1 + it2 (2)
a[left + i] = result[i] a[2]= result[0]=68 a[3]= result[1]=90
A=*[45$_0$, 89$_1$, 68$_2$, 90$_3$, 29$_4$, 34$_5$, 17$_6$]*

i= 1j=4 (j=j+2*i)
3. merge(a, j, j+i, min(j+2*i,n)) merge(a, 4, 5, min(6,7))
left=4; mid=5;right=6
it1 = 0 it2 = 0
result[2(left-right)];
while (left + it1) 4 < 5(mid) and (mid + it2) 5 < 6 (right) – True
if a[4(left + it1)]29 < 34a[5(mid + it2)] – True
result[0(it1 + it2)] = a[4(left + it1)] result[0]=29
it1 += 1 it1=1
while (left + it1) 5 < 5(mid) and (mid + it2) 5 < 4 (right) – False
while (left + it1)5 < 5 mid – False
while (mid + it2) 5 < 6 right - True
result[it1 + it2] = a[mid + it2] result[1] = a[5] result[1]=34
it2 += 1 it2=1
for i = 0 to it1 + it2 (2)
a[left + i] = result[i] a[4]= result[0]=29 a[5]= result[1]=34
A=*[45$_0$, 89$_1$, 68$_2$, 90$_3$, 29$_4$, 34$_5$, 17$_6$]*

i=1; j=6; (j=j+2*i)
j = 6 = n(7) – i(1) кінець циклу по j

1.i=2 j=0
4.merge(a, j, j+i, min(j+2*i,n)) merge(a, 0, 2, min(4,7))
left=0; mid=2;right=4
it1=0; it2=0;
result[4-0] – result[0]; result[1]; result[2]; result[3];
while left + it1 0< 2mid and mid + it2 2 < 4right – True
if a[0(left + it1)] 45 < 68 a[2(mid + it2)] – True
result[0(it1 + it2)] = a[0(left + it1)] result[0]=45
it1 += 1 it1=1

while left + it1 1< 2mid and mid + it2 2 < 4right – True
if a[1(left + it1)] 89 < 68 a[2(mid + it2)] – False
else
result[1(it1 + it2)] = a[2(mid + it2)] result[1]=68
it2 += 1 it2=1

while left + it1 1< 2mid and mid + it2 3 < 4right – True

if a[1(left + it1)] 89 < 90 a[3(mid + it2)] – True
result[2(it1 + it2)] = a[1(left + it1)] result[2]=89
it1 += 1 it1=2

while left + it1 2< 2mid and mid + it2 3 < 4right – False

while left + it12 < 2 mid False

while mid + it2 3 < 4 right
result[3 (it1 + it2)] = a[(3 (mid + it2)] result[3]=90
it2 += 1 it2=2

for i = 0 to it1 + it2 (4)
a[left + i] = result[i] a[0]= result[0]=45 a[1]= result[1]=68 a[2]= result[2]=89 a[3]= result[3]=90
A=[45_0, 68_1, 89_2, 90_3, 29_4, 34_5, 17_6]

i=2; j=4 (j += 2 * i)
5.merge(a, j, j+i, min(j+2*i,n)) merge(a, 4, 6, min(8,7))
left=4; mid=6;right=7
it1=0; it2=0;
result[7-4] – result[0]; result[1]; result[2];
while left + it1 4< 6mid and mid + it2 6 < 7right – True
if a[4(left + it1)] 29 < 17 a[6(mid + it2)] – False
else
result[0(it1 + it2)] = a[6(mid + it2)] result[0]=17
it2 += 1 it2=1
while left + it1 4< 6mid and mid + it2 7 < 7right – False

while left + it14 < 6 mid
result[1(it1 + it2)] = a[4(left + it1)] result[1]=29
it1 += 1 it1=1
while left + it15 < 6 mid
result[2(it1 + it2)] = a[5(left + it1)] result[2]=34
it1 += 1 it1=2

while mid + it2 7< 7right - False

for i = 0 to it1 + it2 (3)
a[left + i] = result[i] a[4]= result[0]=17 a[5]= result[1]=29 a[6]= result[2]=34
A=[45_0, 68_1, 89_2, 90_3, 17_4,29_5, 34_6]
i=2; j=6 (j += 2 * i)
j = 6 = n(7) – i(1) кінець циклу по j

i=4 j=0
6.merge(a, j, j+i, min(j+2*i,n)) merge(a, 0, 4, min(8,7))
left=0; mid=4;right=7

it1=0; it2=0;
result[7-0] – result[0]; result[1]; result[2]; result[3]; result[4]; result[5];
result[6];

while left + it1 0 < 4mid and mid + it2 4<7 right - True
if a[0(left + it1)] 45<17 a[4(mid + it2)] - False
else
result[0(it1 + it2)] = a[4(mid + it2)] result[0]=17
it2 += 1 it2=1

while left + it1 0 < 4mid and mid + it2 5<7 right True
if a[0(left + it1)] 45<29 a[5(mid + it2)] - False
else
result[1(it1 + it2)] = a[5(mid + it2)] result[1]=29
it2 += 1 it2=2
while left + it1 0 < 4mid and mid + it2 6<7 right True
if a[0(left + it1)] 45<34 a[5(mid + it2)] - False
else
result[2(it1 + it2)] = a[6(mid + it2)] result[2]=34
it2 += 1 it2=3
while left + it1 0 < 4mid and mid + it2 7<7 right False

while left + it1 0 < 4mid True
result[3(it1 + it2)] = a[0(left + it1)] result[3]=45
it1 += 1 it1=1
while left + it1 1 < 4mid True
result[4(it1 + it2)] = a[1(left + it1)] result[4]=68
it1 += 1 it1=2
while left + it1 2 < 4mid True
result[5(it1 + it2)] = a[2(left + it1)] result[5]=89
it1 += 1 it1=3
while left + it1 3 < 4mid True
result[6(it1 + it2)] = a[3(left + it1)] result[6]=90
it1 += 1 it1=4
while left + it1 4 < 4mid -False

while mid + it27 < 7right False

for i = 0 to it1 + it2 (7)
a[left + i] = result[i] a[0]= result[0]=17 a[1]= result[1]=29 a[2]= result[2]=34
a[3]= result[3]=45 a[4]= result[4]=68 a[5]= result[5]=89 a[6]= result[6]=90
A=[17_0,29_1, 34_2 45_3, 68_4, 89_5, 90_6]
j = n(7) – i(1) кінець циклу по j
i=8 кінець циклу по i

2. Швидке сортування

Швидке сортування (*quick sort*) чи сортування Хоара – один із найбільш широко використовуваних алгоритмів сортування. Середній час роботи $O(n\log(n))$

В основі швидкого сортування також лежить метод декомпозиції.

1. Масив a[l…r] типу T розбивається на два (можливо порожніх) під масиви a[l…q] та a[q+1…r], таких, що кожний елемент a[l…q]менше чи дорівнює a[q], який у свою чергу, менше будь-якого елемента підмасиву a[q+1…r]. Індекс обчислюється під час процедури розбиття.
2. Підмасиви a[l…q] та a[q+1…r] сортуються за допомогою рекурсивного виклику процедури швидкого сортування.
3. Оскільки підмасиви *сортуються на місці*, для їхнього об'єднання не потрібні жодні дії: весь масив a[l…r] виявляється відсортованим.

Псевдокод алгоритму

```
void quicksort(a: T[n], int l, int r)
   if l < r
     int q = partition(a, l, r)
     quicksort(a, l, q)
     quicksort(a, q + 1, r)
```

Для сортування всього масиву необхідно виконати процедуру
quicksort(a,0,length[a]−1)

Основний крок алгоритму сортування – процедура розбиття (partition), яка переставляє елементи масиву a[l…r] типу T належним чином.

Розбиття здійснюється з використанням наступної стратегії. Перш за все довільно або за правилом вибирається опорний (поділяючій) елемент a[(l+r)/2] (pivot - v). Далі починається перегляд з лівого кінця масиву, який триває до тих пір, поки не буде знайдений елемент, що перевищує значення поділяючого елементу, потім виконується перегляд, починаючи з правого кінця масиву, який триває до тих пір, поки не відшукується елемент, який за значенням менше поділяючого. Обидва елементи, на яких перегляд було перервано, очевидно, знаходяться не на своїх місцях у розділеному масиві, тому вони змінюються місцями. Так продовжуємо далі, поки не переконаємося в тому, що в лівої частині масиву не залишилося жодного елемента, який був би більшим за значенням поділяючого, і жодного елемента в правій частині масиву, який був би меншими за значенням поділяючого елемента.

Змінна v зберігає значення опорного елемента a[(l+r)/2], a i та j відповідно є індексами елементів лівої та правої частини масиву, які порівнюються зі значенням змінної v.

Цикл поділу збільшує значення i та зменшує значення j на 1, причому умова, що жоден елемент ліворуч від i не більше v і жоден елемент праворуч від j не менше v, не порушується. Щойно значення індексів перетинаються, процедура розбиття завершується.

Псевдокод процедури partition

```
int partition(a: T[n], int l, int r)
```

```
v = a[(l + r) / 2]
int i = l
int j = r
while (i <= j)
  while (a[i] < v)
    i++
  while (a[j] > v)
    j--
  if (i >= j)
    break
  swap(a[i++], a[j--])
return j
```

Результат чисельного моделювання наведеного псевдокоду.

```
void quicksort(a: T[n], int l, int r)
  if l < r
    int q = partition(a, l, r)
    quicksort(a, l, q)
    quicksort(a, q + 1, r)
```

```
int partition(a: T[n], int l, int r)
  v = a[(l + r) / 2]
  int i = l
  int j = r
  while (i <= j)
    while (a[i] < v)
      i=i+1
    while (a[j] > v)
      j=j+1
    if (i >= j)
      break
    swap(a[i++], a[j--])
  return j
```

Вихідні дані A=[89_0, 45_1, 68_2, 90_3, 29_4, 34_5, 17_6] n=7

1. quicksort([89_0, 45_1, 68_2, 90_3, 29_4, 34_5, 17_6],0,6) //
   quicksort(a,0,length[a]−1) – тобто l=0, r=6

if 0 < 6 - True
int q = partition([89_0, 45_1, 68_2, 90_3, 29_4, 34_5, 17_6], 0, 6)
int v = a[(l+r)/2];(v =90_3)
int i = 0;
int j = 6;
while(i<=j)(0 < 6) - True
while(a[i]<v)(89_0 <90_v) - True
i++ (i = 1)
while(a[i]<v)(45_1<90_v) – True
i++ (i = 2)
while(a[i]<v)(68_2<90_v) – True
i++ (i = 3)
while(a[i]<v)(90_3<90_v) – False

while(a[j]>v)(17_6 >90_v) – False
if(3>=6) - False
swap(a[i], a[j])(swap(90,17))

89_0, 45_1, 68_2, 17_3, 29_4, 34_5, 90_6

while(i<=j)(3 < =6) - True
while(a[i]<v)(17_3 <90_v) - True
i++ (i = 4)
while(a[i]<v)(29_4 <90_v) – True
i++ (i = 5)
while(a[i]<v)(34_5 <90_v) - True
i++ (i = 6)
while(a[i]<v)(90_6 <90_v) - False
while(a[j]>v)(90_6 >90_v) - False
if(6>=6) – True
break
return j(6) q=6

2.quicksort([89_0, 45_1, 68_2, 17_3, 29_4, 34_5, 90_6], 0, 6) // quicksort(a, l, q)
Тобто l=0, r=6
if 0 < 6 - True
int q = partition([89_0, 45_1, 68_2, 17_3, 29_4, 34_5, 90_6], 0, 6)
int v = a[(l+r)/2];(v =17)
int i = 0;
int j = 6;
while(i<=j)(0 < 6) - True
while(a[i]<v)(89_0 < 17_v) - False
while(a[j]>v)(90_6 >17_v) - True
j-- (j = 5)
while(a[j]>v)(34_5 >17_v) - True
j-- (j = 4)
while(a[j]>v)(29_4 >17_v) - True
j-- (j = 3)
while(a[j]>v)(17_3 >17_v) - False
if(0>=3) - False
swap(a[i], a[j])(swap(89, 17))
17_0, 45_1, 68_2, 89_3, 29_4, 34_5, 90_6

while(i<=j)(0<=3) - True
while(a[i]<v)(17_0 < 17_v) - False
while(a[j]>v)(89_3 >17_v) - True
j-- (j = 2)
while(a[j]>v)(68_2 >17_v) - True
j-- (j = 1)
while(a[j]>v)(45_1 >17_v) - True
j-- (j = 0)
while(a[j]>v)(17_0 >17_v) - False
if(0>=0) - True
break
return j(0) q=0
quicksort([17_0, 45_1, 68_2, 89_3, 29_4, 34_5, 90_6], 0, 0) // quicksort(a, l, q)

Тобто l=0, r=0

if 0 < 0 – False

4. quicksort([17_0, 45_1, 68_2, 89_3, 29_4, 34_5, 90_6], 0 + 1, 6) // quicksort(a, q+1, r)

if 1 < 6 - True

int q = partition([17_0, 45_1, 68_2, 89_3, 29_4, 34_5, 90_6], 1, 6)

int v = a[(l+r)/2];(v =89_3)

int i = 1;

int j = 6;

while (i <= j) (1<=6) True

while (a[i] < v) (45_1 <89_v) – True

i++ (i = 2)

while (a[i] < v) (68_2 <89_v) – True

i++ (i = 3)

while (a[i] < v) (89_3<89_v) – False

while (a[j] > v) (90_6 >89_v) - True

j--(j=5)

while (a[j] > v) (34_5 >89_v) - False

if (i >= j) (3>=5) False

swap(a[i], a[j]) (swap(89,34)

[17_0, 45_1, 68_2, 34_3, 29_4, 89_5, 90_6]

while (i <= j) (3<=5) True

while (a[i] < v) (34_3 <89_v) – True

i++ (i = 4)

while (a[i] < v) (29_4 <89_v) – True

i++ (i = 5)

while (a[i] < v) (89_5 <89_v) – False

while (a[j] > v) (89_5 >89_v) - False

if (i >= j)

break

return j (q=5)

quicksort(a, q+1, r) quicksort(a, 6, 6)

if 6 < 6 – False

quicksort([17_0, 45_1, 68_2, 34_3, 29_4, 89_5, 90_6] l, q) // quicksort(a,0,6) // quicksort(a, 1, q)

if 0 < 6 - True

int q = partition([17_0, 45_1, 68_2, 34_3, 29_4, 89_5, 90_6], 1, 5)

int v = a[(l+r)/2];(v =34_3)

int i =0;

int j = 6;

while (i <= j) (0<=6) True

while (a[i] < v) (17_1 <34_v) – True

i++ (i = 1)

while (a[i] < v) (45_1 <34_v) – False

while (a[j] > v) (90_6 >34_v) - True

j--(j=5)
while (a[j] > v) (89_5 >34_v) - True
j--(j=4)
while (a[j] > v) (29_4 >34_v) - False
if (i >= j) (1>=4) False
swap(a[i], a[j]) (swap(45,29)
([17_0, 29_1, 68_2, 34_3, 45_4, 89_5, 90_6]
while (i <= j) (1<=4) True
while (a[i] < v) (29_1 <34_v) – True
i++ (i = 2)
while (a[i] < v) (68_2 <34_v) – False
while (a[j] > v) (45_4 >34_v) - True
j--(j=3)
while (a[j] > v) (34_4 >34_v) - False
if (i >= j) (2>=3) False
swap(a[i], a[j]) (swap(34,68)
([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6]
while (i <= j) (2<=3) True
while (a[i] < v) (34_2 <34_v) – False
while (a[j] > v) (68_4 >34_v) - True
j--(j=2)
while (a[j] > v) (34_4 >34_v) - False
if (i >= j) (2>=2) False
return j (q=2)
quicksort([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6] 0, 2) // quicksort(a, l, q)
if 0 < 2 - True
int q = partition([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6], 1, 2)
int v = a[(l+r)/2];(v =29_1)
int i =0;
int j = 2;
while (i <= j) (0<=2) True
while (a[i] < v) (17_0 <29_v) – True
i++ (i = 1)
while (a[i] < v) (29_1 <29_v) – False
while (a[j] > v) (34_2 >29_v) - True
j--(j=1)
while (a[j] > v) (29_1 >29_v) - False
if (i >= j) (1>=1) False
return j (q=1)

quicksort([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6] 0, 1) // quicksort(a, 1, q)
if 0 < 1 – True
int q = partition([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6], 0, 1) // quicksort(a, l, q)
int v = a[(l+r)/2];(v =17_o)
int i =0;
int j = 1;
while (i <= j) (0<=1) True

while (a[i] < v) (17_0 <17_v) – False
while (a[j] > v) (29_1 >17_v) - True
j--(j=0)
while (a[j] > v) (17_0 >17_v) - False
if (i >= j) (0>=0) False
return j (q=0)
quicksort([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6] 0, 0) // quicksort(a, 1, q)
if 0 < 0 – False

quicksort([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6] 2, 5) // quicksort(a, 1, 6) // quicksort(a, q+1, r)
if 1 < 6 - True
int q = partition([17_0, 29_1, 34_2, 68_3, 45_4, 89_5, 90_6], 2, 5)
int v = a[(l+r)/2];(v =68_3)
int i =1;
int j = 6;
while (i <= j) (1<=6) True
while (a[i] < v) (17_0 <68_v) – True
i++ (i = 1)
while (a[i] < v) (29_1 <68_v) – True
i++ (i = 2)
while (a[i] < v) (34_2 <68_v) – True
i++ (i = 3)
while (a[i] < v) (68_3 <68_v) – False
while (a[j] > v) (90_6 >68_v) - True
j--(j=5)
while (a[j] > v) (89_5 >68_v) - True
j--(j=4)
while (a[j] > v) (45_5 >68_v) - False
if (i >= j) (3>=4) False
swap(a[i], a[j]) (swap(68,45)
[17_0, 29_1, 34_2, 45_3, 68_4, 89_5, 90_6]
while (i <= j) (3<=4) True
while (a[i] < v) (45_3 <68_v) – True
i++ (i = 4)
while (a[j] > v) (68_4 >68_v) - False
if (i >= j) (4>=4) False
return j (q=4)

quicksort([17_0, 29_1, 34_2, 45_3, 68_4, 89_5, 90_6] 5, 6) // quicksort(a, 5, 6) // quicksort(a, q+1, r)
if 5 < 6 – True
int q = partition([17_0, 29_1, 34_2, 45_3, 68_4, 89_5, 90_6], 5, 6)
int v = a[(l+r)/2];(v =89_3)
int i =5;
int j = 6;
while (i <= j) (5<=6) True
while (a[i] < v) (89_0 <89_v) – False
while (a[j] > v) (90_6 >89_v) - True

j--(j=5)

while (a[j] > v) (89_5 >89_v) - False

if (i >= j) (5>=5) False

return j (q=5)

quicksort([17_0, 29_1, 34_2, 45_3, 68_4, 89_5, 90_6] 6, 6) // quicksort(a, 6, 6) // quicksort(a, q+1, r)

if 6 < 6 – False

End

3 Завдання:

1. Відсортувати за допомогою сортування злиттям
2. Відсортувати за допомогою швидкого сортування
3. Порівняти між собою алгоритми сортування, підрахувавши кількість операцій під час виконання моделювання

Варіанти завдань:

Варіант	Послідовність
1	53, 12, 68, 63, 3, 47, 50, 61, 41
2	98, 40, 42, 88, 61, 87, 79, 97, 82
3	70, 80, 61, 92, 12, 23, 67, 65, 50
4	29, 1, 24, 69, 52, 97, 27, 10, 88
5	95, 43, 98, 41, 68, 67, 10, 7, 69
6	61, 32, 27, 45, 75, 58, 5, 50, 99
7	78, 77, 4, 62, 8, 69, 46, 11, 49
8	28, 76, 27, 10, 5, 35, 95, 16, 33
9	31, 40, 22, 50, 53, 68, 97, 12, 15
10	18, 20, 2, 88, 61, 17, 79, 97, 82
11	41, 52, 47, 65, 95, 38, 15, 50, 99
12	11, 42, 67, 55, 65, 78, 25, 50, 69
13	53, 5, 44, 47, 35, 83, 82, 85, 28
14	77, 89, 74, 68, 70, 49, 5, 62, 51
15	19, 75, 43, 31, 5, 66, 62, 34, 76
16	50, 57, 78, 34, 41, 68, 47, 61, 38
17	86, 36, 14, 50, 64, 21, 2, 83, 82
18	80, 27, 37, 36, 91, 53, 86, 66, 98
19	21, 44, 22, 50, 63, 68, 97, 12, 15
20	7, 89, 4, 68, 70, 49, 10, 62, 51
21	54, 65, 7, 33, 86, 29, 11, 91, 12
22	50, 80, 19, 86, 35, 7, 60, 48, 51
23	10, 90, 95, 30, 45, 60, 57, 28, 5
24	90, 10, 15, 80, 100, 6, 57, 5, 29
25	53, 100, 44, 74, 53, 38, 82, 65, 28
26	47, 50, 61, 41,53, 12, 68, 63, 3
27	87, 79, 97, 82,98, 40, 42, 88, 61
28	12, 23, 67, 65, 50, 70, 80, 61, 92
29	69, 52, 97, 27, 10, 88,29, 1, 24
30	41, 68, 67, 10, 7, 69, 95, 43, 98
31	58, 5, 50, 99, 61, 32, 27, 45, 75
32	46, 11, 49, 78, 77, 4, 62, 8, 69

Варіант	Послідовність
33	35, 95, 16, 33, 28, 76, 27, 10, 5
34	68, 97, 12, 15, 31, 40, 22, 50, 53
35	79, 97, 82, 18, 20, 2, 88, 61, 17
36	38, 15, 50, 99, 41, 52, 47, 65, 95

4.Вимоги до представлення звіту

Звіт повинен містити:

1. Титульну частину (лист) з вказівкою на назву лабораторної роботи, ПІБ студента, групу, номер варіанту та послідовність, яку необхідно отсортувати.
2. Псевдокоди сортування злиттям та швидкого сортування, результат їх чисельного моделювання за варіантами завдань
3. Результати порівнянь алгоритмів сортування злиттям та швидкого сортування між собою за кількістю операцій порівняння та присвоювання
4. Висновки

Лабораторна робота №3

Алгоритм пірамідального сортування

У роботі розглядаються алгоритм пірамідального сортування (*Heapsort*, «Сортування купою») як удосконалене сортування бульбашкою, в якій елемент спливає (min-heap) / тоне (max-heap) багатьма шляхами.

Питання:

1. Теоретичні відомості про купи
2. Представлення масиву у вигляді максимальної купи
3. Сортування масиву за допомогою купи
4. Приклад моделювання
5. Завдання
6. Оформлення звіту

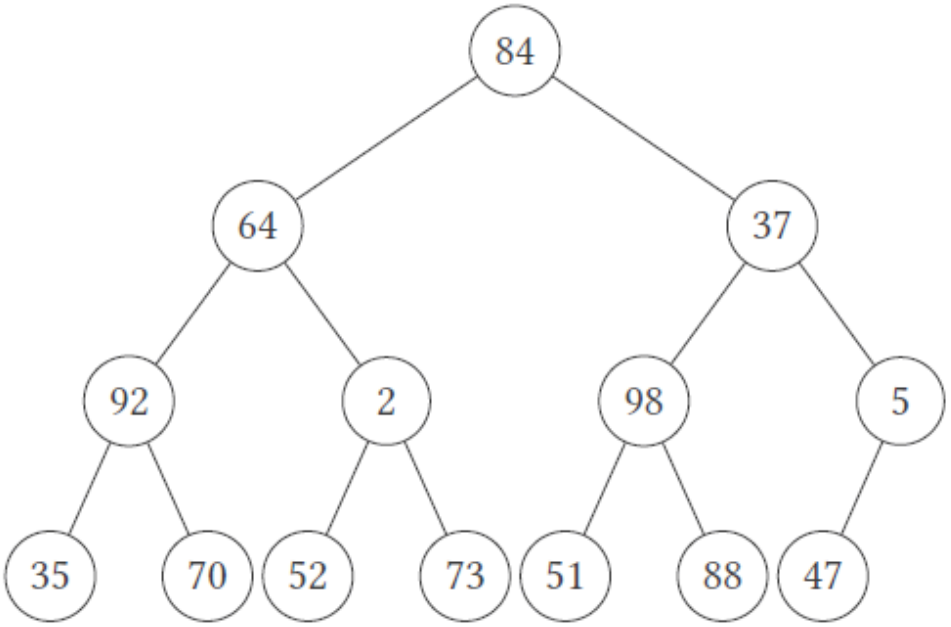
1. Теоретичні відомості про купи

Представимо сортувальну процедуру, схожу на сортування методом вибору, в якій знаходиться найбільший елемент масиву, який поміщаємо на кінець. Це як би сортування методом вибору навпаки: замість пошуків мінімуму, який має зайняти першу позицію, ми шукаємо максимум, який має зайняти останню позицію, |A| – 1. Замість знаходження другого найменшого елемента для другої позиції ми шукаємо другий найбільший елемент для позиції |A| – 2. Припустимо, що ми впорядковуємо елементи в масиві A таким чином, що у нас A[0] ≥ A[i] для всіх i < |A|. Тоді A[0] – це максимум, який ми повинні помістити в кінець A, помінявши місцями A[0] та A[n – 1]. Тепер A[n – 1] містить найбільший з наших елементів, а A[0] попереднє значення A[n – 1]. Далі продовжуємо сортувати елементи від A[0] до A[n – 2] так, щоб в результаті A[0] ≥ A[i] для всіх i < |A| – 1. Повторимо процедуру обміну A[0], який є найбільшим з елементів A[0], A[1], …, A[n – 2], на A[n – 2]. Тепер A[n – 2] міститиме другий найбільший елемент. Потім ми знову продовжуємо перетворювати A[0], A[1], …, A[n – 3] так, що у нас A[0] ≥ A[i] для всіх i < |A| – 2, і міняємо місцями A[0] та A[n – 3] і т.д. У результаті всі елементи масиву A будуть розміщені у правильному порядку.

Спробуємо представити масив A у вигляді бінарного дерева, де кожній батьківській вершині A[i] відповідають дочірні вершини A[2i + 1] та A[2*i* + 2] (Рис.1). Якщо для кожного *i* ми маємо $A[i] \geq A[2i + 1]$ та $A[i] \geq A[2i + 2]$, тоді кожна батьківська вершина дерева більша або дорівнює своїм дочірнім вершинам. А значення кореневої вершини дерева більше за всі інші елементи дерева тоді A[0] ≥ A[*i*] для всіх *i* < |A|. Іноді таке бінарне дерево називають «максимальною купою».

Для довідки: купа – це спеціалізована структура даних типу дерево, яка задовольняє *властивості купи:* якщо *B* є вузлом-нащадком вузла *A*, то ключ(*A*) ≥ ключ(*B*). З цього випливає, що елемент із найбільшим ключем завжди є кореневим вузлом купи, тому іноді такі купи

називають *max-купами* (в якості альтернативи, якщо порівняння перевернути, то найменший елемент завжди буде кореневим вузлом, такі купи називають *min-купами*). Не існує жодних обмежень щодо того, скільки вузлів-нащадків має кожен вузол купи, хоча на практиці їх кількість *звичайно не більше двох*. Купа є максимально ефективною реалізацією абстрактного типу даних, що називається чергою із пріоритетом. Купи мають вирішальне значення в деяких ефективних алгоритмах на графах, таких як алгоритм Дейкстри на d-купах і сортування методом піраміди. На рис.1 показано представлення масиву A з 14-ти елементів у вигляді бінарного дерева



84	64	37	92	2	98	5	35	70	52	73	51	88	47
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Рис. 1 Представлення елементів масиву у вигляді вершин та ребер дерева.

Далі необхідно вирішити питання, яким чином представити дерево, яке відповідає елементам масиву A у вигляді максимальної купи?

2. Представлення масиву у вигляді максимальної купи.

Якщо *останній* рівень дерева - це рівень h, ми починаємо з батьківських вершин рівня $h - 1$, перебираючи їх *справа наліво*. Ми порівнюємо кожну вершину-батька з її дочірніми вершинами. Якщо значення батьківської вершини менше, тоді міняємо її місцями із найбільшою з її дочірніх вершин. Коли ми закінчуємо з рівнем h – 1, знаємо, що для всіх вузлів даного рівня у нас $A[i] \geq A[2i + 1]$ та $A[i] \geq A[2i + 2]$ (Рис.2). Сірим кольором відзначені вершини, які поміняли місцями. Дерево, яке ми отримали у результаті, зображено праворуч. Під кожним деревом знаходиться масив A, в якому відбуваються справжні перестановки, тому що насправді існує лише масив, а дерево – лише наочна візуалізація.

