МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ПОЛІТЕХНІЧНИЙ КОЛЕДЖ
ОДЕСЬКОГО НАЦІОНАЛЬНОГО ПОЛІТЕХНІЧНОГО УНІВЕРСИТЕТУ
ВІДДІЛЕННЯ КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курс лекції

з дисципліни

«Алгоритми та структури даних»

Херсон — 2015

Зміст

Перелік ілюстрацій

Перелік таблиць

Listings

Лекція № 1

Систення інформація за допомогою бінарних дерев

1.1 Вступ

Дерево — це структура даних, що представляє собою сукупність елементів і відносин, що утворюють ієрархічну структуру цих елементів (рис. 1.1). Кожен елемент дерева називається вершиною (вузлом) дерева. Вершини дерева з'єднані спрямованими дугами, які називають гілками дерева. Початковий вузол дерева називають коренем дерева, йому відповідає нульовий рівень. Листям дерева називають вершини, в які входить одна гілка і не виходить жодної гілки.

Кожне дерево має такі властивості:

- існує вузол, в який не входить ні однієї дуги (корінь);
- в кожну вершину, крім кореня, входить одна дуга.

Дерева особливо часто використовують на практиці при зображенні різних ієрархій.

Алгоритм Хаффмана - адаптивний жадний алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю. Був розроблений в 1952 році аспірантом Массачусетського технологічного інституту Девідом Хаффманом при написанні їм курсової роботи. В даний час використовується в багатьох програмах стиснення даних.

1.2 Ключові терміни

Стиснення даних - це процес, що забезпечує зменшення обсягу даних шляхом скорочення їх надмірності.

Стиснення без втрат (повністю оборотне) - це метод стиснення даних, при якому раніше закодована порція даних відновлюється після їх розпакування повністю без внесення змін.

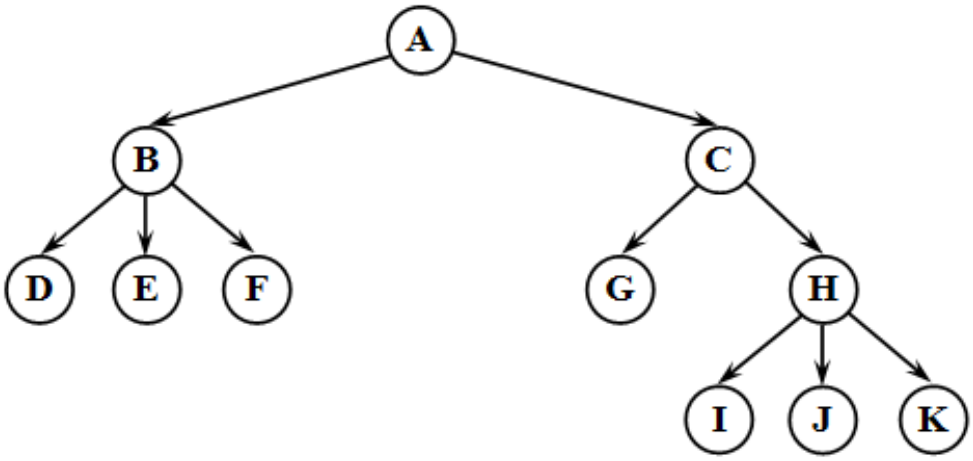


Рис. 1.1 – Дерево

Стиснення з втратами - це метод стиснення даних, при якому для забезпечення максимального ступеня стиску вихідного масиву даних частину містяться в ньому даних відкидається.

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

Алфавіт коду - це множина всіх символів вхідного потоку.

Кодовий символ - це найменша одиниця даних, що підлягає стисненню.

Кодове слово - це послідовність кодових символів з алфавіту коду.

Токен - це одиниця даних, записувана в стислий потік деяким алгоритмом стиснення.

Фраза - це фрагмент даних, що поміщається в словник для подальшого використання в стисненні.

Кодування - це процес стиснення даних.

Декодування - це зворотний кодуванню процес, при якому здійснюється відновлення даних.

Ставлення стиснення - це величина для позначення ефективності методу стиснення, що дорівнює відношенню розміру вихідного потоку до розміру вхідного потоку.

Коефіцієнт стиснення - це величина, зворотна відношенню стиснення.

Середня довжина кодового слова - це величина, яка обчислюється як зважена ймовірностями сума довжин всіх кодових слів.

Статистичні методи - це методи стиснення, що привласнюють коди змінної довжини символам вхідного потоку, причому більш короткі коди присвоюються символам або групам символам, що має більшу ймовірність появи у вхідному потоці.

Словникове стиснення - це методи стиснення, що зберігають фрагменти даних в деякій структурі даних, званої словником.

Хаффманове кодування (стиснення) - це метод стиснення, привласнює символам алфавіту коди змінної довжини грунтуючись на ймовірність появи цих символів.

Префіксний код - це код, в якому ніяке кодове слово не є префіксом будь-якого іншого кодового слова.

Оптимальний префіксний код - це префіксний код, що має мінімальну середню довжину.

Кодова дерево (дерево кодування Хаффмана, Н-дерево) - це бінарне дерево, у якого: листя позначені символами, для яких розробляється кодування; вузли (у тому числі корінь) позначені сумою ймовірностей появи всіх символів, відповідних листю поддерева, коренем якого є відповідний вузол.

1.3 Розширені теоретичні відомості

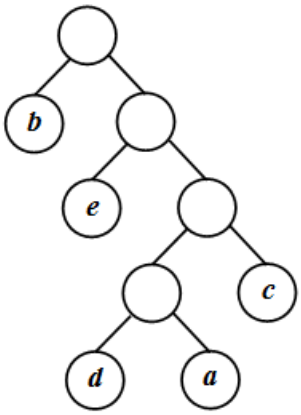
Кодова дерево (дерево кодування Хаффмана, Н-дерево) - це бінарне дерево, у якого:

- ◇ листя позначені символами, для яких розробляється кодування;

Вероятности появления символов

Символ	Вероятность
a	0,1
b	0,4
c	0,2
d	0,05
e	0,25

Кодовое дерево



Оптимальные префиксные коды

Символ	Код
a	1101
b	0
c	111
d	1100
e	10

Рис. 1.2 – Дерево

◇ вузли (у тому числі корінь) позначені сумою ймовірностей появи всіх символів, відповідних листю поддерева, коренем якого є відповідний вузол.,

Метод Хаффмана на вході отримує таблицю частот зустрічальності символів у вихідному тексті. Далі на підставі цієї таблиці будується дерево кодування Хаффмана.

