МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
Кафедра экономической информатики

Е.Н.Живицкая, А.А.Пасовец,
Т.М.Музычина, П.А.Корбит

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

по курсу «Алгоритмизация и программирование решения экономических задач»
для студентов экономической специальности

МИНСК 2001

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
Кафедра экономической информатики

Е.Н.Живицкая, А.А.Пасовец,
Т.М.Музычина, П.А.Корбит

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

по курсу «Алгоритмизация и программирование решения экономических задач»
для студентов экономической специальности

МИНСК 2001

УДК 002.5 +681.3(075.8)
ББК 22.1 Я 73
Л 12

Лабораторный практикум по курсу «Алгоритмизация и программирование решения экономических задач» для студентов экономической специальности.
Е.Н.Живицкая, А.А.Пасовец, Т.М.Музычина, П.А.Корбит. –Мн.:БГУИР, 2001-с.

ISBN 985-444-184-9

В практикуме представлен курс из 8 лабораторных работ, даны краткие теоретические сведения, примеры и варианты задания для лабораторных работ.

УДК 002.5 +681.3(075.8)
ББК 22.1 Я 73

ISBN 985-444-185-7
ISBN 985-444-184-9

СОДЕРЖАНИЕ

Лабораторная работа № 6

Методы сортировки

Цель: Изучить методы сортировки. Составить программу с использованием одного из методов сортировки и проанализировать выбранный метод.

Сортировка является одной из наиболее важных функций во многих приложениях. Ниже рассмотрено несколько методов, используемых для сортировки данных.

1. Пузырьковая сортировка

Пузырьковая сортировка – это один из простейших методов сортировки. Этот метод хорошо работает на простых структурах данных или если данные, которые нужно отсортировать, уже в некоторй степени отсортированы. В алгоритме пузырьковой сортировки выполняются последовательные перемещения через сортируемые записи. Во время каждого перемещения алгоритм сравнивает ключи элементов данных и меняет эти элементы местами, если они расположены не в желаемом порядке. Такие перемещения выполняются толькл между соседними элементами структуры данных. В результате тольтко один элемент помещается на свое правильное место после кажого перемещения. Отсортированные элементы не нуждаются в сравнении в последующих перемещениях. Ниже показан псевдокод для пузырьковой сортировки массива, состоящего из **n** элементов, расположенных в порядке возрастания:

```
For iteration=0 to (n-1)
Begin
    For I=0 to (n-1-iteration)
    Begin
        If array[i]>array[i+1] then
        Swap array[i] and array[i+1]
    End
End
```

Реализация этого алгоритма на языке C++ показана ниже.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>

class sort {
private:
    int *X;        //Список элементов данных
    int n;         //Количество элементов в списке
  public:
     sort (int size) { X = new int[n=size]; }
     ~sort( ) { delete [ ]X; }
     void load_list (int input[ ] );
     void show_list (char *title);
     void bubble_sort( int input[ ]);
};
```

```
void sort::load_list(int input[ ])
{
   for (int i = 0; i < n; i++)
      X[i] = input[i];
}

void sort::show_list( char *title)
{
      cout << "\n" << title;
   for (int i = 0; i < n; i++)
            cout << " " << X[i];
      cout << "\n";
}

void sort::bubble_sort( int input[ ])
{
   int swapped = 1;
   char *title;
   load_list(input);
   show_list("Список элементов, которые должны быть отсортированы в порядке
возрастания с использованием пузырьковой сортировки ");

// Цикл FOR выполняется один раз для каждого элемента массива.
// В конце каждой итерации один элемент "всплывает" на свое
// правильное положение и не учитывается в последующих итерациях.

   for ( int i = 0; i < n && swapped == 1; i++)
   {
// Если в конце итерации не нужно выполнять никаких перестановок.
// это означает. Что список уже отсортирован и цикл можно завершить.

      swapped = 0;
            for (int j = 0; j < n-(i+1) ; j++)
// Если X[j] > X[j+1], то элементы находятся в неправильном порядке.
// Следовательно, их нужно поменять местами.
      if  ( X[j] > X[j+1] )
      {
         int temp;
         temp = X[j];
         X[j] = X[j+1];
         X[j+1] = temp;
         swapped = 1;
      }

   }
   show_list("Отсортированный список элементов в порядке возрастания с использованием
пузырьковой сортировки ");
}