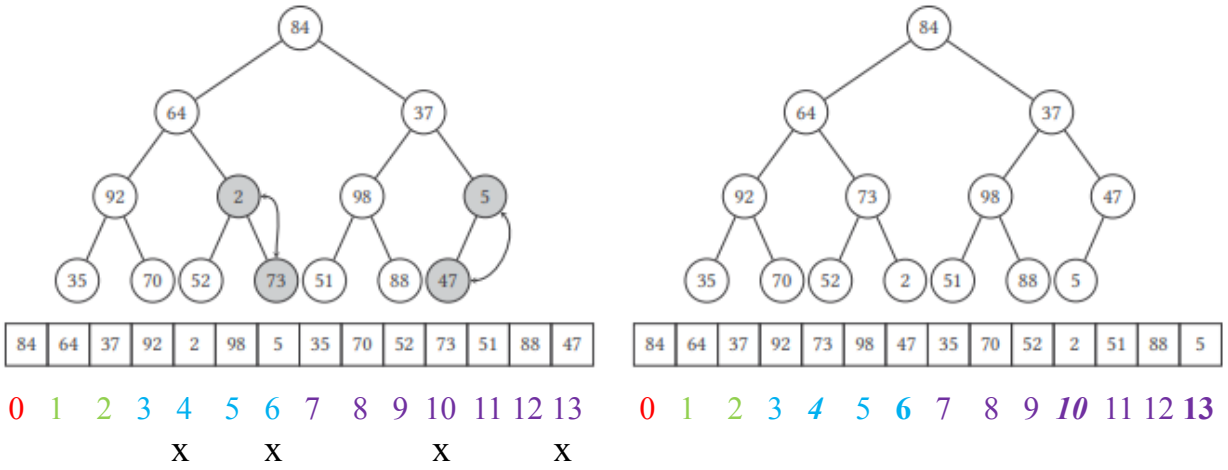


Рис.2 Візуалізація створення максимальної купи на другому рівні.

Далі повторюємо ту саму операцію з рівнем *h – 2*, проте цього разу, якщо ми здійснюємо заміну, потрібно перевірити, що відбувається з нижчим рівнем, тобто з *h – 1* рівнем (Рис. 3). Змінюємо місцями вершини 37 і 98. Якщо ми їх змінюємо, то дізнаємося, що 37 мають дочірні вершини, 51 і 88, які не відповідають нашій умові. Ми можемо це виправити, вчинивши, як минулого разу: знайдемо найбільшу з 51 та 88 і поміняємо її місцями з 37. Ті самі зміни відбуваються з вершиною 64, яка опускається до вершини 70.

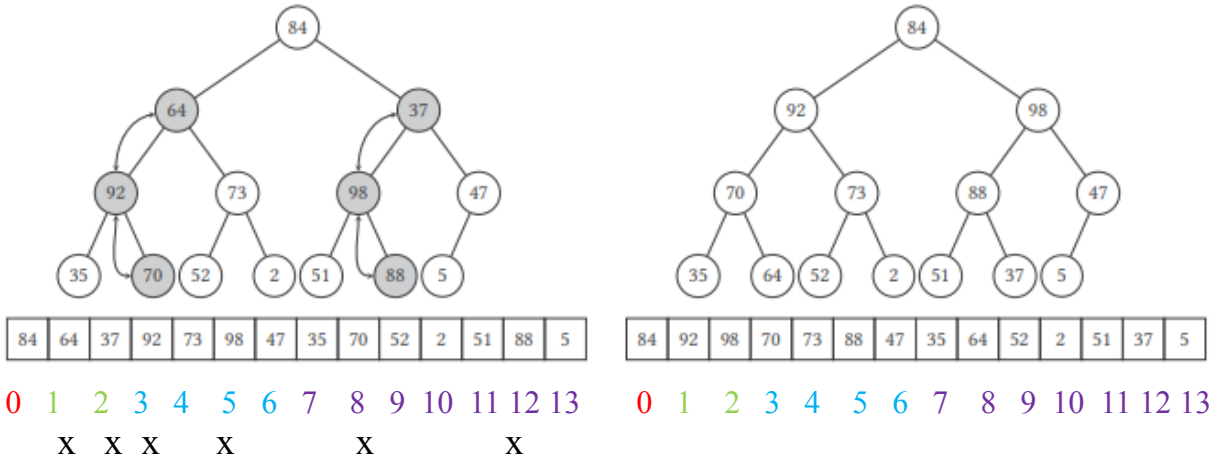


Рис.3 Візуалізація створення максимальної купи на першому рівні

Нарешті ми дісталися кореневого рівня. Нам треба поміняти місцями 84 та 98. Після цього нам треба поміняти 84 та 88 (Рис. 4).

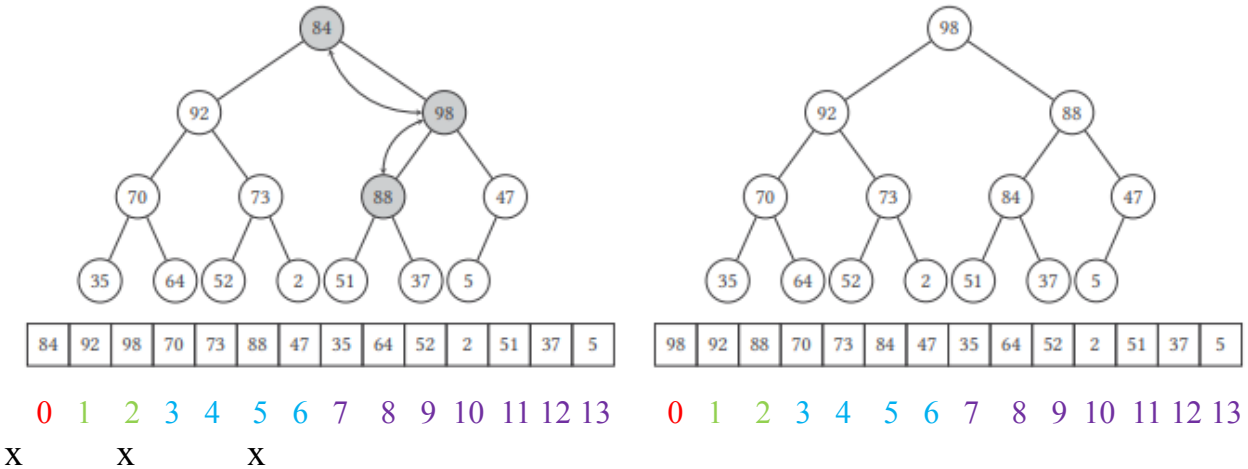


Рис.4 Візуалізація створення максимальної купи на кореневому рівні

Якщо простежити, яким чином вершини переміщуються по конструкції максимальної купи, можна сказати, що зі свого колишнього місця вони *занурилися* на потрібний рівень. Вони опустилися на рівень нижче, тоді як вершини, чиє місце вони зайняли, піднялися нагору і зайняли належні їм місця. Покажемо псевдокод описаної процедури занурення:

```
Sink(A, i, n)
// Вхідні дані: масив A, індекс i елемента, який потрібно занурити на місце,
// n – кількість елементів масиву, що підлягають обробці
// Вихідні дані: масив A з елементом A[i], який занурився на потрібне місце.
k = i
placed ← false
j ← 2k + 1
while not placed and j < n do
        if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then // Compare(A[j], A[j + 1]) < 0 аналогічно  if(A[j] <A[j + 1])
                j ← j + 1
        if Compare(A[k], A[j]) ≥ 0 then     // Compare(A[k], A[j]) ≥ 0 аналогічно  if(A[k] ≥A[j])
                placed ← true
        else
                Swap(A[k], A[j])
        k ← j
        j ← 2k + 1
```

Необхідно зауважити, що алгоритм занурення створює максимальну купу і лежить в основі сортувального алгоритму, який ми розробляємо. Спочатку ми використовуємо Sink, як показано на рисунках 2-4, для перетворення наших даних на купу. На рисунку 2 показано, що відбувається з Sink(A, i, |A|), для i = 6, 5, 4, 3; на рисунку 3 показано, що відбувається із Sink(A, i, |A|) для i = 2, 1; і нарешті на рисунку 4 показана ситуація для Sink(A, 0, |A|).

3. Сортування масиву за допомогою купи

Тепер, отримавши купу, ми знаємо, що максимальний з усіх наших елементів є коренем дерева; він повинен знаходитись у самому кінці, коли всі елементи будуть відсортовані по порядку. Тому дістаємо його з кореня купи і міняємо місцями з останнім елементом купи, A[n – 1]. При цьому необхідно пам'ятати, що дерево насправді – це замаскований масив. Перші n – 1 елементів масиву не максимальна купа, оскільки відбулася заміна кореневого елемента. Тому необхідно запустити алгоритм занурення для перших n – 1 елементів, щоб занурити щойно змінений місцями A[0] у належне йому місце: Sink(A, 0, |A| – 1). По суті, ми працюємо з купою, яка зменшилася на один елемент. Коли алгоритм занурення завершить роботу, у нас знову буде максимальна купа, нагорі якої буде другий за величиною елемент. Ми тоді можемо поміняти місцями A[0] і останній елемент нинішньої купи, елемент A[n – 2]. Елементи A[n – 2] і A[n – 1] тоді будуть знаходитися на відповідних їм позиціях, дозволяючи нам продовжити процес, щоб створити з n – 2 елементів, що залишилися, максимальну купу з Sink(A, 0, |A| – 2) . У кінці всі елементи будуть відсортовані, починаючи з кінця масиву A і до його початку. Покажемо псевдокод сортування купою.

```
HeapSort(A)
// Вхідні дані: не впорядкований масив A
// Вихідні дані: впорядкований масив A
n ← |A|
for i ← (n /2 − 1)  to −1 do                    перша фаза  - побудова максимальної купи
        Sink(A, i, n)
while n > 0 do                                  друга фаза
        Swap(A[0], A[n − 1])
        n ← n − 1
        Sink(A, 0, n)                           занурення
```

Розглянемо принцип роботи HeapSort (A). Спочатку заносимо в змінну n загальний розмір A і приступаємо до *першої фази* сортування купою, в якій (цикл for) ми перетворюємо масив на максимальну купу за допомогою виклику Sink(A, i, n) n = |A| для всіх i, яким відповідають *внутрішні вершини дерева* i = (n − 1)/2=6 . Наприклад на рис. 4 це буде вершина зі значенням 47 Останньою вершиною дерева буде дочірня вершина 5, що знаходиться в позиції n – 1. Сама батьківська вершина перебуває в позиції [(n –1)/2]. Значення i, яким відповідають позиції внутрішніх вузлів, відповідно, дорівнюють [(n –1)/2], [(n –1)/2] – 1, …, 0. При цьому у циклах for межі to не включені, тому, щоб отримати 0, область циклу має бути to –1. Фаза побудови купи у сортуванні купою докладно показана на рисунках 2-4. Після створення купи переходимо до *другої фази* сортування купою, в якій ми запускаємо цикл while. Ми повторюємо цикл кількість разів, що дорівнює кількості елементів у масиві A. Беремо перший елемент, A[0], який є найбільшим з A[0], A[1],…, A[n – 1], міняємо його місцями з A[n – 1], зменшуємо n на один і перебудовуємо купу з перших n – 1 елементів масиву A. У нашому прикладі з A друга фаза сортування купою починається так, як показано на рисунку 5. Кожен елемент береться з вершини купи, тобто з першої позиції A, і поміщається в позиції n – 1, n – 2, …, 1. Щоразу, коли так відбувається, *замінений елемент умовно кладеться нагору купи, а потім занурюється на відповідне йому у дереві місце*. Після цього найбільший з невідсортованих елементів, що залишилися, знову знаходиться в першій позиції A, і ми можемо повторити всю попередню процедуру зі зменшеною купою. На рисунках 5 і 6 ми виділяємо сірим кольором елементи, які вже зайняли свої кінцеві позиції, і прибираємо їх з дерева, щоб підкреслити зменшення купи з кожним циклом. На рисунку 7 показана процедура «занурення», коли замінений елемент (елемент n – 1 зі значенням 5) поміщається у вершину дерева, далі він переміщається ("занурюється") по дорозі вниз на основі порівняння з дочірніми елементами.

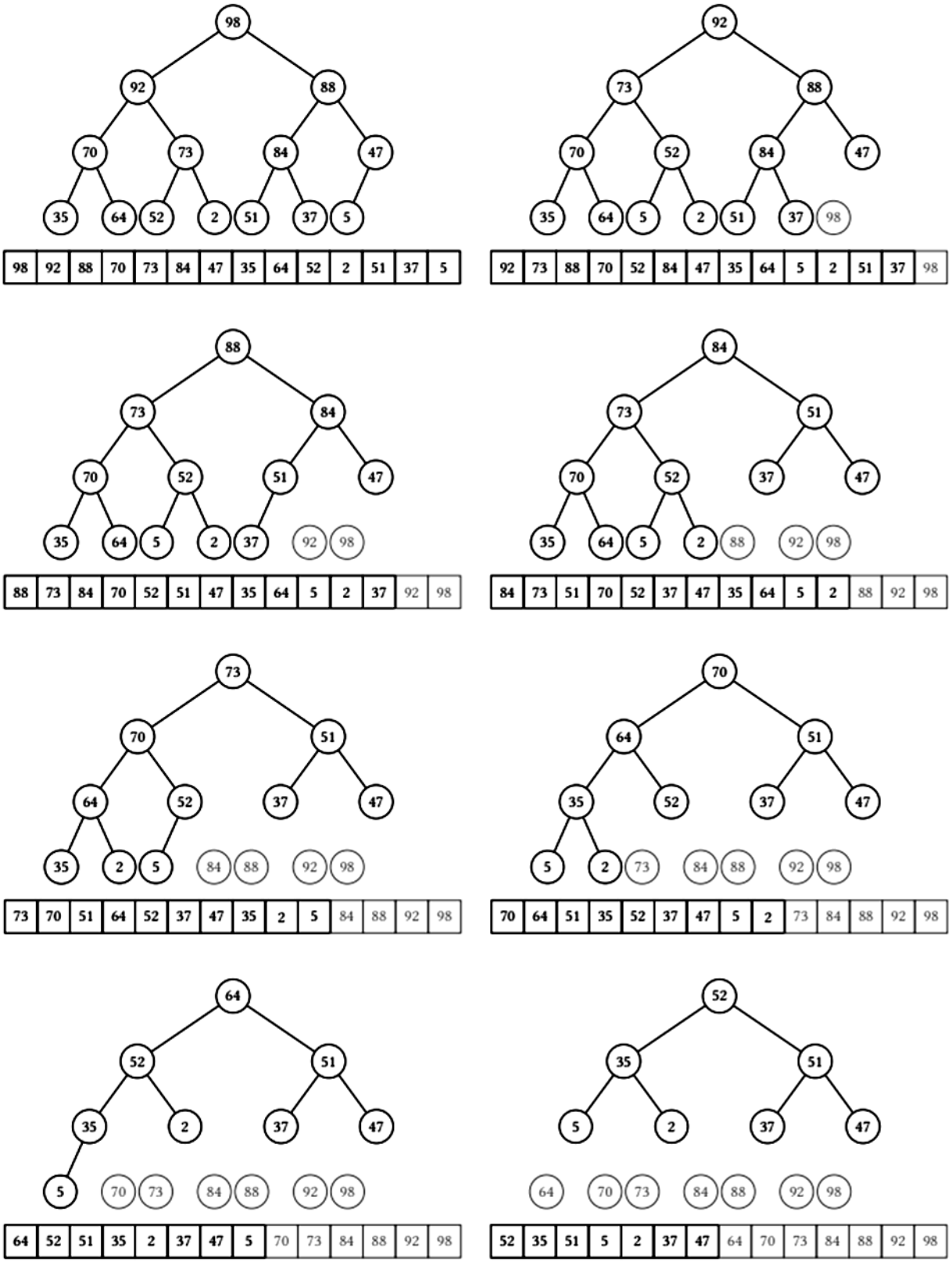


Рис.5 Візуалізація другої фази сортування купою (занурення).

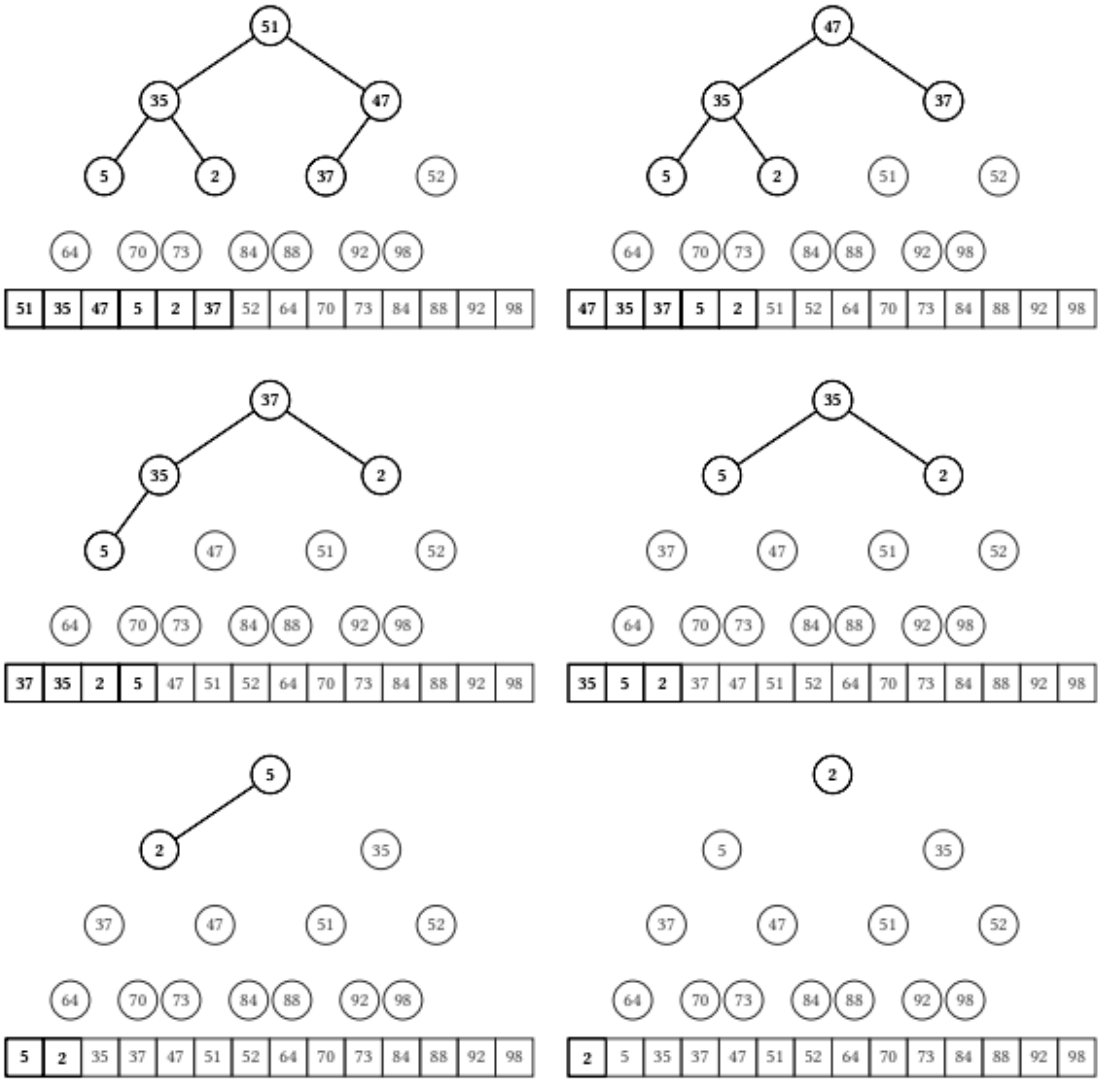


Рис.6 Продовження візуалізації другої фази сортування купою.

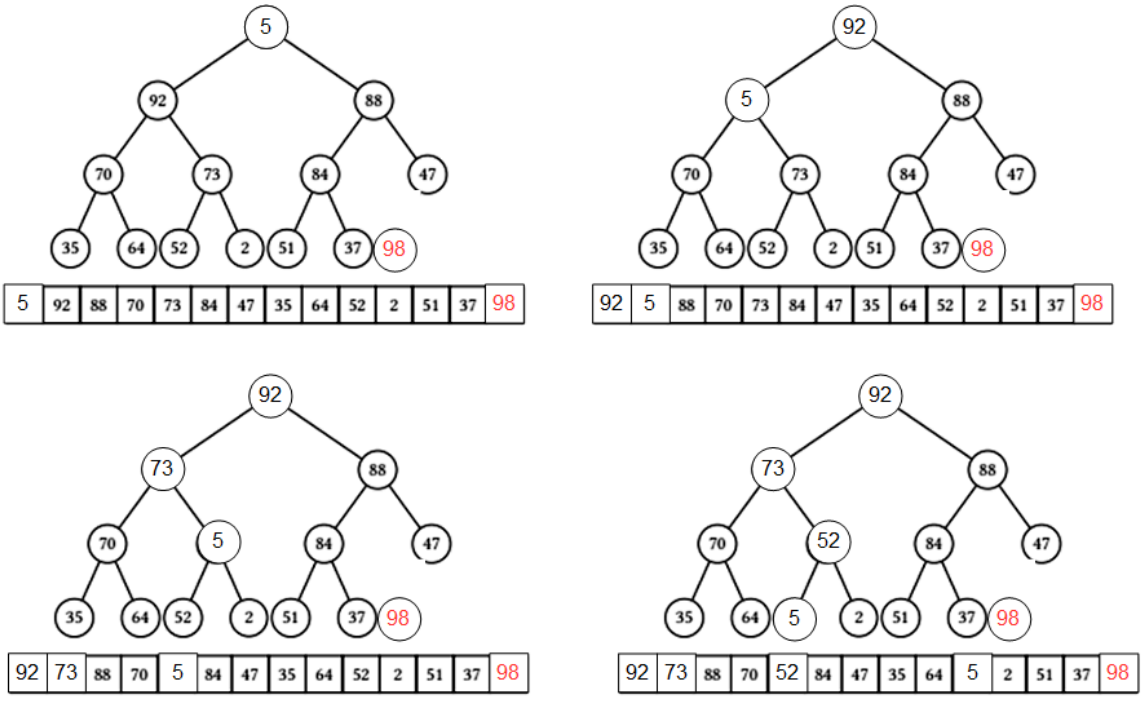


Рис.7 Побудова максимальної купи шляхом занурення для n – 1 елемента.

4. Приклад моделювання.

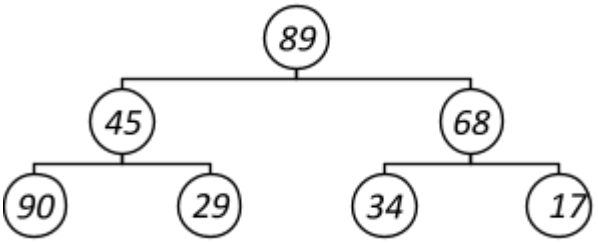
Виконаємо моделювання коду за кроками для наступного прикладу $A=[89_0, 45_1, 68_2, 90_3, 29_4, 34_5, 17_6]$ n=7

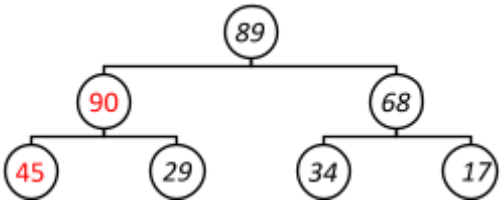
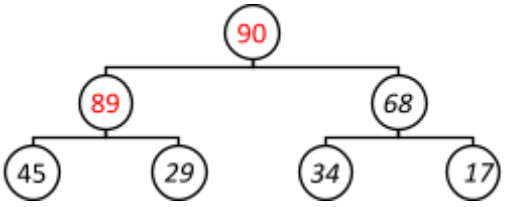
```
HeapSort(A)
// Вхідні дані: не впорядкований масив А
//Вихідні дані: упорядкований масив А
n ← |A|
for i ← (n /2 – 1)  to −1 do                    перша фаза  - побудова максимальної купи
        Sink(A, i, n)
while n > 0 do                                  друга фаза
        Swap(A[0], A[n − 1])
        n ← n − 1
        Sink(A, 0, n)                           просіювання
```

Sink(A, i, n)

```
// Вхідні дані: масив А, індекс і елемента, який потрібно занурити на місце,
// n – кількість елементів масиву, що підлягають обробці
// Вихідні дані: масив А з елементом А[i], який занурився на потрібне місце.
k = i
placed ← false
j ← 2k + 1
while not placed and j < n do
        if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then // Compare(A[j], A[j + 1]) < 0 аналогічно  if(A[j] <A[j + 1])
                j ← j + 1
        if Compare(A[k], A[j]) ≥ 0 then     // Compare(A[k], A[j]) ≥ 0 аналогічно  if(A[k] ≥A[j])
                placed ← true
        else
                Swap(A[k], A[j])
        k ← j
        j ← 2k + 1
```

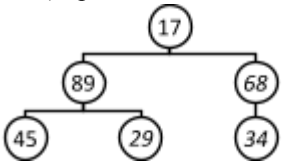
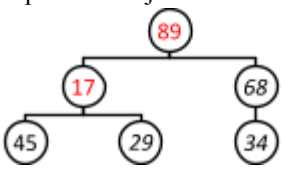
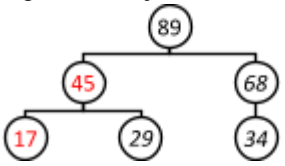
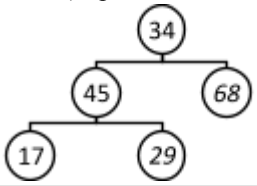
Таблиця 1

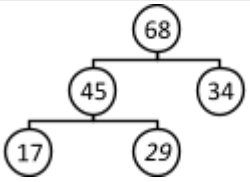
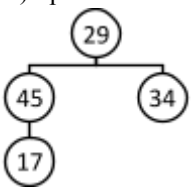
$A=[89_0, 45_1, 68_2, 90_3, 29_4, 34_5, 17_6]$ 0 1 2 3 4 5 6	
n = 7 for i = (7/2 - 1) 2 to -1 do Sink (A, 2, 7)	HeapSort(A) n ← |A| for i ← (n/2 − 1) to −1 do Sink(A, i, n)
k = i (2); placed = false; j = 2k + 1 (5) while not placed and 5 < 7 True	k = i placed ← false j ← 2k + 1 while not placed and j < n do
if 5< 6 and Compare(A[5] (34), A[6] (17)) < 0 False if Compare (A[2] (68), A[5] (34)) ≥ 0 placed ← true k = j (5); j = 2k + 1 (11)	if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then j ← j + 1 if Compare(A[k], A[j]) ≥ 0 placed ← true else Swap(A[k], A[j]) k ← j j ← 2k + 1
while not placed and 11 < 7 False	while not placed and j < n do

A=[89, 45, 68, 90, 29, 34, 17] *0 1 2 3 4 5 6*	
Sink (A, 1, 7) k = i (1); placed = false; j = 2k + 1 (3) while not placed and 3 < 7 True	k = i placed ← false j ← 2k + 1 while not placed and j < n do
if 3 < 6 and Compare(A[3] (90), A[4] (29)) < 0 False if Compare (A[1] (45), A[3] (90)) ≥ 0 False else Swap (A[1] (45), A[3] (90)) k = j (3); j = 2k + 1 (7)	if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then j ← j + 1 if Compare(A[k], A[j]) ≥ 0 placed ← true else Swap(A[k], A[j]) k ← j j ← 2k + 1
while not placed and 7 < 7 False *A=[89, 90, 68, 45, 29, 34, 17]* *0 1 2 3 4 5 6*	while not placed and j < n do 
Sink (A, 0, 7) k = i (0); placed = false; j = 2k + 1 (1) while not placed and 1 < 7 True	k = i placed ← false j ← 2k + 1 while not placed and j < n do
if 1 < 6 and Compare(A[1] (90), A[2] (68)) < 0 False if Compare (A[0] (89), A[1] (90)) ≥ 0 False else Swap (A[0] (89), A[1] (90)) k = j (1); j = 2k + 1 (3) *A=[90, 89, 68, 45, 29, 34, 17]* *0 1 2 3 4 5 6*	if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then j ← j + 1 if Compare(A[k], A[j]) ≥ 0 placed ← true else Swap(A[k], A[j]) k ← j j ← 2k + 1 

while not placed and 3 < 7 True
if 3 < 6 and Compare(A[3] (45), A[4] (29)) < 0 False
if Compare (A[1] (89), A[3] (45)) ≥ 0 True
placed ← true
k = j (3); j = 2k + 1 (7)
	while not placed and j < n do
	if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then
	j ← j + 1
	if Compare(A[k], A[j]) ≥ 0
	placed ← true
	else Swap(A[k], A[j])
	k ← j
	j ← 2k + 1

while not placed and 7 < 7 False *A=[90, 89, 68, 45, 29, 34, 17]* *0 1 2 3 4 5 6* максимальна купа	while not placed and j < n do
i=-1 перехід до другої фази while 7 > 0 True Swap (A[0] (90), A[6] (17)) n = 6	while n > 0 do Swap(A[0], A[n − 1]) n ← n − 1

17, 89, 68, 45, 29, 34, |90	
Sink(A, 0, 6) 17, 89, 68, 45, 29, 34 0 1 2 3 4 5	Sink(A, 0, n) просіювання 
k = i (0); placed = false; j = 2k + 1 (1) while not placed and 1 < 6 True	k = i placed ← false j ← 2k + 1 while not placed and j < n do
if 1 < 5 and Compare(A[1] (89), A[2] (68)) < 0 False if Compare (A[0] (17), A[1] (89)) ≥ 0 False else Swap (A[0] (17), A[1] (89)) k = j (1); j = 2k + 1 (3)	if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then j ← j + 1 if Compare(A[k], A[j]) ≥ 0 placed ← true else Swap(A[k], A[j]) k ← j j ← 2k + 1
while not placed and 3 < 6 True 89, 17, 68, 45, 29, 34 0 1 2 3 4 5	while not placed and j < n do 
if 3 < 5 and Compare(A[3] (45), A[4] (29)) < 0 False if Compare (A[1] (17), A[3] (45)) ≥ 0 False else Swap (A[1] (17), A[3] (45)) k = j (3); j = 2k + 1 (7)	if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then j ← j + 1 if Compare(A[k], A[j]) ≥ 0 placed ← true else Swap(A[k], A[j]) k ← j j ← 2k + 1
while not placed and 7 < 6 False 89, 45, 68, 17, 29, 34 0 1 2 3 4 5	while not placed and j < n do 
while 6 > 0 True Swap (A[0] (89), A[5] (34)) n = 5 34, 45, 68, 17, 29, | 89, 90	while n > 0 do Swap(A[0], A[n − 1]) n ← n − 1 друга фаза
Sink(A, 0, 5) 34, 45, 68, 17, 29 0 1 2 3 4	Sink(A, 0, n) просіювання 
k = i (0); placed = false; j = 2k + 1 (1) while not placed and 1 < 5 True	k = i placed ← false j ← 2k + 1 while not placed and j < n do
if 1 < 4 and Compare(A[1] (45), A[2] (68)) < 0 True j ←1 + 1 (2) if Compare (A[0] (34), A[2] (68)) ≥ 0 False	if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then j ← j + 1 if Compare(A[k], A[j]) ≥ 0 placed ← true else Swap(A[k], A[j])

else Swap (A[0] (34), A[2] (68)) k = j (2); j = 2k + 1 (5)	k ← j j ← 2k + 1
while not placed and 5 < 5 False 68, 45, 34, 17, 29 0 1 2 3 4	
while 5 > 0 True Swap (A[0] (68), A[4] (29)) n = 4 29, 45, 34, 17, |68 89, 90	while n > 0 do Swap(A[0], A[n − 1]) n ← n − 1 друга фаза
Sink(A, 0, 4) 29, 45, 34, 17 0 1 2 3	Sink(A, 0, n) просіювання 

Продовжіть самостійно!!!!