Алгоритм побудови дерева Хаффмана.

1. Символи вхідного алфавіту утворюють список вільних вузлів. Кожен лист має вагу, що може бути дорівнює або ймовірності, або кількістю входжень символу в стискається текст.

2. Вибираються два вільні вузла дерева з найменшими вагами.

3. Створюється їх батько з вагою, рівним їх сумарному вазі.

4. Батько додається в список вільних вузлів, а двоє його дітей видаляються з цього списку.

5. Однією дузі, що виходить з батька, ставиться у відповідність біт 1, інший - біт 0.

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

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

Приклад побудови кодового дерева. Нехай задана вихідна послідовність символів: `aabbbbbbbbccccdeeeee`.

Її вихідний обсяг дорівнює 20 байт (160 біт). Згідно з наведеними на рис. 1.2 даними (таблиця ймовірності появи символів, кодове дерево і таблиця оптимальних префіксних кодів) закодована вихідна послідовність символів буде виглядати наступним чином:

`11011101000000001111111111111001010101010`.

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

Класичний алгоритм Хаффмана має один істотний недолік. Для відновлення вмісту стислого тексту при декодуванні необхідно знати таблицю частот яку використовували при кодуванні. Отже, довжина стислого тексту збільшується на довжину таблиці частот, яка повинна

надсилатися попереду даних, що може звести нанівець всі зусилля зі стиснення даних. Крім того, необхідність наявності повної частотної статистики перед початком власне кодування вимагає двох проходів по тексту: одного для побудови моделі тексту (Таблиці частот і дерева Хаффмана), іншого для власне кодування.

У процесі роботи алгоритму стиснення вага вузлів в дереві кодування Хаффмана неухильно зростає. Перша проблема виникає тоді, коли вага кореня дерева починає перевершувати місткість комірки, в якій він зберігається. Як правило, це 16-бітове значення і, отже, не може бути більше, ніж 65535. Друга проблема, яка заслуговує ще більшої уваги, може виникнути значно раніше, коли розмір найдовшого коду Хаффмана перевершує місткість осередку, який використовується для того, щоб передати його у вихідний потік. Декодеру все одно, якої довжини код він декодує, оскільки він рухається зверху вниз по дереву кодування, вибираючи з вхідного потоку по одному біту. Кодер же повинен починати від листа дерева і рухатися вгору до кореня, збираючи біти, які потрібно передати. Зазвичай це відбувається зі змінною типу «ціле», і, коли довжина коду Хаффмана перевершує розмір типу «ціле» в бітах, настає переповнення.

1.4 Приклади обчислень

Програмну реалізацію алгоритму надано у додатку Б.1.

Лекція № 2

Представлення виразів за допомогою дерев

Мета роботи

2.1 Вступ

2.2 Ключові терміни

2.3 Розширені теоретичні відомості

2.4 Приклади обчислень

Лекція № 3

Представлення багаторозгалужених дерев

Мета роботи

3.1 Вступ

3.2 Ключові терміни

3.3 Розширені теоретичні відомості

3.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 4

Представлення графів

Мета роботи

4.1 Вступ

4.2 Ключові терміни

4.3 Розширені теоретичні відомості

4.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 5
Алгоритми на графах

5.1 Вступ

Знаходження найкоротшого шляху на сьогоднішній день є життєво необхідним завданням і використовується практично скрізь, починаючи від знаходження оптимального маршруту між двома об'єктами на місцевості (наприклад, найкоротший шлях від будинку до університету), в системах автопілота, для знаходження оптимального маршруту при перевезеннях, комутації інформаційного пакету в мережах і т.п.

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

Розглянемо три найбільш ефективних алгоритму знаходження найкоротшого шляху:

- ◇ алгоритм Дейкстри;
- ◇ алгоритм Флойда;
- ◇ Переборні алгоритми.

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

5.2 Ключові терміни

Алгоритм Дейкстри - це алгоритм знаходження найкоротшого шляху від однієї з вершин графа до всіх інших, який працює тільки для графів без ребер негативного ваги.

Алгоритм Флойда - це алгоритм пошуку найкоротшого шляху між будь-якими двома вершинами графа.

Хвильовий алгоритм - це переборний алгоритм, який заснований на пошуку в ширину і складається з двох етапів: поширення хвилі і зворотний хід.

Найкоротший шлях - це шлях в графі, тобто послідовність вершин і ребер, інцидентних двом сусіднім вершинам, і його довжина.

Переборний алгоритм - це алгоритм обходу графа, заснований на послідовному переборі можливих шляхів.

5.3 Розширені теоретичні відомості

5.3.1 Алгоритм Дейкстри

Даний алгоритм є алгоритмом на графах, який винайдений нідерландським вченим Е. Дейкстрой в 1959 році. Алгоритм знаходить найкоротший відстань від однієї з вершин графа до всіх інших і працює тільки для графів без ребер негативного 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 вважає 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

1. Всім вершинам, за винятком першої, присвоюється вага рівний нескінченності, а першій вершині - 0.

2. Всі вершини не виділені.

3. Перша вершина оголошується поточної.

