Федеральное агентство по образованию
Государственное образовательное учреждение
высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Е. А. Аксёнова, А. В. Соколов

Алгоритмы и структуры данных на C++

Учебное пособие

Петрозаводск
Издательство ПетрГУ
2008

ББК 22.183.49
А424
УДК 681.3.062

Рецензенты:
кандидат технических наук *О. Ю. Богоявленская*,
кандидат технических наук *Р. В. Воронов*

Печатается по решению редакционно-издательского совета
Петрозаводского государственного университета

Аксёнова Е. А.

А424 Алгоритмы и структуры данных на C++/ Е. А. Аксёнова, А. В. Соколов.– Петрозаводск: Изд-во ПетрГУ, 2008. – 81 с.

ISBN 978-5-8021-0869-7

В основу данного учебного пособия легли лекции, практические и лабораторные занятия, проводившиеся авторами в Петрозаводском государственном университете по дисциплине "Информатика" для студентов I курса математического факультета специальностей "прикладная математика и информатика" и "информационные системы и технологии". Цель данного курса – обучить студентов основным конструкциям широко распространенного объектно-ориентированного языка программирования C++, некоторым базовым структурам данных и алгоритмам их обработки.

Пособие рассчитано на студентов технических и физико-математических специальностей вузов.

ББК 22.183.49
УДК 681.3.062

ISBN 978-5-8021-0869-7

Содержание

Предисловие

Язык программирования C++, который является расширением языка C, получил в последнее время широкое распространение. Язык программирования C был создан сотрудниками фирмы Bell Labs. Он широко используется в мире как язык системного и в меньшей степени прикладного программирования. В 1980 году сотрудник той же фирмы Б. Строуструп в ряде работ рассмотрел возможность введения в язык C инструмента работы с абстрактными типами данных – классами. Продолжение работы в этом направлении привело к появлению языка C++, который в настоящее время стал широко использоваться как язык и системного, и прикладного программирования. Этим обоснован выбор языка C++ в качестве базового языка учебного пособия.

Изложение разделов, связанных с программированием задач со сложными структурами данных, опирается в основном на ставшую классической монографию Д. Кнута "Искусство программирования для ЭВМ". Изучение данной книги достаточно сложно для студентов. Вместе с тем, по мнению авторов, материал, изложенный в книге Д. Кнута, должен быть необходимым элементом обучения студентов университетов соответствующих специальностей. Также при изучении этих тем полезно обратиться к некоторым другим источникам [3–8].

В пособии не приводится формальное описание синтаксиса языка C++, а выбран принцип изложения материала, основанный на описании семантики конструкций с использованием русского языка и примеров программ на языке C++. В данном курсе не ставится задача обучения объектно-ориентированному программированию в полной мере, но вводятся некоторые языковые возможности работы с классами, необходимые для реализации изучаемых структур данных и алгоритмов. Рассмотрено управление доступом к компонентам класса, шаблоны (`template`), необходимые для реализации параметризованных классов и функций, перегрузка операций (придание нового смысла знакам операций) и некоторые другие возможности работы с классами. Предполагается, что читатель знаком с синтаксисом основных конструкций языка C, которые вошли в язык C++. Поэтому из множества пересечения этих языков мы кратко остановимся лишь

на работе с функциями, с указателями и со структурами. Отметим также, что в пособии нет описания какой-либо конкретной среды программирования. Лабораторные занятия проходят с использованием операционной системы Linux и свободно распространяемого транслятора языка C++.

Авторы выражают благодарность В. В. Антонову, сотруднику фирмы "Elcoteq Design Center", за ряд полезных замечаний, которые были учтены при изложении материала учебного пособия.

Глава 1. Некоторые конструкции языка C++

1.1. Передача параметров в языках C и C++

Рассмотрим простую задачу: по заданному значению стороны квадрата a вычислить по формулам $p = 4a$ и $s = a^2$ его периметр и площадь, используя соответствующую функцию. Опишем несколько способов передачи значения стороны квадрата в функцию и возврата вычисленных значений периметра и площади из функции.

