МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

І. М. Кузьменко,
О. А. Дацюк

Базові алгоритми та структури даних

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці
спеціальності 121 «Програмна інженерія»

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент *Голінко І.М., к. т. н., доцент каф. АТЕП*

Відповідальний редактор *Гагарін О.О., к. т. н., доцент*

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол №6 від 24.06.2022 р.)
за поданням Вченої ради ТЕФ
(протокол № 8 від 31.05.2022 р.)

Вирішення прикладних задач в області інформаційних технологій потребує адаптації інформації із загального опису на математичну основу, потім як алгоритм, і далі - на мову програмування, або навпаки, в зворотному порядку. Знання способів побудови ефективних алгоритмів, використання структур даних знадобляться всім, хто буде ефективно аналізувати інформацію та створювати конкурентні програмні продукти. Поряд з розглядом теоретичних питань, навчальний посібник містить теорію та лабораторні завдання, які допоможуть зрозуміти основні поняття алгоритмів та практичні принципи структур даних.

Навчальний посібник також призначений для студентів та для спеціалістів в області інформаційних технологій, які вивчають алгоритми та структури даних самостійно.

Реєстр № …… обсяг 5 авт.арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

Вступ

Алгоритм – це набір певних правил, ефективний спосіб вирішення певної обчислювальної задачі. Або, по-іншому, алгоритм – це ідея на основі якої написано працюючий код.

Наприклад, алгоритм реалізує задачу: 1) розставити числа у послідовності, 2) отримати найкоротший шлях від А до Б, 3) упорядкувати розклад.

Навіщо вивчають алгоритми. По-перше, алгоритми та структури даних це частина комп'ютерних наук.

Наприклад: 1) Робота мережевого роутера опирається на алгоритм пошуку найкоротшого маршруту.

2) Ефективність шифрування з відкритим ключем залежить від алгоритмів шифрування.

3) Комп'ютерна графіка використовує примітиви, отримані на основі геометричних алгоритмів.

4) Бази даних ґрунтуються на деревах пошуку.

По-друге, алгоритми - це частина know-how.

Наприклад: пошукові машини використовують ряд алгоритмів для визначення релевантності веб-сторінок. Визнано, що ріст продуктивності роботи за рахунок покращення алгоритмів перевищує ріст за рахунок підвищення швидкодії машин, процесорів.

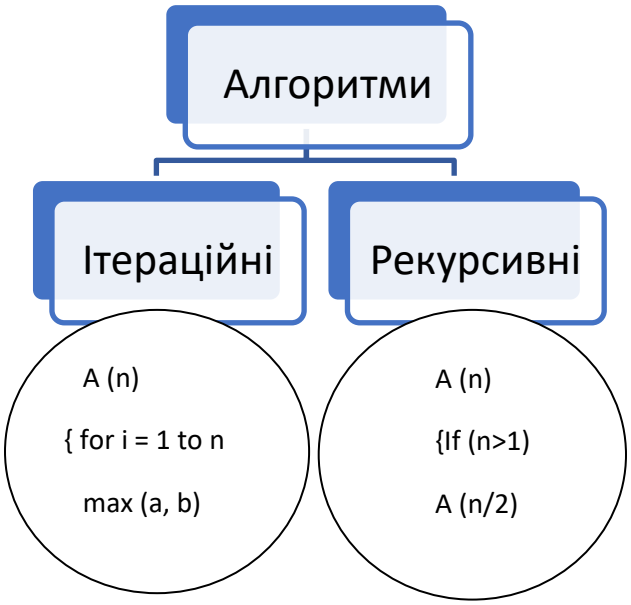
По третє, алгоритми використовують також в економіці, медицині. Біологія, наприклад, використовує алгоритми для визначення схожості геномів.

І останнє – вивчення та розробка алгоритмів потребує ретельності та винахідливості.

І саме останнє – посібник написано в співавторстві: розділ 3 Дацюк О.А., інше - Кузьменко І.М.

1. Ітераційні алгоритми та їх часова складність

Всі алгоритми для оцінки їх складності розділимо на ітераційні та рекурсивні. Приклади таких алгоритмів показано на рисунку 1.1. Розглянемо ітераційні алгоритми та їх часову складність. Приклади, наведені нижче, написані з використанням псевдокоду – коду мови, якої не існує. Вважаємо їх типовими прикладами і розглянемо детальніше.



Для оцінки часової складності оцінимо як залежить кількість символів, які друкуються, від параметрів циклу.

В прикладі 1 питання - скільки буде виведено * при роботі циклу з n елементів має відповідь - сніжинок буде n. Тому 0(n).

Рисунок. 1.1. Приклади алгоритмів

Приклад 1 A ( )

```
{int i,
        for (i=1 to n)
        Print (‘*’)
}
```

В прикладі 2 розглянемо часову складність алгоритму цикл в циклі. В цьому разі кількість сніжинок буде n^2 і часова складність $O(n^2)$

Приклад 2 A ( )

```
{int i, j
        for (i=1 to n)   - n
        for (j=1 to n)   - n
                Print (‘*’)
}
```

В наступному 3 прикладі обраховано суму S для цього цикл з ітератором і виконувався k разів. В таблиці 1.1 показано зв'язок між сумою S та кількістю ітерацій і.

Приклад 3

```
A ( )
{int i=1, S=1
while (S<=n)
        {i++ ,
        S = S +i,
        Print (*)
        }
}
```

Таблиця 1.1. Зв'язок між сумою S та кількістю ітерацій і.

S	1	3	6	10	15	…	n
i	1	2	3	4	5	…	K

Сума членів арифметичної прогресії $\frac{k(k+1)}{2} = S = n$

$$\frac{k^2 + k}{2} = n$$

Домінуюча функція $k^2 = n$. Звідси $k = \sqrt{n}$ і часова складність $O(\sqrt{n})$.

В прикладі 4 розглянуто цикл, обмежений квадратом ітератора і. Кількість сніжинок, які цикл надрукує, складає k. Як видно з таблиці 1.2 $k = \sqrt{n}$. Тобто, часова складність даного циклу $O(\sqrt{n})$. Таблиця 1.2. Зв'язок між сумою S та кількістю ітерацій і

Приклад 4

```
A ( )
{int i=1
for (i=1, i²≤n, i++)
        Print ('*')
}
```

Таблиця 1.2. Зв'язок між кількістю ітерацій і та друком *

Значення i	1	4	9	16	…	n
Друк	*	*	*	*		Всього $k = \sqrt{n}$

В прикладі 5 розглянемо часову складність алгоритму з потрійним циклом. В цьому разі для підрахунку сніжинок нижче наведено таблицю 1.3.

Приклад 5

```
A ( )
{int i, j, k, n
for (i=1, i<=n, i++)
  {for (j=1, j<=i, j++)
    {for (k=1, k <= 100, k++)
      Print ('*');
      }
   }
}
```

Таблиця 1.3. Зв'язок між ітераторами та кількістю сніжинок *

i=1	i=2	i=3	i=4	i=5	i=6	i=n
j=1 раз	j=2 рази	j=3	j=4	j=5	j=6	j=n
k=100 разів	k=100·2	k=100·3	k=100·4	k=100·5	k=100·6	k=100·n

Кількість сніжинок складе

$$100 + 100 \cdot 2 + 100 \cdot 3 + 100 \cdot 4 \ldots + 100 \cdot n = 100 \cdot (1 + 2 + 3 + \ldots + n) = 100\left(\frac{n(n+1)}{2}\right)$$

$$= 50n^2 + 50n$$

Тобто, домінуюча функція для $50n^2 + 50n$ буде n^2 і, відповідно, часова складність $O(n^2)$.

В прикладі 6 розглянемо часову складність алгоритму з потрійним циклом. В цьому разі для підрахунку сніжинок наведено таблицю 1.4.

В прикладі 6 розглянемо часову складність алгоритму з потрійним циклом.

Приклад 6

```
A ( )
{int i, j, k, n
for (i=1, i<=n, i++)
  {for (j=1, j<=i^2, j++)
    {for (k=1,kj<=n/2, k++)
      {Print (*);
      }
    }
  }
```

Для підрахунку кількості сніжинок занесемо дані для кожного з циклів у таблицю 1.4.

Таблиця 1.4. Зв'язок між ітераторами та кількістю сніжинок

i=1	i=2	i=3		i=n
j=1 раз	$j=2^2$ рази	j=9		$j=n^2$
$k = \frac{n}{2}$	$k = \frac{n}{2}$	$k = \frac{n}{2}$	···	$k = \frac{n}{2}$
Сніжинок $\frac{n}{2} \cdot 1$	$\frac{n}{2} \cdot 4$	$\frac{n}{2} \cdot 9$	···	$\frac{n}{2} \cdot n^2$

Всього сніжинок буде надруковано:

$$\frac{n}{2}\cdot 1+\frac{n}{2}\cdot 4+\frac{n}{2}\cdot 9+\cdots\frac{n}{2}\cdot n^2=\frac{n}{2}(1+4+9+\cdots n^2)$$

З використанням формули суми квадратів для виразу у дужках, отримаємо:

$$\frac{n}{2}\cdot\left[\frac{n(n+1)(2n+1)}{6}\right]=\frac{n}{2}\frac{1}{6}(2n^3+n+3n^2)=\frac{1}{12}(2n^4+3n^3+n^2)$$

В останньому виразі домінуюча функція n^4 визначає часову складність O(n^4).

В прикладі 7 розглянемо часову складність циклічного алгоритму де ітератор зростає мультиплікативно. Підрахунок сніжинок наведено нижче в таблиці 1.5.

Приклад 7

```
A ( )
{for (i=1, i<n, i=i·2)
  Print ('*');
}
```

Таблиця 1.5. Зв'язок між ітератором, кроком k та кількістю сніжинок

k	0	1	2	3	…	k
i=	1	2	4	8	...	n
сніжинок	*	*	*	*	...	*

Як видно з таблиці 1.5, сніжинок всього (k+1), тобто цикл виконається (k+1) разів. Також з таблиці видно, що $2^k \Rightarrow n$. Звідси $k=\log_2 n$ і часова складність алгоритму для функції $\log_2 n+1$ складе O(log n).

На основі аналогічних викладок, якщо в циклі i=i·3, маємо часову складність алгоритму для функції $\log_3 n+1$ рівну O(log n).

Якщо в циклі i=i·10, то часова складність алгоритму для функції $\log_{10} n+1$ буде рівною O(log n). Тобто, в цілому O(log n).

В прикладі 8 розглянемо часову складність алгоритму з потрійним циклом. В цьому разі для підрахунку часової складності розглянемо кожен із циклів.

Приклад 8

```
A ( )
{int i, j, k
for (i=n/2, i<=n, i++)                  // цикл буде виконано n/2 разів
  for (j=1, j<=n/2, j++)    -           // цикл буде виконано n/2 разів
    for (k=1, k<=n, k=k·2)              // цикл буде виконано log2 n + 1 разів
      Print (*)
    }
```

За аналогією з прикладом 1, перші два цикли будуть виконуватися по $\frac{n}{2}$ разів кожен. Останній цикл, за аналогією з прикладом 7, буде виконано $\log_2 n + 1$ Тобто, всього сніжинок буде надруковано $\frac{n}{2}\frac{n}{2}(\log_2 n + 1)$. Визначимо домінуючу функцію та отримаємо нотацію Ландау для даного прикладу $O(n^2 \cdot \log n)$

В прикладі 9, за аналогією з прикладом 8, розглянемо часову складність алгоритму з потрійним циклом. В цьому разі для підрахунку часової складності розглянемо кожен із циклів.

Приклад 9

```
A ( )
{int i, j, k
for (i=n/2, i<=n, i++)            // цикл буде виконано n/2 разів
  for (j=1, j<=n, j=2·j)  // цикл буде виконано log2 n + 1 разів
    for (k=1,k<=n, k=k·2)        // цикл буде виконано log2 n + 1 разів
      Print ('*')
    }
```

За аналогією з прикладом 1, перший цикл буде виконуватися $\frac{n}{2}$ разів. Останні два цикли, за аналогією з прикладом 7, буде виконано $(\log_2 n + 1)$ разів. Тобто, всього сніжинок буде надруковано $\frac{n}{2} \cdot \log_2 n \cdot \log_2 n = \frac{n}{2}(\log_2 n)^2$. Тому, нотація Ландау для даного прикладу має вигляд $O(n \cdot (\log n)^2)$

В прикладі 10, за аналогією з прикладом 8, розглянемо часову складність циклічного алгоритму. Тут n виступає як визначник кроку алгоритму.

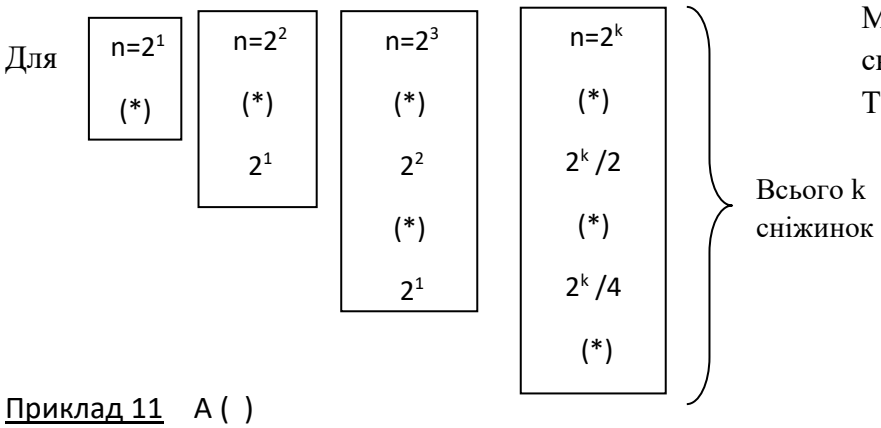
Приклад 10

```
A ( )                    //для n≥2
{while (n>1)
 {n=n/2
   Print (*)  } }
```

Обрахуємо кількість сніжинок, залежно від n. Результати показано на рисунку 1.2. Звідси - часова складність даного алгоритму складає O(log n).

Аналогічним чином, якщо n=n/5, то домінуюча функція $\log_5 n$. Якщо n=n/10, то домінуюча функція $\log_{10} n$, що не змінює часову складність даного алгоритму - O(log n).

В прикладі 11, розглянемо часову складність алгоритму цикл в циклі. Перший цикл буде виконано рівно n разів. Другий цикл має змінний ітератор j. Тут і виступає як визначник кроку алгоритму.



Маємо для n = 2^k кількість сніжинок буде k.
Тобто, k = $\log_2$ n

Приклад 11

```
A ( )
{for (i=1, i<=n, i++)
  for (j=1, j<=n, j=j+i)
    Print (*),
  }
```

Аналіз даного псевдокоду наведено в таблиці 1.6 нижче.

Таблиця 1.6. Зв'язок між ітераторами та кількістю сніжинок

i=	1	i=2	i=3	…	i=k	i=n
j=	1 … n	j=1 … n	j=1 … n	…	j=1 … n	j=1 … n
Сніжинок	n	$\frac{n}{2}$	$\frac{n}{3}$	...	$\frac{n}{k}$	$\frac{n}{n}$

Всього сніжинок буде надруковано $n+\frac{n}{2}+\frac{n}{3}+\cdots+\frac{n}{k}+\frac{n}{n}=n(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k}+\frac{1}{n})$.

Оскільки в дужках – гармонічне число $H_n=\sum_{k=1}^{n}\frac{1}{k}$, що має асимптотичну границю $H_{n,1}=\gamma+\ln(x)$ то отримуємо значення нотації Ландау $O(n\log n)$.

В прикладі 12, розглянемо часову складність алгоритму цикл в циклі. Перший цикл буде виконано рівно n разів. Другий цикл має змінний ітератор j, який виступає як визначник кроку алгоритму.

Як видно з таблиці 1.7, якщо k=1 маємо·2n сніжинок, якщо k=2 маємо·3n сніжинок, якщо k=3 маємо·4n сніжинок.

Маємо, для k буде надруковано (k+1)n сніжинок, де $n=2^{2^k}$. Звідси

$\log_2 n=2^k$ і $k=\log_2(\log_2 \mathrm{n})$

$n(\log_2(\log_2 \mathrm{n})+1)=\mathrm{n}\log_2(\log_2 \mathrm{n})+\mathrm{n}$.

Домінуючий доданок у функції визначає нотацію Ландау для даного прикладу $O(n\log(\log n))$

Приклад 12 A ( )

```
{int n=2^(2^k)                    // n – const, що залежить від k.
for (i=1, i<=n, i++)              //цикл виконається n  разів
  {int j=2
  while (j<=n)
        {j=j^2
        Print (*);
        } }    }
```

Таблиця 1.7. Зв'язок між ітераторами та кількістю сніжинок

	k=2	k=3
n=4	n=16	$n=2^8=256$
j=2, 4	j=2, 4, 16	j=2, 2^2,2^4,2^8
$\underbrace{**}$	$\underbrace{***}$	$\underbrace{****}$
2n сніжинок	3n сніжинок	4n сніжинок

2. Рекурсивні алгоритми та їх часова складність

Часова складність рекурсивних алгоритмів зводиться до обрахунку часу через розв'язання рекурентного співвідношення. В свою чергу рекурентне співвідношення отримується з аналізу алгоритму з використанням математичної індукції. В загальному випадку псевдокод функції, що виконується рекурсивно, містить базовий випадок рекурсії та виклик власне рекурсивної функції має вигляд:

```
A(n)
{If (.....)
Return (A(n/2)+A(n/2))
}
```

Отримаємо рекурентне співвідношення наступним чином: якщо час виконання оператора розгалуження If (який описує базовий випадок рекурсії) складає C, і час виконання рекурсивного виклику 2T(n/2), то час виконання наведеного псевдокоду складе T(n) = C +2 T(n/2).

Розглянемо наступні три приклади. Перший приклад наведено нижче

Приклад 1

```
A(n)                    /виклик A(n)
{ If (n>1)              //базовий випадок рекурсії
  Return (A(n-1))       //рекурсивний виклик A(n-1)
}
```

Визначимо рекурентне відношення.

Маємо час виконання T(n)=1+T(n-1). Далі – зворотна підстановка:

T(n-1)=1+T(n-2), T(n-2)=1+T(n-3).

Отримаємо:

T(n)=1+1+T(n-2)=1+1+1+T(n-3)=3+ T(n-3)

В результаті, на основі математичної індукції:

$$T(n)=k+T(n-k) \qquad (2.1)$$

Умова зупинки рекурсії – базовий випадок:

T(n)=k+T(n-k)=1+T(n-1); n>1.

Оскільки для n=1 виконається лише оператор If (n>1). Тобто T(n)=1 для n=1.

В загальному випадку, для n=1: T(n-k) = T(1) звідси n-k=1, або k=n-1

Підставимо в формулу (2.1) і маємо зворотну підстановку:

T(n)=k+T(n-k) = n-1+T(n-n+1) = n-1+T(1)=n-1+1

Тобто, нотація Ландау для даного прикладу O(n).

На рисунку 2.1 показано розміщення викликів рекурсивної функції. Виклики рекурсивної функції фіксуються в пам’яті комп’ютера, або точніше - в частині операційної пам’яті комп’ютера, що називається «стек». Власне «стек» - це своєрідна структура даних, що розміщаються лише чітко впорядковано і ця структура крім викликів рекурсивної функції, може зберігати також інші дані. В цілому всі дані, що зберігаються у стеку (включно з викликами рекурсивних функцій), описуються загальним правилом. Це загальне правило описує порядок їх впорядкування - «хто останнім зайшов той першим вийшов» або “last in first out” (LIFO).

Відповідно до правила LIFO на рисунку нижче показано зберігання викликів рекурсивних функцій у вигляді дерева рекурсії. На рисунку зліва показано, що функція A(n), виклик якої знаходиться у стеку за загальним правилом LIFO не може бути обрахована, оскільки вона звертається до функції A(n-1). У свою чергу функція A(n-1), виклик якої теж знаходиться у стеку теж не може бути обрахована, оскільки звертається до функції A(n-2) і т.д. Це відбувається до моменту настання базового випадку рекурсії n=1. Цей процес носить назву прямий хід рекурсії.

Після настання базового випадку рекурсії за правилом LIFO звільняється пам'ять стека, зайнята під останню рекурсивну функцію. Дані передаються в передостанню функцію, яка починає виконуватися і звільняє стек і т.д. Цей процес носить назву зворотний хід рекурсії. Як видно з рисунка нижче, у стеку в найгіршому випадку буде знаходитися (n-1) функція. Тобто, можна сказати, що використання алгоритмом пам’яті, і розмір пам’яті, яка використовується, прямо залежить від кількості даних n.

Таким чином, оцінюючи алгоритм, можна говорити також і про просторову складність (англ. Space complexity).

Як видно з нижченаведеного дерева рекурсії на рисунку 2.1, розмір пам’яті буде функцією від (n-1) або – з використанням нотації Ландау - просторова складність O(n). Дерево рекурсії для Прикладу 1

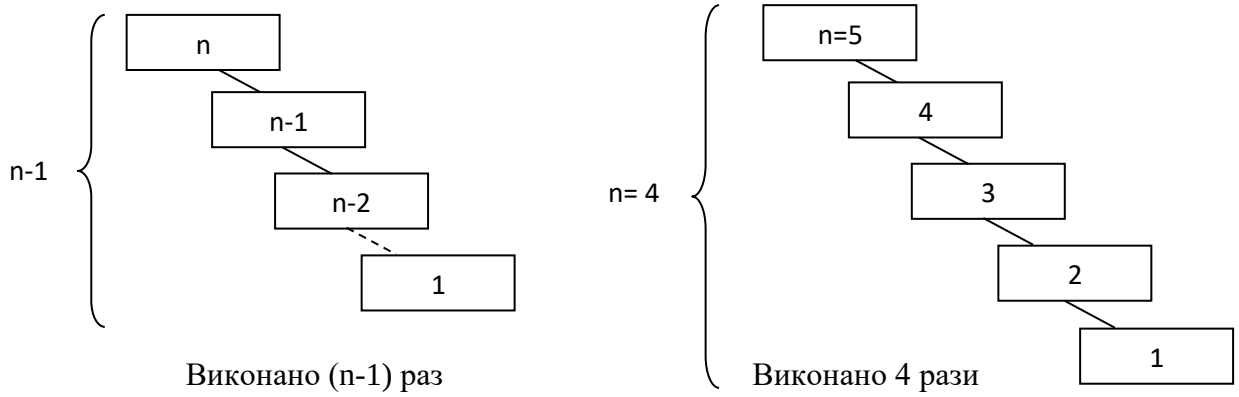


Рисунок 2.1. Розміщення викликів рекурсивної функції

Зокрема, справа показано, що для рекурсивної функції A(n=5) маємо 4 рекурсивних функції, які розміщаються в стеку та базовий випадок рекурсії. Це ілюструє просторову складність O(n) для даного прикладу.

Розглянемо приклад 2 – обрахунок факторіала з використанням рекурсивної функції та визначимо часову та просторову складність даного алгоритму.

Алгоритм обрахунку факторіалу наведено нижче у прикладі 2 як псевдокод.

Приклад 2

```
Factorial (n)                    // виклик Factorial(n)
{If n=0
        Return 1                 // базовий випадок рекурсії
else
        Return n factorial (n-1) // рекурсивний виклик Factorial(n-1)
}
```

Для обрахунку часової складності визначимо рекурентне відношення. Для n змінних час виконання:

$$T(n)=C+T(n-1).$$

І далі:

$$T(n-1) =C+T(n-2),$$

$$T(n-2) =C+T(n-3).$$

$$T(n)=3\cdot C+T(n-3)$$

Маємо рекурентне відношення:

$$T(n) =k\cdot C+T(n-k)$$

Базовий випадок рекурсії за k = n дає T(0)=1 для вищевказаного алгоритму. Отримуємо:

$$T(n)= n\cdot C+T(0) =n\cdot C+1.$$

Таким чином, часова складність даного алгоритму обрахунку факторіалу O(n).

Визначимо просторову складність рекурсивного алгоритму обрахунку факторіалу. Для цього збудуємо дерево рекурсії функції F(5), яка обраховує 5! (рисунок 2.2).

Як видно на дереві рекурсії простежується прямий та зворотний хід. Зокрема, прямий хід рекурсії заносить у стек виклики рекурсивних функцій (ліві гілки дерева), позначені на рисунку як F(5), F(4), .., F(0).

Від базового випадку рекурсії (0!=1) починається зворотний хід (праві гілки дерева).

На наступному рисунку 2.3 показано заповнення стеку при прямому ході рекурсії. Стрілочками вправо показані результати, отримані при зворотному ході рекурсії.

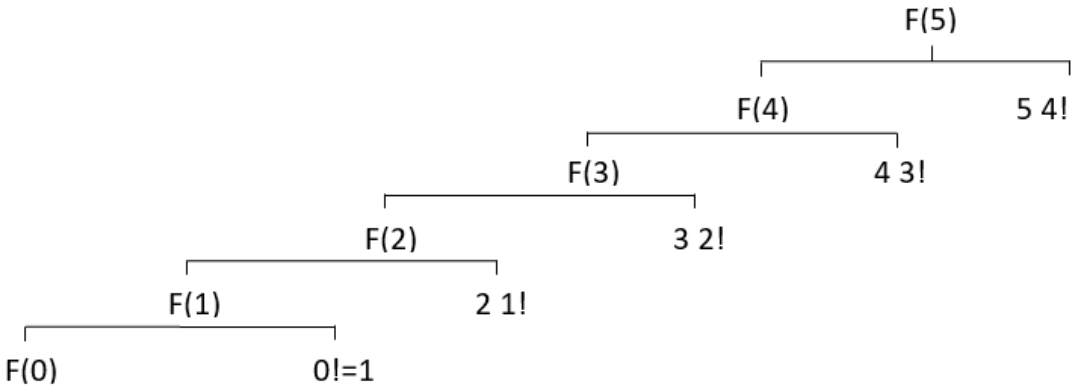


Рисунок 2.2. Дерево рекурсії

Стрілочками вниз показано передачу результатів на попередній рівень рекурсії.

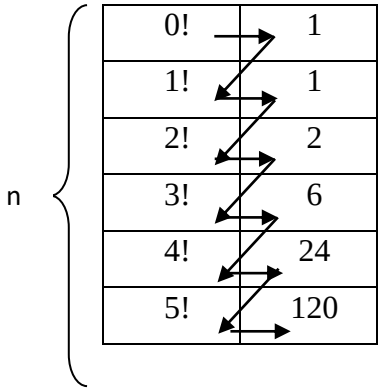


Рисунок 2.3. Заповнення стеку при прямому ході рекурсії

Як видно з цього рисунку, найбільша глибина стеку для обрахунку 5! складає 5 (оскільки базовий випадок 0!=1 звільняє стек), або, в цілому, для n! глибина стеку (n).

Тобто, в цілому просторова складність рекурсивного алгоритму обрахунку n! визначається висотою дерева рекурсії (n), що відповідає глибині заповнення стеку. Тобто, нотація Ландау складає O(n).

Розглянемо приклад 3 – обрахунок факторіала з використанням рекурсивної функції та визначимо часову і просторову складність даного алгоритму.

Наступний приклад 3 наведено нижче у вигляді псевдокоду. Від наведених раніше, приклад 3 відрізняється тим, що містить ітераційну та рекурсивну частини коду.

Приклад 3

```
Recursive (n)
{For (i = 0, i < n, i = +2)
```

```
        {Print (*) }                    // ітерація, виконується n/2 разів
{If ( n < = 0)
        Return1;                        // базовий випадок рекурсії
  else
        Return Recursive (n-5);         // рекурсивний виклик
  }
```

Для обрахунку часової складності визначимо рекурентне відношення. Для n змінних, якщо n>0 маємо час виконання:

$$T(n) = \frac{n}{2} + 1 + T(n-5).$$

Далі:

$$T(n-5) = \frac{n-5}{2} + 1 + T(n-5-5) = \frac{n}{2} - \frac{5}{2} + 1 + T(n-10)$$

$$T(n-10) = \frac{n-10}{2} + 1 + T(n-10-5) = \frac{n}{2} - \frac{10}{2} + 1 + T(n-15)$$

Підставимо T(n-5), T(n-10) в T(n):

$$T(n) = \frac{n}{2} + 1 + \overbrace{\frac{n}{2} - \frac{5}{2} + 1}^{T(n-5)} + \overbrace{\frac{n}{2} - \frac{10}{2} + 1}^{T(n-10)} + T(n-15) == 3 \cdot \frac{n}{2} + const + T(n - 3 \cdot 5)$$

Маємо рекурентне відношення:

$$T(n) = k \cdot \frac{n}{2} + T(n - k \cdot 5) + const$$

Базовий випадок рекурсії має місце, якщо:

$$T(n - k \cdot 5) = T(0),$$

оскільки з псевдокоду видно, що $T(0)$ приводить до виконання 1 операції (Return 1):

$$T(0) = 1$$

Звідси:

$n - k \cdot 5 = 0$ та $k = n/5$.

Тоді рекурентне відношення має вигляд $T(n) = \frac{n}{5}\,\frac{n}{2} + 1 + const = \frac{n^2}{10} + const,$

і часова складність рекурсії O(n^2).

Оскільки отримана часова складність рекурсивної частини псевдокоду функції, наведеної в прикладі 3 є вищою, ніж часова складність ітераційної частини псевдокоду O(n^2) > O(n), то загальна часова складність функції визначається часовою складністю рекурсії O(n^2).

Визначимо просторову складність функції, наведеної в прикладі 3. Просторова складність ітераційної частини функції складає O(1), оскільки потребує виділення лише пам'яті для 2 змінних i, n та не залежить від розміру масиву. Тут i – ітератор та n – кількість елементів у масиві.

Просторову складність рекурсивної частини функції визначимо, за висотою дерева рекурсії функції Recursive(15), наприклад. Оскільки розглядаємо лише прямий хід рекурсії, відповідно до прикладу 3, то дерево рекурсії на кожному рівні має лише ліву гілку. Зображення дерева рекурсії для n=15 дано на рисунку 2.4.

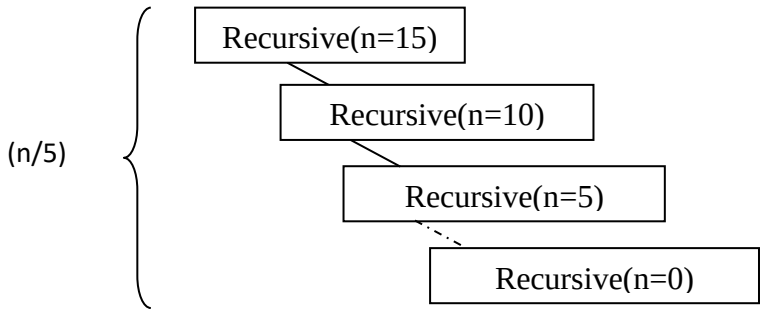


Рисунок 2.4. Дерево рекурсії для прикладу 3

Як видно з рисунка 2.4, висота дерева рекурсії для даного прикладу складає 15/5 =3, оскільки базовий випадок відразу дає результат. В цілому це дає функцію, що визначає найбільшу висоту дерева рекурсії (n/5). Звідси просторова складність рекурсивної частини функції із даного прикладу O(n).

І оскільки просторова складність рекурсивної частини функції із даного прикладу O(n) перевищує просторову складність ітераційної частини функції із даного прикладу O(1), то для функції в цілому маємо просторову складність O(n).

Розглянемо приклад 4 – обрахунок чисел Фібоначчі з використанням рекурсивної функції та визначимо часову та просторову складність даного алгоритму.

Числа Фібоначчі широко поширені в природі – спіралі на шишках, насіння на соняшнику, морські черепашки слідують за шаблоном чисел Фібоначчі. Перелік перших 11 чисел наведено в таблиці 2.1.

Таблиця 2.1. Числа Фібоначчі

n	0	1	2	3	4	5	6	7	8	9	10
fib(n)	0	1	1	2	3	5	8	13	21	34	55

Формула для обрахунку чисел Фібоначчі *Fib(n)*=$\begin{cases} n, \text{ для } n = 0, \text{або } n = 1 \\ f(n-1) + f(n-2), n > 1 \end{cases}$

Алгоритм обрахунку чисел Фібоначчі, що реалізує вищевказану формулу з використанням рекурсії показано у прикладі 4, як псевдокод.

Приклад 4

```
Fib(n)
{If (n < = 1)                    // базовий випадок рекурсії
Return n;
else
Return (Fib(n-1)+Fib(n-2));      // рекурсивний виклик функції
}
```

Для обрахунку часової складності алгоритму, отримаємо рекурентну формулу часу виконання алгоритму для n змінних:

$T(n)=C+T(n-1)+T(n-2),$ (2.2)

де C – константа, кількість часу на виконання лінійних операцій.

Припустимо, що час виконання приблизно рівний у наступних рекурсивних функцій:

$T(n-1) \approx T(n-2)$ (2.3)

Тоді формулу (2.2) можна переписати так:

$T(n) = C + 2T(n-2)$ (2.4)

Тут:

$T(n-2) = C + T(n-3) + T(n-4).$ (2.5)

За аналогією з припущенням (2.3):

$T(n-3) \approx T(n-4).$

Тобто, формулу (2.4) перепишемо з урахуванням (2.5) так:

$$T(n) = C + 2[C + 2T(n-4)] = 3C + 4T(n-4)$$

Дана формула отримана для другого рівня рекурсії, тому k = 2.

Для наступного рівня рекурсії (k = 3) маємо:

$$T(n-4) = C + T(n-5) + T(n-6).$$

За аналогією $T(n-5) \approx T(n-6)$, тому:

$$T(n) = 3C + 4[C + 2T(n-6)] = 7\,C + 8 \cdot T(n-6)$$

Для наступного рівня рекурсії (k = 4) маємо:

$$T(n - 6) = C+T(n-7)+T(n-8),$$

$$T(n-7) \approx T(n-8)$$

$$T(n)= 15\,C+16 \cdot T(n-8)$$

На основі цього напишемо рекурентне відношення, для k рівня рекурсії:

$$T(n) = (2^k-1) \cdot C + 2^k \cdot T(n-2k). \quad (2.6)$$

Оскільки базовий випадок рекурсії:

$$T(0) = T(1) = C, \quad (2.7)$$

$$\text{то } T(n-2k) = T(0), \quad (2.8)$$

звідси:

$$n-2k = 0,$$

$$k = n/2$$

Підставимо останню залежність у (2.6) і врахуємо (2.7, 2.8), маємо:

$$T(n) = (2^k-1) \cdot C + 2^k \cdot T(n-2k) = (2^{n/2}-1) \cdot C + 2^{n/2} \cdot T(0) = (2^{n/2}-1) \cdot C + 2^{n/2} \cdot C = = 2^{n/2} \cdot 2C-C.$$

Якщо з останнього виразу виділити домінуючу функцію, отримаємо нотацію Ландау для рекурсивного алгоритму обрахунку чисел Фібоначчі в найкращому випадку $O(2^{n/2})$.

2.1. Часова та просторова складність алгоритму визначення чисел Фібоначчі

Обрахуємо числову складність алгоритму визначення чисел Фібоначчі у найгіршому випадку. Для цього знову визначимо рекурентне відношення.

Однак, при цьому приймемо допущення:

$$T(n-2) \approx T(n-1).$$

Тоді формулу (2.4) можна записати наступним чином:

$$T(n) = C+2T(n-1), \text{ для } k = 1. \quad (2.9)$$

Далі:

$$T(n-1) = C+2T(n-2), \text{ для } k = 2 \quad (2.10)$$

$$T(n-2) = C+2T(n-3), \text{ для } k = 3 \quad (2.11)$$

Підставивши в (2.9) формули (2.10) та (2.11) отримаємо:

$$T(n) = 7C + 8T(n-3)$$

Тобто, маємо рекурентне відношення:

$$T(n) = (2^k-1)\cdot C + 2^k\cdot T(n-k). \quad (2.12)$$

Із базового випадку рекурсії маємо:

$$T(0) = C. \quad (2.13)$$

Тобто:

$$T(n-k)=T(0). \quad (2.14)$$

Звідси n-k=0, n=k.

Підставимо у рекурентну формулу (2.12) і отримаємо:

$$T(n) = (2^n-1)\cdot C + 2^n\cdot T(n-k)$$

З урахуванням (2.13) і (2.14):

$$T(n) = (2^n-1)\cdot C + 2^n\cdot C = 2C\cdot 2^n - C.$$

Домінуюча функція дозволяє визначити нотацію Ландау $O(2^n)$ для часової складності рекурсивного алгоритму обрахунку чисел Фібоначчі у найгіршому випадку.

Як видно з нотації Ландау для рекурсивного обрахунку чисел Фібоначчі, часова складність алгоритму є значною, що відносить алгоритм до класу NP – не поліноміальних алгоритмів.

NP алгоритми мають значний - експоненційний час виконання на відміну від P алгоритмів, що виконуються за поліноміальний час, який менший, ніж експоненційний.

Для розуміння значної часової складності алгоритму рекурсивного обрахунку чисел Фібоначчі зобразимо дерево рекурсії для функції Fib(5) на рисунку 2.5.

