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

Теорія алгоритмів

МЕТОДИЧНІ ВКАЗІВКИ
ДО КОНТРОЛЬНИХ РОБІТ ДЛЯ СТУДЕНТІВ ІНСТИТУТУ ДИСТАНЦІЙНОЇ ТА ЗАОЧНОЇ ОСВІТИ

(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 КОНТРОЛЬНИХ РОБІТ ДЛЯ СТУДЕНТІВ ІНСТИТУТУ ДИСТАНЦІЙНОЇ ТА ЗАОЧНОЇ ОСВІТИ

(спеціальність 122 – «Комп'ютерні науки»)

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

Одеса НУОП 2024

Методичні вказівки до контрольних робіт для здобувачів Інституту дистанційної та заочної освіти з дисципліни «Теорія алгоритмів» (спеціальність 122 – «Комп'ютерні науки») / Укладач: О.О.Арсірій, С.Ю. Смик – Одеса: НУОП, 2024. – 64 с.

Укладач: Арсірій Олена Олександрівна,

к.т.н., доцент

Смик Сергій Юрійович,

к.т.н., доцент

ЗАГАЛЬНІ ПОЛОЖЕННЯ

Мета вивчення дисципліни формування у здобувачів вищої освіти комплексу теоретичних і практичних знань, умінь і навичок з теорії алгоритмів для застосування у професійній діяльності. Програма також спрямована на формування компетентностей, важливих для особистісного розвитку фахівців та їхньої конкурентоспроможності на сучасному ринку праці

Завданням навчальної дисципліни є надання студентам систематичних знань з компютерних наук. Засвоєння специфічної фахової термінології, певного мінімуму знань з теорії та практики – неодмінна передумова майбутньої професійної діяльності фахівця в обставинах сучасного економічного життя. І тому вивчення цього інтергрованого курсу є обов'язковим елементом підготовки спеціаліста з вищою освітою незалежно від спеціальності.

Обов'язковим компонетом навчального процесу з дисципліни «теорія алгоритмів» є підготовка студентом контрольної роботи із запропонованої тематики завдань для контрольної роботи.

Метою написання контрольної роботи є формування у студентів вміння самостійно досліджувати питання теорії алгоритмів, які не можуть бути достатньою мірою розглянуті в форматі установчого лекційного заняття і потребують додаткової самостійної роботи з бібліотечним фондом та електронними інформаційними ресурсами.

Завдання – навчити студентів: самостійно відбирати потрібну літературу та джерела.

ВИМОГИ ДО ВИКОНАННЯ КОНТРОЛЬНОЇ РОБОТИ

Контрольна робота має бути виконаною самостійно. За допомогою контрольної роботи студент глибше вивчає найбільш складні проблеми навчальної дисципліни та вчиться правильно її оформлювати.

Підготовка контрольної роботи містить такі етапи:

1) підбір і вивчення спеціальної літератури та нормативно-правових актів;

2) викладення змісту контрольної роботи;

3) оформлення контрольної роботи;

1. Підбір і вивчення спеціальної літератури та нормативно-правових актів

Роботу над контрольною роботою потрібно починати з вивчення стосовно обраної тематики відповідного розділу підручника, навчального посібника, конспектів лекцій. Після того як загальне уявлення про обрану тематику склалося, студенту слід приділити серйозну увагу підбору і вивченню літератури, орієнтовний список якої наведений у методичних вказівках. Однак, запропонований перелік джерел не повинен зв'язувати ініціативу студента. Він може та мусить використати інші роботи, самостійно підібрані внаслідок вивчення бібліографії за обраною проблематикою. Вивчаючи ту чи іншу наукову працю, студент повинен сприймати її крізь призму тих основних проблем, що їх вирішував автор. Без усвідомлення проблеми неможливо виділити головне й істотне, важко відокремити тезу від аргументів і практично неможливо перебороти формальне ставлення до змісту досліджуваної праці.

2. Викладання змісту контрольної роботи

Після підбору і вивчення літератури, слід приступити до узагальнення та систематизації зібраного матеріалу. Виклад матеріалу повинен бути чітким, логічним та послідовним. Викладати матеріал у контрольній роботі рекомендується в безособовій формі висловлювання (наприклад «вважаємо», «думаємо», «рахуємо» та ін.). Необхідно вживати терміни, властиві даній науці, уникати незрозумілих понять та складних граматичних оборотів. Терміни, окремі слова і словосполучення допускається змінювати прийнятими текстовими скороченнями, значення яких зрозуміле з контексту контрольної роботи.

3. Оформлення контрольної роботи

Студент повинен пам'ятати, що через оформлення контрольної роботи, її зовнішній вигляд, викладача формує першу думку про зміст матеріалу. Тому кожному студенту необхідно опанувати техніку й етику оформлення контрольної роботи та дотримуватись стандартних вимог, які висуваються щодо контрольних робіт.

Починається робота з титульного аркуша, який повинен мати: назву міністерства, навчального закладу, кафедри, на якій писалася контрольна робота, номер завдання, прізвище студента, номера групи та факультету, де навчається студент.

Кожна структурна частина роботи повинна починатися з нової сторінки та мати заголовок, який відповідає частині завдання контрольної роботи. Заголовки слід розташовувати посередині рядка і друкувати великими літерами без крапок у кінці, не підкреслюючи. Якщо заголовок складається з двох і більше речень, тоді їх розділяють крапкою. Перенесення слів у заголовку не допускається.

Усі сторінки, починаючи з другої, послідовно нумеруються з проставленням арабських цифр за загальним правилом у нижньому правому куті, без крапки в кінці. Слід мати на увазі, що першою сторінкою

контрольної роботи є титульний аркуш, на якому нумерація сторінки не ставиться, але враховується при нумерації наступної сторінки.

Контрольна робота пишеться чітким, розбірливим почерком, або друкується на одному боці аркуша білого паперу формату А4 (розмір 210х297 мм) через два міжрядкових інтервали для друкарської машинки і півтора – для комп'ютера, з обов'язковим додержанням при цьому такої ширини полів: зверху і знизу - 20 мм, зліва – 25-30 мм, справа – 10 мм. На одній сторінці повинно бути не більше ніж 32-40 рядків.

Загальний обсяг контрольної роботи не повинен перебільшувати 15 друкованих сторінок комп'ютерного тексту через 1,5 інтервал або 12 сторінок шкільного зошиту.

Оформлення списку використаної літератури є важливою складовою написання контрольної роботи. У список включаються тільки ті джерела, які використовувались при написанні контрольної роботи або на які зроблено посилання в самій роботі. Список літератури в загальний обсяг контрольної роботи не включається, але нумерація сторінок продовжується.

Наприклад:

1. Опришко В. Державно-правова реформа в Україні: основні напрямки // Право України. – 1998. – № 1 – С. 27-32.

2. Прокопенко В.І. Трудове право України: Підруч. – 3-тє вид. – Х.: Консум, 2002. – С.528.

Важливе значення має правильне оформлення посилань на джерела та матеріали, які студент використовує при написанні контрольної роботи. Рекомендується такий варінт оформлення посилань: нумерація усіх посилань з визначенням номера джерела даного посилання у списку використаної літератури. У такому разі посилання оформлюються у квадратних дужках з вказанням сторінки.

ПРАВИЛА ВИБОРУ НОМЕРУ ЗАВДАННЯ ДЛЯ КОНТРОЛЬНОЇ РОБОТИ

Номери завдань для контрольної роботи обираються за двома останніми цифрами номера залікової книжки:

1) Якщо останні цифри номера залікової книжки, скажімо, 01, 02, 03 тощо або 12, 15 (по 21 включно), відповідно номери контрольних завдань будуть: 1,2,3 та інші або 12, 15, 21.

2) Якщо останні цифри закінчуються нулем. Тоді нуль відкидається і номер контрольного завдання визначається за двома попередніми цифрами, а також у порядку, передбаченому у 1-ому правилі.

Наприклад:

останні цифри – 120, 130, 150 (включно до 210), відповідно контрольні завдання будуть за номерами – 12, 13, 15, 21.

3) У випадку, коли дві останні цифри або передостанні перед нулем цифри номера залікової книжки становлять більше 21, номер контрольного завдання буде визначатися за сумою двох цифр.

Наприклад: останні цифри номера залікової книжки – 31, 79, 91 або 370, 810 тощо, тоді номери контрольних завдань відповідно будуть: 2+4=6, 7+9=16, 9+1=10, 3+7=10, 8+1=9.

ЗАВДАНЯ ДЛЯ КОНТРОЛЬНОЇ РОБОТИ

Самостійна 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 i 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".

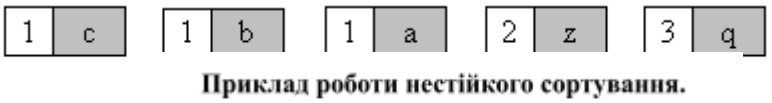


Рис. 3

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

$$A_0 \le A_1 \le \cdots \le A_{i-1} \mid \overbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}$$

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.

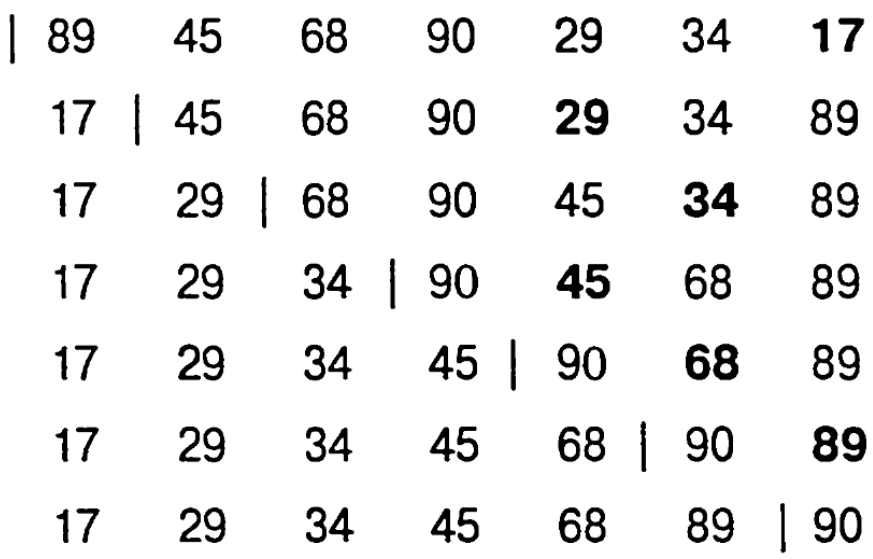


Рис. 5 Приклад сортування вибором

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).