Передача данных через глобальные переменные

Первый возможный способ передачи информации между функциями – это использование глобальных (описанных в начале программы и доступных всем функциям) переменных и функций без параметров.

```
#include <stdio.h>
 void f();
 float a,p,s;

 int main(){
             scanf("%f",&a);
             f();
             printf("a=%f p=%f s=%f",a,p,s);
           }

 void f()  {
             p=4*a;
             s=a*a;
           }
```

Передача данных через указатели в стиле языка C

Во втором способе мы используем передачу параметров по значению. В этом случае, чтобы вернуть из функции значения параметров `p` и `s`, эти формальные (описанные в функции) параметры надо описать как указатели и передать из функции `main` адреса соответствующих локальных переменных – фактических параметров (параметры, описанные там, где происходит обращение к функции, и которые каким-либо способом подставляются на места формальных параметров).

```
#include <stdio.h>
 void f(float a, float *p, float *s);

 int main(){
            float al,pl,sl;
            scanf("%f",&al);
            f(al,&pl,&sl);
            printf("a=%f p=%f s=%f",al,pl,sl);
           }

 void f(float a, float *p, float *s)
    {
     *p=4*a;
     *s=a*a;
    }
```

На рисунке 1.1 схематично изображена передача информации между функциями. Значение фактического параметра `al` передается формальному параметру `a`. Этот процесс изображен непрерывной линией. Формальным параметрам `p` и `s` передаются адреса соответственно фактических параметров `pl` и `sl`. Это изображено прерывистой стрелкой.

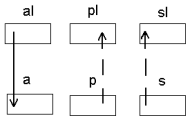

Рис. 1.1

Передача данных по ссылке в стиле языка C++

Ссылка – это разыменованный указатель, т. е. для доступа к значению переменной, расположенной по адресу, заданному ссылкой, не надо использовать операцию *. Можно рассматривать ссылки как новые имена переменных.

```
#include <stdio.h>
 void f(float a, float &p, float &s);

 int main(){
            float al,pl,sl;
            scanf("%f",&al);
            f(al,pl,sl);
            printf("a=%f p=%f s=%f",al,pl,sl);
           }

 void f(float a, float &p, float &s)
     {
      p=4*a;
      s=a*a;
     }
```

На рисунке 1.2 схематично изображена передача информации между функциями. Значение фактического параметра `al` передается формальному параметру `a`. Этот процесс изображен непрерывной линией. Формальные параметры `p` и `s` становятся новыми именами фактических параметров `pl` и `sl`. Это изображено прерывистой стрелкой.

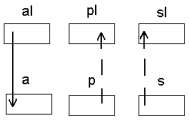

Рис. 1.2

Передача имени функции в качестве параметра

Рассмотрим пример функции, которая в качестве параметра получает имя другой функции.

Вычислим значение определенного интеграла по формуле трапеций

$$\int_a^b f(x)dx \approx h\left(\frac{f_0}{2} + f_1 + \ldots + f_{n-1} + \frac{f_n}{2}\right),$$

$$h = \frac{b-a}{n},\ f_0 = f(a),\ f_i = f(a+ih),\ f_n = f(b).$$

```
#include <stdio.h>
 float integr(float a, float b,int n, float f(float));
 float g(float);

 int main()
   {
     float a,b;
     int n;
     scanf("%f %f %d",&a,&b,&n);
                              // вызов функции integr
     printf("Значение интеграла=%f",integr(a,b,n,g));
     scanf("%d",&n);
    }

 float g(float x)
    {
     return x*x;// подынтегральная функция
    }

 float integr(float a, float b,int n, float f(float))
    {
     float x,s,h; int i;
     h=(b-a)/n;        // шаг
     s=(f(a)+f(b))/2;  // запоминаем значения
                       //функции в крайних точках
     x=a;
     for(i=1;i<n;i++)
         {             // двигаясь с шагом h
           x=x+h;      // вычисляем значения
           s=s+f(x);   // функции в точках разбиения
         }
```

```
    return h*s;       // возвращаем значение интеграла
   }
```

В этой программе подынтегральная функция `y=x*x` передается в функцию `integr` в качестве параметра. Для вычисления интеграла от другой функции требуется заменить функцию `float g(float x)` и перекомпилировать программу.

