Міністерство освіти і науки України
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Алгоритми та структури даних
Частина 1

Методичні рекомендації
до виконання лабораторних робіт студентами
денної та заочної форми навчання спеціальностей
122 "Комп’ютерні науки",
123 "Комп’ютерна інженерія" та
125 "Кібербезпека та захист інформації"

Кропивницький 2025

Міністерство освіти і науки України
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Мелешко Є. В., Лисенко І.А.

Алгоритми та структури даних
Частина 1

Методичні рекомендації
до виконання лабораторних робіт студентами
денної та заочної форми навчання спеціальностей
122 "Комп’ютерні науки",
123 "Комп’ютерна інженерія" та
125 "Кібербезпека та захист інформації"

Затверджено
на засіданні кафедри
кібербезпеки та
програмного
забезпечення
Протокол №_1
від 25.08.2025

Кропивницький 2025

Алгоритми та структури даних. Частина 1. Методичні рекомендації до виконання лабораторних робіт студентами денної та заочної форми навчання спеціальностей 122 "Комп’ютерні науки", 123 "Комп’ютерна інженерія" та 125 "Кібербезпека та захист інформації". – Кропивницький: ЦНТУ, 2025. – 52 с.

Методичні рекомендації адресовані майбутнім розробникам програмного забезпечення, що навчаються за спеціальностями 122 "Комп’ютерні науки", 123 "Комп’ютерна інженерія" та 125 "Кібербезпека та захист інформації". Вони охоплюють базові алгоритми та структури даних, зокрема рекурсію, фрактальні методи візуалізації, алгоритми сортування й пошуку даних, а також роботу з двійковими деревами та хеш-таблицями. Ці методичні рекомендації можуть використовуватися також студентами інших спеціальностей. Вони містять основні теоретичні положення та практичні завдання, необхідні для засвоєння учбового матеріалу.

Укладачі: Мелешко Єлизавета Владиславівна,
доктор технічних наук, професор
Лисенко Ірина Анатоліївна,
кандидат технічних наук

Рецензенти: Дрєєв Олександр Миколайович,
кандидат технічних наук, доцент
Минайленко Роман Миколайович,
кандидат технічних наук, доцент

ЗМІСТ

Вступ

Методичні рекомендації до лабораторних робіт з дисципліни «Алгоритми та структури даних» для студентів спеціальностей 122 "Комп’ютерні науки", 123 "Комп’ютерна інженерія" та 125 "Кібербезпека та захист інформації" містять в собі теоретичний матеріал, завдання та інструкції виконання до лабораторних робіт і призначені для студентів, що вивчають програмування, а також можуть бути корисними для школярів, вчителів та просто особистостей, що цікавляться інформаційними технологіями.

Також у цих методичних рекомендаціях міститься приклад оформлення звіту з лабораторної роботи, приклад оформлення лістингу програми, шкала оцінювання успішності студентів та список рекомендованої літератури.

Дані методичні рекомендації починають знайомити з базовими алгоритмами та стурктурами даних у програмуванні, зокрема, з рекурсію, фрактальними методами візуалізації, алгоритмами сортування й пошуку даних, двійковими деревами та хеш-таблицями. Приводиться багато практичних прикладів та ілюстрацій, які допомагають краще засвоїти викладений матеріал.

Лабораторна робота № 1

Тема: Рекурсія

Мета: Вивчення рекурсивного способу опису алгоритмів

Теоретичні відомості

Рекурсія (самоповторення) – дія, що вертається до "самої себе". Є два види рекурсії: (1) *пряма рекурсія* означає, що процедура викликає саму себе; (2) *непряма рекурсія* означає, що одна процедура викликає іншу процедуру, а та у свою чергу прямо або побічно приводить до виклику першої процедури.

Рекурсію варто використовувати тільки тоді, коли задача легко піддається рекурсивному рішенню. Будь-яка задача, що може бути вирішена рекурсивно, також може бути вирішена й без рекурсії за допомогою циклів.

Алгоритм називається рекурсивним, якщо він прямо або побічно звертається до самого себе. Часто в основі такого алгоритму лежить рекурсивне визначення якогось поняття. Наприклад, про факторіал числа N можна сказати, що N! = N*(N - 1)!, якщо N > 0, і N! = 1, якщо N = 0. Це – рекурсивне визначення.

Будь-яке рекурсивне визначення складається із двох частин. Ці частини прийнято називати *базовою* й *рекурсивною* частинами. Базова частина є нерекурсивною й задає визначення для деякої фіксованої частини об'єктів. Рекурсивна частина визначає поняття через нього ж і записується так, щоб при ланцюжку повторних застосувань вона редукувалася б до бази.

Приклад

Завдання. Написати рекурсивну програму пошуку мінімального елемента масиву.

Рішення. Опишемо функцію Pmin, що визначає мінімум серед перших n елементів масиву a. Параметрами цієї функції є кількість елементів у розглянутій частині масиву – n і значення останнього елемента цієї частини – a[n]. При цьому якщо n>2, то результатом є мінімальне із двох чисел – a[n] і мінімального числа з перших (n-1) елементів масиву. У цьому полягає рекурсивний виклик. Якщо ж n=2, то результатом є мінімальне з перших двох елементів масиву. Щоб знайти мінімум всіх елементів масиву, потрібно звернутися до функції Pmin, вказавши як параметри значення розмірності масиву й значення останнього його елемента. Мінімальне із двох чисел визначається за допомогою функції Min, параметрами якої є ці числа.

Псевдокод:

```
Дано масив  mas[1..n]

Функція Min (a, b : цілий тип) : цілий тип
Початок
  якщо a>b то Min := b інакше Min:=a;
Кінець

Функція Pmin(n, b : цілий тип) : цілий тип
```

Початок
якщо n = 2 то Pmin := Min(n, mas[1])
інакше Pmin := Min(mas[n], Pmin(n-1, mas[n]));
Кінець

Виклик рекурсивної функції Pmin у програмі:
Вивести “Мінімальний елемент масиву – ” Pmin(n, mas[n]))

Завдання:

1) Обчислити послідовність Фібоначчі до *i*-го елемента двома способами: за допомогою циклу та за допомогою рекурсії. Числа Фібоначчі визначаються за наступною формулою:

$F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$, n - натуральне додатне число.

2) Відповідно до свого варіанту виконати завдання з використанням рекурсії:

1. Написати функцію додавання двох чисел, використовуючи тільки додавання одиниці.
2. Написати функцію множення двох чисел, використовуючи тільки операцію додавання.
3. Перевірити, чи є фрагмент рядка з i-го по j-й символ паліндромом.
4. Задана послідовність додатніх чисел, за якою слідує від’ємне число. Написати функцію для знаходження суми цих позитивних чисел.
5. Дана монотонна послідовність, в якій кожен натуральний номер К зустрічається рівно К разів: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ... По вказаному користувачем натуральному N виведіть перші N членів цієї послідовності.
6. Написати функцію визначення, чи є задане натуральне число простим.
7. Обчислити суму елементів одномірного масиву за допомогою рекурсивної функції.
8. Написати рекурсивну функцію, що обчислює довжину рядка.
9. Знайти максимальний елемент в масиві $a_1 \ldots a_n$, використовуючи очевидне відношення $\max(a_1 \ldots a_n) = \max((\max a_1 \ldots a_{n-1}), a_n)$.
10. Напишіть рекурсивну функцію зведення в ступінь, що користується наступною властивістю: $a^n = a * a^{n-1}$.
11. Обчислити добуток елементів одномірного масиву за допомогою рекурсивної функції.
12. Напишіть функцію зведення в ступінь для негативних значень n: $a^{-n}=1/a^n$.
13. Напишіть функцію, що знаходить n-не просте число.
14. Знайти n-ний член геометричної прогресії, $b_n = b_1 * q^{n-1}$.
15. Знайти суму n членів арифметичної прогресії.
16. Знайти суму n членів геометричної прогресії.
17. Знайти суму перших N чисел Фібоначчі за допомогою рекурсії.

18. Знайти n-ий член арифметичної прогресії $a_n=a_1+d*(n-1)$. a_1, d, n – вводяться користувачем.

19. Напишіть програму, що виводить перші k пар простих чисел. Два числа a та b утворюють пару простих чисел, якщо вони обидва прості й b=a+2.

20. Дано масив цілих чисел, всі числа, що межують з «1» замінити на «0».

Контрольнізапитання:

1. Дайте означення рекурсії. Чим відрізняються **пряма** і **непряма** рекурсія? Наведіть по одному прикладу для кожного виду.
2. Що таке **базовий випадок** (базова частина) та **рекурсивна частина** рекурсивного означення? Поясніть на прикладі факторіала або чисел Фібоначчі.
3. Чому в будь-якому рекурсивному алгоритмі обов’язково повинна бути умова завершення? Що станеться, якщо її немає або вона ніколи не виконується?
4. Поясніть принцип роботи рекурсивної функції Pmin для пошуку мінімального елемента масиву: – які в неї параметри, – який зміст має рекурсивний виклик Pmin(n-1, mas[n]), – що відбувається, коли n = 2.
5. Поясніть, як одну з задач лабораторної (наприклад, обчислення чисел Фібоначчі, зведення в ступінь або сума елементів масиву) можна реалізувати:
   а) рекурсивно;
   б) за допомогою циклу.
   У чому переваги й недоліки кожного підходу для цієї задачі?

Лабораторна робота № 2

Тема: Фрактали

Мета: Навчитися реалізовувати алгоритми побудови геометричних та алгебраїчних фракталів

Виділяють чотири типи фракталів: геометричні, алгебраїчні, стохастичні й на основі систем ітеруємих функцій (використовуються аффінні перетворення). Фрактали мають властивості самоподоби й дробової розмірності.

Найбільш відомим геометричним фракталом є трикутник Серпінського.

Трикутник Серпінського – це геометричний фрактал, що був запропонований польським математиком Вацлавом Серпіньським в 1915 році. Також відомий як «серветка» або «решітка» Серпінського.

Побудова. Береться рівносторонній трикутник. На першому кроці малюється трикутник з вершинами в середині сторін початкового трикутника.

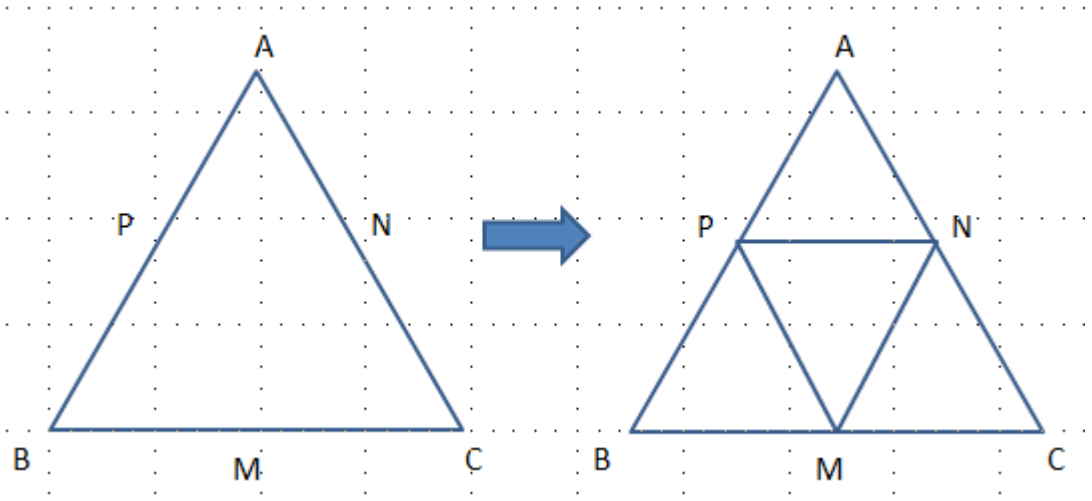


Рисунок 1 – Нульове та перше покоління трикутника Серпінського

На другому кроці аналогічні трикутники із трьох менших трикутників, які з’явилися після першого кроку, і т. д.