//main( ) : Тест пузырьковой сортировки
void main(void)
```

```
{
// Создание нового объекта с помощью метода bubble_sort
      sort sort_obj(5);
   static int unsorted_list[] = {54,6,26,73,1};
sort_obj.bubble_sort(unsorted_list);
}
```

Результат выполнения прграммы:

Список элементов, которые должны быть отсортированы в порядке возрастания с использованием пузырьковой сортировки 54 6 26 73 1

Отсортированный список элементов в порядке возрастания с использованием пузырьковой сортировки 1 6 26 54 73

Анализ пузырьковой сортировки

Пузырьковая сортировка обладает несколькими характеристиками:

• После каждой итерации только один элемент данных помещается в свою правильную позицию.

• При пузырьковой сортировке сравниваются и переставляются смежные элементы данных.

• В каждой итерации внутреннего цикла выполняется не более **(n-iteration-1)** перестановок.

• Худший случай — когда элементы данных отсортированы в обратном порядке.

• Лучший случай — когда элементы данных уже отсортированы в правильном порядке.

• Пузырьковая сортировка легко реализуется.

2. Быстрая сортировка

Быстрая сортировка — это наиболее эффективный алгоритм внутренней сортировки. Его производительность сильно зависит от выбора точки разбиения. При быстрой сортировке используются три стратегии:

1. Массив разбивается на меньшие подмассивы.
2. Подмассивы сортируются.
3. Отсортированные подмассивы объединяются.

Быструю сортировку можно реализовать несколькими способами, но цель каждого подхода заключается в выборе элемента данных и помещении его в правильную позицию (этот элемент называется точкой разбиения), таким образом, чтобы все элементы слева от точки разбиения оказались меньше (или предшествовали) точки разбиения, а все элементы справа от точки разбиения оказались больше (или следовали точки разбиения. Выбор точки разбиения и метод, используемый для разбиения массива, оказывают большое влияние на общую производительность реализации. Остановимся на рекурсивной реализации быстрой сортировки. Псевдокод этой реализации можно записать следующим образом:

1. Выбрать элемент данных и сделать его точкой разбиения таким образом, чтобы он разбивал массив на левый и правый подмассивы, как было описано выше.
2. Применить быструю сортировку к левому подмассиву.
3. Применить быструю сортировку к правому подмассиву.

Выбор точки разбиения имеет критическое значение. В эффективном методе разбиения может использо-ваться следующая стратегия:

1. Выбрать ключ первого элемента данных как точку разбиения. Другими словами, **Pivot = X[first]**.
2. Инициализировать два указателя поиска, **i** и **j**, чтобы **i = first** (наименьший индекс подмассива), а **j = last** (наибольший индекс подмассива).

3. Используя указатель поиска **i**, найти, начиная слева, элемент данных, который больше или равен точке разбиения. Это можно выполнить, используя следующий псевдокод:

Пока A[i] <= Pivot и i < last,
продолжать увеличение i на 1
иначе прекратить увеличение i

4. Используя указатель поиска **j**, найти, начиная справа, элемент данных, который меньше или равен точке разбиения. Это можно выполнить, используя следующий псевдокод:

Пока A[j] >= Pivot и j > last,
продолжать уменьшение j на 1
иначе прекратить уменьшение j

5. Если **i < j**, то поменять местами **A[i]** и **A[j]**.

6. Повторить пункты 2-4, пока не выполнится условие **i > j**.

7. Поменять местами точку разбиения и **A|j].**

После выполнения п.7 точка разбиения будет расположена в своей правильной позиции. Ниже показана реализация этого кода на языке C++.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>

class sort {
   private:
      int *X;          // Список элементов данных
      int n;           // Количество элементов в списке
   public:
       sort (int size) { X = new int[n=size]; }
       ~sort( ) { delete [ ]X; }
       void load_list (int input[ ] );
       void show_list (char *title);
       void quick_sort( int first, int last);
};

void sort::load_list(int input[ ])
{
   for (int i = 0; i < n; i++)
      X[i] = input[i];
}

void sort::show_list( char *title)
{
      cout << "\n" << title;
   for (int i = 0; i < n; i++)
           cout << " " << X[i];
      cout << "\n";
}

void sort::quick_sort( int first, int last)
{
// Переменная temp используется как временное хранилище при перестановке
```

```
        int temp;
    if (first < last)
    {
// Принимаем за точку разбиения первый элемент списка
        int pivot = X[first];
// Переменная i используется для просмотра слева
        int i = first;
// Переменная j используется для просмотра справа
        int j = last;
        while (i < j)
        {
// Поиск элемента, который больше или равен выбранной точке
// разбиения. Поиск слева
            while (X[i] <= pivot && i < last)
                i += 1;
// Поиск элемента, который меньше или равен выбранной
// точке разбиения. Поиск справа.
            while (X[j] >= pivot && j > first)
                j -= 1;
            if (i < j) //swap(X[i],X[j])
                        {
                            temp = X[i];
                            X[i] = X[j];
                            X[j] = temp;
                        }

        }
            //swap(X[j],X[first])
            temp = X[first];
            X[first] = X[j];
            X[j] = temp;

// Рекурсивное применение быстрой сортировки к двум частям массива
        quick_sort(first, j-1);
        quick_sort(j+1, last);
    }
}

//main( ) : Тест быстрой сортировки
void main(void)
{
   sort sort_obj (5);
   static int unsorted_list[] = {54,6,26,73,1};
        sort_obj.load_list(unsorted_list);
        sort_obj.show_list("Список элементов, которые должны быть отсортированы в
порядке возрастания с использованием быстрой сортировки ");
    sort_obj.quick_sort(0,4);
        sort_obj.show_list("Отсортированный список элементов в порядке возрастания с
использованием быстрой сортировки ");
}
```

Результат выполнения прграммы:

Список элементов, которые должны быть отсортированы в порядке возрастания с использованием быстрой сортировки 54 6 26 73 1

Отсортированный список элементов в порядке возрастания с использованием быстрой сортировки 1 6 26 54 73

Анализ быстрой сортировки

Быструю сортировку следует рассмотреть одной из первых при выборе метода внутренней сортировки. Этот алгоритм содержит сложную фазу разбиения и простую фазу слияния. В лучшем случае выполняется работа порядка $\mathbf{n\log_2 n}$; в худшем случае выполненная работа эквивалентна работе при сортировке выбором, т.е. $\mathbf{O(n^2)}$ [1]. Производительность быстрой сортировки сильно зависит от выбора точки разбиения.

3. Сортировка вставками

Сортировка вставками — очень простой метод сортировки, при котором элементы данных используются как ключи для сравнения. Этот алгоритм сначала упорядочивает **A[0]** и **A[1]**, вставляя **A[1]** перед **A[0]**, если **A[0] > A[1]**. Затем оставшиеся элементы данных по очереди вставляются в этот упорядоченный список. После **k**-й итерации элемент **A[k]** оказывается в своей правильной позиции и элементы от **A[0]** до **A[k]** уже отсортированы. Псевдокод для этого метода выглядит следующим образом:

```
For done = 0 to n-1
begin
        temp = array [done]
        for I = done -1 to 0
        begin
                while (array[i]>temp) {
                        array[i+1]=array[i]
                        I++;
                }
        end
        array[i+1]=temp
end
```

Реализация алгоритма сортировки вставками показана ниже.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>

class sort {
  private:
     int *X;          // Список элементов данных
     int n;           // Количество элементов в списке
     int scan_no;
  public:
      sort (int size) { X = new int[n=size]; }
      ~sort( ) { delete [ ]X; }
      void load_list (int input[ ] );
      void show_list (char *title);
```

[1] Эта запись называется О- записью. Характеризует зависимость алгоритмов от размера входных данных; n- размер входных данных.