Эквивалентную программу можно получить, если заменить прототип функции `integr` на `float integr(float a, float b, int n, float (* f) (float))`. В качестве параметра в функцию `integr` передается указатель на подынтегральную функцию. Описание указателя на функцию требует задания типов параметров и типа возвращаемого значения. Например, `int(*f)(int,int)` – это указатель на функцию с двумя целыми параметрами, которая возвращает целое значение, а `int *f(int, int)` – это функция с двумя целыми параметрами, возвращающая указатель на целое значение.

1.2. Массивы и указатели

Рассмотрим несколько примеров программ работы с указателями и массивами. В следующем примере описан указатель `int *list`, затем ему с помощью оператора `new` присваивается адрес нового блока свободной памяти из трех элементов типа `int`, а затем выделенная память заполняется тремя значениями.

```
#include <stdio.h>
 int main()
   {
     int *list; // указатель на 0-й элемент массива list
     int i;
     list=new int[3];
     *list=421;                // значение 0-го элемента
     *(list+1)=53;             // значение 1-го элемента
     *(list+2)=1806;           // значение 2-го элемента
     printf("список адресов");
     for(i=0; i<3; i++)
         printf("%p", list+i);
```

```
    printf("список значений");
    for(i=0; i<3; i++)
        printf("%d", *(list+i));
    delete list;
}
```

На рисунке 1.3 изображены значения элементов в массиве list.

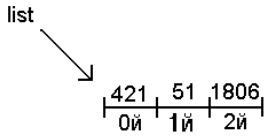

Рис. 1.3

В следующем примере выводятся значения некоторых указателей и значения элементов, расположенных по адресам, на которые указывают указатели. Также выводятся значения адресов указателей. Для сокращения записи вывод указателя, значения по его адресу и адреса этого указателя оформлен через директиву препроцессора `define`.

```
#include <stdio.h>
#define pr(x) printf("p=%v, *p=%d, &p=%u\n", x, *x, &x);

int main()
 {
    int a[]={100, 200, 300};
    int *p1;
    int *p2;
    p1=a;
    pr(p1);
    p1++;
    pr(p1);
    p2=&a[2];
    pr(p2);
    p2++;
    pr(p2);
 }
```

На выходе получаем:

```
p=65520, *p=100, &p=65518
p=65522, *p=200, &p=65518
p=65524, *p=300, &p=65516
p=65526, *p=0   , &p=65516
```

На рисунке 1.4 изображен массив `a` (значения элементов и их адреса).

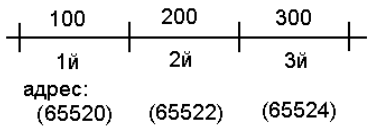

Рис. 1.4

Теперь вычислим сумму элементов массива, состоящего из `n` элементов, с помощью указателя `x`. Заметим, что работа с динамическими (состоящими из переменного числа элементов) массивами в языках C и C++ не допускается. Для работы с переменным числом элементов надо использовать указатели и операцию динамического (во время выполнения программы) выделения памяти `new`. Существует библиотека STL (Standard Template Library), в которой реализован класс `vector<>`, позволяющий работать с переменным числом элементов. Далее будет показано, что в языке C++ можно вводить новые типы данных и операции.

```
#include <stdio.h>
int main()
 {
    int *x,n,s;
    scanf ("%d", &n);
    s=0;
    x=new int[n];
    for(int i=0; i<n; i++)
      {
        scanf("%d", x+i);
        s=s+ *(x+i);
      }
    printf("s=%d", s);
```

```
}
```

На рисунке 1.5 изображены значения элементов в массиве x.