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

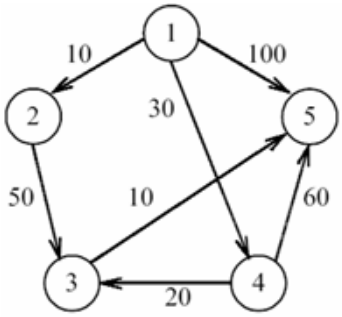
5. Серед невиділених вершин шукається вершина з мінімальною вагою. Якщо така не знайдена, тобто вага всіх вершин дорівнює нескінченності, то маршрут не існує. Отже, вихід. Інакше, поточної стає знайдена вершина. Вона ж виділяється.

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

7. Перехід на крок 4.

У програмній реалізації алгоритму Дейкстри побудуємо безліч 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 менше, ніж для інших, що залишилися вершин. При цьому будемо використовувати масив D, в який записуються довжини найкоротших шляхів для кожної вершини. Коли безліч 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,



Итерация	S	w	**D**[2]	**D**[3]	**D**[4]	**D**[5]
начало	{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

Массив **P**: | ⊠ | 1 | 4 | 1 | 3 |

Кратчайший путь из 1 в 5: {1, 4, 3, 5}

Рис. 5.1 – Демонстрація алгоритму Дейкстри

тобто більше будь 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, безпосередньо попередню вершині v в найкоротшому шляху (рис. 5.1). Демонстрація алгоритмом Дейкстри

Лістинг 5.1 – Опис функції 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;
}
```

Складність алгоритму Дейкстри залежить від способу знаходження вершини, а також способу зберігання безлічі невідвіданих вершин і способи оновлення довжин.

Якщо для представлення графа використовувати матрицю суміжності, то час виконання цього алгоритму має порядок O (n 2), де n - кількість вершин графа.

5.3.2 Алгоритм Флойда

Розглянутий алгоритм іноді називають алгоритмом Флойда -Уоршелла. Алгоритм Флойда -Уоршелла є алгоритмом на графах, який розроблений в 1962 році Робертом Флойдом і Стівеном Уоршеллом. Він служить для знаходження найкоротших шляхів між усіма парами вершин графа.

Метод Флойда безпосередньо грунтується на тому факті, що в графі з позитивними вагами ребер всякий Неелементарні (що містить більше 1 ребра) найкоротший шлях складається з інших найкоротших шляхів.

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

В алгоритмі Флойда використовується матриця A розміром nxn, в якій обчислюються довжини найкоротших шляхів. Елемент A [i, j] дорівнює відстані від вершини i до вершини j, яке 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-> j шляхом i-> k-> j. Така заміна виконується систематично в процесі виконання даного алгоритму.

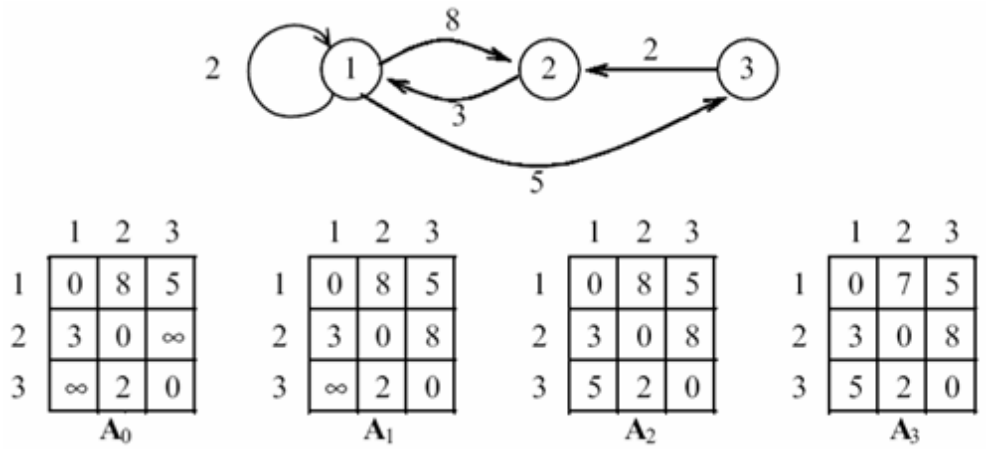


Рис. 5.2 – Демонстрація алгоритму Флойда

Крок 0. Визначаємо початкову матрицю відстані A 0 і матрицю послідовності вершин S 0. Кожен діагональний елемент обох матриць дорівнює 0, таким чином, показуючи, що ці елементи в обчисленнях не беруть участь. Вважаємо k = 1.

Основний крок k. Задаємо рядок k і стовпець k як провідну рядок і провідний стовпець. Розглядаємо можливість застосування заміни описаної вище, до всіх елементів A [i, j] матриці A k-1. Якщо виконується нерівність A [i, k] + A [k, j] <A [i, j], (i ne k, j ne k, i ne j) , Тоді виконуємо наступні дії:

створюємо матрицю A k шляхом заміни в матриці A k-1 елемента A [i, j] на суму A [i, k] + A [k, j]; 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 -й 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 -й. На кожній ітерації перебираються всі пари вершин і шлях між ними скорочується за допомогою i -й вершини (рис. 5.2). Демонстрація алгоритму Флойда

Лістинг 5.2 – Опис функції 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];
}
```

Зауважимо, що якщо граф неорієнтовний, то всі матриці, одержувані в результаті перетворень симетричні і, отже, досить вираховуватимуть тільки елементи, розташовані вище головної діагоналі.

Якщо граф представлений матрицею суміжності, то час виконання цього алгоритму має порядок O (n 3), оскільки в ньому присутні вкладені одна в одного три циклу.

5.3.3 Переборні алгоритми

Переборні алгоритми по суті своїй є алгоритмами пошуку, як правило, пошуку оптимального рішення. При цьому рішення конструюється поступово. У цьому випадку зазвичай говорять про перебір вершин дерева варіантів. Вершинами такого графа будуть проміжні або кінцеві варіанти, а ребра вказуватимуть шляху конструювання варіантів.

Розглянемо Переборні алгоритми, засновані на методах пошуку в графі, на прикладі задачі знаходження найкоротшого шляху в лабіринті.

Постановка завдання.