```
        void insertion_sort( int input[ ]);
};

void sort::load_list(int input[ ])
{
   for (int i = 0; i < n; i++){
      X[i] = input[i];

      }
}

void sort::show_list( char *title)
{
      cout << "\n" << title;
   for (int i = 0; i < n; i++)
           cout << " " << X[i];
      cout << "\n";
}

void sort::insertion_sort( int input[ ])
{

   char *title;
// Массив S используется для хранения элементов в том виде, в каком они получены
      int S[100];
      load_list(input);
   show_list("Список элементов, которые должны быть отсортированы в порядке возрастания с использованием сортировки вставками ");

      S[0] = X[0];

// На каждой итерации цикла FOR элемент X[i] сравнивается с элементами
// в отсортированном списке S для поиска его положения в массиве S
      for (int i = 1; i < n ; i++)
   {
            int temp = X[i];
            int j = i-1;

            while (( S[j] > temp ) && ( j >= 0))
                     {
               S[j+1] = S[j];
               j--;
                     }

         S[j+1] = temp;

      }

// Метод show_list использует приватный массив X. Следовательно, мы копируем
// отсортированный массив S в X , чтобы его можно было напечатать
      for (int m = 0; m < n; m++)
           X[m] = S[m];
```

```
   show_list("Отсортированный список элементов в порядке возрастания с использованием сортировки вставками ");
}

void main(void)
{
   sort sort_obj(5);
   static int unsorted_list[] = {54,6,26,73,1};
   sort_obj.insertion_sort(unsorted_list);
}
```

Результат выполнения прграммы:

Список элементов, которые должны быть отсортированы в порядке возрастания с использованием сортировки вставками 54 6 26 73 1

Отсортированный список элементов в порядке возрастания с использованием сортировки вставками 1 6 26 54 73

Анализ сортировки вставками

Сортировка вставками обладает следующими характеристиками:

• После каждой итерации только один элемент помещается в свою правильную позицию.

• При сортировке вставками выполняется меньше перестановок, чем в пузырьковой сортировке.

• Наихудший случай – когда все элементы данных отсортированы в обратном порядке.

• Наилучший случай – когда элементы *почти* отсортированы в правильном порядке.

• Сортировка вставками легко реализуется.

Варианты задания:

1. Показать, как быстрая сортировка сортирует файл EASY QUESTION.
2. Используя метод пузырьковой сортировки отсортировать данные в порядке убывания. Для получения элементов сортировки использовать генератор случайных чисел (rand•), таких элементов получить не менее 50. Элементы должны храниться в отдельном файле.
3. Используя метод быстрой сортировки отсортировать данные в порядке возрастания. Для получения элементов сортировки использовать генератор случайных чисел (rand•), таких элементов получить не менее 40. Элементы должны храниться в отдельном файле.
4. Используя метод сортировки вставками отсортировать данные в порядке возрастания. Для получения элементов сортировки использовать генератор случайных чисел (rand•), таких элементов получить не менее 45. Элементы должны храниться в отдельном файле.
5. Используя метод пузырьковой сортировки отсортировать данные в порядке возрастания. Для получения элементов сортировки использовать значения функции $Y(x) = \cos(5x)$, полученные на интервале $[0, 2\pi]$ изменения аргумента x с шагом $h=2\pi/50$. Элементы должны храниться в отдельном файле.
6. Используя метод быстрой сортировки отсортировать данные в порядке убывания. Для получения элементов сортировки использовать значения функции $S(x) = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$, полученные на интервале $[0, 3\pi]$ изменения аргумента x с шагом $h=3\pi/45$. Элементы должны храниться в отдельном файле.

• Функция **int rand(void)** вычисляет последовательность псевдослучайных чисел в диапазоне значений 0-RAND_MAX. Функция **rand** возвращает псевдослучайное целое число.

7. Используя метод сортировки вставками отсортировать данные в порядке убывания. Для получения элементов сортировки использовать значения функции $S(x) = \sum_{k=0}^{n} \frac{\cos(\frac{\pi k}{4})}{k!} x^k$, полученные на интервале [0, 5π] изменения аргумента x с шагом h=5π/40. Элементы должны храниться в отдельном файле.
8. Используя метод пузырьковой сортировки отсортировать данные в порядке убывания. Для получения элементов сортировки использовать значения функции $Y(x) = \frac{1}{2} - \frac{\pi}{4}|\sin(x)|$, полученные на интервале [0, 7π] изменения аргумента x с шагом h=7π/60. Элементы должны храниться в отдельном файле.
9. Используя метод пузырьковой вставками, отсортировать текст, состоящий не менее, чем из 250 символов, в алфавитном порядке. Сортировку производить по порядковому номеру символа в алфавите. Текст должен храниться в отдельном файле.
10. Используя метод сортировки вставками, отсортировать текст, состоящий не менее, чем из 200 символов, в алфавитном порядке. Сортировку производить по порядковому номеру символа в алфавите. Текст должен храниться в отдельном файле.
11. Используя метод быстрой сортировки, отсортировать текст, состоящий не менее, чем из 250 символов, в алфавитном порядке. Сортировку производить по порядковому номеру символа в алфавите. Текст должен храниться в отдельном файле.
12. Используя пузырьковый метод сортировки, упорядочить сгенерированные в произвольном порядке 150 двоичных чисел по возрастанию. Сгенерированные данные хранить в отдельном файле.
13. Используя метод быстрой сортировки, упорядочить сгенерированные в произвольном порядке 200 шестнадцатиричных чисел по убыванию. Сгенерированные данные хранить в отдельном файле.
14. Используя метод сортировки вставками упорядочить сгенерированные в произвольном порядке 250 двоичных чисел по убыванию. Сгенерированные данные хранить в отдельном файле.
15. Используя метод сортировки вставками отсортировать неупорядоченную последовательность шестнадцатиричных чисел (не менее 150) в порядке возрастания. Последовательность для сортировки хранить в отдельном файле.
16. Используя пузырьковый метод сортировки, отсортировать неупорядоченную последовательность шестнадцатиричных чисел (не менее 240) в порядке возрастания. Последовательность для сортировки хранить в отдельном файле.

Лабораторная работа № 7

Алгоритмы поска данных

Цель: Изучить алгоритмы поиска данных. Разработать программу по одному из алгоритмов поиска данных.

Поиск – это процесс нахождения данных среди набора элементов. Который удовлетворяет определенному критерию.

1. Линейный поиск

Линейный поиск – это самый простой тип поиска, потому что при его выполнении просто просматривается множество элементов данных и эти элементы сравниваются с искомыми данными, пока не обнаружится совпадение. Линейный поиск прост в реализации, но не всегда эффективен. Предположим, что нужно найти определенный

элемент данных **S** в неотсортированном массиве **X** , состоящем из **N** целочисленных элементов. Этого можно достичь с помощью следующего псевдокода:

Пусть i=0

Сравнить X[i] с S. Если они совпадают, возвратить i, иначе увеличить i на 1.

Повторить шаг 2 и просмотреть массив, пока не обнаружится совпадение или пока не будет просмотрен весь массив.

Реализация этого алгоритма на языке C++ показана ниже.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>

class Search {
   private:
      int *X;          // Список элементов данных
      int N;           // Количество элементов
   public:
       Search (int size) { X = new int[N=size]; }
       ~Search( ) { delete [ ]X; }
       void load_list (int input[ ] );
       void show_list (char *title);
       int  linear_search( int S);
};

void Search::load_list(int input[ ])
{
   for (int i = 0; i < N; i++)
      X[i] = input[i];
}

void Search::show_list( char *title)
{
   cout << "\n" << title;
   for (int i = 0; i < N; i++)
      cout << " " << X[i];
   cout << "\n";
}

int Search::linear_search( int S)
{
   for ( int j = 0; j < N; j++)
   {   if  (X[j] == S)
      // Совпадение найдено
      // Возвратить j
         return(j);
   }
   return(-1);
}

//main( ) : Тест для линейного поиска
void main(void)
```

```
{
   int search_key;
   Search search_obj (10);
   static int list_to_search[] = {54, 6,26,73,1,100,36,41,2,83};

    cout << "\n"
<< " C++ реализация линейного поиска "
<< " \n";

   search_obj.load_list(list_to_search);

   cout  <<  "\n" << " Введите ключ для поиска:   ";
   cin  >>  search_key;

   search_obj.show_list("Просматривается следующий список элементов:  ");
   cout << "\n\n\n";

   int result = search_obj.linear_search(search_key);
   if (result != -1)
       cout << "\n" << " Результат поиска: "
<< "X[" << result << "] = "
<< search_key;
   else
       cout << "\n" << " Результат поиска: "
<< search_key
<< " Не найден в списке \n";

   cout << "\n\n\n";

}
```

