МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Одеська національна академія зв'язку ім. О.С. Попова

Кафедра інформаційних технологій

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Конспект лекцій для студентів напряму
6.050103 – Програмна інженерія

Частина 2. Алгоритми пошуку, стиснення даних, внутрішнього та зовнішнього сортування, алгоритми на графах

Одеса 2017

УДК 004.43 План НМВ 2017 р.

Рецензент: к.ф-м.н., доцент кафедри Математичного забеспечення комп'ютерних систем ОНУ ім. І.І. Мечнікова, Антоненко О.С.

Алгоритми та структури даних: конспект лекцій.
Частина 2. Алгоритми пошуку, стиснення даних, внутріщнього та зовнішнього сортування, алгоритми на графах / Укладачі: О.Д. Воробйова, Л.В. Глазунова – Одеса:ОНАЗ ім. О.С. Попова, 2017. – 52 с.

Конспект лекцій містить короткі теоретичні відомості зі структур даних та основних алгоритмів обробки даних, таких як пошук елемента у лінійних структурах чи пошук елемента у тексті, сортування масиву, стиснення даних, пошук найкоротшого шляху. Алгоритми та структури даних – найважливіший розділ комп'ютерних наук і програмування. Завдяки розвитку інформаційних технологій та алгоритмам ми сьогодні маємо можливість швидко знаходити інформацію в Інтернеті (зокрема, шукати по картинках), знаходити найкоротші шляхи, аналізувати геноми та інші. Алгоритми використовуються практично в усіх областях комп'ютерних наук – в аналізі зображень, Інтернет-пошуку, машинному навчанні, біоінформатиці, криптографії, кодуванні, мережах, розподілених системах, компіляторах.

Конспект лекцій призначено студентам для здобуття теоретичних і практичних знань з напряму 6.050103 – Програмна інженерія.

СХВАЛЕНО
на засіданні кафедри
інформаційних технологій
та рекомендовано до друку.
Протокол № 4
від 28 грудня 2016 р.

ЗАТВЕРДЖЕНО
методичною радою
академії зв'язку.
Протокол № 5
від 28.02.2017 р.

ПЕРЕДМОВА

Дисципліна «Алгоритми та структури даних» присвячена формуванню наукової системи мислення, вмінню проектувати алгоритми і структури даних. Перед вивченням даного навчального курсу передбачається, що студенти володіють знаннями в області найпростіших алгоритмів, володіють мовою програмування високого рівня, володіють основами об'єктно-орієнтованого підходу в аналізі і проектуванні систем.

Поняття "алгоритм і структура даних" є центральним у сфері комп'ютерних технологій, однак, щоб називати деякі структури даних і алгоритми якісними та ефективними, слід використовувати точні прийоми їх аналізу. Як природний критерій якості можно виділити, по-перше, час виконання. Також важливим є обсяг витрачених ресурсів пам'яті і дискового простору. Увагу також слід приділити надійності і достовірності рішень, їх стабільності.

У результаті вивчення другої частини навчальної дисципліни студент повинен *знати:*

- критерії оцінки ефективності алгоритмів;
- алгоритм послідовного пошуку;
- алгоритм бінарного пошуку;
- алгоритм прямого пошуку в тексті ;
- алгоритм Кнута, Морріса і Пратта пошуку в тексті;
- алгоритм Бойєра і Мура пошуку в тексті;
- алгоритм Хаффмана стиснення даних;
- реалізацію алгоритму Хаффмана з використанням кодових дерев;
- класифікацію алгоритмів сортування;
- бінарне сортування пірамідою;
- алгоритм сортування Шелла;
- алгоритм швидкого сортування Хоара;
- алгоритм сортування злиттям;
- алгоритми на графах: пошук в глибину, пошук в ширину;
- алгоритм пошуку найкоротшого шляху Дейкстри;
- алгоритм пошуку найкоротшого шляху Флойда.

вміти:

- вибирати необхідний алгоритм для розв’язання поставленої задачі;
- оцінювати ефективність обраного алгоритму;
- програмувати основні алгоритми пошуку, сортування, стиснення та пошуку найкоротшого шляху.

Лекція 5
АЛГОРИТМИ ПОШУКУ В ЛІНІЙНИХ СТРУКТУРАХ

Аналіз алгоритма

Алгоритм – це формально описана обчислювальна процедура, що отримує вихідні дані (input), які називаються також його аргументами, та яка видає результат обчислення на вихід (output).

output Функція(input)
{ процедура; }

Алгоритм визначає функцію (відображення) F: X->Y,
де X – множина вихідних даних, Y – множина значень.

Ефективність алгоритму визначається різними характеристиками, що залежать від вихідних даних:

– *час сортування* − основний параметр, що характеризує швидкодію алгоритму;

– *пам'ять* − один із параметрів, який характеризується тим, що низка алгоритмів вимагає виділення додаткової пам'яті під тимчасове зберігання даних. При оцінці використовуваної пам'яті не буде враховуватися місце, яке займає вхідний масив даних, та витрати, які не залежать від вхідної послідовності, наприклад, на зберігання коду програми.

Часто кількість вихідних даних характеризується натуральним числом n, тоді використовуються позначення: час роботи алгоритму − $T(n)$, обсяг пам'яті − $M(n)$. Наприклад, в задачі сортування важливо тільки кількість елементів.

Вихідні дані можуть характеризуватися декількома числами: наприклад, завдання визначення входження рядка-шаблона T довжиною k в рядок S довжиною n – час роботи залежить від двох параметрів $T(k, n)$.

Асимптотичне наближення

Для позначення асимптотичної поведінки часу роботи алгоритму або обсягу пам'яті використовуються позначення Θ, O, Ω.

Визначення. Для функції $g(n)$ записи $\Theta(n)$, $O(n)$, $\Omega(n)$ означають таку множину функцій:

$\Theta(g(n)) = \{f(n)$: існують позитивні константи c_1, c_2 і n_0 такі, що $0 < c_1 g(n) < f(n) < c_2 g(n)$ для всіх $n > n_0\}$

$O(g(n)) = \{f(n)$: існують позитивні константи c і n_0 такі, що $0 < f(n) < cg(n)$ для всіх $n > n_0\}$

$\Omega(g(n)) = \{f(n)$: існують позитивні константи c і n_0 такі, що $0 < cg(n) < f(n)$ для всіх $n > n_0\}$.

Записується як $T(n) = O(g(n))$.

Приклади апроксимаціїї часу роботи алгоритму: $T(n) = \Theta(n)$ – лінійний час, $T(n) = \Theta(n^2)$ – квадратичний час, $T(n) = \Theta(\log n)$ – логарифм, $T(n) = \Theta(n \log n)$, $T(n) = \Theta(e^n)$.

Алгоритми пошуку

Одною із найважливіших дій зі структурованою інформацією є пошук. Пошук − процес знаходження конкретної інформації в раніше створеній множині даних. Зазвичай дані являють собою записи, кожна з яких має хоча б один ключ. *Ключ пошуку* – це поле запису, за значенням якого відбувається пошук. Ключі використовуються для відмінності одних записів від інших. Метою пошуку є знаходження всіх записів (якщо вони є) з даним значенням ключа.

Структуру даних, в якій відбувається пошук, можна розглядати як таблицю символів (таблицю імен або таблицю ідентифікаторів) – структуру, яка містить ключі і дані, і допускає дві операції – включення нового елемента і повернення елемента з заданим ключем. Іноді таблиці символів називають словниками за аналогією з добре відомою системою упорядкування слів в алфавітному порядку: слово – ключ, його тлумачення – дані.

Поставимо задачу пошуку в лінійних структурах. Нехай задана множина даних, яка описується як масив, що складається з певної кількості елементів. Перевіримо, чи входить заданий ключ у даний масив. Якщо входить, то знайдемо номер цього елемента масиву, тобто визначимо перше входження заданого ключа (елемента) у вихідний масив.

Таким чином, визначимо загальний алгоритм пошуку даних:

Крок 1. Обчислення елемента, що часто передбачає отримання значення елемента, ключа елемента та інші.

Крок 2. Порівняння елемента з еталоном або порівняння двох елементів (в залежності від постановки задачі).

Крок 3. Перебір елементів множини, тобто проходження за елементами масиву.

Основні ідеї різних алгоритмів пошуку зосереджені в методах перебору і стратегії пошуку. Розглянемо основні алгоритми пошуку в лінійних структурах більш докладно.

Послідовний (лінійний) пошук

Послідовний (лінійний) пошук − це найпростіший вид пошуку заданого елемента на деякій множині, здійснюваний шляхом послідовного порівняння чергового розглянутого значення з шуканим до тих пір, поки ці значення не збігатимуться.

Ідея цього методу полягає в наступному. Множина елементів проглядається послідовно в деякому порядку, яка гарантує, що будуть переглянуті всі елементи множини (наприклад, зліва направо). Якщо в ході перегляду множини буде знайдений шуканий елемент, перегляд припиняється з позитивним результатом; якщо ж буде переглянуто усю множину, а елемент не буде знайдений, алгоритм повинен видати негативний результат.

Алгоритм послідовного пошуку

Крок 1. Вважаємо, що значення змінної циклу $i = 0$.

Крок 2. Якщо значення елемента масиву $x\ [i]$ дорівнює значенню ключа key, то повертаємо значення, яке дорівнює номеру шуканого елемента,

алгоритм завершує роботу. В іншому випадку значення змінної циклу збільшується на одиницю $i = i + 1$.

Крок 3. Якщо $i < k$, де k − число елементів масиву x, то виконується крок 2, в іншому випадку − робота алгоритму завершена і повертається значення, що дорівнює 1.

За наявності в масиві декількох елементів зі значенням key даний алгоритм знаходить тільки перший з них (з найменшим індексом).

```
int LinearSearch(int *x, int k, int key){
  int i = 0;
  for ( i = 0 ; i < k ; i++ )
    if ( x[i] == key )
      break;
  return i < k ? i : -1;
   }
```

Час виконання даного алгоритму пошуку для дійсних чисел дорівнює n/ε, де n − кількість елементів множини, ε − точність. Пошук на дискретній множині з n елементів здійснюється в гіршому випадку за n ітерацій, а в середньому цей алгоритм вимагає $n/2$ ітерацій циклу. Отже, час роботи послідовного пошуку пропорційний $O(n)$. Ніяких обмежень на порядок елементів у масиві даний алгоритм не вимагає.

Існує модифікація алгоритму послідовного пошуку, яка прискорює пошук. Ця модифікація є невеликим удосконаленням розглянутого алгоритму пошуку.

Ідея *пошуку з бар'єром* полягає в тому, щоб не перевіряти кожен раз в циклі умову, яка пов'язана з межами множини. Це можна забезпечити, встановивши в даній множині так званий бар'єр. Під *бар'єром* розуміють будь-який елемент, який задовольняє умовам пошуку. Тим самим буде обмежена зміна індексу.

Вихід з циклу, в якому тепер залишається тільки умова пошуку, може відбутися або на знайденому елементі, або на бар'єрі. Існують два способи установки бар'єра: додатковим елементом або замість крайнього елемента масиву.

```
   // опис функції послідовного пошуку з бар'єром
int LinearSearchWithBarrier(int *x, int k, int key){
    x = (int *)realloc(x,(k+1)*sizeof(int));
    x[k] = key;
    int i = 0;
    while ( x[i] != key )
        i++;
    return i < k ? i : -1;
}
```

Зауважимо, що пошук з бар'єром працює швидше, але час роботи алгоритму залишається таким самим $O(n)$, де n − кількість елементів множини. Набагато більший інтерес являють методи, які не тільки працюють швидко, але і реалізують алгоритми з меншою складністю.

Бінарний (двійковий) пошук

Бінарний (двійковий, дихотомічний) пошук − це пошук заданого елемента на впорядкованій множині, здійснюваний шляхом кількаразового ділення цієї множини на дві частини таким чином, що шуканий елемент потрапляє в одну з цих частин. Пошук закінчується при збігу шуканого елемента з елементом, який є межею між частинами множини або за відсутності шуканого елемента.

Бінарний пошук застосовується до відсортованої множини і полягає в послідовному розбитті множини навпіл і пошуку елемента тільки в одній половині на кожній ітерації.

Таким чином, ідея цього методу полягає в наступному. Пошук потрібного значення серед елементів упорядкованого масиву (за зростанням або за спадною) починається з визначення значення центрального елемента цього масиву. Значення даного елемента порівнюється з шуканим значенням і залежно від результатів порівняння виконуються певні дії. Якщо шукане і центральне значення будуть рівні, то пошук завершується успішно. Якщо шукане значення менше центрального або більше, то формується масив, що складається з елементів, які знаходяться ліворуч чи праворуч від центрального відповідно. Потім пошук повторюється в новому масиві (рис. 5.1).

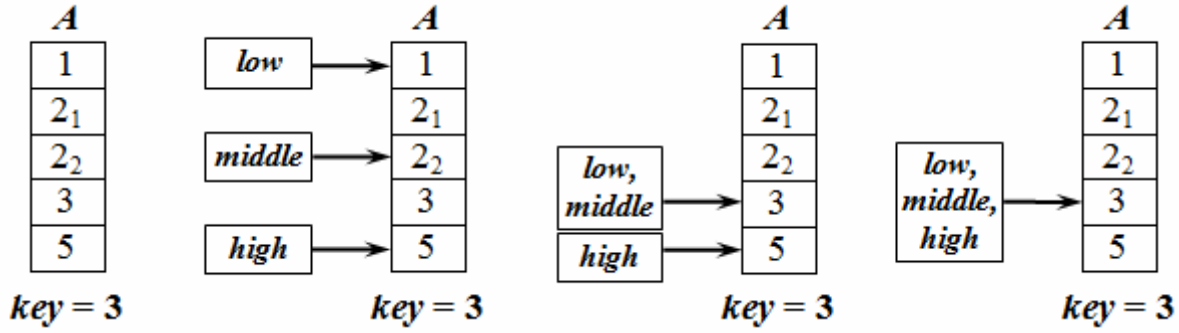


Рисунок 5.1 − Демонстрація алгоритму бінарного пошуку

Алгоритм бінарного пошуку

Крок 1. Визначити номер середнього елемента масиву middle = (high + low) / 2.

Крок 2. Якщо значення середнього елемента масиву дорівнює шуканому, то повертаємо значення, що дорівнює номеру шуканого елемента, й алгоритм завершує роботу.

Крок 3. Якщо шукане значення більше значення середнього елемента, то візьмемо як масив усі елементи праворуч від середнього, інакше візьмемо в як масив усі елементи ліворуч від середнього (в залежності від характеру впорядкованості). Перейдемо до Кроку 1.

У масиві може зустрічатися кілька елементів зі значеннями, що дорівнюють ключу. Даний алгоритм знаходить перший, який збігся з ключем елемент, але в порядку проходження в масиві може бути ні першим, ні останнім серед рівних ключу. Наприклад, в масиві чисел 1, 5, 5, 5, 5, 5, 5, 7, 8 з ключем key = 5 збіжиться елемент з порядковим номером 4, який не належить ні до першого, ні до останнього.

Існують дві модифікації даного алгоритму для пошуку першого й останнього входження. Все залежить від того, як обирається середній елемент: округленням в менший або більший бік. У першому випадку середній елемент відноситься до лівої частини масиву, а в другому − до правої.

//опис функції бінарного пошуку

```
int BinarySearch(int *x, int k, int key){
  bool found = false;
  int high = k - 1, low = 0;
  int middle = (high + low) / 2;
  while ( !found && high >= low ){
    if (key == x[middle])
      found = true;
    else if (key < x[middle])
      high = middle - 1;
    else
      low = middle + 1;
      middle = (high + low) / 2;
  }
  return found ? middle : -1 ;
}
```

У процесі роботи алгоритму бінарного пошуку розмір фрагмента, де цей пошук повинен тривати, щораз зменшується приблизно в два рази. Це забезпечує час роботи алгоритму пропорційний O(log *n*), де *n* – кількість елементів множини.