Рис. 6 Приклад сортування в

Таким чином, у процесі вставки ми "просіва[illegible] 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
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

Варіант	Послідовність
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

Самостійна робота №2

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:

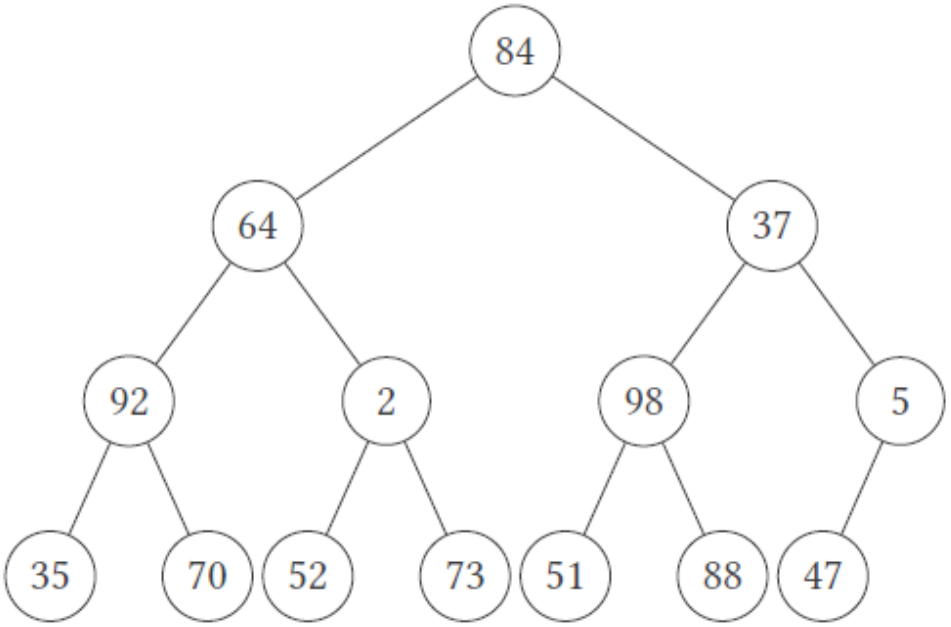
1. Теоретичні відомості про купи
2. Представлення масиву у вигляді максимальної купи
3. Сортування масиву за допомогою купи
4. Приклад моделювання
5. Завдання
6. Оформлення звіту

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 $A[0] \geq A[i]$ 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

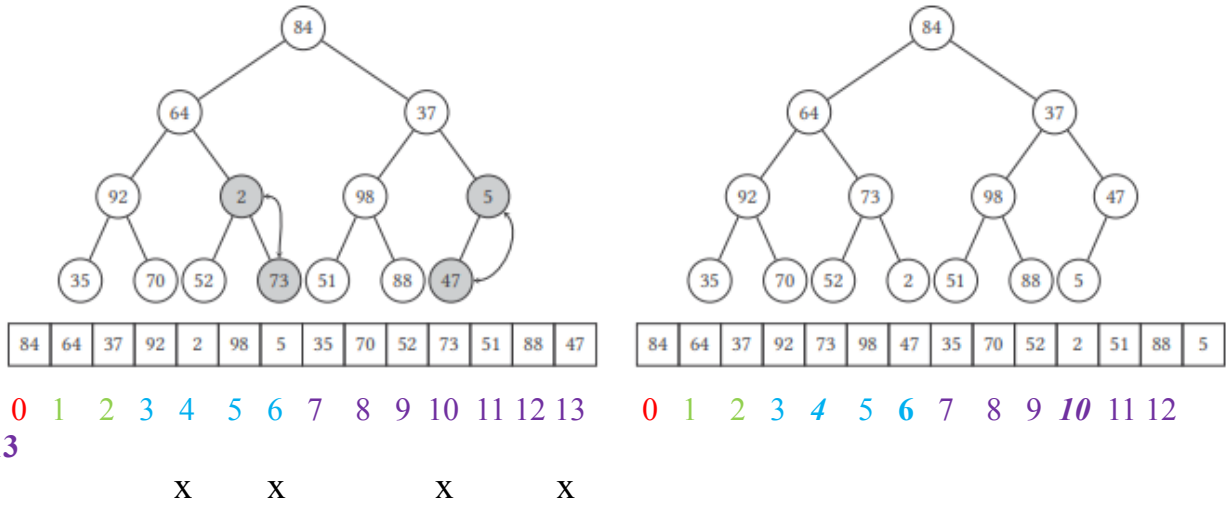
Рис. 1 Представлення елементів масиву у вигляді вершин та ребер дерева.

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

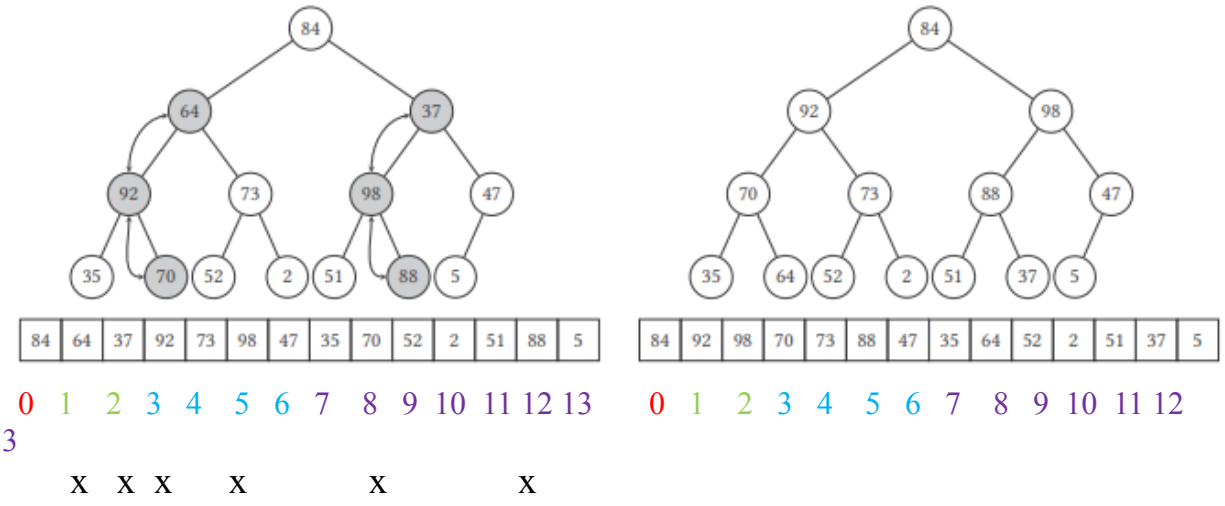
вершини менше, тоді міняємо її місцями із найбільшою з її дочірніх вершин. Коли ми закінчуємо з рівнем h – 1, знаємо, що для всіх вузлів даного рівня у нас A[i] ≥ A[2i + 1] та A[i] ≥ A[2i + 2] (Рис.2). Сірим кольором відзначені вершини, які поміняли місцями. Дерево, яке ми отримали у результаті, зображено праворуч. Під кожним деревом знаходиться масив A, в якому відбуваються справжні перестановки, тому що насправді існує лише масив, а дерево – лише наочна візуалізація.



x x x x

Рис.2 Візуалізація створення максимальної купи на другому рівні.

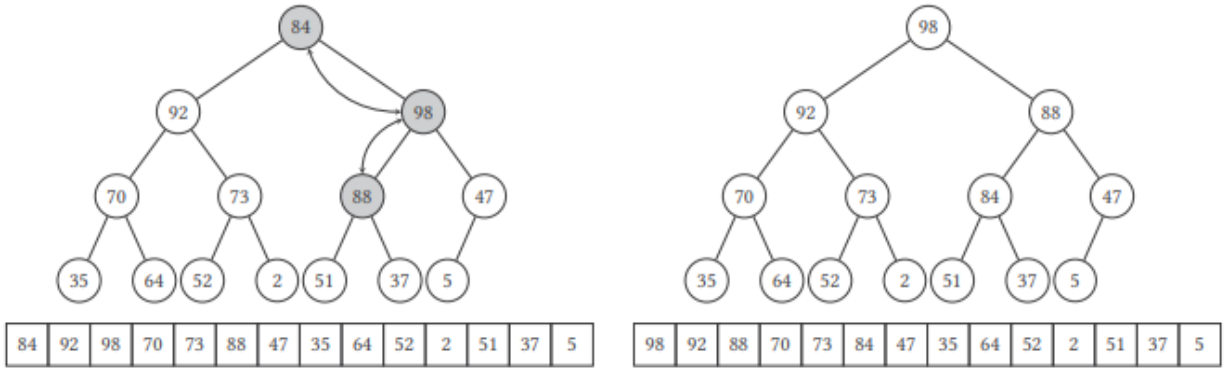
Далі повторюємо ту саму операцію з рівнем *h – 2*, проте цього разу, якщо ми здійснюємо заміну, потрібно перевірити, що відбувається з нижчим рівнем, тобто з *h – 1* рівнем (Рис. 3). Змінюємо місцями вершини 37 і 98. Якщо ми їх змінюємо, то дізнаємося, що 37 мають дочірні вершини, 51 і 88, які не відповідають 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.



x x x x x x

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).



x x x

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.

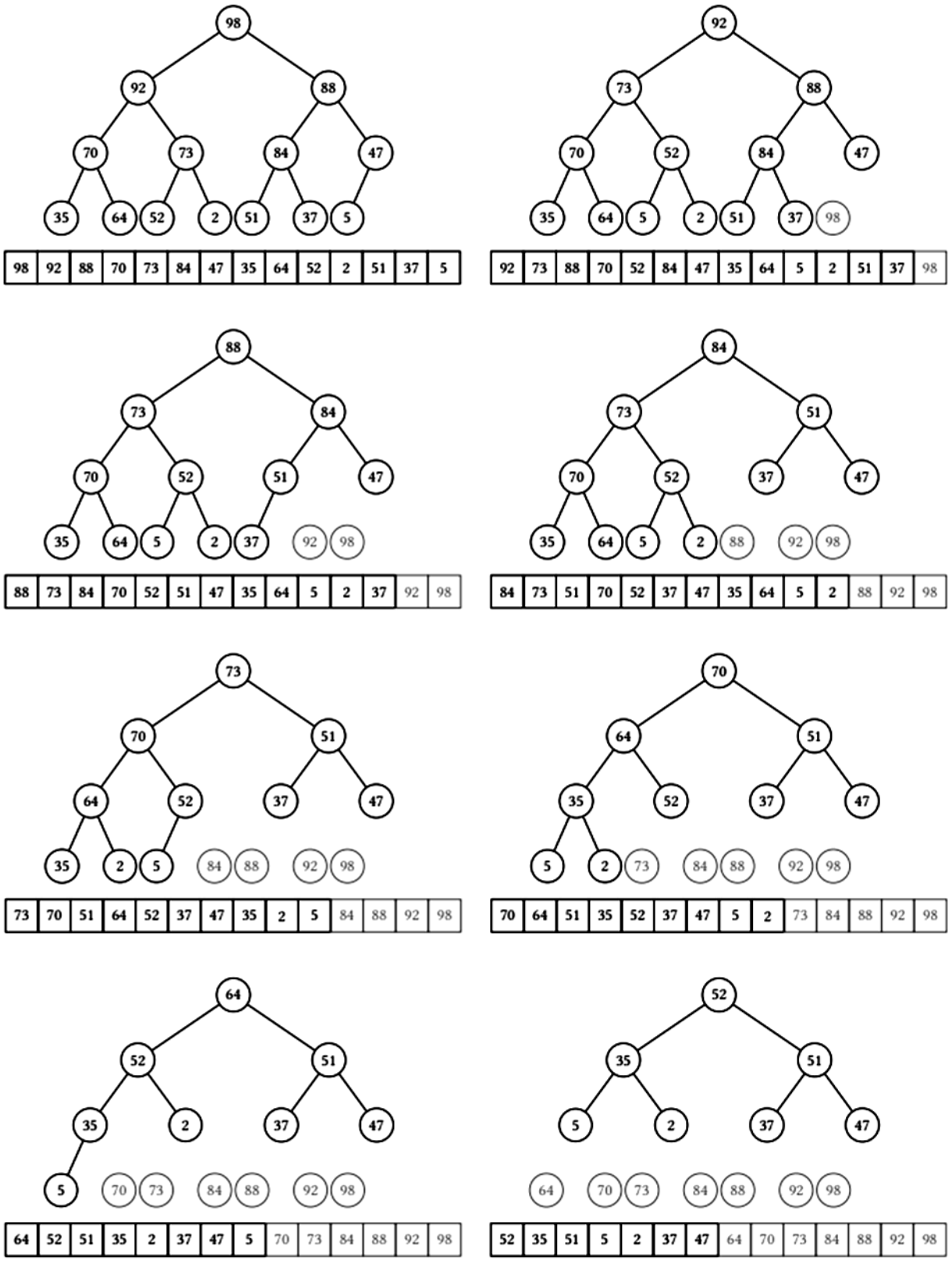


Рис.5 Візуалізація другої фази сортування купою (занурення).