Лабіринт, що складається з прохідних і непрохідних клітин, заданий матрицею A розміром mxn. Елемент матриці A [i, j] = 0, якщо клітина (i, j) прохідна. В іншому випадку $A_{[i,j]} = \infty$.

Потрібно знайти довжину найкоротшого шляху з клітини (1, 1) в клітину (m, n).

Фактично дана матриця суміжності (тільки в ній нулі замінені нескінченності, а одиниці - нулями). Лабіринт являє собою граф.

Вершинами дерева варіантів в даній задачі є шляхи, що починаються в клітці (1, 1). Ребра - показують хід конструювання цих шляхів і з'єднують два шляхи довжини k і k + 1, де другий шлях виходить з першого додаванням до шляху ще одного ходу.

Перебір з поверненням

Даний метод заснований на методі пошуку в глибину. Перебір з поверненням вважають методом проб і помилок ("спробуємо сходити в цю сторону: не вийде - повернемося і спробуємо в іншу"). Так як перебір варіантів здійснюється методом пошуку в глибину, то доцільно під час роботи алгоритму зберігати поточний шлях в дереві. Цей шлях являє собою стек Way.

Також необхідний масив Dist, розмірність якого відповідає кількості вершин графа, який зберігає для кожної вершини відстань від неї до вихідної вершини.

Нехай поточної є деяка клітина (на початку роботи алгоритму - клітина (1, 1)). Якщо для поточної клітини є клітина-сусід Neighbor, відсутня в Way, в яку на цьому шляху ще не ходили, то додаємо Neighbor в Way і поточної клітці присвоюємо Neighbor, інакше витягти з Way.

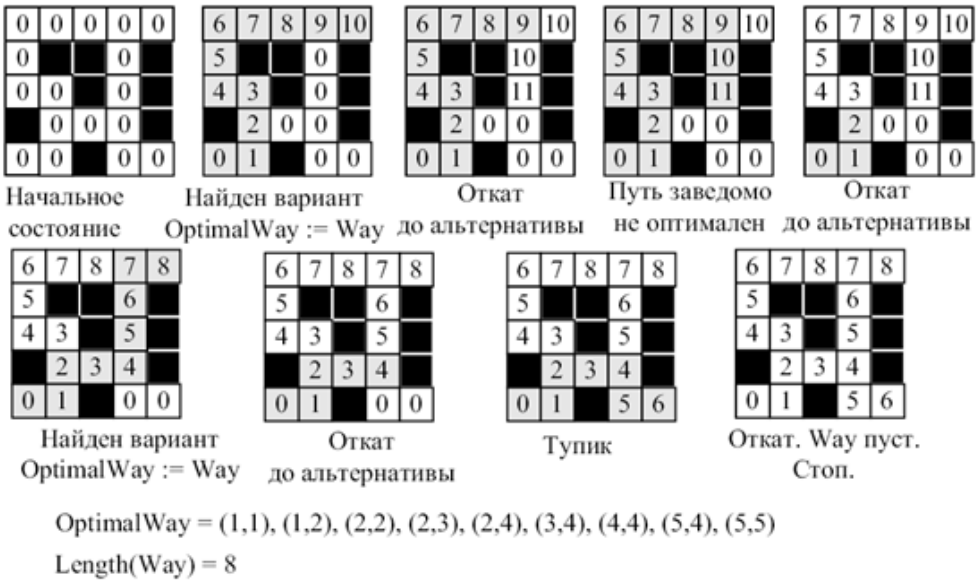


Рис. 5.3 – Демонстрація алгоритму перебору з поверненням

Наведене вище опис дає чітко зрозуміти, чому цей метод називається перебором з поверненням. Поверненню тут відповідає операція "витягти з Way яка зменшує довжину Way на 1.

Перебір закінчується, коли Way порожній і робиться спроба повернення назад. У цій ситуації повертатися вже нікуди (рис. 5.3).

Way є поточним шляхом, але в процесі роботи необхідно зберігати і оптимальний шлях OptimalWay.

Удосконалення алгоритму можна зробити наступним чином: не дозволяти, щоб довжина Way була більше або дорівнює довжині OptimalWay. У цьому випадку, якщо і буде знайдений якийсь варіант, він свідомо не буде оптимальним. Таке удосконалення в загальному випадку означає, що як тільки поточний шлях стане свідомо неоптимальним, треба повернутися назад. Дане поліпшення алгоритму дозволяє в багатьох випадках сильно скоротити перебір. Демонстрація алгоритму перебору з поверненням

```
[label=code:pe,captionОпис= функції переборного алгоритму методом пошуку в глибину]
 / * Опис функції переборного алгоритму методом пошуку в глибину * /
void Backtracking (int n, int m, int ** Maze) {
  int Begin, End, Current;
  Begin = (n - 1) * m;
  End = m - 1;
  int * Way, * OptimalWay;
  int LengthWay, LengthOptimalWay;
  Way = new int [n * m];
  OptimalWay = new int [n * m];
  LengthWay = 0;
  LengthOptimalWay = m * n;
  for (int i = 0; i <n * m; i ++)
    Way [i] = OptimalWay [i] = -1;
  int * Dist;
  Dist = new int [n * m];
```

```
for (int i = 0; i <n; i ++)
  for (int j = 0; j <m; j ++)
    Dist [i * m + j] = (Maze [i] [j] == 0? 0: -1);
Way [LengthWay ++] = Current = Begin;
while (LengthWay> 0) {
  if (Current == End) {
    if (LengthWay <LengthOptimalWay) {
      for (int i = 0; i <LengthWay; i ++)
        OptimalWay [i] = Way [i];
      LengthOptimalWay = LengthWay;
     }
    if (LengthWay> 0) Way [- LengthWay] = -1;
    Current = Way [LengthWay-1];
   }
  else {
    int Neighbor = -1;
    if ((Current / m - 1)> = 0 &&! Insert (Way, Current - m) &&
      (Dist [Current - m] == 0 || Dist [Current - m]> LengthWay)
      && Dist [Current] <LengthOptimalWay)
        Neighbor = Current - m;
     else
      if ((Current% m - 1)> = 0 &&! Insert (Way, Current - 1) &&
        (Dist [Current - 1] == 0 || Dist [Current - 1]> LengthWay)
        && Dist [Current] <LengthOptimalWay)
          Neighbor = Current - 1;
       else
        if ((Current% m + 1) <m &&! Insert (Way, Current + 1) &&
         (Dist [Current + 1] == 0 || Dist [Current + 1]> LengthWay)
        && Dist [Current] <LengthOptimalWay)
          Neighbor = Current + 1;
        else
         if ((Current / m + 1) <n &&! Insert (Way, Current + m) &&
          (Dist [Current + m] == 0 || Dist [Current + m]> LengthWay)
         && Dist [Current] <LengthOptimalWay)
           Neighbor = Current + m;
    if (Neighbor! = -1) {
      Way [LengthWay ++] = Neighbor;
      Dist [Neighbor] = Dist [Current] + 1;
      Current = Neighbor;
     }
     else {
      if (LengthWay> 0) Way [- LengthWay] = -1;
      Current = Way [LengthWay-1];
     }
   }
 }
if (LengthOptimalWay <n * m)
  cout << endl << "Yes.␣Length␣way␣=" << LengthOptimalWay << endl;
```