Контрольні питання

1. Чим можна пояснити різноманіття алгоритмів пошуку в лінійних структурах?

2. У чому переваги пошуку з бар'єром порівнянно з послідовним пошуком?

3. Знаходження якого по порядку елемента в лінійній множині (першого, останнього) гарантує алгоритм прямого пошуку? Як в цьому випадку повинен бути виконаний перегляд?

4. Знаходження якого за порядком елемента в лінійній множині (першого, останнього) гарантує алгоритм бінарного пошуку? Відповідь обґрунтуйте.

Лекція 6
АЛГОРИТМИ ПОШУКУ В ТЕКСТІ

Робота в текстовому редакторі, пошукові запити в базі даних, завдання в біоінформатиці, лексичний аналіз програм вимагають ефективних алгоритмів роботи з текстом. Завдання пошуку слова в тексті використовуються в криптографії, різних розділах фізики, стисненні даних, розпізнаванні мови та інших сферах людської діяльності.

Введемо низку визначень, які будуть використовуватися далі у викладі матеріалу.

Алфавіт – кінцева множина символів.

Рядок (слово) − це послідовність символів з деякого алфавіту. **Довжина рядка** − кількість символів у рядку.

Рядок будемо позначати символами алфавіту, наприклад $X = x\ [1]\ x\ [2]\ ...\ x\ [n]$ − рядок довжиною n, де $x\ [i]$ − i -й символ рядка X, що належить алфавіту. Рядок, що не містить жодного символу, називається **порожнім.**

Рядок X називається **підрядком** рядка Y, якщо знайдуться такі рядки $Z1$ і $Z2$, що $Y = Z1\ X\ Z2$. При цьому $Z1$ називають лівим, а $Z2$ - правим крилом підрядка. Підрядком може бути і сам рядок. Іноді при цьому рядок X називають входженням в рядок Y. Наприклад, рядки *hrf* і *fhr* є підрядками рядка *abhrfhr*.

Підрядок X називається **префіксом** рядка Y, якщо є такий підрядок Z, де $Y = XZ$. Причому сам рядок є префіксом для самому собі (тому що знайдеться нульовий рядок L, де $X = XL$). Наприклад, підрядок *ab* є префіксом рядка *abcfa*.

Підрядок X називається **суфіксом** рядка Y, якщо є такий підрядок Z, де $Y = ZX$. Аналогічно, рядок є суфіксом самому собі. Наприклад, підрядок *bfg* є суфіксом рядка *vsenfbfg*.

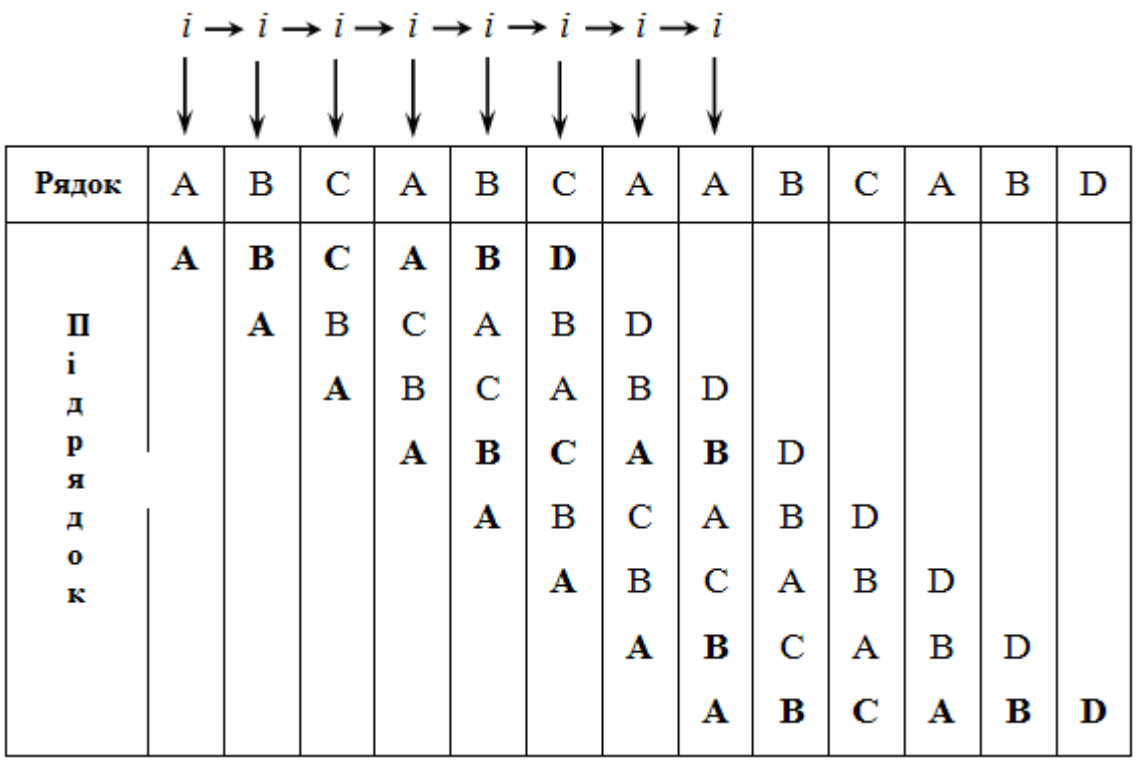
Поставимо задачу пошуку в лінійних структурах. Нехай задана множина даних, яка описується як масив, що складається з певної кількості елементів. Перевіримо, чи входить заданий ключ в даний масив. Якщо входить, то знайдемо номер цього елемента масиву, тобто визначимо перше входження заданого ключа (елемента) у вихідному масиві.

Прямий пошук

Даний алгоритм ще називається алгоритмом послідовного пошуку, він є самим простим і очевидним.

Основна ідея алгоритму прямого пошуку полягає в посимвольному порівнянні рядка з підрядком. У початковий момент відбувається порівняння першого символу рядка з першим символом підрядка, другого символу рядка з другим символом підрядка і т. д. Якщо відбувся збіг усіх символів, то фіксується факт знаходження підрядка. В іншому випадку відбувається зсування підрядка на одну позицію вправо і повторюється посимвольне порівняння, тобто порівнюється другий символ рядка з першим символом підрядка, третій символ рядка з другим символом підрядка і т. д. (рис. 6.1) Символи, які порівнюються, на рисунку виділені жирним. Розглянуті зсуви підрядка повторюються до тих пір, поки кінець підрядка не досягне кінця рядка

або не відбудеться повний збіг символів підрядка з рядком, тобто знайдеться підрядок.



Рядок	A	B	C	A	B	C	A	A	B	C	A	B	D
Підрядок	**A**	**B**	**C**	**A**	**B**	**D**							
		A	B	C	A	B	D						
			A	B	C	A	B	D					
				A	**B**	**C**	**A**	**B**	D				
					A	B	C	A	B	D			
						A	B	C	A	B	D		
							A	**B**	C	A	B	D	
								A	**B**	**C**	**A**	**B**	**D**

Рисунок 6.1 − Демонстрація алгоритму прямого пошуку

```
//опис функції прямого пошуку підрядка у рядку
int DirectSearch(char *string, char *substring){
  int sl, ssl;
  int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )
    cout << "Невірно заданий рядок\n";
  else if ( ssl == 0 )
    cout << "Невірно заданий підрядок\n";
  else
    for (int i = 0; i < sl - ssl + 1; i++)
      for (int j = 0; j < ssl; j++)
        if ( substring[j] != string[i+j] )
          break;
        else if ( j == ssl - 1 ){
          res = i;
          break;
        }
  return res;
}
```

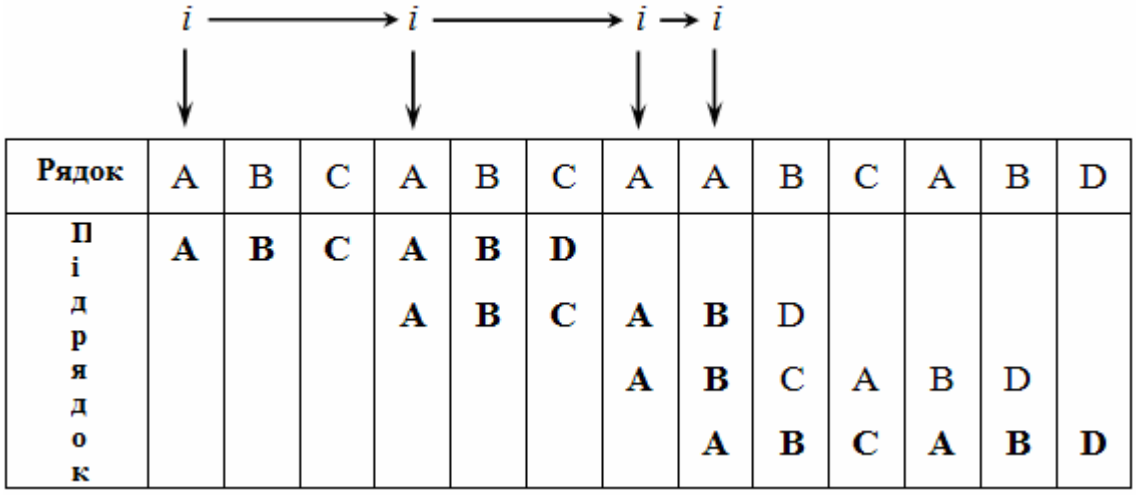
Даний алгоритм є маловитратним і не потребує попереднього обробленю в додатковому просторі. Більшість порівнянь алгоритму прямого пошуку є зайвими. Тому в найгіршому випадку алгоритм буде малоефективний, так як його час роботи буде пропорційний O((*n*–*m* + 1) *m*), де *n* і *m* − довжини рядка і подрядка відповідно.

Алгоритм Кнута, Морріса і Пратта

Розглянутий алгоритм грунтується на тому, що після часткового збігу початкової частини підрядка з відповідними символами рядка фактично відома пройдена частина рядка і можна обчислити деякі значення, за допомогою яких потім швидко просуватися по рядку.

Основною відмінністю алгоритму Кнута, Морріса і Пратта (КМП) від алгоритму прямого пошуку полягає в тому, що зсув підрядка виконується не на один символ на кожному кроці алгоритму, а на деяку змінну кількість символів. Отже, перед тим як здійснювати черговий зсув, необхідно визначити величину зсуву. Для підвищення ефективності алгоритму необхідно, щоб зсув на кожному кроці був би якомога більшим (рис. 6.2). На рис. 6.2 символи, які зазнали порівняння, виділені жирним шрифтом.

Якщо для довільного підрядка визначити всі його початкові символи, що одночасно є його кінцевими символами, і вибрати з них найдовшу послідовність (не рахуючи, звичайно, сам рядок), то таку процедуру прийнято називати обчисленням префікс-функції. У реалізації алгоритму КМП використовується передобробка шуканого підрядка, яка полягає у створенні префікс-функції на її основі. При цьому використовується наступна ідея: якщо префікс (він же суфікс) рядка довжиною *i* довше одного символу, то він одночасно є префіксом підрядка довжиною *i*-1. Таким чином, перевіряємо префікс попереднього підрядка, якщо той не підходить, то перевіряємо префікс його префікса і т.д. Діючи так, знаходимо найбільший шуканий префікс.



Рядок	A	B	C	A	B	C	A	A	B	C	A	B	D
Підрядрядок	**A**	**B**	**C**	**A** **A**	**B** **B**	**D** **C**	**A** **A**	**B** **B** **A**	D C **B**	A **C**	B **A**	D **B**	**D**

Рисунок 6.2 − Демонстрація алгоритму КМП

//опис функції алгоритму КМП

```
int KMPSearch(char *string, char *substring){
  int  sl, ssl;
  int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )
    cout << "Невірно заданий рядок\n";
  else if ( ssl == 0 )
    cout << "Невірно заданий підрядок\n";
  else {
```

```
    int  i, j = 0, k = -1;
    int  *d = new int[1000];
    d[0] = -1;
    while ( j < ssl - 1 ) {
      while ( k >= 0 && substring[j] != substring[k] )
        k = d[k];
      j++;
      k++;
      if ( substring[j] == substring[k] )
        d[j] = d[k];
      else
        d[j] = k;
    }
    i = 0;
    j = 0;
    while ( j < ssl && i < sl ){
      while ( j >= 0 && string[i] != substring[j] )
        j = d[j];
      i++;
      j++;
    }
    delete [] d;
    res =  j == ssl ? i - ssl : -1;
  }
  return res;
}
```

Точний аналіз розглянутого алгоритму є складним процесом. Д. Кнут, Д. Морріс і В. Пратт доводять, що для даного алгоритму потрібний час роботи складає близько O($m + n$), де n і m − довжини рядка і підрядока відповідно, що значно краще, ніж при прямому пошуку.

Алгоритм Бойєра і Мура

Алгоритм Бойєра і Мура вважається найбільш швидким серед алгоритмів, призначених для пошуку підрядка в рядку. Він був розроблений Р. Бойєром і Д. Муром у 1977 році. Перевага цього алгоритму в тому, що необхідно зробити деякі попередні обчислення над підрядком, щоб порівняння підрядка з вихідном рядком здійснювати не в усіх позиціях − частина перевірок пропускаються, як ті що не дають результату.

Існує безліч варіацій алгоритму Бойєра і Мура. Розглянемо найпростіший із них, який складається з наступних кроків. Спочатку будується таблиця зсувів для шуканого підрядка. Далі відбувається зсув початку рядка і підрядка і починається перевірка з останнім символом підрядка. Якщо останній символ рядка і відповідний йому при накладенні символ рядка не збігаються, підрядок зсувається щодо рядка на величину, отриману з таблиці зсувів, і знову проводиться порівняння, починаючи з останнього символу підрядка. Якщо ж символи збігаються, проводиться порівняння передостаннього символу підрядка і т.д. Якщо всі символи підрядка збіглися з накладеними символами рядка, це означає, що підрядок знайдено і пошук закінчено. Якщо ж якийсь (не останній) символ підрядка не збігається з відповідним символом рядка, далі

зсуваємо підрядок на один символ вправо і знову починаємо перевірку з останнього символу. Весь алгоритм виконується до тих пір, поки або не буде знайдено входження шуканого підрядка, або не буде досягнутий кінець рядка (рис. 6.3). На рисунку символи, які зазнали порівняння, виділені жирним шрифтом.

Величина зсуву в разі незбігу останнього символу обчислюється, виходячи з наступного: зрушення підрядка повинно бути мінімальним, таким, щоб не пропустити входження підрядка в рядок. Якщо даний символ рядка зустрічається в підрядку, то зміщуємо підрядок таким чином, щоб символ рядка збігся з найбільш правим входженням цього символу в підрядку. Якщо ж підрядок взагалі не містить цього символу, то зрушуємо підрядок на величину, що дорівнює його довжині так, що перший символ рядка накладається на наступний за символом, який перевірявся.

Величина зсуву для кожного символу підрядка залежить тільки від порядку символів у підрядку, тому зсув зручно обчислити заздалегідь і зберігати у вигляді одновимірного масиву, де кожному символу алфавіту відповідає зміщення відносно останнього символу підрядка.