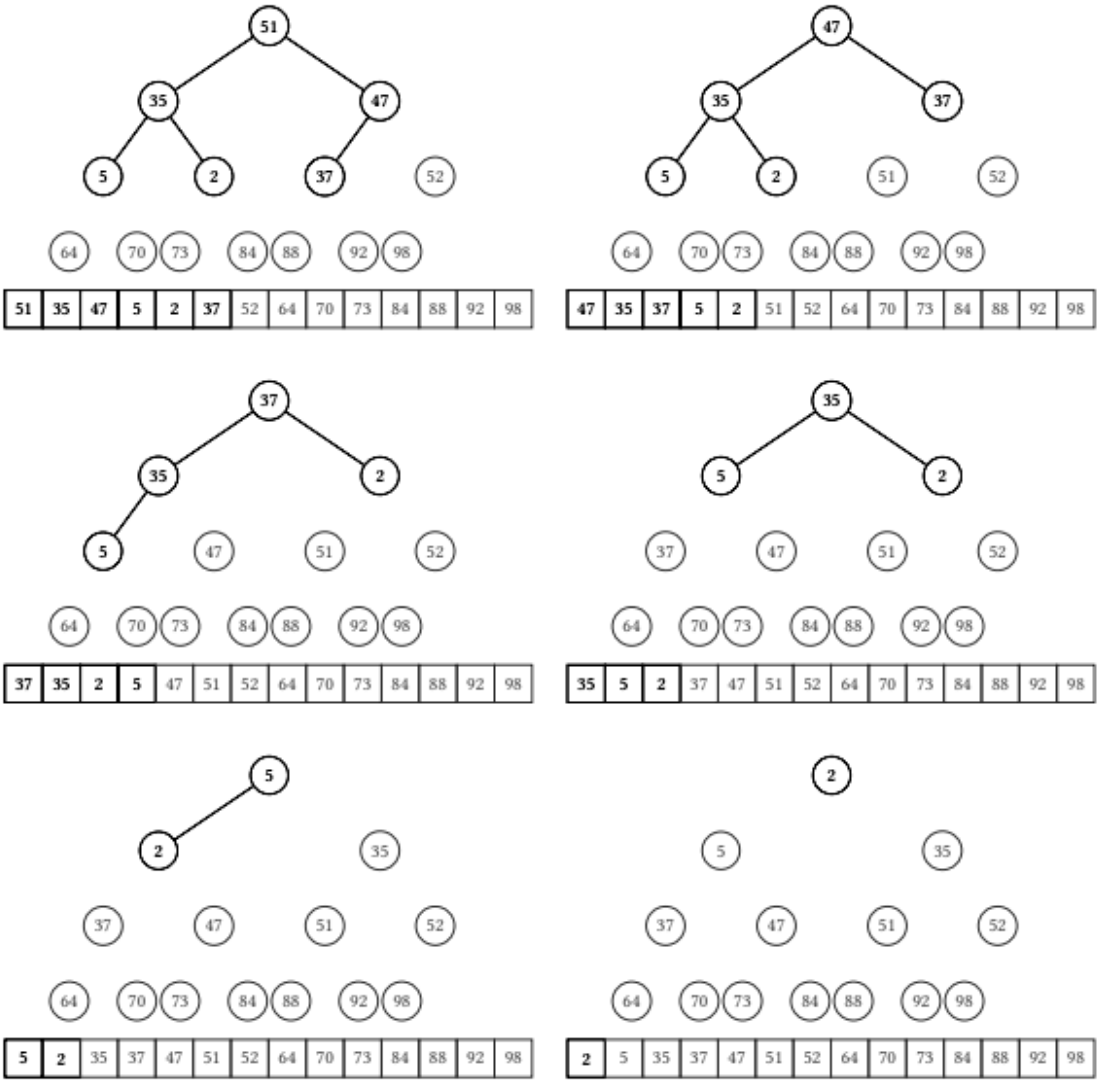


Рис.6 Продовження візуалізації другої фази сортування купою.

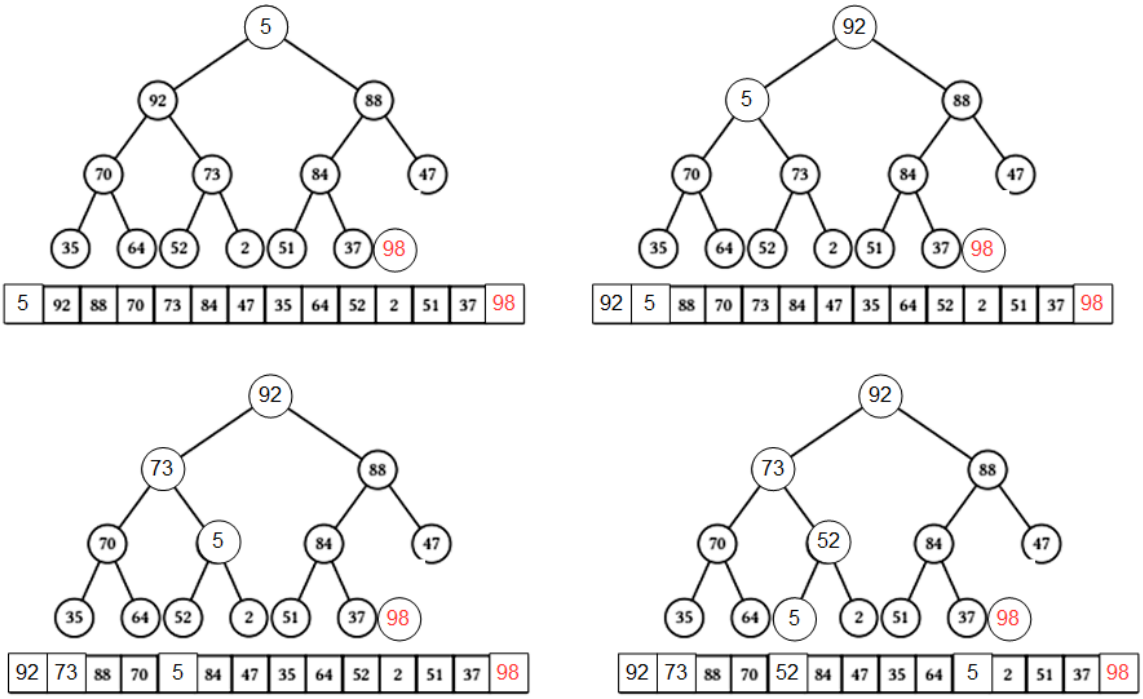


Рис.7 Побудова максимальної купи шляхом занурення для n – 1 елемента.

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

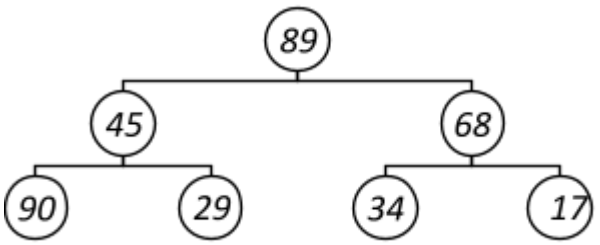
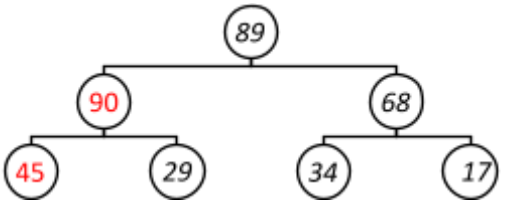
```
HeapSort(A)
// Вхідні дані: не впорядкований масив A
//Вихідні дані: упорядкований масив A
n ← |A|
for i ← (n /2 – 1)  to −1 do                    перша фаза  - побудова максимальної купи
        Sink(A, i, n)
while n > 0 do                                  друга фаза
        Swap(A[0], A[n − 1])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

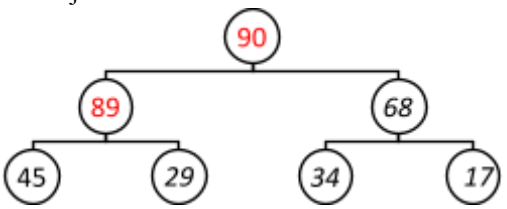
Sink(A, i, n)