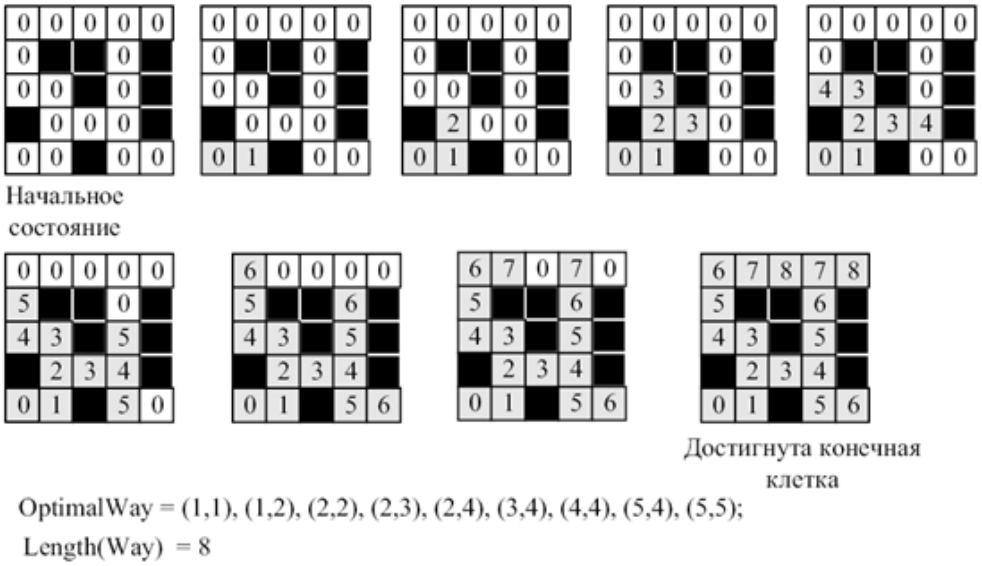


Рис. 5.4 – Демонстрація хвильового алгоритму

```
  else cout << endl << "No" << endl;
}
```

5.3.4 Хвильовий алгоритм

Цей переборний алгоритм, який заснований на пошуку в ширину, складається з двох етапів:

- ◇ поширення хвилі;
- ◇ зворотний хід.

Поширення хвилі і є власне пошук в ширину, при якому клітини позначаються номером кроку методу, на якому клітина відвідується. При зворотному ході, починаючи з кінцевої вершини, йде відновлення шляху, по якому в неї потрапили шляхом включення до нього клітин з мінімальною позначкою (рис. 5.4). Важливою особливістю є те, що відновлення починається з кінця (з початку воно найчастіше неможливо). Демонстрація хвильового алгоритму

Зауважимо, що перебір методом пошуку в ширину в порівнянні з перебором з поверненням, як правило, вимагає більше допоміжної пам'яті, яка необхідна для зберігання інформації, щоб побудувати шлях при зворотному ході і помітити відвідані вершини. Однак він працює швидше, оскільки абсолютно виключається відвідування однієї і тієї ж клітини більш ніж один раз.

5.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 6

Використання багаторозгалужених дерев

Мета роботи

6.1 Вступ

6.2 Ключові терміни

6.3 Розширені теоретичні відомості

6.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 7
Алгоритми сортування

7.1 Вступ

Алгоритмом сортування називається алгоритм для впорядкування деякого безлічі елементів. Зазвичай під алгоритмом сортування подразумевают алгоритм упорядкування безлічі елементів за зростанням або спаданням.

У разі наявності елементів з однаковими значеннями, у впорядкованій послідовності вони розташовуються поруч один за одним у будь-якому порядку. Однак іноді буває корисно зберігати первісний порядок елементів з однаковими значеннями.

Алгоритми сортування мають велике практичне застосування. Їх можна зустріти там, де мова йде про обробку та зберіганні великих обсягів інформації. Деякі завдання обробки даних вирішуються простіше, якщо дані заздалегідь впорядкувати.

7.2 Ключові терміни

7.3 Розширені теоретичні відомості

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

Практично кожен алгоритм сортування можна розбити на 3 частини:

1. порівняння, визначальне впорядкованість пари елементів;
2. перестановку, змінюють місце пару елементів;
3. власне сортують алгоритм, який здійснює порівняння і перестановку елементів до тих пір, поки всі елементи множини не будуть впорядковані.

7.3.1 Оцінка алгоритмів сортування

Жодна інша проблема не породила такої кількості найрізноманітніших рішень, як завдання сортування. Універсального, найкращого алгоритму сортування на даний момент не існує. Проте, маючи приблизні характеристики вхідних даних, можна підібрати метод, який працює оптимальним чином. Для цього необхідно знати параметри, за якими буде проводитися оцінка алгоритмів.