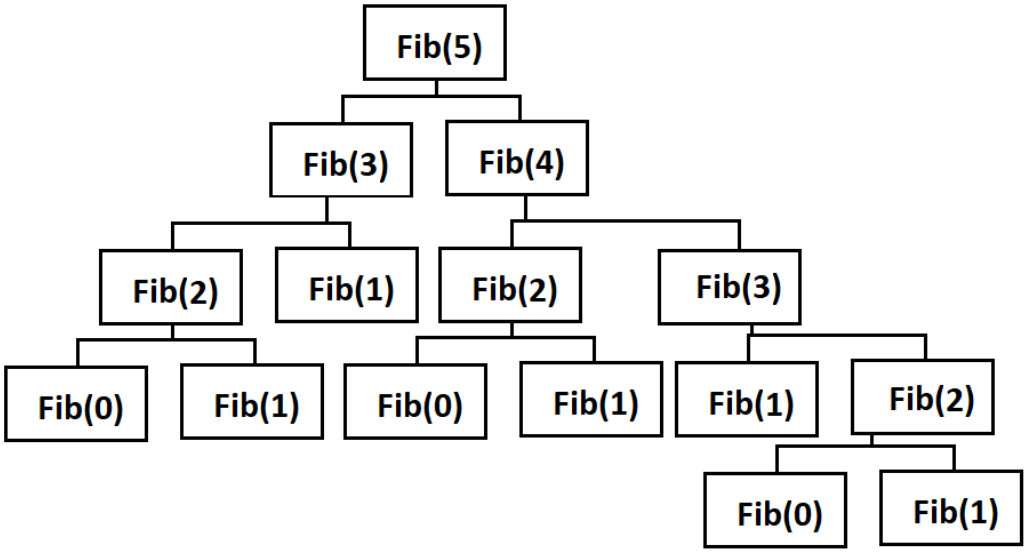


Рисунок 2.5. Дерево рекурсії для обрахунку чисел Фібоначчі

Як видно з наведеного дерева рекурсії, щоб алгоритму обрахувати одну функцію Fib(5), потрібно, окрім іншого, чотири рази обрахувати Fib(1) та три рази Fib(0). Звідси і витікає така значна часова складність алгоритму рекурсивного обрахунку чисел Фібоначчі.

Визначимо просторову складність алгоритму рекурсивного обрахунку чисел Фібоначчі.

Розглянемо вищенаведене дерево рекурсії алгоритму обрахунку чисел Фібоначчі. Як видно з рисунку, найбільша висота дерева рекурсії відповідає гілці дерева, що містить функції Fib(5), Fib(4), Fib(3), Fib(2), Fib(1). Тут Fib(1) – базовий випадок рекурсії.

Тобто, з вищенаведеного рисунку дерева рекурсії видно, що для n=5 - висота дерева, що відповідає глибині стека рекурсії, буде 5 в максимальному, найгіршому випадку.

В цілому для n висота дерева буде n. Таким чином, просторова складність алгоритму рекурсивного обрахунку чисел Фібоначчі складає O(n).

2.2. Часова складність ітераційного алгоритму обрахунку чисел Фібоначчі

Для порівняння з рекурсивним розглянемо ітераційний алгоритм обрахунку чисел Фібоначчі, псевдокод якого наведено нижче.

```
Fibo (n)
{A0=0
  A1=1
  Print (A0)
  Print (A1)
  for (i=2, i≤n, i++)
          {A2=A0+A1
           Print (A2)
           A0=A1
           A1=A2
          }
}
```

Оцінимо часову складність даного алгоритму. Якщо кожна команда виконується за 1 одиницю часу, то часова складність для n змінних:

$$T(n) = 4 + \underbrace{4 + 4 + 4 + 4 + 4 \ldots + 4}_{n-2 \text{ разів}} = 4 + 4(n-2) = 4n + 2\,,$$

де (n-2) кількість ітерацій в циклі.

Домінуюча функція вказує, що алгоритм ітераційного обрахунку чисел Фібоначчі має часову складність O(n).

Оцінимо просторову складність алгоритму ітераційного обрахунку чисел Фібоначчі.

Оскільки в пам'яті розміщаються 4 змінні i, A0, A1, A2 то завантаження пам'яті є сталим і не залежить від кількості n змінних. Тобто, просторова складність алгоритму ітераційного обрахунку чисел Фібоначчі O(1).

В цілому, якщо порівняти числову та просторову складність алгоритму ітераційного обрахунку чисел Фібоначчі та алгоритму рекурсивного обрахунку чисел Фібоначчі, то видно, що алгоритм ітераційного обрахунку чисел Фібоначчі має переваги. Його часова (O(n)< O(2n)) та просторова складності менші (O(1) < O(n)), ніж у рекурсивного.

2.3. Лабораторна робота. Рекурсивні та ітераційні алгоритми

Для закріплення матеріалу пропонується нижченаведена робота.

Мета роботи – реалізувати завдання з використанням ітераційного алгоритму та з використанням рекурсії та порівняти ці алгоритми.

В ході роботи реалізувати наступні задачі:

1)Відповідно до варіанту реалізувати завдання у двох варіантах: з використанням ітераційного алгоритму та з використанням рекурсії.

2)Збільшуючи кількість елементів (глибину рекурсії), отримати переповнення стеку (Stack Overflow).

3) Визначити час виконання ітераційного алгоритму та алгоритму з використанням рекурсії для кількості елементів 100, 5 000 та для n, близьких до Stack Overflow .

4) За результатами зробити висновки – який з алгоритмів працює краще і чому. Описати обрані алгоритми.

Завдання для роботи

1.В одномірному масиві знайти суму додатних елементів, розміщених до першого від'ємного елемента.

2.Для заданого рядка, надрукувати голосні у прямому та приголосні у зворотному порядку.

3.Вивести всі додатні елементи масиву, потім всі від'ємні у зворотному порядку.

4.Перевернути останнє слово заданого речення.

5.Підрахувати кількість цифр у заданому реченні.

6.Знайти добуток ненульових елементів масиву.

7.Вивести в зворотному порядку літери другого слова речення.

8.Піднести число у цілу степінь.

9.Вивести в зворотному порядку цифри цілого числа.

10. Вивести символи першого слова рядку у прямому та зворотному порядку.

11. Знайти суму членів ряду 1+1/2+1/3+…+1/n.

12. Вивести символи другого слова рядку у прямому та зворотному порядку.

13. Вивести символи першого слова , яке починається на "а".

14. Вивести голосні речення у прямому та зворотному порядку.

15. Вивести знаки арифметичних операцій, які входять в рядок символів, у зворотному порядку.

16. Вивести парні, потім непарні в зворотному порядку цифри введеного числа.

17. Для одномірного масиву вивести елементи у прямому, а потім у зворотному порядку.

18. Підрахувати кількість входжень в рядок сполучення prog.

19. Вилучити з тексту всі входження слова ***по***.

20. Видалити зайві пропуски в реченні, щоб слова розділялись лише одним.

21. Вилучити із третього слова усі попередні входження останньої літери.

22. В одномірному масиві знайти суму від'ємних елементів, розміщених до першого додатного елемента.

23. Вивести всі від'ємні елементи масиву, потім всі додатні у зворотному порядку.

24. Вставити в текст після всіх входжень слова ***по*** число «1».

3. Алгоритми сортування

Сортування – процес переставлення об’єктів множини у визначеному порядку з метою полегшення наступного пошуку елементів у відсортованій множині.

Елементи сортування є практично в усіх задачах (телефонні книги, відомості, звіти). Найчастіше сортування використовують для подальшого пошуку даних.

Існує багато різних алгоритмів сортування, причому вибір алгоритму залежить від структури даних, тому і методи сортування поділяють на такі категорії: сортування масивів та сортування файлів.

Основна відмінність між сортуванням масивів і сортуванням послідовних файлів полягає в тому, що кожен елемент масиву є доступним у будь-який час. Це значить, що в будь-який час будь-який елемент масиву може порівнюватися з будь-яким іншим елементом масиву і будь-які два елементи масиву можуть мінятися місцями. Напроти, у послідовному файлі в будь-який момент часу досяжний лише один елемент. Через ці відмінності методи сортування істотно відрізняються для цих двох видів сортування.

У загальному випадку при сортуванні даних тільки частина інформації використовується як ключ сортування, що використовується для порівнянь. Коли виконується обмін, передається вся структура даних. Наприклад, у списку поштових відправлень як ключ сортування може використовуватися поштовий індекс, а в операціях обміну до поштового індексу додаються повне ім'я й адреса.

Практично кожен алгоритм сортування можна розбити на три частини:

- порівняння, що визначає впорядкованість пари елементів;
- перестановку, що змінює місцями пару елементів;
- власне сортуючий алгоритм, що здійснює порівняння і перестановку елементів доти, поки всі елементи множини не будуть впорядковані.

3.1. Сортування масивів

Основна вимога до методів сортування – економне витрачання пам’яті, тому сортування треба виконувати in situ (на місці) тобто такими методами, які практично не потребують додаткової пам'яті, роблячи майже усі перестановки всередині масиву.

Одна з головних характеристик алгоритму – це час його роботи. Час роботи алгоритму залежить від кількості необхідних порівнянь.

Прості алгоритми для сортування N елементів потребують час роботи пропорційний N^2, у той час як більш складні алгоритми використовують час пропорційний N·log(N). (Можна довести, що ніякий алгоритм сортування не може використовувати менш, ніж N·logN порівнянь між ключами.)

Мірою ефективності алгоритму внутрішнього сортування є кількість необхідних порівнянь значень ключа (C, compare) і число перестановок елементів (M, move). При цьому треба пам'ятати, що пересилання займає більше часу, ніж порівняння.

Ці параметри визначаються у залежності від **N** - кількості елементів масиву (або файлу).

При цьому усі методи сортування ділять на дві групи:

– "прості" методи, які легко програмуються, але вимагають звичайно порядку $\mathbf{n^2}$ операцій порівняння та обміну (це багато);

– та "довершені або модифіковані" методи, які програмуються більш громіздким кодом, але чим більшим є n, тим більшою є перевага таких методів щодо швидкості.

Існує багато алгоритмів сортувань. Хоча деякі методи в середньому можуть бути краще інших, жоден з методів не буде ідеальним для всіх ситуацій, тому кожен програміст повинен мати у своєму розпорядженні кілька різних типів сортування. Який алгоритм обрати залежить від особливостей поставленої задачі та питання наскільки критичним параметром є час роботи програми. Якщо час роботи програми не є критичним, як правило вибирають найбільш простий та зрозумілий алгоритм.

Як було зазначено раніше, у деяких програмах сортування краще використовувати прості алгоритми. Якщо кількість елементів, які потрібно відсортувати не велика (скажімо менше ніж 500 елементів), то можливо, що використання простого алгоритму буде більш ефективним, чим розробка і налагодження складного алгоритму. Елементарні методи завжди придатні для маленьких файлів (наприклад менших чим 50 елементів). Але варто запам'ятати те, що ці алгоритми не варто використовувати для великих, довільно впорядкованих файлів.

Визначають три класи сортування масивів in situ:

– 1. Сортування простим вибором (щоразу знаходимо найменший елемент та його номер і міняємо його місцями з поточним):

- шукаємо найменший елемент і ставимо його на першу позицію, наступний найменший – на другу і так далі;
- проходимо масив стільки разів, скільки елементів-1.

2. Сортування простим обміном (попарно перевіряючи елементи, при потребі міняємо їх місцями):

- проходимо масив стільки разів, скільки у ньому є елементів-1;
- щоразу переглядаємо увесь масив (можна до кінця відсортованої частини);
- при потребі міняємо елементи місцями.

3. Сортування простими включеннями / вставками (береться перший невідсортований елемент і для нього шукається місце):

- масив ділиться на 2 частини: відсортовану та невідсортовану.

Сортування вставкою

- проходимо масив стільки разів, скільки елементів, починаючи з другого;
- записуємо елемент у допоміжну змінну, і проходимо масив у зворотньому напрямку, поки не знайдемо правильне місце для елемента;
- при цьому всі пройдені елементи зміщуються на 1.

Двома найпростішими сортами є сортування вставкою та сортування вибором, обидва з яких ефективні для масивів малих розмірів через низькі витрати, але не ефективні для масивів великих розмірів. Сортування вставкою, як правило, швидше, ніж сортування вибором на практиці, через меншу кількість порівнянь і хорошу продуктивність для майже відсортованих даних, і, таким чином, є кращим на практиці, але сортування вибором використовує менше записів, і тому використовується, коли продуктивність запису є обмежуючим фактором.

Сортування простим вибором

Один з найпростіших методів сортування працює в такий спосіб: знаходимо найменший елемент у масиві й обмінюємо його з елементом, який знаходиться на першому місці, потім повторюємо процес із другої позиції у масиві або файлі і знайдений елемент обмінюємо з другим елементом і так далі поки весь масив / файл не буде відсортований.

Цей метод називається сортування вибором оскільки він працює циклічно вибираючи найменший з елементів, що залишилися, як показано на рисунку 3.1.

Рисунок 3.1. Сортування простим вибором, крок перший

Потім операція повторюється для підмасиву a[2..n], як показано на рисунку 3.2.

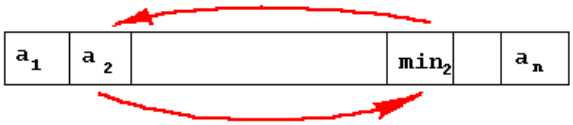


Рисунок 3.2. Сортування простим вибором, крок другий

Тобто на кожному кроці шукається елемент, найменший з тих, що залишилися невпорядкованими. Якщо знайдено min, то він обмінюється значенням з першим елементом.

Сортування простим вибором ефективне для невеликих масивів. Можна відмітити, що через те, що в цьому методі кожен елемент переміщається не більше одного разу, його варто використовувати для впорядкування великих записів з маленькими ключами, як показано на рисунку 3.3. Кількість порівнянь при використанні цього методу буде n*(n-1)/2.

Кількість перестановок у гіршому випадку складає (n^2). Але середня кількість перестановок є (n*ln n), що в ряді випадків робить цей метод кращим.

Масив до впорядкування	22	20	-2	-40	88	-77	-22
1 крок	-77	20	-2	-40	88	22	-22
2 крок	-77	-40	-2	20	88	22	-22
3 крок	-77	-40	-22	20	88	22	-2
4 крок	-77	-40	-22	-2	88	22	20
5 крок	-77	-40	-22	-2	20	22	88
6 крок	-77	-40	-22	-2	20	22	88

Рисунок 3.3. Приклад сортування масиву простим вибором

Сортування простим вибором невигідно використовувати у випадку, коли базовий масив впорядкований у зворотному порядку.

Сортування простими включеннями

Сортування включенням (вставкою) − це метод який майже настільки ж простий, як і сортування вибором. Але цей метод вважається більш гнучким. Цей метод часто використовують при впорядкуванні гральних карт: беремо один елемент і вставляємо його

в потрібне місце серед тих, що ми вже відібрали (тим самим залишаючи їх відсортованими).

Алгоритм сортування:

Перший елемент завжди можна вважати відсортованим масивом довжини 1.

Починаючи з 2-го елемента, у циклі поточний елемент, порівнюється з попередніми та ставиться на потрібне місце, як показано на рисунку 3.4.

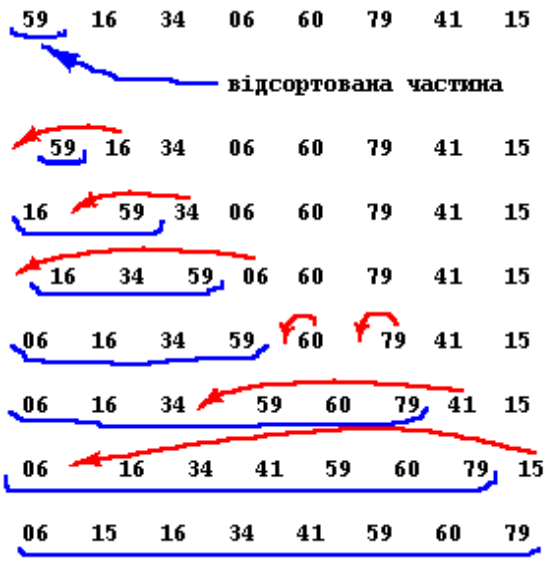


Рисунок 3.4. Схематичний приклад сортування простим включенням

На рисунку відсортована частина виділена синім кольором. Червона стрілочка показує переміщення чергового елемента.

Можливі два виходу з циклу:

– знайдено елемент з ключем, меншим за **x**, тоді **x** ставиться перед ним;

– досягнуто лівий кінець послідовності.

Розглянемо ще один приклад, як показано на рисунку 3.5.

Масив до впорядкування	22	20	-2	-40	88	-77	-22
1 крок	20	22	-2	-40	88	-77	-22
2 крок	-2	20	22	-40	88	-77	-22
3 крок	-40	-2	20	22	88	-77	-22
4 крок	-40	-2	20	22	88	-77	-22
5 крок	-77	-40	-2	20	22	88	-22
6 крок	-77	-40	-22	-2	20	22	88

Рисунок 3.5. Приклад сортування масиву простим включенням

Сортування простими вставками у чомусь схоже на сортування вибором. Аналогічним чином робляться проходи по частині масиву, і аналогічним же чином на його початку "виростає" відсортована послідовність. Але у сортуванні вибором можна було чітко заявити, що на і-му кроці елементи a[1]...a[i] розташовані на остаточних місцях і нікуди більш не перемістяться. Тут же подібне твердження буде більш слабким: послідовність a[1]...a[i] впорядкована, але її елементи не обов'язково знаходяться у своїй останній позиції, оскільки їх ще можуть пересунути праворуч менші елементи, які зустрінуться пізніше під час виконання алгоритму. Масив буде повністю відсортованим, коли ми досягнемо правого краю масиву.

Якщо довжина нашого масиву дорівнює n, то нам потрібно пройтися по n - 1 елементам. Щораз нам може знадобитися зсунути n - 1 інших елементів. От чому цей метод вимагає таки багато часу. Сортування вставками відноситься до числа методів сортування по місцю.

Цей алгоритм описує стійке сортування (порядок елементів з однаковими ключами залишається незмінним). Сортування вставкою рекомендують використовувати для масивів з частково упорядкованими елементами. І не рекомендують використовувати, якщо елементи впорядковані у зворотному порядку.

Кількість перевірок на і-му кроці дорівнює щонайбільше i-1, щонайменше 1, тому в середньому – i/2. В середньому загальна кількість перевірок:

$$C = \sum_{i=2}^{n} \frac{i}{2} = \frac{(2+3+...+n)}{2} = \frac{(2+n)(n-1)}{4} = O(n^2)$$

При цьому:

$$C_{\min} = n-1; \quad C_{\max} = \frac{(n-1)n}{2}.$$

Кількість обмінів M дорівнює щонайбільше i, щонайменше 0 на і-му кроці, тобто i/2 у середньому. Тому:

$$M_{\min} = 0; \quad M_{\max} = \frac{(n-1)n}{2}; \quad M_{cep} = \frac{(n-1)n}{4}.$$

Алгоритм можна модифікувати, якщо перевірки на кожному кроці об’єднати в одну умову. Для цього на початок масиву ставлять елемент-прапорець. Якщо цикл дійшов до прапорця, він зупиняється.

Сортування простим обміном (метод “бульбашки)

За основу взято принцип порівняння та обміну пар сусідніх елементів доти, поки не будуть впорядковані всі елементи – від кінця до початку. Найлегші елементи масиву начебто "випливають" наверх, а "найважчі" - тонуть.

Алгоритмічно це можна реалізувати в такий спосіб. Ми будемо переглядати весь масив "знизу нагору" і міняти місцями елементи. Міняти елементи будемо у тому випадку, якщо "нижній" елемент менше, ніж "верхній". Таким чином, ми виштовхнемо наверх самий "легкий" елемент масиву − звідси аналогія з бульбашкою. Наступний прохід робиться до другого зверху елемента, таким чином другий за величиною елемент піднімається на правильну позицію... Робимо проходи по всій зменшуваній нижній частині масиву (тобто для тих, котрі знаходяться "нижче" максимального) доти, поки в ній не залишиться тільки один елемент. На цьому сортування закінчується, тому що послідовність упорядкована по зростанню, як показано на рисунку 3.7.

Як видно, алгоритм досить простий, але, як іноді зауважують, він є неперевершеним у своїй неефективності.

Масив до впорядкування	22	20	-2	-40	88	-77	-22
1 крок	22	20	-2	-40	88	-77	-22
2 крок	20	-2	-40	22	-77	-22	88
3 крок	-2	-40	20	-77	-22	22	88
4 крок	-40	-2	-77	-22	20	22	88
5 крок	-40	-77	-22	-2	20	22	88
6 крок	-77	-40	-22	-2	20	22	88

Рисунок 3.6. Приклад сортування масиву простим обміном

Цей алгоритм можна оптимізувати: у прикладі останні проходи на сортування не впливають. У такому випадку потрібно запам’ятати, чи здійснювався на даному проході обмін, якщо ні, алгоритм завершити. Можна також запам’ятовувати місце останнього обміну, щоб не рухатися до раніше встановленої межі, якщо в цьому немає необхідності (тобто, кожний наступний прохід починати не з початку (j = 1), а з того місця, де на попередньому проході відбувся перший обмін).

Крім того, слід звернути увагу на таку ситуацію:

Якісно інше поліпшення алгоритму можна отримати з наступного спостереження. Хоча легка бульбашка знизу підніметься вгору за один прохід, важкі бульбашки опускаються із найменшою швидкістю: один крок за ітерацію. Так що масив:

2 3 4 5 6 **1**

буде відсортований за 1 прохід, а сортування послідовності:

6 1 2 3 4 5

вимагає 5 проходів.

Щоб оптимізувати впорядкування даних, можна змінювати напрямок проходів почергово. Одержаний алгоритм називають "шейкер-сортуванням"

У найкращому випадку кількість проходів по масиву буде n-1. У випадку, якщо мінімальний елемент стоїть в кінці масиву, нам знадобиться n-1 операцій перестановки елементів, для того щоб мінімальний елемент потрапив на перше місце в масиві.

Кількість порівнянь при використанні цього методу: $C = \frac{(n-1)n}{2}$

Кількість обмінів: $M_{min} = 0;\ M_{max} = \frac{(n-1)n}{2};\ M_{сер} = \frac{(n-1)n}{4}$

3.2. Модифіковані методи сортування

Хоча існує багато алгоритмів сортування, у практичній реалізації переважають кілька алгоритмів. Так, наприклад, сортування вставкою широко використовується для невеликих наборів даних, а для великих наборів даних використовується асимптотично ефективне сортування. До таких алгоритмів сортування можна віднести сортування злиттям або швидке сортування. Ефективні реалізації зазвичай використовують гібридні алгоритми, які поєднують асимптотично ефективний алгоритм для загального сортування з сортуванням вставкою для невеликих списків у нижній частині рекурсії. Високо налаштовані реалізації використовують більш складні варіанти, такі як Timsort (сортування злиттям, сортування вставкою та додаткова логіка), що використовуються в Android, Java та Python, і introsort (швидке сортування та гіпсове сортування), що використовується у деяких варіантах C++ реалізації та в .NET.

Для більш обмежених даних, таких як числа у фіксованому інтервалі, використовуються сортування розподілу, такі як сортування підрахунком або сортування за коренем. Бульбашкове сортування і його варіанти рідко використовуються на практиці, але зазвичай зустрічаються в навчанні та теоретичних оглядах.

Удосконалення простих методів включеннями і вибором привело до появи поліпшених методів: шейкер-впорядкування, сортування Шелла, пірамідальне сортування. У деяких випадках виявилося, що удосконалення самого гіршого методу – сортування

бульбашкою – дає кращий метод сортування масивів. Винахідник цього методу К. Хоор назвав цей метод швидким сортуванням.

Головним недоліком простих методів є те, що обмін ведеться в основному між сусідніми елементами. Тому бажано, багато модифікованих методів спрямовані на можливість проводити якомога ширші обміни.

Модифікація сортування вибором (Selection sort)

Алгоритм сортування простим вибором – це просто перебір всіх елементів масиву. Сьогодні існує багато модифікацій такого виду сортування. Розглянемо декілька модифікацій:

- Pancake sort (Сортування млинців);
- Cycle sort (Циклічне сортування);
- Bingo sort (Бінго- сортування);
- Double selection sort (Двостороннє сортування вибором).

Швидкість сортування вибором не залежить від того відсортовані дані в початковому масиві чи ні.

Так, наприклад, якщо масив майже відсортований, то сортування вставками впорядкує такий масив набагато швидше ніж сортування вибором або бульбашкою, навіть швидше, ніж швидке сортування. А масив, впорядкований у зворотному порядку, для сортування вставками є виродженим випадком. Тоді масив буде сортуватися максимально довго.

А от для сортування вибором реверсна впорядкованість масиву, часткова чи повна, на швидкість сортування не вплине. Також для класичного випадку сортування вибором неважливо, чи масив містить повторювані елементи чи складається з унікальних значень. На швидкість роботи даного методу це практично не вплине.

Але є алгоритми, які при деяких наборах даних будуть працювати значно швидше або економніше з точки зору економії пам’яті. Наприклад, бінго-сортування створений для випадків, які коли масив складається з повторюваних елементів.

Бінго сортування / Bingo sort

Особливістю цього методу є те, що в невпорядкованій частині масиву не тільки запам’ятовується максимальний елемент, а і визначається максимум для наступної ітерації. Це дозволяє не шукати кожен раз заново повторювані максимуми, а ставити їх відразу на своє місце, як тільки цей максимум в черговий раз зустріли в масиві.

Масив до впорядкування	5	4	4	5	2	3	2	5	1
1 крок	5	4	4	5	2	3	2	1	5
2 крок	2	4	4	1	2	3	5	5	5
3 крок	2	2	3	1	4	4	5	5	5
4 крок	2	2	1	3	4	4	5	5	5
5 крок	1	2	2	3	4	4	5	5	5

Рисунок 3.7. Приклад сортування масиву методом Бінго

Алгоритмічна складність для цього методу не змінюється і залишається, як для випадку сортування вибором. Але якщо масив складається з повторюваних чисел, то бінго-сортування впорядкує масив в десятки разів швидше, ніж звичайне сортування вибором.

Циклічне сортування / Cycle sort

Циклічне сортування цінне з практичної точки зору тим, що елементи переставляються тільки у тому випадку, коли елемент ставиться на своє кінцеве місце. Цей метод можна використовувати, якщо перезапис елементів у масиві вимагає мінімальну кількість змін. Тобто цей метод максимально дбайливо працює з фізичною пам’яттю.

Алгоритм роботи наступний: перебираємо масив і дивимося на яке місце в масиві потрібно вставити черговий елемент з цієї комірки. На місці, куди потрібно вставити знаходиться якийся елемент. Цей елемент потрібно запам’ятати в буфері обміну. Для цього елемента теж шукаємо його місце в масиві. Вставляємо його на потрібне місце, а в буфер відправляємо елемент, який опинився в місці вставки. Для нового числа в буфері проходимо ті ж кроки. Такий пошук триває доти, поки черговий елемент в буфері обміну виявиться тим елементом, який потрібно вставити саме в початкову клітинку (поточне місце в масиві в головному циклі алгоритму). Коли ми знайдемо місце для початкового елемента, тоді у зовнішньому циклі можна перейти до наступної клітинки і повторити для неї ту ж процедуру.

У традиційних модифікаціях сортуваннях вибором ми шукаємо максимум/мінімум, щоб поставити їх на останнє/перше місце. У cycle sort мінімум/максимум на перше місце в підмасиві начебто знаходиться сам, в процесі того, як кілька інших елементів ставляться на свої законні місця десь в середині масиву.

Алгоритмічна складність цього методу залишається $O(n^2)$. На практиці циклічне сортування часто працює в декілька разів повільніше, ніж сортування вибором, оскільки кількість порівнянь та переміщень по масиву збільшується. Це є ціна за мінімально можливу кількість перестановок елементів.

Сортування млинців / Pancake sort

Млинцеве сортування – алгоритм впорядкування елементів масиву завдяки перевороту окремих частини масиву.

До масиву застосовується тільки одна операція – переворот частини масиву.

Завдяки чому максимальний елемент переміщується в кінець масиву. Для цього необхідно виконати наступні кроки:

1) вибираємо підмасив рівний по довжині поточному масиву;
2) шукаємо позицію максимального елемента;
3) якщо максимальний елемент розташований не в кінці підмасиву, то:
    3.1) перевертаємо частину масиву від початку до максимального значення;
    3.2) перевертаємо весь обраний підмасив;
    3.3) зменшуємо робочу область масиву і переходимо до другого кроку.

Наприклад, початкова купка млинців виглядає, як показано на рисунку 3.8 [1].

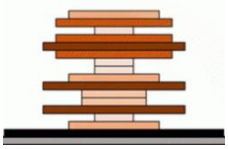


Рисунок 3.8. Масив «млинців» до сортування

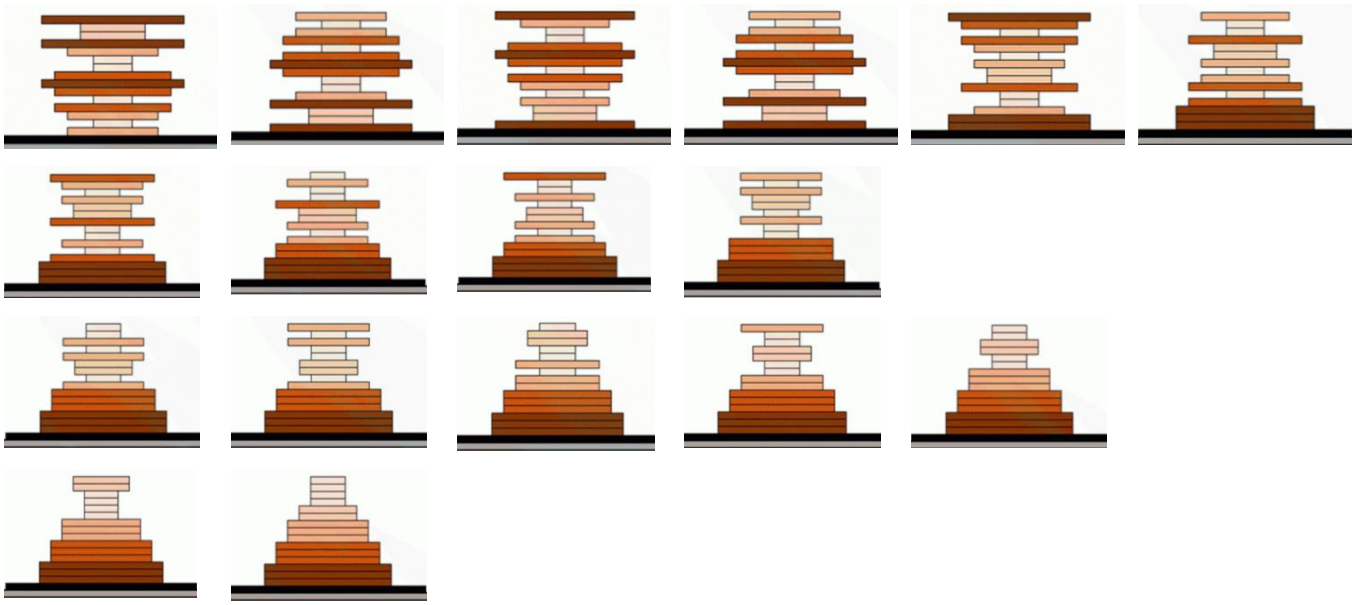


Рисунок 3.9. Приклад сортування масиву «млинців»

Для пришвидшення результату роботи алгоритму потрібно мінімізувати кількість переворотів. У найпростішому варіанті в невідсортованій частині масиву шукаємо максимальний елемент.

Коли максимум знайдений, робимо два перевороти елементів масиву. Спочатку перевертаємо елементи так, щоб максимум опинився на протилежному кінці. Потім перевертаємо весь невідсортований підмасив. Схоже до перевертанні тарілки з млинцями. Після другого розвороту максимум встановлюється на своє місце.

Звичайно, такі перевороти масивів, призводять до алгоритмічної складності O($\mathbf{n^3}$). Адже при розворот частини масиву потребує додаткового циклу. Але, сортування способом перегортання млинців часто згадують, як цікаве сортування з математичної точки зору. Пошук мінімальної кількості необхідних переворотів, достатніх для сортування, є нетривіальною задачею. Існують більш складні постановки завдання, їх називають млинці з підгорілою стороною.

Ефективність сортування вибором залежить від ефективності пошуку мінімального/максимального елемента в невідсортованих частинах масиву. У всіх розглянутих алгоритмах пошук здійснюється подвійним перебором. В цьому випадку алгоритмічна складність буде краще, ніж O(n^2).

Існує окремий підхід до організації процесу пошуку елементів масиву. Цей підхід розглядає набір даних як купу. Пошук проводиться алгоритмами пошуку в купі.

Складність проведення подібними методами сортування можна показати на наступному рисунку 3.10.

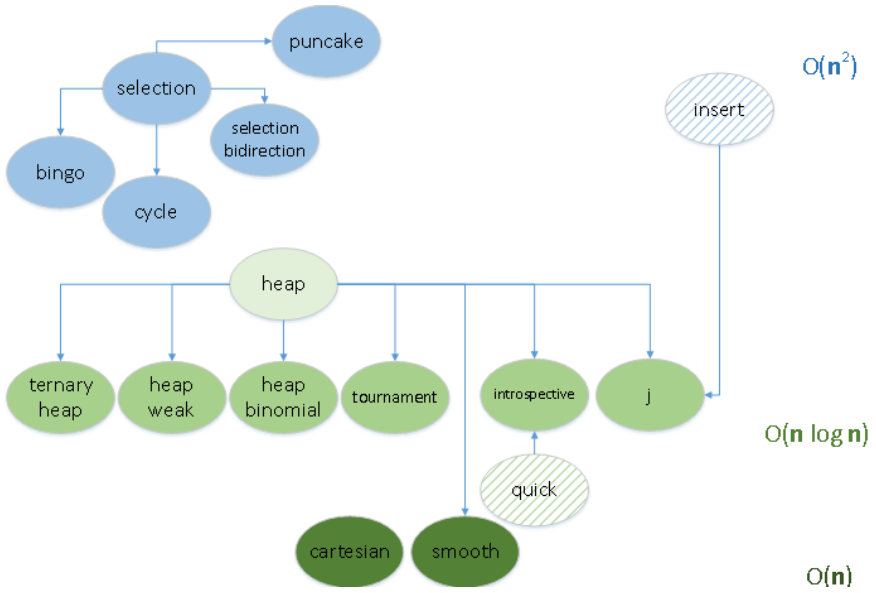


Рисунок 3.10. Групування за складністю методів сортування, модифікація сортування вибором

Модифікації сортування бульбашкою

Сортування бульбашкою досить схоже на сортування вставкою. Але бульбашка має дещо вищі накладні витрати. У випадку, якщо маси майже відсортований, бульбашкове сортування витрачає O(n) часу, але вимагає принаймні 2 проходи по масиву (тоді як сортування вставкою вимагає щось близьке до 1 проходу).

Тому, у класичному випадку складність такого методу сортування вважається $O(n^2)$ і враховує порівняння та заміни. Але адаптивний метод, для майже відсортованого масиву має складність O(n). До модифікацій цього методу сортування належать такі різновиди:

- Stooge sort (придуркувате сортування);
- Slow sort (вяле сортування);
- Stupid sort (тупе сортування);
- Gnome sort (сортування гнома);
- Shaker sort (Шейкер сортування);
- Cocktail sort (коктейльне сортування);
- Odd-even sort (сортування парні-непарні);
- Comb sort (сортування гребінцем);
- Quick sort (швидке сортування);
- K-sort (К-сортування).

Шейкер-сортування

Наведений нижче алгоритм Шейкер-сортування є вдосконаленим варіантом сортування простим обміном і має також назву метод “бульбашки з обмеженнями”.

Суть її полягає в тому, що напрями переглядів чергують: за проходом до кінця множини слідує прохід від кінця до початку вхідної множини. При перегляді в прямому напрямку запис з найбільшим ключем ставиться на своє місце в послідовності, при перегляді у зворотному напрямі – запис з самим меншим, як показано на рисунку 3.11.

Масив до впорядкування	44	55	12	42	94	18	06	67
1 крок (i=2, k=8)	06	44	55	12	42	94	18	67
2 крок (i=3, k=8)	06	44	12	42	55	18	67	94
3 крок (i=3, k=7)	06	12	44	18	42	55	67	94
4 крок (i=4, k=7)	06	12	18	44	42	55	67	94
5 крок (i=4, k=4)	06	12	18	42	44	55	67	94

Рисунок 3.11. Приклад шейкер-сортування

Цей алгоритм досить ефективний для задач відновлення впорядкованості, коли початкова послідовність вже була впорядкована, але піддалася не дуже значним змінам. Впорядкованість в послідовності з одиночною зміною (6 1 2 3 4 5) буде гарантовано відновлена усього за два проходи. Обмін двох елементів – набагато більш коштовна операція, ніж порівняння ключів, а при Шейкер-сортуванні зменшилась кількість порівнянь, але не обмінів.

Вважається, що бульбашкове та шейкер працює повільніше, ніж сортування вставками та вибором. Шейкер-сортування вигідно використовувати, якщо елементи масиву майже впорядковані.

Для шейкер-методу аналіз зробити складно. Але цей метод має сильну перевагу якщо масив "майже відсортовано", тобто, коли кількість елементів m, що не стоять на місці, є незначною у порівнянні з n, n << m. Тоді їх буде розставлено рівно за m проходів.

Метод швидкого сортування QuickSort / Хоара (рекурсивний алгоритм)

Метод відноситься до групи методів обміну, тобто до тієї ж, що і метод "бульбашки". Але корінна відмінність його полягає у організації обмінів між далекими за розташуванням елементами, що призводить до істотного прискорення роботи.

Ідея сортування методом QuickSort наступна: обирається деякий елемент xm, який називається медіаною / відокремлюючим елементом. На першому кроці потрібно всі елементи масиву, які мають значення, менші ніж xm перемістити на початок масиву, а після значення xm перемістити всі елементи, значення яких більші або рівні xm. Наступний крок: рекурсивно проходимо обидві частини, до та після елементі xm. Якщо вважати, що елемент xm розміщений на k-му місці, то ми в кожній частині з частин масиву від A[1] до A[k-1] та від A[k+1] до кінця знову обираємо медіану, і переносимо елементи відносно медіани вліво і вправо. Робота зупиняється, коли у кожному з підмасивів залишиться лише по одному елементу. Це означатиме, що масив відсортовано.

Для організації роботи на першому етапі робиться інтуїтивне припущення, що якщо робити обміни не сусідніх елементів, як у методі "бульбашки", а самих дальніх одне від одного, то це має прискорити роботу програми. Розгляньмо конкретний масив для прикладу, як показано на рисунку 3.12.