```
// 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])
```

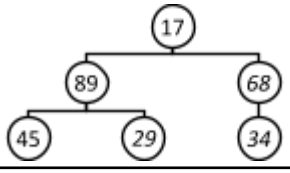
k ← j
j ← 2k + 1

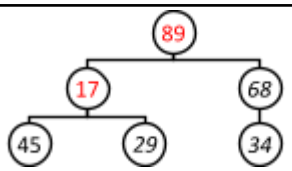
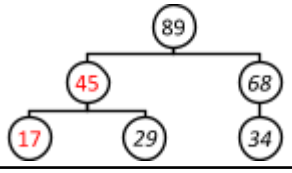
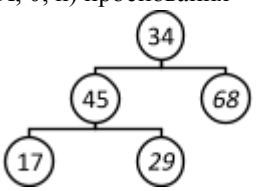
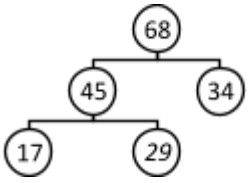
Таблиця 1

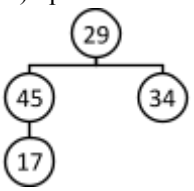
A=[89$_0$, 45$_1$, 68$_2$, 90$_3$, 29$_4$, 34$_5$, 17$_6$] *0 1 2 3 4 5 6*	
n = 7 for i = (7/2 - 1) 2 to -1 do Sink (A, 2, 7)	HeapSort(A) n ← |A| for i ← (n/2 − 1) to −1 do Sink(A, i, n)
k = i (2); placed = false; j = 2k + 1 (5) while not placed and 5 < 7 True	k = i placed ← false j ← 2k + 1 while not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

0 1 2 3 4 5	
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 друга фаза Swap(A[0], A[n − 1]) n ← n − 1
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 else Swap (A[0] (34), A[2] (68)) k = j (2); j = 2k + 1 (5)	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 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	while n > 0 do друга фаза Swap(A[0], A[n − 1]) n ← n − 1

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

Варіант	Послідовність
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

Самостійна робота №3

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

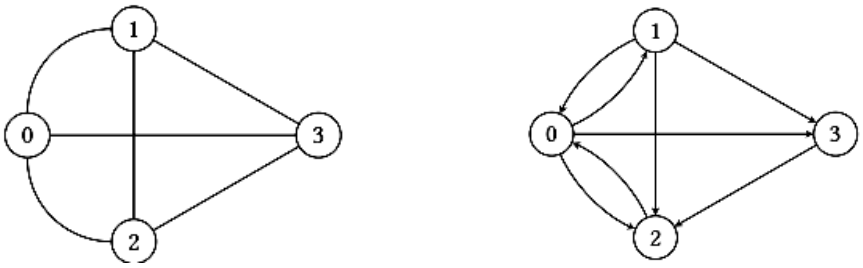
1. Представлення графів, обходи графів.

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 *орграфами*. 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>}

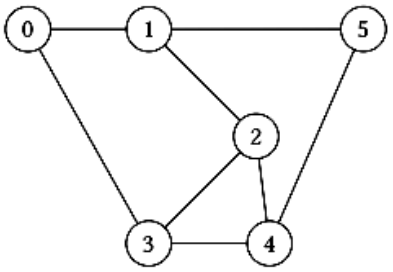
При розробці алгоритмів з використанням структур даних у вигляді графів їх представляють двома способами:

— 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.

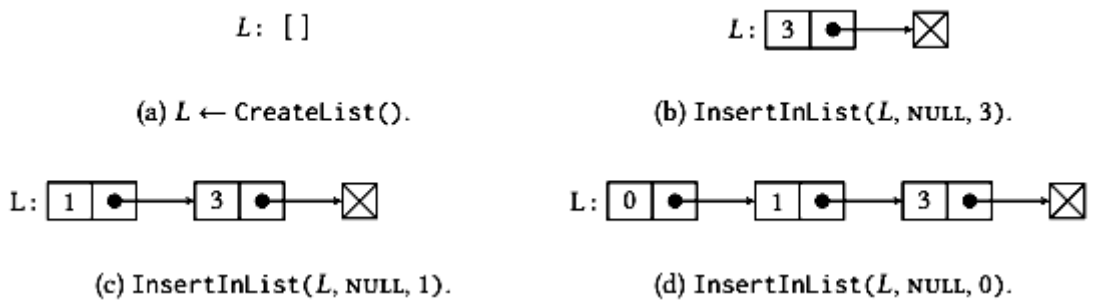
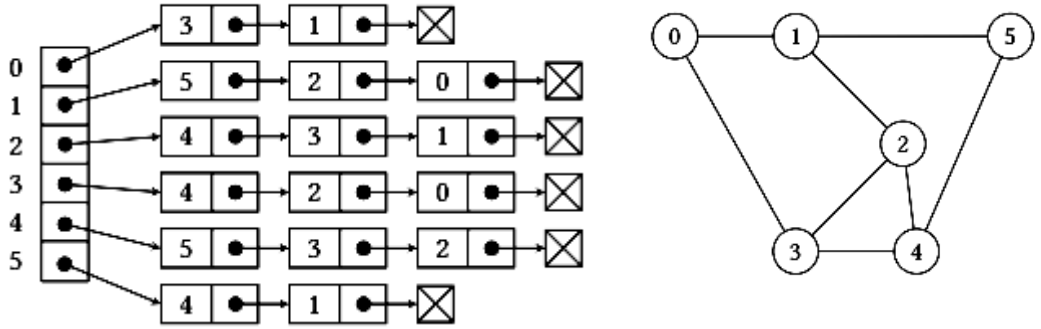


Рис. 3 Додавання вузлів на початок списку

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)

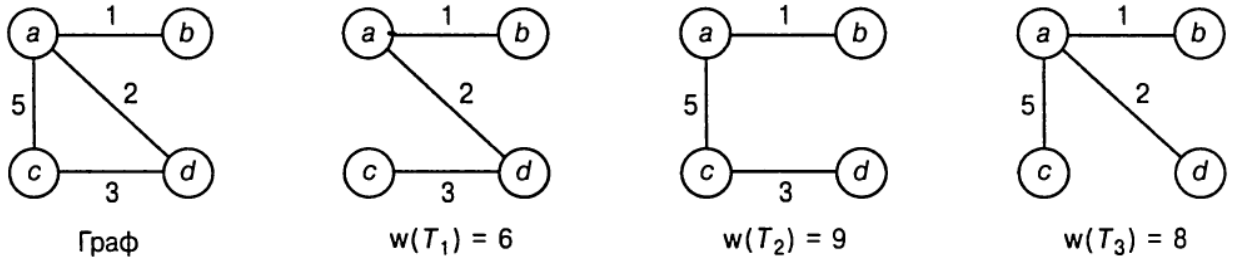


Рис. 5. Граф та його кістякові дерева. T_1 – мінімальне кістякове дерево

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" (рос. пограничных) 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:

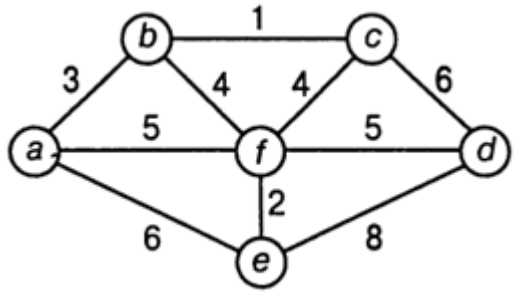
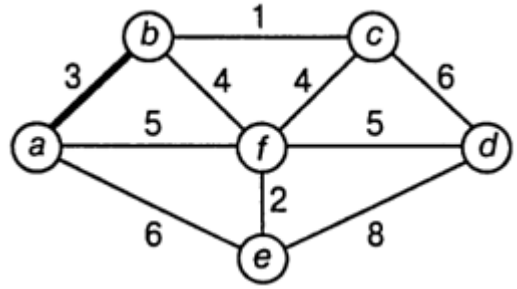
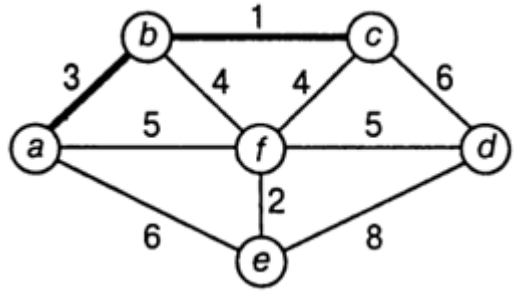
V_T – переглянуті вершини

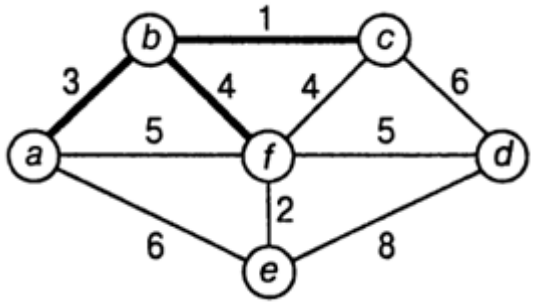
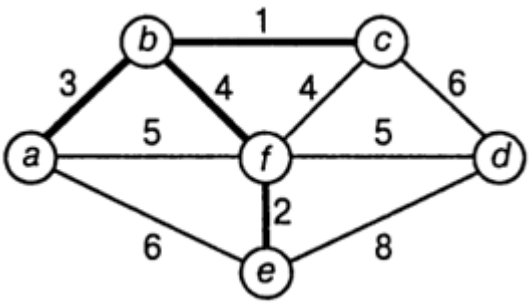
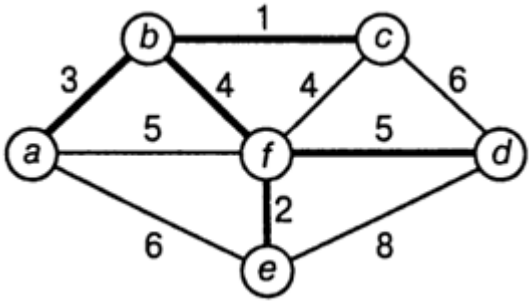
E_T, множина ребер, що становлять мінімальне кістякове дерево графа G

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}
	{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}

Зображення	V_T	E_T	E	V - V_T
	{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:

```
#include <iostream>
#include <cstring>
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

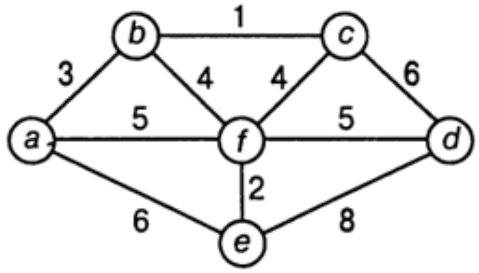
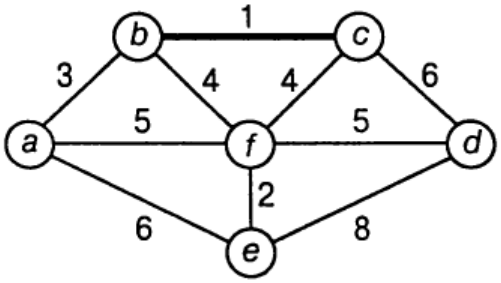
Алгоритм Кру(а)скала (Kruskal's algorithm) також відносится до класу «жадібних». Джозеф Крускал (Joseph Kruskal) запропонував его, будучи студентом-другокурсником. Алгоритм Крускала шукає мінімальне кістякове дерево зваженого зв'язного графа $G = (V, E)$ як ациклічний підграф з |V| - 1 ребрами, сума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

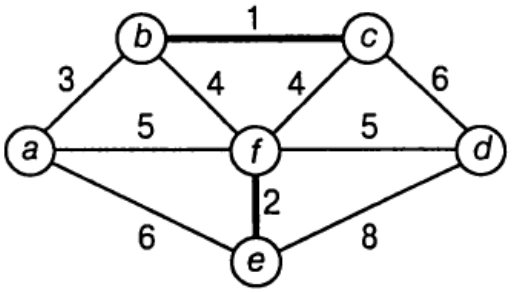
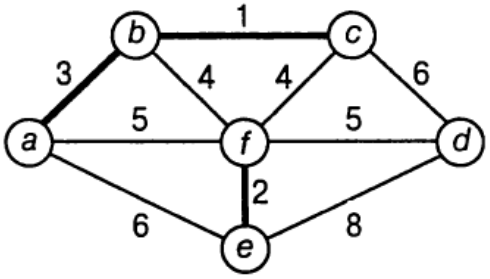
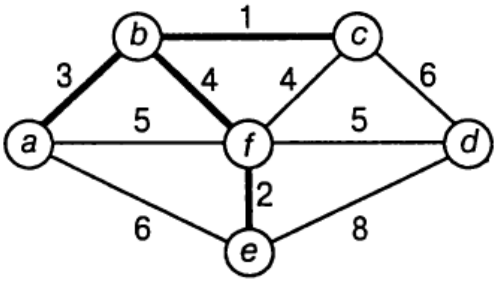
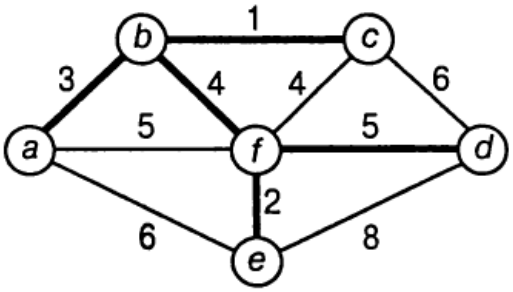
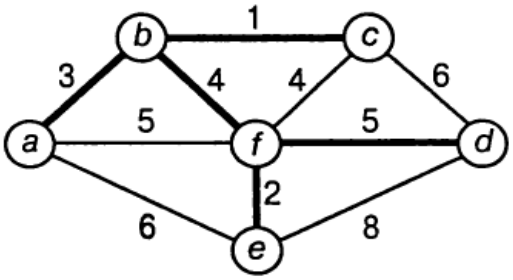
Алгоритм Kruskal (G)