Завдання:

1. Відсортувати послідовність за варіантами завдань за допомогою сортування купою.

Варіанти завдань:

Варіант	Послідовність
1	53, 12, 68, 63, 3, 47, 50, 61, 41
2	98, 40, 42, 88, 61, 87, 79, 97, 82
3	70, 80, 61, 92, 12, 23, 67, 65, 50
4	29, 1, 24, 69, 52, 97, 27, 10, 88
5	95, 43, 98, 41, 68, 67, 10, 7, 69
6	61, 32, 27, 45, 75, 58, 5, 50, 99
7	78, 77, 4, 62, 8, 69, 46, 11, 49
8	28, 76, 27, 10, 5, 35, 95, 16, 33
9	31, 40, 22, 50, 53, 68, 97, 12, 15
10	18, 20, 2, 88, 61, 17, 79, 97, 82
11	41, 52, 47, 65, 95, 38, 15, 50, 99
12	11, 42, 67, 55, 65, 78, 25, 50, 69
13	53, 5, 44, 47, 35, 83, 82, 85, 28
14	77, 89, 74, 68, 70, 49, 5, 62, 51
15	19, 75, 43, 31, 5, 66, 62, 34, 76
16	50, 57, 78, 34, 41, 68, 47, 61, 38
17	86, 36, 14, 50, 64, 21, 2, 83, 82
18	80, 27, 37, 36, 91, 53, 86, 66, 98
19	21, 44, 22, 50, 63, 68, 97, 12, 15
20	7, 89, 4, 68, 70, 49, 10, 62, 51

Варіант	Послідовність
21	54, 65, 7, 33, 86, 29, 11, 91, 12
22	50, 80, 19, 86, 35, 7, 60, 48, 51
23	10, 90, 95, 30, 45, 60, 57, 28, 5
24	90, 10, 15, 80, 100, 6, 57, 5, 29
25	53, 100, 44, 74, 53, 38, 82, 65, 28
26	47, 50, 61, 41,53, 12, 68, 63, 3
27	87, 79, 97, 82,98, 40, 42, 88, 61
28	12, 23, 67, 65, 50, 70, 80, 61, 92
29	69, 52, 97, 27, 10, 88,29, 1, 24
30	41, 68, 67, 10, 7, 69, 95, 43, 98
31	58, 5, 50, 99, 61, 32, 27, 45, 75
32	46, 11, 49, 78, 77, 4, 62, 8, 69
33	35, 95, 16, 33, 28, 76, 27, 10, 5
34	68, 97, 12, 15, 31, 40, 22, 50, 53
35	79, 97, 82, 18, 20, 2, 88, 61, 17
36	38, 15, 50, 99, 41, 52, 47, 65, 95

Виконати моделювання та показати графічно фазу створення максимальної купи (рис.2-4), використовуючи процедуру Sink(A, i, n), підрахуйте кількість операцій.
Виконати моделювання і показати графічно фазу сортування купою (рис.5,6), а також фази просіювання (рис.7), використовуючи процедуру Sink(A, 0, n), підрахуйте кількість операцій.

5.Вимоги до представлення звіту

Звіт повинен містити:

1. Титульну частину (лист) з вказівкою на назву лабораторної роботи, ПІБ студента, групу, номер варіанту та послідовність, яку необхідно отсортувати.
2. Псевдокод сортування купою, результат його чисельного моделювання (див. Таблицю 1) за варіантами завдань
3. Результати порівнянь сортування купою (за порівняннями та присвоюваннями) із сортуванням за алгоритмами ЛР1 та ЛР2
4. Висновки

Лабораторна робота №4

Алгоритми на графах. Мінімальне кістякове дерево

Розглядаються графи, засоби представлення графів, визначення мінімального кістякового дерева графа, алгоритми Прима (Prim) і Крускала (Kruskal) для побудови мінімального кістяка дерева

Питання:

1. Представлення графів
2. Визначення мінімального кістякового дерева графа
3. Алгоритм Прима
4. Алгоритм Крускала
5. Завдання і оформлення звіту дивись у файлі завдання з Практичної роботи №4

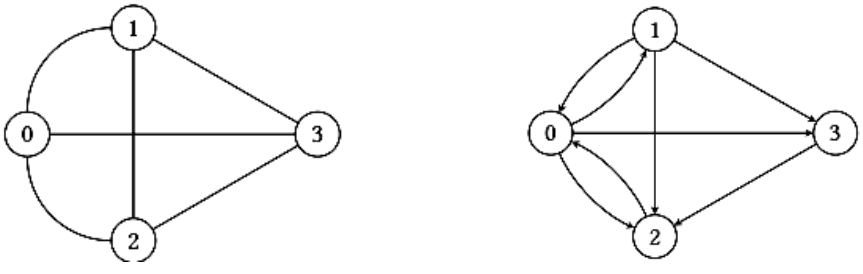
1. Представлення графів, обходи графів.

Граф – абстрактний математичний об'єкт, що являє собою множину *вершин* графа і набір *ребер*, тобто з'єднань між парами вершин. Наприклад, як безліч вершин можна прийняти безліч аеропортів, що обслуговуються деякою авіакомпанією, а як безліч ребер представити регулярні рейси цієї авіакомпанії в ці аеропорти. У деяких графах потрібно встановити напрямок ребрам як у випадку з аеропортом; такі графи називаються *орієнтованими* графами, чи *оргграфами*. В іншому випадку ми маємо справу з *неорієнтованими* графами. Якщо в графі є шлях від одного вузла до іншого, такий граф називається *зв'язним*. Інакше йдеться про *незв'язний* граф. Графи із циклами називаються циклічними, а графи без циклів – ациклічними. Досить широке застосування мають орієнтовані ациклічні графи (ОАД).

Як правило, множина вершин позначається як V, а множина ребер – E. У такому разі граф G являє собою $G = (V, E)$. У неорієнтованих графах множина E це множина, що складається з двох наборів елементів (V_i, V_j) для кожного ребра між двома вершинами графа V_i и V_j. Для графа на рис.1,а множина вершин та ребер і відображається так:

V = {0, 1, 2, 3}.

E = {(0, 1), (0, 2) (0, 3), (1, 2), (1, 3), (2, 3)}



а б

Рис. 1 Приклади неорієнтованого (а) та орієнтованого (б) графів

У орієнтованих графах множина E – це множина, що складається з кортежів двох елементів $<V_i, V_j>$ для кожного ребра між двома вершинами графа V_i та V_j. Порядок V_i та V_j важливий, так як він відповідає за напрямок зв'язку між вершинами..

Граф на рисунку 1,б відображається як:

V = {0, 1, 2, 3}

E = {<0, 1>, <0, 2>, <0, 3>, <1, 0>, <1, 2>, <1, 3>, <2, 0>, <3, 2>}

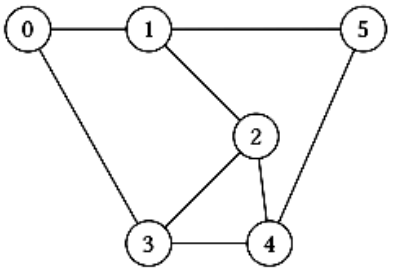
При розробці алгоритмів з використанням структур даних у вигляді графів їх представляють двома способами:

— у вигляді матриці суміжності (вагових коефіцієнтів);

— у вигляді зв'язаних списків суміжних вершин.

Матриця суміжності графа є квадратною матрицею, в якій для кожної вершини відведені рядок та стовпець. У матриці суміжності на перетині рядка під номером *i* та стовпця *j* містяться «0», якщо між вершинами графа V_i і V_j відсутнє ребро та «1», якщо таке ребро є. Матриця суміжності для графа, зображеного на рисунку 2,б, показано на рисунку 2,а. Як бачимо, розмір матриці суміжності V^2

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	0	1	0	0
1	1	0	1	0	0	1
2	0	1	0	1	1	0
3	1	0	1	0	1	0
4	0	0	1	1	0	1
5	0	1	0	0	1	0



а б

Рис. 2 Представлення графа як матриці суміжності

Щоб зменшити обсяг необхідної пам'яті для представлення вершин графа використовують масив, кожен елемент якого позначає одну з вершин та запускає кортеж, що складається з вершин, що є сусідами з обраною вершиною. Такий кортеж називається списком суміжних вершин графа. Список – це структура даних, у якій містяться елементи – вузли списку, які складаються з двох частин. Перша частина містить дані, що описують елемент, а друга частина містить покажчик на наступний елемент списку. Щоб створити представлення графа за допомогою списку суміжних вершин, необхідні:

— *L*← *CreateList*() створює та повертає новий, порожній список.

— *InsertInList*(*L*, *p*, *d*) додає до списку *L* вузол, що містить дані *d*, після вузла *p*. Якщо *p* – null, тоді нам потрібен новий вузол, який стане новою головою списку.

Щоб переглянути елементи списку L, нам потрібно викликати:

n ← *GetNextListNode*(*L*, *null*) та отримати перший елемент; потім, поки n ≠ null. Можемо повторно призначити n ← GetNextListNode(L, n). Можна отримати доступ до даних усередині вузла, наприклад, за допомогою функції *GetData*(*n*), яка повертає дані *d*, що зберігаються у вузлі *n*

Наприклад, якщо є числа 3, 1 і 0 і ми в такому порядку додаємо їх на початок списку, то список збільшиться від [ ] до [0, 1, 3], як показано на рисунку 3.

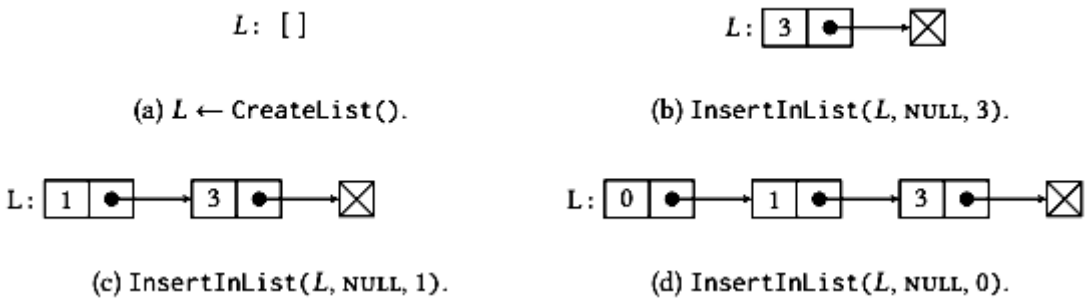
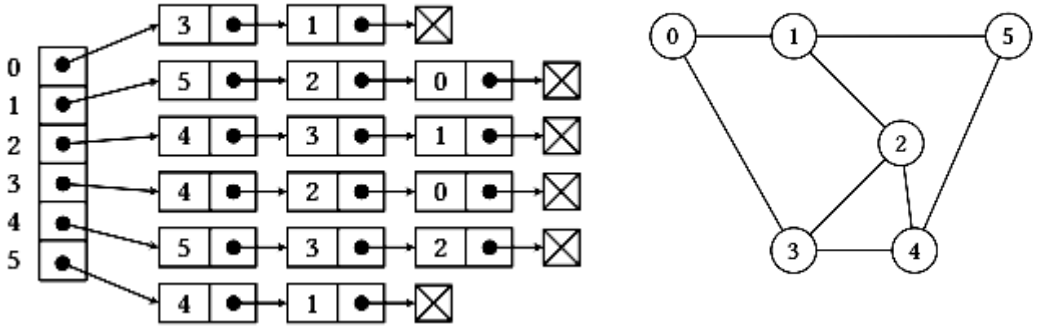


Рис. 3 Додавання вузлів на початок списку

Таким чином, за допомогою структури даних – списки ми створюємо представлення графа у вигляді списку суміжності. У цьому представленні ми маємо один список суміжності для кожної вершини. Усі вони об'єднані в масив, який вказує на початок списків суміжних вершин. Якщо є масив A, об'єкт A[i] даного масиву вказує на початок списку суміжності вузла i. Якщо поруч із вузлом і немає інших вузлів, A[i] вказуватиме на null..

Для графа на рисунку 4, б показаний список суміжності на рисунку 4,а Зліва на рисунку 4,а розміщений масив, що містить голови списку суміжності графа; кожному об'єкту масиву відповідає певна вершина. Список суміжності містить ребра вершини, що стоїть на чолі. Наприклад, третій елемент масиву містить голову списку суміжності для вершини 2. Кожен список суміжності складається із сусідніх вершин, розташованих у числовому порядку. Наприклад, щоб створити список суміжності для вершини 1, ми викликаємо додавання вузлів на початок списку *InsertInList* (1, null, *d*) три рази: для вершин 0, 2 та 5, саме в такому порядку. Таким чином, додавши їх у порядку 0, 2, 5, ми у результаті отримаємо список [5, 2, 0].

Список суміжних вершин для графа, зображеного на рисунку 4, б показаний на рисунку 4,а



а б

Рис. 4 Представлення графа як матриці суміжності

2. Визначення мінімального кістякового дерева графа

У багатьох практичних ситуаціях виникає така задача: потрібно з'єднати *n* даних точок таким чином, щоб з кожної точки існував шлях до будь-якої іншої, причому сумарна вартість з'єднань повинна бути мінімальною. Точки

можна представити вершинами графа, а вартості з'єднань – вагами ребер. У такій моделі задача перетворюється на задачу про мінімальне кістякове дерево, формально визначається наступним чином.

Кістякове (*покриваюче рос. остовное*) *дерево* (*spanning tree*) зв'язного графа є *зв'язним ациклічним* підграфом (тобто деревом), яке містить всі вершини графа. *Мінімальне кістякове дерево* (*minimum spanning tree*) зваженого зв'язкового графа є кістякове дерево з найменшою вагою, де вага дерева визначається як сума ваг усіх його ребер. Задача про мінімальне кістякове дерево є задача пошуку мінімального кістякового дерева для даного зваженого зв'язного графа (рис. 5)

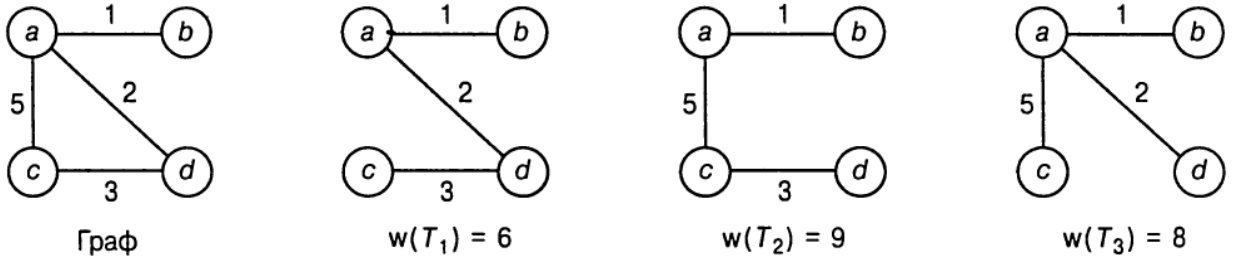


Рис. 5. Граф та його кістякові дерева. T_1 – мінімальне кістякове дерево

3. Алгоритм Прима

Алгоритм Прима (Prim's algorithm) був розроблений у 1957 році. Згідно з алгоритмом, мінімальне кістякове дерево будується як послідовність піддерев, що розширюються. Початкове піддерево такої послідовності складається з єдиної вершини, довільно вибраної з множини вершин графа V. На кожній ітерації ми розширюємо поточне дерево *жадібним* чином, додаючи до нього найближчу вершину, тобто з'єднану з вихідною вершиною дерева ребром з мінімальною вагою, що не входить у дерево. Алгоритм завершує роботу після того, як всі вершини виявляються включеними в дерево, яке будується. Оскільки алгоритм розширює дерево по одній вершині за ітерацію, загальна кількість таких ітерацій дорівнює n-1 де n – кількість вершин графа.

Жадібний алгоритм – алгоритм, що полягає у прийнятті *локально оптимальних рішень* на кожному етапі, допускаючи, що кінцеве рішення також виявиться оптимальним.

Алгоритм Prim (G)
// Вхідні дані: Зважений зв'язний граф $G = (V, E)$
// Вихідні дані: E_T, множина ребер, що складають мінімальне
// кістякове дерево графа G
$V_T \leftarrow \{v_0\}$
$E_T \leftarrow \emptyset$
for i ← 1 to |V| - 1 **do**
Пошук ребра з мінімальною вагою $e^* = (v^*, u^*)$ серед всіх
ребер (v, u) таких, що $v \in V_T$ та $u \in V - V_T$
$V_T \leftarrow V_T \cup \{u^*\}$
$E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Дамо наступні пояснення. Відповідно до алгоритму Прима кожній вершині, що не входить до поточного дерева, необхідно додати інформацію про найкоротше ребро, що з'єднує її з вершиною дерева, яке будується.

Варіант 1:

Ми можемо зробити це, приєднуючи до вершин дві мітки: ім'я суміжної вершини дерева і ваги відповідного ребра. Вершини, які не є суміжними для жодної з вершин дерева, можуть бути позначені міткою ∞, яка вказує на "нескінченну" відстань до вершин дерева і нульовим значенням у полі найближчої вершини дерева.

Варіант 2

Алгоритм полягає у поділі всіх вершин, що не входять у дерево, на дві множини – "примежових" і "непомічених". Примежова множина містить лише ті вершини, які, не належать дереву та є суміжними принаймні з однією з його вершин. Непомічені вершини – всі інші вершини графа, які називаються тому, що ще не досліджені алгоритмом. При використанні таких міток пошук наступної вершини, що додається в поточне дерево T = (V_T, E_T), стає простою задачею пошуку вершини з найменшою міткою відстані серед вершин множини V - V_T.

Після того як ми знайдемо вершину u*, яка має бути додана в дерево, треба виконати наступні дві операції:

1. Перемістити вершину u* з множини V - V_T до множини вершин дерева V_T
2. Для кожної вершини, що залишаються у множині V - V_T і зв’язаних з u* більш коротким ребром, ніж її поточна мітка відстані, слід оновити її мітку імені суміжної вершини ім'ям u*, а мітку відстані - відстанню до вершини u*

У таблиці 1 показаний хід побудови мінімального кістякового дерева за алгоритмом Прима

Позначення:

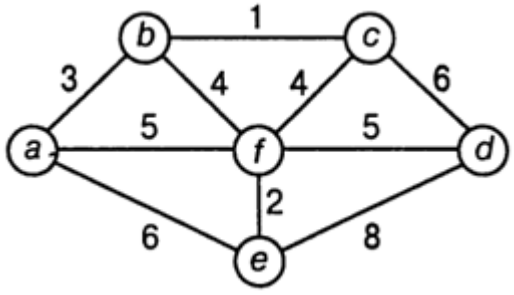
V_T – переглянуті вершини

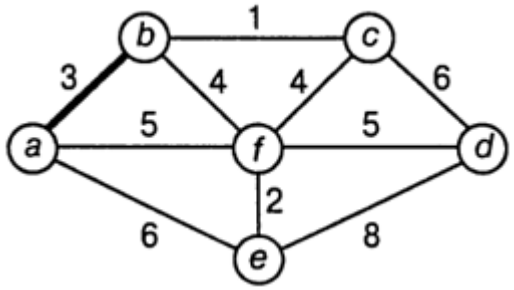
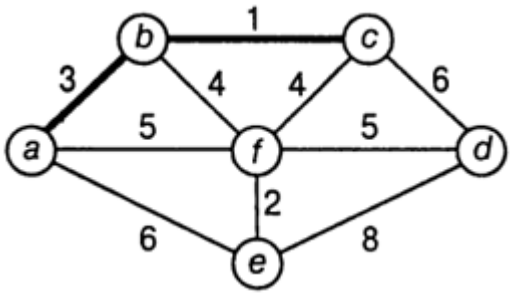
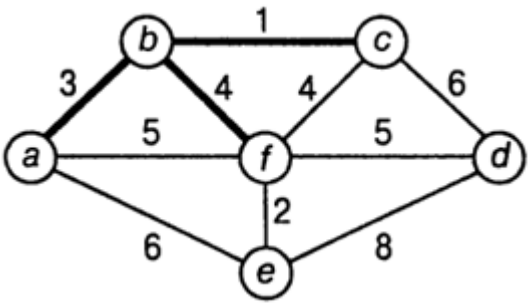
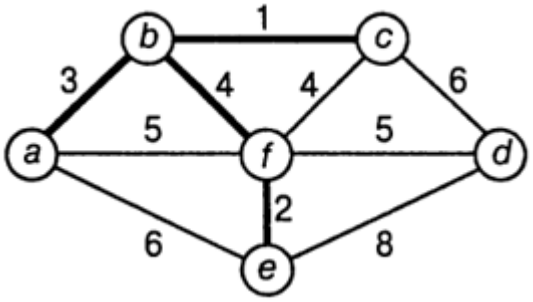
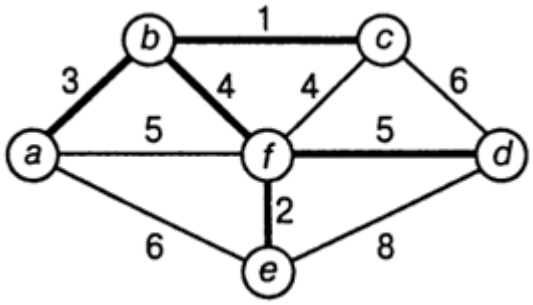
E_T, множина ребер, що становлять мінімальне кістякове дерево графа G

E – множина ребер примежових переглянутим вершинам

V - V_T – непереглянуті вершини

Таблиця 1

Зображення	V_T	E_T	E	V - V_T
	{a}		(a,b) = 3 v (a,f) = 5 (a,e) = 6	{b,c,d,e,f}

Зображення	V_T	E_T	E	$V - V_T$
	{a,b}	(a,b) = 3	(a,f) = 5 (a,e) = 6 (b,c) = 1 **v** (b,f) = 4	{c,d,e,f}
	{a,b,c}	(a,b) = 3 (b,c) = 1	(a,f) = 5 (a,e) = 6 (b,f) = 4 **v** (c,f) = 4 (c,d) = 6	{d,e,f}
	{a,b,c,f}	(a,b) = 3 (b,c) = 1 (b,f) = 4	*(a,f) = 5 c* (a,e) = 6 *(c,f) = 4 c* (c,d) = 6 (f,e) =2 **v** (f,d) =5	{d,e}
	{a,b,c,f,e}	(a,b) = 3 (b,c) = 1 (b,f) = 4 (f,e) =2	(a,f) = 5 c (a,e) = 6 c (c,f) = 4 c (c,d) = 6 (f,d) =5 **v**	{d}
	{a,b,c,f,e,d}	(a,b) = 3 (b,c) = 1 (b,f) = 4 (f,e) =2 (f,d) =5	(a,f) = 5 c (a,e) = 6 c (c,f) = 4 c (c,d) = 6 c	{}

Мінімальне кістякове дерево побудовано. Його вартість становить:

$$e^* = \sum_{i=0}^{|E_T|-1} e_i^* = 15.$$

Код алгоритму Прима:

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
```

```
#define INF 9999999
// number of vertices in grapj
#define V 6
// create a 2d array of size 6x6
//for adjacency matrix to represent graph
int G[V][V] = {
  {INF, 3, INF, INF, 6, 5},
  {3, INF, 1, INF, INF, 4},
  {INF, 1, INF, 6, INF, 4},
  {INF, INF, 6, INF, 8, 5},
  {6, INF, INF, 8, INF, 8},
  {5, 4, 4, 5, 2, INF},
};
int main () {
  int no_edge;            // number of edge
  // create a array to track selected vertex
  // selected will become true otherwise false
  int selected[V];
 // set selected false initially
  memset (selected, false, sizeof (selected));
 // set number of edge to 0
  no_edge = 0;
  // the number of egde in minimum spanning tree will be
  // always less than (V -1), where V is number of vertices in
  //graph
  // choose 0th vertex and make it true
  selected[0] = true;
  int x;            //  row number
  int y;            //  col number
  // print for edge and weight
  cout << "Edge" << " : " << "Weight";
  cout << endl;
  while (no_edge < V - 1) {
  //For every vertex in the set S, find the all adjacent vertices
  // , calculate the distance from the vertex selected at step 1.
  // if the vertex is already in the set S, discard it otherwise
  //choose another vertex nearest to selected vertex  at step 1.
      int min = INF;
      x = 0;
      y = 0;
      for (int i = 0; i < V; i++) {
        if (selected[i]) {
            for (int j = 0; j < V; j++) {
              if (!selected[j] && G[i][j]) { // not in selected and there is an edge
                  if (min > G[i][j]) {
                      min = G[i][j];
                      x = i;
                      y = j;
                  }
              }
          }
        }
      }
      cout << x <<  " - " << y << " :  " << G[x][y];
      cout << endl;
      selected[y] = true;
      no_edge++;
    }
  return 0;
}
```

Запустивши наведений вище код, ми отримаємо виведення у вигляді:

```
Edge : Weight
0 - 1 :  3
1 - 2 :  1
1 - 5 :  4
5 - 4 :  2
5 - 3 :  5
```

4. Алгоритм Крускала

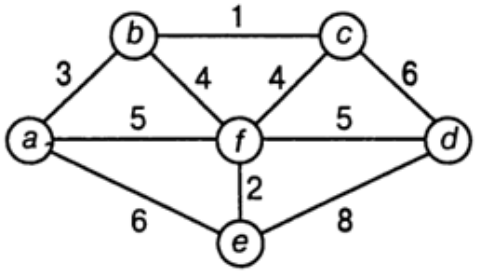
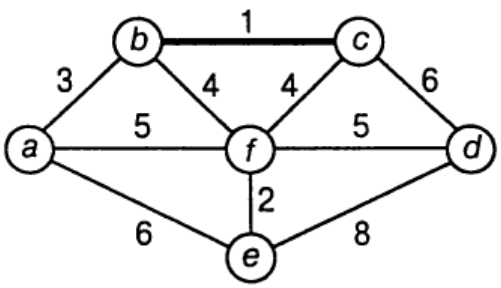
Алгоритм Кру(а)скала (Kruskal's algorithm) також відносится до класу «жадібних». Джозеф Крускал (Joseph Kruskal) запропонував его, будучи студентом-другокурсником. Алгоритм Крускала шукає мінімальне кістякове дерево зваженого зв'язного графа $G = (V, E)$ як ациклічний підграф з |V| - 1 ребрами, сума ваг яких мінімальна. При цьому алгоритм будує мінімальне

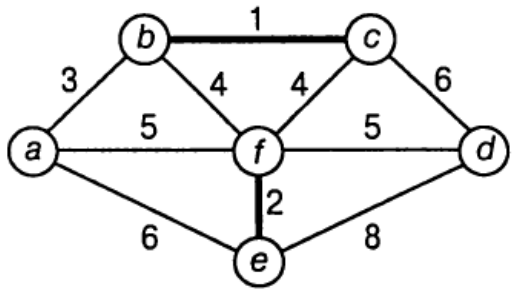
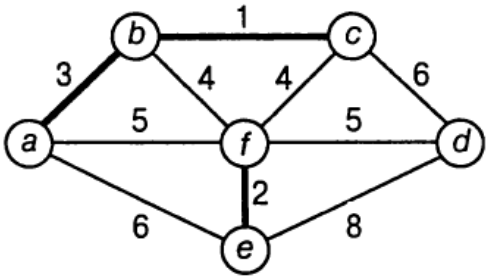
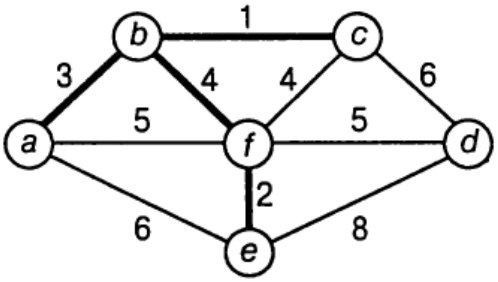
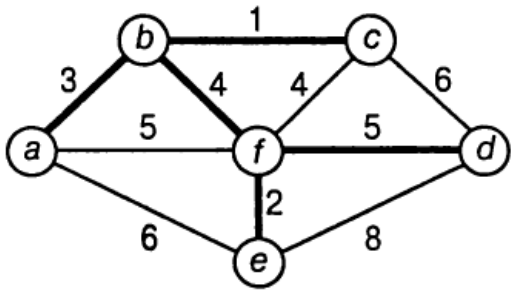
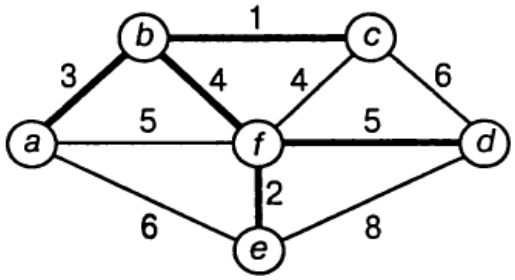
кістякове дерево як послідовність підграфів, що розширюється, які завжди ациклічні, але на проміжних стадіях не завжди зв'язні. Алгоритм починає з сортування ребер графа в неспадному порядку їх ваг. Потім, починаючи з порожнього поточного підграфа, переглядає відсортований список і додає чергове ребро до списку поточного підграфа, якщо не створюється цикл; інакше ребро просто пропускається.

Алгоритм Kruskal (G)
// Вхідні дані: Зважений зв'язний граф $G = (V, E)$
// Вихідні дані: E_T, множина ребер, що складають мінімальне
// кістякове дерево G
Сортування множини E в неспадному порядку ваг ребер
$e^*_{i_0} \le ... e^*_{i_{|E|-1}}$
$E_T \leftarrow \emptyset$
ecounter←0
k←0
while *ecounter*< |V| - 1**do**
    k=*k*+1
    if $E_T \cup \{e_{i_k}\}$ – ациклічний граф
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$; *ecounter= ecounter+1*
return E_T

У таблиці 2 показаний хід побудови мінімального кістякового дерева за алгоритмом Крускала

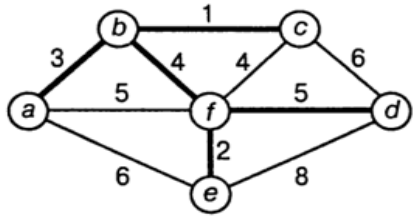
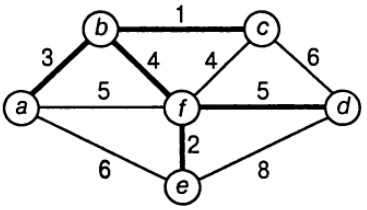
Таблиця 2

Зображення	E_T	E
	{}	{bc =1 ef=2 ab=3 bf =4 cf=4 af =5 df =5 ae=6 cd=6 de =8}
	{ bc =1}	{**bc =1** ef=2 ab=3 bf =4 cf=4 af =5 df =5 ae=6 cd=6 de =8}

Зображення	E_T	E
	{ bc =1 ef=2 }	{bc =1 **ef=2** ab=3 bf =4 cf=4 af =5 df =5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 }	{bc =1 ef=2 **ab=3** bf =4 cf=4 af =5 df =5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 bf =4 }	{bc =1 ef=2 ab=3 **bf =4** cf=4 af =5 df =5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 bf =4 df =5 }	{bc =1 ef=2 ab=3 bf =4 cf=4 (c) af =5 (c) **df =5** ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 bf =4 df =5 }	{bc =1 ef=2 ab=3 bf =4 cf=4 (c) af =5 (c) **df =5** ae=6 (c) cd=6(c) de =8 (c)} Дерево побудовано

Мінімальне кістякове дерево побудовано. Його вартість становить:

$e^* = \sum_{i=0}^{|E_T|-1} e_i^* = 15$. І воно повністю збігається з мінімальним кістяковим деревом, побудованим за алгоритмом Прима. Але відрізняється за послідовністю ребер

Алгоритм Прима		Алгоритм Крускала	
	(a,b) = 3 (b,c) = 1 (b,f) = 4 (f,e) =2 (f,d) =5		{ bc =1 ef=2 ab=3 bf =4 df =5 }

Код алгоритму Крускала:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
#define edge pair<int,int>
class Graph {
private:
    vector<pair<int, edge>> G; // graph
    vector<pair<int, edge>> T; // mst
    int *parent;
    int V; // number of vertices/nodes in graph
public:
    Graph(int V);
    void AddWeightedEdge(int u, int v, int w);
    int find_set(int i);
    void union_set(int u, int v);
    void kruskal();
    void print();
};
Graph::Graph(int V) {
    parent = new int[V];
    //i 0 1 2 3 4 5
    //parent[i] 0 1 2 3 4 5
    for (int i = 0; i < V; i++)
        parent[i] = i;
    G.clear();
    T.clear();
}
void Graph::AddWeightedEdge(int u, int v, int w) {
    G.push_back(make_pair(w, edge(u, v)));
}
int Graph::find_set(int i) {
    // If i is the parent of itself
    if (i == parent[i])
        return i;
    else
        // Else if i is not the parent of itself
        // Then i is not the representative of his set,
        // so we recursively call Find on its parent
        return find_set(parent[i]);
}
void Graph::union_set(int u, int v) {
    parent[u] = parent[v];
}
```

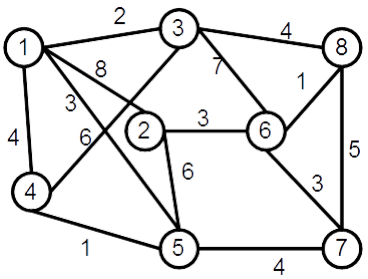
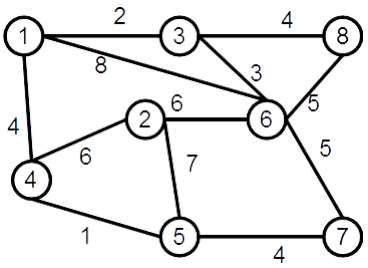
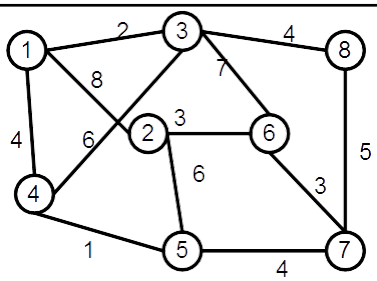
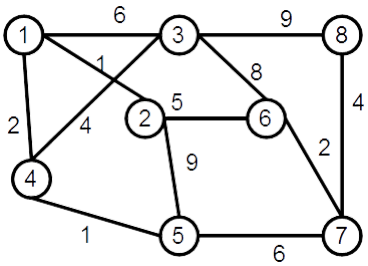
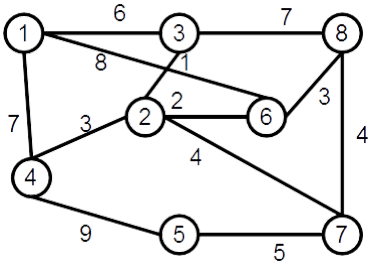
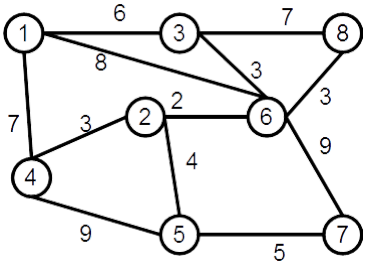
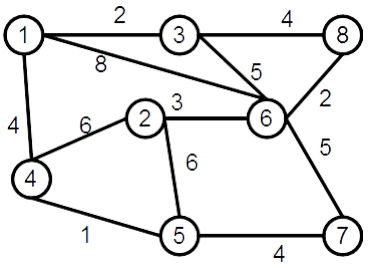
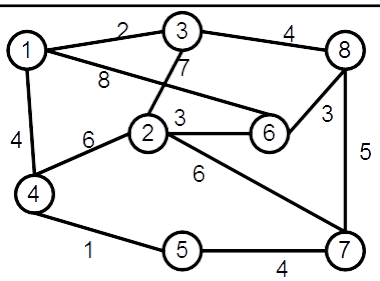
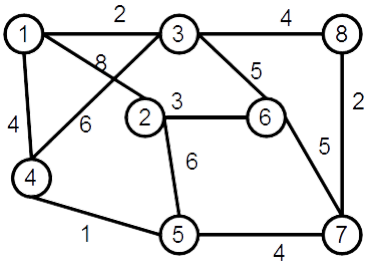
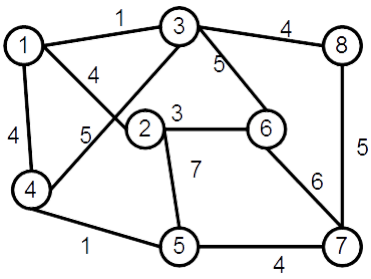
```
45. void Graph::kruskal() {
46.     int i, uRep, vRep;
47.     sort(G.begin(), G.end()); // increasing weight
48.     for (i = 0; i < G.size(); i++) {
49.         uRep = find_set(G[i].second.first);
50.         vRep = find_set(G[i].second.second);
51.         if (uRep != vRep) {
52.             T.push_back(G[i]); // add to tree
53.             union_set(uRep, vRep);
54.         }
55.     }
56. }
57. void Graph::print() {
58.     cout << "Edge :" << " Weight" << endl;
59.     for (int i = 0; i < T.size(); i++) {
60.         cout << T[i].second.first << " - " << T[i].second.second << " : "
61.                 << T[i].first;
62.         cout << endl;
63.     }
64. }
65. int main() {
66.     Graph g(6);
67.     g.AddWeightedEdge(0, 1, 3);
68.     g.AddWeightedEdge(0, 4, 6);
69.     g.AddWeightedEdge(0, 5, 5);
70.     g.AddWeightedEdge(1, 0, 3);
71.     g.AddWeightedEdge(1, 2, 1);
72.     g.AddWeightedEdge(1, 5, 4);
73.     g.AddWeightedEdge(2, 1, 1);
74.     g.AddWeightedEdge(2, 3, 6);
75.     g.AddWeightedEdge(2, 5, 4);
76.     g.AddWeightedEdge(3, 2, 6);
77.     g.AddWeightedEdge(3, 4, 8);
78.     g.AddWeightedEdge(3, 5, 5);
79.     g.AddWeightedEdge(4, 2, 4);
80.     g.AddWeightedEdge(4, 0, 6);
81.     g.AddWeightedEdge(4, 2, 8);
82.     g.AddWeightedEdge(4, 5, 2);
83.     g.AddWeightedEdge(5, 0, 5);
84.     g.AddWeightedEdge(5, 1, 4);
85.     g.AddWeightedEdge(5, 2, 4);
86.     g.AddWeightedEdge(5, 3, 5);
87.     g.AddWeightedEdge(5, 4, 2);
88.     g.kruskal();
89.     g.print();
90.     return 0;
91. }
```

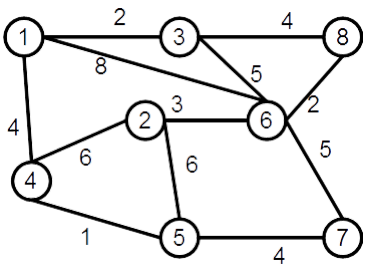
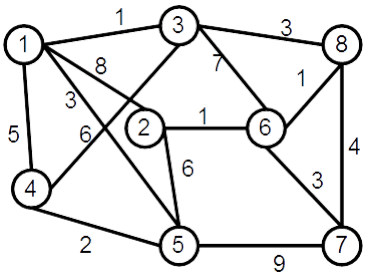
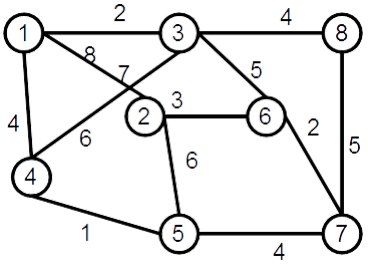
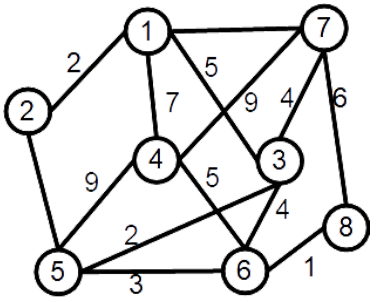
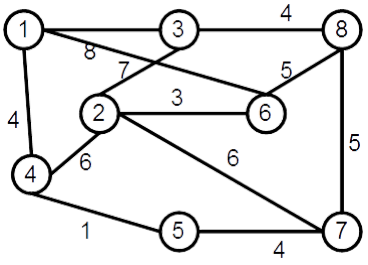
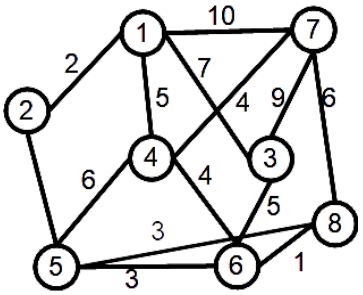
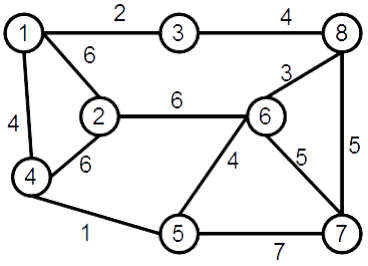
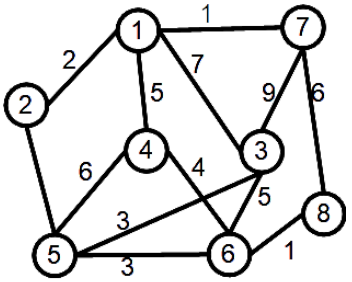
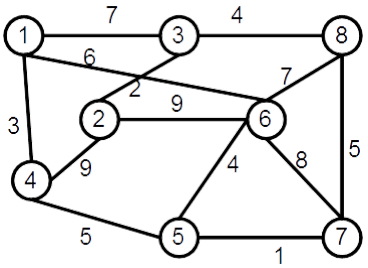
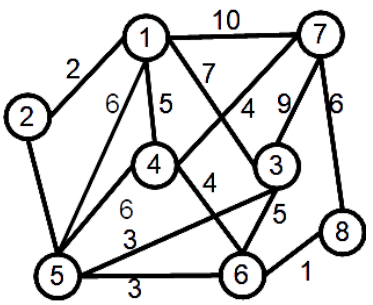
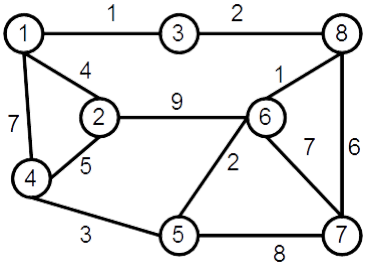
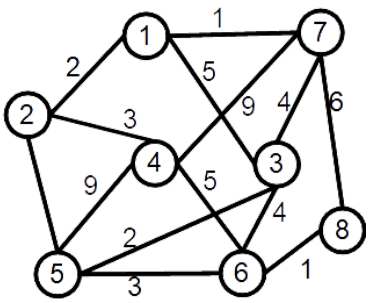
Завдання до лабораторної роботи №4

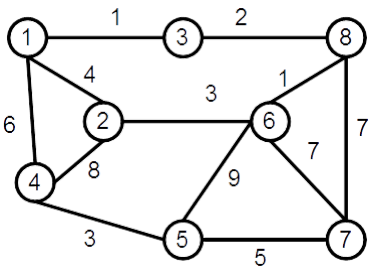
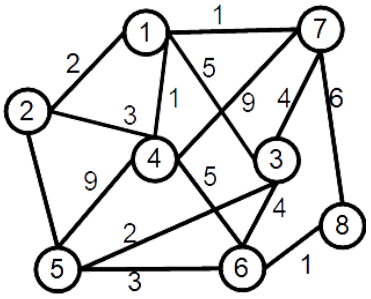
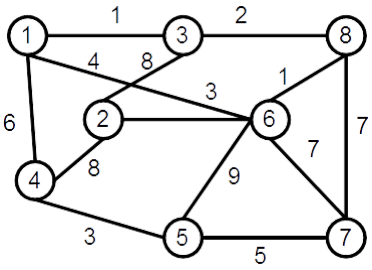
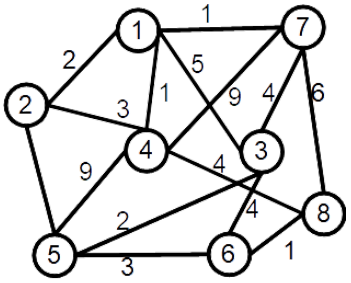
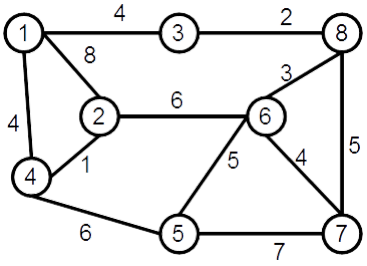
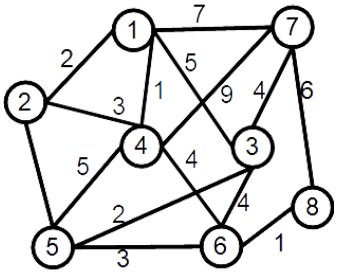
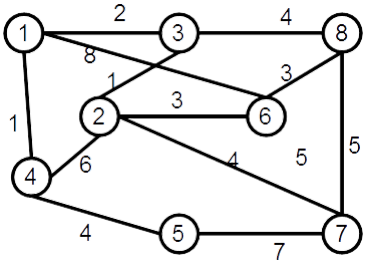
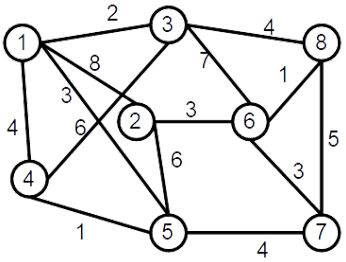
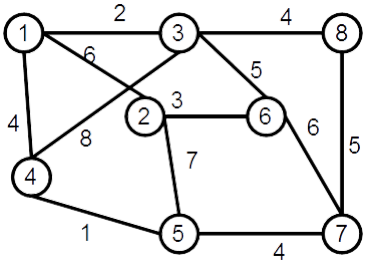
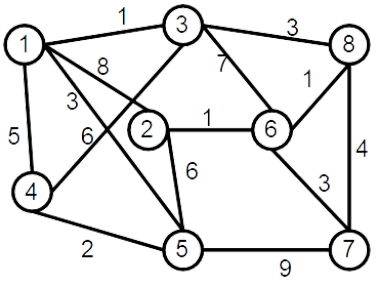
1. Побудувати матриці та списки суміжності для зважених неорієнтованих графів за варіантами завдань
2. Побудувати за допомогою алгоритму Прима мінімальне кістякове дерево (МКД) для графа за варіантами завдань та показати графічно хід побудови МКД за алгоритмом Прима (Таблиця1)
3. Запрограмувати алгоритм Прима. Запустити програму, показати результат за варіантами завдань. Показати, що результати ручної та автоматизованої побудови МКД співпадають.
4. Побудувати за допомогою алгоритму Крускала МКД для графа за варіантами завдань та показати графічно хід побудови МКД за алгоритмом Крускала (Таблиця 2).
5. Запрограмувати алгоритм Крускала. Запустити програму, показати результат за варіантами завдань. Показати, що результати ручної та автоматизованої побудови МКД співпадають.

6. Порівняти результати побудови МКД за двома алгоритмами, поясніти різницю, якщо вона буде
7. Звіт повинен містити:
− матриці суміжності та списки суміжності графів за варіантами завдань;
− хід побудови МКД за алгоритмами Пріма та Крускала,
− лістинги програм, результати виконання програм з контрольного прикладу за варіантами завдань,
− порівняння алгоритмів Пріма та Крускала в висновках щодо практичної роботи №4.

Варіанти завдань:

№	Граф	№	Граф
1		17	
2		18	
3		19	
4		20	
5		21	

№	Граф	№	Граф
6		22	
7		23	
8		24	
9		25	
10		26	
11		27	

№	Граф	№	Граф
12		28	
13		29	
14		30	
15		31	
16		32	

Лабораторна робота №5

Алгоритми на графах. Пошук найкоротшого шляху

Розглядаються алгоритми Дейкстри (Dijkstra) та Флойда (Floyd) для пошуку найкоротших шляхів між парами вершин (all-pairs shortest-paths problem)

Питання:

1. Задача пошуку найкоротшого шляху
2. Алгоритм Дейкстри
3. Алгоритм Флойда
4. Завдання
5. Оформлення звіту

1. Задача пошуку найкоротшого шляху.

Розглянемо задачу пошуку найкоротшого шляху з однієї вершини: *для даної вершини зваженого зв'язного графа, що називається вихідною (source), треба знайти найкоротший шлях до вихідної вершини (stock)*. Важливо підкреслити, що тут нас *не цікавить єдиний найкоротший шлях, що починається у вихідній вершині і проходить через всю решту вершин*. Це значно складніша задача (що являє собою версію задачі комівояжера). Завдання пошуку найкоротших шляхів з однієї вершини вимагає знайти сімейство шляхів, кожен з яких веде від *вхідної* вершини до будь-якої іншої вершини графа, хоча деякі шляхи, звичайно ж, можуть мати спільні ребра.

Задача про найкоротший шлях - одна з найпоширеніших в алгоритмах. Наприклад, побудова дорожнього маршруту – найбільш очевидна з усіх задач про найкоротший шлях, коли вам потрібно якнайшвидше дістатися з пункту відправлення до пункту призначення. Найкоротший шлях може позначатися як загальна відстань, яку вам потрібно покрити, або ж, як мінімально потрібний час, за який ви потрапити до вашого пункту призначення.

2. Алгоритм Дейкстри

Алгоритм Дейкстри знаходить найкоротші шляхи до вершин графа в порядку їх віддалення від заданої вхідної вершини. Спочатку він знаходить найкоротший шлях від вхідної вершини до *першої* найближчої, потім до *другої* найближчої і т.д. У загальному випадку перед початком *i*-ої ітерації алгоритм визначає найкоротші шляхи до *i*-1 інших вершин, найближчих до вихідної. Ці вершини, вхідна вершина та ребра найкоротших шляхів, що ведуть до них з вхідної вершини утворюють піддерево T_i даного графу. Оскільки ваги всіх ребер мають *позитивне* значення (неотрицательное), чергова найближча до вхідної вершина може бути знайдена серед вершин, суміжних з піддеревом T_i. Множину вершин, суміжних з вершинами T_i можна назвати "примежовими" (з відомими вагами); саме з них алгоритм Дейкстри вибирає чергову вершину, найближчу (за вагою) до вхідної. При цьому всі

інші вершини розглядаються як примежові, з'єднані з вершинами дерева ребрами з *нескінченими* вагами. Для визначення *i*-ої найближчої вершини алгоритм обчислює для кожної примежової вершини *u* суму відстані до найближчої вершини дерева *v* (визначеної вагою ребра *w(u,v)*) та довжини d_v найкоротшого шляху з вхідної вершини у вершину *v* (раніше визначену алгоритмом), а потім вибирає вершину з найменшою сумою. Головним в алгоритмі Дейкстри є те, що достатньо порівняти довжини таких спеціальних шляхів.

Для спрощення роботи алгоритму позначаємо кожну вершину двома мітками. Числова мітка *d* вказує довжину найкоротшого шляху до цієї вершини від вихідної, визначену алгоритмом. Перед початком алгоритму ці мітки набувають значення – «∞». Друга мітка показує ім'я попередньої вершини на такому шляху, тобто батьківський вузол дерева, яке будується. Ця мітка залишається *порожньою* «-» у вихідної вершини та вершин, які не є суміжними з жодною вершиною поточного дерева. При використанні таких міток пошук наступної найближчої вершини *u** стає простим завданням пошуку примежової вершини з найменшим значенням *d*. Неоднозначності розв'язуються довільним чином.

Визначивши вершину, що додається в дерево *u**, ми повинні виконати дві операції:

1. Перемістити вершину *u* * із множини примежових у множину вершин дерева.
2. Для кожної вершини *u*, що залишається, зв'язаної з вершиною *u* * ребром з вагою *w* (*u**, *u*) такою, що $d_{u^*} + w(u^*, u) < d_u$, обновити мітки *u*, замінивши їх на *u* * та $d_{u^*} + w(u^*, u)$, відповідно.

Операція використання міток та використання черги алгоритму Дейкстри схожі з алгоритмом Прима. В обох випадках будується і розширюється піддерево вершин шляхом вибору чергової вершини з черги з пріоритетами, що містить вершини, що залишаються поза деревом. Однак алгоритми призначені для вирішення різних задач та працюють з пріоритетами, які обчислюються різними способами: *алгоритм Дейкстри* порівнює *довжини шляхів* і повинен *підсумовути ваги ребер*, у той час як *алгоритм Прима* порівнює *ваги ребер* такими як вони є. Наведемо псевдокод алгоритму Дейкстри:

Dijkstra(*G*, *s*)
// Алгоритм Дейкстри для пошуку найкоротших шляхів з однієї вершини
// Вхідні дані: Зважений зв'язний граф $G = (V, E)$ та його вершина *s*
// Вихідні дані: довжина d_v найкоротшого шляху від *s* до *v*
// і передостання вершина p_v на цьому шляху для всіх вершин $v \in V$
Initialize(Q) // Ініціалізація порожньої черги
for (Для) кожної вершини $v \in V$ **do**
$d_v \leftarrow \infty$; $p_v \leftarrow$ NULL
Insert(Q,v,d_v) // Ініціалізація пріоритету вершини у черзі Q
$d_s \leftarrow 0$; // Присвоювання нульової відстані початковому вузлу *s*
Decrease(Q,s,d_s) // Оновлення пріоритету *s* значенням d_s
$V_T \leftarrow \emptyset$

for i ← 0 to |*V*| - 1 **do**

*u** ← *DeleteMin*(*Q*) // Вибір вузла з мінімальним пріоритетом, присвоювання його *u** та видалення з його черги

$V_T \leftarrow V_T \cup \{u^*\}$ // Поміщення вузла *u** з мінімальним пріоритетом у множину вершин дерева

for (Для) кожної вершини *u* з $V - V_T$, суміжної з *u**do**

if $d_{u^*} + w(u^*, u) < d_u$

$d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$

Decrease(*Q*, *u* d_u) // Оновлення пріоритету *u* значенням d_u

Для порівняння за алгоритмом Прима

for i ← 1 to |V| - 1 **do**

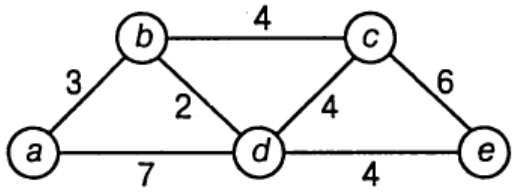
Пошук ребра з мінімальною вагою *e** = (*v**,*u**) серед усіх ребер (*v*, *u*) таких, що $v \in V_T$ та u ∈ $V - V_T$

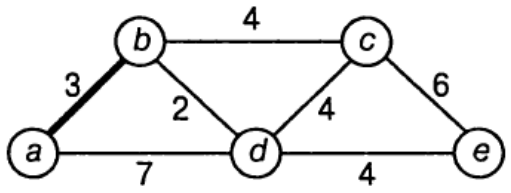
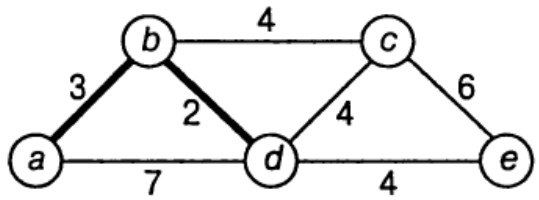
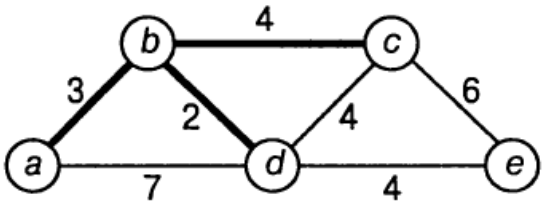
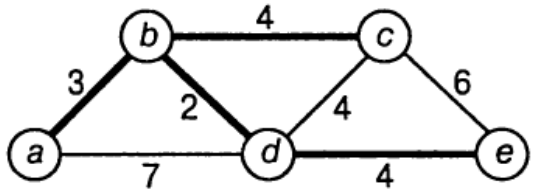
$V_T \leftarrow V_T \cup \{u^*\}$

$\boldsymbol{E_T \leftarrow E_T \cup \{e^*\}}$

Дамо наступні пояснення. Псевдокод алгоритму Дейкстри описує роботу алгоритму з використанням явної вказівки операцій над двома множинами вершин з мітками: множини V_T вершин, для яких вже знайдені найкоротші шляхи, та черги з пріоритетами Q, що містить примежові вершини. (Зауважимо, що у наведеному псевдокоді V_T містить задану вихідну вершину, а черга - суміжні з нею вершини після виконання ітерації 0.) У таблиці 1 показаний хід знаходження найкоротшого шляху у графі за алгоритмом Дейкстри.