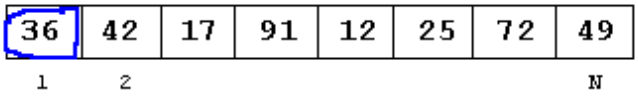


Рисунок 3.12. QuickSort сортування, масив до сортування

Візьмемо за медіану елемент a[1]. Порівняймо його з останнім елементом a[N]. Якщо вони впорядковані один відносно одного, то a[1] порівнюємо з a[N-1], a[N-2], ..., поки не зустрінеться елемент, менший від медіани. Тоді їх треба поміняти місцями, як показано на рисунку 3.13.

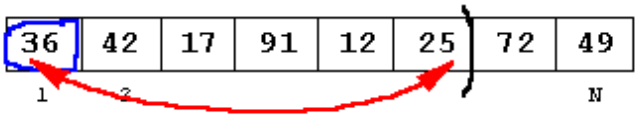


Рисунок 3.13. QuickSort переміщення першого елемента на місце медіани

Далі слід порівнювати медіану з наступним елементом, що стоїть за попереднім місцем розташування медіани (в даному випадку - a[N-2]). Якщо впорядкованість порушується, треба здійснити обмін, як показано на рисунку 3.14.

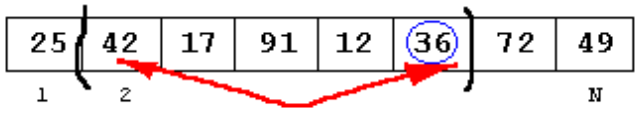


Рисунок 3.14. Пошук місця для елемента-медіани

І взагалі, після кожного обміну слід міняти "кінець" масиву, з яким порівнюється значення медіани. При цьому непроаналізована частина завжди буде залишатися "всередині, "зліва" від неї будуть елементи, менші за медіану, а "справа" - більші, як показано на рисунку 3.15.

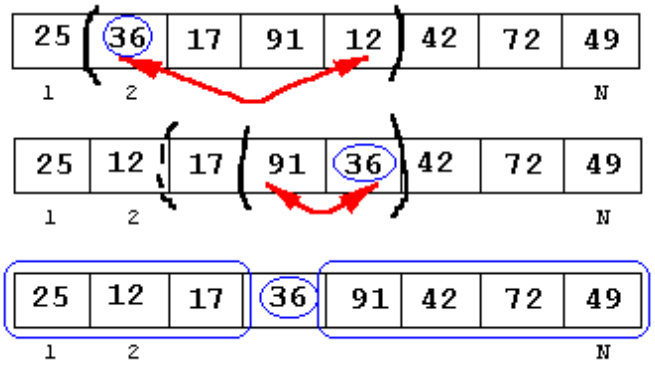


Рисунок 3.15. Встановлення елемента-медіани на своє місце

Після того, як медіану порівняно з усіма елементами у масиві, вона, стрибаючи, стане на своє місце, тобто зліва від медіани стануть всі елементи, менші від неї, а справа - більші. Далі достатньо лише окремо відсортувати праву та ліву частини масиву, бо положення медіани вже не мінятиметься.

Якщо реалізовано фрагмент, що виконує роботу першого етапу щодо всього масиву, треба застосувати його до лівої та правої частин.

Для визначення місця медіани її порівнюють з усіма іншими елементами масива рівно по одному разу, тому на першому етапі порівнянь буде N-1. На другому етапі у кожній з підчастин обирається по медіані, для яких робиться загалом N-3 порівняння. На пересічному i-му $C_{\min} = \sum_{i=0}^{[\log_2 N]+1} (N-i).$ етапі робитиметься (N+1-2i) порівнянь. Якщо медіана вдало поділяє масив на дві рівні за довжиною частини, то етапів буде близько ~ $\log_2 N$, , як показано на рисунку 3.16.

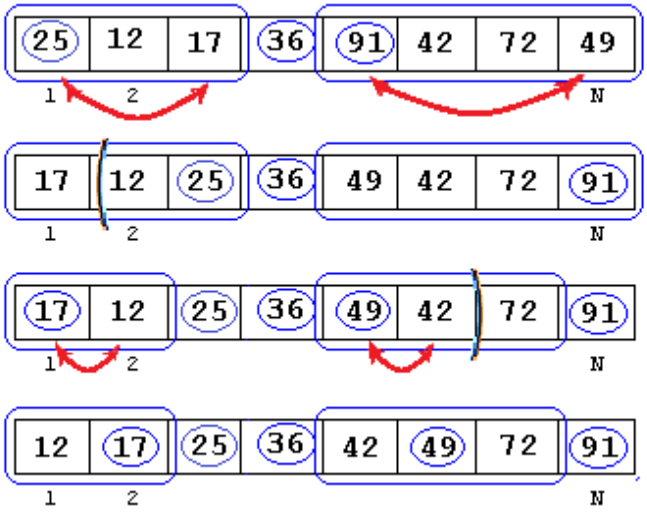


Рисунок 3.16. QuickSort робота з лівою та правою частинами масиву відносно медіани

Перед оцінюванням кількості обмінів зауважимо, по-перше, що метод QuickSort є стійким: якщо за медіану береться перший елемент, то у вже відсортованому масиві ніяких обмінів не відбудеться, але кількість порівнянь буде максимальною:

$$M_{\min} = 0; \quad C_{\max} = (N-1) + (N-2) + \ldots + 2 + 1 = \frac{N(N-1)}{2}.$$

Оцінювання середньостатистичних значень M та C є нелегкою задачею з огляду на необхідність використання апарату теорії ймовірностей, але обидві величини будуть порядку ~ N log 2 N.

- O(n^2) часу, але зазвичай O(n·lg(n)) часу.

Часто в якості опорного елемента пропонується вибрати медіану (середину масиву). Однак можна підібрати приклад, при якому алгоритм з вибором медіани в якості опорного елемента буде видавати неправильну відповідь. Відомі стратегії: вибирати

постійно один і той самий елемент, наприклад, перший або останній; вибрати елемент випадковим чином.

Недолік вибору в якості опорного одного із крайніх елементів масиву — при передачі параметром уже відсортованого масиву такий вибір призводить до найгіршого випадку.

Недолік вибору опорного елемента випадковим чином — залежність швидкості алгоритму від реалізації генератора псевдовипадкових чисел. Якщо генератор працює повільно і видає погані послідовності псевдовипадкових чисел, можлива затримка роботи алгоритму.

На швидкодію методу істотно впливає, чи медіана реально є серединним елементом. Що ближчим до справжньої медіани виявляється взятий у цій якості елемент, то ближче до поданої оцінки буде реальна кількість операцій. Тому, якщо застосування програми буде носити масовий характер. а кількість елементів масиву є значною, варто на початку етапу оцінити можливе значення медіани, наприклад, взявши середнє арифметичне значення кількох "випадкових" елементів масиву. В принципі, для вибору медіани можна застосувати і інші методики.

Розглянемо інший приклад.

Наприклад вибрали медіану xm приблизно посередині. Тоді масив переглядається зліва направо, поки не зустрінеться елемент a[i]>xm, а потім справа наліво, поки не зустрінеться елемент a[j]<xm. Елементи a[i] та a[j] міняються місцями, поки і та j не зустрінуться, як показано на рисунках 3.17, 3.18.

Рисунок 3.17 QuickSort сортування, приклад 2

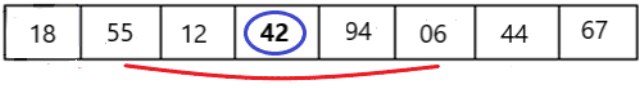


Рисунок 3.18. QuickSort сортування, переміщення елементів відносно медіани

В результаті ми отримали два підмасиви: зліва елементи з меншими ключами, справа — з більшими відносно елемента з ключем 42.

Тепер необхідно зробити те саме з обома отриманими частинами масиву, частинами частин і т.д., поки кожна частина не буде містити в собі тільки один елемент, як показано на рисунку 3.19. Це досить ефективний алгоритм.

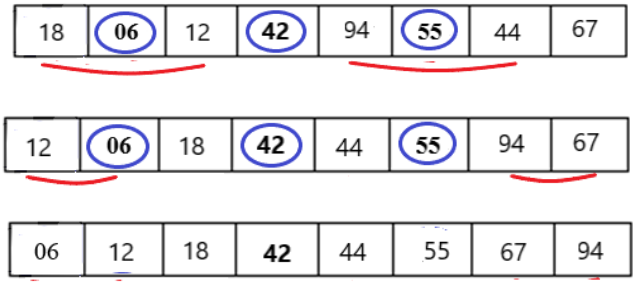


Рисунок 3.19. QuickSort сортування, робота з лівою та правою частинами відносно медіани

Сортування Гнома

Сортування гнома (Gnome sort) – простий в реалізації алгоритм сортування масиву, названий на честь садового гнома, який в такий спосіб сортує садові горщики.

Сортування під час обміну не повертається на початок масиву, а робить один крок назад. Цього достатньо для проведення сортування.

Можна зекономити на русі вперед. Якщо потрібно зробити декілька кроків вперед, відповідно треба буде робити і кроки назад. Якщо запам'ятати місце, з якого почалися перестановки, можна відразу «перестрибнути» на це місце після перестановки елементів.

Сортування гребінцем

Comb sort – алгоритм сортування масиву, є модифікованим варіантом сортування бульбашкою. Особливість сортування гребінцем полягає у врахуванні значень в кінці масиву, які суттєво уповільнюють сортування бульбашкою.

У випадку сортування бульбашкою порівнюються два сусідні елементи масиву даних, відстань між якими дорівнює одиниці. При сортуванні гребінцем обирають більші відстані між порівнюваними елементами. При сортуванні спочатку обирають велику відстань, яку поступово зменшують до одиниці. На першому кроці порівнюється перший і останній елемент масиву. На кожній ітерації розрив між елементами зменшується. Коли відстань між елементами масиву буде дорівнювати одиниці, масив сортується методом бульбашки.

Оптимальне значення фактора зменшення визначається аналогічно правилу золотого перерізу $1/(1-e^{-\varphi}) \approx 1.247$, де e – основа натурального логарифма, а φ – золотий перетин.

Сортування обміном

Іноді відрізняють методи сортування обміном і сортування бульбашкою. Сортування обміном працює шляхом порівняння першого елемента з усіма елементами над ним. У разі необхідності елементи міняють місцями. В результаті перший елемент буде правильнм для остаточного порядку сортуванняю Аналогічні кроки проводять для визначення другого елемента і так далі.

Якщо список уже відсортовано, цей алгоритм він може бути швидшим, ніж бульбашкове сортування.

Алгоритми бульбашкових сортувань дещо подібні до сортування вставкою. У випадку майже відсортованих даних, бульбашкове сортування може мати часову складність O(n). Як і всі прості сортування мають складність $O(n^2)$. Модернізовані методи можуть працювати краще і використовуватись на більших масивах даних, як показано на рисунку 3.20.

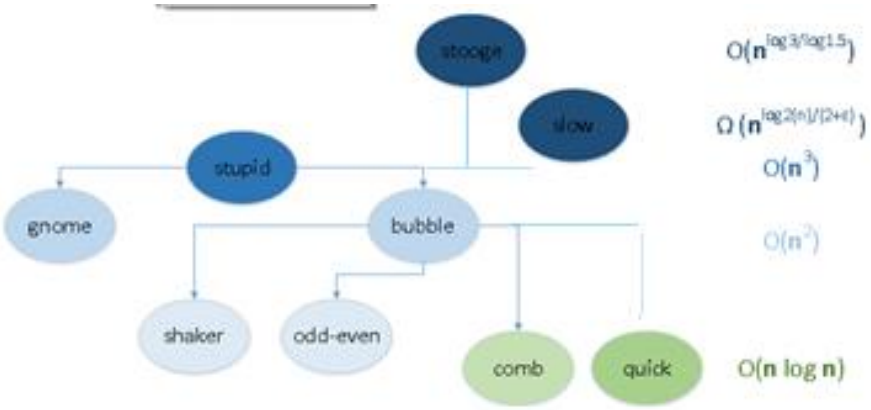


Рисунок 3.20. Групування за складністю методів сортування, модифікація сортування обміном

1.2.3. Модифікації сортування вставками

Сортування вставками є одним з елементарних алгоритмів сортування з $O(n^2)$. Але у випадку майже відсортованого масиву, цей метод є адаптивним O(n). Це також є стабільний метод сортування і використовується як рекурсивний базовий варіант для масивів невеликого розміру. У випадку роботи з великими масивами використовують модифіковані алгоритми:

- сортування простими вставками,
- сортування вставками з бінарним пошуком,
- сортування злиттям,
- парне сортування простими вставками,
- сортування Шелла,

- сортування деревом,
- сортування бібліотекаря,
- пас’янсне сортування,
- сортування «Ханойська вежа»,
- сортування таблицею Юнга,
- сортування вивертанням.

Метод Шелла

Ще одним розвитком методу сортування вставками є сортування методом Шелла, інша назва сортування вставками зі зменшенням відстані.

Він запропонував проводити послідовне впорядкування підмасивів з елементів, які знаходяться на великих відстанях. При цьому на кожному наступному етапі відстані між елементами в групах мають зменшуватися.

Для ілюстрації алгоритму розглянемо його покрокове описання. Не обмежуючи загального, у якості прикладу спочатку зупинимося на масиві з кількістю елементів, що є степенем двійки, тобто :

1. На першому етапі окремо групуються і сортуються елементи, розміщені на відстані N/2 . Це є впорядкування N/2 підмасиві по 2 елементи, яке називатимемо N/2-сортування .

2. На другому етапі виконується впорядкування N/4 підмасивів по 4 елементи на відстані N/4 - N/4-сортування і т.д.

На останньому етапі виконується одинарне сортування (впорядкування на відстані 1). Візьмемо для прикладу масив, як показано на рисунку 3.21.

60	16	41	06	59	79	34	15

Рисунок 3.21. Метод Шелла, масив до сортування

На першому етапі масив уявно ділиться на підмасиви з елементами, що стоять один від одного, наприклад, на 4 елементи, як показано на рисунку 3.22.

60				59			
	16				79		
		41				34	
			06				15

Рисунок 3.22. Метод Шелла, розподіл на підмасиви

Кожен з них впорядковується окремо, , як показано на рисунку 3.23.

59				60			
	16				79		
		34				41	
			06				15

Рисунок 3.23. Метод Шелла, впорядкування підмасивів

Сортування проводиться у кожному підмасиві методом простого включення, , як показано на рисунку 3.24.

59	16	34	06	60	79	41	15

Рисунок 3.24. Масив після впорядкування підмасивів, етап 1

На другому етапі підмасиви утворюються елементами через один, як показано на рисунку 3.25.

59		34		60		41	
	16		06		79		15

Рисунок 3.25. Метод Шелла, формування підмасивів, етап 2

Ці підмасиви також впорядковуємо. Для нашого прикладу, після впорядкування одержуємо результат, , показаний на рисунку 3.26.

34		41		59		60	
	06		15		16		79

Рисунок 3.26. Метод Шелла, впорядкування підмасивів, етап 2

У загальному вигляді отримали масив, як показано на рисунку 3.27.

34	06	41	15	59	16	60	79

Рисунок 3.27. Метод Шелла, масив після етапу 2

Після цього весь масив сортується разом. За рахунок попередніх етапів він виявляється вже близьким до відсортованого, тому обмінів необхідно вже не так багато, як показано на рисунку 3.28.

06	15	16	34	41	59	60	79

Рисунок 3.28. Метод Шелла, відсортований масив

При роботі з цим методом рекомендують розподіл на підмасиви робити кроками, які не є степенями двійки.

Рекомендовано наступні послідовності кроків:

$$h_{k-1} = 3h_k + 1, \quad h_K = 1, \quad K = [\log_3 n] - 1$$
$$h_{k-1} = 2h_k + 1, \quad h_K = 1, \quad K = [\log_2 n] - 1.$$

В цьому випадку кількість операцій пропорційна $n^{1,2}$.

Зрозуміло, що алгоритм сортування Шелла успадковує базові властивості сортування вставкою. Тому цей алгоритм добре адаптується у випадку сортування частково впорядкованих масивів.

Вважається, що часова складність методу сортування Шелла $O(n^{3/2})$, а адаптивна часова складність для майже відсортованого масиву $O(n \cdot \lg(n))$.

Сортування методом злиття (сортування послідовностей)

Алгоритм роботи дещо схожий на алгоритм швидкого сортування. Масив ділиться навпіл, до кожної половини застосовується рекурсивно та сама процедура сортування злиттям, а відсортовані частини з'єднуються в один впорядкований масив.

Принцип роботи алгоритму полягає в об’єднанні двох упорядкованих масивів в один упорядкований масив (часто двох упорядкованих ділянок масиву в одну упорядковану ділянку іншого масиву). Для цього елементи масивів порівнюються і за результатами порівняння в новий масив записується елемент або з першого, або з другого масиву. Один з масивів під час злиття може закінчитися раніше. В такому випадку елементи іншого масиву, які ще не були опрацьовані, потрібно додати до нового масиву.

Час злиття упорядкованих масивів лінійно залежить від їх сумарної довжини. Враховуючи цей факт, неважко буде довести, що повне сортування методом злиття, як і сортування методом Хоара, потребує виконання $O(n \log n)$ базових операцій над елементами масиву розмірності n.

У якості прикладу для цього методу часто наводять формування однієї вишикуваної за зростом колони солдатів з двох вишикуваних за зростом колон. Людина, яка керує побудовою такої колони, порівнює зріст перших солдат у обох колонах і вказує, якому з них треба залишатися у своїй колоні, а кому ставати на нове місце у новій колоні.

Так продовжується, поки одна з колон не вичерпається, а решта іншої колони додається до нової.

Розглянемо алгоритм сортування масиву А злиттям:

Спочатку елементи масиву порівнюємо попарно А[1] і А[2], А[3] і А[4] і т.д. Тобто вважаємо, що ми працюємо з упорядкованими послідовностями довжиною ln=1. Після цього кроку у нас утворилися впорядковані послідовності довжиною ln=2. Якщо кількість елементів масиву непарна, то останній елемент А[n] залишається на місці, як послідовність довжиною 1.

Далі ми працюємо з утвореними упорядкованими послідовностями. Порівнюємо групи елементів А[1]А[2] з А[3]А[4], А[5]А[6] з А[7]А[8] і т.д. В результаті отримаємо впорядковані ділянки масиву, довжиною 4 елементи (ln=4).

Звичайно, наш масив може не бути кратним 4. Тоді у нас залишиться невпорядкований хвостик. Визначимо довжину частинки масиву після останньої четвірки елементів h = n mod 4. Якщо h =1 або h =2 останні, то останні h елементів масиву вже впорядковані після проходження попередніх кроків. В такому випадку їх просто залишають на місці. Якщо h =3, це значить, що в нас є впорядкована послідовність А[n-1]А[n-2] та послідовність А[n]. В цьому випадку їх потрібно об’єднати у впорядковані послідовність довжиною h.

На наступних кроках послідовність щоразу подвоюється через злиття впорядкованих елементів. Робота алгоритму зупиняється, коли довжина упорядкованих ділянок ln >= n.

Розглянемо приклад сортування масиву, довжиною 11 елементів методом злиття.

Нехай початковий масив виглядає так:

<10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0>

На першому кроці порівняємо попарно послідовності, довжиною в 1 елемент:

ln =1: <10><9> ; <8><7> ; <6><5> ; <4><3> ; <2><1> ; <0>

отримали < <9><10> ; <7><8> ; <5><6> ; <3><4> ; <1><2> ; <0>

Далі, порівнюємо впорядковані послідовності між собою:

ln =2: <9, 10> <7, 8> ; <5, 6> <3, 4>; <1, 2> <0>

ln =4: < 7, 8, 9, 10 > < 3, 4, 5, 6 > ; < 0, 1, 2 >

ln =8: < 3, 4, 5, 6, 7, 8, 9, 10 > < 0, 1, 2 >

ln =16: < 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 > .

Отже, після 4 кроків ми отримали впорядкований масив.

В наведеному прикладі розглянуто злиття двох масивів. На практиці, кількість впорядкованих послідовностей може бути більша.

Сортування злиттям можна проводити рекурсивно.

Цей алгоритм є одним із перших ефективних алгоритмів сортування. Його запропонував Дж. Нейман у 1945 році.

Недоліком цього алгоритму є те, що для зручності використовують додатковий масив, куди записують нову впорядковану послідовність. А при великих розмірах масивів, може виникнути ситуація, коли користувачу не вистачить об'єму пам'яті для зберігання двох масивів.

Нижня оцінка часу задачі сортування масиву за числом порівнянь

Відомо, що будь-який алгоритм простого сортування масиву, тобто метод, у якому використовуються тільки порівняння і перестановки, робить мінімум $O(n \log^2 n)$ порівнянь. Отже, складність у гіршому випадку $C(n)$ буде $O(n \log^2 n)$.

Можна припустити, що кожній перестановці передує деяке порівняння. Тоді загальна кількість перестановок, що здійснюється алгоритмом, не перевищує $C(n)$ і, отже, ефективні алгоритми сортування повинні мати складність $O(n\log^2 n)$ як за порівняннями, так і за перестановками. Розглянемо деякі такі алгоритми.

Модифікація методів сортування злиттям

Алгоритмів сортувань злиттям існує багато. Кожен з них має свої рекомендації до застосування.

Сортування злиттям працюють за наступним принципом:

1. Шукають (іноді формують) впорядковані підмасиви.
2. Впорядковані підмасиви об'єднують в один масив.

Сам по собі впорядкований підмасив нам нічим не допоможе. Але, якщо в масиві існує декілька впорядкованих підмасивів, ми можемо з мінімальними затратами провести злиття таких масивів в один впорядкований масив.

Об'єднати два впорядкованих масиви в один – дуже проста операція. Рухаємось по масивах і порівнюємо їх елементи. Менший записується в загальний масив. Автором даної концепції є Джон фон Нейман. Один з методів сортування злиттям має назву неймановсько сортування.

Серед найпростіших методів сортувань злиттям відомі такі методи:

- природнє нейманівське злиття;
- пряме злиття Боуза-Нельсона;
- збалансоване злиття зверху-вниз та знизу-вверх;
- багатофазне сортування злиттям;

- каскадне сортування злиттям;
- осциловане сортування;
- нитковидна та незмішане сортування.

Для перевірки впорядкованості елементів масиву можна використовувати декілька методів: перевірка впорядкованості за зростанням, перевірка впорядкованості за спаданням, перевірка з поверненням напряму сортування.

Сортування розподілом

Також відомі сортування розподілом. Ми розподіляємо числа за окремими характеристиками і в результаті отримуємо відсортований масив.

- bucket sort (сотування відрами);
- thanos sort (сортування таноса);
- counting sort (сортування підрахунком);
- pigeonhole sort (голупине сортування);
- підрозрядні сортування (по молодшим LSD radix sort та по старишим розрядам MSD radix sort) ;
- підрахунково-розрядні сортування;
- bead sort (бісерне сортування);
- підрахункові сортування з приблизним розподілом ;
- сортування «американський прапор».

В сортуваннях розподілом елементи розподіляються і перерозподіляються по класах до тих пір, доки масив не впорядкується.

У всіх випадка алгоритм сортування розкидає елементи масиву по класах за якоюсь певною ознакою. Якщо після такого розподілу масив не впорядкувався, проводять уточнення ознак і елементи розподіляють по уточненим класам. Ці кроки повторюють до тих пір, поки масив не відсортується.

В такому підході майже завжди не використовують порівняння елементів між собою. В основному такі методи мають лінійну часову складність, але вимагають додаткову пам'ять. Тому, що згруповані елементи потрібно десь зберігати.

Використовують такі методи сортування для роботи з цілими числами та рядками. Такі алгоритми сортування зручно застосовувати для масивів, які мають однакові дані.

Сортування піраміди (дерево)

Для роботи з деревовидними структурами, як правило, використовують відповідні типи даних. Спробуємо описати таку структуру у вигляді масиву. Для цього розмістимо елементи масиву так, щоб у кожному наступному рядку кількість елементів збільшилась у 2 рази. Тоді у нас вийде деякий аналог піраміди, де у першому рядку буде розміщено перший елемент, у другому елементи з індексами 2, 3, у третьому, елементи з індексами 4, 5, 6, 7, у наступному 8, 9, 10, 11, 12, 13, 14, 15, як показано на рисунку 3.29.

1

2 3

4 5 6 7

8 9 10 11 12 13 14 15

Рисунок 3.29. Приклад побудови піраміди

Останній рядок може бути неповним. Наприклад, ми можемо задати піраміду з 12 елементів.

Якщо вузол n парний, тоді від вузла (n div 2) проводиться стрілка лише до вузла n. Вузли 2k та 2k+1 називаються дочірніми вузлами до k, а сам вузол називається батьківським вузлом. Корінь дерева – вершина піраміди. Кожен вузол є вершиною піраміди відносно її дочірніх вершин, як показано на рисунку 3.30.

1

2 3

4 5 6 7

8 9 10 11 12

Рисунок 3.30. Піраміда до сортування

Якщо між елементами піраміди провести стрілки, отримаємо дерево. Гілки дерева, зобразимо як стрілки від батьківської вершини до дочірньої. Елементи такої піраміди називаються називати вузлами. Вершина піраміди називається коренем дерева.

Проведемо стрілки так, щоб від кожного k-го елемента ішло дві стрілки до елементів 2k та 2k+1, де k<n div 2, як показано на рисунку 3.31.

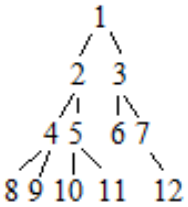


Рисунок 3.31. Піраміда-дерево

Побудуємо піраміду так, щоб значення кожного елемента-батька були не менші ніж значення у його дочірніх вузлах. Тобто,

A[1] >= A[2] та A[1] >= A[3], A[2] >= A[4] та A[2] >= A[5] і т.д.

Тоді, для кожного k=1, 2, … , n div 2, отримаємо:

A[k] >= A[2*k] та A[k] >=A[2*k+1].

У випадку, якщо елемент n парний A[n div 2], він має тільки одну дочірню вершину A[n] , як показано на рисунку 3.32.

44

12 44

12 7 33 2

11 10 3 2 28 27

Рисунок 3.32. Піраміда до сортування

Цій піраміді відповідає послідовне розташування <44, 12, 44, 12, 7, 33, 2, 11, 10, 3, 2, 28, 27>.

З розташування елементів видно, що кожна вершина має найбільше значення, відносно своєї піраміди. Так, A[1] має найбільше значення відносно всієї піраміди. А елемент A[2] має найбільше значення серед елементів A[2], A[4], A[5], A[8], A[9], A[10], A[11] , як показано на рисунку 3.33.

1	44
2 3	12 44
4 5 6 7	12 7 33 2
8 9 10 11 12	11 10 3 2 28 27

Рисунок 3.33. Розташування елементів до сортування

Розглянемо сортування піраміди.

Для початку порівняємо і поміняємо місцями перший і останній елементи. Це елементи $A[1]$ і $A[n]$. Тепер елемент $A[n]$ має найбільше значення і знаходиться на своєму місці, як показано на рисунку 3.34.

1	**27**
2 3	12 44
4 5 6 7	12 7 33 2
8 9 10 11 12	11 10 3 2 28 **44**

Рисунок 3.34. Піраміда, обмін елементів

Розглянемо впорядкування елементів A[1], A[2], ... , A[n-1].

Після перестановки елементів, початкова умова A[1]>=A[2], A[1]>=A[3] порушена. Для, того, щоб відновити баланс (коректність умови) поміняємо місцями значення A[1] з якимось із дочірніх елементів A[2] або A[3], для цього потрібно визначити, чиє значення максимальне. В нашому випадку це буде A[3]: 27<44, тобто A[1]<A[3], , як показано на рисунку 3.35.

1	**44**
2 3	12 **27**
4 5 6 7	12 7 33 2
8 9 10 11 12	11 10 3 2 28 44

Рисунок 3.35. Перебалансування елементів піраміди

Але, після цього обміну елементів, може порушитися умова побудови піраміди для вершини A[3]. Відновимо цю умову так само, як і для вершини A[1]. При порівнянні вузлів A[3] з вузлами A[6] та A[7], визначимо і «перевернемо» частинку піраміди так, щоб виконувалася наша початкова умова. Перевіримо і, за необхідності, поміняємо інші вузли піраміди. Після відновлення коректного розміщення елементів в піраміді, максимальний елемент буде стояти на своєму місці.

1	44
2 3	12 **33**
4 5 6 7	12 7 **27** 2
8 9 10 11 12	11 10 3 2 28 44

Рисунок 3.36. Збалансована піраміда

Тепер нам потрібно повторити цей алгоритм роботи і поміняти значення першого елемента з передостаннім. Після чого потрібно буде знову збалансувати нашу піраміду і

відновити умову, за якою піраміда побудована. Тільки цього разу, балансування для частини масиву A[1], … , A[n-1] , як показано на рисунку 3.36.

Опишемо загальний алгоритм балансування піраміди. Для цього позначимо значення A[1], як A[k]. Перевірка значень повинна проводитись у частині масиву A[k], … , A[n-1]. Алгоритм балансування вузлів працює так само, як і при роботі з усім масивом A[1], … , A[n-1].

Опишемо загальний випадок відновлення балансу у частині масиву A[i], … , A[j].

Будемо вважати, що у частині масиву A[i], … , A[j], де 2*i+1>= j, треба відновити описану раніше властивість (A[k] >= A[2*k] та A[k] >=A[2*k+1]). Тобто після певних перестановок ця умова порушена та A[i]<max{A[2*i], A[2*i+1]}.

Якщо A[2*i]>A[2*i+1], k будемо визначати за формулою: k=2*i. Інакше : k=2*i+1. Поміняємо місцями значення A[i] та A[k]. Після обміну нам потрібно провести перевірку та обмін елементів у частині масиву A[k], … , A[j].

У випадку, коли 2*i=j, перевірка та обмін елементів проводяться тільки для значень A[i] та A[j], причому k=j.

Якщо ж 2*i>j, наша умова не може бути порушеною в частині масиву A[i], … , A[j].

В результаті елемент A[i] буде мати максимальне значення серед A[i], A[2*i], A[2*i+1]. Це може бути використано при початкового збалансованого масиву. Тоді спочатку балансується частина масиву A[ndiv2], … , A[n], а потім A[ndiv2-1], … , A[n].

Сортування вибором при допомозі дерева

Цей алгоритм аналогічний попередньому. Подібні алгоритми використовуються на структурах типу список, дерево. Але ми розглянемо випадок, коли дерево зберігається у вигляді масиву.

Алгоритм сортування деревом TreeSort власне кажучи є поліпшенням алгоритму сортування вибором. Процедура вибору найменшого елемента удосконалена як процедура побудови так званого сортуючого дерева. Дерево сортування – це структура даних, у якій представлений процес пошуку найменшого елемента методом попарного порівняння елементів, що стоять поруч. Алгоритм сортує масив у два етапи.

I етап : побудова сортуючого дерева;

II етап : просівання елементів по сортуючому дереву.

Розглянемо приклад:

Нехай масив A складається з 8 елементів. Аналогічно, описаному вище принципу, другий рядок складається з мінімумів елементів першого рядка, які стоять поруч. Кожний

наступний рядок складений з мінімумів елементів попереднього рядка, що стоять поруч, як показано на рисунку 3.37.

A[2] = min(A[1], A[2])

A[3] = min(A[3], A[4])

A[5] = min(A[5], A[6])

A[8] = min(A[7], A[8])

A[3] = min(A[2], A[3])

A[5] = min(A[5], A[8])

A[5] = min(A[3], A[5])

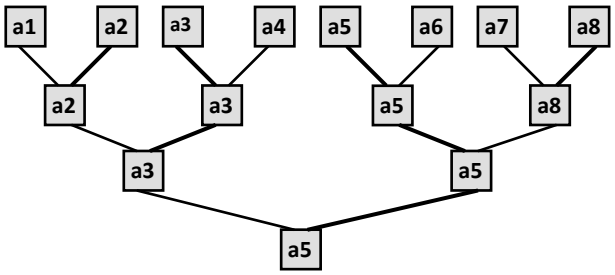


Рисунок 3.37 Приклад дерева

Ця структура даних називається сортуючим деревом. У корені сортуючого дерева розташований найменший елемент. Крім того, у дереві побудовані шляхи елементів масиву від листів до відповідного величині елемента вузла – розгалуження (шлях мінімального елемента a5 – від листа a5 до кореня відмічений товстою лінією), як показано на рисунку 3.38.

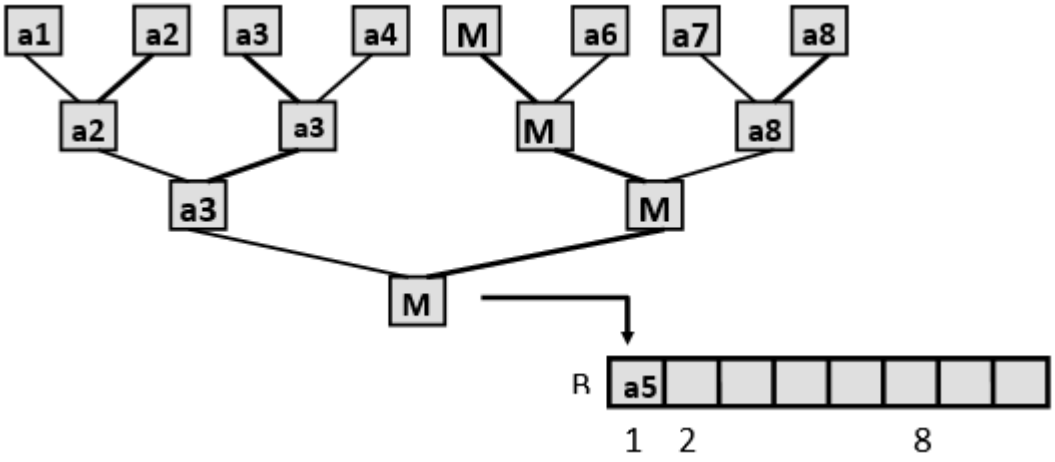


Рисунок 3.38. Просівання елементів дерева

Потім здійснюється просівання елемента уздовж шляху, відзначеного символом M, починаючи з листка, сусіднього з M і до кореня. Крок просівання - це вибір найменшого з двох елементів, що зустрілися на шляху до кореня дерева і його пересилання у вузол, відзначений M.

A[6] = min(M , A[6])

A[6] = min(A[6], A[8])

A[3] = min(A[3], A[6])

B[2] := A[3]

Просівання 2-го елемента показано на рисунку 3.38. Символ M більше, ніж будь-який елемент масиву.

Просівання елементів відбувається доти, поки весь вихідний масив не буде заповнений символами M, тобто n раз:

For i := 1 to n do begin

Відзначити шлях від кореня до листка символом M;

Просіяти елемент уздовж відзначеного шляху;

B[I] := корінь дерева

end;

Обґрунтування правильності алгоритму очевидно, оскільки кожне чергове просівання викидає в масив У найменший з елементів масиву A, що залишилися.

Сортуюче дерево можна реалізувати, використовуючи або двовимірний масив, або одномірний масиві ST[1..N], де N = 2n-1.

У загальному випадку, коли n не є ступенем 2, сортуюче дерево будується трохи інакше. "Зайвий" елемент (елемент, для якого немає пари) переноситься на наступний рівень. Легко бачити, що при цьому глибина сортуючого дерева дорівнює $(\log^2 n) + 1$. Удосконалення алгоритму II етапу очевидно. Оцінки при цьому змінюють лише мультиплікативні множники. Алгоритм TreeSort має істотний недолік: для нього потрібно додаткова пам'ять розміру 2n - 1.

Порівняння методів сортування

Розглянемо порівняння методів сортування, наведеного в [1]. Таблиця алгоритмів сортувань, які використовують порівняння елементів масиву.

Такі алгоритми не мають складність кращу ніж O(n log n) у середньому випадку. Деякі алгоритми є повільними в порівнянні з тими, які обговорювались вище, наприклад bogosort з необмеженим часом виконання і сортування stooge, яке має $O(n^{2.7})$ час виконання.

Ці види зазвичай використовуються для навчальних цілей, щоб продемонструвати, як оцінюється час виконання алгоритмів.

Таблиця 3.1. Порівняння методів сортування

Name	Best	Average	Worst	Memory	Stable	Method
Quicksort	n log n	n log n	n^2	log n	No	Partitioning
Merge sort	n log n	n log n	n log n	*n*	Yes	Merging

Продовження Таблиці 3.1.

In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging
Introsort	n log n	n log n	n log n	log n	No	Partitioning & Selection
Heapsort	n log n	n log n	n log n	1	No	Selection
Insertion sort	*n*	n^2	n^2	1	Yes	Insertion
Block sort	*n*	n log n	n log n	1	Yes	Insertion & Merging
Timsort	*n*	n log n	n log n	*n*	Yes	Insertion & Merging
Selection sort	n^2	n^2	n^2	1	No	Selection
Cubesort	*n*	n log n	n log n	*n*	Yes	Insertion
Shellsort	n log n	$n^{4/3}$	$n^{3/2}$	1	No	Insertion
Bubble sort	*n*	n^2	n^2	1	Yes	Exchanging
Exchange sort	n^2	n^2	n^2	1	Yes	Exchanging
Tree sort	n log n	n log n	n log n	*n*	Yes	Insertion
Cycle sort	n^2	n^2	n^2	1	No	Selection
Library sort	n log n	n log n		*n*	No	Insertion
Patience sorting	*n*	n log n	n log n	*n*	No	Insertion & Selection
Smoothsort	*n*	n log n	n log n^2 n }	1	No	Selection
Strand sort	*n*	n^2	n^2	*n*	Yes	Selection
Tournament sort	n log n	n log n	n log n	*n*	No	Selection
Cocktail shaker sort	*n*	n^2	n^2	1	Yes	Exchanging
Comb sort	n log n	n^2	n^2	1	No	Exchanging
Gnome sort	*n*	n^2	n^2	1	Yes	Exchanging
Odd–even sort	*n*	n^2	n^2	1	Yes	Exchanging

У таблиці 3.2 описано цілочисельні алгоритми сортування та інші алгоритми сортування, які не використовують порвіняння елементів масиві між собою. Такі алгоритми не обмежуються $\Omega (n \log n)$.