// 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
$e^*_{i_0} \leq ... e^*_{i_{|E|-1}}$
$E_T \leftarrow \emptyset$
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

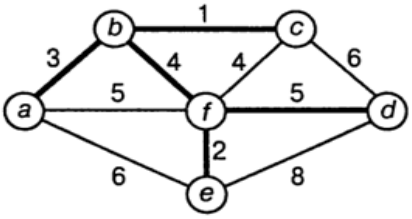
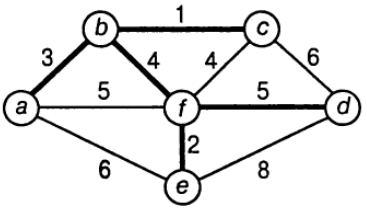
Таблиця 2

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
```

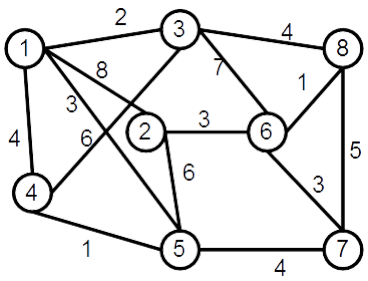
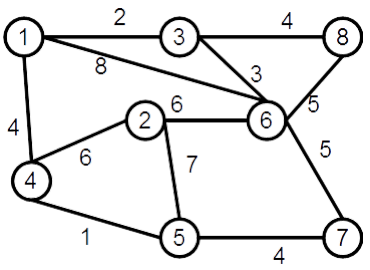
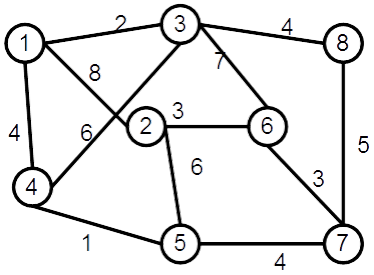
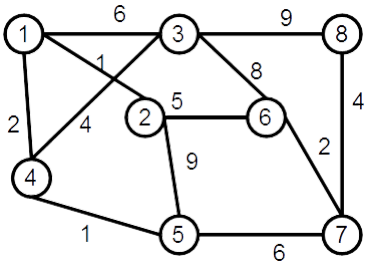
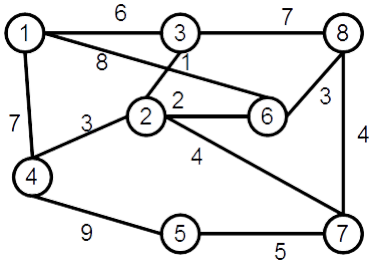
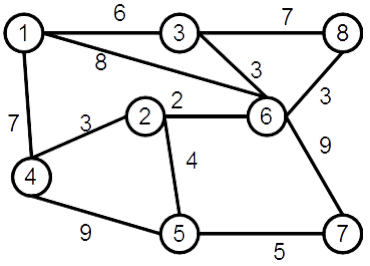
```
        return find_set(parent[i]);
}
void Graph::union_set(int u, int v) {
    parent[u] = parent[v];
}
void Graph::kruskal() {
    int i, uRep, vRep;
    sort(G.begin(), G.end()); // increasing weight
    for (i = 0; i < G.size(); i++) {
        uRep = find_set(G[i].second.first);
        vRep = find_set(G[i].second.second);
        if (uRep != vRep) {
            T.push_back(G[i]); // add to tree
            union_set(uRep, vRep);
        }
    }
}
void Graph::print() {
    cout << "Edge :" << " Weight" << endl;
    for (int i = 0; i < T.size(); i++) {
        cout << T[i].second.first << " - " << T[i].second.second << " : "
                << T[i].first;
        cout << endl;
    }
}
int main() {
    Graph g(6);
    g.AddWeightedEdge(0, 1, 3);
    g.AddWeightedEdge(0, 4, 6);
    g.AddWeightedEdge(0, 5, 5);
    g.AddWeightedEdge(1, 0, 3);
    g.AddWeightedEdge(1, 2, 1);
    g.AddWeightedEdge(1, 5, 4);
    g.AddWeightedEdge(2, 1, 1);
    g.AddWeightedEdge(2, 3, 6);
    g.AddWeightedEdge(2, 5, 4);
    g.AddWeightedEdge(3, 2, 6);
    g.AddWeightedEdge(3, 4, 8);
    g.AddWeightedEdge(3, 5, 5);
    g.AddWeightedEdge(4, 2, 4);
    g.AddWeightedEdge(4, 0, 6);
    g.AddWeightedEdge(4, 2, 8);
    g.AddWeightedEdge(4, 5, 2);
    g.AddWeightedEdge(5, 0, 5);
    g.AddWeightedEdge(5, 1, 4);
    g.AddWeightedEdge(5, 2, 4);
    g.AddWeightedEdge(5, 3, 5);
    g.AddWeightedEdge(5, 4, 2);
    g.kruskal();
    g.print();
    return 0;
}
```

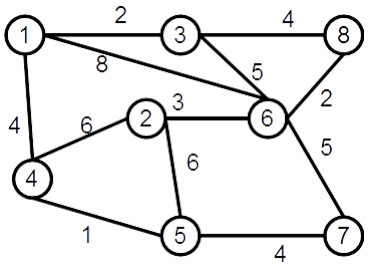
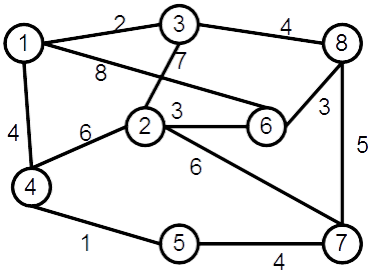
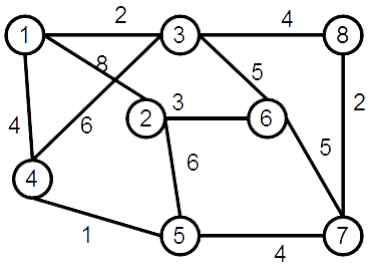
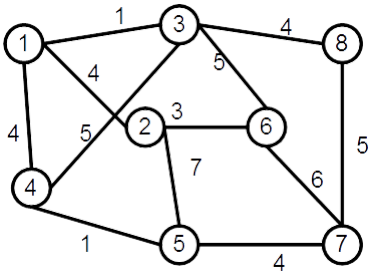
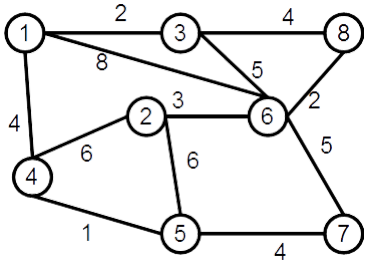
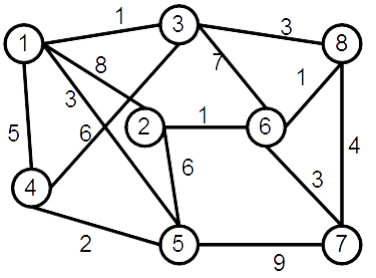
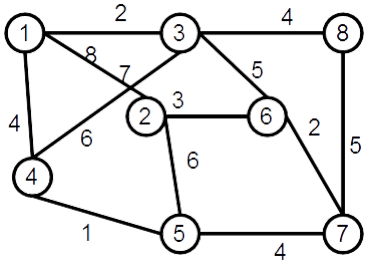
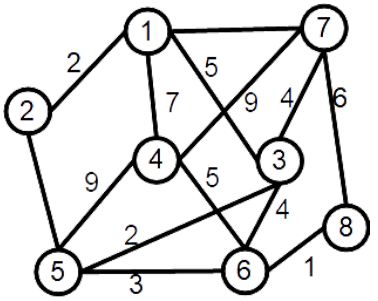
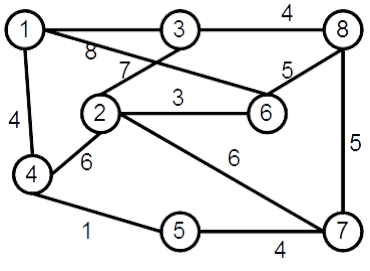
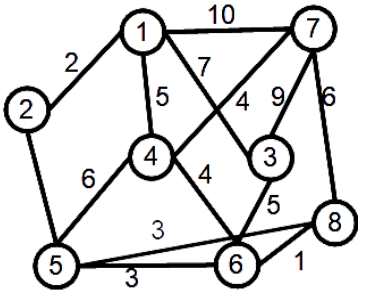
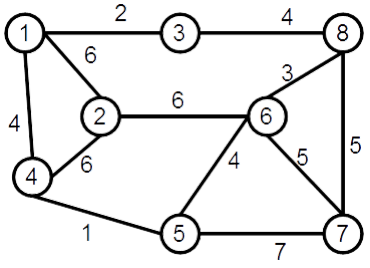
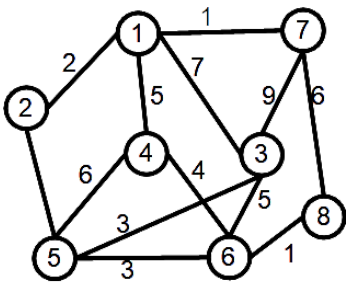
Завдання до лабораторної роботи №4

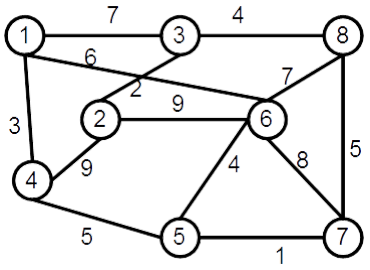
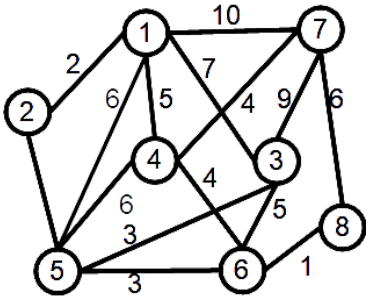
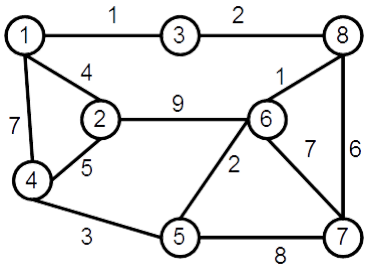
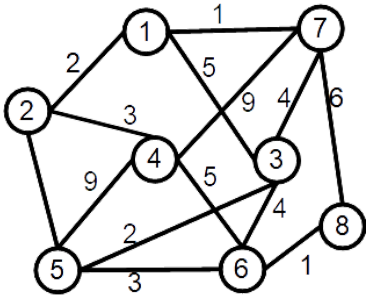
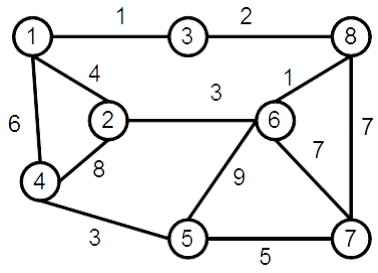
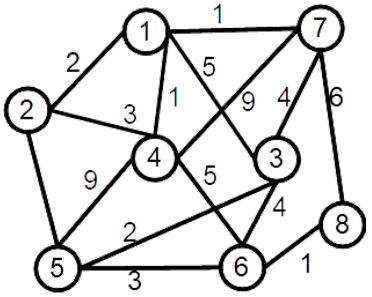
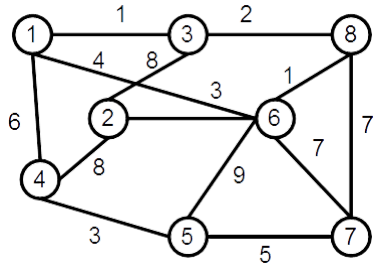
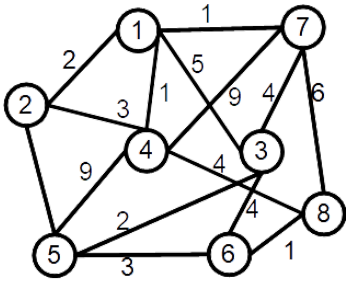
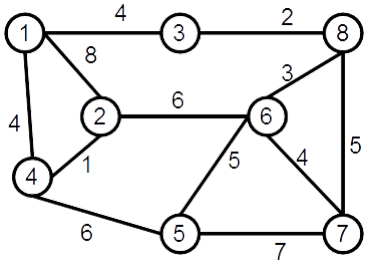
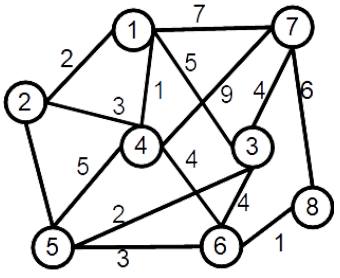
1. Побудувати матриці та списки суміжності для зважених неорієнтованих графів за варіантами завдань
2. Побудувати за допомогою алгоритму Прима мінімальне кістякове дерево (МКД) для графа за варіантами завдань та показати графічно хід побудови МКД за алгоритмом Прима (Таблиця1)
3. Запрограмувати алгоритм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

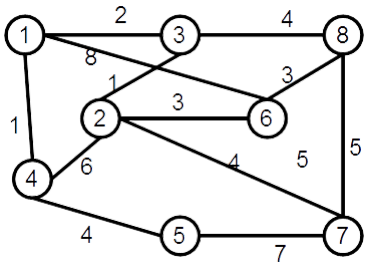
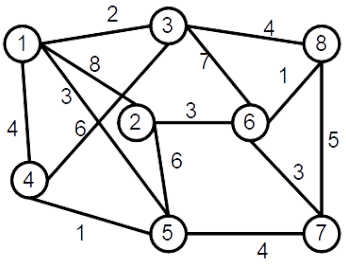
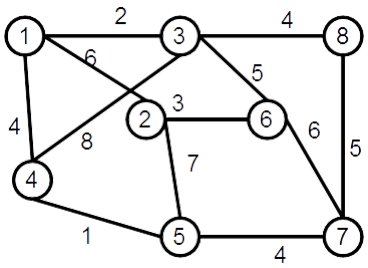
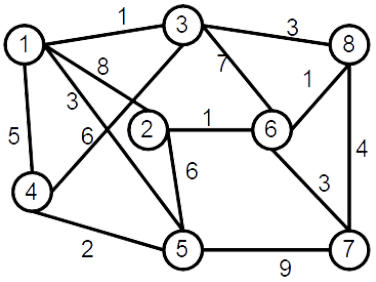
4. Побудувати за допомогою алгоритму Крускала МКД для графа за варіантами завдань та показати графічно хід побудови МКД за алгоритмом Крускала (Таблиця 2).
5. Запрограмувати алгоритм Крускала. Запустити програму, показати результат за варіантами завдань. Показати, що результати ручної та автоматизованої побудови МКД співпадають.
6. Порівняти результати побудови МКД за двома алгоритмами, поясніть 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	

№	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	

№	Граф	№	Граф
15		31	
16		32	

Самостійна робота №4

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
```