Рисунок 2 – П’ять перших поколінь трикутника Серпінського

Після нескінченного повторення цієї процедури, від суцільного трикутника залишається підмножина – трикутник Серпінського.

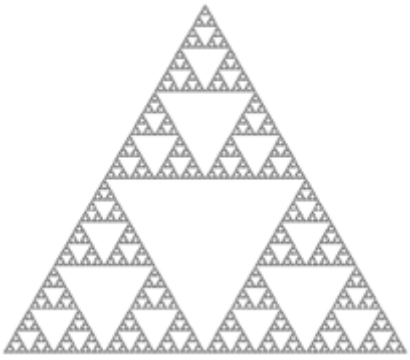


Рисунок 3 – Трикутник Серпінського

Розглянемо графічні інструменти середовища програмування Delphi, які можна використати для візуалізації фракталів.

Для виведення зображення використаємо компонент Image та його властивість Canvas.

Приклад процедури, що малює лінію на канві компонента Image:

```
 procedure Line(x1,y1,x2,y2:real; c:TCanvas);
begin
 c.Moveto(round(x1),round(y1));
 c.lineto(round(x2),round(y2));
 end;
```

Приклад виклику процедури в програмі:

```
line(x1,y1,x2,y2, Form1.Image1.Canvas);
```

Оновлення малюнка:

image1.Canvas.Refresh – слід використати перед виведенням зображення

Встановлення кольору малюнка:

image1.Canvas.Brush.Color:=clWhite

Очищення малюнка:

image1.Canvas.FillRect(Rect(0, 0, Image1.Width, Image1.Height)) – замальовування білим квадратом

або:

image1.Picture:=nil – повністю прибирається зображення на малюнку, повернення до кольору форми

Псевдокод рекурсивної процедури побудови даного фрактала:

Процедура Трикутник_Серпінського(age – номер покоління, x1,y1,x2,y2,x3,y3 – координати вершин поточного трикутника)

Допоміжні змінні: xd,yd,xe,ye,xf,yf – для розрахунку координат наступних трикутників

Початок
age = age + 1
якщо age = FinalAge то // FinalAge – номер останнього покоління
початок
// вивести трикутник
виведення лінії з координатами (x1,y1) та (x2,y2)
виведення лінії з координатами (x2,y2) та (x3,y3)
виведення лінії з координатами (x3,y3) та (x1,y1)
кінець
інакше
початок
// обчислення координат вершин трикутників для наступного покоління
xd:= (x1+x2)/2
yd:= (y1+y2)/2
xe:= (x2+x3)/2
ye:= (y2+y3)/2
xf:= (x1+x3)/2
yf:= (y1+y3)/2

//рекурсивний виклик процедури побудови фракталу
Трикутник_Серпінського(age,x1,y1,xd,yd,xf,yf)
Трикутник_Серпінського(age,xd,yd,x2,y2,xe,ye)
Трикутник_Серпінського(age,xf,yf,xe,ye,x3,y3)
кінець
Кінець

Приклад, координат вершин трикутника для нульового покоління, які можна використати при виклику рекурсивної функції у програмі:
x1:=10; y1:=10;
x2:=320; y2:=470;
x3:=630; y3:=10;

Найвідоміший алгебраїчний фрактал - множина Мандельброта.

Множина Мандельброта – обмежена та зв’язна множина на комплексній площині, межа якої утворює фрактал. Названа на честь Бенуа Мандельброта, який вивчав і популяризував її.

Множина Мандельброта – фрактал, визначений як множина точок *c* на комплексній площині, для яких ітеративна послідовність:

$z_0 = 0$

$z_{n+1} = {z_n}^2 + c$

не йде на нескінченність.

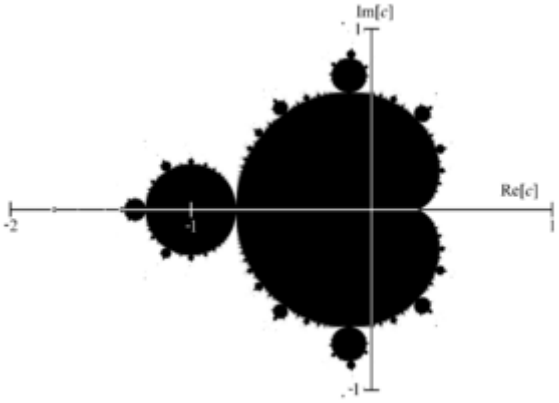


Рисунок 4 – Множина Мандельброта

Розширене визначення

Таким чином, вищевказана послідовність може бути розкрита для кожної точки c на комплексній площині в такий спосіб:

$$c = x + i \cdot y$$
$$Z_0 = 0$$
$$\begin{aligned} Z_1 &= Z_0^2 + c \\ &= x + iy \\ Z_2 &= Z_1^2 + c \\ &= (x + iy)^2 + x + iy \\ &= x^2 + 2ixy - y^2 + x + iy \\ &= x^2 - y^2 + x + (2xy + y)i \end{aligned}$$
$$Z_3 = Z_2^2 + c = ...$$

і так далі.

Якщо переформулювати ці вирази у вигляді ітеративної послідовності значень координат комплексної площини x і y, тобто замінивши z_n на $x_n+i\cdot y_n$, а c на p+i·q, ми одержимо:

$$x_{n+1} = {x_n}^2 - {y_n}^2 + p$$
$$y_{n+1} = 2x_n y_n + q$$

Візуально, всередині множини Мандельброта можна виділити нескінченну кількість елементарних фігур, причому сама найбільша в центрі являє собою кардіоїду.

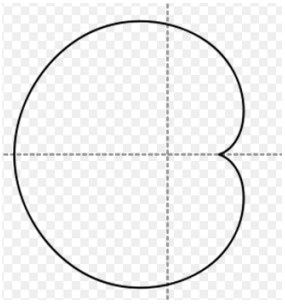


Рисунок 5 – Кардіоїда

Також є набір овалів, що торкаються кардіоїди, розмір яких поступово зменшується, прагнучи до нуля. Кожний із цих овалів має свій набір менших

овалів, діаметр яких також прагне до нуля й т.д. Цей процес триває нескінченно, утворюючи фрактал. Також важливо, що ці процеси розгалуження фігур не вичерпують повністю множину Мандельброта: якщо розглянути зі збільшенням додаткові «гілки», то в них можна побачити свої кардіоїди й кола, не пов'язані з головною фігурою. Сама найбільша фігура (видима при розгляданні основної множини) з них перебуває в області від –1,78 до –1,75 на негативній осі дійсних значень.

Рисунок 6 – Фрагмент границі множини Мандельброта в кольоровому варіанті

Суворо математично, зображення множини Мандельброта повинне бути чорно-білим. Точка або потрапляє всередину множини, або ні. Незважаючи на це, за допомогою комп'ютера можна побудувати й кольорові зображення. Найпоширенішим способом є розфарбовування точок зовні множини в колір, що дорівнює кількості ітерацій, за яку точка йде в «нескінченність».

Розглянемо функцію роботи з графікою в Delphi, що допоможе візуалізувати дану множину.

Зафарбовування точки (i,j) в колір color:

image1.Picture.Bitmap.Canvas.Pixels[i,j]:=color

Змінна, що містить в собі колір точки має бути типу TColor. Тип **TColor** визначений як довге ціле (LongInt). Його чотири байти містять інформацію про частки синього (B), зеленого (G), і червоного (R) кольорів. Частка кожного кольору може мінятися від 0 до 255. Значення змінній привласнюється так color:=RGB(R, G, B), або так color:=$00BBGGRR (де букви – числові значення)

Приклад, color:=RGB(0,0,0) – чорний колір, color:=RGB(255,255,255) – білий колір, color:=RGB(0,0,255) – синій колір.

Розглянемо програмну реалізацію. Побудова множини Мандельброта зводиться до використання формули:

$$Z_{i+1}=Z_i^2+C$$

де Z і C - комплексні числа. Як видно, формула по суті являє собою звичайну рекурсію. Знаючи правила роботи з комплексними числами дану формулу можна спростити й привести до наступного вигляду:

$$x_{i+1}=x_i^2-y_i^2+a$$

$$y_{i+1}=2*x_i*y_i+b$$

Побудова множини Мандельброта зводиться до наступного. Для кожної точки (a,b) проводиться серія обчислень за наведеними формулами, причому x_0 і y_0 приймаються рівними нулю, тобто точка у формулі виступає як константа. На кожному кроці обчислюється величина $r=sqrt(x_2+y_2)$. Значенням r , як не важко помітити, є відстань точки з координатами (x,y) від початку координат ($r=sqrt[(x-0)^2+(y-0)^2]$). Вихідна точка (a,b) вважається приналежною множині Мандельброта, якщо вона ніколи не віддаляється від початку координат на якесь критичне число. Для відображення можна підрахувати швидкість віддалення від центра, якщо наприклад точка пішла за критичну відстань, і залежно від цього зафарбувати цю точку у відповідний колір. Повне зображення множини Мандельброта можна одержати на площині від -2 до 1 по осі x та від -1.5 до 1.5 по осі y. Також відомо, що для одержання прийнятної точності досить 100 ітерацій (за теорією їх повинно бути нескінченно багато). Нижче представлений псевдокод функції, що реалізує виконання ітерацій і визначення приналежності точки множині Мандельброта, точніше на виході одержуємо колір для відповідної точки. Як критичне число взяте число 2.

Примітка. Щоб не обчислювати корінь, можна порівнювати квадрат відстані r^2 із квадратом критичного числа, тобто порівнювати (x^2+y^2) і 4.

Функція Мандельброт(a,b – координати точки): колір точки

Допоміжні змінні x,y – для обчислення x_{i+1} та y_{i+1}
xy, x2,y2 – для обчислення x*y, x^2 та y^2
r – критичне число, за яке не віддаляються точки приналежні множини Мандельброта
k – лічильник ітерацій

```
Початок
r:=0, x:=0, y:=0 // початкові значення змінних
k:=100 //кількість ітерацій
поки k>0  виконувати
початок
    x2:=x*x
    y2:=y*y
    xy:=x*y
    r:=sqrt(x2+y2)
    якщо (r>2) тоді вихід з циклу
    x:=x2-y2+a
    y:=2*xy+b
    k:= k-1
кінець
```

якщо(r<2) тоді функція повертає чорний колір //точка належить множині

інакше функція повертає білий колір //точка не належить множині
Кінець

Нижче наведений процедури, що відображає дану множину.