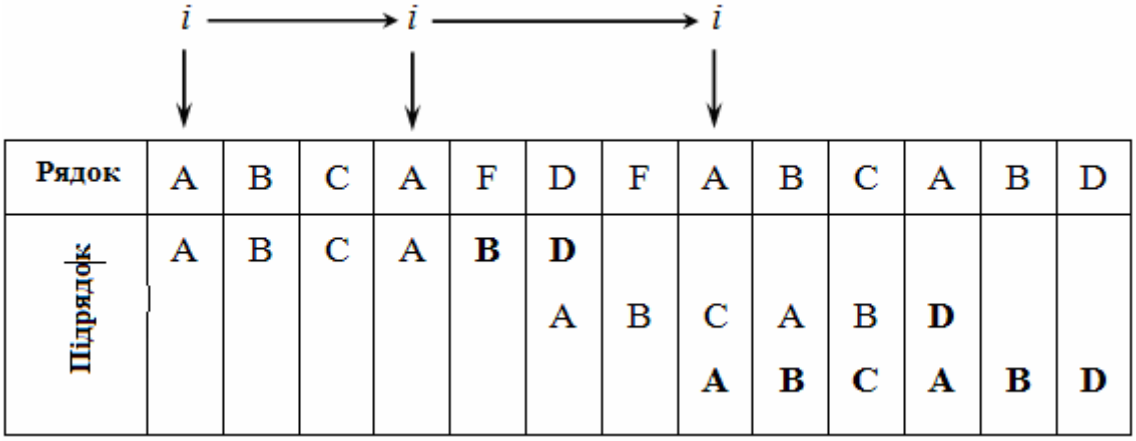


Рисунок 6.3 − Демонстрація алгоритму Бойєра і Мура

//опис функції алгоритму Бойєра і Мура

```
int BMSearch(char *string, char *substring){
  int  sl, ssl;
  int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )
    cout << "Невірно заданий рядок\n";
  else if ( ssl == 0 )
    cout << "Невірно заданий підрядок\n";
  else {
    int  i, Pos;
    int  BMT[256];
    for ( i = 0; i < 256; i ++ )
      BMT[i] = ssl;
    for ( i = ssl-1; i >= 0; i-- )
      if ( BMT[(short)(substring[i])] == ssl )
        BMT[(short)(substring[i])] = ssl - i - 1;
```

```
      Pos = ssl - 1;
      while ( Pos < sl )
        if ( substring[ssl - 1] != string[Pos] )
          Pos = Pos + BMT[(short)(string[Pos])];
        else
          for ( i = ssl - 2; i >= 0; i-- ){
         if ( substring[i] != string[Pos - ssl + i + 1] ) {
      Pos += BMT[(short)(string[Pos - ssl + i + 1])] - 1;
              break;
            }
            else
              if ( i == 0 )
                return Pos - ssl + 1;
            cout << "\t" << i << endl;
          }
    }
    return res;
  }
```

Алгоритм Бойєра і Мура на хороших даних дуже швидкий, а ймовірність появи поганих даних вкрай мала. Тому він оптимальний в більшості випадків, коли немає можливості провести попередню обробку тексту, в якому відбувається пошук. Таким чином, даний алгоритм є найбільш ефективним у звичайних ситуаціях, а його швидкодія підвищується при збільшенні підрядка або алфавіту. У найгіршому випадку час роботи даного алгоритму O($m + n$).

Контрольні питання

1. Чим можна пояснити різноманіття алгоритмів пошуку в лінійних структурах?
2. У чому переваги алгоритму пошуку з бар'єром порівнянно з алгоритмом послідовного пошуку?
3. Знаходження якого за порядком елемента у лінійній множині (першого, останнього) гарантує алгоритм прямого пошуку? Як в цьому випадку повинен бути виконаний перегляд?
4. Знаходження якого за порядком елемента у лінійній множині (першого, останнього) гарантує алгоритм бінарного пошуку? Відповідь обґрунтуйте.

Лекція 7
АЛГОРИТМИ СТИСНЕННЯ ДАНИХ

Стиснення даних – це процес, що забезпечує зменшення обсягу даних шляхом скорочення їх надмірності. Стиснення даних пов'язано з компактним розташуванням порцій даних стандартного розміру. Стиснення даних можна розділити на два основних типи:

– *стиснення без втрат (повністю оборотне)* − це метод стиснення даних, за якого раніше закодована порція даних відновлюється після їх розпакування повністю без внесення змін. Для кожного типу даних, як правило, існують свої оптимальні алгоритми стиснення без втрат.

– *стиснення з втратами* − це метод стиснення даних, за якого для забезпечення максимального ступеня стиснення вихідного масиву даних частина його даних відкидається. Для текстових, числових і табличних даних використання програм, що реалізують подібні методи стиснення, є неприйнятними. В основному такі алгоритми застосовуються для стиснення аудіо-та відео, статичних зображень.

Алгоритм стиснення даних (алгоритм архівації) − це алгоритм, який усуває *надмірність* запису даних.

Введемо низку визначень, які будуть використовуватися далі у викладанні матеріалу.

Алфавіт коду − множина всіх символів вхідного потоку. При стисненні англомовних текстів зазвичай використовують множину зі 128 ASCII кодів. При стисненні зображень множина значень пікселя може містити 2, 16, 256 або іншу кількість елементів.

Кодовий символ − найменша одиниця даних, що підлягає стисненню. Зазвичай символ − це 1 байт, але він може бути бітом, або чим-небудь ще.

Кодове слово − це послідовність кодових символів з алфавіту коду. Якщо всі слова мають однакову довжину (число символів), то такий код називається *рівномірним (фіксованої довжини)*, а якщо ж допускаються слова різної довжини, то − *нерівномірним (змінної довжини)*.

Код – повна множина слів.

Токен − *одиниця даних*, що записується у стислий потік деяким алгоритмом стиснення. Токен складається з декількох полів фіксованой або змінной довжини.

Фраза − *фрагмент даних*, що поміщається у словник для подальшого використання у стисненні.

Кодування − процес стиснення даних.

Декодування − зворотний кодуванню процес, за якого здійснюється відновлення даних.

Відношення стиснення − одна з найбільш часто використовуваних величин для позначення *ефективності методу стиснення*.

$$\text{Відношення стиснення} = \frac{\text{Розмір вихідного потоку}}{\text{Розмір вхідного потоку}}$$

Значення 0,6 означає, що дані займають 60% від початкового об'єму. Значення більше 1 означають, що вихідний потік більше вхідного (негативний стиск, або розширення).

Коефіцієнт стиснення – величина, зворотна відношенню стиснення

$$\text{Коефіцієнт стиснення} = \frac{\text{Розмір вхідного потоку}}{\text{Розмір вихідного потоку}}$$

Значення понад 1 означають стиснення, а значення менше 1 − розширення.

Середня довжина кодового слова − це величина, яка обчислюється як зважена ймовірністю сума довжин усіх кодових слів.

$$L_{\text{сер}} = p_1 L_1 + p_2 L_2 + ... + p_n L_n,$$

де p_i – імовірності кодових слів; L_1, L_2, ..., L_n – довжина кодового слова.

Існують два основних способи проведення стиснення:

- *статистичні методи* − методи стиснення, які присвоюють коди змінної довжини символам вхідного потоку, причому більш короткі коди присвоюються символам або групам символів, які мають велику ймовірність появи у вхідному потоці. Кращі статистичні методи застосовують кодування Хаффмана;
- *словникове стиснення* − це методи стиснення, що зберігають фрагменти даних у"словнику" (деяка структура даних). Якщо рядок нових даних, що надходять на вхід, ідентичний будь-якому фрагменту, який вже знаходиться у словнику, у вихідний потік поміщається покажчик на цей фрагмент. Кращим словниковим методом є метод Зіва-Лемпела.

Розглянемо кілька відомих алгоритмів стиснення даних більш докладно.

Метод Хаффмана

Цей алгоритм кодування інформації був запропонований Д.А. Хаффманом у 1952 році. Хаффманівське кодування (стиснення) − це метод стиснення, який привласнює символам алфавіту коди змінної довжини, грунтуючись на ймовірності появи цих символів.

Ідея алгоритму полягає в наступному: знаючи ймовірності входження символів у вихідний текст, можна описати процедуру побудови кодів змінної довжини, що складаються з цілої кількості бітів. Символам з більшою ймовірністю присвоюються коротші коди. Таким чином, в цьому методі при стисненні даних кожному символу присвоюється оптимальний *префіксний код*, заснований на ймовірності його появи в тексті.

Префіксний код − це код, в якому ніяке кодове слово не є префіксом будь-якого іншого кодового слова. Ці коди мають змінну довжину.

Оптимальний префіксний код − це префіксний код, який має мінімальну середню довжину.

Алгоритм Хафмана можна розділити на два етапи

Крок 1. Визначення ймовірності появи символів у початковому тексті.

Спочатку необхідно прочитати вихідний текст повністю і підрахувати ймовірності появи символів у ньому (іноді підраховують, скільки разів зустрічається кожен символ). Якщо при цьому враховуються всі 256 символів, то не буде різниці у стисканні текстового або файла іншого формату.

Крок 2. Знаходження оптимального префіксного коду.

Далі знаходяться два символи *a* і *b* з найменшими ймовірностями появи і замінюються одним фіктивним символом *x*, який має ймовірність появи, що дорівнює сумі ймовірностей появи символів *a* і *b*. Потім, використовуючи цю процедуру рекурсивно, знаходиться оптимальний префіксний код для меншої множини символів (де символи *a* і *b* замінені одним символом *x*). Код для вихідної множини символів виходить із кодів символів, які заміщаються, шляхом додавання 0 або 1 перед кодом такого символу, і ці два нових коди приймаються як коди замінюваних символів. Наприклад, код символу *a* буде відповідати коду *x* з доданим нулем перед цим кодом, а для символу *b* перед кодом символу *x* буде додана одиниця.

Коди Хаффмана мають унікальний префікс, що і дозволяє однозначно їх декодувати, незважаючи на їх змінну довжину.

Приклад 1. Програмна реалізація методу Хаффмана.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

void Expectancy();
long MinK();
void SumUp();
void BuildBits();
void OutputResult(char **Result);
void Clear();

const int MaxK = 1000;
long k[MaxK + 1], a[MaxK + 1], b[MaxK + 1];
char bits[MaxK + 1][40];
char sk[MaxK + 1];
bool Free[MaxK + 1];
char *res[256];
long i, j, n, m, kj, kk1, kk2;
char str[256];

int _tmain(int argc, _TCHAR* argv[]){
  char *BinaryCode;
  Clear();
  cout << "Введіть рядок для кодування : ";
  cin >> str;
  Expectancy();
  SumUp();
  BuildBits();
  OutputResult(&BinaryCode);
  cout << "Закодований рядок : " << endl;
  cout << BinaryCode << endl;
  system("pause");
```

```
  return 0;
}
```
//опис функції обнулення даних у масивах
```
void Clear(){
  for (i = 0; i < MaxK + 1; i++){
    k[i] = a[i] = b[i] = 0;
    sk[i] = 0;
    Free[i] = true;
    for (j = 0; j < 40; j++)
      bits[i][j] = 0;
  }
}
```
//опис функції обчислення ймовірності вхождення кожного символу у тексті
```
void Expectancy(){
  long *s = new long[256];
  for ( i = 0; i < 256; i++)
    s[i] = 0;
  for ( n = 0; n < strlen(str); n++ )
    s[str[n]]++;
  j = 0;
  for ( i = 0; i < 256; i++)
    if ( s[i] != 0 ){
      j++;
      k[j] = s[i];
      sk[j] = i;
    }
  kj = j;
}
```
//опис функції надходження мінімальної частоти символу у вхідному тексті
```
long MinK(){
  long min;
  i = 1;
  while ( !Free[i] && i < MaxK) i++;
  min = k[i];
  m = i;
  for ( i = m + 1; i <= kk2; i++ )
    if ( Free[i] && k[i] < min ){
      min = k[i];
      m = i;
    }
  Free[m] = false;
  return min;
}
```
//опис функції обліку сумарної частоти символів
```
void SumUp(){
  long s1, s2, m1, m2;
  for ( i = 1; i <= kj; i++ ){
    Free[i] = true;
    a[i] = 0;
    b[i] = 0;
  }
  kk1 = kk2 = kj;
```

```
  while (kk1 > 2){
    s1 = MinK();
    m1 = m;
    s2 = MinK();
    m2 = m;
    kk2++;
    k[kk2] = s1 + s2;
    a[kk2] = m1;
    b[kk2] = m2;
    Free[kk2] = true;
    kk1--;
  }
}
```

//опис функції формування префіксних кодів

```
void BuildBits(){
  strcpy(bits[kk2],"1");
  Free[kk2] = false;
  strcpy(bits[a[kk2]],bits[kk2]);
  strcat( bits[a[kk2]] , "0");
  strcpy(bits[b[kk2]],bits[kk2]);
  strcat( bits[b[kk2]] , "1");
  i = MinK();
  strcpy(bits[m],"0");
  Free[m] = true;
  strcpy(bits[a[m]],bits[m]);
  strcat( bits[a[m]] , "0");
  strcpy(bits[b[m]],bits[m]);
  strcat( bits[b[m]] , "1");
  for ( i = kk2 - 1; i > 0; i-- )
    if ( !Free[i] ) {
      strcpy(bits[a[i]],bits[i]);
      strcat( bits[a[i]] , "0");
      strcpy(bits[b[i]],bits[i]);
      strcat( bits[b[i]] , "1");
    }
}
```

//опис функції виводу даних