Таблиця 3.2. Порівняння методів сортування, які не використовують порівняння елементів масиві між собою

Name	Best	Average	Worst	Memory	Stable	$n \ll 2^k$
Pigeonhole sort	—	$n + 2^k$	$n + 2^k$	2^k	Yes	Yes
Bucket sort (uniform keys)	—	$n + k$	$n^2 \cdot k$	$n \cdot k$	Yes	No
Bucket sort (integer keys)	—	$n + r$	$n + r$	$n + r$	Yes	Yes
Counting sort	—	$n + r$	$n + r$	$n + r$	Yes	Yes
LSD Radix Sort	n	$n \cdot k/ d$	$n \cdot k/ d$	$n + 2^d$	Yes	No
MSD Radix Sort	—	$n \cdot k/ d$	$n \cdot k/ d$	$n + 2^d$	Yes	No
MSD Radix Sort (in-place)	—	$n \cdot k$	$n \cdot k$	2	No	No
Spreadsort	n	$n \cdot k /d$	$n \cdot (k/ s + d)$	$k /d \cdot 2^d$	No	No
Burstsort	—	$n \cdot k/ d$	$n \cdot k /d$	$n \cdot k /d$	No	No
Flashsort	n	$n + r$	n^2	n	No	No
Postman sort	—	$n \cdot k /d$	$n \cdot k/ d$	$n + 2^d$	—	No

Таблиця 3.3. Порівняння методів сортування з необмеженим часом виконання

Name	Best	Average	Worst	Memory	Stable	Comparison
Bead sort	n	S	S	n^2	N/A	No
Simple pancake sort	—	n^2	n^2	$\log n$	No	Yes
Spaghetti (Poll) sort	n	n	n	n^2	Yes	Polling
Slowsort	$n^{\log n}$	$n^{\log n}$	$n^{\log n}$	n	No	Yes

Продовження Таблиці 3.3

Sorting network	Varies	Varies	Varies	Varies	Varies (stable sorting networks require more comparisons)	Yes
Bitonic sorter	$\log^2 n$ parallel	$\log^2 n$ parallel	$\log^2 n$ non-parallel	1	No	Yes
Bogosort	*n*	$(n \times n!)$	unbounded (certain), $(n \times n!)$ (expected)	1	No	Yes
Stooge sort	$n^{\log 3 / \log 1.5}$	$n^{\log 3 / \log 1.5}$	$n^{\log 3 / \log 1.5}$	*n*	No	Yes

3.3. Гібридні методи сортування

Часто можемо помітити деяку схожість між алгоритмами сортування. Іноді важко зрозуміти який базовий метод впорядкування елементів масиву використовує той чи інший модифікований алгоритм. Якщо ж в алгоритмі сортування комбінуються різні методи, його можна віднести до гібридних методів сортування. Як приклад, можна навести такі гібридні методи, показані в таблиці 3.4.

Таблиця 3.4. Порівняння гібридних методів сортування

	Sort	Merge	Insertion	Quick	Bucket	Counting	Radix	Heap	Tree
Insertion+Mergee	Merge-Insertion sort	+	+						
	Timsort	+	+						
	Wikisort	+	+		+				
	Hash table sort	+	+			+			
Quicksort	Introspective sort		+	+				+	
	Multikey sort			+			+		
	Spread sort	+		+	+		+		
Anothe	Tree-counting sort					+			+
	Jsort		+					+	

Пов'язані алгоритми

Пов’язані алгоритми включають часткове сортування, наприклад сортування лише k найменших елементів списку або виділення найменшого елемента. Їх можна неефективно вирішити шляхом повного сортування, але існують більш ефективні алгоритми, часто отримані шляхом узагальнення алгоритму сортування. Найвідомішим прикладом [https://en.wikipedia.org] є quickselect, який пов'язаний з швидким сортуванням. Деякі алгоритми сортування можна отримати шляхом повторного застосування алгоритму відбору quicksort і quickselect. Їх можна розглядати як один і той самий поворотний рух, який відрізняється лише тим, чи повторюється рухз обох сторін (швидке сортування, розділяй і володарюй) або алгоритм рухаєтья в одну сторону (швидкий вибір).

Своєрідною протилежністю алгоритму сортування є алгоритм перемішування. Він принципово відрізняються, оскільки вимагає джерела випадкових чисел. Перетасування також може бути реалізовано за допомогою алгоритму сортування, а саме шляхом випадкового сортування: присвоєння випадкового числа кожному елементу списку, а потім сортування на основі випадкових чисел. Але це зазвичай на практиці не використовується. Існує добре відомий простий і ефективний алгоритм перетасування: перемішування Фішера–Йейтса

Сортування послідовностей

Для освоєння більш довершених методів слід спочатку познайомитися зі структурами даних (дерева, списки, стеки тощо), що розглянуті пізніше.

Метод простого злиття (двофазний метод)

Для методу необхідні принаймні три пристрої для запису послідовностей. На першому етапі (проході), першій фазі, послідовність ділиться на дві підпослідовності рівної довжини. Далі ці підпослідовності позиціонуються у свої початки.

Візьмемо для прикладу масив, показаний на рисунку 3.39.

11	15	09	01	14	04	10	05	06	00	12	02	03	07	13	08

Рисунок 3.39. Просте злиття, масив до сортування

11	09	14	10	06	12	03	13
15	01	04	05	00	02	07	08

Рисунок 3.40. Формування двох послідовностей

Загальна кількість записів може бути невідомою, тому записи у дві підпослідовності направляють через один, як показано на рисунку 3.40.

Друга фаза першого проходу - злиття цих двох підпослідовностей з утворенням пар (по одному елементу з кожної підпослідовності), що є впорядкованими всередині себе. Тобто, на другій фазі з кожної підпослідовності читається по одному елементу, а записуються з них спочатку менший, а потім - більший.

Внаслідок цього маємо послідовність пар, кожна з яких є послідовністю довжиною 2, що вже є відсортованою, як показано на рисунку 3.41.

11	15	01	09	04	14	05	10	00	06	02	12	03	07	08	13

Рисунок 3.41 Формування послідовностей з пар

На другому проході знову ділимо послідовність на дві з рівною кількістю пар (перша фаза другого проходу), як показано на рисунку 3.42.

11	15	04	14	00	06	03	07
01	09	05	10	02	12	08	13

Рисунок 3.42. Формування пар двох послідовностей

Друга фаза другого проходу - злиття пар по одній з кожного пристрою у четвірки, всередині яких запроваджено впорядкованість, як показано на рисунку 3.43.

01	09	11	15	04	05	10	14	00	02	06	12	03	07	08	13

Рисунок 3.43. Злиття пар

У загальному випадку на k-му проході послідовність розкладається на групи по 2k елементів, причому всередині кожна група вже є впорядкованою підпослідовністю.

01	09	11	15	00	02	06	12
04	05	10	14	03	07	08	13

Рисунок 3.44. Формування послідовностей з четвірок

Процес продовжується до тих пір, поки довжина 2k не стане довше за n - довжину всієї послідовності. У даному разі послідовність буде розбито на дві з четвірками, як показано на рисунку 3.44. Четвірки склеються у вісімки, як показано на рисунку 3.45.

01	04	05	09	10	11	14	15	00	02	03	06	07	08	12	13

Рисунок 3.45. Злиття четвірок

Після цього залишиться останній етап, хід якого є очевидним.

Метод простого злиття (однофазний метод)

Якщо у наявності є четвертий пристрій для запису послідовності, то можна поєднати фази кожного проходу. Тоді на кожному етапі дві послідовності, на які спочатку ділимо несортовану послідовність, вважаємо вхідними, а дві інші, у які підчас цього проходу записується інформація, вважаються вихідними.

На k-му проході маємо 2 послідовності, у яких записи утворюють групи по 2k штук і є відсортованими в середині кожної групи. Протягом проходу з кожних двох груп довжиною 2k утворюється злиттям одна група довжиною 2k+1 і нові групи записуються по черзі на вихідні два носії.

Вихідні та вхідні носії послідовно міняються місцями до тих пір, поки весь файл не утворюватиме єдиної впорядкованої послідовності.

Сортування масиву рядків

Сортування рядків – це впорядкування даних рядкового типу за алфавітом (фактично за номером символів, що складають рядок, у кодовій таблиці). Сортування рядків виконується при виведенні списків (за прізвищами, за назвами), імен файлів при роботі з директорією і т.ін.

Порівняння рядків проводять зліва направо, доки не знайдуть не співпадіння символів. Рядок, у якому перший неоднаковий символ має менший номер в таблиці ASCII кодів, ставиться на пере місце:

«Шевченко»< Шевчук»

З двох подібних за символами рядків різної довжини меншим є коротший: ’ABR’ < ’ABRAKAD’.

Сортування здійснюється відповідно до значення якої-небудь ознаки (наприклад, за табельним номером).

4. Алгоритми пошуку

Алгоритми пошуку – це клас алгоритмів задача яких визначити для шуканого елемента номер в масиві. Для спрощення роботи алгоритму вважається, що масив попередньо відсортований за зростанням. Для описаних нижче алгоритмів пошуку є характерними два патерни, шаблони, підходи до роботи. Перший – описаний в алгоритмі лінійного пошуку і носить назву метод грубої сили. Метод грубої сили перебирає всі варіанти, шукаючи необхідний.

Другий шаблон – описаний в алгоритмі бінарного пошуку та ін. Коли проблема пошуку розбивається на під проблеми і шаблон носить назву divide and conquer.

Для ознайомлення з алгоритмами пошуку розглянемо наступні.

4.1. Рекурсивний алгоритм бінарного пошуку (Binary search)

Псевдокод рекурсивного алгоритму бінарного пошуку наведено нижче.

```
int bin_search (int arr[], int x, int start, int end)

{

int mid=(start+end)/2;

if (arr[mid] = = x

{return mid}

if (start = = end)

{return -1 }

if (arr[mid] > x)

{return bin_search (arr, x, start, mid-1)}

if (arr[mid]<x)

{return bin_search (arr, x, mid+1, end)}

}
```

Для порівняння з ітераційним алгоритмом обрахуємо часову складність (Time complexity) даного алгоритму. Тут N – кількість елементів у масиві, C – час виконання команд.

Маємо:

Для першої рекурсії $T(N)=C+T(N/2)$ (4.1)

Для другої рекурсії $T(N/2)=C+T(N/4)$ (4.2)

Для третьої рекурсії $T(N/4)=C+T(N/8)$. (4.3)

Якщо підставити (4.2) в (4.1), а (4.3) в (4.2) маємо:

$T(N)=3C+T(N/8)$

Тоді, скориставшись математичною індукцією, для k-ї рекурсії:

$T(N)=kC+T(N/2^k)$ (4.4)

Припустимо, на k рекурсії отримаємо лише один елемент і його часова складність T(1). Або:

$T(N/2^k)=T(1) = C$ (4.5)

В цілому, із (4.5):

$T(N/2^k)=T(1) => N/2^k=1 => N=2^k => k = \log_2 N$

Останній вираз підставимо в (4.4):

$T(N)=\log_2 N\ C+T(1)=C\ \log_2 N+C$

Якщо перейти до нотації Ландау, відкинувши константи, отримаємо O(logN). Основа алгоритму в нотації Ландау також нівелюється на основі операцій з логарифмами.

4.2. Ітераційний алгоритм бінарного пошуку (Binary search)

Псевдокод ітераційного алгоритму бінарного пошуку елемента в упорядкованому за зростанням масиві наведено нижче.

```
int Bit_Search_ItMeth (int A[], N, key)
{ l=1, r=N
while (l<=r)
   {int mid=(l+r)/2;
   If (key==A[mid])
      {return mid}
   If(key<A[mid])
        {r=mid-1}
   else
```

```
            {l=mid+1;}
        }
        return 0;
    }
```

Наприклад, дано масив А із 13 елементів. Задача алгоритму визначити який номер (key) у числа «12», якщо воно є (див рисунок 4.1).

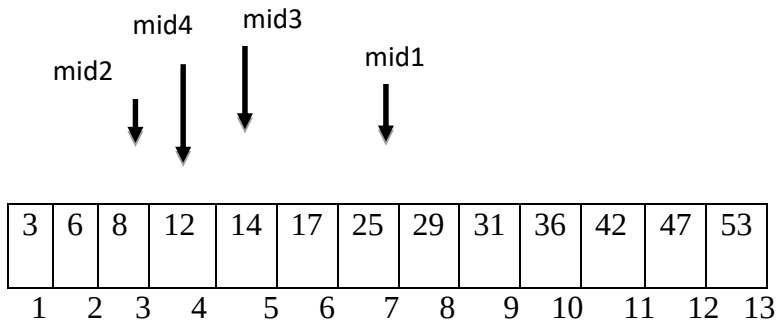


Рисунок 4.1. Принцип роботи алгоритму

Покрокову роботу алгоритму наведено в таблиці 4.1.

Таблиця 4.1. Покрокова робота алгоритму

L	r	Int mid=(l+r)/2	A[mid]
1	13	mid1 = 7	A[7]=25
1	6	mid2 = 3	A[3]=8
4	6	mid3 = 5	A[5]=14
4	4	mid4 = 4	A[4]=12

Оцінимо часову складність даного алгоритму. В найкращому випадку розміщення даних шуканий елемент буде знайдено з першого разу. В цьому разі час пошуку не залежить від кількості елементів у масиві і є сталим. Тобто, Best case O(1).

В іншому випадку, як і для рекурсивного алгоритму бінарного пошуку маємо:

T(N)=C+T(N/2), T(N/2)=C+T(N/4), T(N/4)=C+T(N/8) і в цілому T(N)=kC+T(N/2k). Тут $k=\log_2 N$.

Тобто, в найгіршому випадку (Worst case) нотація Ландау складе O(log N).

Для середнього випадку отримаємо $k=\log_2 (N/2)$ і оскільки константи не входять в нотацію Ландау - Average case O(log N).

4.3. Алгоритм тернарного пошуку (Ternary Search)

Для пошуку порядкового номеру елемента в масиві розглядається впорядкований за зростанням масив. Задача – знайти порядковий номер елемента в масиві, якщо цей елемент є. Ідея базується на алгоритмі бінарного пошуку, проте масив розбивається не на дві частини, а на три, що має прискорити пошук. Псевдокод алгоритму наведено нижче:

```
int TernSearch (int left, right, key, A[])
{while (right >= left)
        {int mid1 = left +( right - left)/3
        int mid2 = right -( right - left)/3
                if (A[mid1] = = key)
                        {return mid1}
                If (A[mid2] = = key)
                        {return mid2}
                If (key<A[mid1]) { right = mid1-1}
                elseIf (key >A[mid2]) { left = mid2+1}
                else { left = mid1+1,  right = mid2-1}
        }
        return -1
}
```

C

Приклад роботи алгоритму тернарного пошуку.

Дано впорядкований за зростанням масив з n = 5 елементів (рисунок 4.2), задача - перевірити чи є “60” в масиві та якщо є , то надати номер

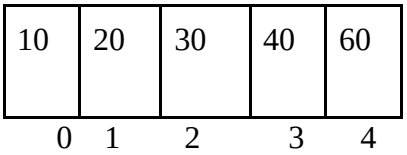


Рисунок 4.2. Впорядкований за зростанням масив

Відповідно до роботи алгоритму, маємо left = 0, right = 4, n=5 key = 60. Задача визначити номер key в списку.

mid1 = left +(right - left)/3 = 0+(4-0)/3=1

mid2 = right - (right - left)/3 =4-(4-0)/3=3

Оскільки key = 60>A[3] = 40 маємо left = mid2+1 = 4. Надалі mid1 = left +(right - left)/3 = 4+(4-4)/3=4 та A[mid1]=A[4] = 60 = = key

Обрахуємо часову складність алгоритму тернарного пошуку. Для n змінних в масиві маємо час виконання

$T(n)=C+T(n/3),$ (4.6)

де C = стала – час виконання команд, як показано на псевдокоді алгоритму.

Масив A ділиться на 3 під масиви кожного разу. час виконання кожного із 3 під масивів

$T(n/3)=C+T(n/3/3)=C+T(n/9)$ (4.7)

$T(n/9)=C+T(n/9/3)=C+T(n/27)$ (4.8)

Підставимо (4.8) в (4.7) та (4.7) в (4.6)

$T(n)=C+C+C+T(n/27)=3C+T(n/27)=3C+T(n/3^3)$. За математичною індукцією приходимо до формули:

$T(n)=kC+T(n/3^k)$ (4.9)

На певному часовому етапі

$T(n/3^k) = T(1) = const$ (4.10)

Звідси $n/3^k=1$ та отримаємо $n=3^k => k = \log_3 n$.

Підставимо k та (4.6) в (4.9):

$T(n)=\log_3 n*C+C$, де C, T(1)-константи. Тому часова складність алгоритму тернарного пошуку $T(n)<\log n$ і нотація Ландау O(log n).

В середньому і найгіршому випадку O(log n), як показано вище. В найкращому випадку - відразу перший елемент є шуканим. Тоді час виконання алгоритму буде сталим і рівним C а часова складність складе O(1).

4.4. Алгоритм пошуку стрибками (Jump Search)

З порівняння часової складності алгоритмів Бінарного та Тернарного пошуку видно, що час виконання залежить від розбиття масиву на під масиви. І якщо в бінарному алгоритмі таких під масивів було 2, а в тернарному 3, то маємо час виконання пропорційний логарифму за основою 2 та 3. А оскільки log3 N < log2 N то очевидно є сенс розбивати масив для пошуку на ще більшу кількість під масивів.

На цій ідеї збудовано алгоритм пошуку стрибками, де встановлено максимальну кількість під масивів на які розбивається початковий масив.

Псевдокод алгоритму приведено нижче.

```
int JumpSearch (int A[], N, key)
{int BlockSize=√N
start=0
end=BlockSize
while (A[end]<=key & end<N)
        {start = end
        end = end + BlockSize
        If(end > N-1)
                {end=N}
        }
For (i=start to (end-1))
        {If (A[i]=key)
                {return i}
        }
return -1
}
```

На рисунку 4.3 розглянемо роботу алгоритму для масиву із 12 елементів. Маємо N=12, BlockSize = $\sqrt{N}$ = 3. Завдання - визначити місце в масиві елемента key = 120.

	v			v			v			v	
0	1	2	3	4	5	6	7	8	9	10	11
10	20	30	40	50	60	70	80	90	100	110	120

Рисунок 4.3. Робота алгоритму

«v» позначено start-end елементи в блоках

jump0: start=0 end=3 A[3]=40<=120 & 3<12

jump1: start=3 end=6 A[6]=70<=120 & 6<12

jump2: start=6 end=9 A[9]=100<=120 & 9<12

jump3: start=9 end=12 A[12] немає тому A[11]<=120 & 12<=12

Далі в межах Jump3 виконується блок For в псевдокоді як лінійний пошук, як показано на рисунку 4.4.

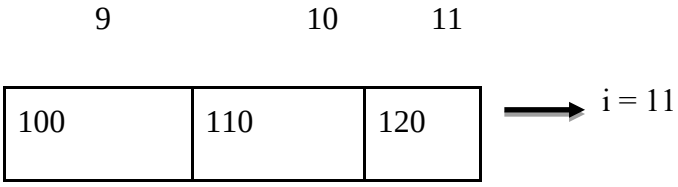


Рисунок 4.4. Робота блоку For

Обрахуємо часову складність алгоритму пошуку стрибками. Якщо n - загальна кількість елементів в масиві A[], m - розмір блока, то n/m-кількість блоків.

Маємо часову складність O(n/m+(m-1)). Тут перший доданок визначає кількість стрибків, а другий – час роботи лінійного пошуку в заданому блоці.

Виходячи з мінімізації значення часової складності, обрахуємо екстремум функції O(n/m+(m-1)), прирівнявши похідну функції нулеві. Тобто, визначимо оптимальне значення m:

$$\frac{d}{dm}\left(\frac{n}{m}+(m-1)\right)=0$$

$$-\frac{n}{m^2}+1=0 \quad n=m^2\,,\; m=\sqrt{n}$$

Підставимо отримане значення m, що і було використане у псевдокоді, у формулу O(n/m+(m-1)) = O ($2\sqrt{n}$- 1). Тобто, часова складність алгоритму O($\sqrt{n}$).

4.5. Алгоритм лінійного пошуку (Linear search)

Попередні алгоритмів пошуку побудовано на парадигмі «divide and conquer» - ділення проблеми на під проблеми для спрощення задачі. В той же час алгоритм лінійного пошуку відноситься до алгоритмів, що використовують «грубу силу» (brute force algorithm) – правильна відповідь знаходиться шляхом перебору всіх можливих варіантів. Іншим прикладом таких алгоритмів «грубої сили» є алгоритми підбору пароля в криптографії.

Псевдокод алгоритму лінійного пошуку наведено нижче

```
LinearSearch (A[], N, key)
{For (i=0 to (N-1))
        {if A[i]=key
                {return i}
        else
                {return -1}
        }
}
```

Обрахуємо часову складність алгоритму лінійного пошуку. Часова складність алгоритму лінійного пошуку в найкращому випадку, якщо шуканий елемент key знайдеться відразу є сталою і не залежить від розміру масиву - best case O(1). У найгіршому випадку, коли шуканий елемент key буде останнім у масиві, часова складність визначатиметься перебором всіх варіантів - worst case O(n). Аналогічна ситуація буде і у якомусь проміжному випадку, який варіюється від найкращого до найгіршого - average case O(n).

Єдиною перевагою лінійного пошуку, на відміну від інших алгоритмів, є відсутність роботи з відсортованим масивом.

4.6. Експоненційний пошук або галопуючий пошук (Exponential Search)

Алгоритм експоненційного пошуку використовується для пошуку номеру елемента в упорядкованому масиві. Псевдокод експоненційного пошуку наведено нижче

```
ExponentialSearch (A[], left, right, key)
{If (left > = right)
```

{return -1}

i = 1

while (i<(right - left))

{If A[] < key

i=i*2

else break the loop

}

call BinarySearch (A[], i/2, min (i,right), key)

}

Із псевдокоду видно, що після встановлення інтервалу пошуку запускається бінарний алгоритм пошуку в знайденому інтервалі.

Приклад роботи алгоритму експоненційного пошуку key =111 в масиві A показано на рисунку 4.5.

0	1	2	3	4	5	6	7	8	9	10	11
11	21	31	41	51	61	71	81	91	101	111	121

Рисунок 4.5. Приклад роботи алгоритму експоненційного пошуку

Покроково отримаємо

i=1 A[1]=21<111

i=2 A[2]=31<111

i=4 A[4]=51<111

i=8 A[8]=91<111

i=16 A[11]=121 < 111- False call BinarySearch(A[], i/2 =8, i=11, key)

Далі працює алгоритм BinarySearch, як показано на рисунку 4.6.

8	9	10	11
91	101	111	121

Рисунок 4.6. Робота алгоритму BinarySearch

На першій ітерації ціле значення mid складе

mid = (8+11)/2 = 9. A[9]=101 < 111. Далі left = mid+1= 10.

Друга ітерація в BinarySearch

mid = (10+11)/2 = 10. A[10]=111 = = key =111.

Обрахуємо часову складність алгоритму експоненційного пошуку і відобразим її в таблиці 4.2. Підрахуємо скільки разів виконується Exponential Search

Таблиця 4.2. Час виконання алгоритму

k =	0	1	2	3	k
$i=2^k$	1	2	4	…	$2^k \leq n$
Виконання	C	C	C	…	C

В найгіршому випадку алгоритм експоненційного пошуку виконується $2^k = n$ разів, тобто $k = \log_2 n$

Час виконання алгоритму складає C+C+C+… , всього k разів. Тобто, маємо T(n) = k*C = $\log_2$n*C, де C=const. Або при переході до нотації Ландау, маємо O(log n)

Частина BinarSearch має нотацію O(log m), де m – інтервал для пошуку в масиві, $i/2 \leq m \leq i$.

Оскільки m< n, то O(log m) < O(log n) і часова складність визначається за більшим значенням, то O(log n) в найгіршому випадку.

Якщо для середнього випадку вживати i, де i - поточний індекс до якого добігає алгоритм в частині Exponential Search, то отримаємо $2^k = i$, тобто $k = \log_2 i$. Тоді нотація Ландау для середнього випадку O(log i).

В найкращому випадку, якщо алгоритм експоненційного пошуку відразу, за першу ітерацію знайде шуканий елемент, маємо сталий час виконання, який не залежить від розміру масиву. Тобто, T(n) = C, де C – стала і часова складність складе O(1).

Алгоритм Exponential Search, на відміну від інших алгоритмів пошуку, можна використовувати якщо не відоме значення right кінця масиву.

Просторова складність алгоритму експоненційного пошуку.

Просторова складність алгоритму (space complexity) визначає розмір пам’яті для роботи алгоритму. Відповідно до алгоритму Інтерполяційного пошуку, окрім масиву A[] який вже задано і тому не враховується, в пам’ять для роботи алгоритму необхідно занести наступні змінні key, left, right, i. Значення змінних може змінюватися в ході роботи алгоритму, однак пам'ять зайнята ними буде звільнена лише після закінчення роботи алгоритму. Підрахуємо кількість пам’яті. Якщо всі 4 змінні матимуть тип int, то в

цілому потрібно 16 В пам’яті. І це значення не залежить від кількості елементів n у масиві. Тобто, маємо - для роботи алгоритму потрібно сталий розмір пам’яті, або просторова складність алгоритму O(1).

Аналогічну просторову складність O(1) мають і розглянуті раніше ітераційні алгоритми пошуку.

4.7. Алгоритм інтерполяційного пошуку (Interpolation Search)

Ідея алгоритму полягає у визначенні місця елемента в зростаючому списку на основі лінійної інтерполяції даних. Власне лінійна інтерполяція ґрунтується на рівнянні прямої, що має вигляд [2]:

$$y = mx + b \tag{4.11}$$

Припустимо, що пошук проводиться в масиві A, впорядкованому за зростанням, з мінімальним індексом left та максимальним індексом right. Задачею є встановлення місця в масиві шуканого елемента key.

На основі рівняння прямої (4.11):

$$A[left] = m\ left + b \tag{4.12}$$

$$A[right] = m\ right + b \tag{4.13}$$

$$key = m\ position + b, \tag{4.14}$$

де position - індекс в масиві A шуканого елемента key. Тобто A[position] = key. Визначимо m, віднявши від (4.13) рівняння (4.12)

A[right] - A[left] = m (right - left).

Звідси:

$$m = \frac{A[right] - A[left]}{right - left} \tag{4.15}$$

Визначимо *position*, віднявши від (4.14) рівняння (4.12):

$key - A[left] = m(position - left)$, $m \cdot position = m \cdot left + (key - A[left])$ і далі $position = \frac{m \cdot left + (key - A[left])}{m}$.

З урахуванням (4.15), маємо інтерполяційну формулу:

$$position = left + (key - A[left]) \frac{right - left}{A[right] - A[left]}$$

Псевдокод алгоритму Інтерполяційного пошуку

```
int InterpSearch (int A[], left, right, key)
{int position
    If (key >=A[left]&key<=A[right])
        {position = left+(key-A[left])(right - left)/(A[right]-A[left])}
    If(A[position] = = key)
        {return position}
    If (A[position] < key)
        {return InterpSearch(A[], (position+1), right, key)}
    If (A[position] > key)
        {return InterpSearch (A[], left, (position-1), key)}
return -1
}
```

Для оцінки просторової складності алгоритму інтерполяційного пошуку розглянемо три випадки з різною часовою складністю цього алгоритму інтерполяційного пошуку.

Приклад 1. Дано заповнений за зростанням масив А. До того ж заповнення за зростання лінійне, рівномірне - різниця двох сусідніх елементів є сталою. В цьому разі знайти position для key елементу можна за одну рекурсію, як показано на рисунку 4.7 нижче.

0	1	2	3	4	5	6	7
12	32	52	72	92	112	132	152

Рисунок 4.7. Робота алгоритму

Для даного прикладу маємо key = 92 (припустимо), left = 0, right = 7, A[left] = 12, A[right] = 152.

Визначимо position, користуючись Інтерполяційним пошуком

$$position = left + (key - A[left])\frac{right - left}{A[right] - A[left]} = 0 + (92 - 12)\frac{7 - 0}{152 - 12} = 4,\ \text{A[4] = 92 = =key}$$

Приклад 2. Дано заповнений за зростанням масив А. Проте, заповнення за зростання не є лінійним та не рівномірне - різниця двох сусідніх елементів не є сталою. В цьому разі знайти position для key елементу можна за n ітерацій, де n – розмір масиву, як показано на рисунку 4.8.

0	1	2	3
10	20	30	1010

Рисунок 4.8. Заповнений за зростанням масив

Для даного прикладу маємо key = 30 (припустимо), left = 0, right = 3, A[left] = 10, A[right] = 1010.

Визначимо position, користуючись інтерполяційним пошуком

I рекурсія $position = 0 + (30 - 10)\frac{3-0}{1010-10} \approx 0$, маємо A[0] = 10 < key = 30 тому left = position+1=1

II рекурсія

$position = 1 + (30 - 20)\frac{3-1}{1010-20} \approx 1$, маємо A[1] = 20 < key = 30 тому left = position+1=2

III рекурсія

$position = 2 + (30 - 30)\frac{3-2}{1010-30} \approx 2$, маємо A[2] = 30 == key = 30

Тобто, у найгіршому випадку - заповнення масиву не є лінійним та рівномірним, щоб дійти до третього елементу – потрібно 3 ітерації. Щоб дійти до 4 – потрібно чотири. Маємо, часову складність інтерполяційного пошуку O(n).

Приклад 3. Дано заповнений за зростанням масив A. Проте, заповнення за зростанням даних відповідає біноміальному розподілу, що відповідає середньому випадку часової складності алгоритму. В цьому разі знайти position для key елементу можна алгоритм, що має часову складність O (log (log n)). Для обґрунтування нижче дано таблицю 4.3. експериментальних даних [1], які вказують на час роботи, залежно від розміру n масиву.

Таблиця 4.3. Експериментальні дані часу виконання алгоритму

k	Nk	log (log n)	Середнє значення часу виконання
6	64	2,585	2,451
10	1024	3,322	3,343
14	16 384	3,807	3,992
18	262 144	4,17	4,097

4.8. Лабораторна робота. Алгоритми пошуку

Мета роботи – для заданого впорядкованого масиву оцінити час роботи двох алгоритмів пошуку елемента. Вивести порядковий номер елемента, що шукаємо.

В роботі розглядаються наступні алгоритми пошуку елемента у впорядкованому масиві

1. Бінарний пошук (Binary Search)
2. Експоненційний пошук (Exponential Search)
3. Інтерполяційний пошук (Interpolation Search)
4. Пошук стрибками (Jump Search)
5. Лінійний пошук (Linear Search)
6. Потрійний пошук (Ternary Search)

Завдання до роботи. За умовою роботи спочатку формуємо впорядкований одновимірний масив з N цілих чисел. Надалі - відповідно варіанту, наведеного в таблиці 1, записати два алгоритми пошуку елемента в масиві та отримати порядковий номер елемента.

В ході роботи оцінити час роботи обраних двох алгоритмів (для варіантів 1…15) чи кількість проведених порівнянь (для варіантів 16…30) за двох умов – 1) за умови відсутності елемента у впорядкованому масиві та 2) за умови наявності елемента у впорядкованому масиві, вивести його номер у масиві

Таблиця 4.4. Алгоритми пошуку у впорядкованому масиві, відповідно до варіанту

Варіант	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Завдання	1,2	1,3	1,4	1,5	1,6	2,3	2,4	2,5	2,6	3,4	3,5	3,6	4,5	4,6	5,6

Отримані результати занести в таблицю 4.5.

Тобто, відповідно до таблиці 4.4, студент, варіант якого 2 виконує лабораторну з алгоритмами 1,3 – Бінарного та Інтерполяційного пошуку та визначає час роботи цих алгоритмів.

Обов’язково зробити висновки – який з алгоритмів працює краще і чому. Описати обрані алгоритми.

Таблиця 4.5. Результат роботи - час роботи обраних алгоритмів

К-сть елементів N у впорядкованому масиві	N =10	N=1 000	N=1 000 000	N=1 000 000 000
Час чи кількість порівнянь, за відсутності елемента				
Час чи кількість порівнянь, за наявності елемента				

5. Базові структури даних

В цілому структури даних – це спосіб зберігання та керування даними в пам’яті машини для організації ефективного доступу до них та їх зміни [3-5].

Розглянемо структури даних масив та список (Array, List).

5.1. Масив

Масив (Array) - структура даних, що зберігає та керує даними в цільній і не розривній частині пам’яті. Саме тому менеджеру пам’яті зрозуміло де розміщено перший, другий і т.д. елементи масиву.

Розглянемо часову складність алгоритмів керування даними при роботі з масивом. Зокрема, несортованим масивом. Розглянемо алгоритми вставки, пошуку, знаходження мінімального (максимального), видалення мінімального (максимального) елемента в масиві.

В таблиці 5.1 наведено часову складність (time complexity) ряду базових алгоритмів при роботі з несортованим масивом.

Таблиця 5.1. Часова складність ряду базових алгоритмів

Алгоритм	Вставки	Пошуку	Знаходження min	Видалення min
Несортований масив	O(1)	O(n)	O(n)	O(n)

Як видно з вищенаведеної таблиці, алгоритм вставки має часову складність O(1) – маємо вільне місце і вносимо туди дані (фактично операція присвоювання).

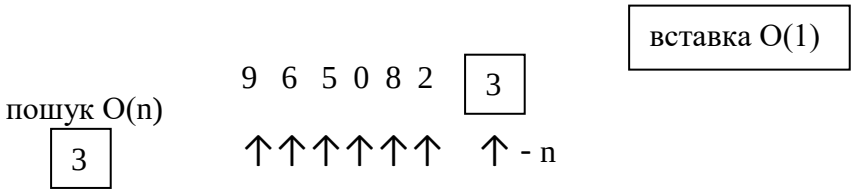


Рисунок 5.1. Вставка в масив

Однак, алгоритм вставки працює в парі з алгоритмом пошуку – варто спочатку встановити чи є вільне місце у місце у виділеній під масив пам’яті.

І власне алгоритм пошуку має часову складність O(n), оскільки алгоритму пошуку у найгіршому випадку буде переглядати весь масив для пошуку вільного місця, зокрема.

Як приклад, на рисунку 5.1. показано масив, що містить 6 цілих чисел у який потрібно додати нове число «3». Для цього спочатку запускається алгоритм пошуку, що проходить весь масив, шукаючи вільне місце у виділеній під масив пам'яті та має часову складність O(n).

Надалі, після цього виконується алгоритм вставки «3» на вільне місце, що має часову складність O(1).

На рисунку 5.2 показано роботу алгоритму пошуку мінімального елемента. Верхній рядок вказує на значення елемента в масиві, а нижній – на встановлене значення мінімального елемента.

Зрозуміло, що при роботі з не впорядкованим масивом потрібно перебрати всі елементи для знаходження мінімального.

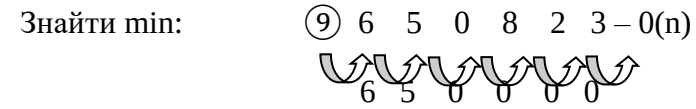


Рисунок 5.2. Пошук мінімального

На рисунку 5.3 показано роботу алгоритму видалення мінімального елемента. Спочатку виконується алгоритм пошуку, часова складність якого O(n), в даному прикладі виконується за n/2 операцій. Однак, оскільки в масиві не допустимі порожні місця, далі запускається алгоритм зсуву даних для заповнення порожнього місця в масиві. Часова складність алгоритму зсуву O(n), в даному прикладі виконується за n/2 операцій.

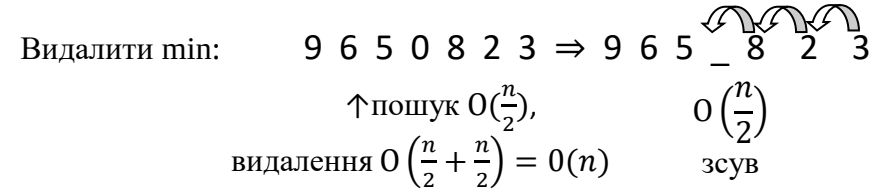


Рисунок 5.3. Видалення з масиву

Розглянемо впорядкований масив та ті ж алгоритми вставки, пошуку, знаходження мінімального (максимального), видалення мінімального (максимального) елемента у впорядкованому масиві.

В таблиці 5.2 наведено часову складність (time complexity) ряду базових алгоритмів при роботі з впорядкованим масивом.

Таблиця 5.2. Часова складність ряду базових алгоритмів

Алгоритм	Вставка	Пошук	Знайти min	Видалити min
Впорядкований масив (за зростанням)	O(n)	$O(\log n)$	O(1)	O(n)

Алгоритм вставки у впорядкований масив потребує виконання трьох алгоритмів: алгоритму пошуку вільного місця, що має часову складність $O(\log n)$ – наприклад, алгоритм бінарного пошуку; алгоритму вставки у вже знайдене вільне місце, що має часову складність O(1), як вказувалося раніше; та алгоритму зсуву елементів масиву, щоб забезпечити його впорядкованість. Часова складність алгоритму зсуву O(n).

Маємо часову складність алгоритму вставки в цілому $O(\log n+1+n)$ і за домінуючу функцію маємо O(n).

Алгоритм знаходження мінімального (максимального) елементу має фіксований час роботи та часову складність O(1), оскільки знайти мінімальний (максимальний) елемент у впорядкованому масиві це взяти перший, або останній елемент.