процедура Візуалізація_множини

Допоміжні змінні:
x_min,y_min,x_max,y_max,hx,hy,x,y – для визначення координат точок
i,j – лічильники
n – розмір зображення
color – колір точки

Початок
x_min:=-1.5, x_max:=2 // координати площини
y_min:=-1.5, y_max:=1.5
n:=300; // розмір площини буде nxn

y:=y_min
hx:=(x_ max-x_min)/n
hy:=(y_ max-y_min)/n
для j від 0 до n виконувати
початок
x:=x_min
для i від 0 до n виконувати
початок
color:=MandelBrot(x,y)
Зафарбовування точки (x,y) кольором color
x:=x+hx
кінець
y:=y+hy
кінець
Кінець

Завдання 1: Реалізувати програму, що будує трикутник Серпінського, значення покоління фрактала повинне вводилося користувачем. Додати функцію створення стохастичного (рандомізованого) фракталу на основі даного.

Завдання 2: Реалізувати програму, що будує множину Мандельброта в чорно-білому та кольоровому варіантах на вибір користувача. Додати функцію вибору масштабу зображення.

Завдання 3 (додаткове): Реалізувати на вибір побудову одного з фракталів – крива Коха, крива дракона, множина Жюліа, Атрактор Лоренца або ін., та навести короткий теоретичний опис побудови вибраного фракталу.

Контрольні питання:

1. Дайте означення фрактала. Які основні властивості фракталів згадуються в теорії (не менше двох) і що вони означають?
2. Перелічіть типи фракталів і наведіть приклади.
3. Опишіть алгоритм побудови трикутника Серпінського.
4. Що таке множина Мандельброта?
5. Як перетворюються координати пікселя зображення (i, j) у координати точки на комплексній площині (x, y) під час побудови множини Мандельброта? Що задають змінні x_min, x_max, y_min, y_max, n, hx, hy в процедурі візуалізації?

Лабораторна робота № 3

Тема: Робота з рядками

Мета: Збереження та маніпулювання текстовими даними. Використання масивів та колекцій.

Тема: Алгоритми сортування масивів

Мета: Навчитися реалізовувати прості алгоритми сортування вибором, вставкою та бульбашкове

Сортування вибором. Цей метод заснований на наступному правилі. Вибирається мінімальний (максимальний) елемент послідовності й обмінюється з першим елементом послідовності. Очевидно, один елемент при цьому стане на своє місце у відсортованій частині послідовності. Далі усе вище сказане треба повторити в невідсортованій частині послідовності й т.д.

```
FOR I = 1 TO N-1 DO
 BEGIN
 <привласнити K індекс найменшого елемента з a[I]...a[N]>
 <поміняти місцями a[I] і a[K]>
 END
```

Сортування вставкою. Цей метод полягає в тому, що на кожному кроці беруть *і-ий* елемент послідовності й передають його в готову відсортовану частину послідовності, вставляючи його на своє місце. Алгоритм сортування вставкою виглядає наступним чином:

```
FOR I := 2 TO N DO
    BEGIN X := a[I];
       <вставити X на підходяще місце в a[1],a[2],...,a[I]>
    END
```

Сортування обміном (бульбашкове). Алгоритм обміну заснований на принципі порівняння й обміну пари сусідніх елементів доти, поки не будуть відсортовані всі елементи.

```
FOR I:= 2 TO N DO
     FOR J:= N DOWNTO I DO
          IF a[ J-1] > a[J] THEN
          BEGIN X := a[ J-1]; a[ J-1] := a[J]; a[J] := X  END
```

Завдання: Досліджувати вивчені алгоритми сортування в такий спосіб:

1. Описати цілочисельний масив, що складається з N елементів (N – константа).

2. Значення елементів масиву задати випадковим чином. Вивести отриманий масив на екран.

3. Для створеного масиву реалізувати різні алгоритми сортування ("бульбашкове", вставками, вибором), при цьому визначаючи кількість проходів масиву, порівнянь і перестановок.

4. Визначити час, витрачений на кожний зі способів.

(!) Увага: для визначення "чистого" часу сортування, виконаєте його в другий раз без використання елементів дослідження (підрахунку проходів і ін.).

Приклад визначення часу роботи програми в С++:

```
#include <ctime> // підключення файлу з функцією clock()
//          ...
    unsigned int start_time =  clock(); // початковий час
    // фрагмент коду програми, час роботи якого потрібно визначити
    unsigned int end_time = clock(); // кінцевий час
    unsigned int search_time = end_time - start_time; // час роботи
```

5. Порівняти методи сортування при розмірі масиву N = 10, 100, 1000, 10000, 30000. Результатом дослідження повинна стати таблиця (у звіті):

N	Кількість проходів			Кількість порівнянь			Кількість перестановок			Час		
	"бульбашкова"	вставками	вибором	"бульбашкова	вставками	вибором	"бульбашкова	вставками	вибором	пухирець	вставками	вибором

Вказівки: 1) щоб у дослідженні використовувався той самий масив, збережіть його в додатковому масиві, і виконуйте операцію копіювання масиву для підготовки до чергового сортування; 2) з метою перевірки після кожного способу виводьте відсортований масив на екран.

Контрольні питання:

1) У чому полягає принцип роботи сортування вибором? Які дві основні дії виконуються на кожному кроці?
2) Як працює сортування вставками? Що означає «вставити елемент у вже відсортовану частину масиву»?
3) Чому сортування обміном називають «бульбашковим»? Як у цьому методі рухаються великі елементи масиву?
4) Які показники ефективності сортування ви досліджуєте в лабораторній (не менше двох) і що вони означають?
5) Навіщо для вимірювання «чистого» часу сортування виконують алгоритм вдруге без підрахунку проходів, порівнянь і перестановок?

Лабораторна робота № 4

Тема: Складні алгоритми сортування. Швидке сортування Хоара

Мета: Навчитися реалізовувати складні алгоритми сортування

Швидке сортування Хоара (англ. *quicksort*) – алгоритм сортування, розроблений англійським інформатиком Чарльзом Хоаром в 1960 р. Один зі швидких універсальних алгоритмів сортування масивів, хоча й має ряд недоліків.

Короткий опис алгоритму:

1 Крок вибрати базовий елемент.

2 Крок порівняти всі інші елементи з базовим, на основі порівняння розбити множину елементів на три – «менші, ніж базовий», «рівні базовому» і «більші, ніж базовий» та розташувати їх у порядку менші-рівні-більші.

3 Крок повторити **рекурсивно** для «менших» і «більших» елементів.

Примітка: на практиці звичайно розділяють сортуєму множину елементів не на три, а на дві частини: наприклад, «менші» і «рівні й більші». Такий підхід у загальному випадку виявляється ефективніше, тому що для здійснення такого поділу досить одного проходу по сортуємій множині й однократного обміну лише деяких обраних елементів.

Детальний опис алгоритму

Швидке сортування використовує стратегію «Розділяй і володарюй». Кроки алгоритму такі:

1 Крок. Вибираємо в масиві деякий елемент, що будемо називати *базовим елементом*. З погляду коректності роботи алгоритму спосіб вибору базового елемента може бути будьяким. З погляду підвищення ефективності алгоритму вибиратися повинна медіана, але без додаткових відомостей про сортуємі дані її звичайно неможливо одержати.

Відомі стратегії вибору базового елемента:

- вибирати постійно той самий елемент, наприклад, середній або останній по розташуванню;

- вибирати елемент із випадково обраним індексом.

2 Крок. Операція *розділення* масиву: реорганізуємо масив таким чином, щоб всі елементи, менші або рівні базовому елементу, виявилися ліворуч від нього, а всі елементи, більші базового – праворуч від нього.

Звичайний алгоритм операції:

1. Два індекси – l (left) і r (right), прирівнюються до мінімального й максимального індексу розділюваного масиву відповідно.

2. Обчислюється індекс базового елемента m (medium).

3. Індекс l послідовно збільшується доти, поки l-й елемент не виявиться

більшим або рівним базовому.

4. Індекс r послідовно зменшується доти, поки r-й елемент не виявиться меншим або рівним базовому.

5. Якщо r = l – знайдена середина масиву – операція розділення закінчена, обидва індекси вказують на базовий елемент.

6. Якщо l < r – знайдену пару елементів потрібно обміняти місцями й продовжити операцію розділення з тих значень l і r, які були досягнуті. Варто врахувати, що якщо яка-небудь границя (l або r) дійшла до базового елемента, то при обміні значення m змінюється на r-й або l-й елемент відповідно.

3 Крок. Рекурсивно впорядковуємо підмасиви, що лежать ліворуч і праворуч від базового елемента. Базою рекурсії є набори, що складаються з одного або двох елементів. Один елемент повертається у незмінному вигляді, а у випадку двох сортування зводиться до перестановки двох елементів. Всі такі відрізки вже впорядковані в процесі розділення масиву.

Оскільки в кожній ітерації (на кожному наступному рівні рекурсії) довжина оброблюваного відрізка масиву зменшується, щонайменше, на одиницю, термінальна гілка рекурсії буде досягнута завжди й обробка гарантовано завершиться.

Оцінка ефективності алгоритму

Сортування Хоара є істотно поліпшеним варіантом алгоритму сортування за допомогою прямого обміну (його варіанти відомі як «Бульбашкове сортування» та «Шейкерне сортування»), відомого, у тому числі, своєю низкою ефективністю. Принципова відмінність полягає в тому, що після кожного проходу елементи розділяються на дві незалежні групи. Цікавий факт: поліпшення самого неефективного прямого методу сортування дало в результаті ефективний поліпшений метод.

Швидке сорування володіє цілим рядом недоліків. Його ефективність проявляється не для всіх масивів, наприклад, якщо обране базове значення виявляється співпадаючим з максимальним значенням, то швидке сортування перетворюється в найповільніше.

Найкращий випадок. Для цього алгоритму найкращий випадок – якщо в кожній ітерації кожний з підмасивів ділився б на два рівних по величині масиву. У результаті кількість порівнянь, що робляться швидким сортуванням, була б рівна значенню рекурсивного виразу $C_N = 2C_{N/2}+N$, що в явному виразі дає приблизно N lg N порівнянь. Це дало б найменший час сортування.

Середній випадок. Дає в середньому $O(n \log n)$ обмінів при впорядкуванні n елементів. У реальності саме така ситуація звичайно має місце при випадковому порядку елементів і виборі базового елемента із середини масиву або випадково.

На практиці (у випадку, коли обміни є більш витратною операцією, ніж порівняння) швидке сортування значно швидше, ніж інші алгоритми з оцінкою $O(n \lg n)$, через те, що внутрішній цикл алгоритму може бути ефективно реалізований майже на будь-якій архітектурі. $2C_{N/2}$ покриває видатки по сортуванню двох отриманих підмасивів; N – це вартість обробки кожного елемента, використовуючи один або інший покажчик. Відомо також, що приблизне значення цього виразу дорівнює $C_N = N \lg N$.

Найгірший випадок. Гіршим випадком, мабуть, буде такий, при якому на кожному етапі масив буде розділятися на вироджений підмасив з одного базового елемента й на підмасив із всіх інших елементів. Таке може відбутися, якщо в якості базового на кожному етапі буде обраний елемент або найменший, або найбільший із всіх оброблюваних.
Гірший випадок дає $O(n^2)$ обмінів. Але кількість обмінів і, відповідно, час роботи – це не найбільший його недолік. Гірше те, що в такому випадку глибина рекурсії при виконанні алгоритму досягне n, що буде означати n-кратне збереження адреси повернення й локальних змінних процедури розділення масивів. Для великих значень n найгірший випадок може призвести до вичерпання пам'яті під час роботи алгоритму. Втім, на більшості реальних даних можна знайти рішення, які мінімізують ймовірність того, що знадобиться квадратичний час.