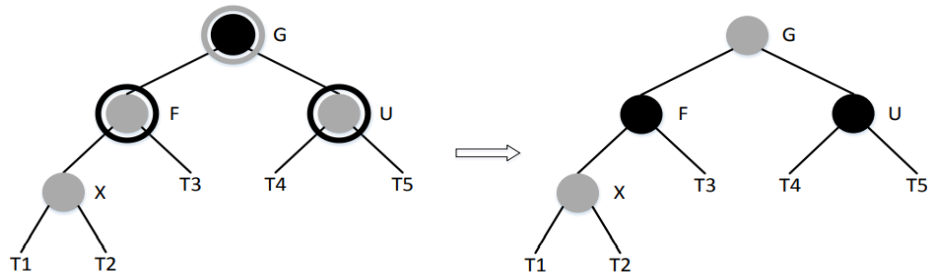
У циклі по елементах масиву запускається функція **Node** insert(x, a[i]). Бінарне дерево пошуку будується так: перший елемент стає кореневим елементом дерева, після чого наступні елементи порівнюються з кореневим. Якщо елемент менше кореневого – він йде в ліве піддерево і порівнюється з лівим нащадком 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).

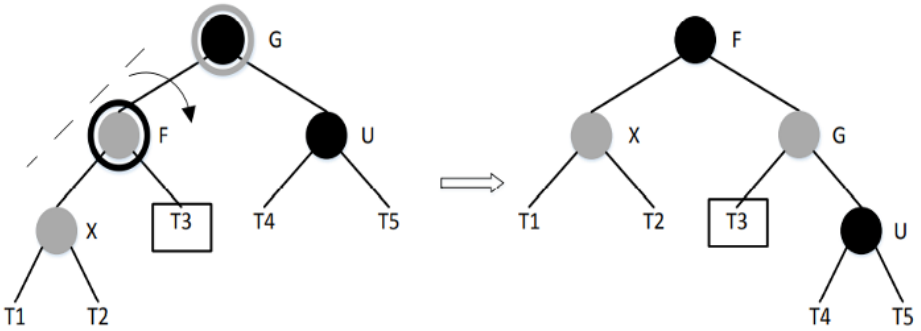
Таблиця 1 Правила балансування при вставці вузла *X* **(червоного)** у ЧЧД

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

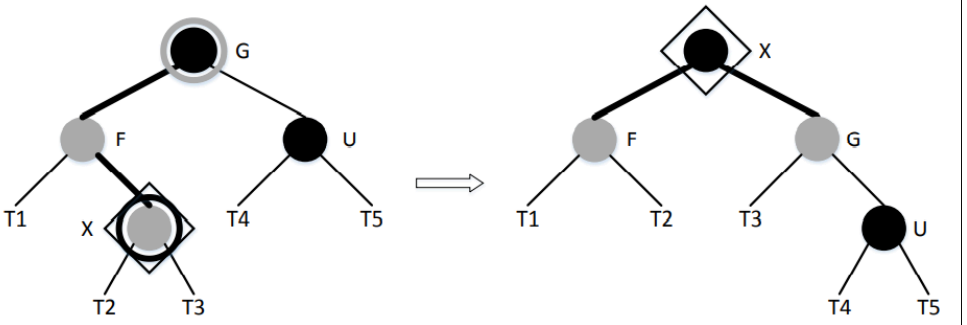


Правий поворот виконується «навколо» зв'язку між *G* і *F*, роблячи *F* новим коренем піддерева, правим дочірнім вузлом якого стає *G* , а колишній правий нащадок вузла *F (T3)* – лівим нащадком *G*.

Випадок 3

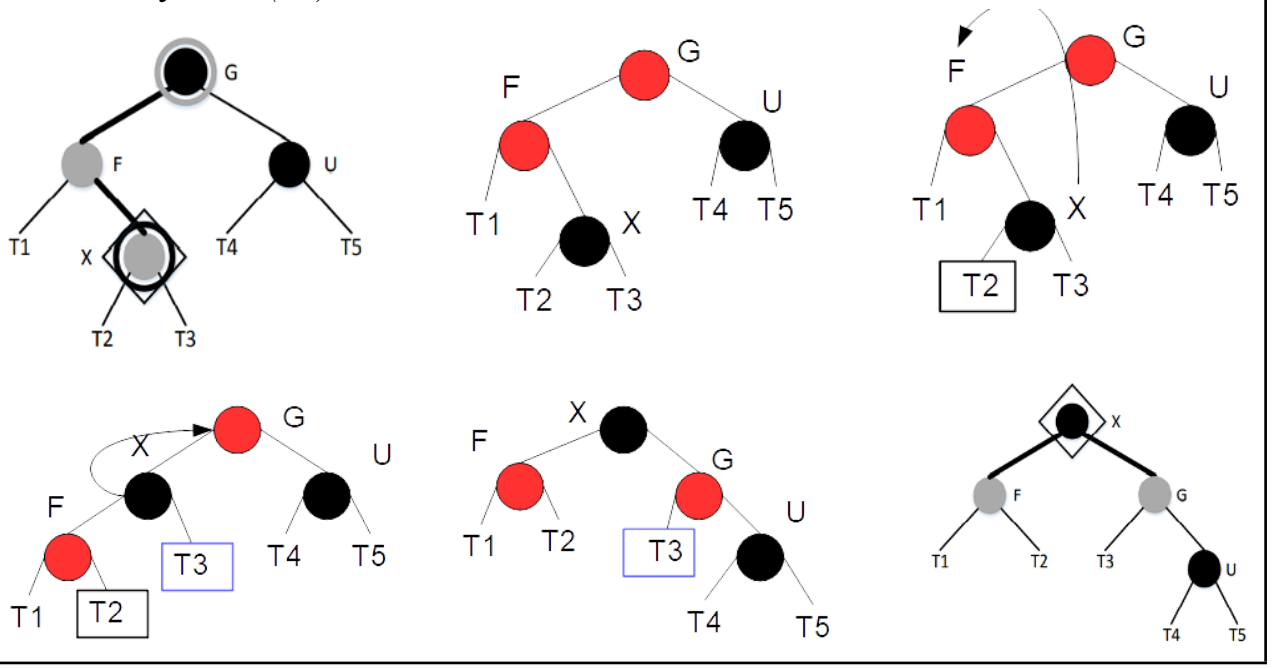
Дядько *U* вузла *X*, що додається, чорний і при цьому ланцюжок вузлів *X-F-G* утворює кут

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.