Алгоритм видалення мінімального елемента у впорядкованому масиві містить два алгоритми – знаходження мінімального (часова складність O(1) та зсуву масиву (часова складність O(n), як показано на рисунку 5.4. Часова складність такого алгоритму визначається домінуючою часовою складністю алгоритму зсуву і складає O(n)

Видалити min: Знайти і зсунути $0(1+n) = 0(n)$.

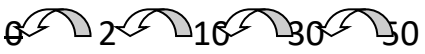


Рисунок 5.4. Видалення мінімального

Таким чином, використання масиву для зберігання даних має ряд недоліків. Зокрема, якщо масив не впорядкований, то алгоритми знаходження мінімального чи видалення мають часову складність O(n), хоча алгоритм вставки в цьому випадку працює чудово з часовою складністю O(1). Проте, можна використати впорядкований масив і в цьому випадку алгоритми знаходження мінімального, або видалення, мають часову складність O(1). Однак, алгоритм вставки, на відміну від попереднього, працює з часовою складністю O(n).

В цілому, масив – як структура даних не може задовольнити всі вимоги при роботі з базовими алгоритмами. Саме тому існує цілий ряд інших структур даних, при роботі з якими часова складність базових алгоритмів є більш прийнятною.

Однією з таких структур є зв’язані списки.

5.2. Зв’язаний список

Зв’язаний список – це структура даних, що не потребує виділення цілого і не розривного розміру пам’яті, на відміну від масиву.

В цілому, в структурах даних використовують

- одно(бічно) зв’язаний список,
- дво (бічно) зв’язаний список,
- кільцевий список,
- кільцевий дво(бічно) зв’язаний список.

Однозв’язний список використовується найчастіше. В ньому кожен вузол містить значення та адресу наступного вузла, як показано на рисунку нижче. Тобто, кожен вузол зв’язаний через адресу з наступним вузлом. Адреса – це вказівник на наступний вузол.

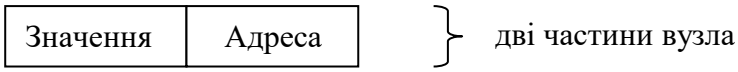


Рисунок 5.5. Вузол списку

На рисунку 5.6 дано логічну схему однозв’язного списку, який містить три вузли. Кожен з вузлів, має свою адресу в пам’яті. Наприклад 100, 200, 300. Також є початковий вказівник Head на перший елемент списку.

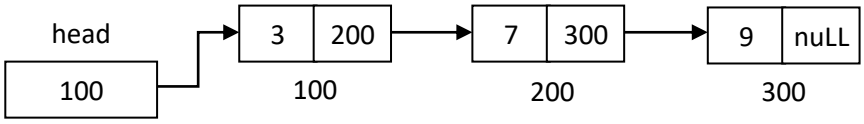


Рисунок 5.6. Логічна схема однозв’язного списку

Список називається однозв’язний, оскільки є лише один вказівник на наступний вузол. Тобто, по такому списку можна рухатися в одному напрямку. Наприклад, вузол №2 знає лише наступний вузол №3 і не знає попереднього, №1.

Реалізація однозв’язного списку мовами C та Java наведена нижче

Реалізація C:	Реалізація Java

struct node
{ int data;
struct node* next;
}
	Class Node{
	ind data;
	Node next=nuLL;
	}

- дво(бічно) зв’язаний список.

  Двобічно зв’язаний список (DLL) складається з вузлів. Кожен вузол має три частини: значення та два вказівники з адресами на наступний вузол та на попередній, як показано на рисунку 5.7.

Рисунок 5.7. Вузол двозв’язного списку

Логічну схему двобічно зв’язаного списку подано на 5.8.

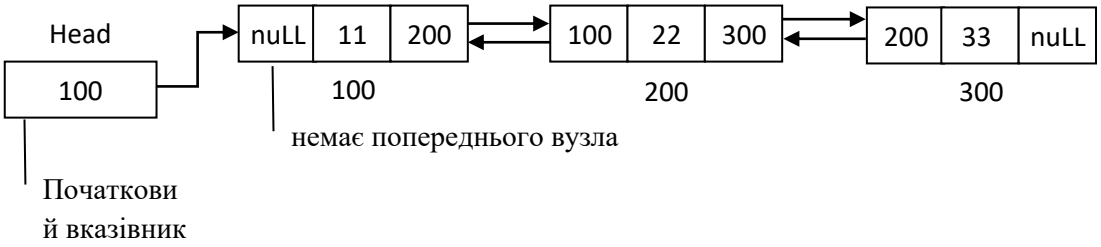


Рисунок 5.8. Логічна схема двозв’язаного списку

Приклад реалізації вузла DLL мовою С

```
struct node
{ int data;
struct node * next;
struct node * prev;
}
```

- Кільцевий список, логічна схема якого наведена нижче. Логічна схема кільцевого списку, як різновиду однозв’язного списку подана на рисунку 5.9.

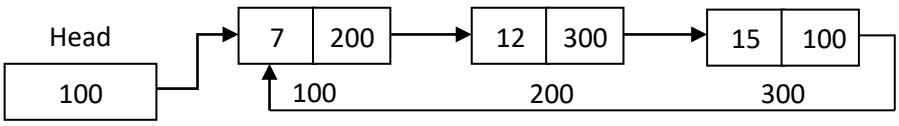


Рисунок 5.9. Логічна схема кільцевого списку

- Кільцевий дво(бічно) зв’язаний список, логічна схема якого дана нижче на рисунку 5.10.

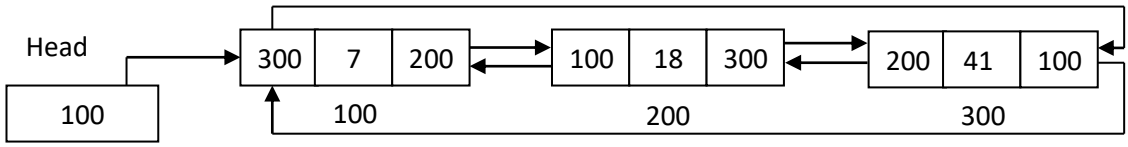


Рисунок 5.10. Логічна схема кільцевого двобічно зв’язаного списку

Розглянемо властивості списків на прикладі однозв’язного списку.

5.3. Однозв’язний список

Розглянемо однозв’язний і не впорядкований список. Часова складність основних алгоритмів при роботі з однозв’язним списком наведена у таблиці 5.3.

Таблиця 5.3. Часова складність основних алгоритмів

	Вставка	Пошук	Знайти min	Видалити min
Однозв’язний список, не впорядкований	O(1)	O(n)	O(n)	O(n)

Вставка елемента «1» у не впорядкований список має часову складність O(1). При вставці вказівник від Head вказує на новий елемент, а вказівник від нового елемента вказує на існуючий список.

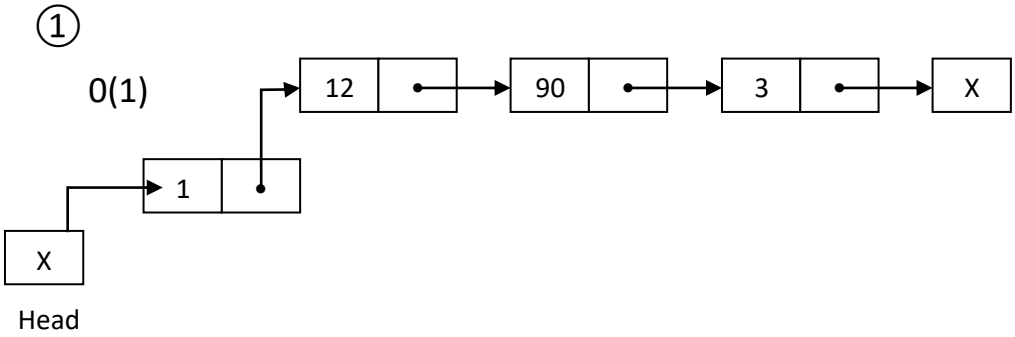


Рисунок 5.11. Вставка елемента «1» у не впорядкований список

Алгоритми пошуку і знаходження мінімального з таблиці 5.3 мають часову складність O(n), аналогічно до роботи з масивом. Алгоритм видалення мінімального має

часову складність O(n), оскільки включає в себе алгоритм знаходження мінімального, складність якого O(n) та алгоритм видалення, складність якого O(1). В цілому часова складність алгоритму видалення мінімального - O(n).

Однозв'язний список. Робота з пам'яттю

Для поглиблення навичок роботи зі зв'язним списком розглянемо роботу менеджера пам'яті. Кожен байт пам'яті має певну адресу, припустимо ці адреси починаються зі 100, як показано на рисунку 5.12.

Для менеджера пам'яті пам'ять виступає як ресурс, яким він керує. Є різні способи керувати пам'яттю, наприклад у мові Java використовується Збірник мусора (Garbage Collector), що автоматично звільнює пам'ять.

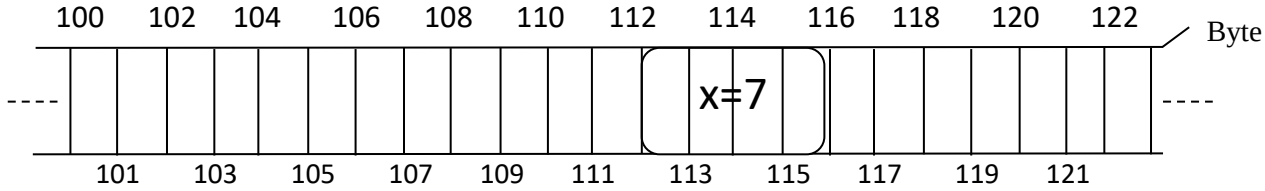


Рисунок 5.12 Логічна схема оперативної пам'яті

Розглянемо, що відбувається з пам'яттю, якщо оголошується змінна і виконується код:

```
int x=7;
```

У цьому разі менеджер пам'яті виділяє 4 Байти (для типу int) для зберігання x і, припустимо, це виділення буде починатися з байту номер 112, як показано на рисунку вище.

Варіант використання класу-обгортки,

```
integer x=3;
```

що займає 16 Байт, не розглядаємо.

2) Розглянемо як працює менеджер пам'яті при оголошенні масиву, що містить, наприклад, три значення типу integer. Припустимо виконується код

```
int [ ] arr = new int [3];
```

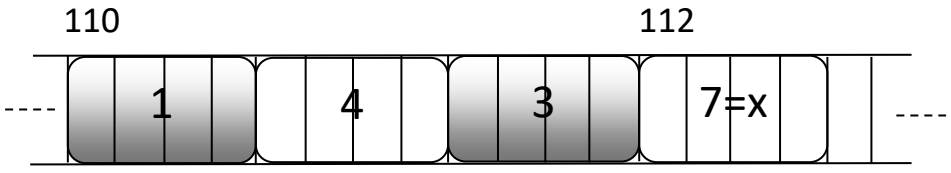


Рисунок 5.13 Логічна схема виділеної оперативної пам'яті

Для цього масиву менеджер виділяє пам'ять, єдиним розміром 3x4=12 Байт. На рисунку 5.13 показано виділену пам'ять від 100 до 112 Байту.

Припустимо, розробник вирішує, що його масив має містити 4 змінних типу Integer. Тобто, менеджеру пам’яті потрібно всього виділити 4 Байти під ще одну змінну. Проте, як писалося вище, за адресом 112 вже розміщено Int x=7. Виходить менеджер не може розширити місце в пам’яті під масив із 4 типу Int. Тоді в пам’яті виділяється новий блок під масив, розміром із 4·4=16 Байт - оскільки Array містить саме блок даних. Старі дані, які були в блоці під масив на 12 Байт будуть перенесені до нового виділеного блоку. Все працює.

Однак, припустимо потрібен масив, що містить 50 значень типу Int. Для цього менеджер пам’яті виділяє в пам’яті блок на 50·4=200 Байт. Наприклад цей блок розпочнеться в 122 Байті та закінчиться в 321 Байті. Однак, якщо, програмісту виявиться потрібними лише 10 елементів із заявлених 50, то (50-10)·4 = 160 Байт пам’яті – втрачено і використано лише 40 Байт. У цьому і полягає головна проблема при використанні масивів - програміст має наперед знати розмір масиву, щоб не втрачати пам’ять.

Вирішенням цієї проблеми для менеджера пам’яті є використання зв’язних списків.

Зв’язний список (Linked List) – це колекція елементів, як і масив (Array). Проте, при роботі менеджера пам’яті зі зв’язним списком немає потреби зберігати елементи послідовно, виділяючи пам'ять єдиним блоком, як для масиву.

При роботі зі зв’язним списком немає потреби виділяти пам’ять блоками, менеджер виділяє блок пам’яті, попписаний в коді та який йому потрібен на даний момент. Якщо в подальшому буде потрібен додатковий блок пам’яті, то – менеджер виділяє пам'ять додатково. Якщо в зв’язному списку, умовно кажучи, буде оголошено лише один елемент, то зберігається один елемент, не турбуючись про не раціональне використання пам’яті. І якщо в подальшому, припустимо, менеджер пам’яті вносить в зв’язний список, а зв’язний список – в пам’ять, зміну типу Int (навіть не одну) то жодних проблем з їх розміщенням в пам’яті не буде, як показано на рисунку 5.4.

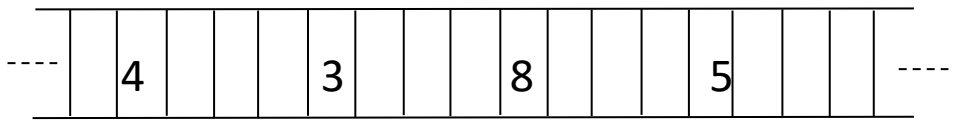


Рисунок 5.14 Логічна схема оперативної пам’яті при роботі зі списком

Цей набір даних, що показано на рисунку 5.14 – колекція в зв’язному списку і її особливістю є відсутність блочного виділення пам’яті, на відміну від масиву з його послідовним розміщенням елементів.

Розглянемо спосіб реалізації колекції в зв’язному списку.

Оскільки елементи колекції в зв’язному списку не розміщені послідовно, то має бути додаткова інформація – де вони розміщені.

Тобто, у кожного елемента має бути адреса наступного елемента колекції і так далі. Таким чином, можна знайти всю колекцію.

Виходить, кожен елемент в зв’язному списку, окрім даних, має містити також адресу наступного елемента. В зв’язному списку це називається вузол, який містить елемент та адресу наступного елемента. Завдяки цьому, зв’язний список містить послідовність, але не потребує послідовного розміщення елементів у пам’яті.

На рисунку 5.15 показано приклад коли адреси з 108 до 112 вже зайняті змінною, типу int. Однак, менеджер пам’яті заносить одну частину зі списку, змінна «4» до адреси 108, а іншу – після адреси 112 – змінна «3». Однак, оскільки в зв’язному списку вузол містить елемент та адресу наступного елемента, то менеджер пам’яті виділяє 4 Байти під значення, типу int та ще 4 байти під адресу наступного елементу.

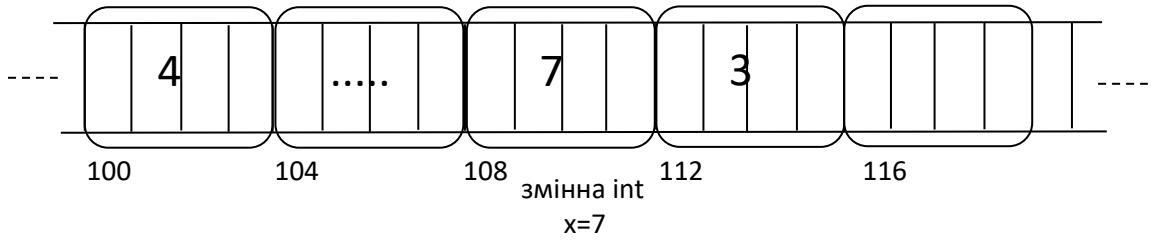


Рисунок 5.15 Логічна схема оперативної пам’яті з зайнятими адресами

Представлення цієї частини пам’яті у вигляді логічної схеми подано на рисунку 5.16.

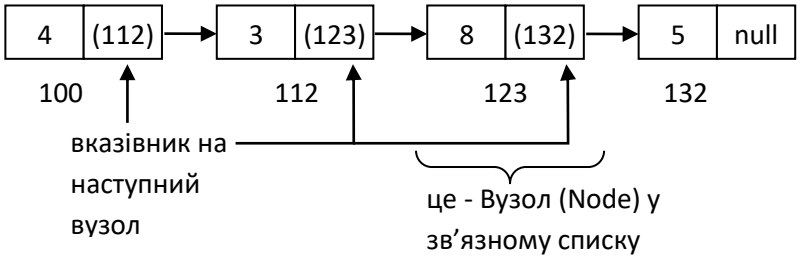


Рисунок 5.16 Логічна схема зв’язного списку

Якщо, припустимо, треба додати елемент в зв’язний список, то він додається в кінець списку. При цьому вказівник null на останній елемент замінюється на вказівник на наступний вузол.

В результаті, на відміну від роботи з пам’яттю при створенні масиву, у даному разі – при роботі зі зв’язним списком, менеджер пам’яті не виділяє ніяких додаткових Байтів, які можливо ніколи не будуть використані.

Як показано на рисунку нижче, при додаванні у зв’язний список «10» створюється новий вузол під який виділяється 8 Bytes.

Описані вище операції показано на рисунку 5.17.

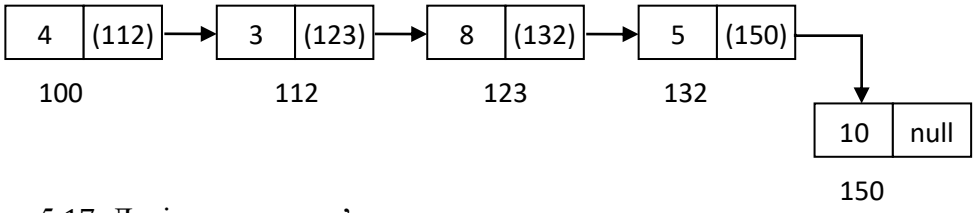


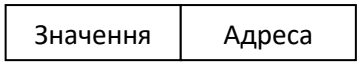
Рисунок 5.17 Логічна схема зв’язного списку

Надалі у останній вузол за адресою 132 замість null заноситься адреса нового вузла, що містить елемент «10» за адресою, наприклад 150.

У новий останній вузол №150 заноситься null, як ознака останнього елементу.

Для декларування в коді зв’язного списку потрібно декларувати вузол, що містить значення та адресу.

Вузол:



- може бути два типи даних в одному вузлі.

Рисунок 5.18 Логічна схема вузла зв’язного списку

Тобто, наступним чином визначається нова структура даних

```
        Static class Node
{
            int value;              //значення
            Node next;              // вказівник на наступну адресу нового вузла
    Public     Node (int d)         // тип змінної у вузлі, на який буде посилання
              {
              value =d;
              next = null;
              }
}
```

До зв’язного списку (Linked List) також додається початкову точку Head pointer – вказівник, що зберігає адресу першого вузла.

Логічна схема має наступний вигляд, показаний на рисунку 5.19.

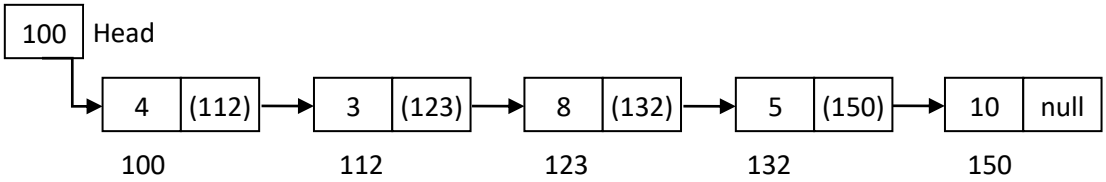


Рисунок 5.19 Логічна схема зв'язного списку з вказівником

Особливістю зв'язного списку є відсутність безпосереднього доступу до елемента колекції, як в масиві, наприклад Arr[2]. Тому, щоб дійти до певного елемента у зв'язному списку, потрібно пройти всіма елементами цього списку. В списку не можна відразу вказати адресу 2-го, N-го елемента колекції, оскільки її ніхто не знає і це є недоліком зв'язного списку.

При роботі зі списком – знаємо адресу, переходимо туди і там шукаємо потрібне значення. Якщо його немає, то беремо наступну адресу, йдемо за нею до наступного значення і т.д.

Тобто, зв'язний список, на відміну від масиву, не має прямого доступу до даних. Зокрема, якщо потрібні дані із останнього вузла – доведеться пройти весь список повністю. Тому пошук у списку має часову складність 0(n), як показано на рисунку нижче.

На рисунку 5.20 показано логічну схему однозв'язного списку, що містить 5 елементів «4, 3, 8, 5, 10». Кожен з елементів розміщено у вузлі, що містить також адресу наступного елементу «112, 123, 132, 150, null». Початкова точка (Head pointer) вказує на початковий елемент однозв'язного списку.

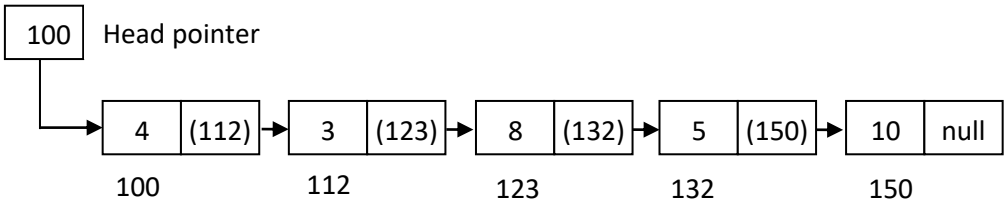


Рисунок 5.20 Логічна схема однозв'язного списку

У випадку вставки у список нового вузла, що містить елемент «18» він займає вільне місце в пам'яті (наприклад 200) та буде містити адресу наступного елементу «100», як показано на рисунку 5.21. Тобто, зв'язний список – це не лінійний масив пам'яті, це колекція елементів.

Таким чином, часова складність основних алгоритмів для роботи з однозв'язним списком відповідає часовій складності аналогічних алгоритмів для роботи з масивом. Проте, на відміну від масиву, просторова складність при роботі з однозв'язним списком може бути меншою, оскільки список – це колекція елементів.

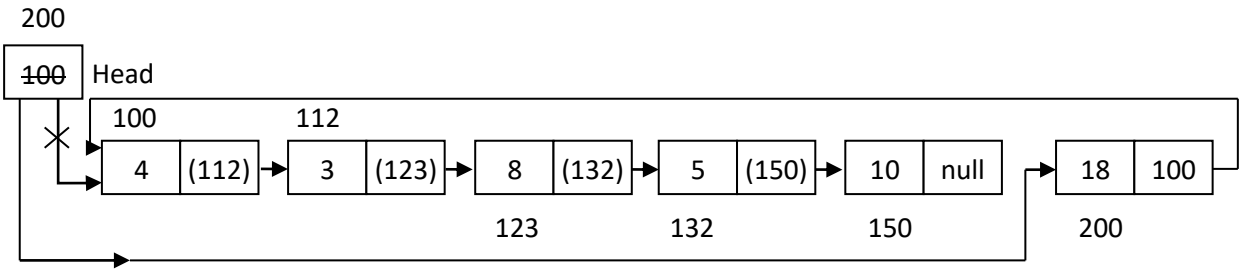


Рисунок 5.21 Логічна схема вставки елемента в однозв'язний список

Тому існують інші структури даних, які не мають вищеназваних недоліків.

5.4. Лабораторна робота. Масив, зв'язаний список

1. Виконати завдання лабораторної за варіантом, використовуючи масив (звичайний масив, що повністю (саме повністю !) заповнений на 10 елементів та потім на 10 000 елементів).

2. Виконати завдання лабораторної за варіантом, використовуючи двобічно зв'язаний список (DLL) на 10 елементів і потім на 10 000 елементів.

Встановити який код працює швидше та заповнити таблицю 5.4. Результати пояснити.

Таблиця 5.4. Результати роботи

Час виконання	10 елементів	10 000 елементів
Масив		
Двозв'язний список		

Додаткові завдання

a) Створити однобічно зв'язаний список, вивести його. Якщо в списку є шуканий елемент, то вилучити його, а два наступні поміняти місцями.

b) Виконати завдання лабораторної за варіантом, використовуючи однобічно зв'язаний список.

c) Створити двобічно зв'язаний список, вивести його. Якщо в двобічно зв'язаному списку є шуканий елемент, то вилучити його.

Варіанти індивідуальних завдань

1. Видалити зі списку (масиву) всі елементи, які знаходяться перед максимальним.
2. Видалити зі списку (масиву) всі нульові елементи.
3. Видалити зі списку (масиву) елементи з К-ого по К N-й.
4. Видалити зі списку (масиву) мінімальний елемент та два його сусіди.
5. Видалити зі списку (масиву) три елементи, які знаходяться після нульового елемента.
6. Всі нульові елементи розмістити в правій частині списку або масиву.
7. Вставити посередині списку (масиву) чотири нулі.
8. Додати перед кожним парним елементом в списку (масиву) елемент зі значенням 0.
9. Додати після кожного від'ємного елемента в списку (масиву) елемент зі значенням 0.
10. Дописати в список (масив) після максимального елементу два елементи, рівні N.
11. Знайти в списку (масиву) найбільшу впорядковану у зростаючому порядку послідовність.
12. Знайти в списку (масиву) найбільшу впорядковану у спадаючому порядку послідовність
13. Знайти в списку (масиву) найбільшу послідовність непарних чисел
14. Знайти в списку (масиву) найбільшу послідовність парних чисел.
15. Із списку (масиву) вилучити n елементів. З них створити інший список (масив).
16. Перевірити, чи є в списку (масиві) послідовність з трьох парних елементів.
17. Перевірити, чи є в списку (масиві) послідовність з п'яти впорядкованих елементів.
18. Перевірити, чи є у списку(масиві) хоч одна послідовність з трьох нулів.
19. Переписати кожні три елементи списку (масиву) у зворотньому порядку.
20. Поміняти кожну пару елементів списку (масиву) місцями (1,2 ↔ 2,1; …)
21. Поміняти у кожній п'ятірці елементів списку (масиву) перший та п'ятий елемент місцями (1,2,3, 4, 5 ⇒ 5,2,3, 4, 1; …).
22. Поміняти у кожній трійці елементів списку (масиву) перший та третій елемент місцями (1,2,3 ⇒ 3,2,1; 5,6,7 ⇒ 7,6,5; …).
23. Якщо перший елемент списку (масиву) додатній, видалити 5 перших елементи масиву, а якщо від'ємний – додати в кінець 3 нових елементи.

5.5. Структура даних стек

Стек - це лінійна структура даних, упорядкований список, колекція однотипних даних, що має власні правила вставки та видалення даних. Правила наступні: дані вставляються за принципом дитячої піраміди - опрацьовується спочатку верхній елемент колекції. Вставляється елемент зверху та видаляється теж зверху. Або - перший зайшов, останній пішов (Last in First out - LIFO).

Структура даних стек використовується як контейнер з функціями вставки даних – Push(x) та видалення даних – Pop(). На розмір стеку вказує «Top». На рисунку 5.11 показано стек, як контейнер та принцип роботи стеку.

Також при роботі зі стеком використовується функція Peek(x) – аналог функції Pop, проте без видалення елементу. Різниця між функціями Pop() та Peek() при роботі зі стеком показана на рисунку 5.22.

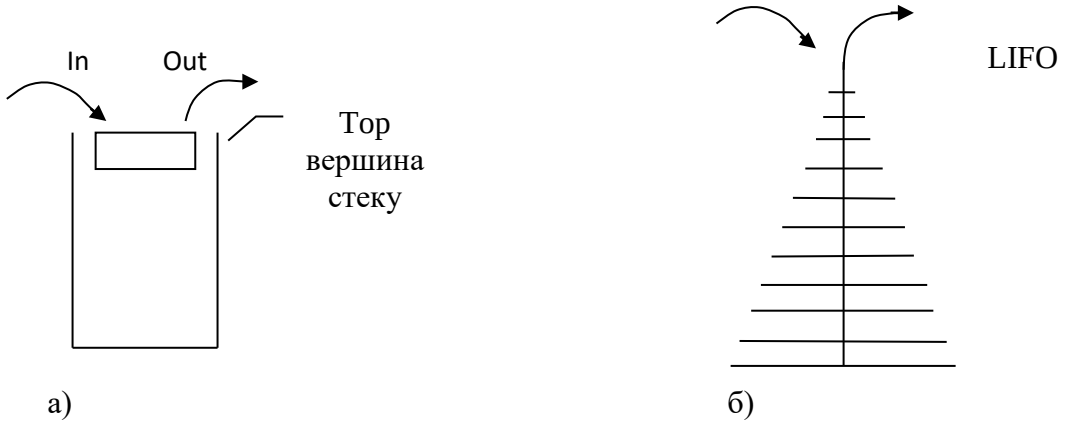


Рисунок 5.22 а)стек, як контейнер, б) принцип роботи стеку

Для контролю стеку використовують дві булеві змінні isEmpty() та isFull(), задачі яких зрозумілі з назви і показані на рисунку 5.23.

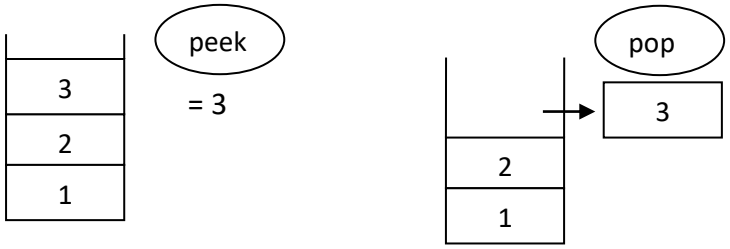


Рисунок 5.23 Відмінності між функціями Pop() та Peek()

Оцінимо часову складність алгоритмів для функцій, що працюють зі стеком. Функції Push і Pop (вставка/видалення елемента) виконують ці операції за сталий час, оскільки працюють з верхнім елементом в стеку, адреса якого відома, тому часова складність алгоритмів для цих функцій O(1).

Якщо в стеку виконується пошук, наприклад з використанням функції Peek, то в найгіршому випадку доведеться перевіряти всі елементи і часова складність алгоритму пошуку складе O(n).

Розглянемо використання структури даних Стек. Нижче наведено кілька прикладів, що пояснюють необхідність саме такої структури даних як стек.

d
c
b
a

1) Завдяки використанню стеку легко виконати реверс. Наприклад, строка «abcd» заноситься в стек і звідти отримуємо «dcba».
2) У текстових редакторах відміна редагування (Ctrl+Z) використовує занесення збережених результатів у стек, а відміна редагування видаляє їх у зворотному порядку.
3) При роботі з рекурсивними функціями виклик функції заноситься у стек і отримується дерево рекурсії.
4) Стек використовується для перевірки парності дужок {{}}.
5) Алгоритм обходу графа в глибину (DFS) базується на використанні стеку.

На рис. 5.24 розглянуто приклад роботи зі стеком з виділенням пам'яті статично на 5 елементів (Size = 5, Top =-1 в початковий момент).

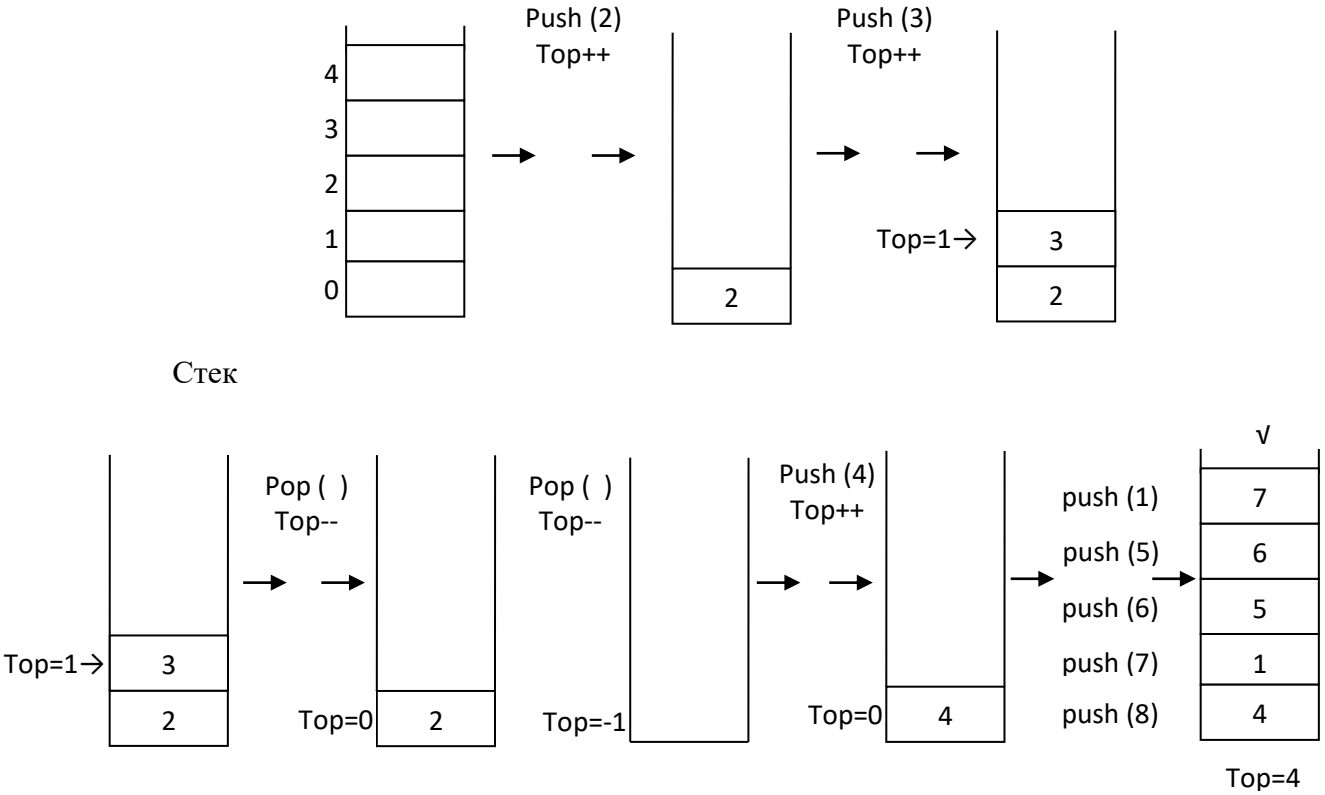


Рисунок 5.24 Робота стеку

Для реалізації стеку використовують два способи – з виділенням пам’яті статично, для цього вказують ємність стеку змінною «Size» та використовують масив.

Другий спосіб реалізувати стек – виділити пам'ять динамічно - для цього використовується однозв’язний список. У цьому випадку ємність стеку може варіюватися.

При спробі виконати функцію push (8) відбувається переповнення стеку (Stack Overflow), оскільки Top = 5 > Size-1 = 4. Тоді булева змінна isFull( ) = true. За аналогією, якщо Top = -1 булева змінна isEmpty = true.

Розглянемо другий спосіб реалізації стеку з використанням однозв’язного списку та динамічним виділенням пам’яті.

На рисунку 5.25 показано стек, реалізований як однозв’язний список, до якого функцією Push() додається новий елемент. Тобто, однозв’язний список реалізує основне правило стеку «останнім зайшов, першим вийшов» (LIFO - Last In First Out).

Відзначимо, що виконання Push() в кінець однозв’язного списку приведе до зростання часової складності до O(n) - пройти всі елементи, щоб вставити останній. Тому цей варіант не розглядаємо.

Як показано на рисунку 5.25, у стеку, що являє собою однозв’язний список, знаходяться дані «5, 8, 7» а функція Push(4) додає новий вузол та змінює значення вказівника на нову адресу пам’яті «50», що показано пунктиром.

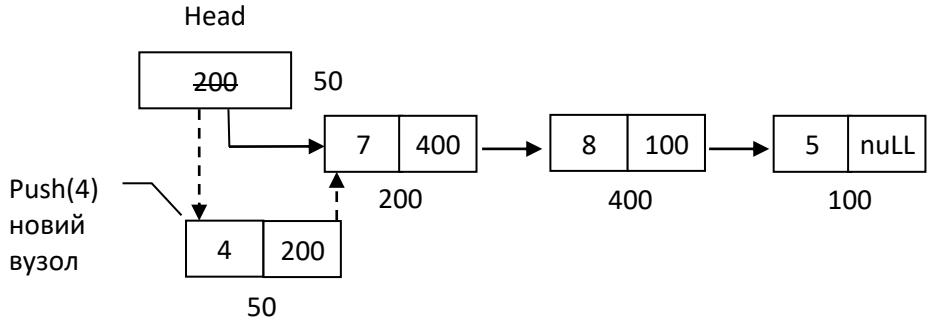


Рисунок 5.25. Реалізація динамічного стеку

На рисунку 5.26 показано стек, що являє собою однозв’язний список з даними «5, 8, 7» та функцію Pop(), яка видаляє верхній вузол зі стеку з даними «7» та змінює значення вказівника Head на нову адресу пам’яті «400», що показано пунктиром.

Надалі вузол з даними «7» залишається в пам’яті до її очистки, проте перестає бути елементом стеку.