Поліпшення:

1. Вибирати базовий елемент випадковим чином, тоді найгірший випадок стає дуже малоймовірним, а очікуваний час виконання алгоритму сортування – $O(n \lg n)$.

2. Вибирати базовим елементом середній із трьох (першого, середнього й останнього елементів). Такий вибір також спрямований проти найгіршого випадку.

3. Щоб уникнути досягнення небезпечної глибини рекурсії в найгіршому випадку (або при наближенні до нього) можлива модифікація алгоритму, що усуває одну гілку рекурсії: замість того, щоб після розподілу масиву викликати рекурсивно процедуру розподілу для обох знайдених підмасивів, рекурсивний виклик робиться тільки для меншого підмасива, а більший обробляється в циклі в межах цього ж виклику процедури. З погляду ефективності в середньому випадку різниці практично немає: накладні витрати на додатковий рекурсивний виклик і на організацію порівняння довжин підмасивів і циклу – приблизно одного порядку. Зате глибина рекурсії ні за яких умов не перевищить $\log_2 n$, а в найгіршому випадку виродженого розподілу вона взагалі буде не більше 2 – вся обробка пройде в циклі першого рівня рекурсії.

4. Розбивати масив не на дві, а на три частини.

Приклад. Розглянемо приклад, в якому як базой елемент обираються крайні елементи масиву.

Дано масив:

{9,6,3,4,10,8,2,7}

Беремо 9 як базовий елемент. Порівнюємо 9 із протилежно розміщеним елементом, у цьому випадку це 7.

7 менше, ніж 9, отже елементи міняються місцями.

{**7**,6,3,4,10,8,2,**9**}

Далі починаємо послідовно порівнювати елементи з 9, і міняти їх місцями залежно від порівняння.

{7,**6**,3,4,10,8,2,**9**}

{7,6,**3**,4,10,8,2,**9**}

{7,6,3,**4**,10,8,2,**9**}

{7,6,3,4,**9**,8,2,**10**} - 9 і 10 міняємо місцями.

{7,6,3,4,**8**,**9**,2,10} - 9 і 8 міняємо місцями.

{7,6,3,4,8,**2**,**9**,10} - 2 і 9 міняємо місцями.

Після такого перекидання елементів весь масив розбивається на дві підмножини розділені елементом 9.

1 підмножина {7,6,3,4,8,2}

2 підмножина {10}

Далі по вже відпрацьованому алгоритму сортуються ці підмножини. Підмножину, що складається з одного елемента, звичайно ж сортувати не потрібно. Вибираємо в першій підмножині базовий елемент 7.

{7,6,3,4,8,2}

{**2**,6,3,4,8,**7**} - міняємо місцями 2 і 7.

{2,**6**,3,4,8,**7**}

{2,6,**3**,4,8,**7**}

{2,6,3,**4**,8,**7**}

{2,6,3,4,**7**,**8**}- міняємо місцями 7 і 8

Одержали знову дві підмножини.

{2,6,3,4}

{8}

А далі все відбувається аналогічно.

Порівняння алгоритмів сортування

Час – основний параметр, що характеризує швидкодію алгоритму. Називається також обчислювальною складністю. Для сортування важливе *найгірша*, *середня* й *найкраща* поведінка алгоритму. Для типового

алгоритму гарна поведінка – це $O(n \log n)$ і погана поведінка — це $O(n^2)$. Ідеальна поведінка для сортування – $O(n)$.

Пам'ять – ряд алгоритмів вимагає виділення додаткової пам'яті під тимчасове зберігання даних. Як правило, ці алгоритми вимагають $O(\log n)$ пам'яті. Алгоритми сортування, що не споживають додаткової пам'яті, відносять до *сортувань на місці*.

Властивості алгоритмів сортування

Стабільність (англ. *stability*) – стабільне сортування не міняє взаємного розташування елементів з однаковими ключами, але, як правило, більш повільне, ніж нестабільне і може потребувати більше пам’яті.

Природність поводження – ефективність методу при обробці вже впорядкованих або частково впорядкованих даних. Алгоритм поводиться природно, якщо враховує цю характеристику вхідної послідовності й працює краще.

Використання операції порівняння. Алгоритми, що використовують для сортування порівняння елементів між собою, називаються заснованими на порівняннях. Мінімальна трудомісткість *найгіршого випадку* для цих алгоритмів становить $O(n \log n)$, але вони відрізняються гнучкістю застосування. Для спеціальних випадків (типів даних) існують більше ефективні алгоритми.

Порівняння властивостей розглянутих раніше алгоритмів сортування

Назва алгоритму сортування	***Обчислювальна складність***	***Використовувана пам’ять***	***Стабільність***	***Особливості***
Бульбашкове	n^2	1	стабільне	Легкий в реалізації алгоритм. Але один з найповільніших (в класичному варіанті без вдосконалень алгоритму). Швидкість майже не залежить від ступеня впорядкованості даних.
Вибором	n^2	1	стабільне	Простота реалізації. Природність поводження.
Вставкою	n^2	1	стабільне	Ефективний на невеликих наборах даних. Найкращі результати показує на частково впорядкованих даних. Природність поводження.
Швидке	n*logn	log n	**не**стабільне	Один з найшвидших серед відомих алгоритмів сортування. В найгіршому випадку обчислювальна складність стає рівною n^2. При використанні додаткової пам’яті та вдосконалень можна зробити стабільним.

Завдання: Реалізувати алгоритм швидкого сортування Хоара. Порівняти час його роботи з часом роботи простих алгоритмів сортування при довжині масиву N = 10, 100, 1000, 10000, 30000. Результатом дослідження повинна стати таблиця (у звіті):

N	Кількість проходів				Кількість порівнянь				Кількість перестановок				Час			
	бульбашкове	вставками	вибором	швидке	бульбашкове	вставками	вибором	швидке	бульбашкове	вставками	вибором	швидке	пухирець	вставками	вибором	швидке

Додаткове завдання: Реалізувати один з перерахованих нижче алгоритмів та порівняти його швидкість з швидким сортуванням Хоара.

1. Сортування Шелла.
2. Пірамідальне сортування.
3. Порозрядне сортування.
4. Кишенькове сортування.
5. Сортування підрахунком.
6. Сортування комірками.
7. Сортування гребінцем.
8. Сортування перемішуванням.

Контрольні питання:

1. У чому полягає основна ідея швидкого сортування Хоара (яка стратегія використовується)?
2. Що таке базовий елемент (pivot) і які є способи його вибору?
3. Як працює операція розділення масиву: яку роль відіграють індекси l та r?
4. Назвіть найкращу, середню та найгіршу обчислювальну складність швидкого сортування. У якому випадку виникає $O(n^2)$?
5. Чи є швидке сортування стабільним? Скільки додаткової пам’яті воно зазвичай потребує порівняно з простими алгоритмами сортування?

Лабораторна робота № 5

Тема: Алгоритми пошуку

Мета: Навчитися реалізовувати алгоритми послідовного та бінарного пошуку елементів в масивах, а також алгоритм Бойера-Мура для пошуку рядків у тексті

Послідовний пошук

Нехай даний масив з N елементів. Необхідно визначити номер елемента, що має певні властивості (найчастіше мова йде просто про елемент із певним значенням, рівним k), або встановити що такого елемента в масиві немає.

Якщо масив є неупорядкованим, то єдиний алгоритм пошуку, що можна застосувати у цьому випадку - послідовний. Суть методу - послідовно перебираються всі елементи масиву, якщо на якомусь кроці циклу виявляється, що масив закінчився або виявляється шуканий елемент, то цикл закінчується.

Один з варіантів псевдокоду послідовного пошуку:

```
Алгоритм Послідовний пошук (масив a[1..N], шуканий елемент k)
Початок
  введення a[1..N], k
  i:=1
  поки (i<=N) та (a[i]<>k) виконувати inc(i);
  {після закінчення циклу і дорівнює номеру шуканого елемента, а якщо елемент не знайдений, то i=N+1}
Кінець
```

Використання циклу while замість for дозволяє скоротити кількість порівнянь, якщо потрібний елемент перебуває близько до початку масиву. У цьому випадку, як тільки елемент буде знайдений, цикл припинить свою роботу.

Бінарний пошук

Якщо масив упорядкований по зростанню або убуванню, то можна застосувати більш швидкі методи пошуку, такі як, наприклад, бінарний (методом розподілу навпіл).

Розглянемо приклад. Нехай даний масив з 9 елементів, упорядкований по зростанню їхніх значень.

2	4	5	8	9	16	22	28	36

Нехай потрібно знайти номер елемента, значення якого дорівнює 1. Порівняємо це значення з першим елементом масиву 1<2. З умови того, що масив упорядкований по зростанню слідує, що елементи з більшими номерами будуть мати більші значення, тобто інші числа в масиві гарантовано більше ніж 2. Звідси можна зробити висновок, що числа 1 у масиві немає.

Нехай потрібно знайти елемент зі значенням 55. Порівняємо дане число з першим елементом. 55>2. Отже, можна припустити, що в масиві число 55 зустрічається. Тепер порівняємо це число з останнім елементом масиву (з умови впорядкованості масиву, його останній елемент є найбільшим, а всі інші елементи відповідно менше його). 55>36. Отже, у масиві числа 55 бути не може.

Нехай потрібно знайти елемент зі значенням 22. Порівняємо це число із крайніми елементами масиву, і переконаємося, що таке число може бути присутнім серед його елементів.

Візьмемо елемент, що перебуває в середині масиву (якщо число елементів парне, то в яку сторону буде виконуватися округлення - неважливо).

2	4	5	8	9	16	22	28	36

Елемент, що знаходиться в середині масиву не той, який нам потрібний. (9<>22). Проте, цей елемент ділить масив на дві частини: одну в якій шуканого числа точно бути не може (у даному прикладі це ліва половина масиву; з умови впорядкованості всі елементи, що розташовані ліворуч від 9 будуть менше цього числа) і другу, у якій можна продовжувати пошук.

Розглянемо тепер праву частину масиву. Знайдемо елемент, що знаходиться в її середині.

2	4	5	8	9	16	22	28	36

Це значення дорівнює шуканому, процес пошуку закінчений.

Нехай потрібно знайти елемент зі значенням 7. Порівняємо це число із крайніми елементами масиву й переконаємося що пошук має сенс.

Розглянемо елемент, що перебуває в середині масиву.

2	4	5	8	9	16	22	28	36

Цей елемент не дорівнює шуканому, отже, пошук необхідно продовжити. Тому що 9>7, отже, праву половину масиву можна виключити з розгляду, і продовжити пошук тільки в лівій. Розглянемо елемент, що знаходиться в середині лівої половини:

2	4	5	8	9	16	22	28	36

Цей елемент знову не дорівнює 7, отже пошук треба продовжити. З умови впорядкованості масиву, пошук будемо продовжувати в правій частині нашого «укороченого» масиву.

2	4	5	8	9	16	22	28	36

Розглянемо елемент, що перебуває в середині виділеної частини масиву.