Результат выполнения прграммы:
C++ реализация линейного поиска
Введите ключ для поиска:
1 "Enter"
Просматривается следующий список элементов: 54 6 26 73 1 100 36 41 2 83
Результат поиска: X[4]=1

Анализ линейного поиска

Алгоритм, представленный выше, является линейным, потому что в худшем случае он выполняет **n** сравнений; таким образом, он выполняет работу **O(n)**. Лучший случай — когда совпадение находится в самом первом сравнении. Средний случай: **O(n/2)=O(n)**.

2. Алгоритмы поиска на графе

Графы можно использовать для представления некоторых повседневных ситуаций. Например, вы запланировали поездку из Минска в Симферополь. Можно поставить следующий вопрос: Какой воздушный маршрут следует избрать, чтобы добраться из Минска в Симферополь быстрее всех? Еще одним примером служит составление расписания работ, когда есть множество задач и множество зависимостей между ними, согласно которым выполнение определенной задачи можно начать только после завершения одной или нескольких других задач.

Рассмотрим несколько простых определений, связанных с графами:

• **Граф**. Это совокупность узлов и соединений между ними.

• **Узлы**. Это объекты, с которыми могут быть связаны имена и другие свойства.

• **Связи (или дуги)**. Это соединение между двумя узлами.

• **Путь**. Путь из узла **A** в узел **B** — это список узлов, через которые необходимо пройти при переходе из узла **A** в узел **B**. Например, на рис. 7.1 **ABCD** и **ACD** — это два пути из узла **A** в узел **D**.

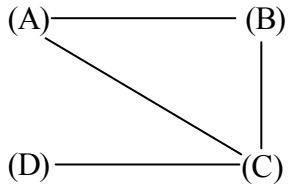

Рис. 7.1. Графическое представление задачи определения пути между узлами

• **Связный граф**. Граф называется связным, если существует путь из одного узла в любой другой узел в этом графе.

• **Неориентированный граф** - это граф, в котором соединения между узлами двунаправленны.

• **Ориентированный граф** - это граф, в котором соединения между узлами однонаправленны.

• **Степень узла A** - это количество инциндентных с ним узлов.

• **Простой путь**. Путь между двумя узлами является простым, если ни один узел в нем не повторяется. Другими словами, такой путь ни через один узел не проходит больше одного раза.

• **Цикл**. Это простой путь, в котором первый и последний узлы совпадают.

• **Дерево**. Это граф, который не содержит циклов.

• **Двоичное дерево**. Это дерево, в котором каждый узел соединен только с тремя другими узлами. Один из этих узлов является родителем данного узла, а два других являются дочерними узлами.