```
void OutputResult(char **Result){
  (*Result) = new char[1000];
  for (int t = 0; i < 1000 ;i++)
    (*Result)[t] = 0;
  for ( i = 1; i <= kj; i++ )
    res[sk[i]] = bits[i];
  for (i = 0; i < strlen(str); i++)
    strcat( (*Result) , res[str[i]]);
}
```

Алгоритм Хаффмана універсальний, його можна застосовувати для стиснення даних будь-яких типів, але він малоефективний для файлів малих розмірів (за рахунок необхідності збереження словника). В даний час цей метод практично не застосовується в чистому вигляді, зазвичай він використовується як один із етапів стиснення в більш складних схемах. Це єдиний алгоритм, який

не збільшує розмір вихідних даних у найгіршому випадку (якщо не брати до уваги необхідність зберігати таблицю перекодування разом з файлом).

Кодові дерева

Розглянемо реалізацію алгоритму Хаффмана з використанням кодових дерев.

Кодове дерево (дерево кодування Хаффмана, Н-дерево) − це бінарне дерево, у якого:

- листя позначені символами, для яких розробляється кодування;
- вузли (в тому числі корінь) позначені сумою ймовірностей появи всіх символів, відповідних листів піддерева, коренем якого є відповідний вузол.

Метод Хаффмана на вході отримує таблицю ймовірностей символів у початковому тексті. Далі на підставі цієї таблиці будується *дерево кодування Хаффмана.*

Алгоритм побудови дерева Хаффмана.

Крок 1. Символи вхідного алфавіту утворюють список вільних вузлів. Кожен лист має вагу, яка може дорівнювати або ймовірності, або кількості входжень символу у вхідний текст.

Крок 2. Вибираються два вільних вузли дерева з найменшими вагами.

Крок 3. Створюється їх батько з вагою, що дорівнює їх сумарній вазі.

Крок 4. Батько додається до списку вільних вузлів, а двоє його дітей видаляються з цього списку.

Крок 5. Одній дузі, що виходить з батьків, ставиться у відповідність біт 1, інший − біт 0.

Крок 6. Повторюємо кроки, починаючи з другого, до тих пір, поки у списку вільних вузлів не залишиться тільки один вільний вузол. Він і буде вважатися коренем дерева.

Існують два підходи до побудови кодового дерева: від кореня до листя і від листя до кореня.

Приклад 1. Побудова кодового дерева.

Нехай задана вхідна послідовність символів:

aabbbbbbbbccccdeeeee.

Її вхідний обсяг дорівнює 20 байтів (160 біт). Згідно з наведеними на рис. 7.1 даними (таблиця ймовірності появи символів, кодове дерево і таблиця оптимальних префіксних кодів) закодована вхідна послідовність символів буде виглядати наступним чином:

11011101000000001111111111111001010101010.

Отже, її обсяг буде дорівнює 42 біта. Коефіцієнт стиснення приблизно дорівнює 3,8.

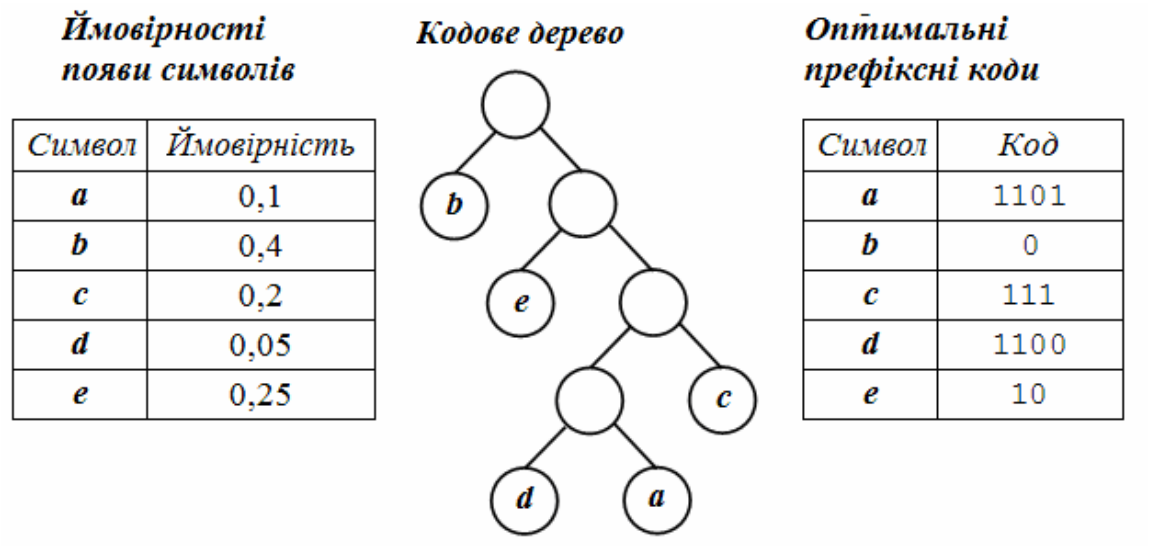


Рисунок 7.1 – Створення оптимального префіксного коду

Приклад 2. Програмна реалізація алгоритму Хаффмана за допомогою кодового дерева.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
struct sym {
      unsigned char ch;
      float freq;
      char code[255];
      sym *left;
      sym *right;
};
void Statistics(char *String);
sym *makeTree(sym *psym[],int k);
void makeCodes(sym *root);
void CodeHuffman(char *String,char *BinaryCode, sym *root);
void DecodeHuffman(char *BinaryCode,char *ReducedString,
                   sym *root);
int chh;//зміна для обліку інформації з рядка
int k=0;
//лічильник кількості різних букв
int kk=0;//лічильник кількості усіх знаків у файлі
int kolvo[256]={0};
//створюємо масив кількості унікальних символів
sym simbols[256]={0};//створюємо масив записів
sym *psym[256];//створюємо масив покажчиків на записи
float summir=0;//сума ймовірностей

int _tmain(int argc, _TCHAR* argv[]){
  char *String = new char[1000];
  char *BinaryCode = new char[1000];
  char *ReducedString = new char[1000];
  String[0] = BinaryCode[0] = ReducedString[0] = 0;
  cout << "Введіть рядок для кодування : ";
  cin >> String;
```

```
  sym *symbols = new sym[k];
  //створення динамічного масиву структур simbols
  sym **psum = new sym*[k];
  //створення динамічного масиву покажчиків на simbols
  Statistics(String);
  sym *root = makeTree(psym,k);
  //виклик функції створення дерева Хаффмана
  makeCodes(root);//виклик функції отримання коду
  CodeHuffman(String,BinaryCode,root);
  cout << "Закодований рядок : " << endl;
  cout << BinaryCode << endl;
  DecodeHuffman(BinaryCode,ReducedString, root);
  cout << "Розкодований рядок : " << endl;
  cout << ReducedString << endl;
  delete psum;
  delete String;
  delete BinaryCode;
  delete ReducedString;
  system("pause");
  return 0;
}
//рекурсивна функція створення дерева Хаффмана
sym *makeTree(sym *psym[],int k) {
  int i, j;
    sym *temp;
    temp = new sym;
    temp->freq = psym[k-1]->freq+psym[k-2]->freq;
    temp->code[0] = 0;
    temp->left = psym[k-1];
    temp->right = psym[k-2];
    if ( k == 2 )
        return temp;
    else {
    //внесення в потрібне місце масиву елемента дерева Хаффмана
      for ( i = 0; i < k; i++)
        if ( temp->freq > psym[i]->freq ) {
          for( j = k - 1; j > i; j--)
            psym[j] = psym[j-1];
            psym[i] = temp;
            break;
        }
    }
  return makeTree(psym,k-1);
}
//рекурсивна функція кодування дерева
void makeCodes(sym *root) {
  if ( root->left ) {
    strcpy(root->left->code,root->code);
    strcat(root->left->code,"0");
    makeCodes(root->left);
  }
  if ( root->right ) {
    strcpy(root->right->code,root->code);
```

```
    strcat(root->right->code,"1");
    makeCodes(root->right);
  }
}
//функція обліку кількості кожного символу та його ймовірності
void Statistics(char *String){
  int i, j;
  //побайтно зчитуємо рядок та складаємо таблицу ймовірностей
  for ( i = 0; i < strlen(String); i++){
    chh = String[i];
    for ( j = 0; j < 256; j++){
      if (chh==simbols[j].ch) {
        kolvo[j]++;
        kk++;
        break;
      }
      if (simbols[j].ch==0){
        simbols[j].ch=(unsigned char)chh;
        kolvo[j]=1;
        k++; kk++;
        break;
      }
    }
  }
  // розрахунок ймовірності
  for ( i = 0; i < k; i++)
    simbols[i].freq = (float)kolvo[i] / kk;
  // у масив покажчиків вносимо адресу записів
  for ( i = 0; i < k; i++)
    psym[i] = &simbols[i];
  //сортування за спаданням
  sym tempp;
  for ( i = 1; i < k; i++)
  for ( j = 0; j < k - 1; j++)
    if ( simbols[j].freq < simbols[j+1].freq ){
      tempp = simbols[j];
      simbols[j] = simbols[j+1];
      simbols[j+1] = tempp;
    }
  for( i=0;i<k;i++)  {
    summir+=simbols[i].freq;
    printf("Ch= %d\tFreq= %f\tPPP= %c\t\n",simbols[i].ch,
            simbols[i].freq,psym[i]->ch,i);
  }
  printf("\n Slova = %d\tSummir=%f\n",kk,summir);
}
//функція кодування рядка
void CodeHuffman(char *String,char *BinaryCode, sym *root){
  for (int  i = 0; i < strlen(String); i++){
    chh = String[i];
    for (int  j = 0; j < k; j++)
      if ( chh == simbols[j].ch ){
        strcat(BinaryCode,simbols[j].code);
```

```
      }
  }
}
//функція декодування рядка
void DecodeHuffman(char *BinaryCode,char *ReducedString,
                   sym *root){
  sym *Current;// покажчик у дереві
  char CurrentBit;// значеня поточного біта кода
  int BitNumber;
  int CurrentSimbol;// індекс символу, який розпаковується
  bool FlagOfEnd;  // прапорець кінця бітової послідовності
  FlagOfEnd = false;
  CurrentSimbol = 0;
  BitNumber = 0;
  Current = root;
  //доки не закінчилась бітова послідовність
  while ( BitNumber != strlen(BinaryCode) ) {
    //доки не прийшли до листів дерева
    while (Current->left != NULL && Current->right != NULL &&
           BitNumber != strlen(BinaryCode) ) {
      //читаємо значеня чергового біта
      CurrentBit = BinaryCode[BitNumber++];
      //біт - 0, то йдемо ліворуч, біт - 1, то праворуч
      if ( CurrentBit == '0' )
        Current = Current->left;
      else
        Current = Current->right;
    }
    //переходимо у листок та формуємо черговий символ
    ReducedString[CurrentSimbol++] = Current->ch;
    Current = root;
  }
  ReducedString[CurrentSimbol] = 0;
}
```

Контрольні питання

1. При кодуванні яких даних можна використовувати стиснення з втратами? Відповідь обґрунтуйте.

2. У чому переваги та недоліки статичних методів і словникового стиснення?

3. Яким чином кодування за алгоритмом Хаффмана через префіксний код гарантує мінімальну довжину коду?

4. За рахунок чого в методі Хаффмана підтримується однозначність відповідності коду кодованого символу?

5. Чому алгоритм Хаффмана малоефективний для файлів малих розмірів?

6. Виконайте кодування за методом Хаффмана через префіксний код символів, які зустрічаються з вірогідністю 0,3; 0,2; 0,1; 0,1; 0,1; 0,05; 0,05; 0,04; 0,03; 0,03. Порівняйте отриманий результат з даними програмної реалізації.

7. Доведіть, що метод Хаффмана кодує інформацію без втрат.

Лекція 8
АЛГОРИТМИ СОРТУВАННЯ МАСИВІВ

Внутрішнє сортування

Алгоритмом сортування називається алгоритм для упорядкування деякої множини елементів. Зазвичай під алгоритмом сортування мають на увазі алгоритм упорядкування множини елементів за зростанням або спаданням.

В алгоритмах сортування лише частина даних використовується як ключ сортування. *Ключем сортування* називається атрибут (або кілька атрибутів), за значенням якого визначається порядок елементів. Таким чином, при написанні алгоритмів сортування масивів слід врахувати, що ключ повністю або частково збігається з даними.

Практично кожен алгоритм сортування можна подылити на три частини:

– порівняння, що визначає впорядкованість пари елементів;

– перестановку, що міняє місцями пару елементів;

– власне алгоритм сортування, який здійснює порівняння і перестановку елементів до тих пір, поки всі елементи множини не будуть упорядковані.

Класифікація алгоритмів сортування

Все різноманіття алгоритмів сортування можна класифікувати за різними ознаками, наприклад, стійкістю, поведінкою, використанням операцій порівняння, потреби у додатковій пам'яті, потреби в знаннях про структуру даних, що виходять за рамки операції порівняння, та інші.

В даному випадку основні типи алгоритмів впорядкування діляться наступним чином:

– *внутрішнє сортування* − це алгоритм сортування, який в процесі упорядкування даних використовує тільки оперативну пам'ять (ОЗП) на комп'ютері. Залежно від конкретного алгоритму і його реалізації дані можуть сортуватися в тій самій області пам'яті, або використовувати додаткову оперативну пам'ять;

– *зовнішнє сортування* − це алгоритм сортування, який при проведенні упорядкування даних використовує зовнішню пам'ять, як правило, жорсткі диски. Зовнішнє сортування розроблено для обробки великих списків даних, які не поміщаються в оперативну пам'ять. Внутрішнє сортування є базою для будь-якого алгоритму зовнішнього сортування − окремі частини масиву даних упорядковано в оперативній пам'яті і за допомогою спеціального алгоритму зчіплюються в один масив, впорядкований за ключем.

Бінарне сортування пірамідою

Даний метод сортування був запропонований Дж. У. Дж. Вільямсом і Р.У. Флойдом в 1964 році. Сортування пірамідаю в деякому роді є модифікацією такого підходу, як сортування вибором, з тією лише відмінністю, що мінімальний (або максимальний) елемент з невідсортованої послідовності вибирається за меншу кількість операцій. Для такого швидкого вибору з цієї невідсортованої послідовності будується деяка структура. Саме суть даного методу і полягає в побудові такої структури, яка називається пірамідою.

Піраміда (сортувальне дерево, бінарна купа) − бінарне дерево з впорядкованим листям (корінь дерева − найменший чи найбільший елемент). Піраміду можна уявити у вигляді масиву. Перший елемент піраміди є найменшим або найбільшим, що залежить від ключа сортування.

Просіювання − це побудова нової піраміди за наступним алгоритмом: новий елемент поміщається у вершину дерева, далі він переміщається ("просівається") по шляху вниз на основі порівняння з дочірніми елементами. Спуск завершується, якщо результат порівняння з дочірніми елементами відповідає ключу сортування.

Послідовність чисел $x_i, x_{i+1}, ..., x_n$ формує *піраміду*, якщо для всіх $k = i, i+1, ..., n/2$ виконуються нерівності $x_k > x_{2k}$, $x_k > x_i$ (або $x_k < x_{2k}$, $x_k < x_{2k+1}$). Елементи x_{2i} та x_{2i+1} називаються нащадками елемента x_i.

Масив чисел 12 10 7 5 8 7 3 є пірамідою. Такий масив зручно зображувати у вигляді дерева. Перший елемент масиву, елементи якого утворюють собою піраміду, є найбільшим (або найменшим). Якщо масив показаний у вигляді піраміди, то масив легко впорядкувати.

Алгоритм пірамідального сортування.

Крок 1. Перетворити масив у піраміду (рис. 8.1 A).

Крок 2. Використовувати алгоритм сортування піраміди (рис. 8.1 B).

Алгоритм перетворення масиву в піраміду (побудова піраміди).

Нехай дано масив $x[1], x[2], ..., x[n]$.

Крок 1. Встановлюємо $k = n/2$.

Крок 2. Перебираємо елементи масиву в циклі справа наліво для $i = k, k–1, ...,1$. Якщо нерівності $x_i > x_{2i}$, $x_i > x_{2i+1}$ не виконуються, то повторюємо перестановки x_i з найбільшим з нащадків. Перестановки завершуються при виконанні нерівностей $x_i > x_{2i}$, $x > x_{2i+1}$.

Алгоритм сортування піраміди.

Розглянемо масив розмірності n, який представляє піраміду $x[1], x[2], ..., x[n]$.

Крок 1. Переставляємо елементи $x[1]$ і $x[n]$.

Крок 2. Визначаємо $n = n–1$. Це еквівалентно тому, що у масиві з подальшого розгляду виключається елемент *x [n]*.

Крок 3. Розглядаємо масив $x[1], x[2], ..., x[n–1]$, який виходить з вихідного за рахунок виключення останнього елемента. Даний масив через перестановки елементів вже не є пірамідою. Але такий масив легко перетворити в піраміду. Це досягається повторенням перестановки значення елемента з $x[1]$ з найбільшим із нащадків. Така перестановка триває до тих пір, поки елемент з $x[1]$ не опиниться на місці елемента $x[i]$ і при цьому будуть виконуватися нерівності $x[i] > x[2i]$, $x[i] > x[2i + 1]$. Тим самим визначається нове місце для значення першого елемента з $x[1]$ (рис. 8.1 C).

Крок 4. Повторюємо кроки 2, 3, 4 до тих пір, поки не отримаємо $n = 1$. Довільний масив можна перетворити в піраміду (рис. 8.1 D – H).