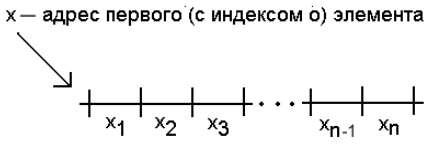

Рис. 1.5

1.3. Передача массивов и указателей в качестве параметров функций

Рассмотрим два варианта функции, вычисляющей сумму элементов массива (через массивы и указатели):

1) через массивы

```
#include <stdio.h>
int sum1(int x[], int n);

int main()
  {
    int y[100];
    int n, i;
    scanf ("%d", &n);
    for(i=0; i<n; i++)
     scanf ("%d", &y[i]);
    printf ("s=%d", sum1(y, n));
  }

int sum1(int x[], int n)
 {
    int s=0;
    for(int i=0; i<n; i++)
     s+=x[i];
    return s;
 }
```

2) через указатели

```
#include <stdio.h>
int sum2(int *x, int n);
int main()
  {
    int *y, n, i;
    scanf ("%d",&n);
    y=new int [n];
    for(i=0; i<n; i++)
      scanf ("%d",y+i);
    printf ("s=%d",sum2(y, n));
  }

int sum2(int *x, int n)
  {
    int s=0;
    for(int i=0; i<n; i++)
     s+=*(x+i);
    return s;
  }
```

В следующем примере вычисляются суммы элементов строк заданной матрицы с помощью массивов и указателей:

1) через массивы

```
#include <stdio.h>
int sum1 (int x[], int n);
int main()
{
 int a[4][3]={{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
 for(int i=0, i<n, i++)
 printf("Сумма эл-тов %d-й строки = %d",i,sum1(a[i],3));
}
```

В данной программе запись `a[i]` означает `& a[i][o]` (адрес начала i-й строки);

2) через указатели

```
#include <stdio.h> int sum2(int *x, int n);
int main()
{
int *a, i, j, m, n;
scanf("%d %d", m, n);
a = new int[m*n];
for(i=0, i<m, i++)
 {
 printf("\n");
 for(j=0, j<n, j++)
 scanf("%d", a+i*n+j);
 printf("Сумма эл-тов %d-й строки = %d",i,sum2(a+i*n,n));
 }
}
```

1.4. Структуры

Структуры служат для объединения переменных разных типов в единый тип. В языке C структуры – это аналог паскалевских записей. Понятие структуры в языке C++ приближено к понятию класса и сохранено в языке для удобства работы программистов.

В этом пункте рассмотрены некоторые методы работы со структурами, а работа с классами будет рассмотрена позже. Дело в том, что техника работы со связными списками будет рассмотрена с использованием структур, а не классов, что проще для первоначального изучения.

Опишем структуру для работы с комплексными числами, состоящую из действительной и мнимой частей числа:

`struct complex{float re; float im;}`.

Теперь при объявлении `complex x` – это переменная типа `complex`, `complex xarr[10]` – массив из 10 элементов типа `complex`, `complex *xptr` – указатель на переменную типа `complex`, `complex &x` – ссылка на структуру.

Рассмотрим пример программы для работы с комплексными числами.

```
#include <stdio.h>
```

```
struct complex{float re; float im;}; //описание структуры,
                                     //re-вещественная часть
                                     //im-мнимая часть
 int main()
  {
     complex y,x = {1.5, 2.4};     //описываем y и x
                                   //с заданным значением
     y=x;                          //присваиваем y значение x
     //выводим re,im-элементы  структуры (компоненты)
     printf("re=%f, im=%f", y.re, y.im);
  }
```

1.5. Передача структур в качестве параметров функции

Рассмотрим несколько способов реализации функции, вычисляющей сумму двух комплексных чисел.