Час сортування - основний параметр, що характеризує швидкодію алгоритму.

Пам'ять - один з параметрів, який характеризується тим, що ряд алгоритмів сортування вимагають виділення додаткової пам'яті під тимчасове зберігання даних. При оцінці використовуваної пам'яті не враховуватиметься місце, яке займає вихідний масив даних і незалежні від вхідної послідовності витрати, наприклад, на зберігання коду програми.

Стійкість - це параметр, який відповідає за те, що сортування не змінює взаємного розташування рівних елементів.

Природність поведінки - параметр, якій вказує на ефективність методу при обробці вже відсортованих, або частково відсортованих даних. Алгоритм поводиться природно, якщо враховує цю характеристику вхідної послідовності і працює краще.

7.3.2 Класифікація алгоритмів сортувань

Все розмаїття і різноманіття алгоритмів сортувань можна класифікувати за різними ознаками, наприклад, по стійкості, по поведінці, по використанню операцій порівняння, за потреби в додатковій пам'яті, по потреби в знаннях про структуру даних, що виходять за рамки операції порівняння, та інші.

Найбільш докладно розглянемо класифікацію алгоритмів сортування за сферою застосування. У даному випадку основні типи впорядкування діляться наступним чином.

Внутрішнє сортування - це алгоритм сортування, який в процесі упорядкування даних використовує тільки оперативну пам'ять (ОЗП) на комп'ютері. Тобто оперативної пам'яті достатньо для завантаження в неї сортованого масиву даних з довільним доступом до будь-якій комірці і власне для виконання алгоритму. Внутрішня сортування застосовується у всіх випадках, за винятком однопрохідного зчитування даних і однопрохідної записи відсортованих даних. Залежно від конкретного алгоритму та його реалізації дані можуть сортуватися в тій же 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. Зовнішня сортування розроблена для обробки великих списків даних, які не поміщаються в оперативну пам'ять. Звернення до різних носіям накладає деякі додаткові обмеження на даний алгоритм: доступ до носія здійснюється послідовним чином, тобто в кожен момент часу можна вважати або записати тільки елемент, наступний за поточним; обсяг даних не дозволяє їм розміститися в ОЗУ.

Внутрішнє сортування є базовою для будь-якого алгоритму зовнішньої сортування - окремі частини масиву даних сортуються в оперативної пам'яті і за допомогою спеціального алгоритму зчіплюються в один масив, упорядкований по ключу.

Лекція № 8
Алгоритми внутрішнього сортування

8.1 Вступ

Внутрішня сортування - різновид алгоритмів сортування або їх реалізацій, при якій обсягу оперативної пам'яті достатньо для поміщення в неї сортованого масиву даних з довільним доступом до будь-якій комірці і, власне, для виконання алгоритму. У цьому випадку сортування відбувається максимально швидко, оскільки швидкість доступу до оперативної пам'яті значно вище, ніж до периферійних пристроїв (відповідно, час доступу значно менше). Залежно від конкретного алгоритму та його реалізації дані можуть сортуватися в тій же області пам'яті, або використовувати додаткову оперативну пам'ять. Внутрішня сортування є базовою для будь-якого алгоритму зовнішньої сортування - окремі частини масиву даних сортуються в оперативної пам'яті і за допомогою спеціального алгоритму зчіплюються в один масив, упорядкований по ключу.

8.2 Ключові терміни

8.3 Розширені теоретичні відомості

8.3.1 Сортування Вибором

Один з найпростіших методів сортування працює таким чином: знаходимо найменший елемент в масиві і обмінюємо його з елементом знаходяться на першому місці, потім повторюємо процес із другої позиції у файлі і знайдений елемент обмінюємо з другим елементом і так далі поки весь масив НЕ БУДЕ відсортований. Цей метод називається сортування вибором оскільки він працює циклічно вибираючи найменший з елементів, що залишилися.

Лістинг 8.1 – Сортування вибором

```
procedure selection;
var
i, j, min, t: integer;
begin
    for i: = 1 to N-1 do
    begin
        min: = i;
        for j: = i + 1 to N do
             if a [j] <a [min] then
min: = j;
        t: = a [min];
a [min]: = a [i];
```

```
a [i]: = t;
    end;
end;
```

У міру просування покажчика і зліва направо через файл, елементи зліва від покажчика знаходяться вже у своїй кінцевій позиції (і їх не більше вже не будуть чіпати), тому масив стає повністю відсортованим до того моменту, коли покажчик досягає правого краю.

Цей метод - один з найпростіших, і від працює дуже добре для невеликих файлів. Його "внутрішній цикл"складається з порівняння a [i] <a [min] (плюс код необхідний для збільшення j та перевірки на те, що він не перевищив N), що навряд чи можна ще спростити. Нижче ми обговоримо те, скільки швидше за все раз ці інструкції будуть виконуватися.

Більше того, незважаючи на те, що цей метод очевидно є методом "грубої сили він має дуже важливе застосування: оскільки кожен елемент пересувається не більше ніж раз, то він дуже хороший для великих записів з маленькими ключами. Це обговорюється нижче. Сортування вставкою

8.3.2 Сортування вставкою

- це метод який майже настільки ж простий, що і сортування вибором, але набагато більш гнучкий. Цей метод часто використовують при сортуванні карт: беремо один елемент і вставляємо його в потрібне місце серед тих, що ми вже обробили (тим самим залишаючи їх відсортованими).