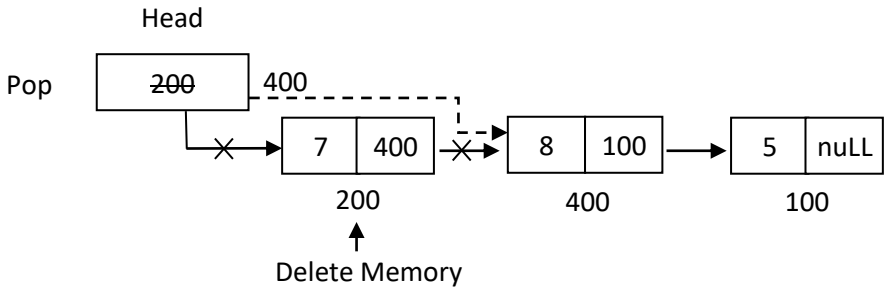
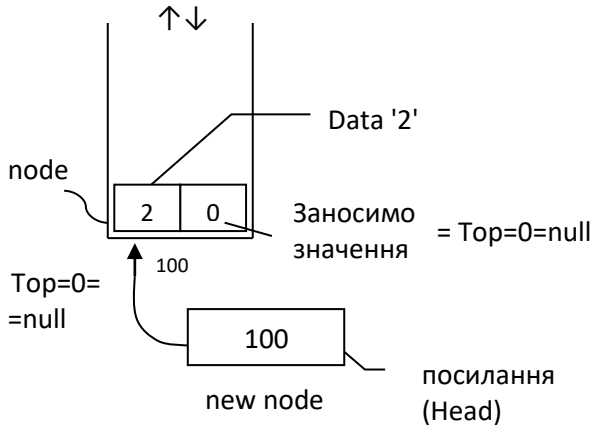


Рисунок 5.26. Виконання функції в динамічному стеку

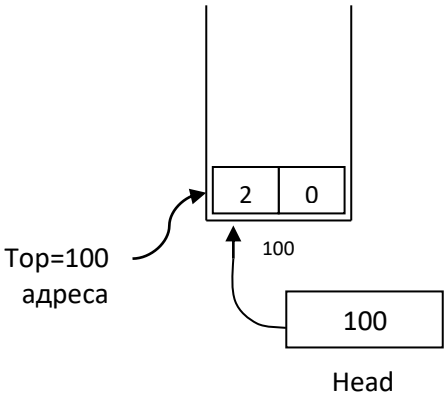
На рисунку 5.27 подано приклад роботи функцій Push та Pop при вставці/видаленні елементів зі стеку, що реалізований як однозв'язний список.

Приклад 1

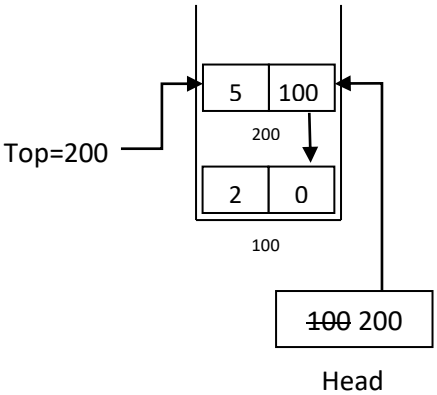
1) Push (2)



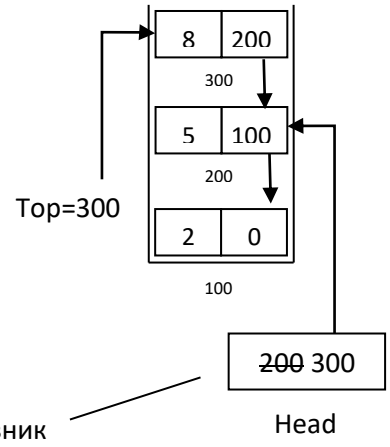
2) результат Push (2)



Push(5)



4)Push (8)



вказівник

Отримали стек, реалізований як список

Рисунок 5.27. Робота функцій в динамічному стеку

Всього стек міститиме три елементи, оформлені як однозв'язний список. Логічна схема стеку показана на рисунку нижче. Вказівник «Top» вказує на адресу в пам'яті (300) верхнього елемента стеку, як показано на рисунку 5.28.

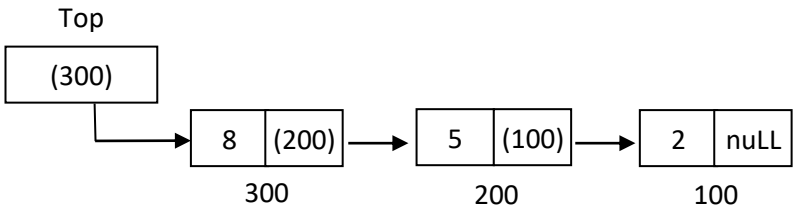


Рисунок 5.28. Робота функцій в динамічному стеку

Наступна функція Push (10) призведе до зміни логічної схеми стеку, відповідно до правила LIFO, як показано на рисунку 5.29.

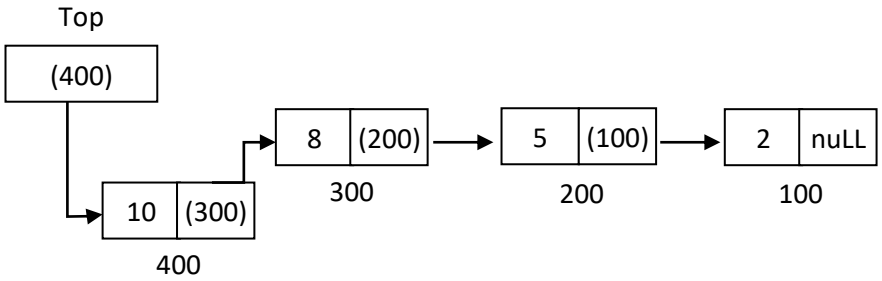


Рисунок 5.29. Вставка в динамічному стеку

Функція Pop() призводить до перепризначення вказівників та зміни значення Top, як показано на рисунку 5.30.

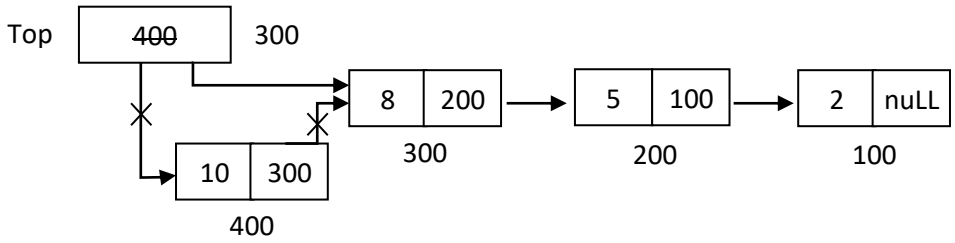


Рисунок 5.30. Робота функцій Pop() в динамічному стеку

5.6. Лабораторна робота. Реалізація стеку

Мета роботи – знайомство зі способами реалізації стеку.

Завдання - реалізувати стек на основі масиву, виконати завдання. Потім реалізувати стек на основі зв'язного списку і теж виконати завдання. Пояснити відмінності реалізації.

1. Якщо верхній елемент стеку додатній, видалити 5 елементів стеку.
2. Створити стек цілих чисел. Переписати з даного стеку всі додатні елементи у другий стек.
3. Створити стек цілих чисел. Обчислити суму максимального та мінімального елементів і перевірити скільки це число ($m = \min + \max$) зустрічається у стеці.
4. Створити стек цілих чисел. Обчислити суму елементів стеку, які менші 10.
5. Створити стек цілих чисел. Обчислити добуток елементів стеку, які менші 5.
6. Створити стек цілих чисел. Знайти що більше: модуль суми від'ємних чи сума додатних.
7. Створити стек цілих чисел. Знайти середнє арифметичне максимального та мінімального елементів стеку.
8. Створити стек цілих чисел. Знайти середнє арифметичне елементів стеку, які належать проміжку від 1 до 6.
9. Створити стек цілих чисел. Знайти середнє арифметичне елементів стеку.
10. Створити стек цілих чисел. Визначити чого більше у стеці: від'ємних чи додатних чисел.
11. Створити стек цілих чисел. Визначити чого більше – парних чи непарних чисел.
12. Створити стек цілих чисел. Видалити мінімальний елемент серед додатніх чисел.
13. Створити стек цілих чисел. Видалити максимальний елемент серед від'ємних чисел.
14. Створити стек цілих чисел і знайти у ньому максимальний та мінімальний елементи, а також номери їх входження у стек.
15. Створити стек цілих чисел – вивести на екран всі парні елементи стеку, та знайти суму непарних.
16. Створити стек символів. Якщо у стеці є великі літери – новий стек заповнити 4 одиницями, якщо ж немає – новий стек заповнити нулями.
17. Створити стек символів. Якщо у стеці більше голосних літер – новий стек заповнити 10 одиницями, якщо ж більше голосних – новий стек заповнити двійками.
18. Створити стек символів. Порахувати чого в стеці більше – голосних, приголосних літер чи інших символів.
19. Створити стек символів. Порахувати чого більше у стеці – літер «а» чи «о».
20. Створити стек символів. Знайти на який позиціях у стеці стоїть літера «а».
21. Створити стек символів. Вивести на екран чого більше – прописних латинських літер чи цифр.

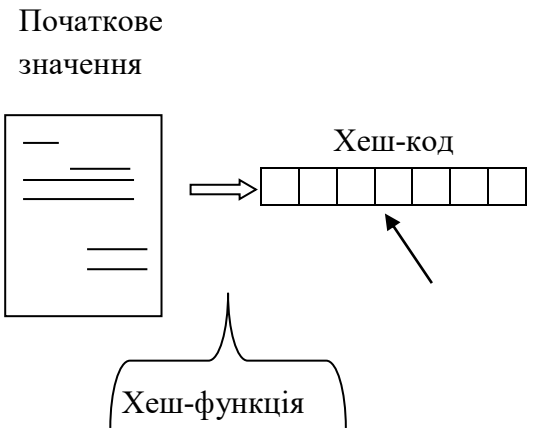
22. Створити стек натуральних чисел. Знайти середнє геометричне максимального та мінімального елементів.

23. Видалити зі стеку всі елементи, які знаходяться перед мінімальним.

6. Хешування даних

Хешування (мішання) процес перетворення даних в певне число, або бітову послідовність [8-9].

Наприклад текст в число до 128 bit, як показано на рисунку 6.1.



Функція, яка перетворює початкове значення в хеш-код фіксованого розміру – Хеш-функція.

Хеш-код – результат роботи хеш-функції.

Рисунок 6.1. Хеш-функція

Хешування будує асоціативні масиви (карти, словники, хеш-мапи), що містять не числа, а хеш-коди.

В структурі даних мови Java інтерфейс Map імплементує Hashtable та Hashmap, як показано на рисунку 6.2. Остання в свою чергу розширює Linked Hashmap.

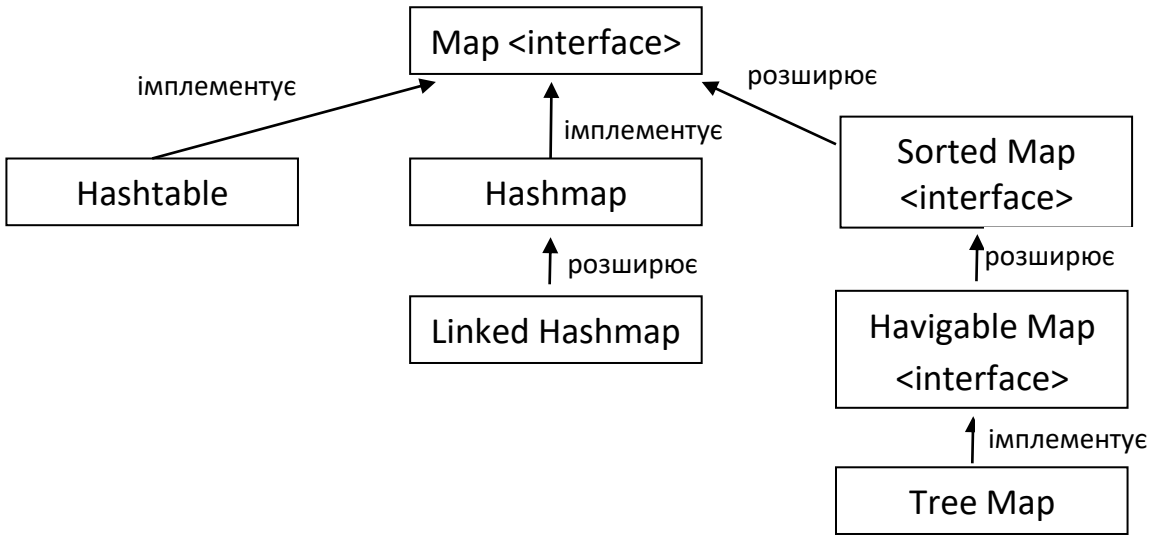


Рисунок 6.2 Інтерфейс Map мови Java

Прикладами хешування даних серед інших є використання хеш-мап (Map):

– для роботи з документами, перекладу тексту,

– пошуку підстроки в строці (Карпа-Рабіна алгоритм)
– для роботи з Базами Даних (прізвище, дата народження заносять в мапу)
– для парсингу даних, баз клієнтів
– у мережах (адреса, пакет, контрольна сума)
– для синхронізації файлів у файлообмінниках (дані, дата)
– у криптографії.

З наведеного вище рисунку видно, що існує кілька підходів до хешування даних. Нижче розглянемо два типи хешування даних.

1. Відкрите хешування (із закритою адресацією).

2.Закрите хешування (із відкритою адресацією).

Пошук за хеш-кодом для хешованих даних має часову складність O(1). Отримання хешованих даних має часову складність O(1) та O(n), для лінійного пошуку та O(log n) - для бінарного пошуку, де n – число ключів.

Оскільки часова складність пошуку прямо залежить від числа ключів, то варто звести початкові значення до меншої кількості ключів n для прискорення пошуку. Однак, при побудові хеш-таблиці за n → 0, можливо, що кілька різних даних отримають ідентичні ключі в Хеш-таблиці $h(k_i) \equiv h(k_j)$. Така ситуація називається колізія даних та є негативною. Розглянемо виникнення колізії даних на наступному прикладі.

Приклад

Припустимо, дано масив K, що містить два значення K =6, 26

Таблиця 6.1. Хеш-таблиця

0	
1	
2	
3	
4	
5	
6	6 26
7	
8	
9	

Задана Хеш-функція $h(K[i]) = K[i]\%m$ будує хеш-таблицю методом ділення, де m =10 – розмір Хеш-таблиці, % - функція, що повертає залишок від ділення.

Маємо K[0]= 6 і хеш функція видає значення запису в Хеш-таблиці K_1 =6%10 = 6.

K_1 = 26 і Хеш-Функція видає значення запису в Хеш-таблиці K_2 = 26%10 = 6.

Це - колізія $k_1 = k_2$, тобто ідентичні записи в Хеш-таблиці відповідають різним початковим даним. Заповнення Хеш-таблиці за умови колізії подано нижче.

В даному прикладі для побудови хеш функції взято метод ділення,

може також використовуватися метод мультиплікації, метод хешування для рядків змінної довжини. В цілому, для хешування даних і уникнення колізії використовують наступні способи.

1 Закрита адресація (закрите хешування) - метод ланцюгів – за кожним ключем створюється однозв'язний список із різними даними.

2 Відкрита адресація (відкрите хешування) використовує три методи:

- лінійне впорядкування даних,
- квадратичне впорядкування даних,
- подвійне хешування даних.

На прикладі нижче розглянемо хешування із закритою адресацією за методом ланцюгів.

Приклад 1

Хешуємо масив із 8 елементів із наступними даними K=3, 2, 9, 6, 11, 13, 7, 12. Використовуємо Хеш-функцію h(K)=2K+3 для Хеш-таблиці, розміром m = 10. Розмір хеш-таблиці m = 10 перевищує розмір масиву початкових даних 8 для зменшення кількості колізій (оскільки в протилежному випадку – якщо розмір Хеш-таблиці менший за розмір початкових даних, то обов'язково виникатимуть колізії). В даному прикладі розглянемо хешування методом ділення та закриту адресацію для зберігання хешованих даних.

хеш=h(K_i)%m=(2K_i+3)%10

Таблиця 6.2 Хеш-таблиця, що містить однозв'язні списки

Ключ	
0	
1	9 | null
2	
3	
4	
5	6 | → 11 | null
6	
7	2 | → 7 | → 12 | null
8	
9	3 | → 13 | null

Таблиця 6.3. Хешування початкових даних з використанням Хеш-функції

Дані	Хеш-функція
3	(2*3+3)%10=9
2	(2*2+3)%10=7
9	(9*2+3)%10=1
6	(6*2+3)%10=5
11	(11*2+3)%10=5 - колізія
13	(13*2+3)%10=9 - колізія
7	(7*2+3)%10=7 - колізія
12	(12*2+3)%10=7 - колізія

Із хеш-таблиці 6.2 видно, що для вирішення проблеми колізії за методом ланцюгів будується ряд однозв'язних списків. Тобто, кожному ключу (хеш-коду) відповідає свій однозв'язний список, що містить початкові дані.

6.1. Відкрите хешування з лінійним впорядкуванням елементів

У випадку хешування із відкритою адресацією використовується лінійне впорядкування даних, квадратичне впорядкування даних, подвійне хешування даних.

Розглянемо нижче приклад хешування із відкритою адресацією та лінійним впорядкуванням даних.

Приклад 2

Розглянемо той же масив із 8 елементів, що і в попередньому прикладі K=3, 2, 9, 6, 11, 13, 7, 12.

Використовуємо ту ж Хеш-функцію h(K)=2K+3 для Хеш-таблиці, розміром m = 10. Однак, на відміну від попереднього прикладу, розглянемо закрите хешування і лінійне впорядкування даних за ключами. Використовуємо метод ділення $h(K_i)\%m=(2K_i+3)\%10$

Розглянемо наведені нижче таблиці. В таблиці справа подано Хешування початкових даних з використанням Хеш-функції. При цьому кількість спроб – правий стовбець вказує на розміщення даних в Хеш-таблиці. Розміщення даних в Хеш-таблиці показано в таблиці зліва, а опис розміщення подано нижче:

Таблиця 6.4 Хеш-таблиця

Ключ	Дані	
0	13	IV
1	9	V
2	12	VI
3		
4		
5	6	Спроб 2
6	11	
7	2	I
8	7	II
9	3	III

Таблиця 6.5. Хешування початкових даних з використанням Хеш-функції

Дані	Хеш-функція	Спроб/ порядок
3	(2*3)+3)%10=9	1
2	(2*2+3)%10=7	1
9	(2*9+3)%10=1	1
6	(2*6+3)%10=5	1
11	(2*11+3)%10=5	2
13	(2*13+3)%10=9	2
7	(2*7+3)%10=7	2
12	(2*12+3)%10=7	6

Перші 4 елементи з масиву K = 3, 2 , 9, 6 розміщаємо в хеш-таблиці 6.4, відповідно до їх хеш-кодів 9, 7, 1, 5. Колізій не виникає.

Елемент 11 має ключ (хеш-код) 5, однак за хеш-кодом 5 вже розміщено елемент 6 – маємо колізію. Тому елемент 11 розміщується на наступне вільне місце з хеш-кодом 6 (із 2 спроби). Аналогічно розміщуються елементи 13 і 7 при цьому, оскільки хеш-код 9 вже

був зайнятий, то елемент 13 розміщується на наступному вільному місці з хеш-кодом 0 (із 2 спроби).

Тобто, вставка K_i за ключем в хеш-таблицю на першу вільну позицію за формулою (h (K_i) + i)%m, де i=0… (m-1) – спроба (i-1).

Зокрема, для K_i =12, та для i=0: (27+0)%=10=7 – колізія, i=1: (27+1)%10=8 – колізія, та для i=5: (27+5)%10=2. Тому за хеш-кодом 2 знаходиться K_i = 12

В результаті, хеш-таблиця з лінійним впорядкуванням елементів має вигляд 13, 9, 12, -, -, 6, 11, 2, 7, 3.

Однак, лінійне впорядкування елементів окрім хеш-коду потребує також фіксування кількості спроб, що потребує додаткової пам'яті. Для усунення цього недоліку, розглянемо наступний спосіб хешування - метод ділення та квадратичного впорядкування для зберігання даних.

6.2. Відкрите хешування з квадратичним впорядкуванням елементів

Розглянемо той же масив із 8 елементів, що і в попередньому прикладі K=3, 2, 9, 6, 11, 13, 7, 12.

Використовуємо ту ж Хеш-функцію h(K)=2K+3 для Хеш-таблиці, розміром m = 10. Однак, на відміну від попереднього прикладу розглянемо закрите хешування методом ділення і квадратичне впорядкування хешованих даних у випадку колізії. У випадку колізії виконується вставка K_i за ключем в хеш-таблицю на першу вільну позицію, відповідно до формули:

$$(h(K_i)+i^2) \% m, \qquad (6.1)$$

де i=0… (m-1) – спроба (i-1), h (K_i) – хеш-функція.

Таблиця 6.6. Хеш-таблиця

Ключ	Дані
0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

Таблиця 6.7. Хешування початкових даних з використанням хеш-функції

Дані	Хеш-функція	Спроб/порядок
3	(2*3)+3)%10=9	1
2	(2*2+3)%10=7	1
9	(2*9+3)%10=1	1
6	(2*6+3)%10=5	1
11	(2*11+3)%10=5	2
13	(2*13+3)%10=9	2
7	(2*7+3)%10=7	2
12	(2*12+3)%10=7	5

Розглянемо наведені таблиці 6.6-6.7. В таблиці справа подано хешування початкових даних з використанням хеш-функції. При цьому кількість спроб – правий стовбець вказує на розміщення хешованих даних в хеш-таблиці. Розміщення хешованих даних в хеш-таблиці показано в таблиці 6.6.

Як видно з таблиці 6.7, що містить хешування початкових даних, для елементів К =3,2, 9, 6 – колізії не виникає і вони розміщуються в Хеш-таблиці, відповідно до ключів.

Для елемента 11 за формулою (6.1) та спроби i=1 маємо хеш-код $(h(k_i) + 0^2)$ % 10 = 5. Це призводить до колізії, тому перераховуємо для спроби i=2:

$(h(k_i) + 1^2)$ %10= 6 – колізії немає, елемент 11 розміщується за даним хеш-кодом в Хеш-таблиці.

Аналогічно для елемента 13 та спроби i=1отримуємо хеш-код 9 – колізія, перераховуємо для i=2: $(h(k_i)+1^2)$%10=(9+1)%10= 0 – колізії немає, елемент 13 розміщується за даним хеш-кодом в Хеш-таблиці.

Для елемента 7 та спроби i=1: $(h(7)+0^2)$ % 10=7 – колізія, перераховуємо для i=2:

$(h(7)+1^2)$%10=8 – колізії немає, елемент 7 розміщується за даним хеш-кодом в Хеш-таблиці.

Далі за аналогією для елемента 12 та спроби i=1 $(h(12)+0^2)$%10=7 колізія,

i=2 $(h(12)+1^2)$%10=8 колізія,

i=3 $(h(12)+2^2)$%10=1 колізія,

i=4 $(h(12)+3^2)$%10=6 колізія,

i=5 $(h(12)+4^2)$%10=3 – колізії немає, елемент 12 розміщується за даним хеш-кодом в Хеш-таблиці.

Маємо квадратичне впорядкування елементів у Хеш-таблиці: 13,9,-, 12, -, 6, 11, 2, 7, 3.

Тобто, кількість колізій при використанні квадратичного впорядкування є меншою, ніж при лінійному, проте – все ще є значною.

Для зменшення кількості колізій використовується також подвійне хешування методом ділення із відкритою адресацією.

6.3. Відкрите Подвійне хешування

Розглянемо той же масив початкових даних із 8 елементів, що і в попередньому прикладі K=3, 2, 9, 6, 11, 13, 7, 12.

Використовуємо ту ж Хеш-функцію h(K)=2K+3 для Хеш-таблиці, розміром m = 10. Однак, на відміну від попереднього прикладу розглянемо подвійне хешування. У випадку

колізії при подвійному хешуванні вставка k_i за ключем в хеш-таблицю на першу вільну позицію виконується за формулою

$$(h_1(k_i) + h_2(k_i) * i) \% m, \qquad (6.2)$$

де m – розмір хеш-таблиці, i=0…(m-1) – спроба (i-1), h_1, h_2 - Хеш-функції

Хеш-функція $h_1(К) = 2*К + 3$

наступна Хеш-функція $h_2(К) = 3*К + 1$

Хешування початкових даних з використанням хеш-функцій h_1, h_2 (на кількість спроб вказує правий стовбець) подано нижче в таблиці 6.8.

Таблиця 6.8. Хешування початкових даних

Дані	h1	h2	Спроба
3	(2*3+3)%10=9	-	1
2	(2*2+3)%10=7	-	1
9	(9*2+3)%10=1	-	1
6	(6*2+3)%10=5	-	1
11	(11*2+3)%10=5	(11*3+1)%10=4	3
13	(13*2+3)%10=9	(13*3+1)%10=0	X
7	(7*2+3)%10=7	(7*3+1)%10=2	X
12	(12*2+3)%10=7	(12*3+1)%10=7	2

Як видно з таблиці 6.8, для елементів К =3, 2, 9, 6 – колізії не виникає і елементи розміщуються в Хеш-таблиці, відповідно до Ключів (Хеш-кодів).

Для елемента 11 за формулою (6.2) та спроби i=0 маємо хеш-код: [(11*2+3)+(11*3+1)*0]%10=5 – колізія.

i=1: [(11*2+3)+(11*3+1)*1]%10=69%10= 9 – колізія

i=2: [(11*2+3)+(11*3+1)*2]%10=(25+69)%10=3 – колізії немає, елемент 11 розміщується за хеш-кодом 3 в хеш-таблиці.

Для елемента 13 за формулою (6.2):

i=0
i=1
i=∞

} [(13*2+3)+(13*3+1)*i]%10=9 – колізія для всіх значень i.

Тому елемент 13 не розміщуємо в хеш-таблиці. В цілому, щоб уникнути таких випадків за подвійного хешування слід обирати хеш-функцію правильно.

Аналогічна ситуація з елементом 7 – колізія для всіх значень i=0…(m-1)=0…9. Тому елемент 7 також не розміщуємо в хеш-таблиці.

i=0: [(7*2+3)+(7*3+1)*0]%10=7 –колізія

i=1: [7*2+3)+(7*3+1)*1]%10=9 –колізія

i=2: [(7*2+3)+(7*3+1)*2]%10=1 –колізія

i=3: [(7*2+3)+(7*3+1)*3]%10=3 –колізія

i=4: [(7*2+3)+(7*3+1)*4]%10=5 –колізія

i=5: [(7*2+3)+(7*3+1)*5]%10=7 –колізія

i=6: [(7*2+3)+(7*3+1)*6]%10=9 –колізія

i=7: [(7*2+3)+(7*3+1)*7]%10=1 –колізія

i=8: [(7*2+3)+(7*3+1)*8]%10=3 –колізія

i=9: [(7*2+3)+(7*3+1)*9]%10=5 –колізія

Для елемента 12 за формулою (6.2):

i=0: [(12*2+3)+(12*3+1)*0]%10=7 – колізія

i=1: [(12*2+3)+(12*3+1)*1]%10=4 – колізії немає, елемент 12 розміщується за хеш-кодом 4 в хеш-таблиці.

Загальний вид хеш-таблиці з подвійним хешуванням елементів подано нижче. В цілому, кількість колізій за подвійного хешування менша, ніж у попередніх випадках, проте два елементи не потрапили в хеш-таблицю 6.9.

Таблиця 6.9. Хеш-таблиця

Ключ (хеш-код)	Дані
0	
1	9
2	
3	11
4	
5	6
6	
7	2
8	
9	3

6.4. Лабораторна робота. Хешування даних

Виконати завдання, відповідно до варіанту та перевірити роботу при колізіях. Хеш-таблицю реалізувати як динамічний масив (якщо не вказано - хеш функцію та розмір таблиці обрати самостійно).

1. Організувати хеш-таблицю з відкритою адресацією, використовуючи хеш-функцію h (k) = trunc (M * Frac (k * d)), де d = (sqrt (5) -1) / 2, M - розмір хеш-таблиці. Організувати метод пошуку по ключу в цій хеш-таблиці. Результат пошуку - номер комірки зі знайденим ключем або (-1).

2. Організувати хеш-таблицю з відкритою адресацією, використовуючи метод пошуку і вставки по ключу. Для формування хеш-адреси використовувати хеш-функцію отриману методом множення і процедуру лінійного впорядкування для вирішення колізії.

3. Організувати хеш-таблицю з відкритою адресацією, використовуючи процедуру пошуку і вставки по ключу. Для формування початкового хеш-адреси використовувати метод ділення, а при виникненні колізії метод квадратичного впорядкування.

4. Побудувати хеш-таблицю з відкритою адресацією, використовуючи подвійне хешування, як спосіб відкритої адресації. Хеш-функції h1 (k) і h2 (k) організувати методом ділення. Формування таблиці - за допомогою методів пошуку і вставки за ключем.

5. Організувати хеш-таблицю, використовуючи хеш-функцію h (k) за методом множення для формування хеш-адреси, вирішення колізій - методом ланцюгів. а) Написати пошук і вставку за ключем. б) Написати метод видалення по ключу.

6. Організувати хеш-таблицю за допомогою методу множення - для вирішення колізій використати лінійне впорядкування. Запрограмувати процедуру пошуку і вставки за ключем.

7. Організувати програмно хешування 100 записів (вузлів) в таблицю, що складається з 20 посилань на однозв'язні лінійні списки (стеки), спочатку порожні. Записи мають невід'ємні цілі ключі k <50, що формуються випадковим чином. Хеш-функцію організувати шляхом множення як окремий метод.

8. Організувати програмно хешування 50 записів в таблицю. що складається з 10 посилань на однозв'язні лінійні списки (черги), спочатку порожні. записи мають невід'ємні цілі ключі k <30, що формуються випадковим чином. Хеш-функцію організувати методом ділення.

9. Організувати хеш-таблицю за допомогою методу ділення - для вирішення колізій використати метод ланцюгів. Реалізувати пошук і вставки по ключу.

10. Організувати хеш-таблицю за допомогою методу ділення - для вирішення колізій використати метод подвійного хешування. Реалізувати вставку і видання по ключу.

6.5. Алгоритми з використанням хешування. Алгоритм Карпа-Рабіна

Алгоритм Карпа-Рабіна перевіряє входження підстроки у строку з використанням хешування.

Припустимо, дано строку (в загальному випадку – текст) та підстроку, як показано на рисунку 6.3.

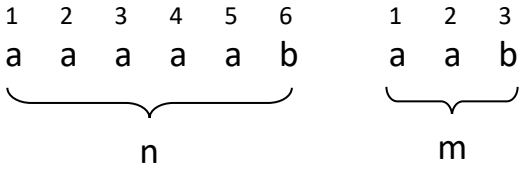


Рисунок 6.3. Строка та підстрока

Задача - перевірити належність підстроки тексту.

Розглянемо алгоритм Карпа-Рабіна, що використовує хешування даних: встановимо принципи роботи алгоритму, оцінимо його часову складність, покажемо способи покращення роботи алгоритму.

Розглянемо принципи роботи даного алгоритму.

Найпростіший підхід перевірити чи містить строка підстроку – взяти текст і послідовно перевірити чи є там підстрока. Псевдокод примітивного пошуку та оцінка його часової складності наведені нижче.

Naive Search

```
(n-m+1) –      for (string [i] i=0 to (n-m+1) i++) {
(m-1)          –        for (substring [j] j=0 to m j++){
1              –          if (string [i] == substring [j]) & (j=3)
  then
1              –                  Print (Exist substring)
    }
  }
```

У вищенаведеному алгоритмі застосовано метод грубої сили – перебір всіх варіантів. Тут n – довжина строки, m – довжина підстроки. Часова складність алгоритму

$(n-m+1)\cdot(m-1)+1+1) = (n-m+1)(m+1) = n\cdot m-m^2+m+n-m+1$

Якщо n>>>m, то часова складність O(n·m)

Розглянемо як працює алгоритм Карпа-Рабіна і порівняємо його часову складність з вищенаведеним алгоритмом з примітивним пошуком.

Для початку, в стороці та підстроці на основі ASCII кодів, наприклад, конвертуємо букви в цифри.

Приклад 1. Припустимо, для прикладу, що

a – 1 (ASCII код 97), b – 2, c – 3, d – 4, e – 5, f – 6, g – 7, h – 8, i – 9, j – 10.

Для підстроки aab, маємо m = 3 – розмір підстроки та її код 112.

Припустимо, хеш-функція рахується як сума кодів h(p):1+1+2=4. Тобто, хеш-код підстроки 4.

Використаємо хеш-код підстроки, щоб знайти її в строці

$\underbrace{a\ a\ a}_{m}\ a\ a\ b,$ n=6 – розмір тексту

Рахуємо хеш-код , виділяючи в строці підстроку, довжиною m = 3 h(p)=1+1+1=3.

Оскільки хеш-код відрізняється від шуканого 3≠4, переходимо до наступної підстроки:

a a a a a b: h(p)=1+1+1=3, 3≠4, переходимо до наступної підстроки

a a a a a b: h(p)=1+1+1=3, 3≠4, переходимо до наступної підстроки

a a a a a b: h(p)=1+1+2=4, 4=4. Хеш-коди тотожні.

Надалі, методом грубої сили, підстрока перевіряється посимвольно a≡a, a≡a, b≡b. В цьому - суть алгоритму Карпа-Рабіна.

Однак, часова складність наведеного вище прикладу алгоритму Карпа-Рабіна в найгіршому випадку відповідає часовій складності алгоритму за методом грубої сили. Оскільки в алгоритмі Карпа-Рабіна (n-m)*n разів рахується хеш-функція, то домінуюча функція визначає часову складність O(n·m).

Для зменшення часової складності в алгоритмі Карпа-Рабіна використано плаваючий хеш (роллін хеш або ще - кільцевий хеш), який не потрібно кожного разу знову рахувати. Принцип роботи алгоритму Карпа-Рабіна з плаваючим хешем розглянуто нижче на наступному прикладі.

Приклад 2. Припустимо задана строка, довжиною n=8 символів:

a b c d a b c e.

Перевірити чи належить їй підстрока, довжиною m = 3 символи b c e

На основі прийнятих у прикладі 1 позначень, визначимо хеш-код підстроки 2+3+5=10.

Рахуємо хеш-код , виділяючи в строці підстроку, довжиною m = 3.

a b c d a b c e h(p)=1+2+3=6 ≠ 10

Оскільки хеш-код відрізняється від шуканого 6 ≠ 10, переходимо до наступної підстроки:

a b c d a b c e

Однак, при цьому хеш-функція перераховується на основі ковзаючого хешу – віднімається значення першого символу та додається значення наступного. Часова складність такої операції O(1), оскільки потрібен постійний час для перерахунку ковзаючого хешу h(p).

h(p) - 1+4 = (1+2+3)-1+4=2+3+4=9≠10

Оскільки хеш-код відрізняється від шуканого 9 ≠ 10, переходимо до наступної підстроки з використанням ковзаючого хешу.

a b c d a b c e h(p) – 2+1 = 8≠10

a b c d a b c e h(p) – 3+2 = 7≠10

a b c d a b c e h(p) – 4+3 = 6≠10

a b c d a b c e h(p) – 1+5 =10=10 – Оскільки хеші співпадають, можливо в тексті є підстрока.

Надалі, методом грубої сили, підстрока перевіряється посимвольно b≡b, c≡c, e≡e.

Тобто, алгоритм Карпа-Рабіна полягає в наступному:

1. Обчислити h(p) для підстроки.

2. Обчислити h(p) для всіх підстрок тексту.

3. Якщо h(p) підстроки і h(p) тексту рівні, то виконати перевірку посимвольно, методом грубої сили.

Псевдокод описаного вище алгоритму Карпа-Рабіна:

```
1       Function Karp Rabin (string S[1...n], string pattern [1...m])
2 m – { hpattern = hash (pattern [1...m]);              // хеш підстроки
3 n       { for i from 0 to n-m+1;                              // хеш строки
4 m –           hstring = hash (s[i...i+m-1]);
5 1               If hstring == hpattert;               // їх порівняння
```

```
6 1                 If s[i...i+m-1] == pattern [1...m];      //уточненння
7               Return i }
8  Return 'Not found' }
```

Кількість операцій для строк 3 і 4 = $\underbrace{\text{m, m, m, m ...}}_{\text{n-m+1}}$ Звідси часова складність O(n·m).

Але, якщо взяти роллінг хеш – кільцевий хеш, то не потрібно перераховувати хеш у строці 4 кожного разу спочатку. Для роллінг хешу хеш-функція має наступний вигляд:

$$S[i+1...i+m] = S[i...(i+m-1)] - S[i]+S[i+m]$$

Ця формула має постійну часову складність. Тому у строках 3 і 4 псевдокоду, роллінг хеш буде обраховано (n-m+1) разів:

$$\underbrace{\text{2, 2, 2,...}}_{\text{n-m+1}}$$

Звідси часова складність алгоритму Карпа-Рабіна з ролінг хешем складає O(n-m+1) у найгіршому випадку. Або O(n), якщо n>> m, або якщо n = m, то маємо O(1).

Тобто, використання ролінг хешу зменшує часову складність алгоритму Карпа-Рабіна.

При пошуку підстроки в строці за алгоритмом Карпа-Рабіна може бути випадок хибної тотожності хешів. Даний випадок подано в прикладі нижче.

Приклад 3 Припустимо задана строка, довжиною n=11 символів:

c c a c c a a e d b a

Перевірити чи належить їй підстрока, довжиною m = 3 символи d b a.

На основі прийнятих у прикладі 1 позначень, визначимо хеш-код підстроки h(p) = 4+2+1 = 7.

Рахуємо хеш-код , виділяючи кожного разу в строці підстроку, довжиною m = 3

I) $\underbrace{\text{c c a}}$ c c a a e d b a h(p)=3+3+1=7≡7 – хешкод дає true.

Однак перевірка посимвольно false.

II) c $\underbrace{\text{c a c}}$ c a a e d b a h(p)= h(p)-3+3=h(p)=7≡7 – хешкод дає true.

Однак перевірка посимвольно false.

III) c c a $\underbrace{\text{c c a}}$ a e d b a h(p)= h(p)-1+1=h(p)=7≡7– хешкод дає true.

Однак перевірка посимвольно false.

IV) c c a c $\underbrace{\text{c a a}}$ e d b a h(p)= h(p)-3+1=5 не тотожно 7 – хешкод дає true.

V) c c a c c $\underbrace{\text{a a}}$ e d b a h(p)= h(p)-3+5= 7≡7– хешкод дає false.

Однак перевірка посимвольно false.