Первый способ

В примере используется функция сложения двух вещественных чисел отдельно для действительной и мнимой частей.

```
#include <stdio.h>
struct complex{float re; float im;};
float sum(float, float);

int main()
 {
    complex z,x={1.4, 2.5}, y=(2.6, 3.1};
    z.re=sum(x.re, y.re);
    z.im=sum(x.im, y.im);
    printf("re=%f im=%f", z.re, z.im);
 }

float sum(float a, float b)
 {
  return a+b;
 }
```

Второй способ

В примере реализована функция сложения двух комплексных чисел, в которую входные структуры передаются по значению, а сумма описана как указатель на структуру, и в функцию передается ее адрес.

```
#include <stdio.h>
struct complex{float re; float im;};
void sum(complex, complex, complex*);

int main()
{
 complex z, x={1.4, 2.5}, y={2.6, 3.1}; //описания переменных
 sum (x, e, &z);
 printf("re=%f im=%f", z.re, z.im);     // вывод
}

void sum (complex a, complex b, complex *c)
 {
  c -> re = a.re + b.re; // c->re эквивалентно (*c).re
  c -> im = a.im + b.im;
 }
```

Третий способ

В примере реализована функция сложения двух комплексных чисел, в которой все параметры описаны как указатели на структуры.

```
#include <stdio.h>
struct complex{float re; float im;};
void sum (complex *, complex *, complex *);

int main()
 {
    complex z, x={1.4, 2.5}, y={2.6, 3.1};
    sum (&x, &y, &z);
    printf("re=%f im=%f", z.re, z.im);
 }

void sum(complex *a, complex *b, complex *c)
```

```
  {
     c -> re = a->re + b->re;
     c -> im = a->im + b->im;
  }
```

Четвертый способ

В примере все параметры передаются по ссылке на структуру.

```
#include <stdio.h>
struct complex{float re; float im;};
void sum(complex&, complex&, complex&);

int main()
  {
     complex z, x={1.4, 2.5}, y={2.6, 3.1};
     sum(x, y, z);
     printf("re=%f im=%f", z.re, z.im);
  }

void sum(complex& a, complex& b, complex& c)
  {
     c.re = a.re + b.re;
     c.im = a.im + b.im;
  }
```

Пятый способ

Этот вариант отличается от предыдущего тем, что сумма возвращается через значение функции, а не как параметр. Входные параметры передаются по значению.

```
#include <stdio.h>
struct complex{float re; float im;};
complex sum(complex, complex);

int main()
  {
    complex z, x={1.4, 2.5}, y={2.6, 3.1};
    z=sum(x,y);
    printf("re=%f im=%f", z.re, z.im);} // вывод
  }
```

```
complex sum(complex a, complex b)
       {
        complex c;
        c.re = a.re + b.re;
        c.im = a.im + b.im;
        return c;
       }
```

Заметим, что передача структуры по значению подразумевает создание копии. Поэтому для больших структур способ передачи параметров по значению неэффективный.

Глава 2. Линейные структуры данных

2.1. Линейные списки

Линейным списком называется упорядоченная последовательность переменного числа элементов. Основными операциями с линейными списками являются:

1) получить доступ к k-му элементу;
2) включить k-й элемент;
3) исключить k-й элемент;
4) найти необходимый элемент.

Существует два принципиально разных способа представления линейных списков – последовательное и связное (рис. 2.1).

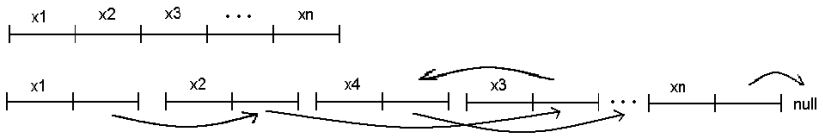

Рис. 2.1

При последовательном представлении элементы списка расположены в последовательных элементах памяти, а при связном – в произвольных элементах, их упорядоченность поддерживается при помощи связей. Связное представление требует дополнительную память для связей, но иногда при связном можно получить неявный выигрыш, совмещая общие части таблицы. Связное представление применяется не только для линейных списков, но также для более общих структур данных.