2.1. Поиск в глубину

Поиск в глубину называется также поиском с возвратом. Это название станет понятным из принципа работы этого метода. Рассмотрим неориентированный граф, который начинает свой обход с узла **A**. Предположим, что степень этого узла равна **d**. Это означает, что к узлу **A** прилегают узлы $\mathbf{A_i}$, где **i=l,2,3,...,d**. Основной принцип этого механизма состоит в пометке узлов, которые уже просмотрены. Поиск в глубину начинается с просмотра узла **A**, а затем со всех непросмотренных узлов, смежных с узлом **A**, например, $\mathbf{A_1}$. Остальные смежные узлы хранятся в стеке, чтобы их можно было просмотреть позже. После того как будут просмотрены все узлы, смежные с узлами, которые, в свою очередь, являются смежными с узлом $\mathbf{A_1}$, выполняется возврат назад и просмотр всех остальных непросмотренных смежных узлов $\mathbf{A_i}$, где **i=2,3,...,d**. Этот процесс продолжается до тех пор, пока не будут просмотрены все узлы графа. Поиск в глубину является примером полного перебора, потому что он позволяет исследовать все узлы графа для определения лучшего решения задачи. Поиск в глубину можно реализовать как рекурсивно, так и нерекурсивно. в виду, что поиск в глубину, по существу, ведется вниз по графу, пока не будет достигнут тупик узел, который не содержит ни одного непросмотренного смежного узла; после этого он начинает вестись снова вверх, чтобы можно было просмотреть другие дочерние (смежные) узлы родителя.

В следующем фрагменте программного кода C++ показана рекурсивная реализация поиска в глубину:

```
void Depth_first search(int n )
{
        int i;
        const int TRUE = 1;
        coast int FALSE = 0;
        int *checked;
        struct node
                {int nodeid; struct node *adj; );
        for ( i = 0; i <= n; i++) checked[i] =FALSE;
        for ( i = 0; i <= n; i++)
                it (checked[i] == FALSE) check(i) ;
}
 void check (int A)
{
        checked [A] = TRUE ;
        For all adjacent vertices Ai ( i = 1,2,3, ..... ,d) of node A
        {
                if (Ai is not yet checked)
                check(Ai) ;
        }
}
```

Метод поиска в глубину можно также показать графически (рис. 7.2).

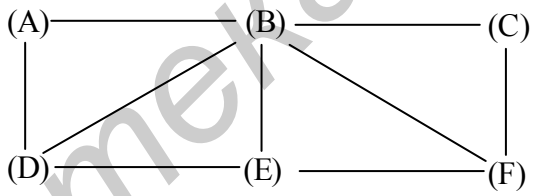

Рис. 7.2. Иллюстрация метода поиска в глубину

На рис. 7.2 поиск в глубину начинается с просмотра и печати узла **A**. С узлом **A** смежны узлы **B** и **D**, следовательно, теперь можно просмотреть любой из этих узлов. Предположим, что выбран узел **B**. При поиске в глубину после просмотра и печати узла **B** другие узлы, смежные с узлом **A** (в данном случае — **D**) оставляются для просмотра позже. С узлом **B** смежны узлы **A**, **C**, **D**, **E** и **F**. Узел **A** уже просмотрен, поэтому допустим, что выбран и просмотрен узел **C**. С узлом **C** смежны узлы **B** и **F**. Поскольку узел **B** уже просмотрен, то просматривается узел **F**. После этого просматривается узел **E** и наконец, узел **D**. Таким образом, узлы будут просмотрены в следующем порядке: **ABCFED**.

Путь, пройденный при поиске в глубину, не является уникальным. Если в начале поиска выбрать другой смежный узел, то будет пройден совсем другой путь.

2.2.Поиск в ширину

Поиск в ширину является альтернативным методом обхода графов. Это также метод полного перебора, т.е. обходятся все узлы графа. В отличие от поиска в глубину, при поиске в ширину сначала просматривают все смежные узлы $\mathbf{A_i}$ **(i=l,2,3,...,d)** данного узла **A**, прежде чем перейти к узлам, смежным с узлами $\mathbf{A_i}$. Этот метод также помечает просмотренные узлы для повторного просмотра,

Псевдокод поиска в ширину:

Создать объект queue_obj, который может функционировать как очередь.

Инициализировать этот объект очереди пустым значением.

Инициализировать массив checked[], чтобы все его элементы были равны FALSE.

Начнем с узла A. Присвоить checked [A] = TRUE.

Добавить узел A к началу очереди.

Выполнить следующие действия, пока объект очереди не пустой:

Присвоить current_node = head (queue_obj) .

Просмотреть все непросмотренные узлы, смежные с current_node.

Добавить все узлы, смежные с current_node, в очередь, чтобы их смежные узлы можно было просмотреть позже.

Освободить очередь queue_obj.

Этот метод можно проиллюстрировать на той же диаграмме, которая использовалась для объяснения работы поиска в глубину (см. рис. 7.2). Начнем, как и прежде, с узла **A**. После просмотра узла **A** просмотрим все его смежные узлы (в данном случае узлы **B** и **D**). Теперь можно просмотреть узлы, смежные с узлом **B** или с узлом **D**. Например, выберем узел **D**. После просмотра узлов, смежных с **D** (в данном случае узла **E**), просмотрим узлы, смежные с **B**. В результате получится путь поиска **ADBEFC**.

Как можно понять из псевдокода и рис. 7.2, в этом методе путь поиска также не уникален, а зависит от выбора следующего узла во время обхода.

Сравнение поиска в глубину и поиска в ширину

Метод поиска в глубину прост для реализации и может быстро привести к хорошему решению. Недостаток этого метода в том, что можно потратить много времени на просмотр путей, которые не приведут к успеху. Простейшей альтернативой поиску в глубину является поиск в ширину, который позволяет гарантированно найти наилучший путь к решению (если оно существует), если количество ветвей является конечным. Чтобы понять это, предположим, что есть путь длины **p** от исходного узла к конечному. При поиске в ширину сначала будут найдены все узлы на уровне 1, затем все узлы на уровне 2 и тд. Наконец, будут просмотрены все узлы на уровне **p**. Таким образом, будет найден не просто путь, а лучший путь.

Поиск в ширину обладает двумя недостатками:

• Он требует много памяти, поскольку количество узлов возрастает экспоненциально при просмотре последующих уровней.

• Если кратчайший путь длинный, то будет просмотрено очень много уровней, что потребует больше времени, чем при поиске в глубину, поскольку поиск в ширину должен проверить все узлы на определенном уровне перед тем, как перейти к следующему уровню.