Побудова піраміди, її сортування і "просіювання" елементів реалізуються за допомогою рекурсії. Базою рекурсії при цьому виступає піраміда з одного

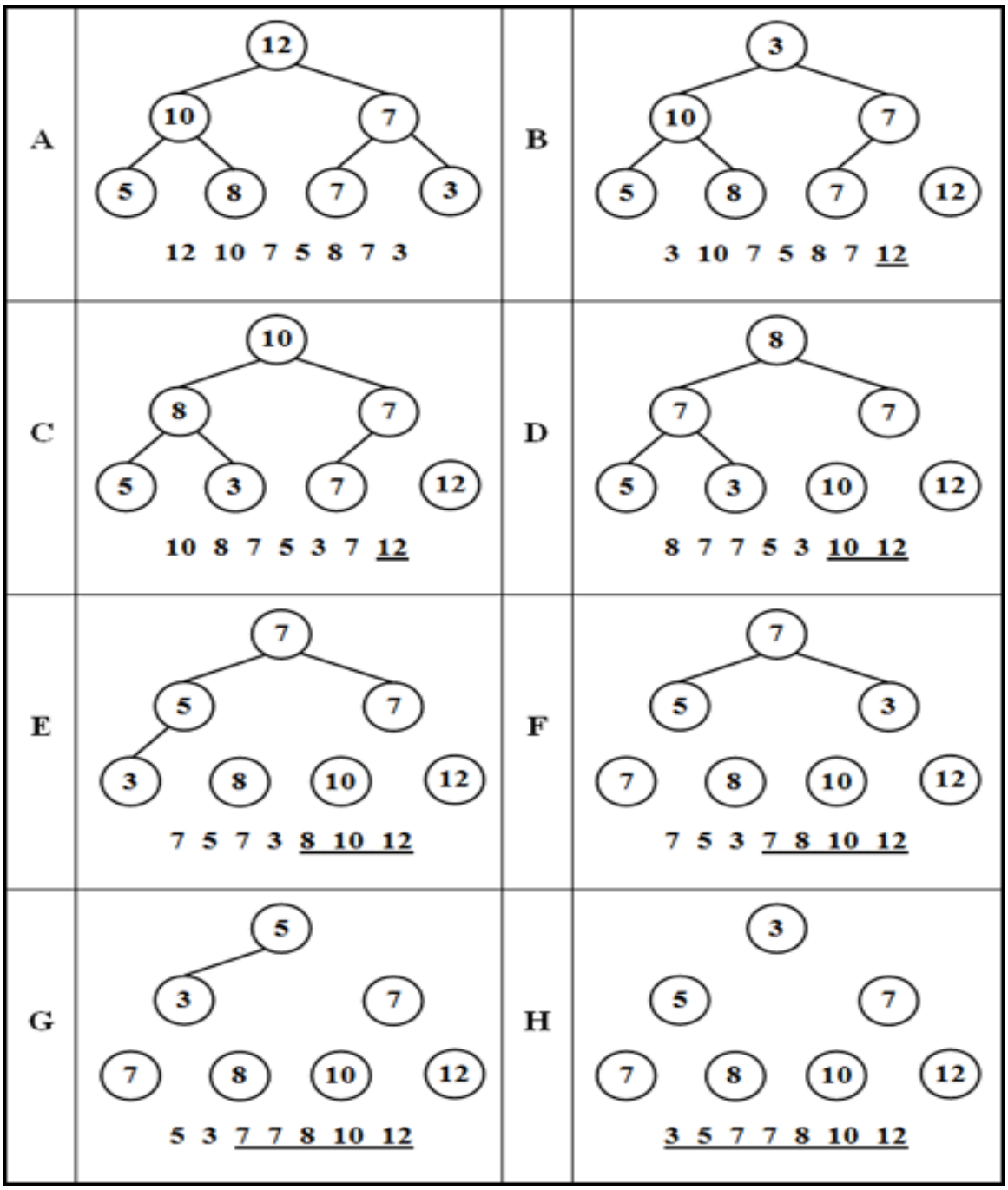


Рисунок 8.1 − Демонстрація бінарного сортування пірамідою за спаданням

елемента, а сортування і просіювання елементів зводяться за допомогою декомпозиції до аналогічних дій з пірамідою з n–1 елемента.

// Опис функції бінарного сортування пірамідою

```
void Binary_Pyramidal_Sort (int k,int *x){
  Build_Pyramid(k/2+1,k-1,x);
  Sort_Piramid(k-1,x);
}
//побудова піраміди
void Build_Pyramid (int k, int r, int *x){
  Sifting(k,r,x);
  if (k > 0)
    Build_Pyramid(k-1,r,x);
}
```

```
//сортування піраміди
void Sort_Piramid (int k, int *x){
  Exchange (0,k,x);
  Sifting(0,k-1,x);
  if (k > 1)
    Sort_Piramid(k-1,x);
}

//"просіювання" елементів
void Sifting (int left, int right, int *x){
  int q, p, h;
  q=2*left+1;
  p=q+1;
  if (q <= right){
    if (p <= right && x[p] > x[q])
      q = p;
    if (x[left] < x[q]){
      Exchange (left, q, x);
      Sifting(q, right, x);
  }
 }
}

//процедура обміну двох елементів
void Exchange (int i, int j, int *x){
  int tmp;
  tmp = x[i];
  x[i] = x[j];
  x[j] = tmp;
}
```

Сортування методом Шелла

Сортування Шелла було названо в честь його винахідника – Дональда Шелла, який опублікував цей алгоритм у 1959 році. Загальна ідея сортування Шелла полягає в порівнянні на початкових стадіях сортування пар значень, розташованих досить далеко один від одного упорядкованому наборі даних. Така модифікація методу сортування дозволяє швидко переставляти далекі невпорядковані пари значень (сортування таких пар зазвичай вимагає значної кількості перестановок, якщо використовується порівняння тільки сусідніх елементів).

Загальна схема методу полягає в наступному.

Крок 1. Відбувається упорядкування елементів $n/2$ пар $(x_i, x_{i/2+1})$ для $1 < i < n/2$.

Крок 2. Упорядковуються елементи в $n/4$ групах з чотирьох елементів $(x_i, x_{i/4+i}, x_{i/2+i}, x_{3i/4+i})$ для $1 < i < n/4$.

Крок 3. Упорядковуються елементи вже в $n/4$ групах з восьми елементів і т.д.

На останньому кроці упорядковуються елементи відразу усьому масиві x_1, x_2, ..., x_n. На кожному кроці для упорядкування елементів у групах використовується метод сортування вставками (рис. 8.2).

//опис функції сортування Шелла

```
void Shell_Sort (int n, int *x){
  int h, i, j;
  for (h = n/2 ; h > 0 ; h = h/2)
    for (i = 0 ; i < n-h ; i++)
      for (j = i ; j >= 0 ; j = j - h)
        if (x[j] > x[j+h])
          Exchange (j, j+h, x);
        else j = 0;
}
//процедура обміну двох елементів
void Exchange (int i, int j, int *x){
  int tmp;
  tmp = x[i];
  x[i] = x[j];
  x[j] = tmp;
}
```

У даний час невідома послідовність h_i, h_{i-1}, h_{i-2}, ..., h_1, *оптимальність* якої доведена. Для досить великих масивів рекомендованою є такая послідовність, що $h_{i+1}=3h_i+1$, а $h_1=1$. Процес починається з h_m, что $h_m > [n/9]$. Іноді *значення h* обчислюють простіше: $h_{i+1}=h_i/2, h_1=1, h_m=n/2$. Це спрощене обчислення *h* і будемо застосовувати далі.

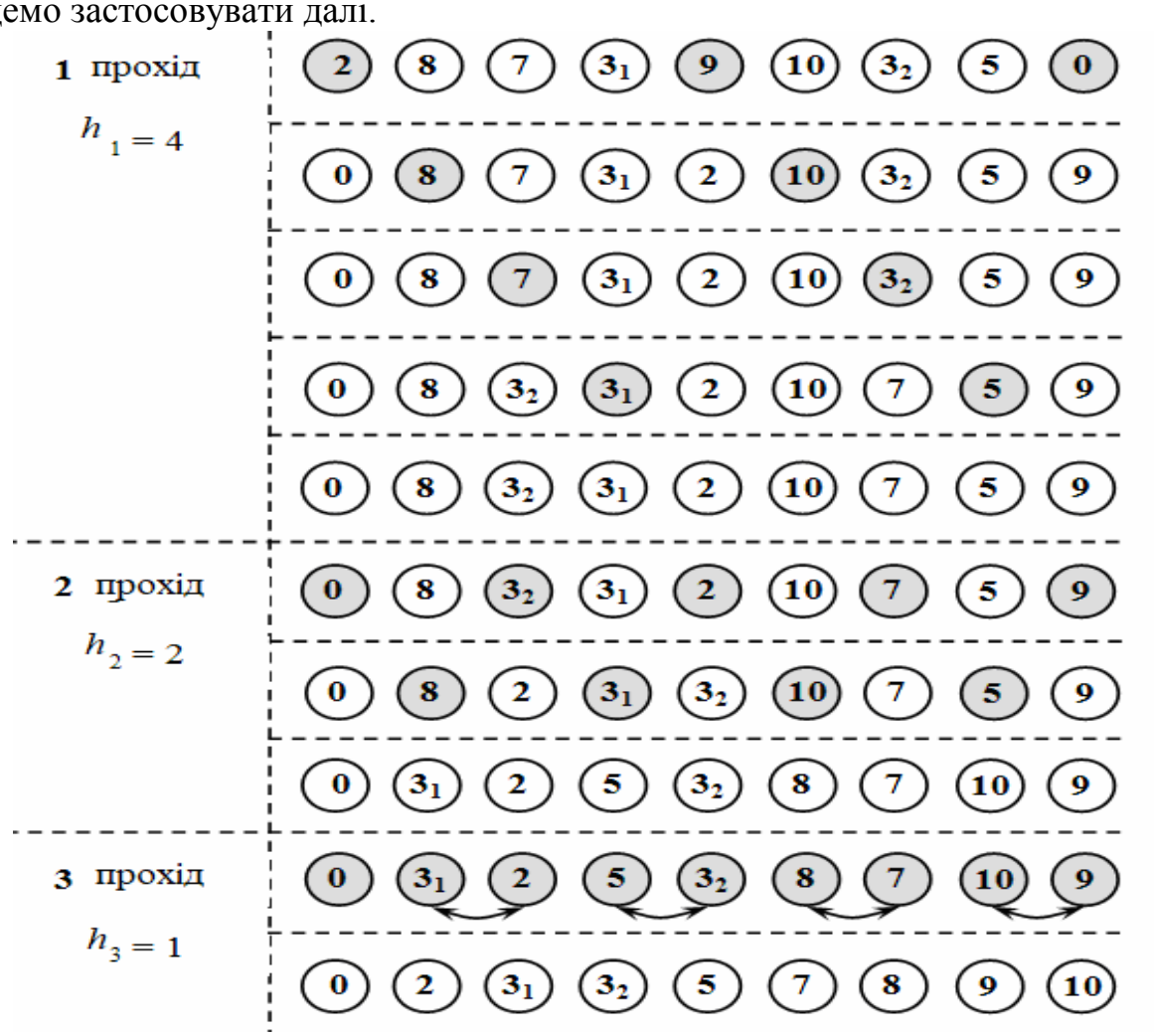


Рисунок 8.2 – Демонстрація сортування за спаданням методом Шелла

Ефективність методу Шелла пояснюється тим, що зсунуті елементи швидко потрапляють на потрібні місця. Середній час для сортування Шелла дорівнює $O(n^{1,25})$, для найгіршого випадку оцінкою є $O(n^{1,5})$.

Швидке сортування Хоара

Метод швидкого сортування був вперше описаний Ч.А.Р. Хоаром у 1962 році. *Швидке сортування* – це загальна назва низки алгоритмів, які відображають різні підходи до отримання критичного параметра, що впливає на продуктивність методу.

Опорним (провідним) елементом називається певний елемент масиву, який обирається певний чином. З точки зору коректності алгоритму вибір опорного елемента байдужий. З точки зору підвищення *ефективності алгоритму* обиратися повинна медіана, але без додаткових відомостей про дані, які упорядковуються, її зазвичай неможливо отримати. Необхідно обирати постійно один і той самий елемент (наприклад, середній або останній за положенням) або обирати елемент із випадково обраним індексом.

Алгоритм швидкого сортування Хоара

Нехай дано масив *x [n]* розмірності *n*.

Крок 1. Обирається опорний елемент масиву.

Крок 2. Масив розбивається на два – лівий і правий – щодо опорного елемента. Реорганізуємо масив таким чином, щоб всі елементи, менші опорного елемента, знаходилися ліворуч від нього, а всі елементи, більше опорного − праворуч від нього.

Крок 3. Далі повторюється крок 2 для кожного з двох новостворених масивів. Кожен раз при повторенні перетворення чергова частина масиву розбивається на дві менших і т. д., поки не вийде масив з двох елементів (рис. 8.3).

Оберемо в якості опорного елемента, який розташований на середній позиції.

//опис функції сортування Хоара

```
void Hoar_Sort (int k, int *x){
  Quick_Sort (0, k-1, x);
}
void Quick_Sort(int left, int right, int *x){
  int i, j, m, h;
  i = left;
  j = right;
  m = x[(i+j+1)/2];
  do {
    while (x[i] < m) i++;
    while (x[j] > m) j--;
    if (i <= j) {
      Exchange(i,j,x);
      i++;
      j--;
    }
  } while(i <= j);
```

```
  if (left < j)
    Quick_Sort (left, j, x);
  if (i < right)
    Quick_Sort (i, right, x); }

//процедура обміну двох елементів
void Exchange (int i, int j, int *x){
  int tmp;
  tmp = x[i];
  x[i] = x[j];
  x[j] = tmp;
   }
```

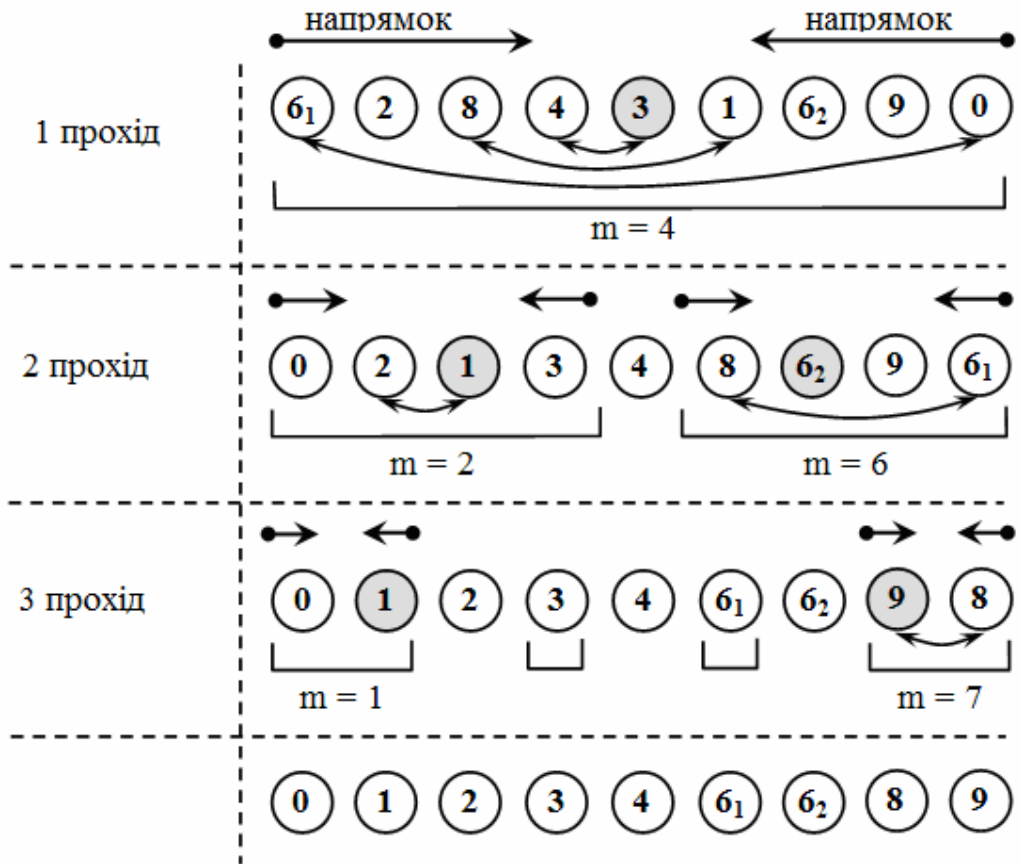


Рисунок 8.3 – Демонстрація швидкого сортування Хоара за спаданням

Ефективність швидкого сортування значною мірою визначається правильністю вибору опорних (ведучих) елементів при формуванні блоків. У найгіршому випадку час роботи алгоритму той самий, що і в алгоритмі бульбашкового сортування, тобто порядку $O(n^2)$. При оптимальному виборі провідних елементів, коли поділ кожного блока відбувається на рівні за розміром частини, час роботи алгоритму збігається з часом роботи найбільш ефективних способів сортування, тобто порядку $O(n\log n)$. В середньому випадку час роботи пропорційний $O(1{,}4n\log n)$.