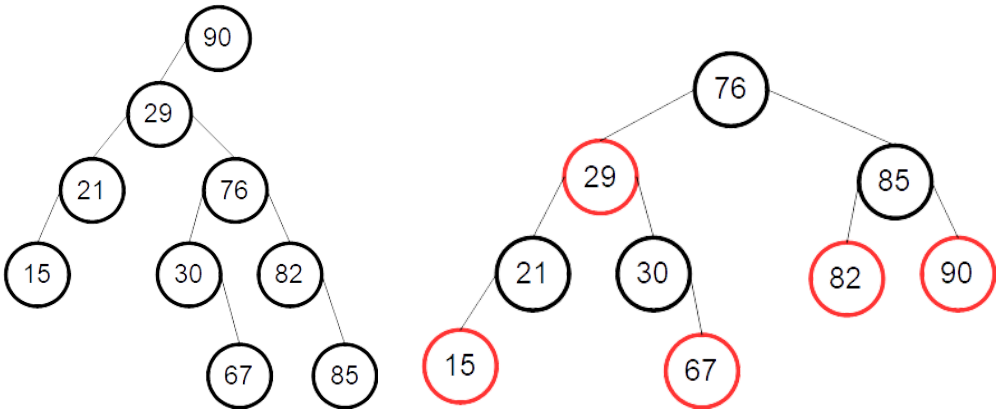


Рисунок 1 Вигляд BST(а) та RB tree (б) побудованих із {90, 29, 76, 21, 30, 15, 82, 67, 85}