Поиск в глубину лучше, чем поиск в ширину, в тех ситуациях, когда существует множество путей из исходного узла в конечный, но каждый из этих путей очень длинный.

3. Задача коммивояжера

Эта задача заключается в поиске ответа на вопрос среди очень большого количества возможных решений. Для таких задач поиска неприемлем линейный поиск, используемый в других типах задач, потому что задача коммивояжера потребовала бы огромного количества времени для выполнения линейного поиска.

Определение задачи. Дано множество из **N** городов. Найти кратчайший маршрут, соединяющий все города без посещения одного и того же города более одного раза.

Проведено множество исследований задач такого рода; на данный момент не существует эффективного метода их решения с использованием полного перебора, потому

что необходимо искать решение среди большого количества маршрутов, каждый из которых может быть очень длинный в зависимости от значения **N**.

Если предположить, что коммивояжер может перемещаться только между определенными парами городов, то эту задачу можно представить с помощью графа. Тогда задача будет состоять в поиске цикла. Существует несколько способов решения этой задачи, но ни один из них не является достаточно эффективным. Первый метод заключается в применении полного перебора. Полный перебор можно выполнить с помощью разновидности поиска в глубину, если не просто помечать просмотренные узлы, но и снимать пометки с узлов, если путь привел в тупик. Другими словами, выполняется поиск в глубину, и на просмотренных узлах ставятся пометки (чтобы не просматривать их повторно). Если текущий путь приводит в тупик, необходимо выполнить возврат и попробовать другой маршрут. Такой возврат требует снятия пометок с узлов, которые уже были просмотрены на маршруте. После этого можно попробовать другую ветвь. Такой полный перебор может потребовать очень много времени, особенно на полном графе.

Для сокращения количества путей используется множество различных методов. Они основаны на применении определенных проверок на узлах в целях исключения тех ветвей, которые заведомо не могут привести к правильному решению. В таких методах применяются различные формы возврата.

При попытке найти простой цикл в таких задачах можно исключить некоторые альтернативы, осознав, что циклы являются *симметричными*, т.е. всегда существует два пути, представляющих один и тот же маршрут, Это можно определить, применив ограничение, согласно которому три узла должны появляться в определенном порядке. Другими словами, из узлов **A**, **B** и **C** узел **B** должен идти после **A**, но перед **C**. При этом узел **C** будет проверяться только после проверки узла **B**.

Задачи поиска пути с наименьшей стоимостью можно выполнить более эффективно, если не продолжать просматривать путь, когда стоимость его части становится выше стоимости наилучшего найденного на данный момент. Поиск решения будет более эффективным, если проверять дочерние узлы данного узла в порядке возрастания стоимостей, чтобы найти оптимальное решение первым. Применяя возвраты и продуманные методы отсечения, можно получить эффективные результаты поиска.

4. Внешний поиск. Индексированный последовательный доступ

Операция поиска часто применяется к дисковым накопителям. На дисках хранятся файлы, и для доступа к ним требуется эффективный способ. В большинстве приложений, использующих дисковый ввод/вывод, скорость доступа к диску является самым критичным элементом, поэтому это время необходимо сократить до минимума.

Методы поиска можно применить и к дискам, но не все методы поиска достаточно эффективны. Например, метод линейного поиска позволяет просто просматривать ключи, пока не будет найдено совпадение или достигнут конец списка. Можно улучшить поиск, используя индекс, чтобы отследить, какие ключи принадлежат каким страницам на диске. Первая страница каждого диска может быть его индексной страницей. Этот подход можно улучшить, используя **главный индекс**, чтобы страница главного индекса содержала информацию о том, на каких дисках находятся различные ключи. Например, главный ключ может указывать, что ключи на первом диске меньше, чем **D**, ключи на втором диске находятся в интервале от **E** до **K** и т.д. Главный индекс может быть достаточно маленьким и размещаться в памяти, позволяя очень быстро обратиться к записи, выполняя для этого только два вида поиска — по главному индексу, чтобы найти диск, содержащий ключ, и по индексной странице на диске, чтобы найти, где на диске хранится запись. Этот метод объединяет методы индексирования с последовательным доступом и поэтому называется **индексированным последовательным доступом**. Недостаток этого метода

заключается в том, что при большом количестве обновлений индекса (при добавлении и удалении записей) для обслуживания индексов может потребоваться много времени.

Варианты задания:

1. Сформировать файл с исходным текстом. Исключить из текста группу символов, расположенных между скобками [,]. Сами скобки тоже должны быть исключены. Предполагается, что внутри каждой пары скобок нет других скобок. Для реализации использовать один из алгоритмов поиска данных.
2. Разработать программу, которая проверяла бы орфографию (правильность написания вводимых слов), сравнивая их со словами из словаря. Использовать при этом в качестве словаря одномерный массив слов (описать в программе в виде константы). Реализовать с использованием одного из алгоритмов поиска данных.
3. Разработать программу, которая читает построчно текст другой программы (хранится в отдельном файле) на языке C++, обнаруживает комментарии и выводит их на печать. Для реализации использовать один из алгоритмов поиска данных.
4. Составить программу, которая читает построчно текст другой программы (хранится в отдельном файле) на языке C++, считывает число символов "{" и "}", сравнивает их количество и выводит на печать сообщение об ошибке, если они не равны между собой. Реализовать с использованием одного из алгоритмов поиска данных.
5. Реализуйте поиск в глубину с использованием стека для графов, которые представлены списками смежности.
6. Реализуйте рекурсивный поиск в глубину для графов, которые представлены списками смежности.
7. Реализуйте поиск в ширину с использованием очереди для графов, представленных списками смежности.
8. Разработайте программу реализации задачи коммивояжера для графов, представленных списками смежности.
9. Приведите содержимое 3-4-5-6 дерева, образованного в результате вставки ключей EASY QUESTIONWITHPLENTY OF KEYS в указанном порядке в первоначально пустое дерево.
10. *Создайте программу для восходящего построения индекса B-дерева**, начиная с массива указателей страниц, содержащих от M до 2M упорядоченных элементов.
11. Реализуйте операцию sort для таблицы символов, основанной на использовании B-дерева.
12. Реализуйте операцию select для таблицы символов, основанной на использовании B-дерева.