Сортування злиттям

Алгоритм сортування злиттям був створенний Джоном фон Нейманом у 1945 році. Він є одним з найшвидших способів сортування.

Злиття − це об'єднання двох або більше впорядкованих масивів в один упорядкований.

Сортування злиттям є одним з найпростіших алгоритмів сортування (серед швидких алгоритмів). Особливістю цього алгоритму є те, що він працює з елементами масиву переважно послідовно, завдяки чому саме цей алгоритм використовується при сортуванні у системах з різними апаратними обмеженнями (наприклад, при сортуванні даних на жорсткому диску). Крім того, сортування злиттям є алгоритмом, який може бути ефективно використаний для сортування таких структур даних, як пов'язані списки.

Даний алгоритм застосовується тоді, коли є можливість використовувати для зберігання проміжних результатів пам'ять, яку можна порівняти з розміром вихідного масиву. Він побудований за принципом "розділяй і володарюй". Спочатку завдання розбивається на кілька підзадач меншого розміру. Потім ці завдання вирішуються за допомогою рекурсивного виклику або безпосередньо, якщо їх розмір досить малий. Далі їх рішення комбінуються, і виходить рішення вихідного завдання (рис. 8.4).

Процедура злиття вимагає два відсортованих масиви. Зауважимо, що масив з одного елемента за визначенням є відсортованим.

Алгоритм сортування злиттям

Крок 1. Розбити наявні елементи масиву на пари і здійснити злиття елементів кожної пари, отримавши відсортовані ланцюжки довжиною 2 (крім, можливо, одного елемента, для якого не знайшлося пари).

Крок 2. Розбити наявні відсортовані ланцюжки на пари, і здійснити злиття ланцюжків кожної пари.

Крок 3. Якщо число відсортованих ланцюжків більше одиниці, перейти до кроку 2.

//опис функції сортування злиттям

```
void Merging_Sort (int n, int *x){
  int i, j, k, t, s, Fin1, Fin2;
  int* tmp = new int[n];
  k = 1;
  while (k < n){
    t = 0;
    s = 0;
while (t+k < n){
      Fin1 = t+k;
      Fin2 = (t+2*k < n ? t+2*k : n);
      i = t;
      j = Fin1;
      for ( ; i < Fin1 && j < Fin2 ; s++){
        if (x[i] < x[j]) {
          tmp[s] = x[i];
          i++;
        }
        else {
          tmp[s] = x[j];
          j++;
        }
```

```
      }
      for ( ; i < Fin1; i++, s++)
        tmp[s] = x[i];
      for ( ; j < Fin2; j++, s++)
        tmp[s] = x[j];
      t = Fin2;
    }
    k *= 2;
    for (s = 0; s < t; s++)
         x[s] = tmp[s];
  }
  delete(tmp);
}
```

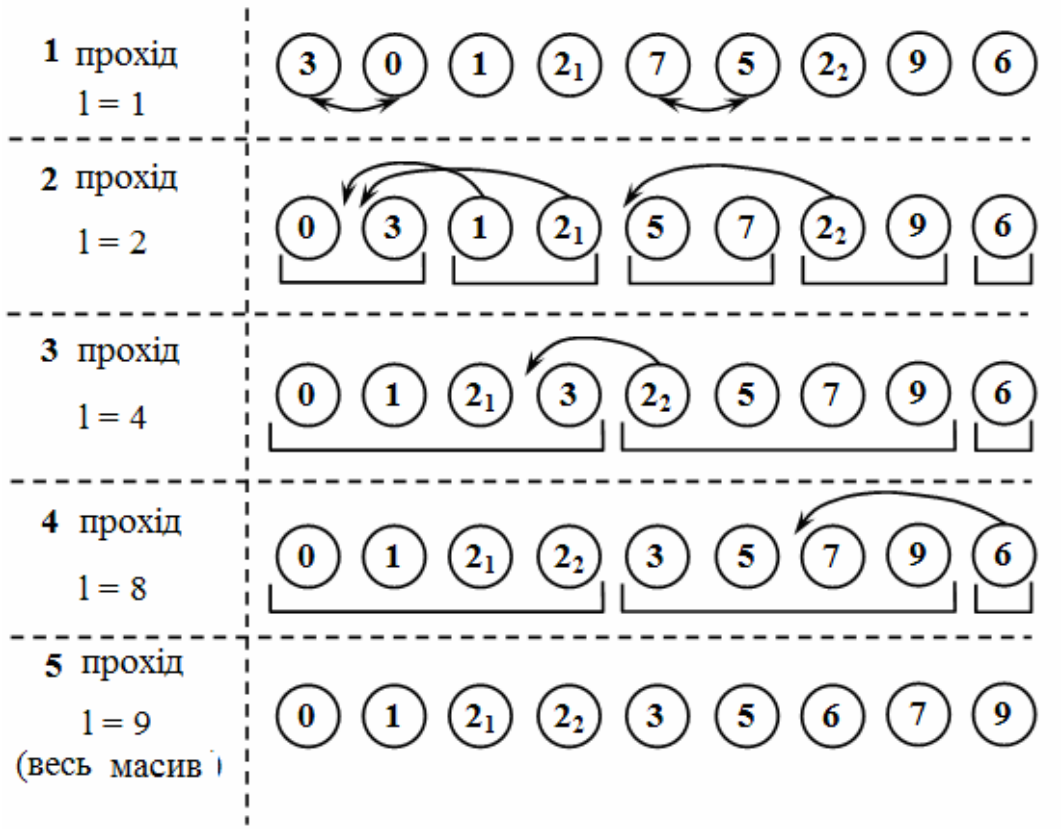


Рисунок 8.4 – Демонстрація сортування злиттям за спаданням

Недолік алгоритму полягає в тому, що він вимагає додаткової пам'яті розміром порядку n (для зберігання допоміжного масиву). Крім того, він не гарантує збереження порядку елементів з однаковими значеннями. Але його час роботи завжди пропорційний $O(n\log n)$.

Контрольні питання

1. Чим можна пояснити різноманіття алгоритмів сортування?
2. Чому на даний момент не існує універсального алгоритму сортування?
3. Як дотримання властивостей стійкості і природності впливає на час роботи алгоритму сортування?

4. За рахунок чого в алгоритмах швидкого сортування відбувається виграш при виконанні операцій порівняння і перестановок?

5. Які з перерахованих алгоритмів найбільш ефективні на майже відсортованих масивах: бінарне пірамідальне сортування, сортування злиттям, сортування Шелла і сортування Хоара? За рахунок чого відбувається виграш?

6. Чому алгоритми швидкого сортування не дають великого виграшу при малих розмірах масивів?

7. У чому переваги і недоліки по відношенню один до одного наступних алгоритмів сортувань: бінарне пірамідальне сортування, сортування злиттям, сортування Шелла і сортування Хоара?

8. Як визначити, яким алгоритмом сортування віддати перевагу при розв’язанні задачі?

Лекція 9
ЗОВНІШНЄ СОРТУВАННЯ

Зовнішні сортування застосовуються до даних, які зберігаються у зовнішній пам'яті. При виконанні такого сортування потрібно працювати з даними, розташованими на *зовнішніх пристроях послідовного доступу*. Для файлів, розташованих на таких пристроях в кожен момент часу доступний тільки один компонент послідовності даних, що є істотним обмеженням порівнянно з сортуванням масивів, де завжди доступний кожен елемент.

Зовнішнє сортування − це сортування даних, які розташовані на зовнішніх пристроях і не вміщаються в оперативну пам'ять.

До найбільш відомих алгоритмів зовнішнього сортування відносяться:

- *сортування злиттям* (просте злиття і природне злиття);
- *поліпшене сортування* (багатофазне сортування та каскадне сортування).

З зазначених зовнішніх сортувань найбільш важливим є метод сортування за допомогою злиття. Перш ніж описувати алгоритм сортування злиттям введемо кілька визначень.

Основним поняттям при використанні зовнішнього сортування є поняття серії. *Серія (упорядкований відрізок)* − це послідовність елементів, яка впорядкована за ключем.

Кількість елементів у серії називається *довжиною серії*. Серія, що складається з одного елемента, впорядкована завжди. Остання серія може мати довжину меншу, ніж решта серії файлів. Максимальна кількість серій у файлі N (всі елементи невпорядковані). Мінімальна кількість серій одна (всі елементи впорядковані).

В основі більшості методів зовнішнього сортування лежить процедура злиття і процедура розподілу. *Злиття* − це процес об'єднання двох (або більше) впорядкованих серій в одну впорядковану послідовність за допомогою циклічного вибору елементів, доступних в даний момент. *Розподіл* − це процес розподілу впорядкованих серій на два і кілька допоміжних файлів.

Фаза − це дія за одноразовим обробленням усієї послідовності елементів. *Двофазне сортування* − це сортування, в якому окремо реалізується дві фази: розподіл і злиття. *Однофазне сортування* − це сортування, в якому об'єднані фази розподілу і злиття в одну.

Двошляховим злиттям називається сортування, в якому дані розподіляються на два допоміжних файли. *Багатошляховим злиттям* називається сортування, в якому дані розподіляються на N ($N > 2$) допоміжних файлів.

Загальний алгоритм сортування злиттям

Спочатку серії розподіляються на два або більше допоміжних файлів. Даний розподіл відбувається за чергою: перша серія записується в перший допоміжний файл, друга − в другій і так далі до останнього допоміжного файла. Потім знову запис серії починається в перший допоміжний файл. Після розподілу всіх серій, вони об'єднуються в довші впорядковані відрізки, тобто з

кожного допоміжного файла береться по одній серії, які зливаються. Якщо в якомусь файлі серія закінчується, то перехід до наступної серії не відбувається. Залежно від виду сортування сформована довша впорядкована серія записується або у вихідний файл, або в один із допоміжних файлів. Після того як всі серії з усіх допоміжних файлів об'єднані в нові серії, потім знову починається їх розподіл. І так до тих пір, поки всі дані не будуть відсортовані.

Виділимо *основні характеристики сортування злиттям*:

- кількість фаз в реалізації сортування;
- кількість допоміжних файлів, на які розподіляються серії.

Розглянемо основні і найбільш важливі алгоритми зовнішніх групувань більш докладно.

Сортування простим злиттям

Один із алгоритмів сортування на основі злиття називається *простим злиттям.*

Алгоритм сортування простим злиттям є найпростішим алгоритмом зовнішнього сортування, заснований на процедурі злиття серією.

В даному алгоритмі довжина серій фіксується на кожному кроці. У вихідному файлі всі серії мають довжину 1, після першого кроку вона дорівнює 2, після другого − 4, після третього − 8, після k -го кроку – 2^k.

Алгоритм сортування простим злиттям

Крок 1. Вихідний файл *f* розбивається на два допоміжних файлів *f*1 і *f*2.

Крок 2. Допоміжні файли *f*1 і *f*2 зливаються у файл *f*, при цьому поодинокі елементи утворюють впорядковані пари.

Крок 3. Отриманий файл *f* знову обробляється, як зазначено в кроках 1 і 2. При цьому впорядковані пари переходять у впорядковані четвірки.

Крок 4. Повторюючи кроки, зливаємо четвірки у вісімки і т.д., кожен раз подвоюючи довжину злитих послідовностей до тих пір, поки не буде впорядкований цілком весь файл (рис. 9.1).

Вихідний файл ***f*: 5 7 3 2 8 4 1**

	Розподіл	*Злиття*
1 прохід	*f1*: 5 3 8 1 *f2*: 7 2 4	*f*: 5 7 2 3 4 8 1
2 прохід	*f1*: 5 7 4 8 *f2*: 2 3 1	*f*: 2 3 5 7 1 4 8
3 прохід	*f1*: 2 3 5 7 *f2*: 1 4 8	*f*: 1 2 3 4 5 7 8

Рисунок 9.1 − Демонстрація сортування двошляховим двофазним простим злиттям

Після виконання *i* проходів отримуємо два файли, що складаються з серій довжини 2^i. Закінчення процесу відбувається при виконанні умови $2^i >= n$. Отже, процес сортування простим злиттям вимагає порядку O (*log n*) проходів за даними.

Ознаками кінця сортування простим злиттям є наступні умови:

– довжина серії не менше кількості елементів у файлі (визначається після фази злиття);

– кількість серій дорівнює 1 (визначається на фазі злиття).

– при однофазному сортуванні другий за рахунком допоміжний файл після розподілу серій залишився порожнім.

//опис функції сортування простим злиттям

```
void Simple_Merging_Sort (char *name){
  int a1, a2, k, i, j, kol, tmp;
  FILE *f, *f1, *f2;
  kol = 0;
  if ( (f = fopen(name,"r")) == NULL )
    printf("\n Вихідний файл не може бути прочитаний...");
  else {
    while ( !feof(f) ) {
      fscanf(f,"%d",&a1);
      kol++;
    }
    fclose(f);
  }
  k = 1;
  while ( k < kol ){
    f = fopen(name,"r");
    f1 = fopen("smsort_1","w");
    f2 = fopen("smsort_2","w");
    if ( !feof(f) ) fscanf(f,"%d",&a1);
    while ( !feof(f) ){
      for ( i = 0; i < k && !feof(f) ; i++ ){
        fprintf(f1,"%d ",a1);
        fscanf(f,"%d",&a1);
      }
      for ( j = 0; j < k && !feof(f) ; j++ ){
        fprintf(f2,"%d ",a1);
        fscanf(f,"%d",&a1);
      }
    }
    fclose(f2);    fclose(f1);    fclose(f);

    f = fopen(name,"w");
    f1 = fopen("smsort_1","r");
    f2 = fopen("smsort_2","r");
    if ( !feof(f1) ) fscanf(f1,"%d",&a1);
    if ( !feof(f2) ) fscanf(f2,"%d",&a2);
    while ( !feof(f1) && !feof(f2) ){
      i = 0;
      j = 0;
      while ( i < k && j < k && !feof(f1) && !feof(f2) ) {
```

```
        if ( a1 < a2 ) {
          fprintf(f,"%d ",a1);
          fscanf(f1,"%d",&a1);
          i++;
        }
        else {
          fprintf(f,"%d ",a2);
          fscanf(f2,"%d",&a2);
          j++;
        }
      }
      while ( i < k && !feof(f1) ) {
        fprintf(f,"%d ",a1);
        fscanf(f1,"%d",&a1);
        i++;
      }
      while ( j < k && !feof(f2) ) {
        fprintf(f,"%d ",a2);
        fscanf(f2,"%d",&a2);
        j++;
      }
    }
    while ( !feof(f1) ) {
      fprintf(f,"%d ",a1);
      fscanf(f1,"%d",&a1);
    }
    while ( !feof(f2) ) {
      fprintf(f,"%d ",a2);
      fscanf(f2,"%d",&a2);
    }
    fclose(f2);
    fclose(f1);
    fclose(f);
    k *= 2;
  }
  remove("smsort_1");
  remove("smsort_2");
}
```

Зауважимо, що для виконання зовнішнього сортування методом простого злиття в оперативній пам'яті потрібно розташувати всього лише дві змінні - для розміщення чергових елементів (записів) у допоміжні файли. Час роботи алгоритму пропорційний O (*log n*), стільки разів будуть прочитані і стільки ж разів записані файли.

Сортування природним злиттям

У разі простого злиття часткова впорядкованість даних, які сортуються, не дає жодної переваги. Це пояснюється тим, що на кожному проході зливаються серії фіксованої довжини. При природному злитті довжина серій не обмежується, а визначається кількістю елементів в уже впорядкованих підпослідовностях, що виділяються на кожному проході.

Сортування, за якого завжди зливаються дві найдовші з можливих послідовностей, є *природним злиттям*. У даному сортуванні об'єднуються серії максимальної довжини (рис. 9.2).