Таблиця 1

Зображення	Вершини дерева V_T *d, p*	Інші вершини $V - V_T$
		*{a,b,c,d,e}* $d_v \leftarrow \infty$; $p_v \leftarrow$ NULL a(-,∞) b(-,∞) c(-,∞) d(-,∞) e(-,∞) мітки приймають значення – «∞»
	{a} $d_s \leftarrow 0$ a(-,0) мітка залишається *порожньою* «-» *DeleteMin*(*Q*)	*{b,c,d,e}* *Decrease*(*Q*,*s*,d_s) **b(a,3)** + c(-,∞) **d(a,7)** e(-,∞)

Зображення	Вершини дерева $V_T d, p$	Інші вершини $V - V_T$
	*{a,b}* b(a,3)	*{c,d,e}* *Decrease*(Q,s,d_s) c(b,3+4) c(b,7) **d(b,3+2)** + **d(b,5)** e(-,∞)
	*{a,b,d}* d(b,5)	*{c,e}* *Decrease*(Q,s,d_s) **c(b,7)** ~~e(d, 5+4)~~ e(d,5+4)
	*{a,b,d,c}* c(b,7)	*{e}* e(d,9) ~~e(c,7+6)~~
	*{a,b,d,c,e}* e(d,9)	*{}*

Перший по черзі сусід вершини a - вершина b, тому що довжина шляху до неї мінімальна (3<7). Довжина шляху в неї через вершину a дорівнює сумі значення мітки вершини a і довжини ребра, що йде з a в b, тобто 0 + 3 = 3. Це менше поточної мітки вершини b, нескінченності, тому нова мітка вершини b дорівнює 7.

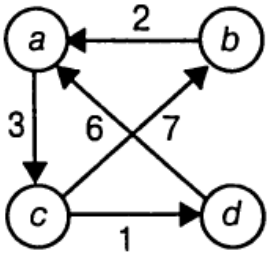
У таблиці 2 показані довжини найкоротших відстаней до решти вершин графа з вершини a

Таблиця 2

Найкоротші шляхи			
З Вершини	В Вершину	Шлях	Вага
a	b	a→b	3
	c	a→b→c	7
	d	a→b→d	5
	e	a→b→d→e	9

3. Алгоритм Флойда

Як ми вже знаємо задача пошуку найкоротших шляхів між усіма парами вершин (all-pairs shortest-paths problem) полягає у пошуку для даного орієнтованого або неорієнтованого зваженого зв'язного графа відстаней від кожної вершини до всіх інших вершин. Для запису довжин найкоротших шляхів зручно скористатися матрицею D розміром n×n, яка називається матрицею відстаней (*distance matrix*): елемент d_{ij} на перетині i-ого рядка и j-го стовпця такої матриці вказує довжину найкоротшого шляху від i-ої вершини до j-ої вершини ($1 \le i,j \le n$) (рис. 1).



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

а б в

Рис. 1 Орієнтований граф (а), його матриця суміжності (б) і матриця відстаней (в)

Для побудови матриці відстаней розроблено алгоритм Флойда (Floyd's algorithm) (Флойда-Уоршелла). За допомогою алгоритму Флойда обчислюється матриця відстаней зваженого графа з n вершинами за допомогою побудови послідовності матриць:

$$D^{(0)}, D^{(1)}, \ldots D^{(k)}, \ldots D^{(n)} \qquad (1),$$

де елемент $d_{ij}^{(k)}$ на перетині i-ого рядка та j-го стовпця матриці $D^{(k)}$, k= 0,1,..., n дорівнює довжині найкоротшого шляху серед всіх шляхів від i-ої вершини до j-ої, у яких проміжні вершини не можуть мати номери, що перевищують k. Зокрема, послідовність починається з матриці $D^{(0)}$ у якій у шляхах не може бути проміжних вершин, $D^{(0)}$ являє собою просто вихідну вагову матрицю графу W. Остання матриця послідовності, $D^{(n)}$ містить довжини найкоротших шляхів серед усіх шляхів, в яких як проміжні вершини можуть бути будь-які з n вершин графу, так що матриця $D^{(n)}$ і є матриця відстаней графу, яку ми обчислюємо. При цьому ми обчислюємо всі елементи кожної матриці $D^{(k)}$ на основі інформації про елементи попередньої матриці $D^{(k-1)}$ у послідовності (1). Якщо $d_{ij}^{(k)}$ - елемент на перетині i-ого рядка та j-го стовпця матриці $D^{(k)}$, то $d_{ij}^{(k)}$ - дорівнює довжині найкоротшого шляху серед всіх шляхів від i-ої вершини v_i до j-ої вершини v_j проміжні вершини яких мають номери не вище k. Таким чином, всі такі шляхи можна розбити на дві непересічні підмножини: ті шляхи, в яких як проміжна не бере участі k-а вершина v_k, та ті, у яких вона є однією із проміжних.

Оскільки шляхи в першій підмножині містять проміжні вершини з номерами не вище k - 1, то найкоротший шлях між цими вершинами, за визначенням наших матриць, має довжину $d_{ij}^{(k-1)}$.

У другій підмножині ми можемо обмежити розгляд лише тих вершин, у яких вершина v_k входить лише один раз. Тобто кожний з таких шляхів складається з шляху від v_i до v_k, причому всі проміжні вершини мають номери

не вище k - 1, і шляхи від v_k до v_j, у якого всі проміжні вершини також мають номери не вище k - 1.

Оскільки довжина найкоротшого шляху від v_i до v_k серед усіх шляхів, які не використовують проміжних вершин з номерами більше k - 1, дорівнює

$d_{ik}^{(k-1)}$, а довжина найкоротшого шляху від v_k до v_j серед усіх шляхів, які не використовують проміжних вершин з номерами більше k - 1, дорівнює

$d_{kj}^{(k-1)}$, довжина найкоротшого шляху від v_i до v_j через вершину v_k дорівнює $d_{ik}^{(k-1)}+ d_{kj}^{(k-1)}$. З урахуванням довжини найкоротших шляхів в обох підмножинах отримуємо наступне рекурентне співвідношення:

$$d_{ij}^{(k)}=\min\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+ d_{kj}^{(k-1)}\} \text{ для } k\geq 1 \text{ , } d_{ij}^{(0)}= w_{ij} \qquad (2)$$

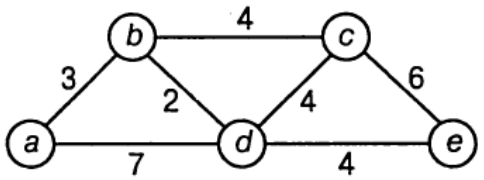
Тобто елемент на перетині i-ого рядка та j-го стовпця поточної матриці відстаней $D^{(k-1)}$ замінюється сумою елементів на перетині того ж i-ого рядка та k-го стовпця, та k-ого рядка та того ж j-го стовпця тоді і тільки тоді, коли ця сума виявляється меншою за поточне значення.

Псевдокод алгоритму Флойда з урахуванням того, що кожна наступна матриця в послідовності (1) може бути записана поверх попередньої має наступний вигляд:

Алгоритм *Floyd* (W [1..n, 1..n])
// Вхідні дані: Вагова матриця W графу
// Вихідні дані: Матриця довжин найкоротших шляхів D
$D \leftarrow W$ // Не вимагається, якщо W може бути перезаписана
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $D[i,j] \leftarrow$ **min**$\{D[i,j], D[i,k]+D[k,j]\}$
return D

У таблиці 3 показаний хід знаходження найкоротшого шляху у графі алгоритму Флойда $D[i,j] \leftarrow$ **min**$\{ D[i,k]+D[k,j], D[i,j]\}$

Таблиця 3



i=1 j=1-5 (от 1 до 5)
$d_{1,1}^{1}=\min\{d_{1,1}+d_{1,1;}\ d_{1,1}\}=\min\{0+0;\ 0\}=0$
$d_{1,2}^{1}=\min\{d_{1,1}+d_{1,2;}\ d_{1,2}\}=\min\{0+3;\ 3\}=3$
$d_{1,3}^{1}=\min\{d_{1,1}+d_{1,3;}\ d_{1,3}\}=\min\{0+\infty;\ \infty\}=\infty$
$d_{1,4}^{1}=\min\{d_{1,1}+d_{1,4;}\ d_{1,4}\}=\min\{0+7;\ 7\}=7$
$d_{1,5}^{1}=\min\{d_{1,1}+d_{1,5;}\ d_{1,5}\}=\min\{0+\infty;\ \infty\}=\infty$
i=2 j=1-5
$d_{2,1}^{1}=\min\{d_{2,1}+d_{1,1;}\ d_{2,1}\}=\min\{3+0;\ 3\}=3$
$d_{2,2}^{1}=\min\{d_{2,1}+d_{1,2;}\ d_{2,2}\}=\min\{3+3;\ 0\}=0$
$d_{2,3}^{1}=\min\{d_{2,1}+d_{1,3;}\ d_{2,3}\}=\min\{3+\infty;\ 4\}=4$
$d_{2,4}^{1}=\min\{d_{2,1}+d_{1,4;}\ d_{2,4}\}=\min\{3+7;\ 2\}=2$

		j				
		1(a)	2(b)	3(c)	4(d)	5(e)
	1(a)	0	3	∞	7	∞
	2(b)	3	0	4	2	∞
i	3(c)	∞	4	0	4	6
	4(d)	7	2	4	0	4
	5(e)	∞	∞	6	4	0

k=1 $D^{(1)}$

$d_{i,j}^{1}=\min\{ d_{i,1}+ d_{1,j};\ d_{i,j}\}$.

$d_{2,5}^{1}=\min\{d_{2,1}+d_{1,5;}\, d_{2,5}\}=\min\{3+\infty;\ \infty\}=\infty$

i=3 j=1-5

$d_{3,1}^{1}=\min\{d_{3,1}+d_{1,1;}\, d_{3,1}\}=\min\{\infty+0;\ \infty\}=\infty$

$d_{3,2}^{1}=\min\{d_{3,1}+d_{1,2;}\, d_{3,2}\}=\min\{\infty+3;\ 4\}=4$

$d_{3,3}^{1}=\min\{d_{3,1}+d_{1,3;}\, d_{3,3}\}=\min\{\infty+\infty;\ 0\}=0$

$d_{3,4}^{1}=\min\{d_{3,1}+d_{1,4;}\, d_{3,4}\}=\min\{\infty+7;\ 4\}=4$

$d_{3,5}^{1}=\min\{d_{3,1}+d_{1,5;}\, d_{3,5}\}=\min\{\infty+\infty;\ 6\}=6$

i=4 j=1-5

$d_{4,1}^{1}=\min\{d_{4,1}+d_{1,1;}\, d_{4,1}\}=\min\{7+0;\ 7\}=7$

$d_{4,2}^{1}=\min\{d_{4,1}+d_{1,2;}\, d_{4,2}\}=\min\{7+3;\ 2\}=2$

$d_{4,3}^{1}=\min\{d_{4,1}+d_{1,3;}\, d_{4,3}\}=\min\{7+\infty;\ 4\}=4$

$d_{4,4}^{1}=\min\{d_{4,1}+d_{1,4;}\, d_{4,4}\}=\min\{7+7;\ 0\}=0$

$d_{4,5}^{1}=\min\{d_{4,1}+d_{1,5;}\, d_{4,5}\}=\min\{7+\infty;\ 4\}=4$

i=5 j=1-5

$d_{5,1}^{1}=\min\{d_{5,1}+d_{1,1;}\, d_{5,1}\}=\min\{\infty+0;\ \infty\}=\infty$

$d_{5,2}^{1}=\min\{d_{5,1}+d_{1,2;}\, d_{5,2}\}=\min\{\infty+3;\ \infty\}=\infty$

$d_{5,3}^{1}=\min\{d_{5,1}+d_{1,3;}\, d_{5,3}\}=\min\{\infty+\infty;\ 6\}=6$

$d_{5,4}^{1}=\min\{d_{5,1}+d_{1,4;}\, d_{5,4}\}=\min\{\infty+7;\ 4\}=4$

$d_{5,5}^{1}=\min\{d_{5,1}+d_{1,5;}\, d_{5,5}\}=\min\{\infty+\infty;\ 0\}=0$

	1(a)	2(b)	3(c)	4(d)	5(e)
1(a)	0	3	∞	7	∞
2(b)	3	0	4	2	∞
3(c)	∞	4	0	4	6
4(d)	7	2	4	0	4
5(e)	∞	∞	6	4	0

k=2 $D^{(2)}$

$d_{i,j}^{2}=\min\{\, d_{i,2}^{1}+ d_{2,j}^{1};\ d_{i,j}^{1}\}.$

i=1 j=1-5

$d_{1,1}^{2}=\min\{d_{1,2}^{1}+d_{2,1}^{1}\, d_{1,1}^{1}\}=\min\{3+3;\ 0\}=0$

$d_{1,2}^{2}=\min\{d_{1,2}^{1}+d_{2,2}^{1}\, d_{1,2}^{1}\}=\min\{3+0;\ 3\}=3$

$d_{1,3}^{2}=\min\{d_{1,2}^{1}+d_{2,3}^{1}\, d_{1,3}^{1}\}=\min\{3+4;\ \infty\}=7$

$d_{1,4}^{2}=\min\{d_{1,2}^{1}+d_{2,4}^{1}\, d_{1,4}^{1}\}=\min\{3+2;\ 7\}=5$

$d_{1,5}^{2}=\min\{d_{1,2}^{1}+d_{2,5}^{1}\, d_{1,5}^{1}\}=\min\{3+\infty;\ \infty\}=\infty$

i=2 j=1-5

$d_{2,1}^{2}=\min\{d_{2,2}^{1}+d_{2,1}^{1}\, d_{2,1}^{1}\}=\min\{0+3;\ 3\}=3$

$d_{2,2}^{2}=\min\{d_{2,2}^{1}+d_{2,2}^{1}\, d_{2,2}^{1}\}=\min\{0+0;\ 0\}=0$

$d_{2,3}^{2}=\min\{d_{2,2}^{1}+d_{2,3}^{1}\, d_{2,3}^{1}\}=\min\{0+4;\ 4\}=4$

$d_{2,4}^{2}=\min\{d_{2,2}^{1}+d_{2,4}^{1}\, d_{2,4}^{1}\}=\min\{0+2;\ 2\}=2$

$d_{2,5}^{2}=\min\{d_{2,2}^{1}+d_{2,5}^{1}\, d_{2,5}^{1}\}=\min\{0+\infty;\ \infty\}=\infty$

i=3 j=1-5

$d_{3,1}^{2}=\min\{d_{3,2}^{1}+d_{2,1}^{1} d_{3,1}^{1}\}=\min\{4+3;\ \infty\}=7$

$d_{3,2}^{2}=\min\{d_{3,2}^{1}+d_{2,2}^{1} d_{3,2}^{1}\}=\min\{4+0;\ 4\}=4$

$d_{3,3}^{2}=\min\{d_{3,2}^{1}+d_{2,3}^{1} d_{3,3}^{1}\}=\min\{4+4;\ 0\}=0$

$d_{3,4}^{2}=\min\{d_{3,2}^{1}+d_{2,4}^{1} d_{3,4}^{1}\}=\min\{4+2;\ 4\}=4$

$d_{3,5}^{2}=\min\{d_{3,2}^{1}+d_{2,5}^{1} d_{3,5}^{1}\}=\min\{4+\infty;\ 6\}=6$

i=4 j=1-5

$d_{4,1}^{2}=\min\{d_{4,2}^{1}+d_{2,1}^{1}\, d_{4,1}^{1}\}=\min\{2+3;\ 7\}=5$

$d_{4,2}^{2}=\min\{d_{4,2}^{1}+d_{2,2}^{1}\, d_{4,2}^{1}\}=\min\{2+0;\ 2\}=2$

$d_{4,3}^{2}=\min\{d_{4,2}^{1}+d_{2,3}^{1}\, d_{4,3}^{1}\}=\min\{2+4;\ 4\}=4$

$d_{4,4}^{2}=\min\{d_{4,2}^{1}+d_{2,4}^{1}\, d_{4,4}^{1}\}=\min\{2+2;\ 0\}=0$

$d_{4,5}^{2}=\min\{d_{4,2}^{1}+d_{2,5}^{1}\, d_{4,5}^{1}\}=\min\{2+\infty;\ 4\}=4$

i=5 j=1-5

$d_{5,1}^{2}=\min\{d_{5,2}^{1}+d_{2,1}^{1}\, d_{5,1}^{1}\}=\min\{\infty+3;\ \infty\}=\infty$

$d_{5,2}^{2}=\min\{d_{5,2}^{1}+d_{2,2}^{1}\, d_{5,2}^{1}\}=\min\{\infty+0;\ \infty\}=\infty$

$d_{5,3}^{2}=\min\{d_{5,2}^{1}+d_{2,3}^{1}\, d_{5,3}^{1}\}=\min\{\infty+4;\ 6\}=6$

$d_{5,4}^{2}=\min\{d_{5,2}^{1}+d_{2,4}^{1}\, d_{5,4}^{1}\}=\min\{\infty+2;\ 4\}=4$

$d_{5,5}^{2}=\min\{d_{5,2}^{1}+d_{2,5}^{1}\, d_{5,5}^{1}\}=\min\{\infty+\infty;\ 0\}=0$

	1(a)	2(b)	3(c)	4(d)	5(e)
1(a)	0	3	**7**	**5**	∞
2(b)	3	0	4	2	∞
3(c)	**7**	4	0	4	6
4(d)	**5**	2	4	0	4
5(e)	∞	∞	6	4	0

k=3 $D^{(3)}$

$d_{i,j}^{3}=\min\{\, d_{i,3}^{2}+ d_{3,j}^{2};\ d_{i,j}^{2}\}.$

i=1 j=1-5

$d_{1,1}^{3}=\min\{d_{1,3}^{2}+d_{3,1}^{2}\, d_{1,1}^{2}\}=\min\{7+7;\ 0\}=0$

$d_{1,2}^{3}=\min\{d_{1,3}^{2}+d_{3,2}^{2}\, d_{1,2}^{2}\}=\min\{7+4;\ 3\}=3$

$d_{1,3}^3=\min\{d_{1,3}^2+d_{3,3}^2\, d_{1,3}^2\}=\min\{7+0;\ 7\}=7$
$d_{1,4}^3=\min\{d_{1,3}^2+d_{3,4}^2\, d_{1,4}^2\}=\min\{7+4;\ 5\}=5$
$d_{1,5}^3=\min\{d_{1,3}^2+d_{3,5}^2\, d_{1,5}^2\}=\min\{7+6;\ \infty\}=13$
i=2 j=1-5
$d_{2,1}^3=\min\{d_{2,3}^2+d_{3,1}^2\, d_{2,1}^2\}=\min\{4+7;\ 3\}=3$
$d_{2,2}^3=\min\{d_{2,3}^2+d_{3,2}^2\, d_{2,2}^2\}=\min\{4+4;\ 0\}=0$
$d_{2,3}^3=\min\{d_{2,3}^2+d_{3,3}^2\, d_{2,3}^2\}=\min\{4+0;\ 4\}=4$
$d_{2,4}^3=\min\{d_{2,3}^2+d_{3,4}^2\, d_{2,4}^2\}=\min\{4+4;\ 2\}=2$
$d_{2,5}^3=\min\{d_{2,3}^2+d_{3,5}^2\, d_{2,5}^2\}=\min\{4+6;\ \infty\}=10$
i=3 j=1-5
$d_{3,1}^3=\min\{d_{3,3}^2+d_{3,1}^2\, d_{3,1}^2\}=\min\{0+7;\ 7\}=7$
$d_{3,2}^3=\min\{d_{3,3}^2+d_{3,2}^2\, d_{3,2}^2\}=\min\{0+4;\ 4\}=4$
$d_{3,3}^3=\min\{d_{3,3}^2+d_{3,3}^2\, d_{3,3}^2\}=\min\{0+0;\ 0\}=0$
$d_{3,4}^3=\min\{d_{3,3}^2+d_{3,4}^2\, d_{3,4}^2\}=\min\{0+4;\ 4\}=4$
$d_{3,5}^3=\min\{d_{3,3}^2+d_{3,5}^2\, d_{3,5}^2\}=\min\{0+6;\ 6\}=6$
i=4 j=1-5
$d_{4,1}^3=\min\{d_{4,3}^2+d_{3,1}^2\, d_{4,1}^2\}=\min\{4+7;\ 5\}=5$
$d_{4,2}^3=\min\{d_{4,3}^2+d_{3,2}^2\, d_{4,2}^2\}=\min\{4+4;\ 2\}=2$
$d_{4,3}^3=\min\{d_{4,3}^2+d_{3,3}^2\, d_{4,3}^2\}=\min\{4+0;\ 4\}=4$
$d_{4,4}^3=\min\{d_{4,3}^2+d_{3,4}^2\, d_{4,4}^2\}=\min\{4+4;\ 0\}=0$
$d_{4,5}^3=\min\{d_{4,3}^2+d_{3,5}^2\, d_{4,5}^2\}=\min\{4+6;\ 4\}=4$

i=5 j=1-5
$d_{5,1}^3=\min\{d_{5,3}^2+d_{3,1}^2 d_{5,1}^2\}=\min\{6+7;\ \infty\}=13$
$d_{5,2}^3=\min\{d_{5,3}^2+d_{3,2}^2 d_{5,2}^2\}=\min\{6+4;\ \infty\}=10$
$d_{5,3}^3=\min\{d_{5,3}^2+d_{3,3}^2 d_{5,3}^2\}=\min\{6+0;\ 6\}=6$
$d_{5,4}^3=\min\{d_{5,3}^2+d_{3,4}^2 d_{5,4}^2\}=\min\{6+4;\ 4\}=4$
$d_{5,5}^3=\min\{d_{5,3}^2+d_{3,5}^2 d_{5,5}^2\}=\min\{6+6;\ 0\}=0$

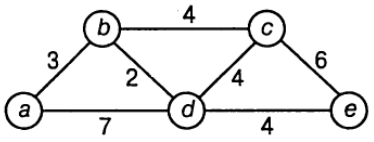
	1(a)	2(b)	3(c)	4(d)	5(e)	**k=4 $D^{(4)}$**
1(a)	0	3	7	5	**13**	
2(b)	3	0	4	2	**10**	
3(c)	7	4	0	4	6	
4(d)	5	2	4	0	4	
5(e)	**13**	**10**	6	4	0	

$d_{i,j}^4=\min\{\ d_{i,4}^3+ d_{4,j}^3;\ d_{i,j}^3\}$.

i=1 j=1-5
$d_{1,1}^4=\min\{d_{1,4}^3+d_{4,1}^3\, d_{1,1}^3\}=\min\{5+5;\ 0\}=0$
$d_{1,2}^4=\min\{d_{1,4}^3+d_{4,2}^3\, d_{1,2}^3\}=\min\{5+2;\ 3\}=3$
$d_{1,3}^4=\min\{d_{1,4}^3+d_{4,3}^3\, d_{1,3}^3\}=\min\{5+4;\ 7\}=7$
$d_{1,4}^4=\min\{d_{1,4}^3+d_{4,4}^3\, d_{1,4}^3\}=\min\{5+0;\ 5\}=5$
$d_{1,5}^4=\min\{d_{1,4}^3+d_{4,5}^3\, d_{1,5}^3\}=\min\{5+4;\ 13\}=9$
i=2 j=1-5
$d_{2,1}^4=\min\{d_{2,4}^3+d_{4,1}^3\, d_{2,1}^3\}=\min\{2+5;\ 3\}=3$
$d_{2,2}^4=\min\{d_{2,4}^3+d_{4,2}^3\, d_{2,2}^3\}=\min\{2+2;\ 0\}=0$
$d_{2,3}^4=\min\{d_{2,4}^3+d_{4,3}^3\, d_{2,3}^3\}=\min\{2+4;\ 4\}=4$
$d_{2,4}^4=\min\{d_{2,4}^3+d_{4,4}^3\, d_{2,4}^3\}=\min\{2+0;\ 2\}=2$
$d_{2,5}^4=\min\{d_{2,4}^3+d_{4,5}^3\, d_{2,5}^3\}=\min\{2+4;\ 10\}=6$
i=3 j=1-5
$d_{3,1}^4=\min\{d_{3,4}^3+d_{4,1}^3 d_{3,1}^3\}=\min\{4+5;\ 7\}=7$
$d_{3,2}^4=\min\{d_{3,4}^3+d_{4,2}^3 d_{3,2}^3\}=\min\{4+2;\ 4\}=4$
$d_{3,3}^4=\min\{d_{3,4}^3+d_{4,3}^3 d_{3,3}^3\}=\min\{4+4;\ 0\}=0$
$d_{3,4}^4=\min\{d_{3,4}^3+d_{4,4}^3 d_{3,4}^3\}=\min\{4+0;\ 4\}=4$
$d_{3,5}^4=\min\{d_{3,4}^3+d_{4,5}^3 d_{3,5}^3\}=\min\{4+4;\ 6\}=6$
i=4 j=1-5
$d_{4,1}^4=\min\{d_{4,4}^3+d_{4,1}^3\, d_{4,1}^3\}=\min\{0+5;\ 5\}=5$
$d_{4,2}^4=\min\{d_{4,4}^3+d_{4,2}^3\, d_{4,2}^3\}=\min\{0+2;\ 2\}=2$
$d_{4,3}^4=\min\{d_{4,4}^3+d_{4,3}^3\, d_{4,3}^3\}=\min\{0+4;\ 4\}=4$
$d_{4,4}^4=\min\{d_{4,4}^3+d_{4,4}^3\, d_{4,4}^3\}=\min\{0+0;\ 0\}=0$

$d_{4,5}^4=\min\{d_{4,4}^3+d_{4,5}^3 d_{4,5}^3\}=\min\{0+4; 4\}=4$ i=5 j=1-5 **$d_{5,1}^4=\min\{d_{5,4}^3+d_{4,1}^3 d_{5,1}^3\}=\min\{4+5; 13\}=9$** **$d_{5,2}^4=\min\{d_{5,4}^3+d_{4,2}^3 d_{5,2}^3\}=\min\{4+2; 10\}=6$** $d_{5,3}^4=\min\{d_{5,4}^3+d_{4,3}^3 d_{5,3}^3\}=\min\{4+4; 6\}=6$ $d_{5,4}^4=\min\{d_{5,4}^3+d_{4,4}^3 d_{5,4}^3\}=\min\{4+0; 4\}=4$ $d_{5,5}^4=\min\{d_{5,4}^3+d_{4,5}^3 d_{5,5}^3\}=\min\{4+4; 0\}=0$	1(a) 2(b) 3(c) 4(d) 5(e) **k=5 $D^{(5)}$** 1(a) 0 3 7 5 **9** 2(b) 3 0 4 2 **6** 3(c) 7 4 0 4 6 4(d) 5 2 4 0 4 5(e) **9** **6** 6 4 0 **$d_{i,j}^5=\min\{ d_{i,5}^4+ d_{5,j}^4; d_{i,j}^4\}$**
i=1 j=1-5 $d_{1,1}^5=\min\{d_{1,5}^4+d_{5,1}^4 d_{1,1}^4\}=\min\{9+9; 0\}=0$ $d_{1,2}^5=\min\{d_{1,5}^4+d_{5,2}^4 d_{1,2}^4\}=\min\{9+6; 3\}=3$ $d_{1,3}^5=\min\{d_{1,5}^4+d_{5,3}^4 d_{1,3}^4\}=\min\{9+6; 7\}=7$ $d_{1,4}^5=\min\{d_{1,5}^4+d_{5,4}^4 d_{1,4}^4\}=\min\{9+4; 5\}=5$ $d_{1,5}^5=\min\{d_{1,5}^4+d_{5,5}^4 d_{1,5}^4\}=\min\{9+0; 9\}=9$ i=2 j=1-5 $d_{2,1}^5=\min\{d_{2,5}^4+d_{5,1}^4 d_{2,1}^4\}=\min\{6+9; 3\}=3$ $d_{2,2}^5=\min\{d_{2,5}^4+d_{5,2}^4 d_{2,2}^4\}=\min\{6+6; 0\}=0$ $d_{2,3}^5=\min\{d_{2,5}^4+d_{5,3}^4 d_{2,3}^4\}=\min\{6+6; 4\}=4$ $d_{2,4}^5=\min\{d_{2,5}^4+d_{5,4}^4 d_{2,4}^4\}=\min\{6+4; 2\}=2$ $d_{2,5}^5=\min\{d_{2,5}^4+d_{5,5}^4 d_{2,5}^4\}=\min\{6+0; 6\}=6$ i=3 j=1-5 $d_{3,1}^5=\min\{d_{3,5}^4+d_{5,1}^4 d_{3,1}^4\}=\min\{6+9; 7\}=7$ $d_{3,2}^5=\min\{d_{3,5}^4+d_{5,2}^4 d_{3,2}^4\}=\min\{6+6; 4\}=4$ $d_{3,3}^5=\min\{d_{3,5}^4+d_{5,3}^4 d_{3,3}^4\}=\min\{6+6; 0\}=0$ $d_{3,4}^5=\min\{d_{3,5}^4+d_{5,4}^4 d_{3,4}^4\}=\min\{6+4; 4\}=4$ $d_{3,5}^5=\min\{d_{3,5}^4+d_{5,5}^4 d_{3,5}^4\}=\min\{6+0; 6\}=6$ i=4 j=1-5 $d_{4,1}^5=\min\{d_{4,5}^4+d_{5,1}^4 d_{4,1}^4\}=\min\{4+9; 5\}=5$ $d_{4,2}^5=\min\{d_{4,5}^4+d_{5,2}^4 d_{4,2}^4\}=\min\{4+6; 2\}=2$ $d_{4,3}^5=\min\{d_{4,5}^4+d_{5,3}^4 d_{4,3}^4\}=\min\{4+6; 4\}=4$ $d_{4,4}^5=\min\{d_{4,5}^4+d_{5,4}^4 d_{4,4}^4\}=\min\{4+4; 0\}=0$ $d_{4,5}^5=\min\{d_{4,5}^4+d_{5,5}^4 d_{4,5}^4\}=\min\{4+0; 4\}=4$ i=5 j=1-5 $d_{5,1}^5=\min\{d_{5,5}^4+d_{5,1}^4 d_{5,1}^4\}=\min\{0+9; 9\}=9$ $d_{5,2}^5=\min\{d_{5,5}^4+d_{5,2}^4 d_{5,2}^4\}=\min\{0+6; 6\}=6$ $d_{5,3}^5=\min\{d_{5,5}^4+d_{5,3}^4 d_{5,3}^4\}=\min\{0+6; 6\}=6$ $d_{5,4}^5=\min\{d_{5,5}^4+d_{5,4}^4 d_{5,4}^4\}=\min\{0+4; 4\}=6$ $d_{5,5}^5=\min\{d_{5,5}^4+d_{5,5}^4 d_{5,5}^4\}=\min\{0+0; 0\}=0$	1(a) 2(b) 3(c) 4(d) 5(e) **k=6 $D^{(6)}$** 1(a) 0 3 7 5 9 2(b) 3 0 4 2 6 3(c) 7 4 0 4 6 4(d) 5 2 4 0 4 5(e) 9 6 6 4 0

Таким чином, на рисунку 2 показані зважений граф, представлення його за допомогою матриці ваг, а також матриця відстаней між будь-якими його двома вершинами, яка отримана за допомогою алгоритму Флойда.