2	4	5	8	9	16	22	28	36

Цей елемент не дорівнює шуканому. Однак, у результаті постійного зменшення область пошуку в масиві зменшилася до трьох елементів, і кожний з

них був розглянутий на якомусь кроці. Отже, процес пошуку закінчений, результат - повідомлення про те, що елемент із таким значенням у масиві відсутній.

На цих прикладах видно, що з кожним кроком розмір області пошуку в масиві зменшується у два рази, що приводить до скорочення числа операцій. Наприклад, за 10 кроків розмір області пошуку буде зменшений в 1024 рази, а для масиву з 1000 000 елементів знадобиться всього 20 кроків. Кількість операцій у даному методі дорівнює O(log n).

Приклад псевдокоду бінарного пошуку може бути наступним:

```
Алгоритм Бінарний пошук (масив a[1..N], шуканий елемент k)
Початок
  введення a[1..N], k
  left:=1;
  right:=N;
  //відразу перевіримо, чи не є шуканий елемент крайнім
якщо a[1]=k тоді вививедення «1» //1 – номер шуканого елементу
інакше якщо a[N]=k тоді вививедення N // N номер шуканого елементу
// порівняємо із крайніми, вирішимо питання: а чи варто взагалі шукати?
інакше якщо  k<a[left] тоді вививедення «Шуканого числа немає»
інакше якщо  k>a[right] тоді вививедення «Шуканого числа немає»
//цикл пошуку: знайдемо номер у середині поточної області пошуку
інакше
        початок
          повторювати
            c:=(left+right) div 2
            якщо a[c]<k тоді left:=c
            якщо a[c]>k тоді right:=c
          умова виходу з циклу (a[c]=k) або ( right-left<=1);      //поки  елемент не знайдений, або поки область пошуку не стала складатися всього із двох сусідніх елементів
            якщо a[c]=k тоді виведення c // в c – номер шуканого елементу
            інакше «Шуканого числа немає»
    кінець
Кінець
```

Тут left і rght - номери елементів масиву, які є границями поточної області пошуку. Наприклад, якщо left=3 і right=7 це значить що пошук ведеться серед елементів з номерами від 3 до 7 і саме в цій області на даному кроці буде шукатися середина.

Алгоритм прямого (послідовного) пошуку підрядка

Позначимо рядок, у якому ведеться пошук як масив елементів T, шуканий підрядок - масив елементів W, N - довжина рядка, M – довжина підрядка.

Ідея алгоритму:

1) i=1;

2) порівняти i-й символ масиву T з першим символом масиву W;

3) збіг → порівняти наступний символ масиву T з другим символом масиву W й так далі;

4) розбіжність → i:=i+1 і перехід на пункт 2.

Умова закінчення алгоритму:

1) підряд M порівнянь вдалі;

2) i+M>N, тобто слово не знайдене.

Приклад псевдокоду прямого пошуку:

Алгоритм Прямий пошук (арг. S: рядок; арг. X: рядок; арг. Place: ціле число, рез. Res: логічний тип)

// Функція повертає результат пошуку у рядку S підрядка X

// Place – місце першого входження

```
Початок
  Res:=FALSE
  i:=1
  поки (i<=довжина(S)-довжина(X)+1) та (Res= FALSE) виконувати
      якщо S[i]=X[i] то
      початок
       …  //цикл порівняння символів в підрядку та рядку
       якщо всі символи збіглися то
      початок
         Res:=TRUE
         Place:=i
     кінець
    інакше i:=i+1
```

Виведення результів пошуку // якщо Res=TRUE - підрядок знайдений і в Place індекс першого символу підрадка в рядку, якщо Res=FALSE – підрядок не знайдено

Кінець

Алгоритм Бойера-Мура

Найпростіший варіант алгоритму Бойера-Мура складається з наступних кроків. На першому кроці ми будуємо таблицю зсувів для шуканого зразка. Процес побудови таблиці буде описаний нижче. Далі ми сполучаємо початок

рядка й зразка й починаємо перевірку з останнього символу зразка. Якщо останній символ зразка й відповідний йому при накладенні символ рядка не збігаються, зразок зсуваэться щодо рядка на величину, отриману з таблиці зсувів, і знову здійнюється порівняння, починаючи з останнього символу зразка. Якщо ж символи збігаються, здійнюється порівняння передостаннього символу зразка й т.д. Якщо всі символи зразка збіглися з накладеними символами рядка, значить ми знайшли підрядок й пошук закінчений. Якщо ж якийсь (не останній) символ зразка не збігається з відповідним символом рядка, ми зсуваємо зразок на один символ вправо й знову починаємо перевірку з останнього символу. Весь алгоритм виконується доти, поки або не буде знайдене входження шуканого зразка, або не буде досягнутий кінець рядка.

Величина зсуву у випадку розбіжності останнього символу обчислюється виходячи з наступних міркувань: зсув зразка повинен бути мінімальним, таким, щоб не пропустити входження зразка в рядку. Якщо даний символ рядка зустрічається в зразку, ми зсуваємо зразок таким чином, щоб символ рядка збігся із самим правим входженням цього символу в зразку. Якщо ж зразок взагалі не містить цього символу, ми зсуваємо зразок на величину, рівну його довжині, так що перший символ зразка накладається на наступний за перевірявшимся символ рядка.

Величина зсуву для кожного символу зразка залежить тільки від порядка символів у зразку, тому зсуви зручно обчислити заздалегідь і зберігати у вигляді одномірного масиву, де кожному символу алфавіту відповідає зсув щодо останнього символу зразка. Пояснимо все вищесказане на простому прикладі. Нехай у нас є алфавіт з п'яти символів: a, b, c, d, e і ми бажаємо знайти входження зразка “abbad” у рядку “abeccacbadbabbad”. Наступні схеми ілюструють всі етапи виконання алгоритму. Таблиця зсувів буде виглядати так.

a	b	c	d	e
1	2	5	0	5

Початок пошуку.

```
a b e c c a c b a d b a b b a d
a b b a d
```

Останній символ зразка не збігається з накладеним символом рядка. Зсуваємо зразок вправо на 5 позицій:

```
a b e c c a c b a d b a b b a d
          a b b a d
```

Три символи зразка збіглися, а четвертий - ні. Зсуваємо зразок вправо на одну позицію:

```
a b e c c a c b a d b a b b a d
            a b b a d
```

Останній символ знову не збігається із символом рядка. Відповідно до

таблиці зсувів зсуваємо зразок на 2 позиції:

```
a b e c c a c b a d b a b b a d
              a b b a d
```

Ще раз зсуваємо зразок на 2 позиції:

```
a b e c c a c b a d b a b b a d
                  a b b a d
```

Тепер, відповідно до таблиці, зсуваємо зразок на одну позицію, і одержуємо шукане входження зразка:

```
a b e c c a c b a d b a b b a d
                    a b b a d
```

Приклад псевдокоду обчислення таблиці зсувів (для алфавіту, що складається з 256 символів, тобто для таблиці ASCII-кодів):

Алгоритм Обчислення таблиці зсувів (Bmt: масив цілих чисел; p: рядок)

```
Початок
 для і від 0 до 255
   Bmt[i] := довжина(p)
 для і від довжина(p) до 1, крок -1
    якщо  Bmt[номер_символа(p[i])] = довжина(p) то
      Bmt[номер_символа(p[i])] := довжина(p) – i;
Кінець
```

Псевдокод алгоритму пошуку:

Алгоритм пошук БМ (арг. startpos: арг. ціле число; арг. s, p: рядки; арг. bmt: масив цілих чисел; рез res: ціле число)

```
змінні  pos, lp, i : цілі числа
початок
 lp := довжина(p)
 pos := startpos + lp –1
 поки pos < довжина(s) виконувати
  якщо p[lp] <> s[pos] то pos := pos + bmt[s[pos]]
  інакше для і від lp - 1 до 1, крок -1
   якщо p[i] <> s[pos – lp + i] то
   початок
     inc(pos)
     break
   кінець
   інакше якщо i = 1 то
   початок
     res := pos – lp + 1
     exit
```

```
    кінець
  res := 0
Кінець
```

Параметр StartPos дозволяє вказати позицію в рядку s, з якої варто починати пошук. Це може бути корисно в тому випадку, якщо ви захочете знайти всі входження p в s. Для пошуку із самого початку рядка варто задати StartPos рівним 1. Якщо результат пошуку не дорівнює нулю, то для того, щоб знайти наступне входження p в s, потрібно задати StartPos рівним значенню «попередній результат плюс довжина зразка».

Завдання 1:

1. Реалізувати алгоритм послідовного пошуку елемента в масиві.
2. Реалізувати алгоритм бінарного пошуку елемента в масиві.
3. Порівняти їх швидкодію.

Завдання 2:

1. Реалізувати алгоритм послідовного пошуку підрядка.
2. Реалізувати алгоритм пошуку Бойера-Мура.

Завдання 3 (додаткове): Реалізувати одне з наведених нижче завдань:

1. Є інформація про наявність місць у вагоні потяга (використати масив, наприклад, з 0 і 1). Вивести на екран номери вільних нижніх місць (нижні місця мають непарні номери). Та з'ясувати, чи можна придбати квитки на два поруч розташовані місця (розміщення в одній купі враховувати не обов'язково).

2. "Лотерея". У лотерейному квитку є 15 чисел від 1 до 90. Вони впорядковані по зростанню. Ведучий дістає шар з деяким випадковим числом. Визначити, чи є це число у квитку?

3. Вводитися послідовність із n натуральних чисел. Необхідно визначити найменше натуральне число, відсутнє в послідовності.

Контрольні запитання:

1. У чому полягає принцип послідовного пошуку в масиві? Коли його доцільно використовувати?

2. Чому для бінарного пошуку масив обов’язково має бути відсортований? Що відбувається з областю пошуку на кожному кроці?

3. Порівняйте орієнтовну кількість операцій: у якого алгоритму складність O(n), а у якого O(log n) – послідовного чи бінарного?

4. У чому основна ідея алгоритму Бойера–Мура при пошуку підрядка в рядку? Чому він зазвичай швидший за прямий пошук?

5. Яку роль відіграє таблиця зсувів у алгоритмі Бойера–Мура і коли відбувається її побудова – до пошуку чи під час нього?

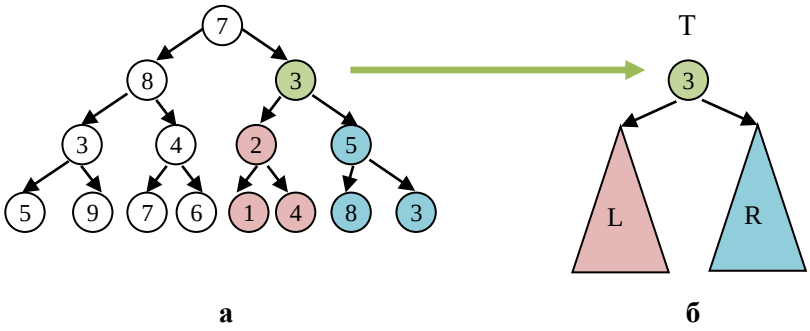
Лабораторна робота № 6

Тема: Алгоритми роботи з двійковими деревами

Мета: Навчитися реалізовувати алгоритми побудови та обходу двійкових дерев

Двійкове дерево – це динамічна структура даних, яка складається з вузлів, кожен з яких містить дані та посилання на двох своїх нащадків - лівого та правого.