Вихідний файл ***f:*** **2 3 17 7 8 9 1 4 6 9 2 3 1 18**

	Розподіл	***Злиття***
1 прохід	*f1*: 2 3 17 ` 1 4 6 9 ` 1 18 *f2*: 7 8 9 ` 2 3	*f*: 2 3 7 8 9 17 1 2 3 4 6 9 1 18
2 прохід	*f1*: 2 3 7 8 9 17 ` 1 18 *f2*: 1 2 3 4 6 9 `	*f*: 1 2 2 3 3 4 6 7 8 9 9 17 1 18
3 прохід	*f1*: 1 2 2 3 3 4 6 7 8 9 9 17 *f2*: 1 18	*f*: 1 1 2 2 3 3 4 6 7 8 9 9 17 18

Рисунок 9.2 − Демонстрація сортування двошляховим двофазним природним злиттям

Алгоритм сортування природним злиттям

Крок 1. Вихідний файл *f* розбивається на два допоміжних файли *f*1 і *f*2. Розподіл відбувається наступним чином: за чергою зчитуються записи a_i вихідної послідовності (невпорядкованої) таким чином, що якщо значення ключів сусідніх записів задовольняють умові $f(a_i) <= f(a_i + 1)$, то вони записуються у перший допоміжний файл *f*1. Як тільки зустрічаються $f(a_i) > f(a_i + 1)$, то записи $a_i + 1$ копіюються у другий допоміжний файл *f*2. Процедура повторюється до тих пір, поки всі записи вихідної послідовності не будуть розподілені по файлах.

Крок 2. Допоміжні файли *f*1 і *f*2 зливаються в файл *f*, при цьому серії утворюють впорядковані послідовності.

Крок 3. Отриманий файл *f* знову обробляється, як зазначено в кроках 1 і 2.

Крок 4. Повторюючи кроки, зливаємо впорядковані серії до тих пір, поки не буде впорядкований цілком весь файл.

Символ "` "позначає ознаку кінця серії.

Ознаками кінця сортування природним злиттям є наступні умови:

− кількість серій дорівнює 1 (визначається на фазі злиття).

− при однофазному сортуванні другий за рахунком допоміжний файл після розподілу серій залишився порожнім.

Природне злиття, у якого після фази розподілу кількість серій у допоміжних файлах відрізняється один від одного не більше ніж на одиницю, називається *збалансованим злиттям*, в іншому випадку − *незбалансоване злиття*.

//Опис функцвї сортування природним злиттям

```
void Natural_Merging_Sort (char *name){
  int s1, s2, a1, a2, mark;
  FILE *f, *f1, *f2;
```

```
s1 = s2 = 1;
while ( s1 > 0 && s2 > 0 ){
  mark = 1;
  s1 = 0;
  s2 = 0;
  f = fopen(name,"r");
  f1 = fopen("nmsort_1","w");
  f2 = fopen("nmsort_2","w");
  fscanf(f,"%d",&a1);
  if ( !feof(f) ) {
    fprintf(f1,"%d ",a1);
  }
  if ( !feof(f) ) fscanf(f,"%d",&a2);
  while ( !feof(f) ){
    if ( a2 < a1 ) {
      switch (mark) {
        case 1:{fprintf(f1,"' "); mark = 2; s1++; break;}
        case 2:{fprintf(f2,"' "); mark = 1; s2++; break;}
      }
    }
    if ( mark == 1 ) { fprintf(f1,"%d ",a2); s1++; }
    else { fprintf(f2,"%d ",a2); s2++;}
    a1 = a2;
    fscanf(f,"%d",&a2);
  }
  if ( s2 > 0 && mark == 2 ) { fprintf(f2,"'");}
  if ( s1 > 0 && mark == 1 ) { fprintf(f1,"'");}
  fclose(f2);    fclose(f1);    fclose(f);

  cout << endl;
  Print_File(name);
  Print_File("nmsort_1");
  Print_File("nmsort_2");
  cout << endl;

  f = fopen(name,"w");
  f1 = fopen("nmsort_1","r");
  f2 = fopen("nmsort_2","r");
  if ( !feof(f1) ) fscanf(f1,"%d",&a1);
  if ( !feof(f2) ) fscanf(f2,"%d",&a2);
  bool file1, file2;
  while ( !feof(f1) && !feof(f2) ){
    file1 = file2 = false;
    while ( !file1 && !file2 ) {
      if ( a1 <= a2 ) {
        fprintf(f,"%d ",a1);
        file1 = End_Range(f1);
        fscanf(f1,"%d",&a1);
      }
      else {
        fprintf(f,"%d ",a2);
        file2 = End_Range(f2);
        fscanf(f2,"%d",&a2);
```

```
        }
      }
      while ( !file1 ) {
        fprintf(f,"%d ",a1);
        file1 = End_Range(f1);
        fscanf(f1,"%d",&a1);
      }
      while ( !file2 ) {
        fprintf(f,"%d ",a2);
        file2 = End_Range(f2);
        fscanf(f2,"%d",&a2);
      }    }
    file1 = file2 = false;
    while ( !file1 && !feof(f1) ) {
      fprintf(f,"%d ",a1);
      file1 = End_Range(f1);
      fscanf(f1,"%d",&a1);
    }
    while ( !file2 && !feof(f2) ) {
      fprintf(f,"%d ",a2);
      file2 = End_Range(f2);
      fscanf(f2,"%d",&a2);    }
    fclose(f2);
    fclose(f1);
    fclose(f);
  }
  remove("nmsort_1");
  remove("nmsort_2");
}
//визначення кінця блока
bool End_Range (FILE * f){
  int tmp;
  tmp = fgetc(f);
  tmp = fgetc(f);
  if (tmp != '\'') fseek(f,-2,1);
  else fseek(f,1,1);
  return tmp == '\'' ? true : false;
}
```

Контрольні питання

1. Чим обумовлено використання алгоритмів зовнішнього сортування?
2. Як витрачається ОЗУ при використанні різних алгоритмів зовнішнього сортування?
3. Яким злиттям, простим або природним, ефективніше об'єднувати два впорядкованих за загальним ключем файли? Відповідь обґрунтуйте.
4. Які ще чинники, крім числа фаз і шляхів, слід враховувати при аналізі ефективності алгоритмів зовнішнього сортування?
5. Як визначити, яким алгоритмом зовнішнього сортування віддати перевагу при розв’язанні задачі?

Лекція 10
АЛГОРИТМИ НА ГРАФАХ

Алгоритми обходу графа

Граф − це сукупність двох кінцевих множин: множини точок і множини ліній, які попарно з'єднують деякі з цих точок. Множина точок називається *вершинами (вузлами) графа*. Множина ліній, що з'єднують вершини графа, називається *ребрами (дугами) графа*.

Орієнтований граф (орграф) − граф, у якого всі ребра орієнтовані, тобто ребрам якого присвоєно напрямок.

Неорієнтований граф (неорграф) − граф, у якого всі ребра неорієнтовані, тобто ребрам якого не задано напрямок.

Змішаний граф − граф, що містить як орієнтовані, так і неорієнтовані ребра.

Петлею називається ребро, що з'єднує вершину саму з собою. Дві вершини називаються *суміжними*, якщо існує ребро, яке їх з'єднує. Ребра, що з'єднують одну і ту ж пару вершин, називаються *кратними*.

Простий граф − це граф, в якому немає ні петель, ні кратних ребер.

Мультиграф − це граф, у якого будь-які дві вершини з'єднані більш ніж одним ребром.

Маршрутом у графі називається кінцева послідовність суміжних вершин і ребер, що з'єднують ці вершини.

Маршрут називається *відкритим*, якщо його початкова та кінцева вершини різні, в іншому випадку він називається *замкнутим*.

Маршрут називається *ланцюгом*, якщо всі його ребра різні. Відкритий ланцюг називається *шляхом*, якщо всі його вершини різні.

Замкнутий ланцюг називається *циклом*, якщо різні всі його вершини, за винятком кінцевих.

Граф називається *зв'язаним*, якщо для будь-якої пари вершин існує шлях, який їх з'єднує.

Вага вершини − число (дійсне, ціле або раціональне), поставлене у відповідність даній вершині (інтерпретується як вартість, пропускна здатність і т. ін.). *Вага (довжина) ребра* − число або кілька чисел, які інтерпретуються по відношенню до ребра як довжина, пропускна здатність і т. ін.

Зважений граф − граф, кожному ребру якого поставлено у відповідність деяке значення (вага ребра).

Вибір структури даних для зберігання графа в пам'яті комп'ютера має принципове значення при розробці ефективних алгоритмів. Розглянемо кілька способів подання графа.

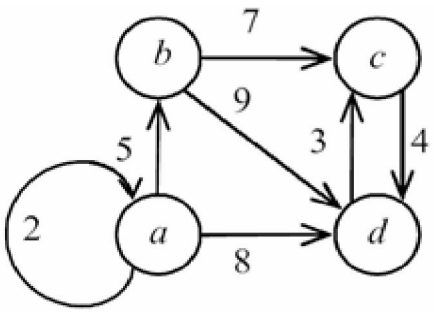


Рисунок 10.1 − Граф

Нехай заданий граф (наприклад, рис. 10.1), у якого кількість вершин дорівнює n, а кількість ребер − m. Кожне ребро і кожна вершина мають вагу − ціле позитивне число. Якщо граф не є позначеним, то вважається, що вага дорівнює одиниці.

1. *Список ребер* − це множина, утворена парами суміжних вершин (рис. 10.2). Для його зберігання зазвичай використовують одновимірний масив розміром m, що містить список пар вершин, суміжних з одним ребром графа. Список ребер більш зручний для реалізації різних алгоритмів на графах порівнянно з іншими способами.

a	*a*	*a*	*b*	*b*	*c*	*d*
a	*b*	*d*	*c*	*d*	*d*	*c*
2	5	8	7	9	4	3

Рисунок 10.2 − Список ребер графа

2. *Матриця суміжності* − це двовимірний масив розмірності $n \times n$, значення елементів якого характеризуються суміжністю вершин графа (рис. 10.3). При цьому значенню елемента матриці присвоюється кількість ребер, які з'єднують відповідні вершини. Даний спосіб дієвий, коли треба перевіряти суміжність або знаходити вагу ребра за двома заданими вершинами.

	a	*b*	*c*	*d*
a	2	5	0	8
b	0	0	7	9
c	0	0	0	4
d	0	0	3	0

Рисунок 10.3 − Матриця суміжності графа

	a	*b*	*c*	*d*
(*a*, *a*)	2	0	0	0
(*a*, *b*)	0	5	0	0
(*a*, *d*)	0	0	0	8
(*b*, *c*)	0	0	7	0
(*b*, *d*)	0	0	0	9
(*c*, *d*)	0	0	0	4
(*d*, *c*)	0	0	3	0

Рисунок 10.4 − Матриця інцидентності графа

3. *Матриця інцидентності* − це двовимірний масив розмірності n x m, в якому вказуються зв'язки між інцидентними елементами графа (ребро і вершина). Стовпці матриці відповідають ребрам, рядки − вершинам (рис. 10.4). Ненульові значення в матриці вказують на зв'язок між вершиною і ребром. Даний спосіб є найбільш ємним для зберігання, але полегшує знаходження циклів у графі.

Існує багато алгоритмів на графах, в основі яких лежить систематичний перебір вершин графа, такий що кожна вершина перейлядається (відвідується) в точності один раз. Тому важливим завданням є визначення хороших методів пошуку у графі.

Під *обходом графа (пошуку на графі)* розуміється процес систематичного перегляду всіх ребер або вершин графа з метою відшукання ребер або вершин, які відповідають деякій умові.

При вирішенні багатьох завдань, що використовують графи, необхідні ефективні методи регулярного обходу вершин і ребер графів. До стандартних і найбільш поширених методів відносяться:

- *пошук в глибину* (Depth First Search, DFS);
- *пошук в ширину* (Breadth First Search, BFS).

Ці методи найчастіше розглядаються на орієнтованих графах, але вони можуть бути застосовані і для неорієнтованих, ребра яких вважаються двонаправленими. Алгоритми обходу в глибину і в ширину лежать в основі вирішення різних завдань обробки графів, наприклад, побудови остовного лісу, перевірки зв'язності, ациклічності, обчислення відстаней між вершинами й інших.

Пошук в глибину

При пошуку в глибину переглядаэться перша вершина, потім необхідно йти вздовж ребер графа, до попадання в глухий кут. Вершина графа є *тупиком*, якщо всі суміжні з нею вершини вже переглянуті. Після потрапляння в глухий кут потрібно повертатися назад уздовж пройденого шляху, поки не буде виявлена вершина, у якій є ще не переглянута вершина, а потім необхідно рухатися в цьому новому напрямку. Процес є завершеним при поверненні в початкову вершину, причому всі суміжні з нею вершини вже повинні бути переглянуті.

Таким чином, основна ідея пошуку в глибину − коли можливі шляхи за ребрами, що виходять із вершин, розгалужуються, потрібно спочатку повністю дослідити одну гілку і тільки потім переходити до інших гілок (якщо вони залишаться нерозглянутими).

Алгоритм пошуку в глибину

Крок 1. Всім вершинам графа присвоюється значення не переглянуті. Вибирається перша вершина, і позначається як переглянута.

Крок 2. Для останньої поміченої як переглянута вершина вибирається суміжна вершина, яка є першою поміченою, що не переглянута, і їй присвоюється значення переглянута. Якщо таких вершин немає, то береться попередня позначена вершина.

Крок 3. Повторювати крок 2 до тих пір, поки всі вершини не будуть позначені як переглянуті (рис. 10.5).

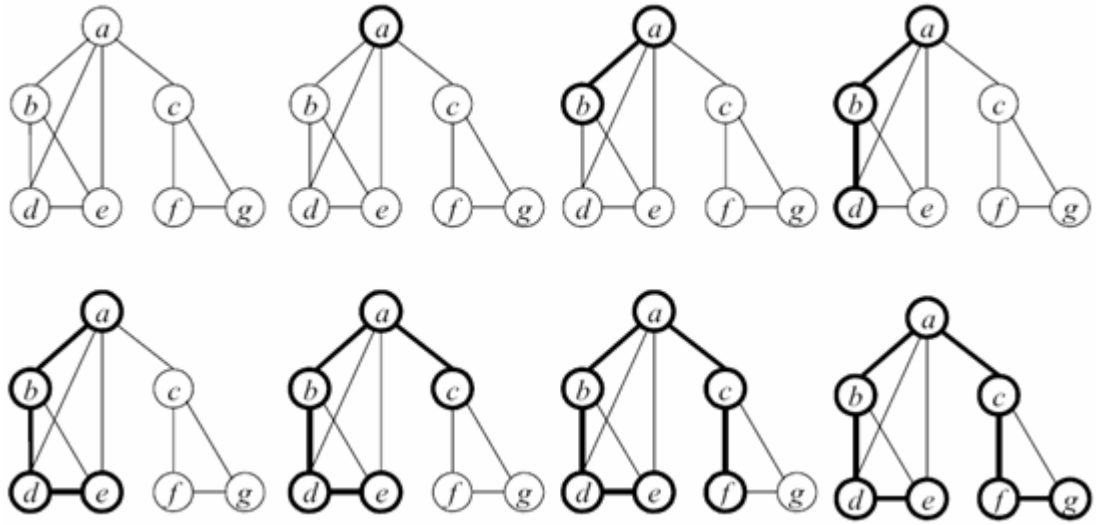


Рисунок 10.5 − Демонстрація алгоритму пошуку в глибину

```
//Опис функції алгоритму пошуку в глибину
void Depth_First_Search(int n, int **Graph, bool *Visited,
                        int Node){
  Visited[Node] = true;
  cout << Node + 1 << endl;
  for (int i = 0 ; i < n ; i++)
    if (Graph[Node][i] && !Visited[i])
      Depth_First_Search(n,Graph,Visited,i);
}
```

Час роботи алгоритму залежить від подання графа. Якщо застосована матриця суміжності, то час роботи алгоритму буде дорівнювати $O(n^2)$, а якщо нематричне подання – $O(n + m)$: розглядаються всі вершини і всі ребра.