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

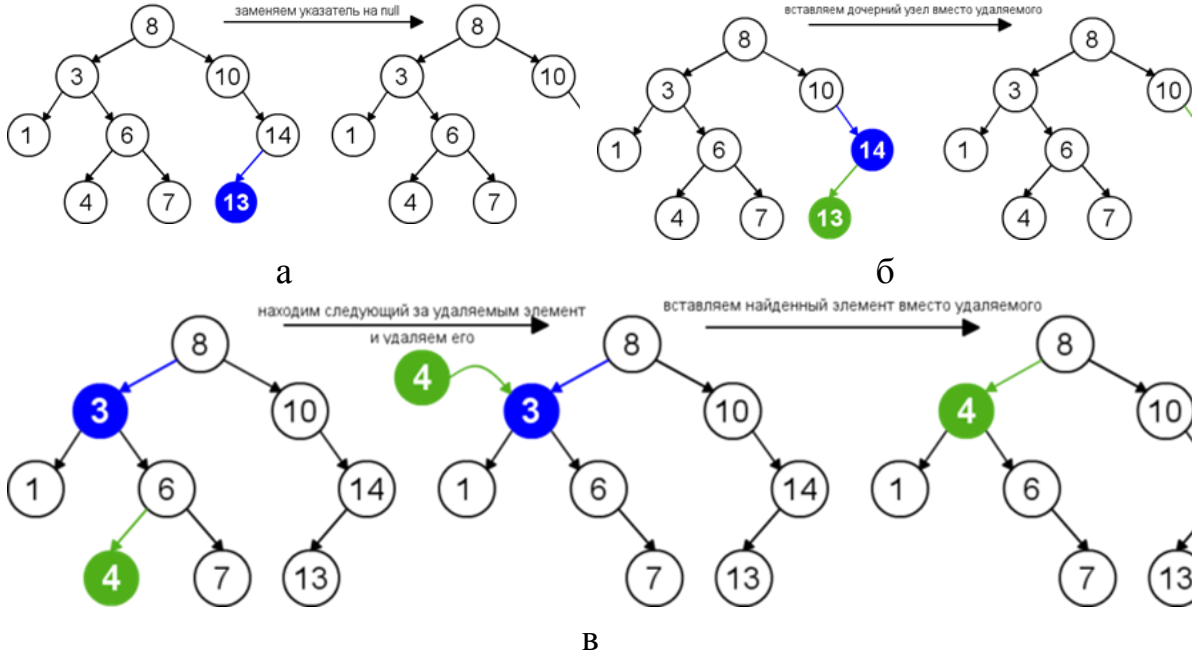
```
Node delete(root : Node, z : T):          // корінь піддерева, ключ, що видаляється
  if root == null
    return root
  if z < root.key //елемент, що видаляється, знаходиться в лівому піддереві
    root.left = delete(root.left, z)
  else if z > root.key //елемент, що видаляється, знаходиться в правому 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.

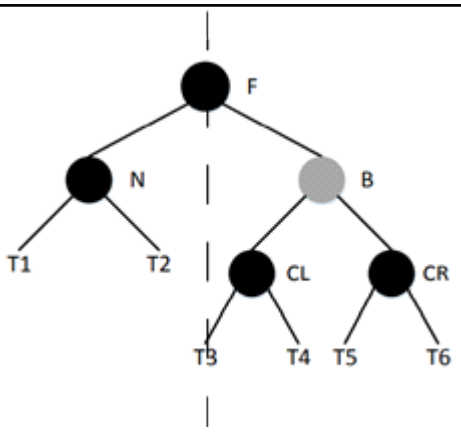
Якщо вузол, що видаляється з дерева, червоний, червоно-чорні властивості дерева зберігаються.

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.

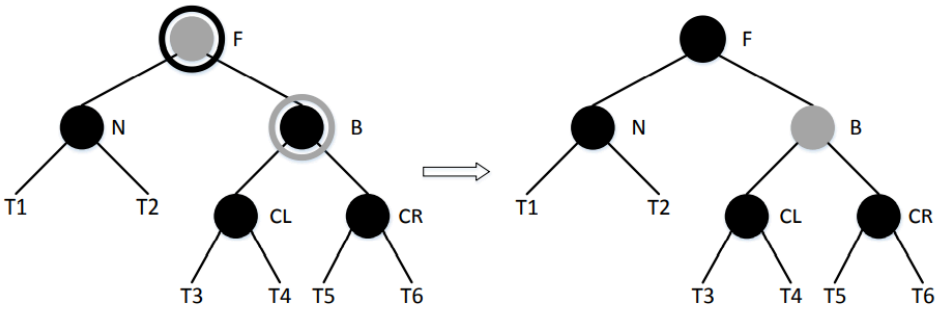
Розглянемо 5 випадків конфігурації дерева після видалення *чорного* вузла, у якого син – *чорний* (Таблиця 3).

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

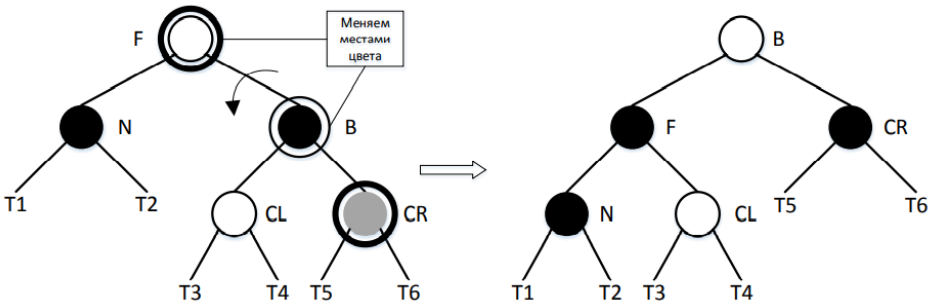
Позначимо сина видаленого вузла за *N*, а батька видаленого вузла за *F*. Після видалення *F* став новим батьком *N*. Позначимо за *B* нового брата *N*, а за *CL* та *CR* – лівого та правого синів *B*, відповідно. На те, якою буде процедура відновлення, впливає тільки комбінація з цих 5 вузлів. Будемо вважати, що *N* – лівий син *F*. Інакше, усі зображення будуть симетричними щодо вертикальної осі, яка проходить через *F*.



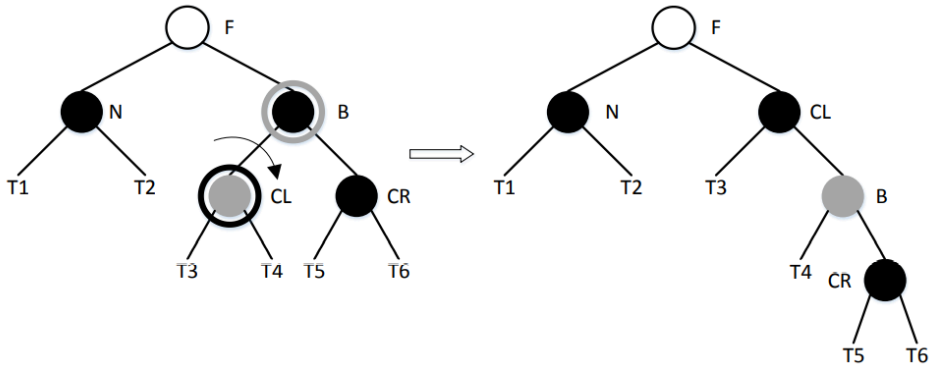
Випадок 1. Батько *F* вузла *N* червоний, інші вузли чорні. (*Батько F вузла N червоний: перефарбування B та F*)

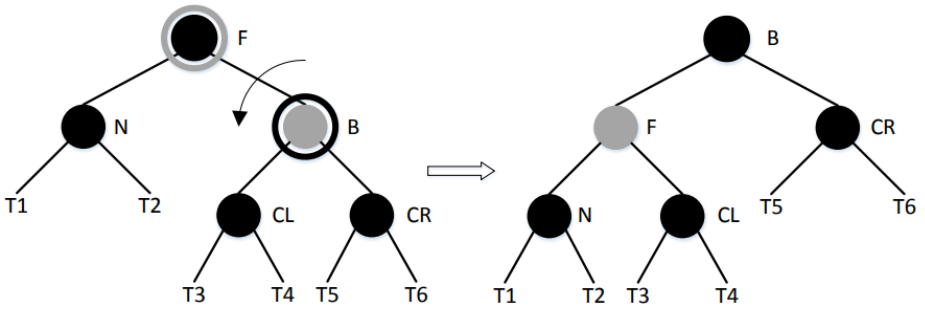


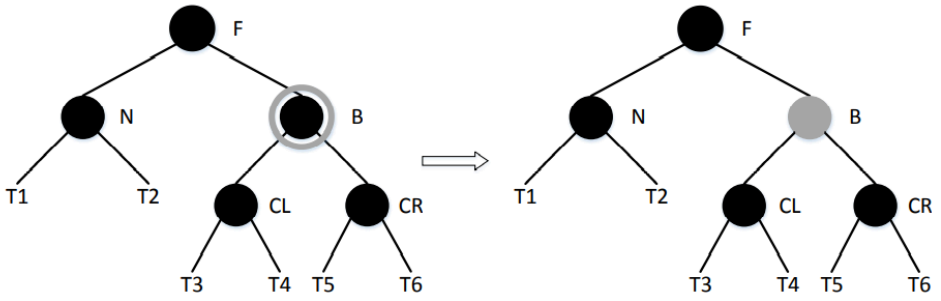
Випадок 2. Брат *B* вузла *N* чорний, а його правий син *CR* червоний. Незалежно від того, чи *F* є чорним або червоним, властивості ЧЧД відновлюються після лівого повороту *F* навколо *B*, перефарбування *CR* в чорний колір і обміну кольорами *F* і *B*.



Лівий поворот навколо F та B, перефарбування CR в чорний та обмін кольорами F та B

<table>
<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> 

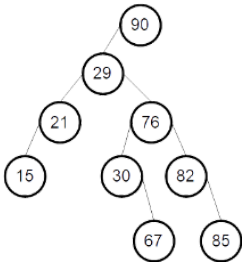
Правий поворот CL навколо B, перефарбування CL та B</td></tr>
<tr><td colspan="2">Правий поворот виконується «навколо» зв'язку між B і CL, роблячи CL новим коренем піддерева, правим дочірнім вузлом якого стає B, а колишній правий нащадок вузла CL (T4) – лівим нащадком B</td></tr>
<tr><td>Випадок 4. Брат 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>
<tr><td>Випадок 5. Усі вузли комбінації (брат B, його діти CL та CR, а також батько F вузла N) чорні.</td><td> 

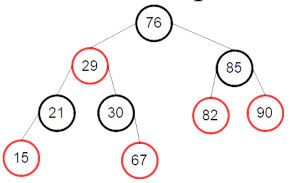
Перефарбування вузла B у червоний колір та продовження процедури нагору</td></tr>
</table>

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

Варіант	Послідовність
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.

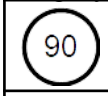
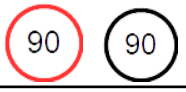
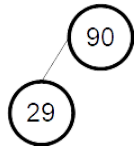
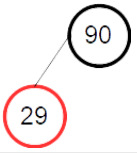
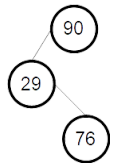
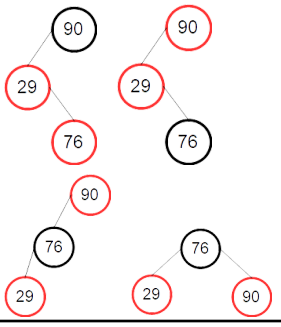
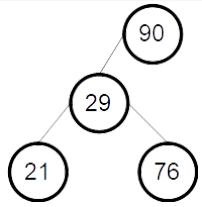
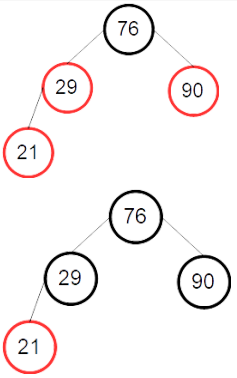
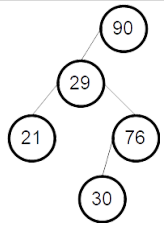
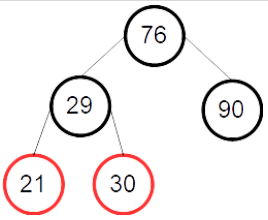
− результати видалення із бінарного дерева пошуку трьох кореневих елементів в такому порядку: праве, ліве піддерева, основний кореневий елемент (зразок таблиця 4 додаткового до ЛР6 файлу)

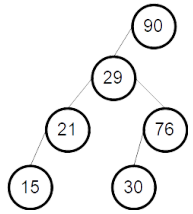
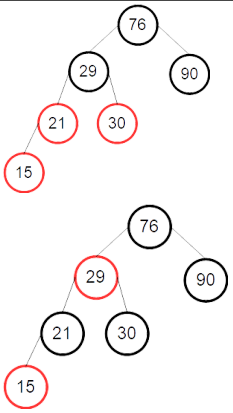
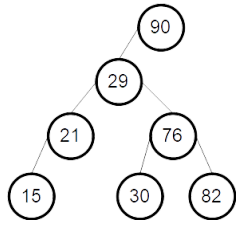
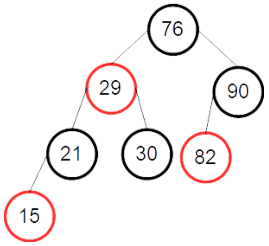
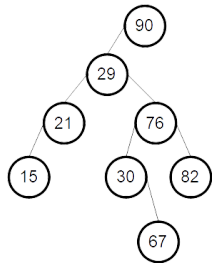
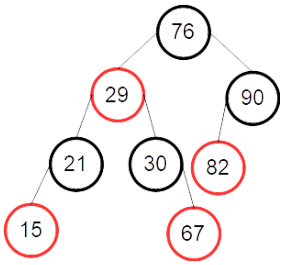
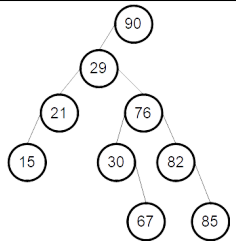
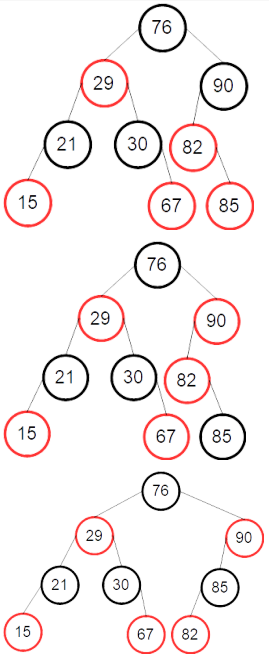
− результати видалення із червоно-чорного дерева трьох чорних елементів, які мають переважно чорних синів (зразок таблиця 5 додаткового до ЛР6 файлу)

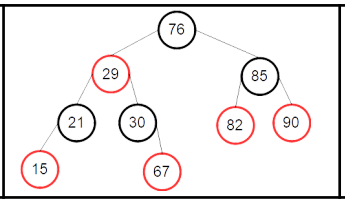
Приклади виконання завдань ЛР6

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

<table>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td>Побудова BST
Додаткових пояснень не потребує</td><td colspan="2">Побудова RB tree. Пояснення дій за алгоритмом вставки обов’язкові</td></tr>
<tr><td> 
</td><td> 
</td><td>Вставляємо елемент 90. Випадок 1 – перефарбування в чорний колір</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td> 
</td><td> 
</td><td>Вставляємо елемент 29. Він стає лівим сином елемента 90. Оскільки батько чорний, немає порушень, нічого не міняємо.</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td> 
</td><td> 
</td><td>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.</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td> 
</td><td> 
</td><td>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.</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 30. Він стає червоним правим сином вузла 29. Порушень немає</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
</table>

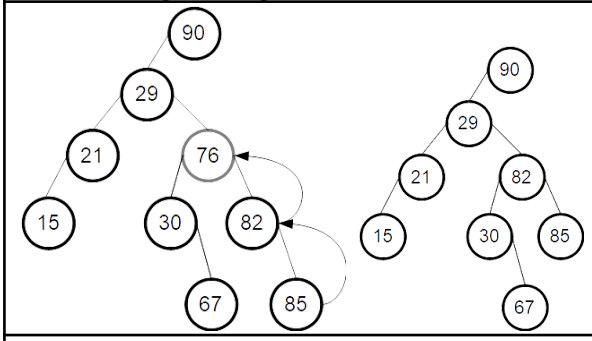
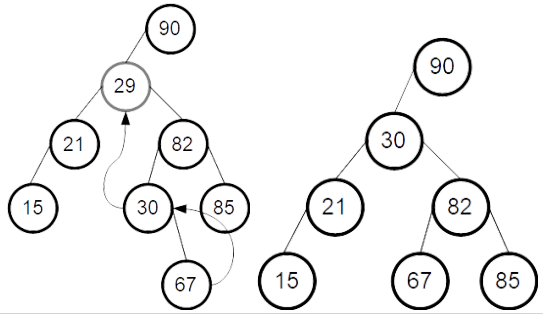
<table>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 15. Він стає червоним лівим сином вузла 21. Червоне-червоне порушення. Випадок 1 - утворюється прямої лінії, дядько доданого вузла червоний. Перефарбовуємо батька (21) та дядька (30) вузла 15 чорним кольором, а дідуся вузла 15 вузел (29) червоним кольором.
Порушень відносно вузла 29 немає. Вимоги стосовно чорної висоти виконуються</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 82. Він стає червоним лівим сином вузла 90. Порушень немає</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 67. Він стає червоним правим сином вузла 30. Порушень немає</td></tr>
<tr><td colspan="3">90, 29, 76, 21, 30, 15, 82, 67, 85</td></tr>
<tr><td> 
</td><td> 
</td><td>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</td></tr>
</table>

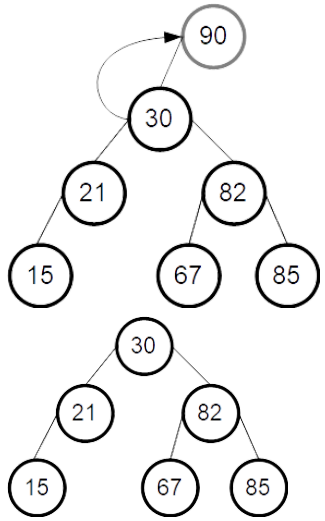
		

Дерево побудовано. Висота бінарного дерева пошуку дорівнює 5. Чорна висота червоно-чорного дерева дорівнює 2, а висота 4

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

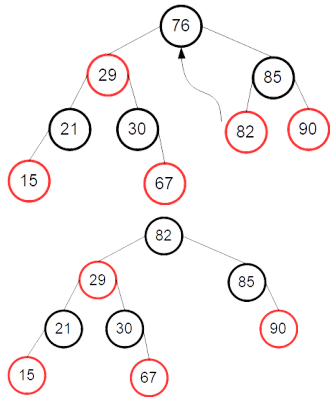
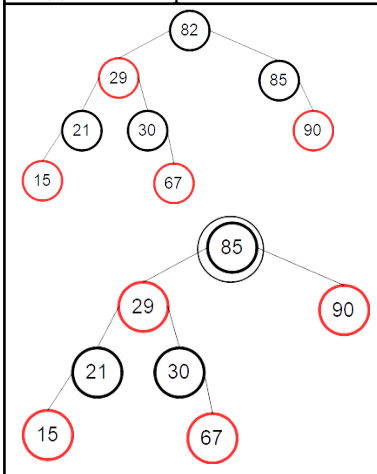
Таблиця 4 Видалення вузлів із бінарного дерева пошуку

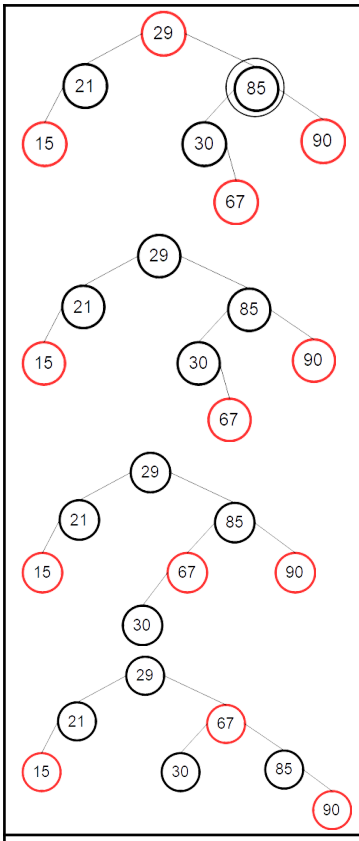
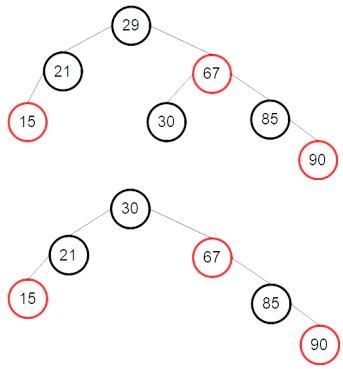
Видаляємо правий кореневий елемент 76	
	Кореневий елемент правого піддерева 76, що видаляється, має два дочірніх вузла (елементи 30 та 82). Шукаємо крайній лівий елемент правого піддерева – це елемент 82 з правим нащадком – елементом 85. Тому елемент 82 стає на місце елемента 76, а елемент 85 на місце 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

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