Перший вузол дерева називається його коренем. Вузли, що не мають нащадків називаються листками. Кожний вузол в дереві задає піддерево, коренем якого являється.



а **б**

Рисунок 1 – Приклад двійкового дерева (а), приклад одного з його піддерев (б), де T - вибраний вузол - корінь піддерева, L - ліве піддерево, R - праве піддерево

Для роботи з двійковим деревом часто необхідно виконувати обхід його вузлів. Найбільш часто використовуються наступні способи обходу двійкового дерева:

1) Обхід в прямому напрямку (префіксний).
2) Обхід в зворотному напрямку (постфіксний).
3) Симетричний обхід (інфіксний).

Алгоритм прямого обходу дерева:

1) Відвідати корінь дерева.
2) Обійти ліве піддерево.
3) Обійти праве піддерево.

Алгоритм зворотного обходу дерева:

1) Обійти ліве піддерево.
2) Обійти праве піддерево.
3) Відвідати корінь дерева.

Алгоритм симетричного обходу дерева:

1) Обійти ліве піддерево.
2) Відвідати корінь дерева.
3) Обійти праве піддерево.

Приклад 1. Дано дерево, зображене на рисунку 2. Необхідно вказати в якому порядку будуть пройдені його вузли при прямому, зворотному та симетричному обходах.

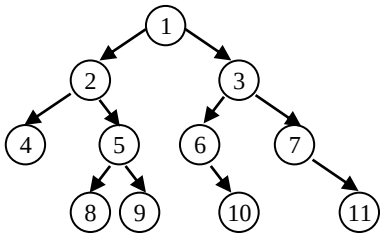


Рисунок 2

Рішення:

Прямий обхід: 1, 2, 4, 5, 8, 9, 3, 6, 10, 7, 11.

Зворотній обхід: 4, 8, 9, 5, 2, 10, 6, 11, 7, 3, 1.

Симетричний обхід: 4, 2, 8, 5, 9, 1, 6, 10, 3, 7, 11.

Приклад 2. Реалізація абстрактного типу даних двійкове дерево на мові програмування Pascal:

```
type
  TreeNode<T> = class
    data: T;
    left, right: TreeNode<T>;

    constructor (d: T; l, r: TreeNode<T>);
    begin
      data := d;
      left := l;
      right := r;
    end;
  end;
```

Приклад 3. Реалізація ідеально-збалансованого двійкового дерева, заповненого випадковими числами на мові програмування Pascal:

```
function CreateTree(n: integer): TreeNode<integer>;
begin
  if n <= 0 then
    Result := nil
  else
    Result := new TreeNode<integer>(
      Random(100),
      CreateTree((n-1) div 2),
      CreateTree(n - 1 - (n-1) div 2);
end;
```

Приклад 4. Реалізація інфіксного обходу на мові програмування Pascal:

```
procedure InfixPrintTree(root: TreeNode<integer>);
begin
  if root = nil then
    exit;
```

```
  InfixPrintTree(root.left);
  write(root.data, ' ');
  InfixPrintTree(root.right);
end;
```

Приклад 5. Реалізація префіксного обходу на мові програмування Pascal:

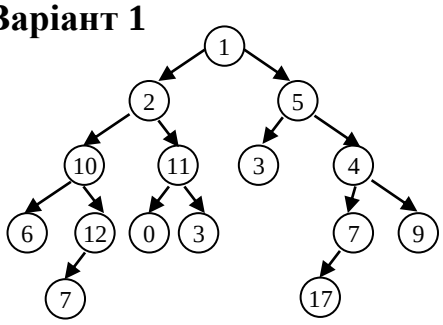
```
procedure PrefixTraverseTree(root: TreeNode<integer>; Action: IntAction);
begin
  if root = nil then
    exit;

  Action(root.data);
  PrefixTraverseTree(root.left, Action);
  PrefixTraverseTree(root.right, Action);
end;
```

Приклад 6. Реалізація постфіксного обходу на мові програмування Pascal:

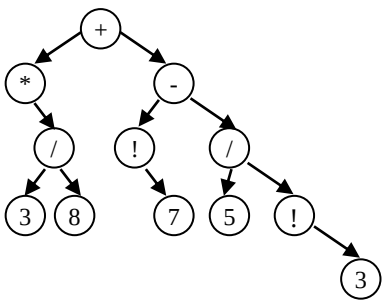
```
procedure PostfixTraverseTree(root: TreeNode<integer>; Action: IntAction);
begin
  if root = nil then
    exit;

  PostfixTraverseTree(root.left, Action);
  PostfixTraverseTree(root.right, Action);
  Action(root.data);
end;
```

Завдання: Реалізувати програмно абстрактний тип даних двійкове дерево та його екземпляр, структура якого відповідає зображеній у Вашому варіанті, заповнити вузли дерева випадковими числами. Реалізувати процедури обходу двійкового дерева в прямому, зворотному та симетричному порядку. При відвідуванні кожного вузла дерева виводити його значення на екран.

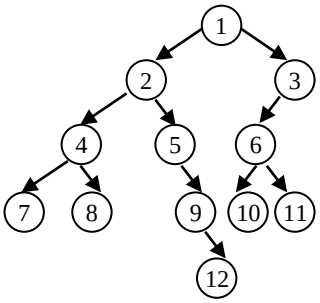
Варіант 1



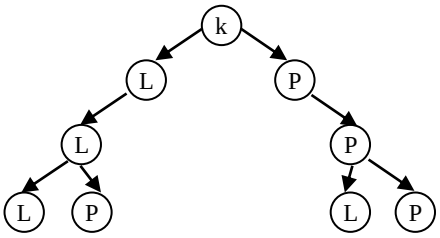
Варіант 2



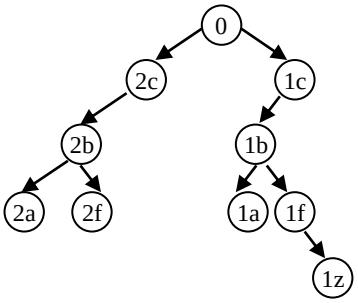
Варіант 3



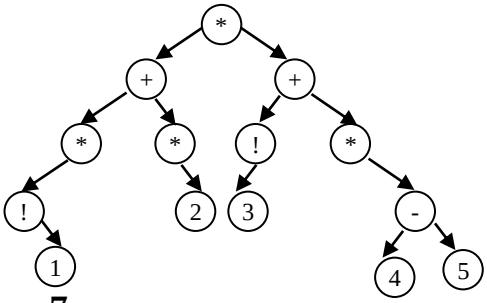
Варіант 4



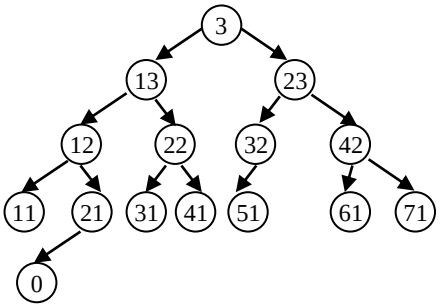
Варіант 5



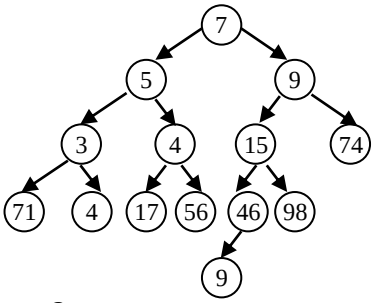
Варіант 6



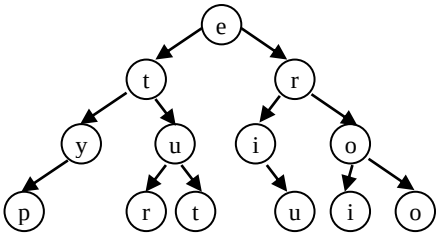
Варіант 7



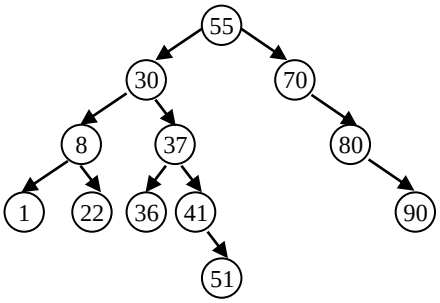
Варіант 8



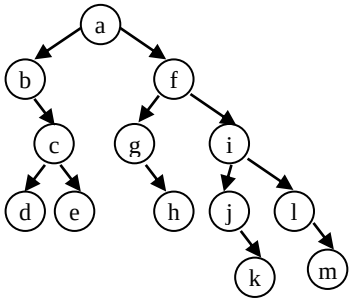
Варіант 9



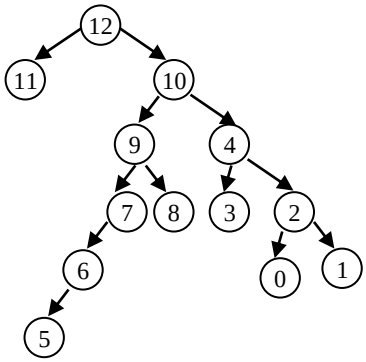
Варіант 10



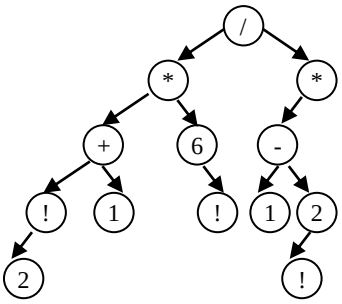
Варіант 11



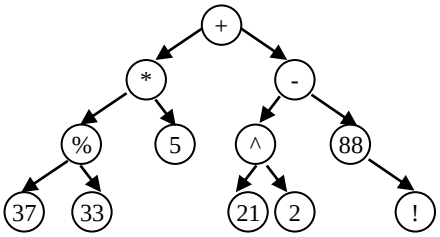
Варіант 12



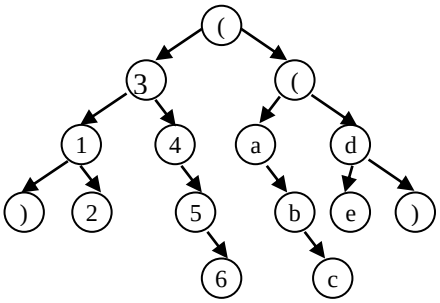
Варіант 13



Варіант 14



Варіант 15



Контрольні питання:

1. Що таке двійкове дерево? Дайте означення кореня, листка та піддерева.
2. Чим відрізняються префіксний, інфіксний та постфіксний обходи? Назвіть порядок відвідування: корінь – ліве – праве тощо.
3. При якому обході (префіксному, інфіксному чи постфіксному) корінь відвідується першим, середнім та останнім?
4. Як рекурсивна процедура обходу дерева використовує посилання left і right? Що вона робить у базовому випадку?
5. Для чого на практиці використовують інфіксний обхід двійкового дерева пошуку (BST)? Що можна отримати на виході?

Лабораторна робота № 7

Тема: Хеш-таблиці

Мета: Навчитися реалізовувати хеш-таблиці методом відкритого та закритого хешування.

Хеш-таблиця - це звичайний масив з незвичайною адресацією, що задається хеш-функцією.

Хеш-функція - функція, що трансформує ключ у деякий індекс у таблиці.