Пошук в ширину

При пошуку в ширину, після перегляду першої вершини, переглядаються всі сусідні з нею вершини. Потім відвідуються всі вершини, що знаходяться на відстані двох ребер від початкової. При кожному новому кроці переглядаються вершини, відстань від яких до початкової на одиницю більше попередньої. Щоб запобігти повторному перегляду вершин, необхідно вести список переглянутих вершин. Для зберігання тимчасових даних, необхідних для роботи алгоритму, використовується черга − упорядкована послідовність елементів, в якій нові елементи додаються в кінець, а старі видаляються з початку.

Алгоритм пошуку в ширину

Крок 1. Всім вершинам графа присвоюється значення не переглянуті. Вибирається перша вершина, і позначається як переглянута (і заноситься у чергу).

Крок 2. Відвідується перша вершина з черги (якщо вона не позначена як переглянута). Всі її сусідні вершини заносяться в чергу. Після цього вона видаляється з черги.

Крок 3. Повторюється крок 2 до тих пір, поки черга не стає порожнею (рис. 10.6).

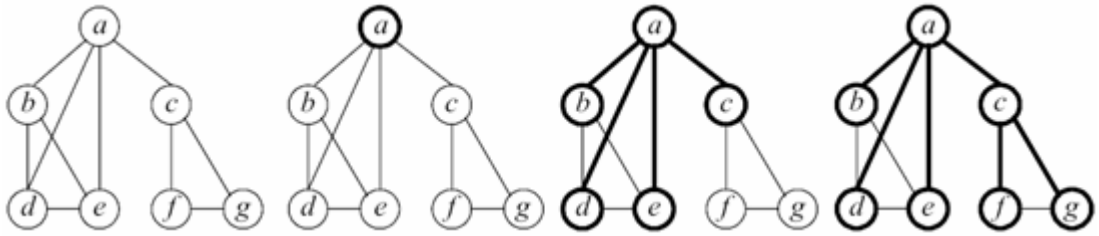


Рисунок 10.6 − Демонстрація алгоритму пошуку в ширину

//Опис функції алгоритму пошуку в ширину

```
void Breadth_First_Search(int n, int **Graph,
                          bool *Visited, int Node){
  int *List = new int[n]; //черга
  int Count, Head;        // покажчик черги
  int i;
  // початкова ініціалізація
  for (i = 0; i < n ; i++)
    List[i] = 0;
  Count = Head = 0;
  // переміщення в чергу вершини Node
  List[Count++] = Node;
  Visited[Node] = true;
  while ( Head < Count ) {
    // взяття вершини з черги
    Node = List[Head++];
    cout << Node + 1 << endl;
    // перегляд усіх вершин, які зв'язані з вершиною Node
```

```
    for (i = 0 ; i < n ; i++)
      // якщо вершина раніше не проглянута
      if (Graph[Node][i] && !Visited[i]){
        // заносимо її в чергу
        List[Count++] = i;
        Visited[i] = true;
      }
   }
}
```

Час роботи алгориту пошуку в ширину при нематричному поданні графа дорівнює O(n + m), бо розглядаються всі n вершин і m ребер. Використання матриці суміжності призводить до оцінки O (n^2).

Контрольні питання

1. Як пов'язані між собою різні способи подання графів?
2. Як від виду або подання графа залежить тимчасова складність алгоритмів пошуку в глибину і в ширину?
3. Як при реалізації в коді виконується повернення з тупикових вершин при обході графа?
4. Як виконується обхід у незв'язаному графі?
5. Чи поширюються поняття "пошук в глибину" і "пошук в ширину" на незв'язаних графах? Відповідь обґрунтуйте.

Лекція 11
АЛГОРИТМИ ЗНАХОДЖЕННЯ НАЙКОРОТШОГО ШЛЯХУ

Найкоротший шлях розглядається за допомогою математичного об'єкта, названого графом. Пошук найкоротшого шляху відбувається між двома заданими вершинами у графі. Результатом є шлях, тобто послідовність вершин і ребер, інцидентних двом сусіднім вершинам, і його довжина.

Розглянемо три найбільш ефективних алгоритмів знаходження найкоротшого шляху:

- *алгоритм Дейкстри;*
- *алгоритм Флойда;*
- *переборні алгоритми.*

Зазначені алгоритми легко виконуються за малої кількості вершин у графі. При збільшенні їх кількості завдання пошуку найкоротшого шляху ускладнюється.

Алгоритм Дейкстри

Даний алгоритм є алгоритмом на графах, який винайдений нідерландським вченим Е. Дейкстрєм у 1959 році. Алгоритм знаходить найкоротшу відстань від однієї з вершин графа до всіх інших і працює тільки для графів без ребер негативної ваги.

Кожній вершині приписується вага − це вага шляху від початкової вершини до даної. Також кожна вершина може бути виділена. Якщо вершина виділена, то шлях від неї до початкової вершини найкоротший, якщо ні − то тимчасовий. Обходячи граф, алгоритм визначає для кожної вершини маршрут, і, якщо він виявляється найкоротшим, виділяє вершину. Вагою даної вершини стає вага шляху. Для всіх сусідів даної вершини алгоритм також розраховує вагу, при цьому ні за яких умов не виділяючи їх. Алгоритм закінчує свою роботу, дійшовши до кінцевої вершини, і вагою найкоротшого шляху стає вага кінцевої вершини.

Алгоритм Дейкстри

Крок 1. Всім вершинам, за винятком першої, присвоюється вага, яка дорівнює нескінченності, а першій вершині − 0.

Крок 2. Всі вершини не виділено.

Крок 3. Перша вершина оголошується поточною.

Крок 4. Вага всіх невиділених вершин перераховується за формулою: вага невиділеної вершини є мінімальне число зі старої ваги даної вершини та суми ваги поточної вершини і ваги ребра, що з'єднує поточну вершину з невиділеною.

Крок 5. Серед невиділених вершин шукається вершина з мінімальною вагою. Якщо така не знайдена, тобто вага всіх вершин дорівнює нескінченності, то маршрут не існує. Отже, вихід. Інакше, поточною стає знайдена вершина. Вона ж виділяється.

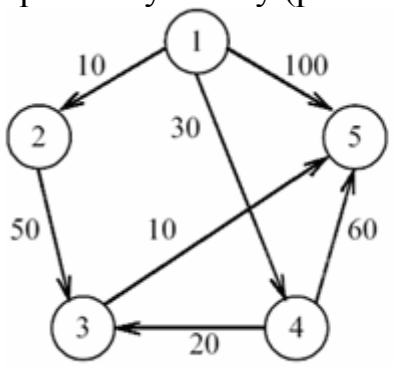
Крок 6. Якщо поточною вершиною виявляється кінцева, то шлях знайдений, і його вага є вага кінцевої вершини.

Крок 7. Перехід до кроку 4.

У програмній реалізації алгоритму Дейкстри побудуємо множину *S* вершин, для яких найкоротші шляхи від початкової вершини вже відомі. На кожному кроці до множини *S* додається та з решти вершин, відстань до якої від початкової вершини менша, ніж для інших вершин. При цьому будемо використовувати масив *D*, в який записуються довжини найкоротших шляхів для кожної вершини. Коли множина *S* буде містити всі вершини графа, тоді масив *D* міститиме довжини найкоротших шляхів від початкової вершини до кожної вершини.

Крім зазначених масивів будемо використовувати матрицю довжин *C*, де елемент *C* [*i*, *j*] − довжина ребра (*i, j*), якщо ребра немає, то її довжина покладається рівною нескінченності, тобто більше будь-якої фактичної довжини ребер. Фактично матриця *C* являє собою матрицю суміжності, в якій всі нульові елементи замінені на нескінченність.

Для визначення самого найкоротшого шляху введемо масив *P* вершин, де *P*[*v*] буде містити вершину, що безпосередньо передує вершині *v* у найкоротшому шляху (рис. 11.1).



Ітерація	*S*	*w*	D[2]	D[3]	D[4]	D[5]
початок	{1}	–	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3,5}	5	10	50	30	60

Масив **P**: | ⊠ | 1 | 4 | 1 | 3 |

Найкоротший шлях з 1 до 5 : {1, 4, 3, 5}

Рисунок 11.1 − Демонстрація алгоритму Дейкстри

//опис функції алгоритму Дейкстри

```
void Dijkstra(int n, int **Graph, int Node){
  bool *S = new bool[n];
  int *D = new int[n];
  int *P = new int[n];
  int i, j;
  int Max_Sum = 0;
  for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
      Max_Sum += Graph[i][j];
  for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
      if (Graph[i][j] == 0)
        Graph[i][j] = Max_Sum;
  for (i = 0 ; i < n ; i++){
    S[i] = false;
    P[i] = Node;
```

```
    D[i] = Graph[Node][i];
  }
  S[Node] = true;
  P[Node] = -1;
  for ( i = 0 ; i < n - 1 ; i++ ){
    int w = 0;
    for ( j = 1 ; j < n ; j++ ){
      if (!S[w]){
        if (!S[j] && D[j] <= D[w])
          w = j;
      }
      else w++;
    }
    S[w] = true;
    for ( j = 1 ; j < n ; j++ )
      if (!S[j])
        if (D[w] + Graph[w][j] < D[j]){
          D[j] = D[w] + Graph[w][j];
          P[j] = w;
        }
  }
  for ( i = 0 ; i < n ; i++ )
    printf("%5d",D[i]);
  cout << endl;
  for ( i = 0 ; i < n ; i++ )
    printf("%5d",P[i]+1);
  cout << endl;
  delete [] P;
  delete [] D;
  delete [] S;
}
```

Якщо для подання графа використовувати матрицю суміжності, то час роботи цього алгоритму має порядок O (n^2), де n − кількість вершин графа.

Алгоритм Флойда

Метод Флойда безпосередньо грунтується на тому факті, що у графа з позитивною вагою ребер всякий складний (що містить більше 1 ребра) найкоротший шлях складається з інших найкоротших шляхів.

Цей алгоритм більш загальний порівняно з алгоритмом Дейкстри, тому що він знаходить найкоротші шляхи між будь-якими двома вершинами графа.

У алгоритмі Флойда використовується матриця A розміром n x n, в якій обчислюються довжини найкоротших шляхів. Елемент A [i, j] дорівнює відстані від вершини i до вершини j, яка має кінцеве значення, якщо існує ребро (i, j), і дорівнює нескінченності в іншому випадку.

Алгоритм Флойда

Основна ідея алгоритму. Нехай є три вершини i, j, k і задані відстані між ними. Якщо виконується нерівність $A[i, k] + A[k, j] < A[i, j]$, то доцільно замінити шлях $i \rightarrow j$ шляхом $i \rightarrow k \rightarrow j$. Така заміна виконується систематично в процесі виконання даного алгоритму.

Крок 0. Визначаємо початкову матрицю відстані $A0$ і матрицю послідовності вершин $S0$. Кожен діагональний елемент обох матриць дорівнює 0, таким чином, показуємо, що ці елементи в обчисленнях не беруть участі. Вважаємо $k = 1$.

Основний крок k. Задаємо рядок k і стовпець k як провідний рядок і ведучий стовпець. Розглядаємо можливість застосування заміни описаної вище, до всіх елементів $A\ [i, j]$ матриці A_{k-1}. Якщо виконується нерівність, тоді виконуємо наступні дії:

1) створюємо матрицю A_k шляхом заміни в матриці A_{k-1} елемента $A\ [i, j]$ на суму $A\ [i, k] + A\ [k, j]$;

2) створюємо матрицю S_k шляхом заміни в матриці S_{k-1} елемента $S\ [i, j]$ на k. Вважаємо $k = k + 1$ і повторюємо крок k.

Таким чином, алгоритм Флойда робить n ітерацій, після i-ї ітерації матриця A буде містити довжини найкоротших шляхів між будь-якими двома парами вершин за умови, що ці шляхи проходять через вершини від першої до i-ї. На кожній ітерації перебираються всі пари вершин і шлях між ними скорочується за допомогою i-ї вершини (рис. 11.2).

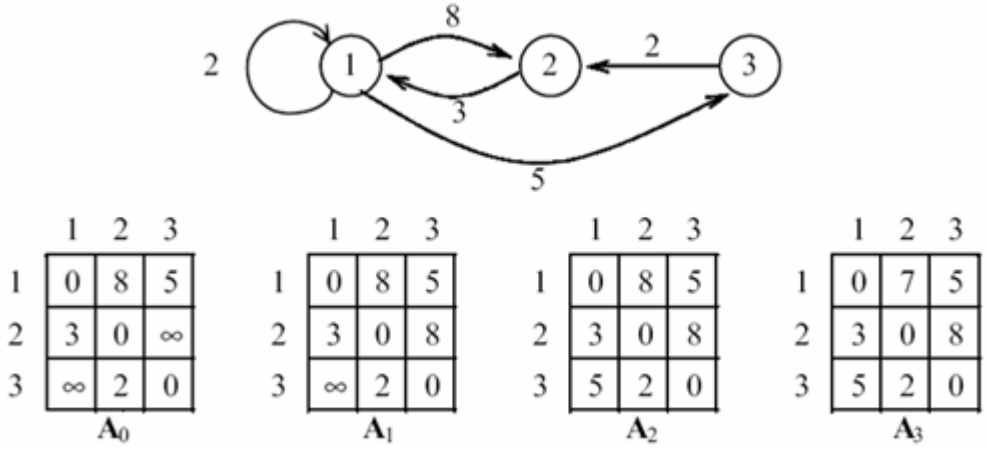


Рисунок 11.2 − Демонстрація алгоритму Флойда

```
//опис функції алгоритму Флойда
void Floyd(int n, int **Graph, int **ShortestPath){
  int i, j, k;
  int Max_Sum = 0;
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      Max_Sum += ShortestPath[i][j];
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      if ( ShortestPath[i][j] == 0 && i != j )
        ShortestPath[i][j] = Max_Sum;
  for ( k = 0 ; k < n; k++ )
    for ( i = 0 ; i < n; i++ )
      for ( j = 0 ; j < n ; j++ )
        if ((ShortestPath[i][k] + ShortestPath[k][j]) <
             ShortestPath[i][j])
          ShortestPath[i][j] = ShortestPath[i][k] +
            ShortestPath[k][j];
}
```

Якщо граф поданий матрицею суміжності, то час роботи цього алгоритму має порядок $O(n^3)$, оскільки в ньому присутні вкладені три цикли.

Контрольні питання

1. З якими видами графів працюють алгоритми Дейкстри і Флойда?
2. Як від подання графа залежить ефективність алгоритму його обходу?
3. За рахунок чого пошук у ширину є досить ресурсомістким алгоритмом?
4. У чому переваги алгоритмів обходу графа в ширину?

ЛІТЕРАТУРА

1. Альфред Ахо. Структуры данных и алгоритмы/ [Альфред Ахо, Джон Э. Хопкрофт, Джеффри Д.Ульман]. − М.: Вильямс, 2007. − 400 с.
2. Вирт Н. Алгоритмы и структуры данных/ Вирт Н.− М., 2012. − 272 с.
3. Седжвик Н. Фундаментальные алгоритмы на С++. Части 1-4/ Седжвик Н. − Диасофт, 2001. − 688 с.
4. Кнут Д. Искусство программирования. Т. 1-4/ Кнут Д.− М.: Вильямс, 2006. − 682 с.
5. Леонов Ю.Г. Программирование на алгоритмическом языке С++: учебное пособие по лабораторному практикуму/ Леонов Ю.Г., Силкина Н.В., Шпинова Е.Д. – Одесса: ОНАС, 2002. − 68 с.
6. Березин Б.Н. Учебный курс С и С++/ Березин Б.Н., Березин С.Б. – М.: Диалог-МИФИ, 2000. – 288 с.

ЗМІСТ