Виходить із 5 показаних вище перевірок чотири перевірки хеш-коду як значення хеш-функції дали хибний результат. Тобто, хеш-коди – тотожні, але підстрока не є частиною строки.

У такому випадку алгоритмом Карпа-Рабіна стає подібним до алгоритму грубої сили і матиме часову складність O(n·m), і якщо $m = n/2 \Rightarrow O(n^2)$.

Таким чином, виникає задача уникнути хибних результатів роботи алгоритму Карпа-Рабіна, що зменшить його часову складність.

Для вирішення цієї проблеми хеш-функція, отримана методом ділення замінюється складнішою хеш-функцією, побудованою за методом множення. Наприклад:

$$h(p) = 10^{m-1} \cdot p[1] + 10^{m-2} \cdot p[2] + 10^{m-3} \cdot p[3]$$

Повертаючись до останнього прикладу, хеш-код для підстроки d b a:

$h(p) = 10^{3-1} \cdot 4 + 10^{3-2} \cdot 2 + 10^{3-3} \cdot 1 = 400 + 20 + 1 = 421$.

Обрано 10^{m-1}, 10^{m-2} оскільки хеш-таблиця містить 10 строк- від 1 до 10, від a до j.

Якщо взяти латиницю, то що хеш-таблиця матиме 26 строк - від 1 до 26 і хеш-функція матиме вигляд:

$h(p) = 26^{m-1} \cdot p[1] + 26^{m-2} \cdot p[2] + \ldots$

m доданків

Якщо взяти нижній реєстр, верхній реєстр - всього 127 символів, то хеш-функція матиме вигляд:

$h(p) = 127^{m-1} \cdot p[1] + 127^{m-2} \cdot p[2] + \ldots$

m доданків

Тоді, для прикладу 3, рахуємо хеш-код на основі хеш-функції, побудованої за методом множення.

I) c c a c c a a e d b a

$h(p) = 10^{m-1} \cdot 3 + 10^{m-2} \cdot 3 + 10^{m-3} \cdot 1 = 100 \cdot 3 + 10 \cdot 3 + 1 \cdot 1 = 331 \neq 421$.

Хеш-коди не тотожні. Далі рахуємо, використовуємо роллінг хеш (круговий хеш, ковзаючу хеш-функцію)

II) c c a c c a ·- h(p)= h(p) – попередній символ + наступний символ

$[[3 \cdot 10^2 + 3 \cdot 10^1 + 1 \cdot 10^0] - 3 \cdot 10^2] \times 10 + 3 \cdot 10^0$ h(p)=313≠421. Хеш-коди не тотожні.

III) c c a c c a $h(p) = [h(p) - 3 \cdot 10^2] \cdot 10 + 3 \cdot 10^0 = 130 + 3 = 133 \neq 421$.

IV) c c a c c a a e d b a $h(p) = [h(p) - 1 \cdot 10^2] 10 + 1 \cdot 10^0 = 331 \neq 421$

V) c c a c c a a e d b a $h(p)= [h(p)-3\cdot10^2]10+1\cdot10^0=311\neq421$

VI) c c a c c a a e d b a $h(p)= [h(p)-3\cdot10^2]\cdot10+5\cdot10^0=115\neq421$

VII) c c a c c a a e d b a $h(p)= [h(p)-1\cdot10^2]\cdot10+4\cdot10^0=154\neq421$

VIII) c c a c c a a e d b a $h(p)= [h(p)-1\cdot10^2]\cdot10+2\cdot10^0=542\neq421$

IX) c c a c c a a e d b a $h(p)= [h(p)-5\cdot10^2]\cdot10+1\cdot10^0=421=421$ – true

Далі посимвольна перевірка останньої підстроки теж дає true - підстроку в строці знайдено.

Виходячи з наведеного, алгоритм Карпа-Рабіна використовує:

1) хеш-функцію на основі множення для уникнення колізій,

2) роллінг хеш (ковзаючу хеш-функцію, кругову хеш-функцію) для зменшення часової складності до O(n), якщо n>> m.

6.6. Алгоритми хешування. Ймовірнісна структура даних, фільтр Блума.

На основі методів хешування даних використовують ймовірнісні структури даних. Як приклад ймовірнісної структури даних розглянемо фільтр Блума. Задачею, що вирішує така структура даних є відповідь - належить елемент множині, чи навпаки – не належить. Тому, на першому етапі, елементи хешуються і заносяться в хеш-таблицю (множину), яка є основою фільтра Блума.

На другому етапі, за потреби, виконується перевірка чи є шуканий елемент в множині (фільтрування елементів) на основі перевірки існування хешу в хеш-таблиці занесених елементів.

На прикладі нижче розберемо принципи роботи алгоритму.

Приклад 1

Припустимо: 1) розмір фільтра Блума заданий і рівний n=5,

2) дві хеш-функції визначено $h_1(x)=x\%5$ та $h_2(x)=(2x+3)\%5$

На першому етапі занесемо два елементи у фільтр Блума. Перший елемент x = 9, для цього виконаємо його хешування. Маємо $h_1(9)=9\%5=4$, $h_2(9)=(2\cdot9+3)\%5=1$

Далі на на місця, рівні h_1, h_2 поставимо 1, інші залишимо 0. Фільтр Блума має вигляд, показаний на рисунку 6.4.

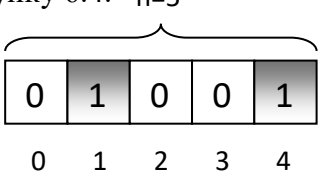


Рисунок 6.4. Фільтр Блума

Занесемо наступний елемент x = 11 в множину. Маємо $h_1(11)=11\%5=1$, $h_2(11)=(2\cdot11+3)\%5=0$. Далі на на місця, рівні h_1, h_2 поставимо 1, інші залишимо 0. Фільтр Блума отримає вигляд, показаний на рисунку 6.5.

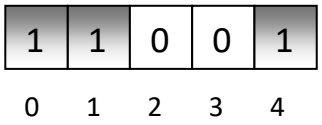


Рисунок 6.5. Фільтр Блума по завершенню

На другому етапі перевіримо, чи належить множині даний елемент, чи ні. Наприклад, перевіримо, чи входить в множину елемент 15, тобто чи є 15 одним із доданих елементів. Для цього обрахуємо його хеш-коди і проаналізуємо їх:

$h_1(15)=15\%5=0$,

$h_2(15)=(2\cdot15+3)\%5=3$

Якби елемент 15 входив у множину, то на місці № 0 і № 3 розміщені були б ‘1’. Проте, на місці № 3 розміщено ‘0’. Тому елемент 15 не входить в множину в жодному разі. Така відповідь носить назву - False Positive.

Перевіримо, чи входить в множину елемент 16, обрахуємо його хеш-коди і проаналізуємо їх:

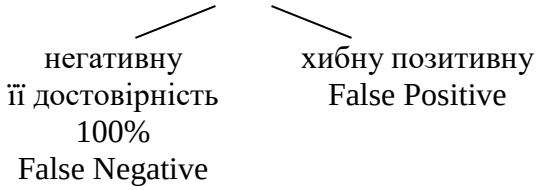
$h_1(16)=16\%5=1$,

$h_2(16)=(2\cdot16+3)\%5=0$

На місці № 0 і № 1 у фільтрі Блума стоїть ‘1’, тому можна припустити, що 16 вже було додано до множини.

Однак, як ми знаємо, насправді елемент 16 не додавався в множину. Тому, виходить фільтр Блума не може дати однозначну позитивну відповідь про входження елемента. Тобто, можливо елемент 16 є в множині. Така відповідь носить назву хибна позитивна відповідь - False Positive.

Тобто, в цілому фільтр Блума на питання про належність елемента множині дає одну з двох відповідей:



Саме тому така структура даних називається – ймовірнісна структура.

Обрахуємо ймовірність того, що замість числа, яке є в множині отримали відповідь, що число можливо є в множині (False Positive).

Розглянемо фільтр Блума, що має вмістити:

m – число елементів,

k – число хеш-функцій.

Тоді mk - кількість одиничок у фільтрі Блума. Оскільки за ймовірністю на 50% нуликів має бути 50% одиничок, то розмір подвоюємо і отримуємо n = 2mk розмір фільтра Блума (або 2mk-1, якщо починати з нуля), як показано на рисунку 6.6.

Рисунок 6.5. Фільтр Блума

Для оцінки верхньої межі похибки отримання відповіді, що число можливо є в множині (False Positive) використовується наступна теорема. Розглянемо її без доведення.

Теорема:

Якщо h_1 і h_2 ..... h_k – незалежні хеш-функції, а у фільтр Блума заноситься довільне число елементів x_1, ... , x_m то ймовірність того, що $y \notin \{ x_1, ... , x_m\}$ яка додається у фільтр Блума, поверхне 1 (тобто False Positive) дорівнює:

$$\Pr\{\text{test (y)} = 1\} \leq \frac{1}{2^k} = (\frac{1}{\sqrt{2}})^{\frac{n}{m}} \sim 0{,}7^{\frac{n}{m}}$$

Тобто, за теоремою оцінюється верхня межа.

Розглянемо приклад використання теореми. Розрахуємо параметри фільтра Блума для занесення m = 100 000 слів словника та пошуку слова в ньому.

Якщо k=10 і h_1, h_2 ..... h_k – хеш-функції, то, на основі теореми, верхня межа хибного визначення елемента в множині:

$$\Pr\{\text{test (y)} = 1\} \leq \frac{1}{2^k} = \left(\frac{1}{2^{10}}\right) = \frac{1}{1024} = 0{,}001, \text{або } 0{,}1\%$$

Загальний розмір фільтра Блума

$n=2mk-1= 2\cdot10^5\cdot10=2\cdot10^6$=2Mbit, або /8=250 KB

Тобто, 250 КВ пам’яті потрібно словнику зі 100 000 слів для відстежування наявності слова, якщо верхня межа похибки відповіді (False Positive) не більша, ніж 0,1%.

Нижче наведено псевдокоди для операцій з фільтром Блума.

Додавання елементів до фільтру з їх хешуванням, де h – хеш-функція:

```
Add (x)
for (i=1 to k){
   id = h[x];
   T [id]=1;
      }
```

Перевірка наявності елемента у фільтрі. Перший for працює з фільтром Блума, другий – виконує поелементну перевірку для False Positive:

```
Test (x)
for (i=1 to k) {
    id = h[x];
    if T [id]==B[id]
      then Rez=False Positive
      else Rez=False Negative
    }
    for (j=1 to x.length)
    { if x[j]==string
        then Print ('Find');
      }
```

Операція видалення елемента з фільтра Блума є не очевидною і полягає у створенні нового фільтра Блума, що містить елементи, які вважаються видалені з першого фільтра. Тобто, перший фільтр Блума - для додавання елементів (BF1-Add) та перевірки наявності, а другий для перевірки чи не видалено даний елемент (BF2-Add).

7. Дерева, як структури даних

Спробуємо узагальнити відомі структури даних у вигляді наступної схеми (рисунок 7.1), що складається з лінійних та не лінійних структур. В свою чергу лінійні структури можна розділити на статичні та динамічні, розмір пам'яті для яких може змінюватися. З рядом лінійних структур ознайомилися раніше, а далі перейдемо до розгляду не лінійних структур даних.

До не лінійних структур даних відносять Дерева, Графи та Хешовані дерева. Ознакою не лінійності є відсутність способу обходу даних зліва-направо, або справа-наліво [6, 7].

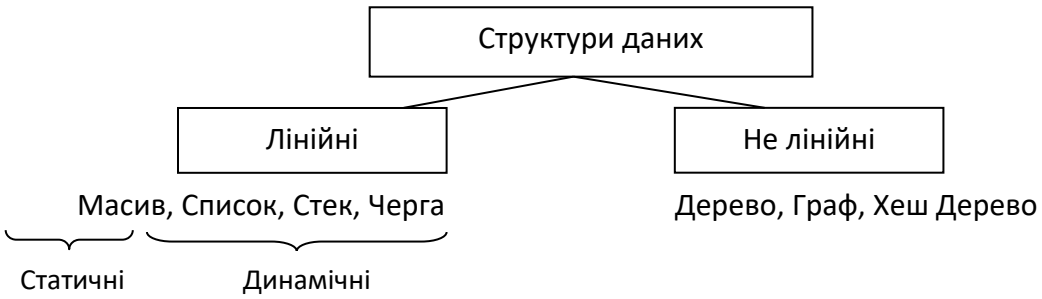


Рисунок 7.1 Структури даних

7.1. Основні теоретичні відомості про дерева

Розглянемо не лінійні структури даних на прикладі дерева. З математики відомо, що дерево – це граф, що не має циклів, де Граф – це об'єкт, що складається з двох множин, множини вершин та множини зв'язків між цими вершинами.

З точки зору даних, будемо розглядати дерево як колекцію примітивів (наприклад - вузлів), зв'язаних у деяку багаторівневу ієрархічну структуру, як показано на рисунку нижче.

Показаний на рисунку 7.2 орієнтований граф є деревом, оскільки в ньому відсутні цикли. Наприклад, як видно з рисунка, існує шлях A→B→E→I, який не є замкненим, оскільки немає шляху I→A.

У дерева на рисунку 7.2 A, B, ... M – вузли дерева; вершина A – корінь, як вершина ієрархії.

Вершина C – батько для вершини G; вершина A - не має батька, вершини B, C, D – діти вершини A.

Вершини I, J, K, L, M – не мають дітей і називаються листи Дерева.

Вершини, які мають дітей – внутрішні вершини Дерева - A, B, C, D, E,G.

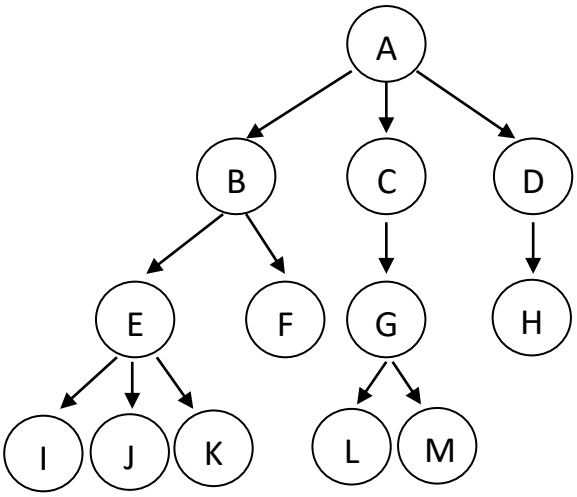


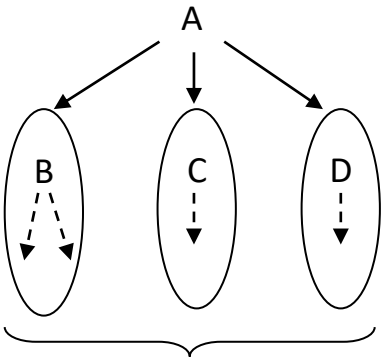
Рисунок 7.2 Орієнтований граф - дерево

Шлях – послідовність відповідних ребер від джерела до вершини призначення. Наприклад : A→B→E→J

Предки – вузли, які лежать вище на шляху. Наприклад: для вершини L предки A, C, G.

Нащадки – вузли, які лежать нижче певної вершини по шляху . Наприклад: для B нащадки E, F, I, J, K.

Піддерево – частина Дерева, що включає всіх нащадків кореневої вершини. Приклад показано на рисунку 7.3.



Три піддерева

Рисунок 7.3 Піддерева

Степінь вершини – кількість дітей у цієї вершини. Наприклад, на рисунку вище маємо: A – 3, B – 2, C – 1. Звідси - степінь листів завжди рівна нулеві.

Степінь дерева - максимальна степінь серед степенів його вершин.

Глибина вершини (рівень вузла) – довжина шляху від кореня до даної вершини (вузла). Наприклад :F – 2; J - 3; A – 0; D – 1.

Висота вершини (вузла) – найдовший шлях від даної вершини до листа. Наприклад: B – 2, C – 2, D – 1, A – 3.

Висота дерева – максимальна висота серед висот його вершин і відповідає висоті кореня дерева.

7.2. Реалізація дерева

Якщо вершини дерева реалізувати як вузол в двозв'язному списку, то дерево можна розглядати як двозв'язний список. Дерево та його логічна схема при реалізації двозв'язним списком зображені на рисунку нижче. Знаком x – помічені null посилання, як показано на рисунку 7.4.

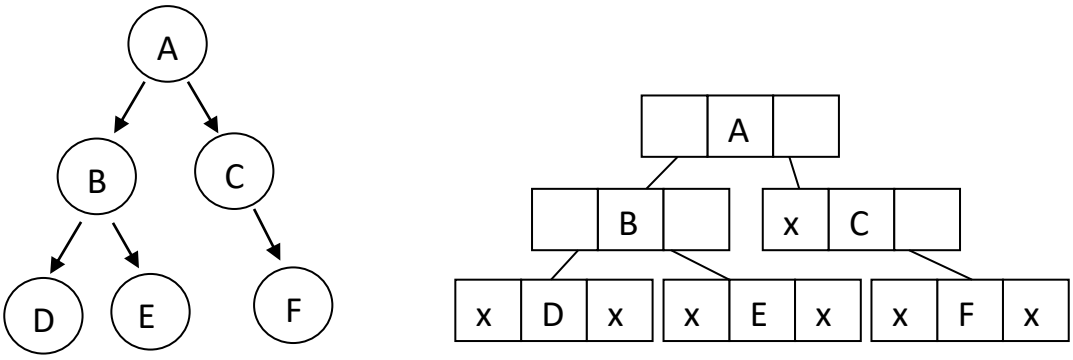


Рисунок 7.4. Реалізація дерева як двозв'язного списку

Псевдокод реалізації вершини (вузла) дерева, як двозв'язного списку з правим та лівим посиланням дано нижче.

```
struct node
{ int data;
  struct node *left;
  struct node *right
  }
```

Як видно з коду, у вигляді двозв'язного списку можна реалізувати саме бінарні дерева, тому розглянемо їх детальніше.

Бінарне дерево – це дерево, кожен вузол якого має максимум двох дітей, тобто: 0, 1, 2. На рисунку 7.5 показано п'ять прикладів бінарних дерев, що відповідають даному визначенню. Зокрема, дерево, що складається лише з одного вузла, називають – вироджене дерево. Дерево, що має лише лівих (правих) нащадків, називають – ланцюг. Однак, всі вони є бінарними деревами за виключенням прикладу справа. Правий приклад не є бінарним деревом, оскільки один з його вузлів має трьох дітей.

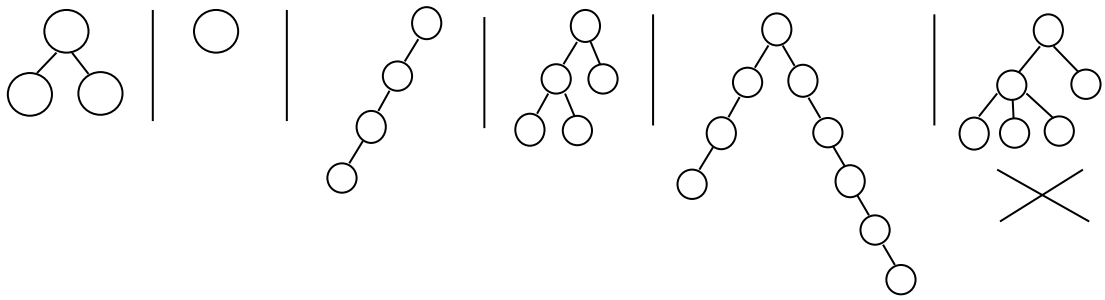


Рисунок 7.5. Приклади бінарних дерев

Приклад реалізації бінарного дерева з використанням двозв’язного списку подано на рисунку 7.6. Знаком x – помічені null посилання.

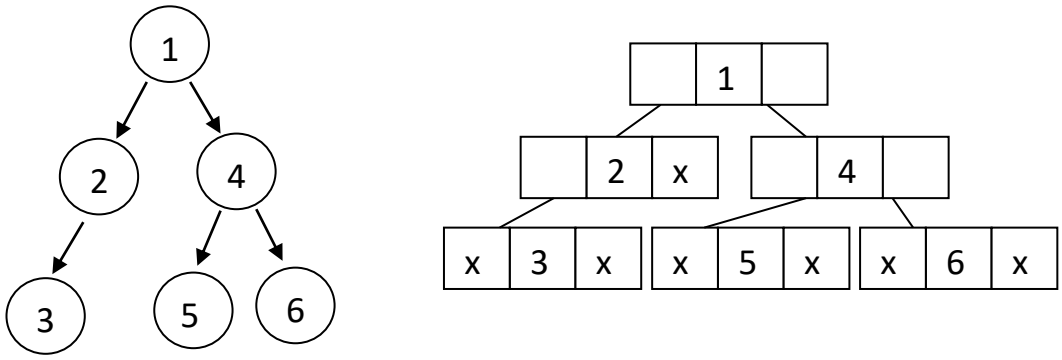
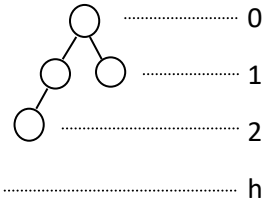


Рисунок 7.6. Реалізація бінарного дерева як двозв’язного списку

Розглянемо властивості бінарних дерев. Підрахуємо максимальну кількість вузлів у бінарного дерева. Розглянемо дерево та наведемо максимальну кількість вузлів на кожному з рівнів дерева, як показано на рисунку 7.7.



	максимальна кількість вузлів
0 рівень	2^0
1 рівень	2^1
2 рівень	2^2
і т.д.	

Рисунок 7.7. Кількість вузлів на кожному з рівнів бінарного дерева

Для рівня *i* максимальна кількість вузлів дерева: 2^i. Тоді максимальна загальна кількість вузлів у бінарного дерева, висотою h:

$$\underbrace{2^0+2^1+2^2+...+2^h}_{\sum},$$

помножимо на 2 і віднімемо ∑:

$$2\sum-\sum=2^{h+1}+2^h+...+2^2+2^1-2^h...-2^2-2^1-2^0=2^{h+1}-2^0=2^{(h+1)}-1.$$

Тобто, максимальна загальна кількість вузлів у Бінарного Дерева *n*, висотою *h* буде:

$$n_{max} = 2^{(h+1)}-1. \quad (7.1)$$

Мінімальна загальна кількість вузлів у Бінарного Дерева буде у випадку, якщо дерево є ланцюгом дорівнює:

$$n_{min} = h+1. \quad (7.2)$$

З формули (7.1) маємо мінімальну висоту дерева, що має *n* вузлів (по верхній межі):

$$h = \left\lceil \log_2(n_{max} + 1) - 1 \right\rceil.$$

З формули (7.2) маємо максимальну висоту дерева, що є ланцюгом та має *n* вузлів:

$$h = n_{min} - 1.$$

7.3. Види бінарних дерев

Розглянемо властивості бінарних дерев різного виду: повного, досконалого та ідеального бінарних дерев.

Повне бінарне дерево, це бінарне дерево (БД) у якому кожна вершина має нуль, або 2 дітей. Іншими словами, у повного бінарного дерева – всі вузли мають двох дітей, крім вузлів, що є листами.

Приклад повного бінарного дерева подано на рисунку 7.8 – зліва. бінарне дерево на рисунку 7.8 справа не є повне, оскільки існує вузол, що має 1 дитину.

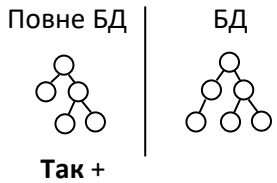


Рисунок 7.8. Повне бінарне дерево

Для повного бінарного дерева справедливо наступне правило - кількість вузлів, які є листами на один більше, ніж внутрішніх вузлів:

$n_{листи} = n_{внутрішні} + 1$

$n = 2h + 1.$

Звідси, максимальна висота дерева, що відповідає кількості рівнів h у повного бінарного дерева:

$h = \frac{n-1}{2}$.

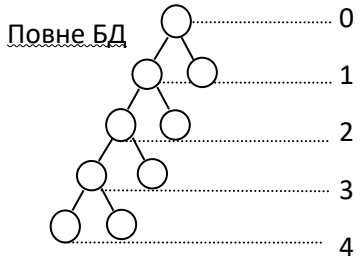


Рисунок 7.9. Повне бінарне дерево

Мінімальна кількість вузлів n у повного бінарного дерева:

Досконале бінарне дерево – таке бінарне дерево, у якого (h-1) рівнів є повними, а рівень h (на якому всі вузли – листи) заповнюється, починаючи зліва. Приклади досконалого бінарного дерева подано на рисунку 7.10, де дерева a), c) є досконалими. Бінарні дерева b), d) на рисунку нижче не будуть досконалими, оскільки не задовольняють умові заповнення листами (починаючи зліва), або попередній рівень не заповнений.

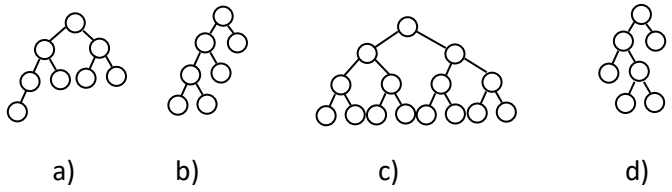


Рисунок 7.10 Досконале бінарне дерево

Мінімальна кількість вузлів n у досконалого бінарного дерева $n=2^h$

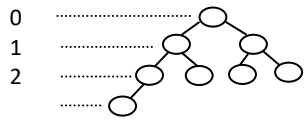


Рисунок 7.11. Досконале бінарне дерево

Звідси, максимальна висота дерева, що відповідає кількості рівнів h у досконалого бінарного дерева, як видно з рисунка 7.11 $h=\log_2 n$.

Ідеальне бінарне дерево – це бінарне дерево у якому всі внутрішні вузли мають рівно двох дітей і всі листи розміщені на одному рівні.

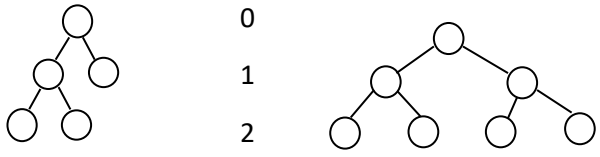


Рисунок 7.12 Ідеальне бінарне дерево

На рисунку 7.12 справа показано ідеальне бінарне дерево, а зліва – бінарне дерево, що не є ідеальним, оскільки не всі листи розміщені на одному рівні.

В таблиці 7.1 подано узагальнюючу інформацію про властивості бінарних дерев різного виду.

Окрім наведених вище видів бінарних дерев, варто згадати вироджене бінарне дерево всі внутрішні вузли якого мають рівно одну дитину.

Таблиця 7.1. Властивості бінарних дерев

	max кількість вузлів	min кількість вузлів	min висота дерева	max висота дерева
Бінарне Дерево	$2^{(h+1)} - 1$	$(h + 1)$	$\lceil \log_2(n+1) - 1 \rceil$	$n - 1$
Повне БД	$2^{(h+1)} - 1$	$(2h + 1)$	$\lceil \log_2(n+1) - 1 \rceil$	$\frac{n-1}{2}$
Досконале БД	$2^{(h+1)} - 1$	2^h	$\lceil \log_2(n+1) - 1 \rceil$	$\log_2 n$

Приклади виродженого бінарного дерева показано на рисунку 7.13 і це дерево є ланцюгом.

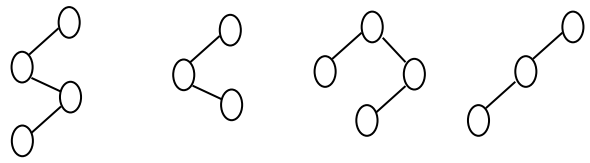


Рисунок 7.13 Вироджене бінарне дерево

7.4. Реалізація бінарного дерева на основі зв'язного списку

Розглянемо псевдокод реалізації бінарного дерева на основі двозв'язного списку. Вузол дерева реалізується як клас, що містить цілочисельні дані та два лінки. Метод Insert формує бінарне дерево так, що дані, менші кореня рекурсивно формують ліву гілку бінарного дерева. А дані, більші кореня рекурсивно формують праву гілку бінарного дерева.

Саме на основі такого впорядкування даних бінарні дерева використовують в алгоритмах пошуку і називають бінарні дерева пошуку. Приклад побудови дерева, відповідно до методу Insert, дано нижче:

```
Class Node{
   int data;
   Node left;
```

```
    Node right;
  }
Insert (value, root)
{ if (root == null)
  { root = new Node (value);
    return;
  }
  if (root.data>value)
  { if (root.left == null)
    { root.left = new node (value);
      return;
    }
  insert (value, root.left);
  }
if (root.data<value)
  { if (root.right == null)
    { root.right = new Node (value);
      return;
    }
  insert (value, root.right);
  }
```

7.5. Реалізація бінарного дерева на основі масиву

Розглянемо приклад бінарного дерева, що показане на рисунку. Дерево з коренем «A» має 9 вершин, що розміщені на чотирьох рівнях – від нульового і до третього. В таблиці справа наведена реалізація цього бінарного дерева на основі масиву, що починається з нуля. Індексація вершин бінарного дерева, що відповідає таблиці 7.2, наведена на рисунку 7.14.

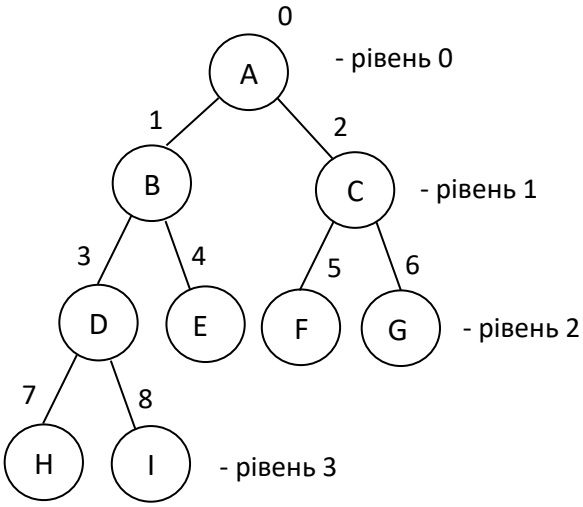


Рисунок 7.14. Бінарне дерево

Таблиця7.2 Індексація вершин дерева

A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

Оскільки для реалізації бінарного дерева на основі масиву потрібно мати саме досконале бінарне дерево, то як видно з наведеного вище рисунку, розглядається Досконале Бінарне Дерево. Тобто Дерево, у якого всі рівні заповнені, а останній рівень, на якому знаходяться листи, заповнено зліва направо.

Наступна формула для масиву (масиву, що починається з нуля) встановлює зв'язок між вузлами досконалого Бінарного Дерева та індексами масиву. Для вузла Дерева з *i*-тим індексом, занумерованим по Дереву зверху вниз, зліва направо:

ліва дитина має індекс (2·i+1),

права дитина має індекс (2·i+2),

батьківська вершина має індекс $\left\lfloor\frac{i-1}{2}\right\rfloor$

Наприклад, на рисунку 7.11, вузол «E» має індекс в масиві i = 4 тому його батьківська вершина має індекс $\left\lfloor\frac{i-1}{2}\right\rfloor = \left\lfloor\frac{4-1}{2}\right\rfloor = 1$. Це вузол «B» на рисунку вище. Ліва та права дитина матимуть індекси:

(2·i+1) = (2·4+1) = 9,

(2·i+2) = (2·4+2)=10

Проте, в масиві ці елементи відсутні, тобто вузол «E» не має дітей.

Аналогічно, формула для масиву (масиву, що починається з «1») встановлює зв'язок між вузлами досконалого бінарного дерева та індексами масиву. Для вузла дерева з *i*-тим індексом, занумерованим по дереву зверху вниз, зліва направо:

ліва дитина має індекс (2·i),

права дитина має індекс (2·i+1),

батьківська вершина має індекс $\left\lfloor\frac{i}{2}\right\rfloor$,

батько $\left\lfloor\frac{i}{2}\right\rfloor$.

Наприклад, на рисунку 7.15 вузол «B» має індекс в масиві i = 2 тому його батьківська вершина має індекс $\left\lfloor\frac{i}{2}\right\rfloor = \left\lfloor\frac{2}{2}\right\rfloor = 1$. Це вузол «A» на рисунку нижче. Ліва та права дитина матимуть індекси:

(2·i) = (2·2) = 4, вузол «D» на рисунку,

(2·i+1) = (2·2+1)=5, вузол «E» на рисунку.

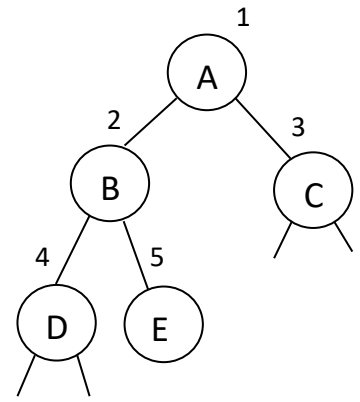
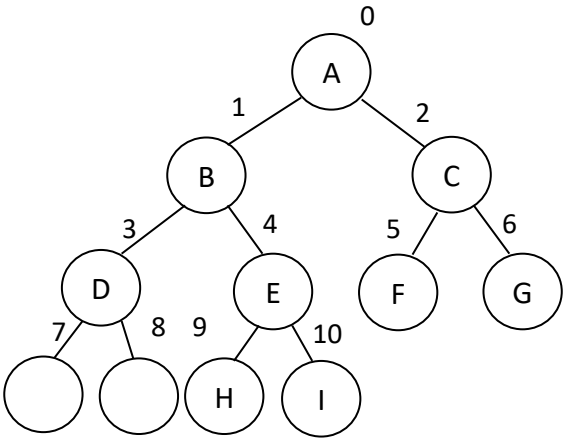


Рисунок 7.15 Бінарне дерево

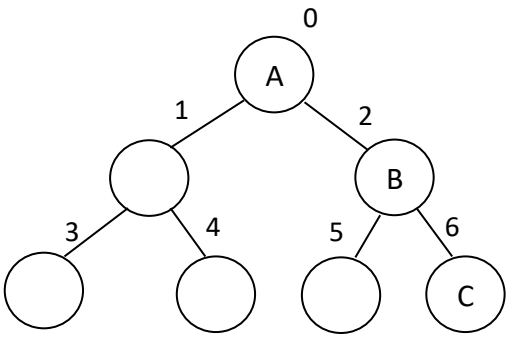


A	B	C	D	E	F	G	-	-	H	I
0	1	2	3	4	5	6	7	8	9	10

Рисунок 7.16 Бінарне дерево та відповідний йому масив

Якщо бінарне дерево, що потрібно реалізувати на основі масиву, не є досконалим бінарним деревом, то його приводять до досконалого. Приклад такого приведення показано на рисунку 7.16, де зображено бінарне дерево та відповідний йому масив.

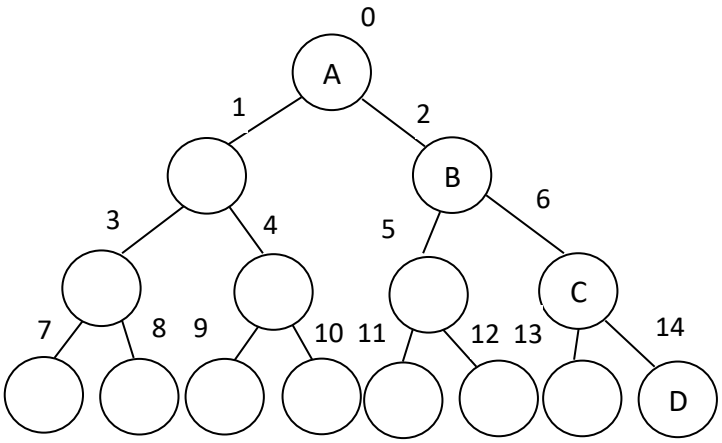
Однак, таке приведення бінарного дерева до досконалого має недоліки. Зокрема, на рисунку 7.17 подано правостороннє бінарне дерево та відповідний йому масив. Як видно з масиву, за рахунок приведення дерева до досконалого, масив не ефективно використовує виділену пам'ять, що є основним недоліком. Зокрема, у даному прикладі завантаженість пам’яті складає менше 50%.



A	-	B	-	-	-	C
0	1	2	3	4	5	6

Рисунок 7.17 Бінарне дерево та відповідний йому масив

На рисунку 7.18, що ілюструє приведення бінарного дерева до досконалого завантаженість пам’яті під масив складає 4/15= 27%.



A	-	B	-	-
0	1	2	3	4

-	C	-	-	-
5	6	7	8	9

-	-	-	-	D
10	11	12	13	14

Рисунок 7.18 Бінарне дерево та відповідний йому масив

Саме тому бінарні дерева частіше реалізують на основі двозв’язних списків, що не мають вказаних недоліків.

7.6. Обхід бінарного дерева

Оскільки дерево, в цілому, та бінарне дерево, зокрема, є не лінійною структурою, його можна обходити – проходити через всі вершини рівно один раз – по-різному.

Базуючись на обході графа в глибину – тобто рухаючись так далеко від кореня Дерева, як можна, розрізняють:

1) Прямий (preorder) порядок обходу дерева (обхід зверху батько і діти). Порядок наступний:

a) корінь,

b) рекурсивно ліве піддерево,

c) рекурсивно праве піддерево.

Наприклад, для зображеного на рисунку дерева, прямий порядок обходу записується наступним чином: A B D C E G F H I

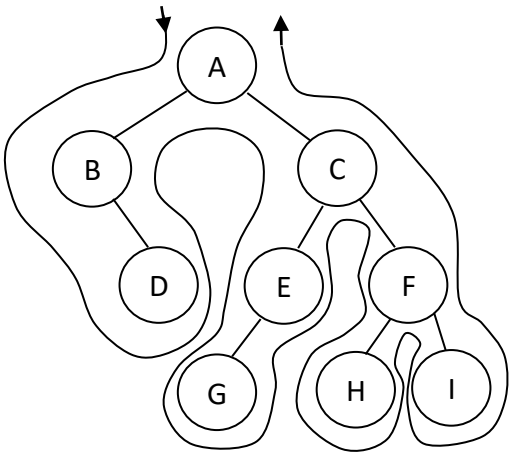


Рисунок 7.19 Обхід дерева

2) Зворотній (postorder) порядок обходу дерева (обхід знизу спочатку діти, потім батько). Порядок наступний:

a) рекурсивно ліве піддерево (LRN),

b) рекурсивно праве піддерево,

c3) корінь.