---

* - задача повышенной трудности.

** B-дерево порядка M – это дерево, которое либо пусто, либо состоит из K-узлов с K-1 ключами и K связями с деревьями, представляющими каждый из K ограниченных ключами интервалов, и обладающее следующими структурными свойствами: K должно находиться в интервале между 2 и M для корня и между M/2 и M для любого другого узла; все связи с пустыми деревьями должны находиться на равном расстоянии от корня.

Лабораторная работа № 8

Хеширование

Цель: Изучить функции хеширования, способы разрешения конфликтов. Разработать программу с использованием основных методов хеширования.

Рассмотрим теперь другой вид поиска, использующий динамические структуры, на следующем методе: произведем над аргументом K некоторые арифметические действия, в результате получим функцию f(K), указывающую адрес, где хранится аргумент K и связанная с ним информация. Вычисляемая функция f(K) будет называться хеш-функцией.

Хеширование — это способ, который подразумевает использование значения ключа поиска для определения его позиции в таблице без сравнения с ключом. При хешировании используется произвольная функция отображения для определения местоположения любого элемента данных и обеспечения приблизительно равномерного распределения элементов данных в памяти, выделенной для выполнения хеширования.

Рассмотрим пример. Предположим, что некоторая фирма выпускает детали и кодирует их семизначными цифрами. Для применения прямой индексации с использованием полного семизначного ключа потребовался бы массив из 100 млн. элементов. Ясно, что это привело бы к потере неприемлемо большого пространства, поскольку совершенно невероятно, что какая-либо фирма может иметь больше чем тысяча наименований изделий. Поэтому необходим некоторый метод преобразования ключа в какое-либо целое число внутри ограниченного диапазона.

Тогда для хранения всего файла будет достаточно массива из 1000 элементов. Этот массив индексируется целым числом в диапазоне от 0 до 999 включительно. В качестве индекса записи об изделии в этом массиве используются три последние цифры номера изделия.

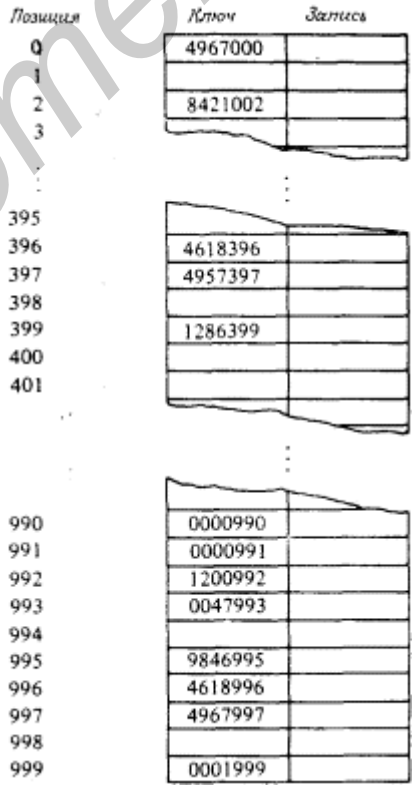

Рис. 8.1. Записи о деталях

Отметим, что два ключа, которые близки друг к другу как числа (такие как 4618396 и 4618996), могут располагаться дальше друг от друга в этой таблице, чем два ключа,. которые значительно различаются как числа (такие как 0000991 и 9846995). Это происходит из-за того, что для определения позиции записи используются только три последние цифры ключа.

Хеширование - это способ сведения хранения одного большого множества к более меньшему.

Функция, которая трансформирует ключ в некоторый индекс в таблице, называется хеш-функцией.

В данном случае h(key):= key mod 1000;

Хеш-таблица - это обычный массив с необычной адресацией, задаваемой хеш-функцией (см. рис. 8.1.).

Этот метод имеет один недостаток. Давайте добавим в таблицу запись с ключом 0596397. Увидим, что данная ячейка уже занята.

Ситуация, когда два или более ключа ассоциируются с одной и той же ячейкой называется коллизией при хешировании.

Следует отметить, однако, что хорошей хеш-функцией является такая функция, которая минимизирует коллизии и распределяет записи равномерно по всей таблице.

Совершенная хеш-функция - эта функция, которая не порождает коллизий.

Разрешить коллизии при хешировании можно 2 методами:

1.методом открытой адресации;

2.методом цепочек .

1. Разрешение коллизий при хешировании методом открытой адресации

Посмотрим, что произойдет, если мы захотим ввести в таблицу некоторый новый номер изделия 0596397. Используя хеш-функцию h(key):=key mod 1000, мы найдем, что h (0596397) =397 и что запись для этого изделия должна находиться в позиции 397 в массиве. Однако позиция 397 уже занята, поскольку там находится запись с ключом 4957397. Следовательно, запись с ключом 0596397 должна быть вставлена в таблицу в другом месте.

Самым простым методом разрешения коллизий при хешировании является помещение данной записи в следующую свободную позицию в массиве. Например, запись с ключом 0596397 помещается в ячейку 398, которая пока свободна, поскольку 397 уже занята. Когда эта запись будет вставлена, другая запись, которая хешируется в позицию 397 (с таким ключом, как 8764397) или в позицию 398 (с таким ключом, как 2194398), вставляется в следующую свободную позицию, которая в данном случае равна 400.

Если ячейка массива h(key) уже занята некоторой записью с другим ключом, то функция rh применяется к значению h(key) для того, чтобы найти другую ячейку, куда может быть помещена эта запись. Если ячейка rh(h(key)) также занята, то хеширование выполняется еще раз и проверяется ячейка rh(rh(h(key))). Этот процесс продолжается до тех пор, пока не будет найдена пустая ячейка. Rh - это функция повторного хеширования, которая воспринимает один индекс в массиве и выдает другой индекс.

Недостатки метода:

Во-первых, он предполагает фиксированный размер таблицы. Если число записей превысит этот размер, то их невозможно вставлять без выделения таблицы большего размера и повторного вычисления значений хеширования для ключей всех записей, находящихся уже в таблице, используя новую хеш-функцию.