Лістинг 8.2 – Сортування вставкою

```
type
Index = 0..n;
var
a: array [1..n] of elem;
procedure Insert;
    var i, j: index;
    x: elem;
    begin
    for i: = 1 to n do
    begin
        x: = a [i]; a [0]: = x; j: = i-1;
        while x.key <a [j] .key do
        begin
        a [j + 1]: = a [j]; j: = j-1;
        end;
        a [j + 1]: = x;
    end;
    end;
```

Також як і в сортуванні вибором, в процесі сортування елементи зліва від покажчика і знаходяться вже в сортованому порядку, але вони не обов'язково знаходяться у своїй останній позиції, оскільки їх ще можуть пересунути направо щоб вставити більш маленькі елементи

зустрінуті пізніше .. Однак масив стає повністю Сортувати коли покажчик досягає правого краю. Бульбашкова Сортування

8.3.3 Бульбашкове сортування

Необхідно проходити через масив, обмінюючи якщо потрібно елементи; коли на якомусь кроці обмінів не буде потрібно - сортування закінчена. Реалізація цього методу дана нижче.

Лістинг 8.3 – Бульбашкове сортування

```
procedure bubble;
var i, j, t: byte;
begin
    for i: = 2 to N do
        for j: = N down to i do
            if x [i-1]> x [j] then
                begin t: = x [j-1]; x [j-1]: = x [j]; x [j]: = t; end;
end;
end;
```

Щоб повірити в те, що вона насправді працює, може знадобитися деякий час. Для цього зауважте, що коли під час першого проходу зустрічаємо максимальний елемент, обмінюємо його з кожним елементом праворуч від нього поки він не опиниться у вкрай правої позиції. На другому проході поміщаємо другу максимальний елемент в передостанню позицію і так далі. Бульбашкова сортування працює також як і сортування вибором, хоча вона і робить набагато більше роботи на те, щоб перемістити елемент у його кінцеву позицію. Характеристики Найпростіших сортувань

Властивість 1 Сортування вибором використовує близько N2 / 2 порівнянь і N обмінів.

Властивість 2 Сортування вставкою використовує близько N2 / 4 порівнянь і N2 / 8 обмінів у середньому, і в два рази більше в найгіршому випадку.

Властивість 3 Бульбашкова сортування використовує близько N2 / 2 порівнянь і N2 / 2 обмінів у середньому і найгіршому випадках.

Властивість 4 Сортування вставкою линейна для "майже сортованих"файлів.

Властивість 5 Сортування вибором линейна для файлів з великими записами і маленькими ключами.

8.4 Приклади обчислень

Лекція № 9
Алгоритми зовнішнього сортування

9.1 Вступ

Зовнішнє сортування - сортування даних, розташованих на периферійних пристроях і не вміщаються в оперативну пам'ять, тобто коли застосувати одну з внутрішніх сортувань неможливо. Варто відзначити, що внутрішня сортування значно ефективніше зовнішньої, так як на звернення до оперативної пам'яті витрачається набагато менше часу, ніж до магнітних дисків, стрічок і т.п.

Найбільш часто зовнішнє сортування використовується в СУБД.

Дані, що зберігаються на зовнішніх пристроях, мають великий обсяг, що не дозволяє їх цілком перемістити в оперативну пам'ять, відсортувати з використанням одного з алгоритмів внутрішнього сортування, а потім повернути їх на зовнішній пристрій. У цьому випадку здійснювалося б мінімальну кількість проходів через файл, тобто було б однократне читання і однократна запис даних. Однак на практиці доводиться здійснювати читання, обробку і запис даних у файл по блоках, розмір яких залежить від операційної системи та наявного обсягу оперативної пам'яті, що призводить до збільшення числа проходів через файл і помітного зниження швидкості сортування.

9.2 Ключові терміни

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

Фаза - це дії по одноразовій обробці всієї послідовності елементів.

Серія (упорядкований відрізок) - це послідовність елементів, що упорядкована по ключу. Кількість елементів в серії називається довжиною серії.

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.

Двоколійним злиттям називається сортування, в якому дані розподіляються на два допоміжних файли.

Багатоколійним злиттям називається сортування, в якому дані розподіляються на N (N$>$ 2) допоміжних файлів.

Исходный файл f: **5 7 3 2 8 4 1**

	Распределение	***Слияние***
1 проход	*f1*: 5 3 8 1 *f2*: 7 2 4	*f*: 5 7 2 3 4 8 1
2 проход	*f1*: 5 7 4 8 *f2*: 2 3 1	*f*: 2 3 5 7 1 4 8
3 проход	*f1*: 2 3 5 7 *f2*: 1 4 8	*f*: 1 2 3 4 5 7 8

Рис. 9.1 – Демонстрація сортування двоколійному двофазним простим злиттям

9.3 Розширені теоретичні відомості

9.3.1 Сортування простим злиттям

Одне з сортувань на основі злиття називається простим злиттям.

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

У 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 - 2k.

Алгоритм сортування простим злиттям

1. Вихідний файл f розбивається на два допоміжних файлу f1 і f2.
2. Допоміжні файли f1 і f2 зливаються в файл f, при цьому поодинокі елементи утворюють впорядковані пари.
3. Отриманий файл f знову обробляється, як зазначено в кроках 1 і 2. При цьому впорядковані пари переходять у впорядковані четвірки.
4. Повторюючи кроки, зливаємо четвірки в вісімки і т.д., кожен раз подвоюючи довжину злитих послідовностей до тих пір, поки не буде впорядкований цілком весь файл (рис. 9.1).

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

Ознаками кінця сортування простим злиттям є наступні умови:

1. довжина серії не менша кількість елементів у файлі (визначається після фази злиття);
2. кількість серій рівно 1 (визначається на фазі злиття).
3. при однофазному сортуванні другий за рахунком допоміжний файл після розподілу серій залишився порожнім.