Наприклад, для зображеного на рисунку 7.19 дерева, зворотній порядок обходу записується наступним чином: D B G E H I F C A.

3) Серединний (inorder) порядок обходу дерева (ліва дитина, батько, права дитина). Порядок наступний:

a) рекурсивно ліве піддерево,

b) корінь,

c) рекурсивно праве піддерево.

Наприклад, для зображеного на рисунку 7.19 дерева, серединний порядок обходу записується наступним чином:

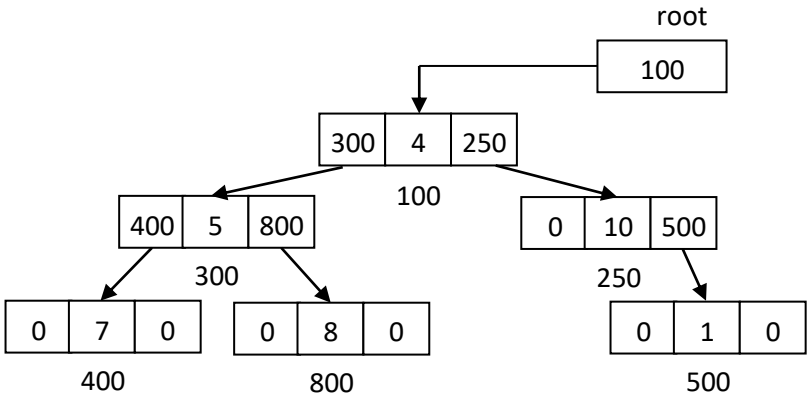


Рисунок 7.20 Бінарне дерево, як список

B D A G E C H F I.

Нижче дано псевдокод алгоритму пошуку в глибину (DFS) з використанням прямого порядку обходу дерева

```
Struct Node {
    char data;
    Node *left;
    Node *right;
  };
void Preorder (Node *root){
  if (root == null) return;      // Прямий обхід дерева на рисунку 7.20
print (“/d”, root→data);       //  за даним алгоритмом 4 5 7 8 10 1.
  Preorder (root→left);
  Preorder (root→right) }
```

Як видно, з псевдокоду, спочатку формується вузол дерева і далі функція Preorder запускає рекурсивний обхід дерева, якщо вказівник вузла не порожній.

Для наведеного псевдокоду встановимо часову складність рекурсивного алгоритму:

$T(n) = 1+T(n-1)$, $T(n-1) = 1+T(n-2)$, ...

$T(n) = 1+(1+T(n-2)) = 2+T(n-2) \Rightarrow T(n) = k+T(n-k)$.

Розглянемо базу індукції: якщо $n=0$, то $T(0)=1$ виконується операція «if». Звідси:

$n-k=0 \Rightarrow k=n$.

З урахуванням $T(0) = 1$, маємо $T(n) = n+T(n-n) = n+ T(0) = n+1$.

Маємо часову складність алгоритму $0(n)$.

7.7. Видалення вузла в бінарному дереві

Алгоритм для додавання вузла в бінарне дерево розглядався раніше у вигляді псевдокоду.

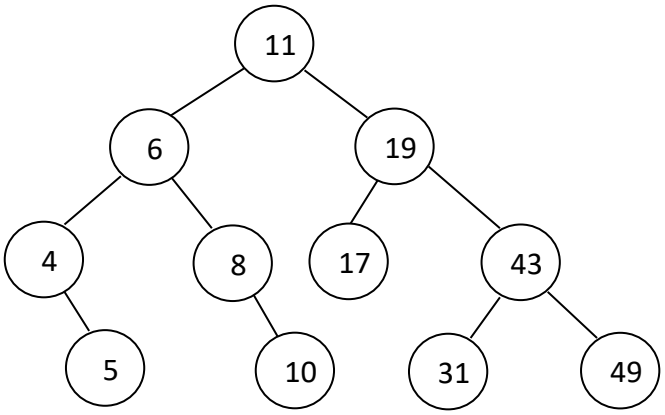


Рисунок 7.21 Додавання вузла в дерево

Припустимо, в бінарне дерево на основі алгоритму, описаного функцією insert, занесено наступні дані: 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31. Зображення отриманого бінарного дерева наведено на рисунку 7.21.

Розглянемо алгоритм видалення вузла з Бінарного Дерева. В загальному випадку при видаленні вузла з Бінарного Дерева може бути три випадки: у вузла, що видаляється, - 0 дітей, або у вузла, що видаляється, - 1 дитина, або у вузла, що видаляється, -2 дитини. Розберемо послідовно ці випадки.

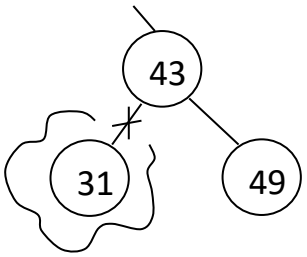


Рисунок 7.22 Видалення листа

1) У вузла, що видаляється і є листом, - 0 дітей. Це найпростіший випадок, тоді для видалення вузла – просто видалити лінк, як показано на рисунку 7.22.

2) У вузла, що видаляється, - 1 дитина. Тоді для видалення вузла він займає місце дитини та видаляється, як в 1 випадку. Наприклад, для видалення вузла «8» виконується обмін даних між вузлами «8» та «10» (8⇄10) а далі вузол «8» який став листом, видаляється за варіантом 1). Аналогічно видаляється і вузол «4», як показано на рисунку 7.23

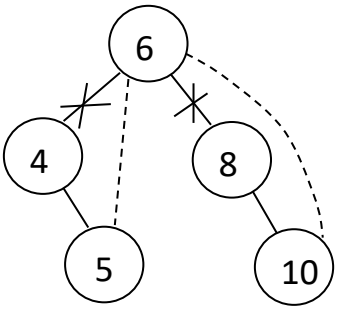


Рисунок 7.23 Видалення вузла

3) У вузла, що видаляється, - 2 дитини. Тоді для видалення вузла формується серединний порядок обходу бінарного дерева, далі виконується обмін даних між вузлами – що видаляється та попереднім inorded predecessor (наступним inorder successor). В результаті вузол, що видаляється, стає листом. Потім лист видаляється за варіантом 1, як показано на рисунку 7.24.

Наприклад, вузол «11» за серединним порядком обходу наведеного вище дерева має попередній у списку вузол «10» і наступний – вузол «17». Замінюємо вузол «11» у списку на вузол «10» (або на вузол «17») та переходимо до видалення вузла, що є листом, за варіантом 1).

Серединний порядок обходу наведеного нижче дерева:

4, 5, 6, 8, 10, 11, 17, 19, 31, 43, 49.

відповідає наведеному нижче рисунку 7.18.

Після заміни в списку вузлів «10» та «11» серединний порядок обходу наведеного нижче бінарного дерева має вигляд (див. рисунок 7.25, зліва):

4, 5, 6, 8, 11, 10, 17, 19, 31, 43, 49 .

Далі переходимо до видалення вузла «11» за варіантом 1. Після цього Бінарне Дерево має вигляд, показаний на рисунку 7.25, справа):

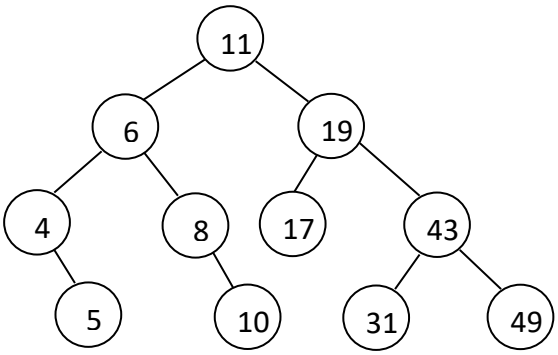


Рисунок 7.24. Бінарне дерево

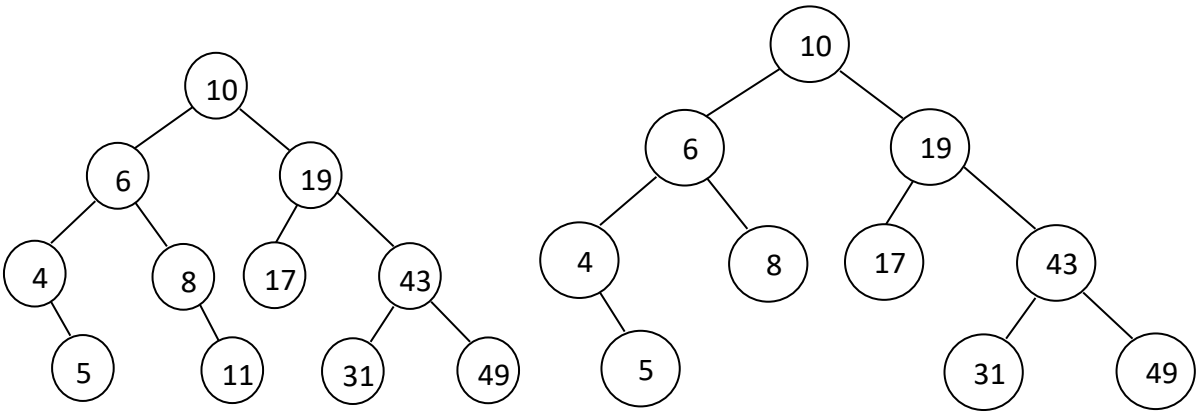


Рисунок 7.25. Бінарне дерево, видалення кореня

Тобто, ідея полягає в тому, щоб знайти максимальний елемент у лівому піддереві, замінити його з коренем та видалити корінь, що став листом. Якщо заміна йде в правому піддереві, то в правому піддереві знаходять мінімальний елемент, замінюють його з коренем та видаляють корінь.

7.8. Лабораторна робота. Бінарні дерева

Виконати завдання, відповідно до варіанту

1. Створити копію дерева в зеркальному відображенні.
2. Побудувати дерево пошуку з елементів правого піддерева ідеально збалансованого дерева.

3. Побудувати генеалогічне дерево. Вивести імена всіх прабабусь по материнській лінії у вигляді: бабуся – Ольга, прабабуся - Тетяна і т.д.

4. По дереву пошуку надрукувати ключі в порядку збільшення значень, розташувати їх у рядок.

5. По генеалогічному дереву вивести імена дідусів, прадідусів.

6. Переставити місцями самий лівий і самий правий елементи дерева пошуку.

7. Перевірити чи є задане бінарне дерево деревом пошуку.

8. Підрахувати кількість вузлів на нижньому рівні (кількість рівнів підрахувати при створенні дерева).

9. Підрахувати кількість вузлів на кожному рівні дерева.

10. Найти суму елементів правого ребра правого піддерева.

11. Надрукувати всі листки (термінальні елементи) дерева.

12. На основі заданого дерева пошуку побудувати ідеально збалансоване дерево.

13. Елементи лівого піддерева збалансованого дерева пошуку зменшити на 1, а правого – збільшити у 10 разів.

14. Для заданої послідовності цілих чисел побудувати дерево пошуку.

15. Визначити, на якому рівні сума елементів максимальна.

16. Визначити чи є в даному дереві елемент із заданим ключем, і, якщо є, то визначити, чи є він коренем, чи вершиною з одним нащадком, вершиною з двома нащадками, листком.

17. Визначити довжину шляху від кореня до вершини з заданим ключем.

18. Вивести елементи заданого ідеально збалансованого дерева за зменшенням.

19. В лівому піддереві замінити значення елементів – листків на 100.

20. В дереві пошуку мінімальний елемент зменшити вдвоє, максимальний збільшити вдвоє.

21. В ідеально збалансованому дереві поміняти місцями максимальний та мінімальний елементи.

7.9. АВЛ – дерева

Бінарне дерево, що використовується в алгоритмі пошуку, найшвидше працює, якщо кожен вузол, що не є листом, містить двох дітей. Таке дерево називається бінарним деревом пошуку. Всі і інші випадки побудови бінарного дерева призводять до зменшення ефективності алгоритму пошуку в бінарному дереві. Тому задача отримати бінарне дерево пошуку, незалежно від порядку вставки, чи видалення елементів, є важливою.

Одним зі способів вирішення задачі є побудова збалансованого бінарного дерева. Зокрема АВЛ-Дерево є збалансованим бінарним деревом. Дерево назване в честь авторів алгоритмів балансування: Адельсона-Вельського та Ландіса. У АВЛ дереві балансування визначається фактором балансування, що може приймати значення {-1; 0; 1}

Фактор балансування *f* є різницею висот лівого та правого піддерев для кожного із вузлів за формулою: висота лівого піддерева мінус висота правого піддерева.

Для оголошення вузла в АВЛ дереві використовується наступний клас. Як видно, кожен вузол окрім значення, містить фактор балансування (height) на основі якого і виконується балансування вузлів.

```
private static class Node {
        int value;
        int height;
        Node left;
        Node right;
    }
```

На рисунку 7.26, як приклад наведено збалансоване бінарне дерево. Фактор балансування для кожного з вузлів наведено поряд. Зокрема, листи A, E мають фактор балансування 0, оскільки у них жодного нащадка. Вузол B має f = 1, оскільки має одне ліве ребро. Аналогічно, вузол D має f = -1, оскільки має одне праве ребро. Вузол C має f = 2-2 = 0, оскільки має два ліві ребра та два праві, що віднімаються.

Оскільки f для будь-якого з вузлів дорівнює {-1; 0; 1}, то бінарне дерево – збалансоване.

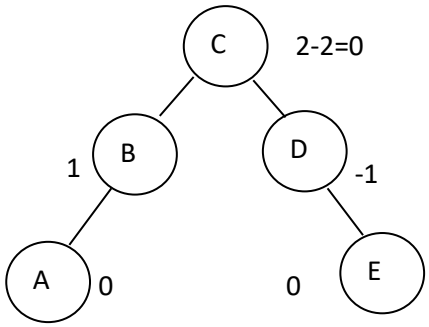


Рисунок 7.26. Збалансоване бінарне дерево

Наступні 5 прикладів – від простішого до складнішого, це випадки балансування АВЛ-дерев, якщо дерево має три вузли.

Приклад 1. Бінарне дерево містить вузли a, b, c, як зображено нижче на рисунку 7.27 зліва Припустимо, a < b < c. Побудова і балансування такого дерева показані на рисунку 7.28.

Фактор балансування кожного з вузлів на дереві зліва наступний f_a = -2, f_b = -1, f_c = 0. Розглядаючи бінарне дерево, як АВЛ-дерево маємо його збалансувати щоб отримати бінарне дерево пошуку (BST). Для балансування дерева виконується лівий поворот (L), що дає збалансоване дерево (показано на рисунку 7.27 справа).

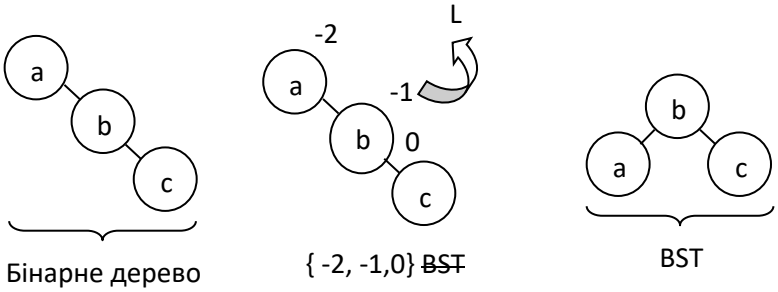


Рисунок 7.27 Балансування бінарного дерева, лівий поворот

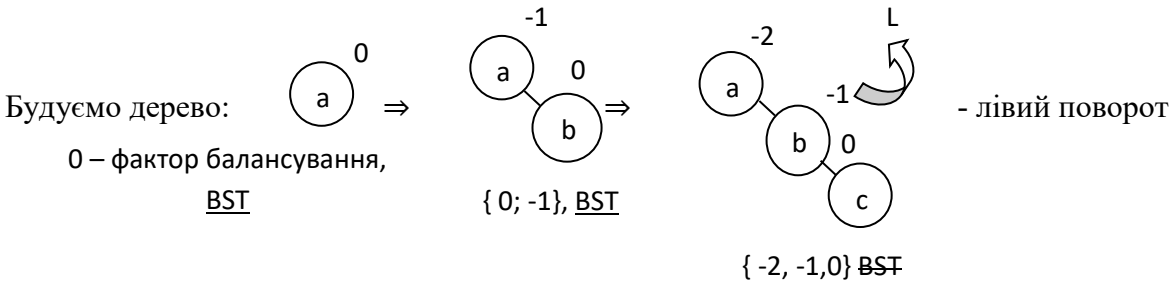


Рисунок 7.28 Побудова і балансування бінарного дерева, лівий поворот

Приклад 2. Бінарне дерево містить вузли c, b, a, як зображено на рисунку 7.29 (a < b < c).

Фактор балансування кожного з вузлів на дереві зліва наступний f_a = 0, f_b = 1, f_c = 2.

Розглядаючи бінарне дерево, як АВЛ-дерево маємо його збалансувати щоб отримати бінарне дерево пошуку (BST). Для балансування дерева виконується правий поворот (R), що дає збалансоване дерево (показано на рисунку 7.29 справа).

У прикладі 2 не можна робити лівий поворот, оскільки отримаємо дерево, що не буде бінарним деревом пошуку і вузол «a» не буде зліва як показано на рисунку 7.28 зліва.

Приклад 3. Бінарне Дерево містить вузли a, c, b як зображено на рисунку 7.30 зліва (a < b < c).

Фактор балансування кожного з вузлів на дереві зліва наступний f_a = -2, f_b = 0, f_c = 1, тобто бінарне дерево – не збалансоване. У цьому випадку для балансування використовується подвійний правий і лівий поворот (RL), що дає збалансоване дерево (показано на рисунку 7.30 справа).

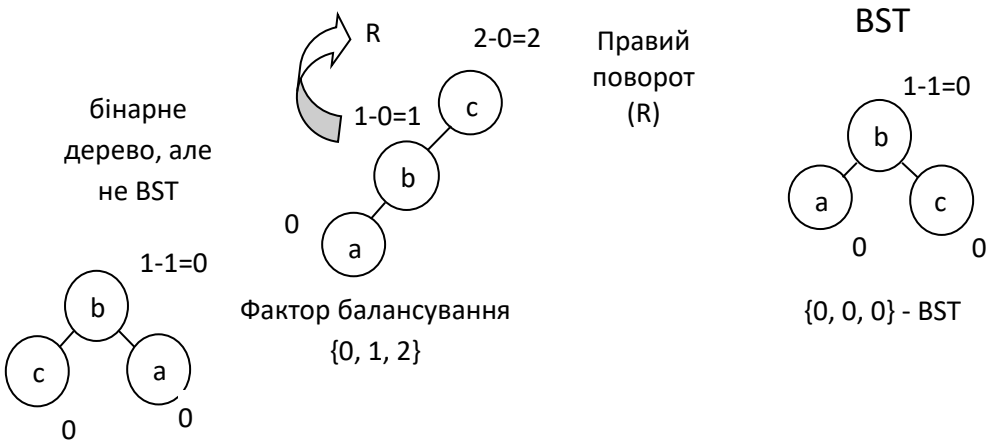


Рисунок 7.29 Балансування бінарного дерева, правий поворот

Приклад 4. Бінарне Дерево містить вузли c, a, b як зображено нижче на рисунку 7.30 зліва (a < b < c).

Фактор балансування кожного з вузлів на дереві зліва наступний f_a = -1, f_b = 0, f_c = 2, тобто бінарне дерево – не збалансоване. У цьому випадку для балансування використовується подвійний лівий і правий поворот (LR), що дає збалансоване дерево (показано на рисунку 7.31 справа).

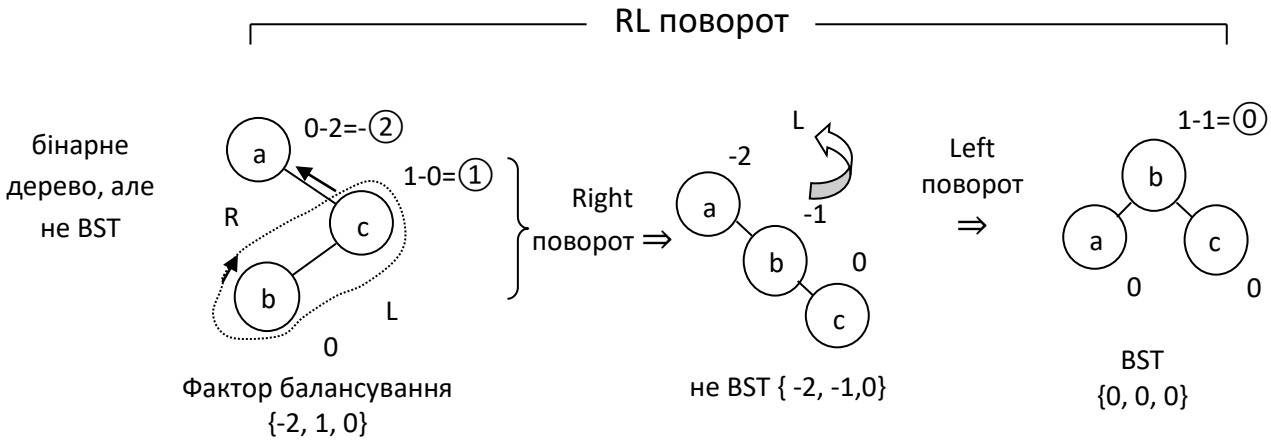


Рисунок 7.30 Балансування бінарного дерева, правий і лівий поворот

Після першого L повороту вузлів у бінарному дереві фактор балансування кожного з вузлів $f_a = 0$, $f_b = 1$, $f_c = 2$, тобто бінарне дерево – все ще не збалансоване (рисунок нижче посередині).

Тому надалі виконується R поворот вузлів у бінарному дереві, що дає фактор балансування кожного з вузлів $f_a = 0$, $f_b = 0$, $f_c = 0$, тобто бінарне дерево – збалансоване, з використанням алгоритмів АВЛ. Саме тому такі бінарні дерева отримали назву АВЛ дерева.

Розглянемо наступний приклад 5, що демонструє побудову АВЛ-дерева з використанням розглянутих вище поворотів R, L, RL, LR.

Приклад 5. Припустимо, дані для внесення у вершини дерева надходять у наступній послідовності 15, 18, 12, 8, 55, 5, 14, 13, 9, 61, 20, 17, 21. Зрозуміло, що всі дані в дереві є унікальними і не можуть повторюватися.

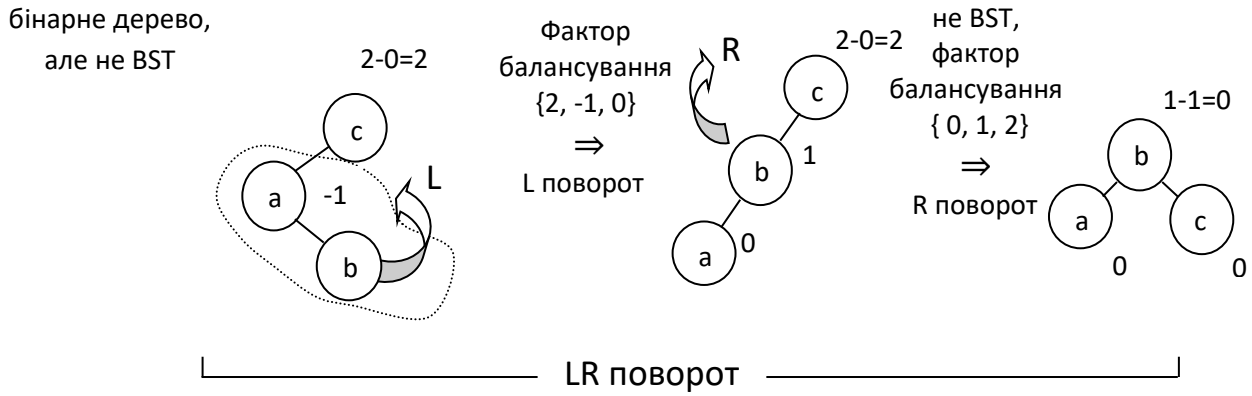


Рисунок 7.31 Балансування бінарного дерева, лівий і правий поворот

На рисунку 7.32, зліва направо, показано процес послідовної побудови АВЛ-дерева. Особливістю побудови є обрахунок фактору балансування всіх вершин на кожному етапі, після додавання вузла. Після цього, за потреби, дерево перебудовується.

Цифра, що вказана на рисунку, поруч з вузлом, є значенням фактору балансування для даної вершини. Коли, при додаванні вузла фактор балансування будь-якої з вершин буде відмінний від -1, 0, 1 – дерево перебудовується і перебалансовується.

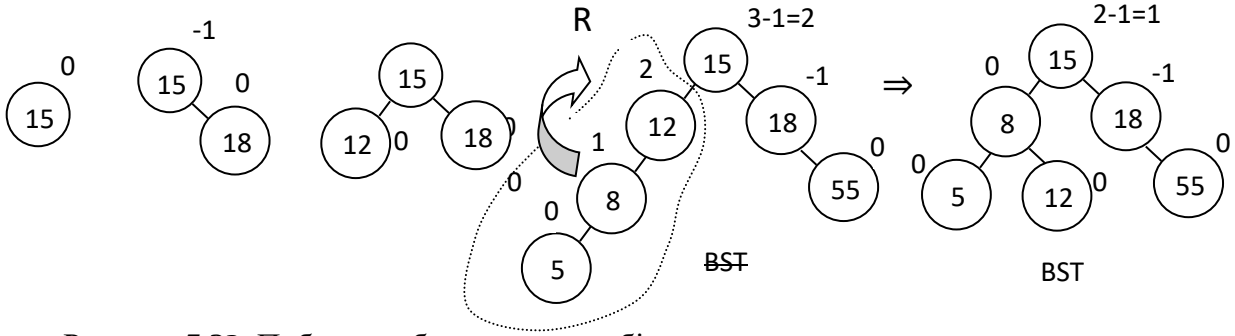


Рисунок 7.32 Побудова збалансованого бінарного дерева

На рисунку 7.33 показано виконання RL повороту після додавання вузла «13».

Після додавання вузла «9» дерево стає не збалансованим і критичний вузол, що містить фактор балансування f= -2 показано на рисунку 7.34. Саме для критичного вузла виконується RL поворот. Після повороту вузол-лист «14» залишається за «13», а вузол-лист «9» не може залишитися на своєму місці. Тому «9» перебалансовується, відповідно до правил побудови бінарного дерева. Результат показано на рисунку 7.33, справа.

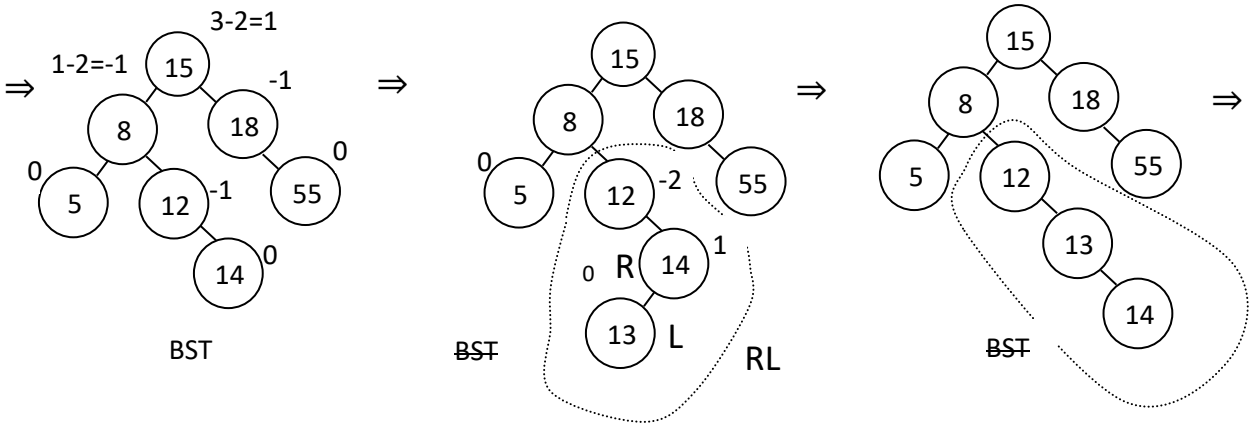


Рисунок 7.33 Виконання RL повороту після додавання вузла «13»

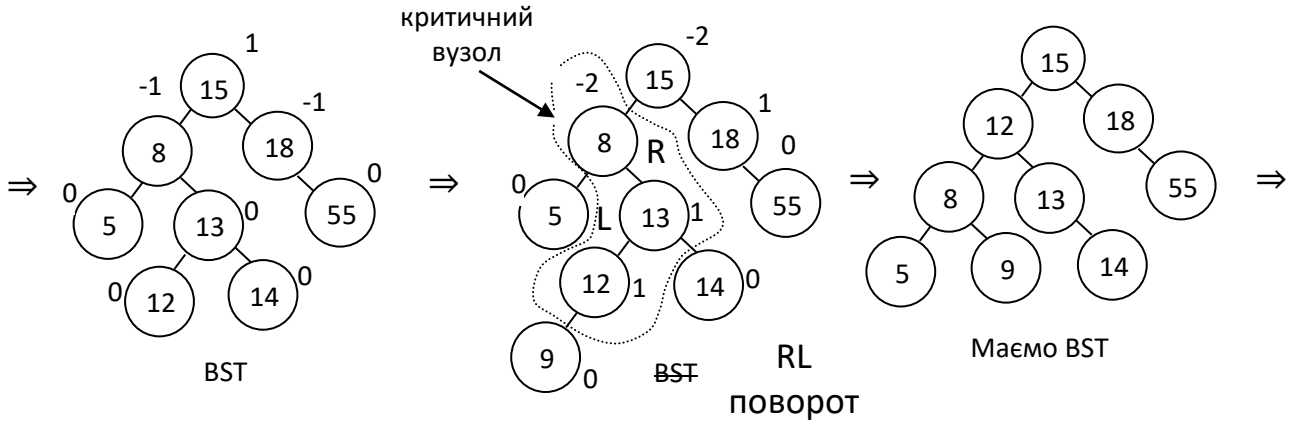


Рисунок 7.34 Виконання RL повороту

В АВЛ-дерево залишилося додати 61, 20, 17, 21. Виконуємо, як показано на рисунку 7.35. Після додавання «61» отримуємо не збалансоване бінарне дерево, що має критичний вузол і який на рисунку нижче помічений пунктиром. Як видно, фактор балансування критичного вузла складає -2. Для балансування виконується лівий поворот

(L) результати якого показано на рисунку нижче, справа. Також до дерева додано вузли «20» та «17», що не порушило його балансування, як видно на рисунку справа.

Додавання вузла «21» дає не збалансоване бінарне дерево, що має критичний вузол з фактором балансування 2 що на рисунку 7.35, зліва помічений пунктиром. Для балансування виконується лівий і правий поворот (LR) результати якого показано на рисунку 7.36.

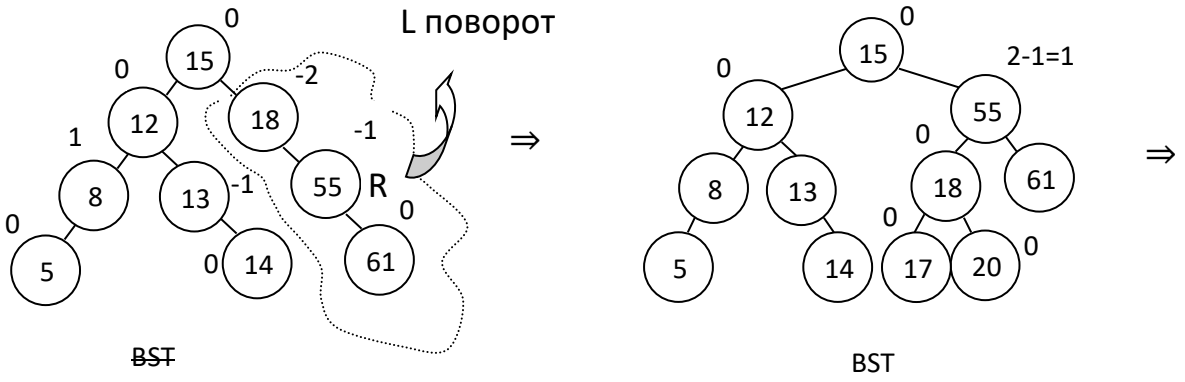


Рисунок 7.35 Виконання L повороту після додавання вузла «61»

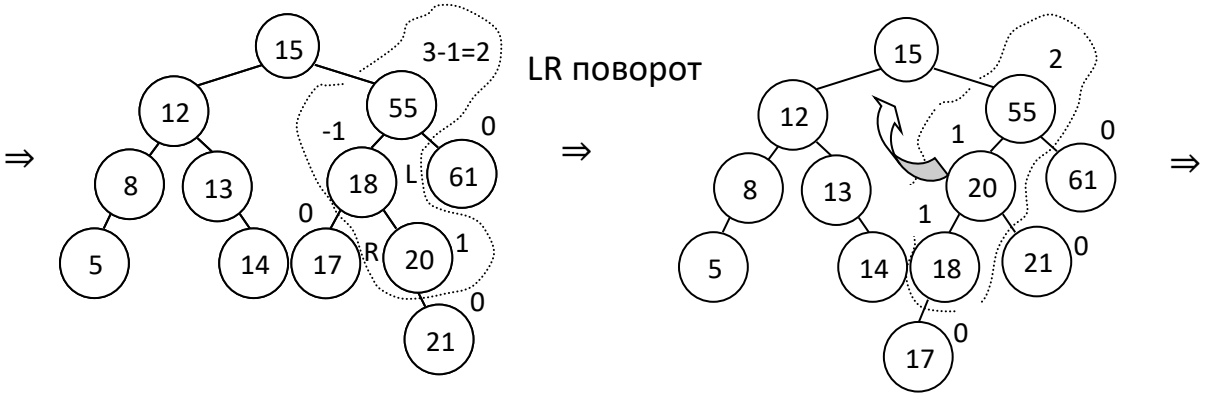
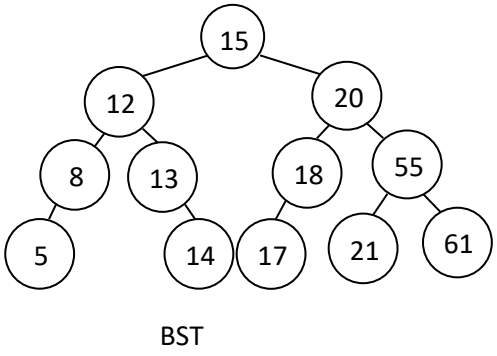


Рисунок 7.36 Виконання LR повороту після додавання вузла «21»



Після LR повороту, вузол «21» не може залишатися на своєму місці і тому він переміщується, відповідно до правил побудови Бінарних Дерев.

Таким чином, АВЛ-дерево збудоване і воно є збалансованим бінарним деревом, як показано на рисунку 7.37.

Рисунок 7.37 Збалансоване бінарне дерево

7.10. Лабораторна робота. АВЛ - дерева, побудова збалансованих бінарних дерев

Мета роботи: теоретично засвоїти та практично реалізувати способи балансування Бінарних дерев – АВЛ-дерев. Теоретично засвоїти та практично реалізувати алгоритм обходу бінарного дерева.

Завдання на роботу. Відповідно до варіанту реалізувати збалансоване Бінарне дерево - АВЛ -дерево, що містить як мінімум 5 вузлів. Для перевірки балансування АВЛ - дерева після кожного додавання (видалення) вузла слід обов'язково (саме обов'язково) виводити Дерево. Для виводу Дерева на консоль використати – прямий preorder (варіанти 1-10), зворотний postorder (варіанти 11-20) та серединний inorder (варіанти 21-30) порядок обходу вузлів Дерева.

Варіанти індивідуальних завдань

1. Вузли дерева містять дати у формі *dd.mm.*
2. Вузли дерева містять код та назву книги (два поля, сортування за кодом)
3. Вузли дерева містять назву та ціну товару (два поля, сортування за назвою)
4. Вузли дерева містять назви країн Америки.
5. Вузли дерева містять таймінг у формі *hh.mm.ss* і подію (два поля, сортування за таймінгом)
6. Вузли дерева містять назви країни Європи та кількість жителів (два поля, сортування за назвою)
7. Вузли дерева містять назви країни Африки та кількість жителів (два поля, сортування за кількістю)
8. Вузли дерева містять список студентів групи.
9. Вузли дерева містять міста України.
10. Вузли дерева містять студентів групи та їх телефони (два поля, сортування за Прізвищем).

Список використаних джерел

1. Сортування вибором: веб-сайт. URL: https://training.epam.ua/#!/News/425?lang=ua
2. Інтерполяційний пошук: веб-сайт. URL: https://www.cs.technion.ac.il/~itai/publications/Algorithms/p550-perl.pdf
3. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. – К.: "Вильямс", 2016. – 400 с.
4. Ковалюк Т.В. Основи програмування. – К.: Видавнича група BHV, 2015. – 394 с.
5. C++ Tutorials: веб-сайт. URL: http://www.cplusplus.com/doc/tutorial
6. Кнут Д. Искусство программирования: Пер. с англ. / Ю.В. Козаченко (общ.ред.), С.Г. Тригуб (пер.с англ.и ред.). — Испр. и доп. изд. —К. : Издательский дом "Вильямс", 2005. - Т. 1 : Основные алгоритмы. — 712с.
7. Вирт Н. Алгоритмы и структуры данных. – М.: Мир, 2019. – 360 с.
8. Лафоре Р. Структуры данных и алгоритмы Java. – С.-Пб.: "Питер", 2017. – 704 с.
9. Шилдт Герберт. Java. Полное руководство 10 вид.. — М-СПб., 2018. — 1488 с.