Во -вторых, из такой таблицы трудно удалить запись.

2. Разрешение коллизий при хешировании методом цепочек

Он представляет собой организацию связанного списка из всех записей, чьи ключи хешируются в одно и то же значение (см. рис. 8.2.).

75 66 42 192 91 40 49 87 67 16 417 130 372 227

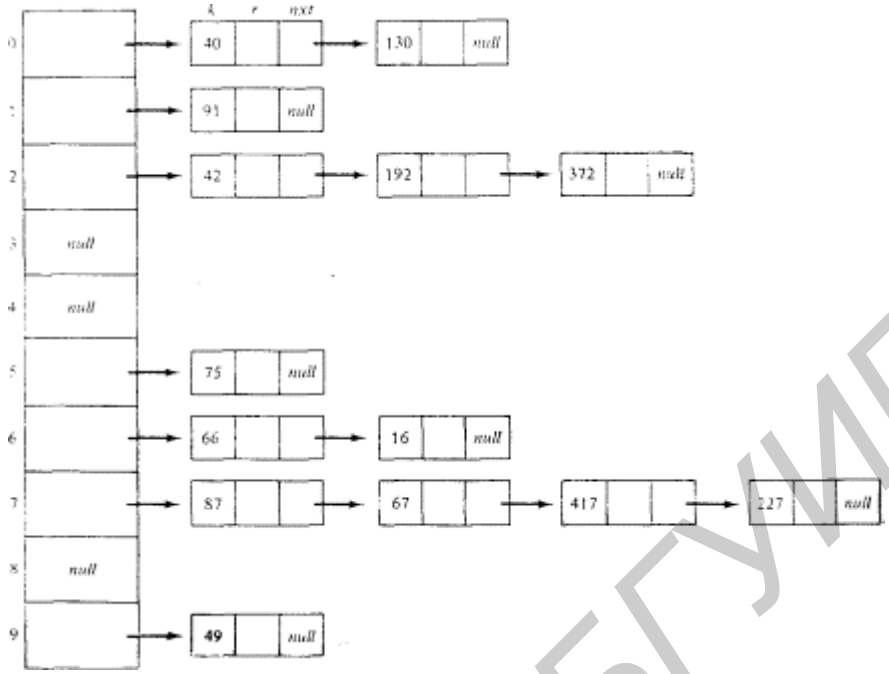

Рис. 8.2. Разрешение коллизий при хешировании методом цепочек

Удаление узла из таблицы, которая построена по методу цепочек, заключается просто в исключении узла из связанного списка. Удаленный узел никак не влияет на эффективность алгоритма поиска. Алгоритм будет работать так, как если бы этот узел никогда не вставлялся в таблицу. Отметим, что эти списки могут быть динамически переупорядочены для получения большей эффективности поиска.

Основным недостатком метода цепочек является то, что для узлов указателей требуется дополнительное пространство.

3. Выбор хеш-функции

Функция хеширования является важной частью процесса хеширования. Эта функция используется для преобразования ключей в адреса таблицы. Она должна легко вычисляться и преобразовывать ключи (обычно целочисленные или строковые значения) в целые числа в интервале от **0** до **TR-1** (где **TR** — это количество записей, которое может вместиться в таблице).

Ясно, что эта функция должна создавать как можно меньше коллизий при хешировании, т.е. она должна равномерно распределять ключи на имеющиеся индексы в массиве. Конечно, нельзя определить, будет ли некоторая конкретная хеш-функция распределять ключи правильно, если эти ключи заранее не известны. Однако, хотя до выбора хеш-функции редко известны сами ключи, некоторые свойства этих ключей, которые влияют на их распределение, обычно известны. Существуют следующие методы задания хеш-функции:

1. Метод деления. Некоторый целый ключ делится на размер таблицы и остаток от деления берется в качестве значения хеш-функции. Эта хеш-функция обозначается
h (key) := key mod m.

2. Метод середины квадрата. Ключ умножается сам на себя и в качестве индекса используется несколько средних цифр этого квадрата.

3. Аддитивный метод для строк (размер таблицы равен 256). Для строк вполне разумные результаты дает сложение всех символов и возврат остатка от деления на 256.

4. Исключающее ИЛИ для строк (размер таблицы равен 256). Этот метод аналогичен аддитивному, но успешно различает схожие слова и анаграммы (аддитивный метод даст одно значение для XY и YX). Метод заключается в том, что к элементам строки последовательно применяется операция "исключающее или". В алгоритме добавляется случайная компонента, чтобы еще улучшить результат.

Варианты задания:

1. Разработайте хеш-функцию для строковых ключей, основанную на идее одновременной загрузки n байтов с последующим выполнением арифметических операций сразу над 32 разрядами.
2. Создайте программу для вычисления функции статистического распределения χ^2 для хеш-значений N ключей при размере таблицы, равном M. Это число определяется равенством:

$$x^2 = \frac{M}{N} \sum_{0 \le i < M} (f_i - \frac{N}{M})^2 ,$$

где fi – количество ключей с хеш-значением i. Если хеш-значения являются случайными, значение этой функции статистического распределения для N>CM должно быть равно $M \pm \sqrt{M}$ с вероятностью 1-1/C.
3. Рассмотрите идею реализации модульного хеширования для целочисленных ключей с помощью соотношения (a*x/M), где a – произвольное фиксированное простое число. Приводит ли это изменение к достаточному перемешиванию разрядов, чтобы можно было использовать значение M, не являющееся простым числом?

Литература

1. Либерти Джесс. C++. Энциклопедия пользователя: Пер. с англ./ Джесс Либерти – К.: Издательство “ДиаСофт”, 200.-584 с.
2. Кнут Дональд Эрвин. Искусство программирования, том 3. Сортировка и поиск, 2-е изд.: Пер. с англ.: Уч. пос. – М.: Издательский дом “Вильямс”, 2000. – 832 с. : ил. – Парал. тит. англ.
3.