Приклад програмної реалізації надано у додатку Б.2;

9.4 Приклади обчислень

Лекція № 10

Характеристика алгоритмів порівняння методів сортування

Мета роботи

10.1 Вступ

10.2 Ключові терміни

10.3 Розширені теоретичні відомості

10.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 11
Алгоритми розподілу обчислювального процессу

Мета роботи

11.1 Вступ

11.2 Ключові терміни

11.3 Розширені теоретичні відомості

11.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 12

Архітектура розподіленних обчислень

Мета роботи

12.1 Вступ

12.2 Ключові терміни

12.3 Розширені теоретичні відомості

12.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 13
Процеси і потоки в обчисленні

Мета роботи

13.1 Вступ

13.2 Ключові терміни

13.3 Розширені теоретичні відомості

13.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 14

Реалізація багатозадачного середовища

Мета роботи

14.1 Вступ

14.2 Ключові терміни

14.3 Розширені теоретичні відомості

14.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 15

Бібліотеки організації розподілених обчислень

Мета роботи

15.1 Вступ

15.2 Ключові терміни

15.3 Розширені теоретичні відомості

15.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 16

Програмна модель OpenMP

Мета роботи

16.1 Вступ

16.2 Ключові терміни

16.3 Розширені теоретичні відомості

16.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 17
Конструкції OpenMP для розподілу робіт

Мета роботи

17.1 Вступ

17.2 Ключові терміни

17.3 Розширені теоретичні відомості

17.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 18

Умови виконання (планування)

Мета роботи

18.1 Вступ

18.2 Ключові терміни

18.3 Розширені теоретичні відомості

18.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 19
Бібліотечні функції OpenMP

Мета роботи

19.1 Вступ

19.2 Ключові терміни

19.3 Розширені теоретичні відомості

19.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 20

Бібліотека MPI

Мета роботи

20.1 Вступ

20.2 Ключові терміни

20.3 Розширені теоретичні відомості

20.4 Приклади обчислень

Контрольні запитання

1. ?

Лекція № 21
Режими обліку повідомлень в MPI

Мета роботи

21.1 Вступ

21.2 Ключові терміни

21.3 Розширені теоретичні відомості

21.4 Приклади обчислень

Контрольні запитання

1. ?

Додаток А

Правила оформлення звіту

А.1 Титульна сторінка лабораторної роботи

МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Звіт до лабораторної роботи №123

з дисципліни «Web-програмування»

Тема: **«Основи мережі Internet»**

Виконав
ст.групи хПР1 Пупкін А.А.

Перевірив
ст.викладач Іванов Б.Б.

Херсон 2012

A.2 Приклади блок-схем

Правила виконання блок-схем задані наступними документами:

- ГОСТ 19.701-90. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения
- ГОСТ 19.002-80. Схемы алгоритмов и программ. Правила выполнения
- ГОСТ 19.003-80. Схемы алгоритмов и программ. Обозначения условные графические

Додаток Б

Лістинги

Б.1 Програмна реалізація алгоритму Хаффмана за допомогою кодового дерева

Лістинг Б.1 – Програмна реалізація алгоритму Хаффмана

```
#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;//переменная для подсчета информация из строки
int k=0;
//счётчик количества различных букв, уникальных символов
int kk=0;//счетчик количества всех знаков в файле
int kolvo[256]={0};
//инициализируем массив количества уникальных символов
sym simbols[256]={0};//инициализируем массив записей
sym *psym[256];//инициализируем массив указателей на записи
float summir=0;//сумма частот встречаемости

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 << "Введите␣строку␣для␣кодирования␣:␣";
  cin >> String;
  sym *symbols = new sym[k];
  //создание динамического массива структур simbols
  sym **psum = new sym*[k];
  //создание динамического массива указателей на simbols
```

```
  Statistics(String);
  sym *root = makeTree(psym,k);
  //вызов функции создания дерева Хаффмана
  makeCodes(root);//вызов функции получения кода
  CodeHuffman(String,BinaryCode,root);
  cout << "Закодированная␣строка␣:␣" << endl;
  cout << BinaryCode << endl;
  DecodeHuffman(BinaryCode,ReducedString, root);
  cout << "Раскодированная␣строка␣:␣" << endl;
  cout << ReducedString << endl;
  delete psum;
  delete String;
  delete BinaryCode;
  delete ReducedString;
  system("pause");
  return 0;
}
//ркурсивная функция создания 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 {
    //внесение в нужное место массива элемента 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);
}
//рекурсивная функция кодирования 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);
  }
}
/*функция подсчета количества каждого символа и его вероятности*/
void Statistics(char *String){
  int i, j;
  //побайтно считываем строку и составляем таблицу встречаемости
  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;
      }
    }
  }
  // расчет частоты встречаемости
  for ( i = 0; i < k; i++)
    simbols[i].freq = (float)kolvo[i] / kk;
  // в массив указателей заносим адреса записей
  for ( i = 0; i < k; i++)
    psym[i] = &simbols[i];
  //сортировка по убыванию
  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);
}
//функция кодирования строки
```

```
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);
      }
  }
}
//функция декодирования строки
void DecodeHuffman(char *BinaryCode,char *ReducedString,
                   sym *root){
  sym *Current;// указатель в дереве
  char CurrentBit;// значение текущего бита кода
  int BitNumber;
  int CurrentSimbol;// индекс распаковываемого символа
  bool FlagOfEnd;  // флаг конца битовой последовательности
  FlagOfEnd = false;
  CurrentSimbol = 0;
  BitNumber = 0;
  Current = root;
  //пока не закончилась битовая последовательность
  while ( BitNumber != strlen(BinaryCode) ) {
    //пока не пришли в лист дерева
    while (Current->left != NULL && Current->right != NULL &&
           BitNumber != strlen(BinaryCode) ) {
      //читаем значение очередного бита
      CurrentBit = BinaryCode[BitNumber++];
      //бит - 0, то идем налево, бит - 1, то направо
      if ( CurrentBit == '0' )
        Current = Current->left;
      else
        Current = Current->right;
    }
    //пришли в лист и формируем очередной символ
    ReducedString[CurrentSimbol++] = Current->ch;
    Current = root;
  }
  ReducedString[CurrentSimbol] = 0;
}
```

Б.2 Опис функції сортування простим злиттям

Лістинг Б.2 – Програмна реалізація функції сортування простим злиттям

```
//Описание функции сортировки простым слиянием
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("\Исходный␣файл␣не␣может␣быть␣прочитан...");
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");
}
```