а

	1(a)	2(b)	3(c)	4(d)	5(e)
1(a)	0	3	∞	7	∞
2(b)	3	0	4	2	∞
3(c)	∞	4	0	4	6
4(d)	7	2	4	0	4
5(e)	∞	∞	6	4	0

б

	1(a)	2(b)	3(c)	4(d)	5(e)
1(a)	0	3	7	5	9
2(b)	3	0	4	2	6
3(c)	7	4	0	4	6
4(d)	5	2	4	0	4
5(e)	9	6	6	4	0

в

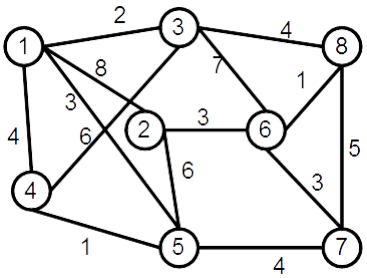
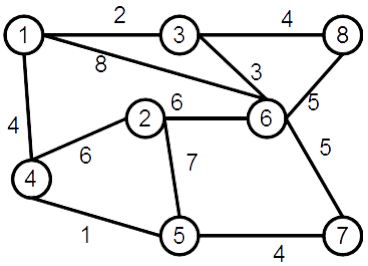
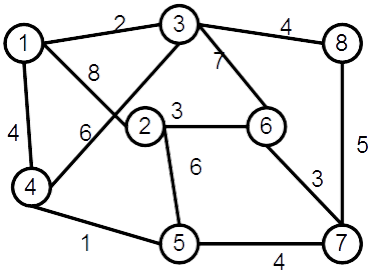
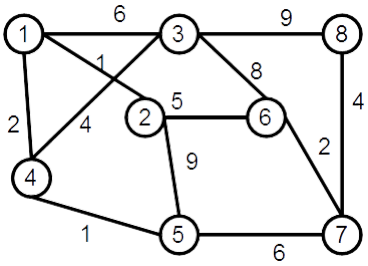
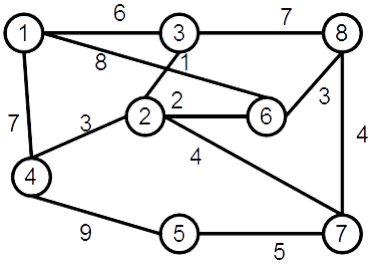
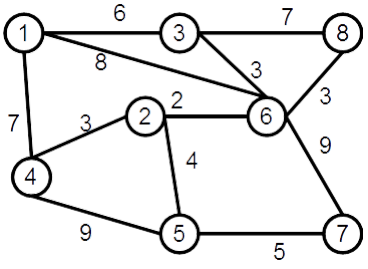
Рис. 2 Зважений граф (а), його матриці суміжності (б) та відстаней (в)

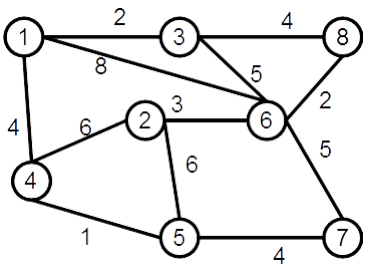
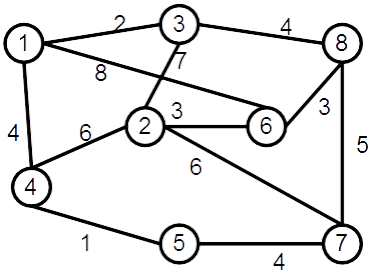
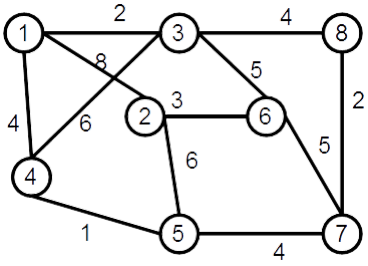
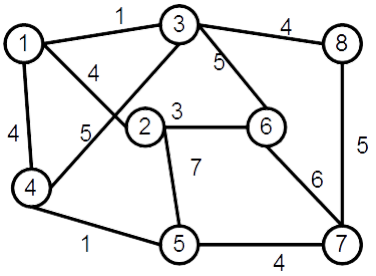
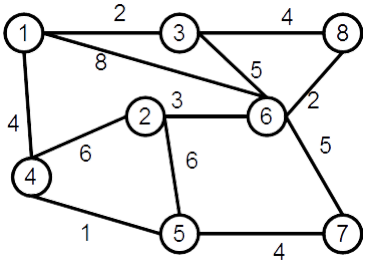
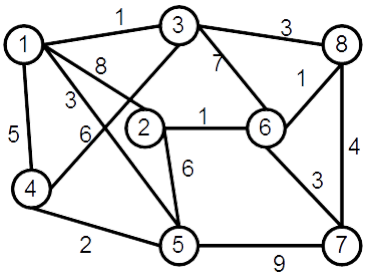
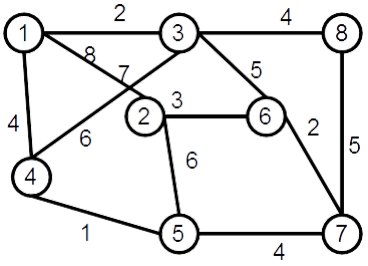
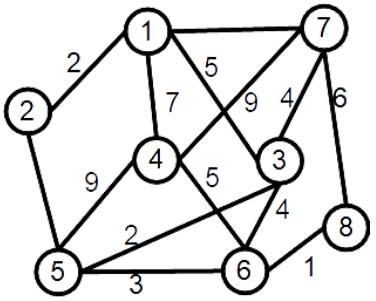
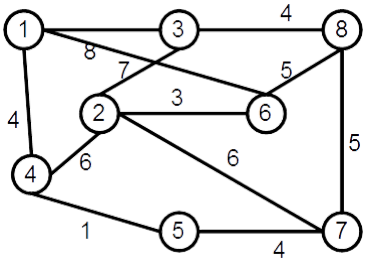
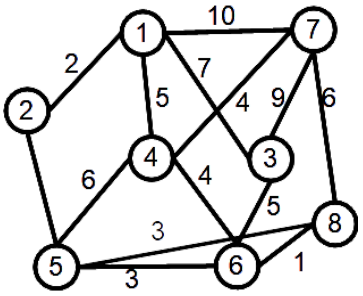
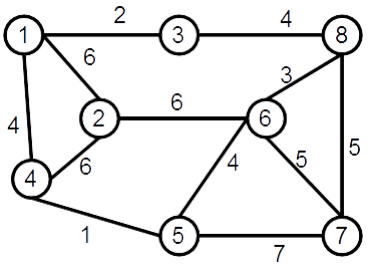
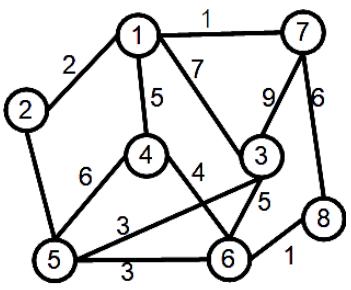
Завдання до лабораторної роботи №5

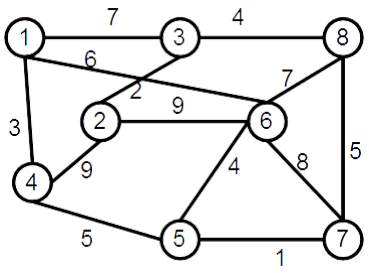
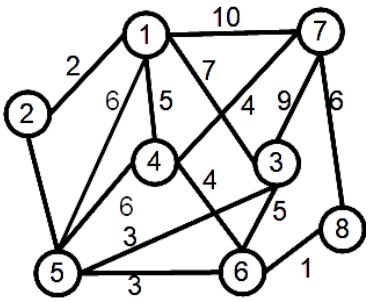
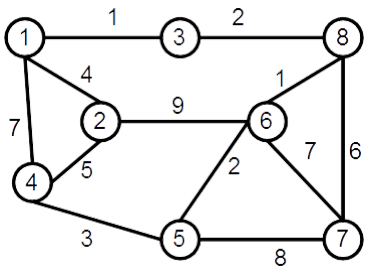
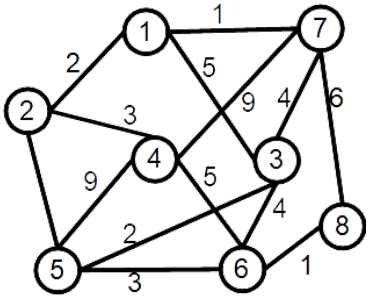
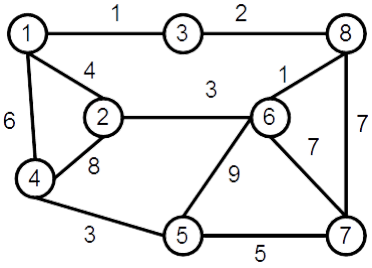
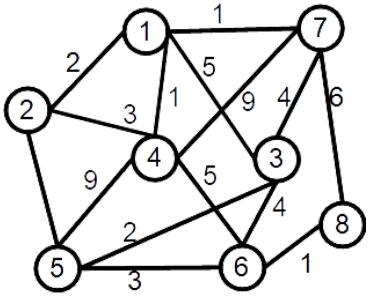
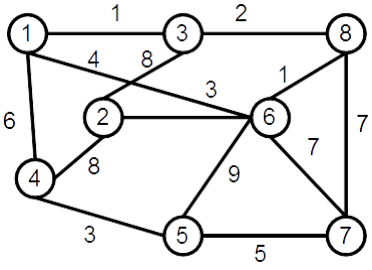
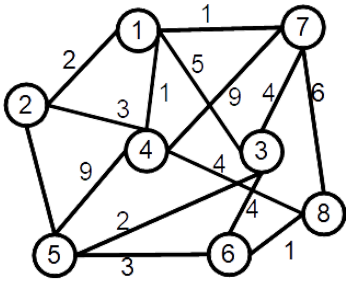
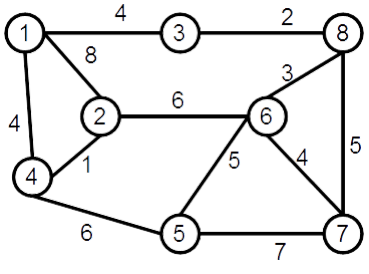
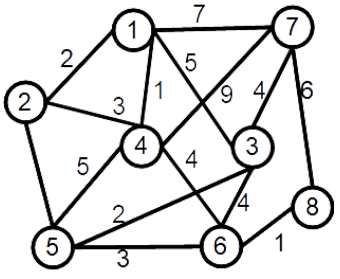
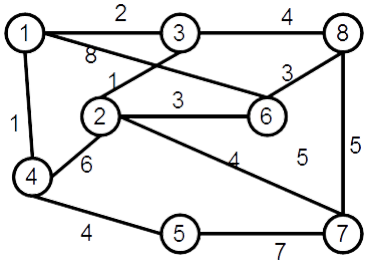
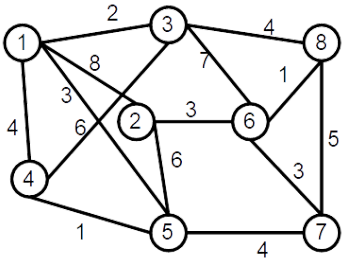
1. Побудувати за допомогою алгоритму Дейкстри найкоротший шлях із будь-якої вершини графа за варіантами завдань заповнити таблиці 1 та 2

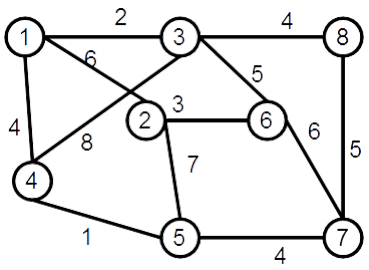
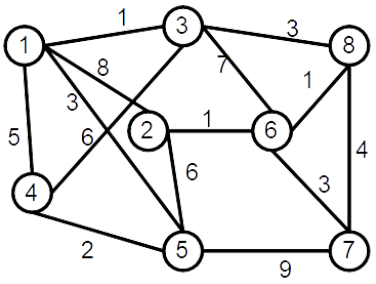
2. Показати графічно хід побудови найкоротшого шляху за алгоритмом Дейкстри, використовуючи процедуру *Dijkstra*(*G*, *s*)
3 Запрограмувати алгоритм Дейкстри. Запустити програму, показати результат за варіантами завдань. Показати, що результати ручної та автоматизованої побудови найкоротшего шляху співпадають
4. Показати хід побудови матриці відстаней за алгоритмом Флойда, використовуючи процедуру *Floyd* (*W* [1..*n*, 1..*n*]). Заповнити таблицю 3.
5.Запрограмувати алгоритм Флойда. Запустити програму, показати результат за варіантами завдань. Показати, що результати ручної та автоматизованої побудови матриць відстаней співпадають
6. Порівняти між собою результати моделювання за аргоритмом Дейкстри та Флойда
7. Звіт повинен містити:
− хід побудови шляхів за алгоритмами Дейкстри та Флойда заповнені таблиці 1,2 та 3
− лістинги програм, результати виконання програм з контрольного прикладу за варіантами завдань,
− порівняння алгоритмів Дейкстри та Флойда в висновках щодо практичної роботи №5.

Варианти завданя:

№	Граф	№	Граф
1		17	
2		18	
3		19	

№	Граф	№	Граф
4		20	
5		21	
6		22	
7		23	
8		24	
9		25	

№	Граф	№	Граф
10		26	
11		27	
12		28	
13		29	
14		30	
15		31	

№	Граф	№	Граф
16		32	

Лабораторна робота №6

Робота з бінарним та червоно-чорним деревами пошуку

Розглядається бінарне дерево пошуку та червоно-чорне дерево як структури даних для вирішення задачі пошуку. Розглядаються властивості цих структур пошуку, алгоритми вставки та видалення елемента, а також обходу бінарного дерева пошуку.

Питання:

1. Основні відомості про бінарні та червоно-чорні дерева пошуку.
2. Вставка вузла в бінарне дерево пошуку
3. Вставка елементів у червоно-чорне дерево (3 випадки)
4. Приклади додавання елементів
5. Обходи бінарного та червоно-чорного дерев пошуку
6. Видалення елементів бінарного дерева пошуку
7. Видалення елементів з червоно-чорного дерева (5 випадків)
8. Приклади видалення елементів
9. Завдання
10. Оформлення звіту

1. Основні відомості про бінарні та червоно-чорні дерева пошуку.

Див. у презентації лекції

2. Вставка вузла в бінарне дерево пошуку

Операція вставки вузла в бінарне дерево пошуку працює *аналогічно* пошуку вузла, відмінністю є те, що якщо виявлено у вузла відсутність дочірнього елемента необхідно «підвісити» на нього елемент, що вставляється.

Розглянемо функцію побудови бінарного дерева пошуку (binary search tree, BST)

Func buildBST (a : int[n]):
for i = 1 to n
Node insert(x, a[i])

```
Node insert(x : Node, z : T):   // x — корінь піддерева, z — ключ, що вставляється
    if x == null
      return Node(z)                    // вставляємо Node з key = z
    else if z < x.key
      x.left = insert(x.left, z)
    else if z > x.key
      x.right = insert(x.right, z)
    return x
```

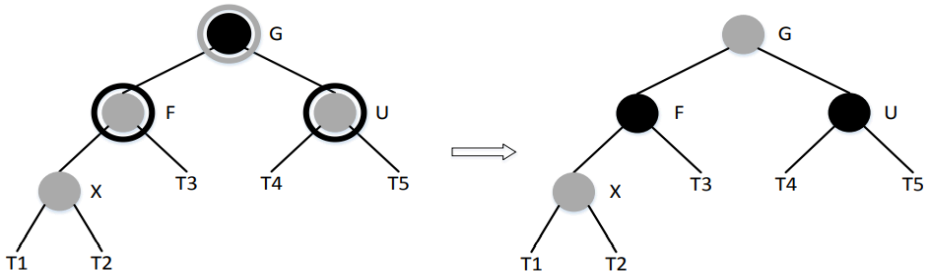
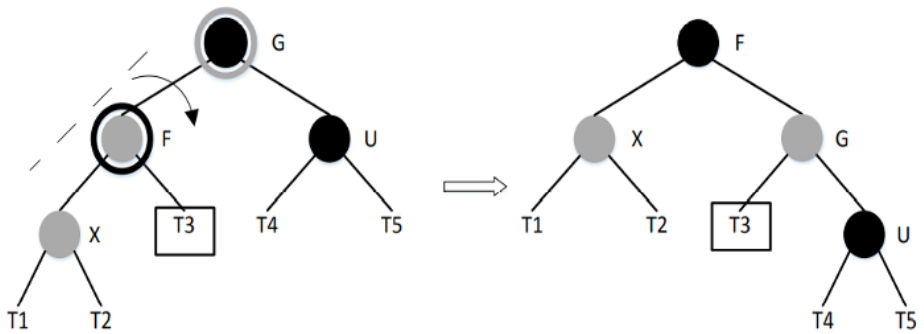
У циклі по елементах масиву запускається функція **Node** insert(x, a[i]). Бінарне дерево пошуку будується так: перший елемент стає кореневим елементом дерева, після чого наступні елементи порівнюються з кореневим. Якщо елемент менше кореневого – він йде в ліве піддерево і порівнюється з лівим нащадком кореневого елемента. Якщо елемент більше кореневого – він іде у праве піддерево і порівнюється з правим нащадком кореневого елемента. Це триває доки для елемента не знайдеться вільне місце.

3. Вставка елементів у червоно-чорне дерево (red-black tree RB tree) (3 випадки)

Виконується за наступним алгоритмом:
Знаходимо лист для вставки нового елемента
Створюємо елемент і фарбуємо його у червоний колір
Перефарбовуючи вузли та виконуючи повороти, відновлюємо структуру червоно-чорного дерева

При вставці вузла *X* **(червоного)** у ЧЧД можливі 6 випадків, що порушують властивості ЧЧД, при цьому 3 випадки **симетричні** іншим трьом (Таблиця 1).

Таблиця 1 Правила балансування при вставці вузла *X* **(червоного)** у ЧЧД

Випадок 1 Дядько *U* вузла *X*, що додається, червоний *Розв'язання:* перефарбування *F* та *U* у чорний колір, а *G* у червоний	
Далі балансування виконується відносно вузла *G*	
Випадок 2 Дядько *U* вузла *X*, що додається, чорний і при цьому ланцюжок вузлів *X-F-G* утворює пряму лінію *Розв'язання:* перефарбування *F* та *G* й	

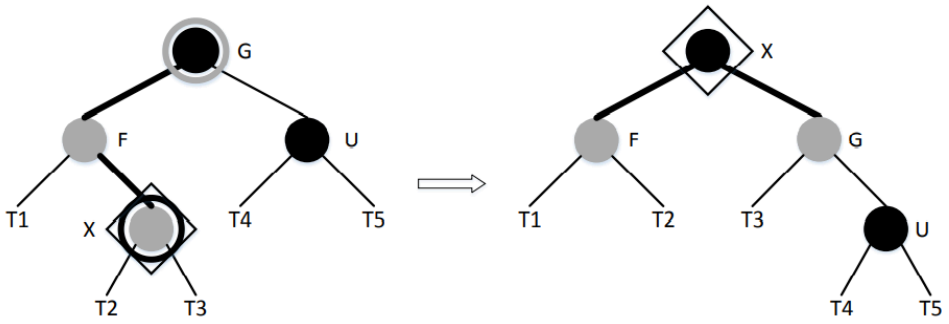
одинарний правий поворот	

Правий поворот виконується «навколо» зв'язку між *G* і *F*, роблячи *F* новим коренем піддерева, правим дочірнім вузлом якого стає *G* , а колишній правий нащадок вузла *F (T3)* – лівим нащадком *G*.

Випадок 3

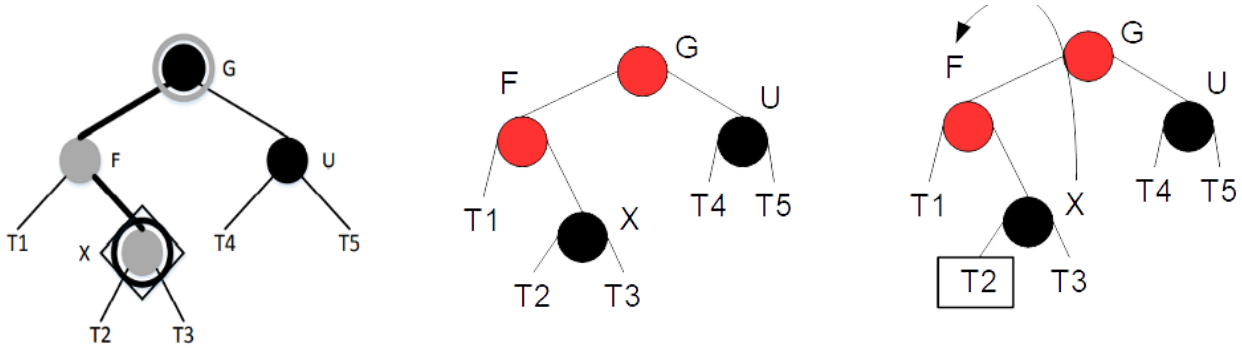
Дядько *U* вузла *X*, що додається, чорний і при цьому ланцюжок вузлів *X-F-G* утворює кут

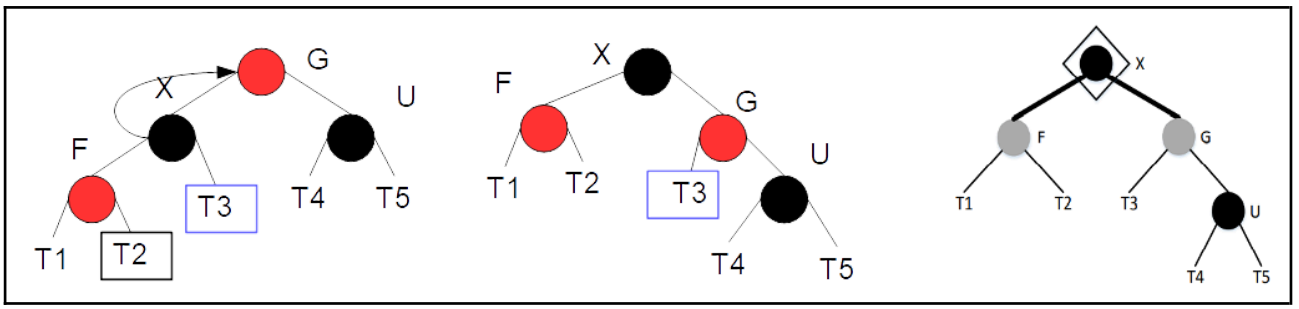
Розв'язання: перефарбування *X* та *G*, і подвійний (лівий та правий) поворот комбінації *X-F-G*, коли нижній вузол *(X)* опиняється нагорі комбінації.



Лівий поворот виконується «навколо» зв'язку між *F* і *X*, роблячи *X* новим коренем піддерева, лівим дочірнім вузлом якого стає *F*, а колишній лівий нащадок вузла *X (T2)* – правим нащадком *F*

Правий поворот виконується «навколо» зв'язку між *G* і *X*, роблячи *X* новим коренем піддерева, правим дочірнім вузлом якого стає *G*, а колишній правий нащадок вузла *X (T3)* - лівим нащадком *G*





4. Приклади додавання елементів

У таблиці 2, яку *розміщено в додатковому до ЛР6* файлі показаний процес побудови бінарного дерева пошуку та червоно-чорного дерева за допомогою вставки вузла для наступної множини елементів: 90, 29, 76, 21, 30, 15, 82, 67, 85. Вигляд отриманих дерев показаний на рисунку 1.

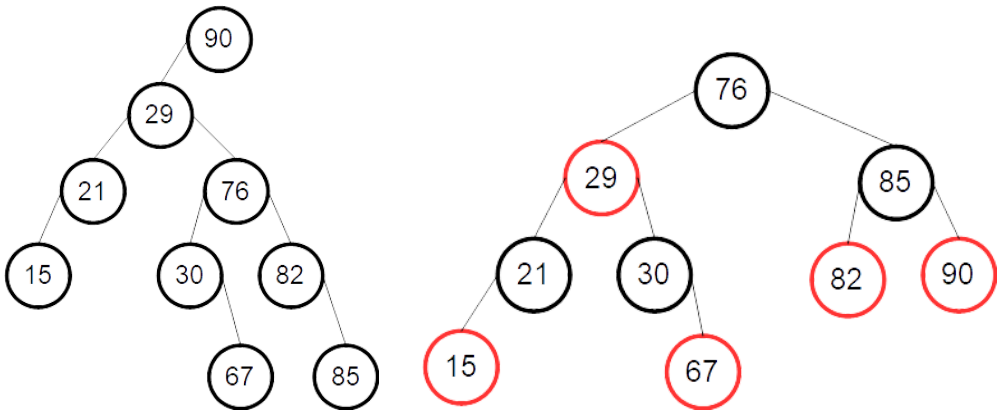


Рисунок 1 Вигляд BST(а) та RB tree (б) побудованих із {90, 29, 76, 21, 30, 15, 82, 67, 85}

5. Обходи бінарного та червоно-чорного дерев пошуку

Є три операції обходу вузлів дерева, що відрізняються порядком обходу вузлів:

inorderTraversal — обхід вузлів у симетричному (відсортованому) порядку,

preorderTraversal — обхід вузлів у прямому порядку: вершина, ліве піддерево, праве піддерево,

postorderTraversal — обхід вузлів у зворотному порядку: ліве піддерево, праве піддерево, вершина.

Обхід вузлів у симетричному (відсортованому) порядку

```
func inorderTraversal(x : Node):
        if x != null
          inorderTraversal(x.left)
          print x.key
          inorderTraversal(x.right)
```

Відповідно до inorderTraversal алгоритму порядок обходу елементів побудованого бінарного дерева пошуку у симетричному порядку наступний: 15, 21, 29, 30, 67, 76, 82, 85, 90.

При цьому стек викликів функції inorderTraversal на кожній ітерації буде виглядати наступним чином:

Стек виклику симетричного обходу

1	90				
2	90	29			
3	90	29	21		
4	90	29	21	**15**	
5	90	29	**21**		
6	90	**29**			
7	90	29	76		
8	90	29	76	**30**	
9	90	29	76	30	**67**
10	90	29	76	30	
11	90	29	**76**		
12	90	29	76	**82**	
13	90	29	76	82	**85**
14	90	29	76	82	
15	90	29	76		
16	90	29			
17	**90**				

Відповідно до inorderTraversal алгоритму порядок обходу елементів червоно-чорного дерева у симетричному порядку наступний: 15, 21, 29, 30, 67, 76, 82, 85, 90.

При цьому стек викликів функції inorderTraversal на кожній ітерації буде виглядати наступним чином:

1	76				
2	76	29			
3	76	29	21		
4	76	29	21	**15**	
5	76	29	**21**		
6	76	**29**			
7	76	29	**30**		
8	76	29	30	**67**	
9	76	29	30		
10	76	29			
11	**76**				
12	76	85			
13	76	85	**82**		
14	76	**85**			
15	76	85	**90**		
16	76	85			
17	76				

Як бачимо в першому и другому випадках кількість ітерацій при симетричному обході однакова. Але в першому випадку в стеку викликів на 9 та 13 ітераціях міститься по 5 викликів відповідно на відміну від другого випадку, де 4 виклики - максимальна кількість викликів. Тому для реалізації inorderTraversal для BST знадобилося більше витрат пам'яті ніж для RB tree.