Ситуація, коли два або більше ключів асоціюються з однією й тією ж коміркою називається колізією при хешуванні.

Слід зазначити, що гарною хеш-функцією є така функція, що мінімізує колізії й розподіляє записи рівномірно по всій таблиці.

Хеш-таблиці часто застосовуються в базах даних, і, особливо, у мовних процесорах типу компіляторів і асемблерів, де вони вдало обслуговують таблиці ідентифікаторів. У таких додатках, таблиця - найкраща структура даних.

Метод відкритого хешування (інша назва: хешування ланцюжками)

У випадку хешування ланцюжками, елементи з однаковими індексами об'єднуються в зв'язаний список. Наступний рисунок ілюструє це.

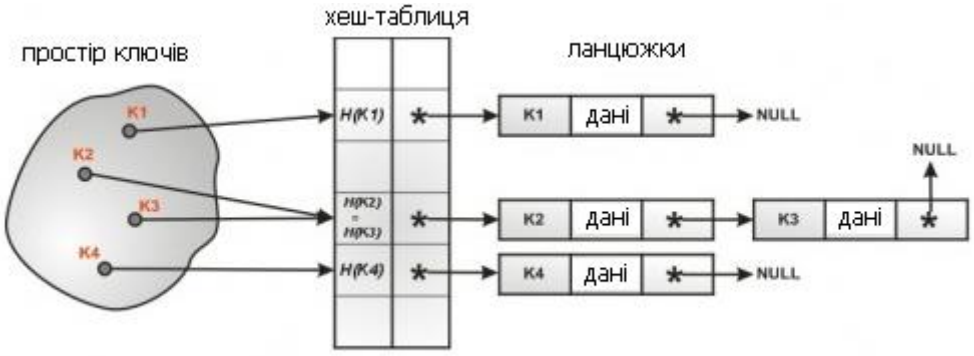


Рисунок 1 – Хешування ланцюжками

Тобто, якщо при додаванні в хеш-таблицю в задану комірку ми зустрічаємо посилання на елемент зв'язаного списку, то трапляється колізія. Тоді, ми просто вставляємо наш елемент як вузол у список. При пошуку ми проходимо по ланцюжках, порівнюючи ключі між собою на еквівалентність, поки не доберемося до потрібного. При видаленні ситуація така ж.

Видалення вузла з таблиці, що побудована за методом ланцюжків, полягає просто у виключенні вузла зі зв'язаного списку. Вилучений вузол ніяк не впливає на ефективність алгоритму пошуку. Алгоритм буде працювати так, ніби цей вузол ніколи не вставлявся в таблицю.

Псевдокод:

```
type
link = ^node;
node = record
key: integer;
st: string;
next: link;
end;
var
mas: array[0..9] of link;
function h(key: integer): integer;
begin
h:=key mod 10;
end;
function search(key1: integer; st1: string): link;
var
i: integer;
q, p, s: link;
begin
i:= h(key1);
q:=nil;
p:=mas[i];
while p <> nil do
begin
if p^.key = key1 then
begin
   search:=p;
   exit;
   end;
q := p;
p := p^.link;
end;
{Якщо ключ не знайдений, вставляємо новий запис}
new(s);
s^.key:=key1;
s^.st:=st1;
s^.next:=nil;
if q = nil then
mas[i]:=s
else
q^.next:=s;
search:=s;
end;
```

Метод закритого хешування (інша назва: відкритої адресації)

Закриті хеш-таблиці особливо ефективні, коли максимальні розміри вхідного набору даних вже відомі.

У випадку методу закритого хешування всі елементи зберігаються безпосередньо в хеш-таблиці, без використання зв'язаних списків. На відміну від хешування з ланцюжками, при використанні методу відкритої адресації може виникнути ситуація, коли хеш-таблиця виявиться повністю заповненою, так що буде неможливо додавати в неї нові елементи. Так що при виникненні такої ситуації рішенням може бути динамічне збільшення розміру хеш-таблиці, з одночасною її перебудовою.

Для розв’язання колізій застосовуються кілька підходів. Найпростіший з них - це метод лінійного дослідження. У цьому випадку при виникненні колізії наступні за поточною комірки перевіряються одна за іншою, поки не знайдеться порожня комірка, куди й записується елемент. А при досягненні останнього індексу таблиці, відбувається перехід на початок, тобто таблиця розглядається як «циклічний» масив. Ілюстрація цього способу представлена на наступному рисунку:

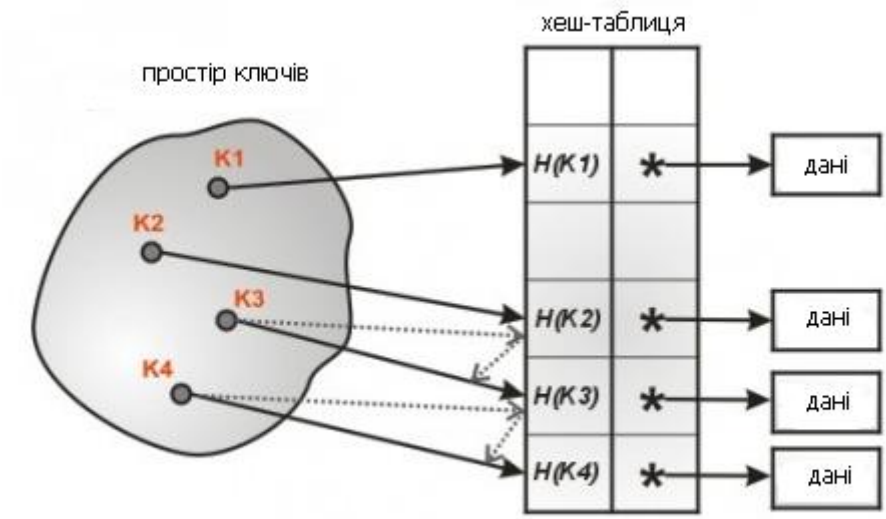


Рисунок 2 – Закрите хешування

Лінійне хешування досить просто реалізується, однак з ним зв'язана істотна проблема - кластеризація. Це явище створення довгих послідовностей зайнятих комірок, що збільшує середній час пошуку в таблиці. Для зниження ефекту кластеризації використовується інша стратегія вирішення колізій - подвійне хешування. Основна ідея полягає в тому, що для визначення кроку зсуву досліджень при колізії в комірці використовується інша хеш-функція, замість лінійного зсуву на одну позицію.

Розглянемо детальніше реалізацію даного методу. Уведемо два масиви:

```
val:  array [0.. n-1] of T;
used: array [0.. n-1] of boolean;
```

У цих масивах будуть зберігатися елементи множини: вона дорівнює множині всіх val [i] для тих i, для яких used [i], причому всі ці val [i] різні. По можливості будемо зберігати елемент t на місці h(t), будемо називати це місце «початковим» для даного елемента. Однак може трапитися так, що новий елемент, який ми хочемо додати, претендує на вже зайняте місце (для якого used істинне). У цьому випадку ми відшукаємо найближче праворуч вільне місце й запишемо елемент туди. ("Праворуч" значить "вбік збільшення індексів"; дійшовши до краю, ми перескакуємо на початок.) За припущенням, число елементів завжди менше n, так що порожні місця гарантовано будуть.

Формально говорячи, у будь-який момент повинна дотримуватися така вимога: для будь-якого елемента множини ділянка праворуч від його «початкового» місця до його фактичного місця повністю заповнене.

Завдяки цьому перевірка приналежності заданого елемента t здійснюється легко: вставши на h(t), рухаємося праворуч, поки не дійдемо до порожнього місця або до елемента t. У першому випадку елемент t відсутній у множині, у другому є присутнім. Якщо елемент відсутній, то його можна додати на знайдене порожнє місце. Якщо є присутнім, то можна його видалити (поклавши used = false).

Одним зі складних питань реалізації хешування з відкритою адресацією є операція видалення елемента. Справа в тому, що при видаленні елемента необхідна властивість "відсутності порожнеч" може порушитися. Тому будемо робити так. Створивши пусте місце, будемо рухатися праворуч, поки не натрапимо на ще одне порожнє місце (тоді на цьому можна заспокоїтися) або на елемент, що стоїть не на «початковому» місці. У другому випадку подивимося, чи не потрібно цей елемент поставити на порожнє місце. Якщо ні, то продовжуємо пошук, якщо так, то закриваємо їм порожнє місце, що виникнуло після операції видалення. При цьому утвориться нова діра, з якою робимо все те ж саме.

Псевдокод:

```
Var
   K: array [0...999] of integer;
Function h(key: integer): integer;
Begin
   h := key mod 1000;
End;
Function rh(i: integer): integer;
Begin
   rh:=i+1 mod 1000;
End;
Procedure insert(key: integer);
Var
   I: integer;
begin
I := h(key); {хешуємо ключ}
```

```
while ((k(i)< >key) and (k(i)< >0)) do
   i := rh(i); {ми повинні виконати повторне хешування}
if k(i) =0 then {вставляємо запис у порожню позицію}
k(i)=key
end;
```

Вибір хеш-функції

Звернемося тепер до питання про те, як вибрати гарну хеш-функцію. Ясно, що ця функція повинна створювати якнайменше колізій при хешуванні, тобто вона повинна рівномірно розподіляти ключі на наявні індекси в масиві. Звичайно, не можна визначити, чи буде деяка конкретна хеш-функція розподіляти ключі правильно, якщо ці ключі заздалегідь не відомі. Однак, хоча до вибору хеш-функції рідко відомі самі ключі, деякі властивості цих ключів, які впливають на їхній розподіл, звичайно відомі.

1) метод ділення. Деякий цілий ключ ділиться на розмір таблиці й залишок від ділення береться як значення хеш-функції. Ця хеш-функція позначається h (key) := key mod m.

2) метод середини квадрата. Ключ множиться сам на себе і як індекс використовується декілька середніх цифр цього квадрата.

```
Function h(key: integer): integer;
Begin
Key:=key*key; {Піднести до квадрата}
Key:=key shl 11;{Відкинути 11 молодших бітів}
H:= key mod 1024;{Повернути 10 молодших бітів}
End;
```

3) адитивний метод для рядків (розмір таблиці дорівнює 256). Для рядків цілком розумні результати дає додавання всіх символів і повернення залишку від ділення на 256.

```
Function h(st: string): integer;
Var
Sum: longint;
I: integer;
Begin
For i:=0 to length(st) do
Sum := sum + ord(st[i]);
H:=sum mod 256;
End;
```

4) виключаюче АБО для рядків (розмір таблиці дорівнює 256). Цей метод аналогічний адитивному, але успішно розрізняє схожі слова й анаграми (адитивний метод дасть одне значення для XY і YX). Метод полягає в тому, що до елементів рядка послідовно застосовується операція "виключаюче або". В алгоритм додається випадковий компонент, щоб ще поліпшити результат.

```
var
rand8: array[0..255] of integer;
```

```
procedure init;
var
i: integer;
begin
randomize;
for i:=0 to 255 do
rand8[i]:=random(255);
end;
function h(st: string): integer;
Var
Sum: longint;
I: integer;
Begin
For i:=0 to length(st) do
Sum := sum + ord(st[i]) xor rand8[i];
H:=sum mod 256;
end;
```

Завдання 1:

1. Реалізувати хеш-таблицю методом відкритого хешування.
2. Реалізувати хеш-таблицю методом закритого хешування.