Обхід вузлів у прямому порядку

```
func preorderTraversal(x : Node)
  if x != null
    print x.key
    preorderTraversal(x.left)
    preorderTraversal(x.right)
```

Відповідно до preorderTraversal алгоритму порядок обходу елементів бінарного дерева у прямому порядку наступний: 90,29,21,15,76,30,67,82,85.
Відповідно до preorderTraversal алгоритму порядок обходу елементів червоно-чорного дерева у прямому порядку наступний:76,29,21,15,30,67,85,82,90.

Обхід вузлів у зворотному порядку

```
func postorderTraversal(x : Node)
  if x != null
    postorderTraversal(x.left)
    postorderTraversal(x.right)
    print x.key
```

Відповідно до postorderTraversal алгоритму порядок обходу елементів бінарного дерева у зворотному порядку наступний:15,21,67,30,85,82,76,29,90.
Відповідно до postorderTraversal алгоритму порядок обходу елементів червоно-чорного дерева у зворотному порядку наступний:15,21,67,30,29,82,90,85,76.

6. Видалення елементів з бінарного дерева пошуку

Рекурсивне видалення вузла із BST

```
Node delete(root : Node, z : T):          // корінь піддерева, ключ, що видаляється
  if root == null
    return root
  if z < root.key //елемент, що видаляється, знаходиться в лівому піддереві
    root.left = delete(root.left, z)
  else if z > root.key //елемент, що видаляється, знаходиться в правому піддереві
    root.right = delete(root.right, z)
  else if root.left != null and root.right != null //елемент, що видаляється, знаходиться в корені і має два дочірні вузли
    root.key = minimum(root.right).key
```

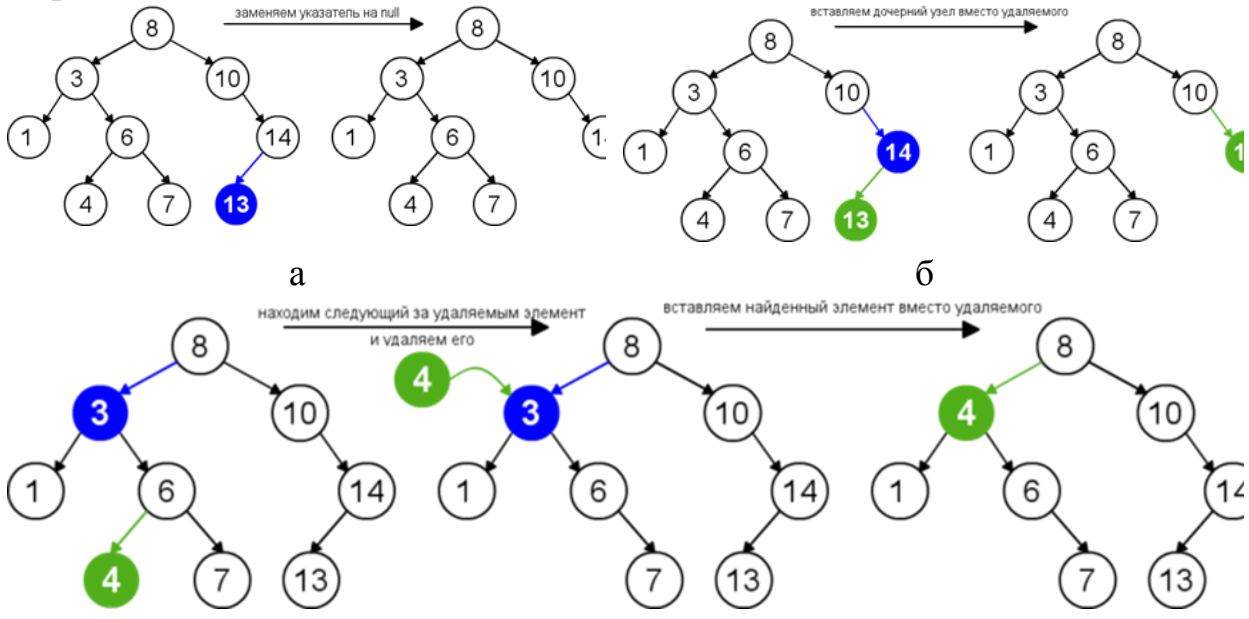
```
    root.right = delete(root.right, root.key)
  else //елемент, що видаляється, має один дочірній вузол, потрібно замінити його нащадком
    if root.left != null
      root = root.left
    else if root.right != null
      root = root.right
    else
      root = null
  return root
```

При рекурсивному видаленні вузла з бінарного дерева слід розглянути три випадки (Рис.2):

1. елемент, що видаляється, знаходиться в лівому піддереві поточного піддерева,
2. елемент, що видаляється, знаходиться в правому піддереві
3. елемент, що видаляється, знаходиться в корені.

У перших двох випадках потрібно рекурсивно видалити елемент з потрібного піддерева.

Якщо елемент, що видаляється, знаходиться в корені поточного піддерева і має два дочірні вузли, то потрібно замінити його мінімальним (крайнім лівим) елементом з правого піддерева і рекурсивно видалити **цей** мінімальний елемент з правого піддерева. Інакше, якщо елемент, що видаляється, має один дочірній вузол, потрібно замінити його нащадком. Час роботи алгоритму — O(h). Рекурсивна функція, що повертає дерево з видаленим елементом z Наприклад



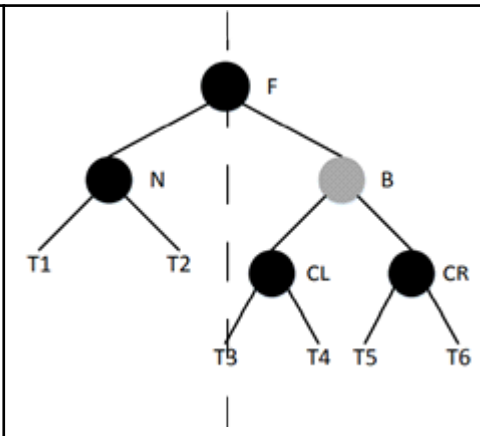
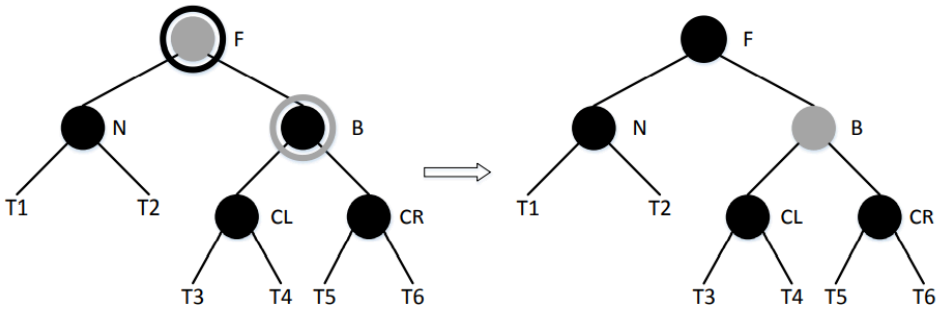
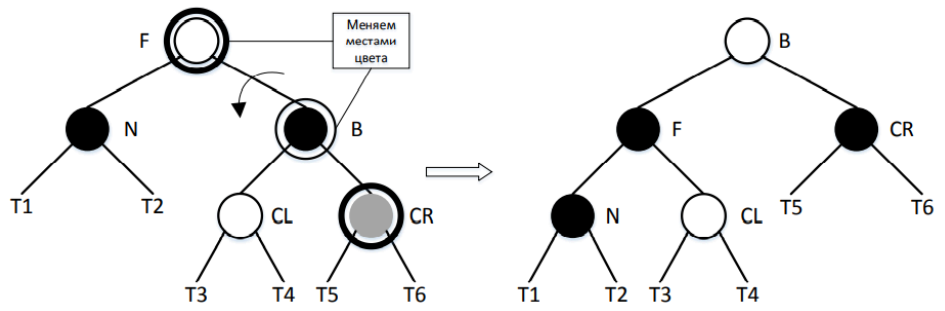
а б в

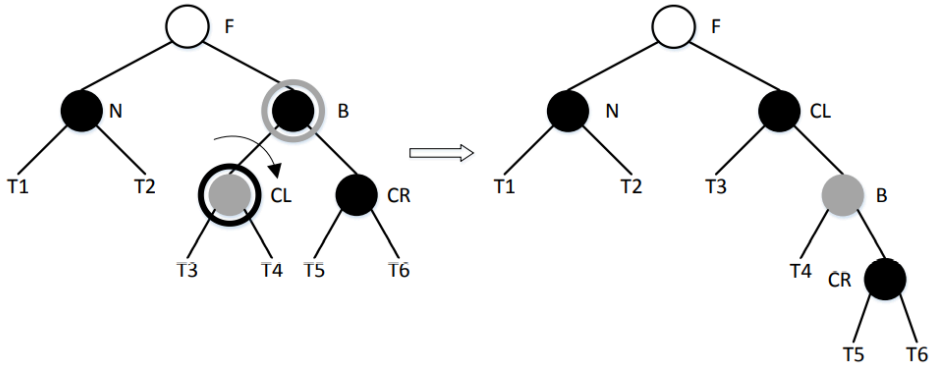
Рис. 2 Видалення вузла із бінарного дерева пошуку

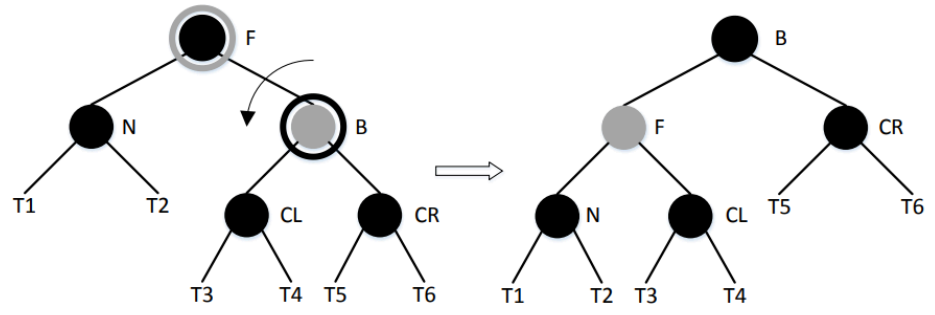
7. Видалення елементів із червоно-чорного дерева

Видалення вузла з ЧЧД, так само, як і вставка вузла, відбувається у два етапи: власне видалення за допомогою алгоритму видалення з BST, та відновлення властивостей ЧЧД, якщо вони були порушені.
Якщо вузол, що видаляється з дерева, червоний, червоно-чорні властивості дерева зберігаються.
Таким чином, відновлення властивостей дерева може знадобитися тільки в тому випадку, якщо вузол, що видаляється, – *чорний*. Якщо при цьому син вузла, що видаляється, червоний, то після видалення достатньо буде перефарбувати сина видаленого вузла в чорний колір, щоб відновити кількість чорних вузлів на цьому шляху.
Розглянемо 5 випадків конфігурації дерева після видалення *чорного* вузла, у якого син – *чорний* (Таблиця 3).

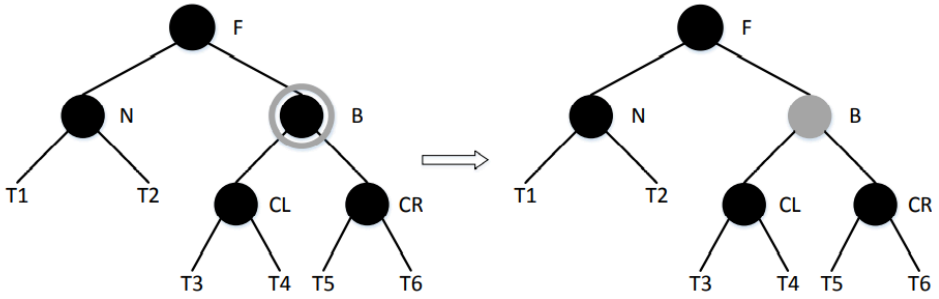
Таблиця 3 Правила балансування ЧЧД після видалення *чорного* вузла, у якого син – *чорний*

Позначимо сина видаленого вузла за *N*, а батька видаленого вузла за *F*. Після видалення *F* став новим батьком *N*. Позначимо за *B* нового брата *N*, а за *CL* та *CR* – лівого та правого синів *B*, відповідно. На те, якою буде процедура відновлення, впливає тільки комбінація з цих 5 вузлів. Будемо вважати, що *N* – лівий син *F*. Інакше, усі зображення будуть симетричними щодо вертикальної осі, яка проходить через *F*.	
Випадок 1. Батько *F* вузла *N* червоний, інші вузли чорні. (*Батько F вузла N червоний: перефарбування B та F*)	
Випадок 2. Брат *B* вузла *N* чорний, а його правий син *CR* червоний. Незалежно від того, чи *F* є чорним або	

<table>
<tr><td>червоним, властивості ЧЧД відновлюються після лівого повороту F навколо B, перефарбування CR в чорний колір і обміну кольорами F і B.</td><td>Лівий поворот навколо F та B, перефарбування CR в чорний та обмін кольорами F та B</td></tr>
<tr><td colspan="2">Лівий поворот виконується «навколо» зв'язку між F і B, роблячи B новим коренем піддерева, лівим дочірнім вузлом якого стає F, а колишній лівий нащадок вузла B (CL) - правим нащадком F.</td></tr>
<tr><td>Випадок 3. Брат B вузла N чорний, його правий син CR чорний, а лівий син CL червоний.</td><td> 

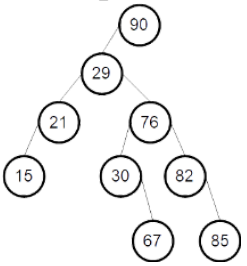
Правий поворот CL навколо B, перефарбування CL та B</td></tr>
<tr><td colspan="2">Правий поворот виконується «навколо» зв'язку між B і CL, роблячи CL новим коренем піддерева, правим дочірнім вузлом якого стає B, а колишній правий нащадок вузла CL (T4) – лівим нащадком B</td></tr>
<tr><td>Випадок 4. Брат B вузла N червоний. У цьому випадку F, CL та CR можуть бути лише чорними</td><td> 

Лівий поворот F навколо B та перефарбування F та B</td></tr>
<tr><td colspan="2">Лівий поворот виконується «навколо» зв'язку між F і B, роблячи B новим коренем піддерева, лівим дочірнім вузлом якого стає F, а колишній лівий нащадок вузла B (CL) - правим нащадком F</td></tr>
</table>

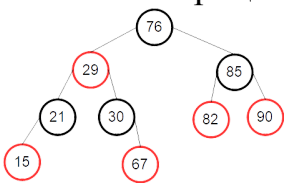
Випадок 5. Усі вузли комбінації (брат *B*, його діти *CL* та *CR*, а також батько *F* вузла *N*) чорні.	 *Перефарбування вузла B у червоний колір та продовження процедури нагору*

8. Приклади видалення елементів

У таблиці 4, яку *розміщено в додатковому до ЛР6* файлі показаний процес почергово видалення кореневих елементів *правого* та *лівого* піддерев, а також основного кореня із побудованого бінарного дерева пошуку



У таблиці 5, яку *розміщено в додатковому до ЛР6* файлі показаний процес почергово видалення із з побудованого червоно-чорного дерева трьох чорних елементів, які мають переважно чорних синів



9. Завдання:

1. Побудувати бінарне дерево пошуку та побудувати, наводячи покроковий червоно-чорне дерево за варіантами завдань (зразок таблиця 2 додаткового до ЛР6 файл)
2. Навести результати симетричного, прямого та зворотного обходів побудованого бінарного дерева пошуку та червоно-чорне дерева. Для симетричного обходу навести стан стека викликів рекурсивної функції inorderTraversal. Порівняйте результати, зробить висновки
3. Видаліть по черзі та опишіть процедуру видалення із побудованого бінарного дерева пошуку трьох кореневих елементів в такому порядку: праве, ліве піддерева, основний кореневий елемент (зразок таблиця 4 додаткового до ЛР6 файлу)
4. Видаліть по черзі та опишіть процедуру видалення із побудованого червоно-чорного дерева трьох чорних елементів, які мають переважно чорних синів (зразок таблиця 5 додаткового до ЛР6 файлу)

Варіанти завдань:

Варіант	Послідовність
1	53, 12, 68, 63, 3, 47, 50, 61, 41
2	98, 40, 42, 88, 61, 87, 79, 97, 82
3	70, 80, 61, 92, 12, 23, 67, 65, 50
4	29, 1, 24, 69, 52, 97, 27, 10, 88
5	95, 43, 98, 41, 68, 67, 10, 7, 69
6	61, 32, 27, 45, 75, 58, 5, 50, 99
7	78, 77, 4, 62, 8, 69, 46, 11, 49
8	28, 76, 27, 10, 5, 35, 95, 16, 33
9	31, 40, 22, 50, 53, 68, 97, 12, 15
10	18, 20, 2, 88, 61, 17, 79, 97, 82
11	41, 52, 47, 65, 95, 38, 15, 50, 99
12	11, 42, 67, 55, 65, 78, 25, 50, 69
13	53, 5, 44, 47, 35, 83, 82, 85, 28
14	77, 89, 74, 68, 70, 49, 5, 62, 51
15	19, 75, 43, 31, 5, 66, 62, 34, 76
16	50, 57, 78, 34, 41, 68, 47, 61, 38
17	86, 36, 14, 50, 64, 21, 2, 83, 82
18	80, 27, 37, 36, 91, 53, 86, 66, 98
19	21, 44, 22, 50, 63, 68, 97, 12, 15
20	7, 89, 4, 68, 70, 49, 10, 62, 51
21	54, 65, 7, 33, 86, 29, 11, 91, 12
22	50, 80, 19, 86, 35, 7, 60, 48, 51
23	10, 90, 95, 30, 45, 60, 57, 28, 5
24	90, 10, 15, 80, 100, 6, 57, 5, 29
25	53, 100, 44, 74, 53, 38, 82, 65, 28
26	47, 50, 61, 41,53, 12, 68, 63, 3
27	87, 79, 97, 82,98, 40, 42, 88, 61
28	12, 23, 67, 65, 50, 70, 80, 61, 92
29	69, 52, 97, 27, 10, 88,29, 1, 24
30	41, 68, 67, 10, 7, 69, 95, 43, 98
31	58, 5, 50, 99, 61, 32, 27, 45, 75
32	46, 11, 49, 78, 77, 4, 62, 8, 69
33	35, 95, 16, 33, 28, 76, 27, 10, 5
34	68, 97, 12, 15, 31, 40, 22, 50, 53
35	79, 97, 82, 18, 20, 2, 88, 61, 17
36	38, 15, 50, 99, 41, 52, 47, 65, 95

10.Оформлення звіту.

Звіт повинен містити:

− хід побудови бінарного дерева пошуку та червоно-чорного дерева за варіантами завдань (зразок таблиця 2 додаткового файлу)

− результати прямого та зворотного обходів побудованого бінарного дерева пошуку та червоно-чорного дерева; для симетричного обходу стан стека викликів рекурсивної функції inorderTraversal та результати обходу.

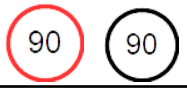
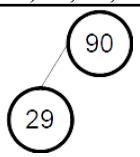
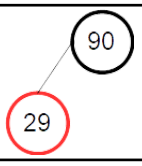
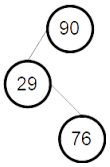
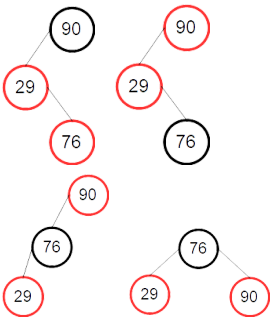
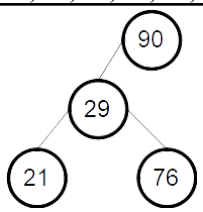
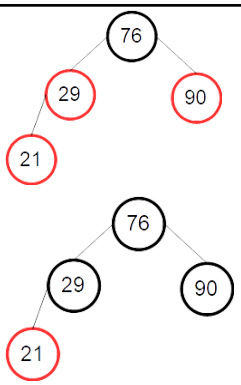
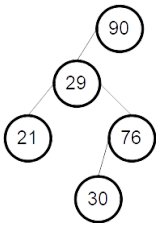
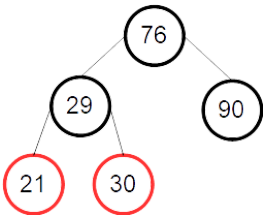
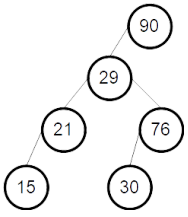
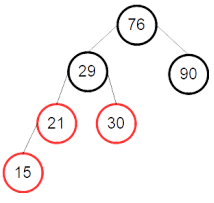
− результати видалення із бінарного дерева пошуку трьох кореневих елементів в такому порядку: праве, ліве піддерева, основний кореневий елемент (зразок таблиця 4 додаткового до ЛР6 файлу)

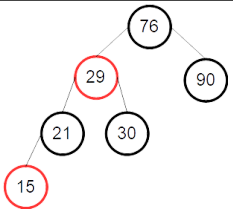
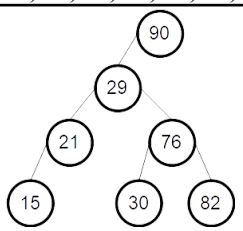
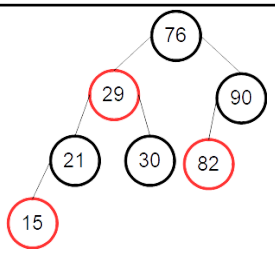
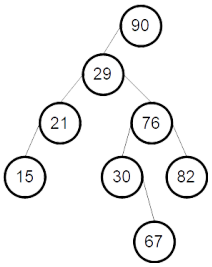
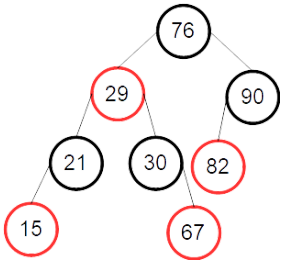
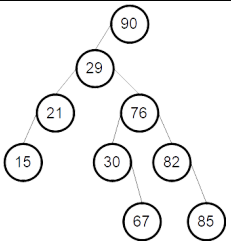
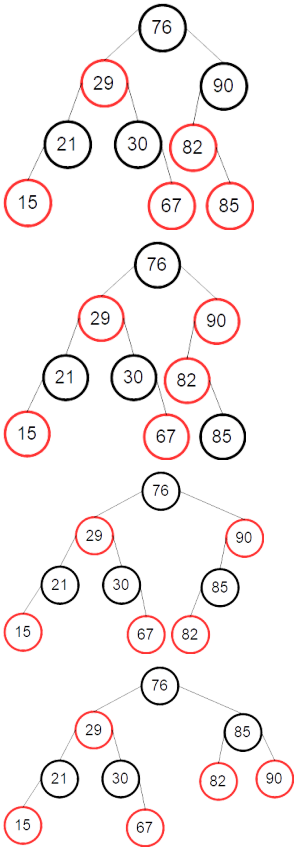
− результати видалення із червоно-чорного дерева трьох чорних елементів, які мають переважно чорних синів (зразок таблиця 5 додаткового до ЛР6 файлу)

Приклади виконання завдань ЛР6

1. За варіантами завдань побудувати бінарне дерево пошуку та червоно-чорне дерево за алгоритмом вставки. Для побудови червоно-чорного дерева навести покроковий опис

Таблиця 2 Побудова бінарного та червоно-чорного дерева пошуку

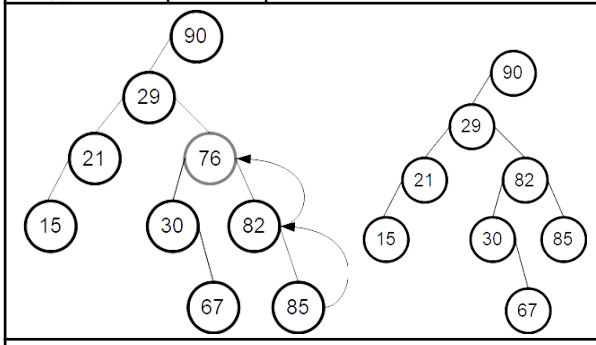
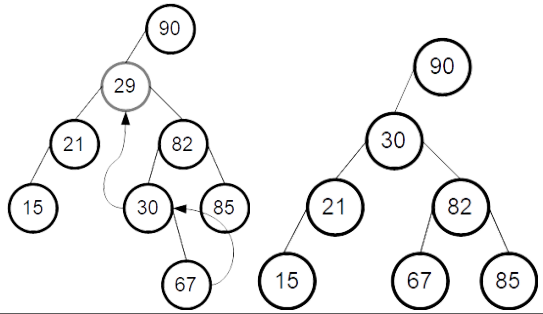
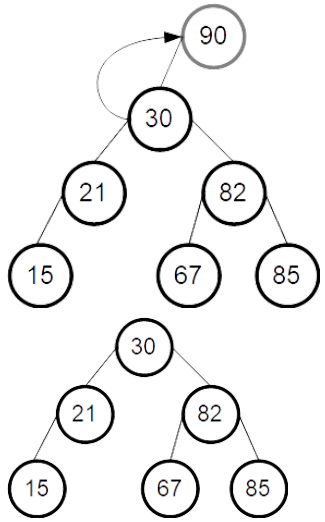
90, 29, 76, 21, 30, 15, 82, 67, 85		
Побудова BST Додаткових пояснень не потребує	Побудова RB tree. Пояснення дій за алгоритмом вставки обов'язкові	
		Вставляємо елемент 90. Випадок 1 – перефарбування в чорний колір
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Вставляємо елемент 29. Він стає лівим сином елемента 90. Оскільки батько чорний, немає порушень, нічого не міняємо.
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Додавання елемента 76. Він стає правим сином елемента 29. Червоне-червоне порушення. **Випадок 3** - утворився кут. Перефарбовуємо 76 чорним кольором, а 90 червоним кольором. Робимо лівий поворот навколо вузла 76, а потім правий поворот навколо вузла 76, вузол 76 став кореневим.
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Додавання елемента 21. Він стає червоним лівим сином вузла 29. Червоне-червоне порушення. **Випадок 1** - утворення прямої лінії, дядько доданого вузла червоний. Перефарбовуємо батька та дядька вузла 21 у чорний колір.
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Додавання елемента 30. Він стає червоним правим сином вузла 29. Порушень немає
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Додавання елемента 15. Він стає червоним лівим сином вузла 21. Червоне-червоне порушення. **Випадок 1** - утворюється прямої лінії, дядько доданого вузла червоний. Перефарбовуємо батька (21) та дядька (30) вузла 15 чорним кольором, а дідуся вузла 15 вузел (29) червоним кольором.

		Порушень відносно вузла 29 немає. Вимоги стосовно чорної висоти виконуються
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Додавання елемента 82. Він стає червоним лівим сином вузла 90. Порушень немає
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Додавання елемента 67. Він стає червоним правим сином вузла 30. Порушень немає
90, 29, 76, 21, 30, 15, 82, 67, 85		
		Додавання елемента 85. Він стає правим сином елемента 82. Червоне-червоне порушення. **Випадок 3** утворення кута. Перефарбовуємо 85 чорним кольором, а 90 червоним кольором. Робимо лівий поворот навколо вузла 82, а потім правий поворот навколо вузла 90

Дерево побудовано. Висота бінарного дерева пошуку дорівнює 5. Чорна висота червоно-чорного дерева дорівнює 2, а висота 4

2. Видаліть по черзі та опишіть процедуру видалення із побудованого бінарного дерева пошуку трьох кореневих елементів в такому порядку: праве, ліве піддерева, основний кореневий елемент

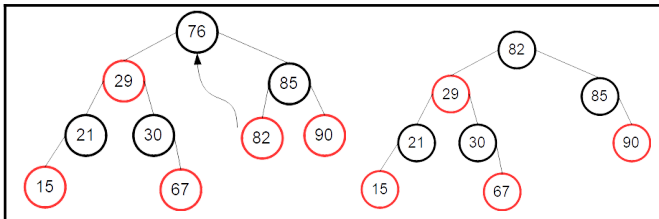
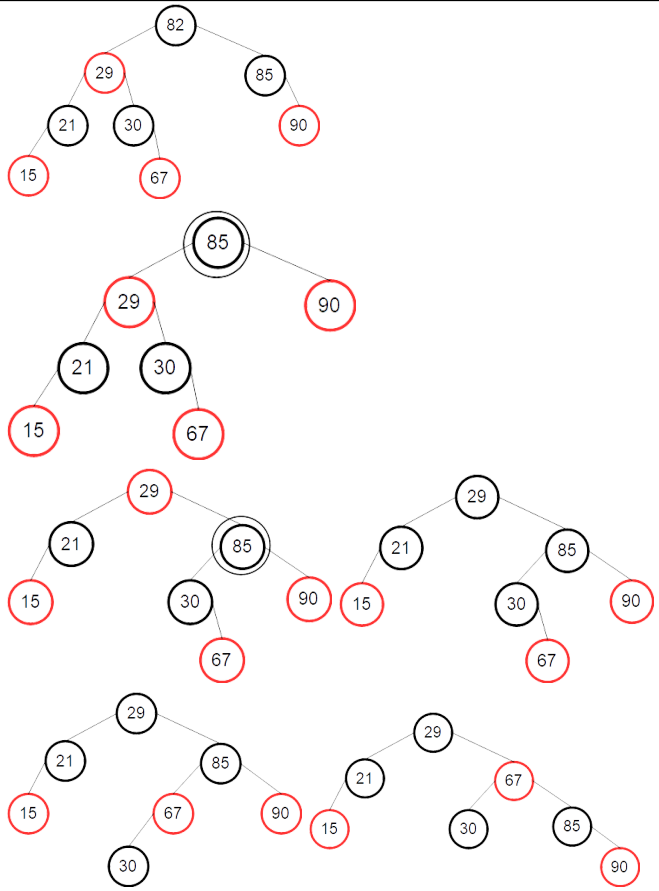
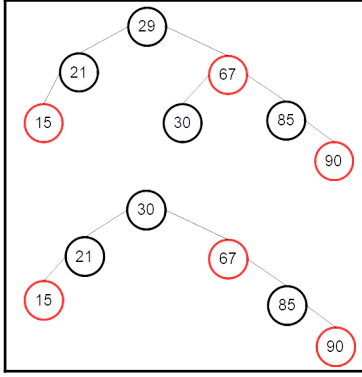
Таблиця 4 Видалення вузлів із бінарного дерева пошуку

Видаляємо правий кореневий елемент 76	
	Кореневий елемент правого піддерева 76, що видаляється, має два дочірніх вузла (елементи 30 та 82). Шукаємо крайній лівий елемент правого піддерева – це елемент 82 з правим нащадком – елементом 85. Тому елемент 82 стає на місце елемента 76, а елемент 85 на місце елемента 82
Видаляємо лівий кореневий елемент 29	
	Кореневий елемент лівого піддерева 29, що видаляється, має два дочірніх вузла (елементи 21 та 82). Шукаємо крайній лівий елемент правого піддерева – це елемент 30 з правим нащадком – елементом 67. Тому елемент 30 стає на місце елемента 29, а елемент 67 на місце елемента 30
Видаляємо кореневий елемент 90	
	Кореневий елемент дерева 90, що видаляється, має тільки один лівий дочірній вузол (елемент 30). Тому він займає місце кореневого елемента 90.

3. Видаліть по черзі та опишіть процедуру видалення із побудованого червоно-чорного дерева трьох чорних елементів, які *переважно* мають чорних синів

Таблиця 5 Видалення вузлів із червоно-чорного дерева

Видаляємо кореневий елемент 76	

	Кореневий елемент піддерева 76, що видаляється, має два дочірніх вузла (червоний 29 та чорний 82). Шукаємо крайній лівий елемент правого піддерева – це червоний елемент 82 – ставимо його на місце 76. Він кореневий тому перефарбовуємо його в чорний колір. Чорну висоту (вл. 4) не порушено
Видаляємо кореневий елемент 82	
	Кореневий елемент дерева 82, що видаляється, має два дочірніх вузла (червоний 29 та чорний 85). Шукаємо крайній лівий елемент правого піддерева – це чорний елемент 85 – ставимо його на місце 82. Маємо порушення чорної висоти. Крім того елемент 85 стає «двічі» чорнім. (Випадок 3) Робимо правий поворот навколо зв'язку вузлів 29 та 85, 29 стає новим коренем, правим дочірнім вузлом стає 85, колишні правий нащадок вузла 29 вузел 30 стає лівим нащадком вузла 85. Перефарбування 29 та 85 (віддає чорний колір) Порушення чорної висоти. Лівий поворот навколо 67 та 30 та правий поворот навколо 67 та 85
Видаляємо кореневий елемент 29	
	Кореневий елемент дерева 29, що видаляється, має два дочірніх вузла (чорний 21 та червоний 67). Шукаємо крайній лівий елемент правого піддерева – це чорний елемент 30 – ставимо його на місце 29. Порушень немає

Лабораторна робота №7

Робота з хеш-таблицями

Зміст:

Розглядається хеш-таблиці із відкритим (open hashing) та закритим (closed hashing) хешуванням як структури для побудови словників. *Відкрите хешування* також називають *хешуванням з роздільними ланцюжками* (separate chaining), а *закрите хешування* також називають *хешуванням з відкритою адресацією* (open addressing) Розглядаються властивості цих структур для виконання словникових операцій, а саме вставки, видалення та пошуку елемента.

1.Основні відомості про хеш-таблиці

Хеш-таблиця – це структура даних, що дозволяє ефективно реалізовувати словники (часто середній час пошуку елементу O(1)). Узагальнює поняття масиву. Наявність прямої індексації елементів масиву забезпечує доступ до довільної позиції у масиві за час O(1). В хеш-таблицях також може бути пряма індексація (якщо можемо виділити масив такого розміру, щоб кожне можливе значення ключа мало свою комірку). Але звичайно кількість ключів, що зберігаються в масиві, менша у порівнянні з кількістю можливих значень ключів. В таких випадках використовують хеш-таблиці, що використовують масиви, розмір яких пропорційний кількості ключів, що реально там зберігаються. Замість безпосереднього використання ключа як індексу масиву індекс обчислюється за значенням ключа.

Таблиці з прямою адресацією. Використовують для невеликої сукупності ключів. Припустимо, маємо справу із динамічною множиною, кожен елемент якої має ключ з $U = \{0,1,..., m\text{-}1\}$, де m є не дуже великим та жодні два елементи не мають однакових ключів. Для представлення динамічної множини використаємо таблицю з прямою адресацією (масив) $T[0..m-1]$, кожна комірка якої відповідає ключу із сукупності $U = \{0,1,..., m\text{-}1\}$. Тоді комірка k вказує на елемент множини з ключом k. Якщо множина не містить елемента з таким ключем, то T[k] = NIL . Тоді виконання словникових операцій (пошук, вставка, видалення) відбувається за час O(1)

```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]
```

```
DIRECT-ADDRESS-INSERT(T, x)
        T[x.key] = x
DIRECT-ADDRESS-DELETE(T, x)
        T[x.key] = NIL
```

Іноді множина реальних ключів визначає комірки в таблиці, які містять покажчики на елементи динамічної множини (рис.1), а іноді елементи можуть зберігатися безпосередньо в таблиці з прямою адресацією (в комірках, що дозволяє економити пам'ять за рахунок відмови від зберігання ключів та супутніх даних елементів в об’єктах, зовнішніх по відношенню до таблиці, та від покажчиків на ці об’єкти у таблиці, але для позначення пустої комірки користуються спеціальним значенням ключа). Якщо елементи можуть зберігатися безпосередньо в таблиці з прямою адресацією, то часто збереження ключа не є необхідним, так як знаючи індекс об’єкта у таблиці будемо знати й ключ.

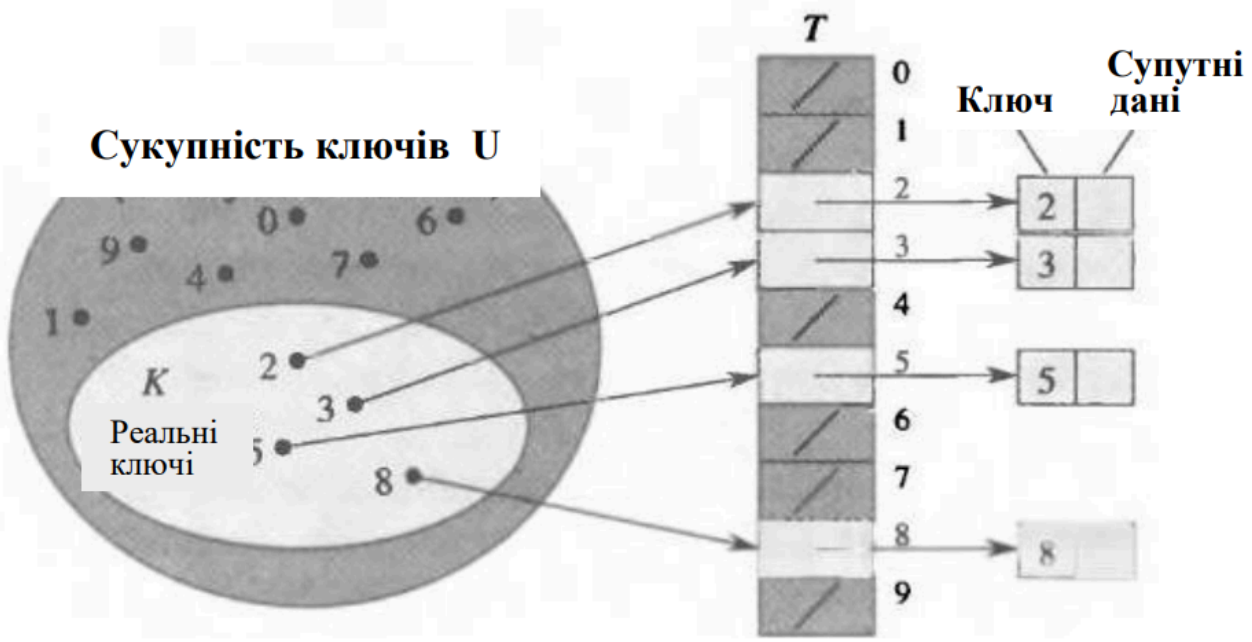


Рис.1 Приклад таблиці з прямою адресацією при створенні словника

Але використання таблиці з прямою адресацією має явний недолік: для вже доволі великої сукупності ключів U зберігання таблиці T буде невигідним.

Використання *хеш-таблиці* дозволяє зменшити вимоги до об’єму пам’яті до $O(|K|)$ та зберегти час пошуку елементу як O(1) у середньому випадку.

Для випадку використання таблиці прямої адресації елемент з ключом k зберігається у комірці k, при використанні хешування – у комірці $h(k)$ (за рахунок використання *хеш-функції* h, що обчислює комірку для заданого ключа k).

Хеш-функція – це така функція $h: U \to \{0, 1, \ldots, m-1\}$, що відображає сукупність ключів U на комірки хеш-таблиці $T[0..m-1]$. Кажуть, що елемент з ключем k хешується в комірку $h(k)$, а величину $h(k)$ називають *хеш-значенням* ключа k. Звичайно розмір хеш-таблиці значно менший U . Хеш-функція

зменшує робочий діапазон індексів масиву і тому замість розміру U значень обходимося масивом розміром *m* (рис. 2)

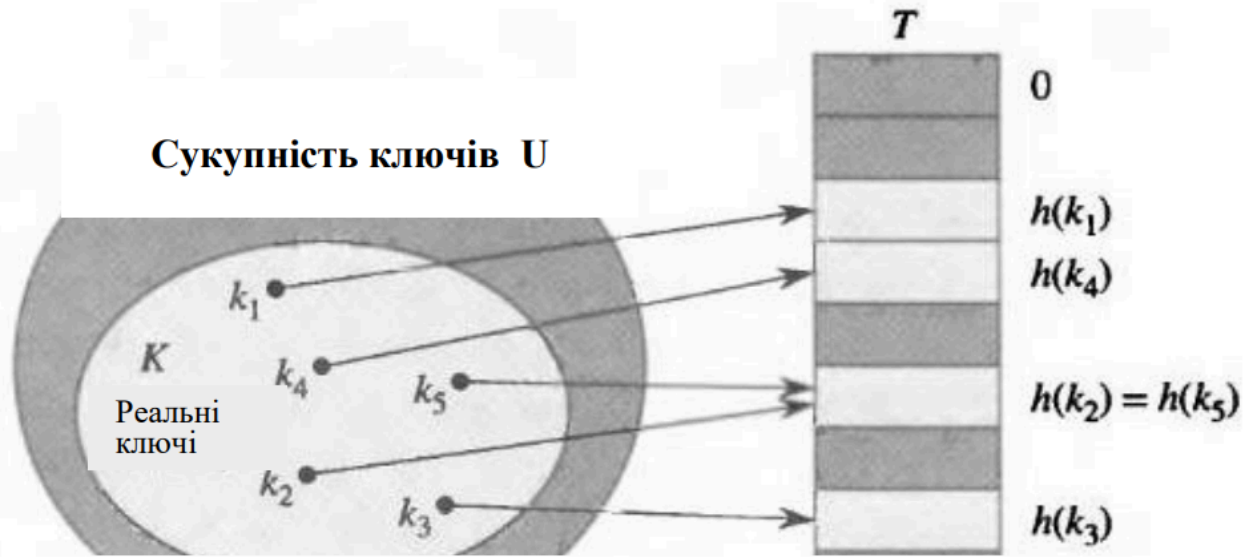


Рис.2 Хешування до хеш-таблиці

Ситуацію, коли два ключа можуть бути хешовані в одну комірку називають *колізією*. Є різні технології вирішення колізій. За допомогою підходящого вибору хеш-функції намагаються їх мінімізувати. Але так як $|U| > m$, то повинно існувати не менше двох ключів, що мають однакове хеш-значення.

2.Відкрите хешування (хешуванням з роздільними ланцюжками)

Найпростішим методом вирішення колізій є метод ланцюжків (separate chaining), за яким розташовують всі елементи, що хешовані до одної комірки, у зв'язаний (дво- або однозв'язний) список. Це відкрите (зовнішнє) хешування. Тоді, наприклад, комірка *j* містить вказівник на заголовок списку всіх елементів, хеш-значення яких дорівнює *j*, а якщо таких елементів немає, комірка містить значення NIL.
CHAINED-HASH-SEARCH(*T*, *k*)
Пошук елементу з ключом *k* у списку *T*[*h*(*k*)]
CHAINED-HASH-INSERT(*T*, *x*)
Вставка x до заголовку списку *T*[*h*(*x*, *key*)]
CHAINED-HASH-DELETE(*T*, *x*)
Видалення *x* із списку *T*[*h*(*x*, *key*)]

Припустимо, що елемент, що вставляємо до таблиці, відсутній в ній. Тоді час, необхідний для вставки – O(1). Інакше перевіряємо наявність, що тягне за собою додаткову вартість виконання пошуку елементу з відповідним ключом перед вставкою. Тоді час роботи буде пропорційним довжині списку. Вилучення елементу матиме вартість таку ж вартість часову вартість.
Проведемо більш докладний аналіз хешування з ланцюжками. Припустимо, маємо хеш-таблицю *T* з *m* комірками, в яких зберігаються *n* елементів. Коефіцієнтом заповнення таблиці α= *n*/*m* (може бути менше, дорівнювати або більше 1) У найгіршому випадку всі *n* ключів можуть бути хешовані в одну

комірку. Тоді маємо список довжиною n. Тому час пошуку для найгіршого випадку буде O(n) плюс час на обчислення хеш-функції, а тоді використання хеш-функції буде невигідне.
У середньому випадку продуктивність хешування залежить від того, як добре хеш-функція розподіляє множину ключів, що зберігається, за m комірками у середньому.

3. Приклад побудови відкритої хеш-таблиці
Розглянемо наступний список слов
A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED
1 2 3 4 5 6 7 8

В якості хеш-функції будемо використовувати просту хеш-функцію для строк – тобто підсумувати номер позиції букви в англійському алфавіті та обчислювати залишок від ділення цієї суми на 13

$A-1$ $J-10$ $S-19$
$B-2$ $K-11$ $T-20$
$C-3$ $L-12$ $U-21$
$D-4$ $M-13$ $V-22$
$E-5$ $N-14$ $W-23$
$F-6$ $O-15$ $X-24$
$G-7$ $P-16$ $Y-25$
$H-8$ $Q-17$ $Z-26$
$I-9$ $R-18$

Таким чином отримаємо
h(A)= 1 mod 13 = 1
h(FOOL) = (6+15+15+12) mod 13=9
h(AND)=6
h(HIS)=10
h(MONEY)=7
h(ARE) = (1+18+5) mod 13= 11
h(SOON) = (19+15+15+14) mod 13= 11
h(PARTED)=12
Тобто:

key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
h(key)	1	9	6	10	7	11	11	12

Тоді побудована хеш-таблиця з роздільними ланцюжками буде виглядати наступним чином:

Таблиця 1

0	1	2	3	4	5	6	7	8	9	10	11	12

Припустимо, що ми хочемо знайти слово KID обчислюємо його хеш-функцію h(KID)= (11+9+4) mod 13=11. Оскільки список, який під'єднано до комірки 11 не є порожнім – то він може містити відповідний ключ. Однак із-за можливих колізій ми не знаємо точної відповіді поки не знайдемо його в списку або не переглянемо список до кінця. Після порівняння KID із ARE, а потім із SOON можемо сказати, що KID в таблиці відсутній.

Додавання та видалення елементів виконується аналогічним чином. Додавання виконується кінець списку, а видалення відбувається шляхом пошуку елемента з подальшим видаленням із списку. Таким чином ефективність роботи пошуку в хеш-таблиці залежить від довжина зв'язних списків, яка в свою чергу залежить від довжини словника, хеш-таблиці та вигляду хеш-функції.

4. Закрите хешування (хешування з відкритою адресацією)

При використанні цього методу безпосередньо в хеш-таблиці зберігаються всі елементи, а якщо комірка пуста, то вона має містити значення NIL.

Тому таблиця може стати повністю заповненою (коефіцієнт заповнення α не може перевищувати 1) і елемент вставити буде неможливо. При виконанні пошуку елементу перевіряються всі комірки таблиці, доки не буде знайдено шуканого елементу або впевнимося у відсутності елементу в хеш-таблиці. Перевагою методу є відмова від покажчиків (економія пам'яті на них дозволяє збільшити розмір таблиці).

Використовують методику повторного хешування. За нею для виконання вставки елементу послідовно перевіряють (досліджують) комірки таблиці, доки не знайдеться пуста комірка, в яку й розміщують ключ. Тобто, замість фіксованого порядку дослідження комірок 0, 1, …, m – 1 (що дає найгірший час O(m)), послідовність досліджуваних комірок залежить від ключа, що вставляють до хеш-таблиці.

З цією метою використовується *розширене* визначення хеш-функції, де в якості другого аргументу включений номер дослідження (який починається з 0):

$$h: U \times \{0,1,...,m\text{-}1\} \rightarrow \{0,1,...,m\text{-}1\}.$$

В методі відкритої адресації потрібно, щоб для кожного ключа k послідовність досліджень $h(k,0), h(k,1), ..., h(k, m\text{-}1)$ являла собою перестановку множини 0,1,...,m-1 такої, щоб могли бути переглянутими всі комірки таблиці.

Для випадку, коли елементи в хеш-таблиці T є ключами без супутньої інформації й ключ k тотожній елементу, що містить ключ k, процедура вставки ключа k може мати наступний вигляд:

```
HASH-INSERT(T, k)
1 i = 0
2       repeat
```

```
3               j = h(k, i)
4               if T[j] == NIL
5                       T[j] = k
6               return j
7               else i = i + 1
8       until i == m
9 error «переповнення хеш-таблиці»
```

У процедурі пошуку ключа k досліджується та ж послідовність комірок, що й при його вставці. Але, якщо зустрічається пуста комірка, то пошук завершується невдачею (якщо припустимо, що елементи не видалялися):

```
HASH-SEARCH(T, k)
1 i = 0
2       repeat
3               j = h(k, i)
4               if T[j] == k
5               return j
6               else i = i + 1
7       until T[j] == NIL або i == m
8       return NIL
```

Процедура видалення з хеш-таблиці більш складна. При вилученні ключа з комірки і хеш-таблиці її не можна просто помітити пустим значенням NIL(це негативно впливає на пошук елементу), таку комірку помічають спеціальним значенням DELETED. Крім того, потрібно дещо змінити наведену процедуру вставки, щоб в ній комірка із значенням DELETED розглядалася як пуста. Але час пошуку тоді перестає залежати від коефіцієнта заповнення α і тому при необхідності вилучень з таблиці користуються методом ланцюжків в якості методу вирішення колізій.

5. Приклад побудови закритої хеш-таблиці

В випадку закритого хешування всі ключі зберігаються безпосередньо в хеш-таблиці, що призводить до вимоги про розмір хеш-таблиці, який повинен дорівнювати кількості ключів. Для вирішення колізій використовують лінійне дослідження (linear probing) коли в випадку виникнення колізій перевіряються всі комірки одна за одною, поки елемент не буде знайдено. Якщо в процесі перевірки досягається кінець таблиці то пошук переходить до першої комірки таблиці, яку розглядаємо як циклічний масив.

Для попереднього прикладу

key	a	fool	and	his	money	are	soon	parted
h(key)	1	9	6	10	7	11	11	12

побудова хеш-таблиці з відкритою адресацією виглядає наступним чином:

Таблиця 2

	хеш-таблиця з відкритою адресацією												
Insert key	0	1	2	3	4	5	6	7	8	9	10	11	12
1		a											

2		a								fool			
3		a					and			fool			
4		a					and			fool	his		
5		a					and	money		fool	his		
6		a					and	money		fool	his	are	
7		a					and	money		fool	his	are	
7		a					and	money		fool	his	are	soon
8		a					and	money		fool	his	are	soon
9	parted	a					and	money		fool	his	are	soon

Знайдемо слово LIT в таблиці. Розраховуємо хеш-функцію h(LIT) = (12+9+20) mod 13= 2. Звертаємось до комірки за номером 2 – вона порожня, тобто слово відсутнє. В випадку пошуку слова KID обчислюємо його хеш-функцію h(KID)= (11+9+4) mod 13=11. Звертаємось до комірки 11 (ARE), ключ не співпадає, звертаємось до наступної комірки 12 (SOON), не має співпадіння, циклічно переходимо до комірки 0 (PARTED) не має, порівнюємо із 1 (A) і тільки коли потрапляємо до порожньої комірки робимо висновок про відсутність слова KID.
Операція видалення є більш складною. Якщо необхідно видалити ключ ARE (11) – комірка стає порожньою, то ми втратимо доступ ключа SOON (12). Тому необхідно «відкладене видалення», тобто комірку 11 в випадку зі ключом ARE відмічають спеціальним чином (DELETED), щоб відрізнити від комірок, які взагалі не були зайняти.

6. Хеш-функції. Побудова некриптографічних хеш-функцій

При побудові якісних хеш-функцій дуже корисною є інформація про розподіл ключів. Вірним підходом при побудові є підбір хеш-функції так, щоб вона не корелювала із закономірностями, яким можуть підкорятися існуючі дані. Іноді до хеш-функцій можуть висувати більш суворі вимоги, ніж за простого рівномірного хешування.
Нехай сукупність ключів можна представити множиною цілих невід'ємних чисел N.
1. *Побудова хеш-функції методом ділення*: $h(k) = k \bmod m$ (відображення ключа k до одної з m комірок за допомогою вказаного *обчислення залишку*). Таке хешування доволі швидке. Але необхідно уникати деяких значень m (не повинно бути степенем 2, так як для $m = 2^p$ значення $h(k)$ буде просто p молодших бітів числа k, або 2^{p-1}). Часто m вибирають простим, досить далеким від числа, що є ступенем 2.
2. *Побудова хеш-функції методом множення*: $h(k)=\lfloor m(kA \bmod 1)\rfloor$, $0 < A<1$. Побудова хеш-функції методом множення виконується в два етапи. Спочатку ми множимо ключ k на константу A й одержуємо дробову частину отриманого добутку. Потім ми множимо отримане значення на m і застосовуємо до нього функцію заокруглення вниз тобто,

$$h(k)=\lfloor m(kA \bmod 1)\rfloor$$

де вираз « kA mod 1» означає одержання дробової частини добутку k, тобто величину $kA - \lfloor kA \rfloor$. Перевага методу множення полягає в тому, що значення m може бути довільним, часто вибирають $m = 2^p$, де p – деяке натуральне, але деякі значення константи A дають кращі результати у порівнянні з іншими. Оптимальне значення константи A залежить від характеристик даних, що хешуються Наприклад. Дональд Кнут запропонував використовувати значення $A \approx (\sqrt{5} - 1)/2 \approx 0.6180339887$, що дає непогані результати

Завдання:

Для вхідних даних та хеш-функції по варіантам завданій:

- додайте елементи до відкритої та закритої та хеш-таблиці (див. таблиці 1 та 2)
- знайдіть кількість порівнянь при пошуку будь-якого елемента із побудованих хеш-таблиць
- вкажіть елемент, для пошуку якого необхідна максимальна кількість порівнянь у кожній хеш-таблиці
- підрахуйте середню кількість порівнянь для пошуку елемента в кожній хеш-таблиці
- вкажіть елемент для видалення якого необхідна максимальна кількість порівнянь та наведіть опис процедуру його видалення із кожної хеш-таблиці

Варіанти завдань:

Варіант	Послідовність	Хеш-фунція
1	53, 12, 68, 63, 3, 47, 50, 61, 41	h (K) =K mod 11
2	98, 40, 42, 88, 61, 87, 79, 97, 82	h (K) =K mod 13
3	70, 80, 61, 92, 12, 23, 67, 65, 50	h (K) =K mod 10
4	29, 1, 24, 69, 52, 97, 27, 10, 88	h (K) =K mod 11
5	95, 43, 98, 41, 68, 67, 10, 7, 69	h (K) =K mod 13
6	61, 32, 27, 45, 75, 58, 5, 50, 99	h (K) =K mod 10
7	78, 77, 4, 62, 8, 69, 46, 11, 49	h (K) =K mod 11
8	28, 76, 27, 10, 5, 35, 95, 16, 33	h (K) =K mod 13
9	31, 40, 22, 50, 53, 68, 97, 12, 15	h (K) =K mod 10
10	18, 20, 2, 88, 61, 17, 79, 97, 82	h (K) =K mod 11
11	41, 52, 47, 65, 95, 38, 15, 50, 99	h (K) =K mod 13
12	11, 42, 67, 55, 65, 78, 25, 50, 69	h (K) =K mod 10
13	53, 5, 44, 47, 35, 83, 82, 85, 28	h (K) =K mod 11
14	77, 89, 74, 68, 70, 49, 5, 62, 51	h (K) =K mod 13
15	19, 75, 43, 31, 5, 66, 62, 34, 76	h (K) =K mod 10
16	50, 57, 78, 34, 41, 68, 47, 61, 38	h (K) =K mod 11
17	86, 36, 14, 50, 64, 21, 2, 83, 82	h (K) =K mod 13
18	80, 27, 37, 36, 91, 53, 86, 66, 98	h (K) =K mod 10
19	21, 44, 22, 50, 63, 68, 97, 12, 15	h (K) =K mod 11
20	7, 89, 4, 68, 70, 49, 10, 62, 51	h (K) =K mod 13

21	54, 65, 7, 33, 86, 29, 11, 91, 12	h (K) =K mod 10
22	50, 80, 19, 86, 35, 7, 60, 48, 51	h (K) =K mod 11
23	10, 90, 95, 30, 45, 60, 57, 28, 5	h (K) =K mod 13
24	90, 10, 15, 80, 100, 6, 57, 5, 29	h (K) =K mod 10
25	53, 100, 44, 74, 53, 38, 82, 65, 28	h (K) =K mod 11
26	47, 50, 61, 41,53, 12, 68, 63, 3	h (K) =K mod 13
27	87, 79, 97, 82,98, 40, 42, 88, 61	h (K) =K mod 10
28	12, 23, 67, 65, 50, 70, 80, 61, 92	h (K) =K mod 11
29	69, 52, 97, 27, 10, 88,29, 1, 24	h (K) =K mod 13
30	41, 68, 67, 10, 7, 69, 95, 43, 98	h (K) =K mod 10
31	58, 5, 50, 99, 61, 32, 27, 45, 75	h (K) =K mod 11
32	46, 11, 49, 78, 77, 4, 62, 8, 69	h (K) =K mod 13
33	35, 95, 16, 33, 28, 76, 27, 10, 5	h (K) =K mod 10
34	68, 97, 12, 15, 31, 40, 22, 50, 53	h (K) =K mod 11
35	79, 97, 82, 18, 20, 2, 88, 61, 17	h (K) =K mod 13
36	38, 15, 50, 99, 41, 52, 47, 65, 95	h (K) =K mod 10

Звіт повинен містити: Контрольний приклад за варіантами завдань та висновки щодо практичної роботи №7

Список рекомендованої літератури

1. Data Structures and Algorithms [Електронний ресурс], Автори: *Alfred V. Aho*, Bell Laboratories, *Murray Hill*, New Jersey *John E. Hopcroft*, Cornell University, Ithaca, New York, *Jeffrey D. Ullman*, Stanford University, Stanford, California, Режим доступу: https://dokumen.tips/documents/data-structures-and-algorithms-alfred-v-aho-john-e-hopcroft-and-jeffrey-d-ullman.html?page=1
2. Горлова Т.М. Теорія алгоритмів. [Електронний ресурс]: конспект лекцій для студентів напряму підготовки 6.050101 «Комп'ютерні науки» денної та заочної форм навчання / Т.М. Горлова, К.Є. Бобрівник, Н.В. Ліманська – К.:НУХТ, 2015. – 95 с. Режим доступу: http://library.nuft.edu.ua/ebook/file/M51.21.pdf
3. Коваль В.С., Струбицький П.Р. Алгоритми і структури даних. – Навчальний посібник Тернопіль: ФОП Шпак В. Б. – 2017. – 74 с. Режим доступу: http://dspace.wunu.edu.ua/bitstream/316497/41016/1/%D0%BD%D0%B0%D0%B2%D1%87.%D0%BF%D0%BE%D1%81%D1%96%D0%B1.pdf
4. Algorithms and Data Structures by Niklaus Wirth [Електронний ресурс] Автор Вирт Н. Режим доступу https://people.inf.ethz.ch/wirth/AD.pdf