Завдання 2 (додаткове):

Перевірка орфографії. Написати програму, що перевіряла б правильність введених слів, використовуючи запропонований словник. Словник зчитується з текстового файлу. Для організації словника використовувати хеш-таблицю. Розв'язання колізій здійснювати методом ланцюжків. Слова для перевірки вводяться із клавіатури. Якщо перевіряємо слово вірне (тобто є в словнику), то виводиться відповідне повідомлення. Якщо слово неправильне, то виводиться повідомлення про помилку та всі можливі варіанти його заміни, якщо вони є. Для варіантів заміни використовувати список адресуємий відповідним рядком хеш-таблиці.

Контрольні питання:

1. Що таке хеш-таблиця і яку роль відіграє хеш-функція?
2. Що називають колізією при хешуванні? Наведіть один приклад способу її розв'язання.
3. У чому різниця між відкритим хешуванням (ланцюжками) та закритим хешуванням (відкритою адресацією)?
4. Як працює лінійне дослідження (linear probing) при закритому хешуванні? Яка його основна проблема?
5. Які властивості має мати «гарна» хеш-функція для цілих чисел або рядків (не менше двох)?

СПИСОК ЛІТЕРАТУРИ

1. Коваленко О. О., Ткаченко О. М., Чехместрук Р. Ю. Алгоритми та структури даних : навчальний посібник (електронне видання). Вінниця : ВНТУ, 2025. – 113 с. https://pdf.lib.vntu.edu.ua/books/2025/Kovalenko_2025_113.pdf
2. Креневич А. П. Алгоритми і структури даних : підручник. Київ : ВПЦ «Київський університет», 2021. – 200 с. https://www.mechmat.univ.kiev.ua/wp-content/uploads/2021/09/pidruchnyk-alhorytmy-i-struktury-danykh.pdf
3. Кублій Л. І. Алгоритми та структури даних. Основи алгоритмізації : підручник. Київ : КПІ ім. Ігоря Сікорського, 2022. – 528 с. https://ela.kpi.ua/handle/123456789/48282
4. Мелешко Є. В., Якименко М. С., Поліщук Л. І. Алгоритми та структури даних : навч. посіб. / М-во освіти і науки України, Центральноукраїн. нац. техн. ун-т. - Кропивницький : Лисенко В.Ф., 2019. – 156 с. https://dspace.kntu.kr.ua/handle/123456789/8944
5. Knuth D. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd Edition 3rd Edition. – Addison-Wesley Professional, 2019. – 672 p.
6. Knuth D. The Art of Computer Programming: Vol. 3: Sorting and Searching 2nd Edition, Kindle Edition. – Addison-Wesley Professional, 2019. – 800 p.
7. Knuth D. Art of Computer Programming, Vol. 2: Seminumerical Algorithms 3rd Edition, Kindle Edition. – Addison-Wesley Professional, 2019. – 672 p.
8. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms, 3rd Edition (The MIT Press) 3rd Edition – The MIT Press, 2019. – 1292 p.
9. Lutz M. Learning Python, 5th Edition Fifth Edition. - O'Reilly Media, 2016. - 1643 p.
10. Lutz M. Python: Pocket Reference Fourth Edition. - O'Reilly Media, 2016. - 210 p.
11. McKinney W. Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter 3rd Edition. - O'Reilly Media, 2022. - 579 p.
12. Алгоритми та структури даних (комп'ютерний практикум) : навч. посібник / уклад. Ю. Є. Грудзинський. Київ : КПІ ім. Ігоря Сікорського, 2022. – 100 с. https://ela.kpi.ua/server/api/core/bitstreams/0db974f9-16fa-459c-9f19-fab0021222ed/content
13. Бугаєва Л. М., Ковалюк Д. О. Алгоритми та структури даних. Комп'ютерний практикум : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. – 34 с. https://ela.kpi.ua/items/c9756f3a-e61f-4068-8d88-bd1e1dd950a9

14. Бульба С. С., Бречко В. О., Далека В. Д. Алгоритми та структури даних : навч.-метод. посібник. Харків : НТУ «ХПІ», 2021. – 141 с. https://repository.kpi.kharkov.ua/handle/KhPI-Press/54935
15. Савеленко О. К., Лисенко І. А., Іванченко О. О. CASE-технології у проектуванні інформаційних систем: навчальний посібник / Мін-во освіти і науки України, Центральноукраїн. нац. техн. ун-т. - Кропивницький: Видавець Лисенко В.Ф., 2018. – 240 с. https://dspace.kntu.kr.ua/handle/123456789/10278
16. https://www.codeproject.com/ – колективний блог з новинами та навчальними статтями про інформаційні технології та програмування.
17. http://stackoverflow.com/ – система питань і відповідей для професійних програмістів та новачків у програмуванні.
18. https://dou.ua/ – український веб-сайт з елементами колективного блогу, створений для розповсюдження новин, аналітичних статей та свіжої інформації пов'язаної із інформаційними технологіями.
19. http://www.algomation.com/ – це платформа для перегляду, обміну і створення візуалізацій алгоритмів.
20. https://prometheus.org.ua/ – українська платформа безкоштовних онлайн-курсів
21. http://moodle.kntu.kr.ua/ – Дистанційна освіта ЦНТУ.
22. http://www.tutorialspoint.com/python/ – Tutorialspoint / Python
23. https://docs.python.org/ – Python's documentation, tutorials, and guides are constantly evolving

Додаток 1
Приклад оформлення звіту з лабораторної роботи

Міністерство освіти і науки України
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Лабораторна робота №_
з дисципліни «Алгоритми та структури даних»
на тему:
_____

Виконав:
студент групи _____
_____
(П.І.Б.)

Перевірив:
викладач
_____
(П.І.Б.)

Кропивницький 20__

Тема: _____
Мета: _____

Варіант №_

Завдання: _____
_____
_____
_____
_____

Хід роботи:

<Повинен містити теоретичні дані, скриншоти розробленого програмного забезпечення та лістинг>

Відповіді на контрольні питання:
1.
2.
3.
…

Додаток 2
Приклад оформлення лістингу програми

Лістинг програми

Файл Lab_2_1.cs

```
// (c) Іваненеко І.І., ЦНТУ, 2025
// дисципліна «Алгоритми та структури даних»
// лабораторна робота №2, варіант 12, завдання 1

using System;
using System.Text;

namespace ConsoleApplication9
{
class Program
    {
        //метод для введення цілих чисел із клавіатури
staticint ReadInt(string prompt)
        {
                Console.Write(prompt);
int x = int.Parse(Console.ReadLine());
return x;
        }
        //метод для виведення масиву на екран
staticvoid PrintArray(int[] array)
        {
for (int i = 0; i < array.Length; i++)
            {
                Console.Write("{0,5} ", array[i]);
            }
            Console.WriteLine();
        }

staticvoid Main(string[] args)
        {
int N = ReadInt("Введіть розмірність масиву: ");

int[] a = newint[N];

            //введемо елементи масиву з клавіатури
for (int i = 0; i < N; i++)
            {
                try
                {
                    a[i] = ReadInt("Введіть " + (i+1).ToString() +
                         "-й елемент масиву: ");
                }
catch(FormatException)
                {
                    Console.WriteLine("Невірний формат числа!");
                }
            }
```

…

Додаток 3
Шкала оцінювання: національна та ECTS

№	Сума балів за всі види навчальної діяльності	Оцінка ЄКТС	Оцінка за національною шкалою для екзамену
1	**90-100**	**A**	«відмінно»
2	**82-89**	**B**	«добре»
3	**74-81**	**C**	
4	**64-73**	**D**	«задовільно»
5	**60-63**	**E**	
6	**35-59**	**FX**	«незадовільно» з можливістю повторного складання
7	**1-34**	**F**	«незадовільно» з обов'язковим повторним вивченням дисципліни

Критерії оцінювання. Еквівалент оцінки в балах для кожної окремої теми може бути різний, загальну суму балів за тему визначено в навчально-методичній карті. Розподіл балів між видами занять (лекції, практичні заняття, самостійна робота) можливий шляхом спільного прийняття рішення викладача і студентів на першому занятті:

1) оцінку **«відмінно» (90-100 балів, A)** заслуговує студент, який:

- всебічно, систематично і глибоко володіє навчально-програмовим матеріалом;
- вміє самостійно виконувати завдання, передбачені програмою, використовує набуті знання і вміння у нестандартних ситуаціях;
- засвоїв основну і ознайомлений з додатковою літературою, яка рекомендована програмою;
- засвоїв взаємозв'язок основних понять дисципліни та усвідомлює їх значення для професії, яку він набуває;
- вільно висловлює власні думки, самостійно оцінює різноманітні життєві явища і факти, виявляючи особистісну позицію;
- самостійно визначає окремі цілі власної навчальної діяльності, виявив творчі здібності і використовує їх при вивченні навчальнопрограмового матеріалу, проявив нахил до наукової роботи.

2) оцінку **«добре» (82-89 балів, B)** – заслуговує студент, який:

- повністю опанував і вільно (самостійно) володіє навчально-програмовим матеріалом, в тому числі застосовує його на практиці, має системні знання достатньому обсязі відповідно до навчально-програмового матеріалу, аргументовано використовує їх у різних ситуаціях;
- має здатність до самостійного пошуку інформації, а також до аналізу, постановки і розв'язування проблем професійного спрямування;
- під час відповіді допустив деякі неточності, які самостійно виправляє, добирає переконливі аргументи на підтвердження вивченого матеріалу.

3) оцінку **«добре» (74-81 бал, C)** - заслуговує студент, який:

- в загальному роботу виконав, але відповідає на екзамені з певною кількістю помилок;

- вміє порівнювати, узагальнювати, систематизувати інформацію під керівництвом викладача, в цілому самостійно застосовувати на практиці, контролювати власну діяльність;

- опанував навчально-програмовий матеріал, успішно виконав завдання, передбачені програмою, засвоїв основну літературу, яка рекомендована програмою.

4) оцінку **«задовільно» (64-73 бали, D)** – заслуговує студент, який:

- знає основний навчально-програмовий матеріал в обсязі, необхідному для подальшого навчання і використання його у майбутній професії; - виконує завдання, але при рішенні допускає значну кількість помилок;

- ознайомлений з основною літературою, яка рекомендована програмою;

- допускає на заняттях чи екзамені помилки при виконанні завдань, але під керівництвом викладача знаходить шляхи їх усунення.

5) оцінку **«задовільно» (60-63 бали, E)** – заслуговує студент, який:

- володіє основним навчально-програмовим матеріалом в обсязі, необхідному для подальшого навчання і використання його у майбутній професії, а виконання завдань задовольняє мінімальні критерії. Знання мають репродуктивний характер.

6) оцінка **«незадовільно» (35-59 балів, FX)** – виставляється студенту, який:

- виявив суттєві прогалини в знаннях основного програмового матеріалу, допустив принципові помилки у виконанні передбачених програмою завдань.

7) оцінку **«незадовільно» (35 балів, F)** – виставляється студенту, який:

- володіє навчальним матеріалом тільки на рівні елементарного розпізнавання і відтворення окремих фактів або не володіє зовсім;

- допускає грубі помилки при виконанні завдань, передбачених програмою;

- не може продовжувати навчання і не готовий до професійної діяльності після закінчення університету без повторного вивчення даної дисципліни.