Некоторые операции со списками эффективнее реализуются при связном представлении, а некоторые – при последовательном, так например:

1) при связном представлении легко включать и исключать элементы, при последовательном для этого необходимо сдвигать информацию;
2) доступ к k-му элементу быстрее при последовательном представлении;
3) при связном представлении легко объединять списки и разбивать их на части.

На практике часто применяются некоторые частные случаи линейных списков.

Стек (LIFO (Last In First Out) список) – это линейный список, в котором все включения и исключения (и обычно всякий доступ) элементов происходят на одном конце списка.

Очередь (FIFO (First In First Out) список) – это линейный список, в котором все включения элементов происходят на одном конце списка, а все исключения (и обычно всякий доступ) – на другом.

Дек – это линейный список, в котором все включения и исключения элементов производятся на обоих концах списка.

2.2. Последовательное представление линейных списков

Стек

Для последовательного представления одного стека достаточно иметь указатель *top* на вершину стека (рис. 2.2).

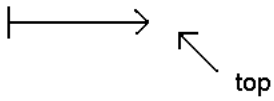

Рис. 2.2

При работе с двумя стеками возможны два способа организации (рис. 2.3):

1) выделить каждому стеку отдельную область памяти;
2) расположить стеки в памяти так, чтобы они росли навстречу друг другу (этот способ является оптимальным).

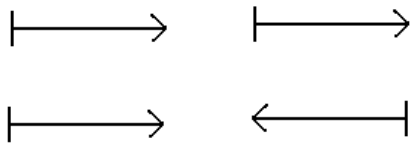

Рис. 2.3

При работе с тремя стеками возможны следующие способы реализации (рис. 2.4):

1) выделить каждому стеку отдельную область памяти;
2) расположить два стека так, чтобы они росли навстречу друг другу, а третий расположить отдельно;
3) расположить два стека так, чтобы они росли навстречу друг другу с концов выделенной для работы области памяти, а третий стек будет расти одновременно навстречу им обоим.

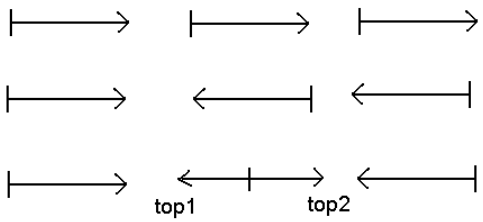

Рис. 2.4

Очередь

При последовательном представлении очереди необходимо хранить указатели на начало F и конец очереди R (рис. 2.5). Если указатель на

конец или начало доходит до конца области памяти, то он перемещается на начало области памяти (рис. 2.6). Такая очередь называется *циклической*.

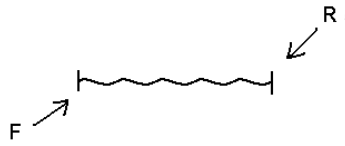

Рис. 2.5

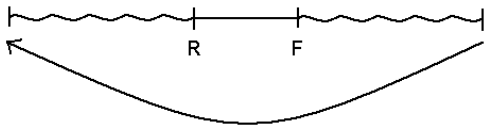

Рис. 2.6

При работе с двумя очередями возможны два способа организации:

1) выделить каждой очереди отдельный кусок памяти (рис. 2.7);

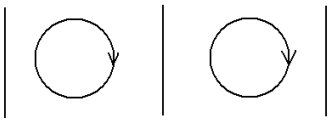

Рис. 2.7

2) очереди двигаются друг за другом по кругу (рис. 2.8) (этот метод работы с очередями предложен в [7]).

2.3. Связное представление линейных списков

Для связного представления стека используется связный список, в котором элементы включаются и исключаются в начале списка. Для

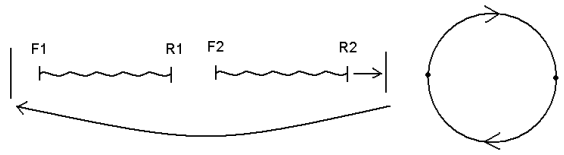

Рис. 2.8

связного представления очереди используется связный список, в котором элементы включаются в конец списка, а исключаются в начале списка (рис. 2.9).

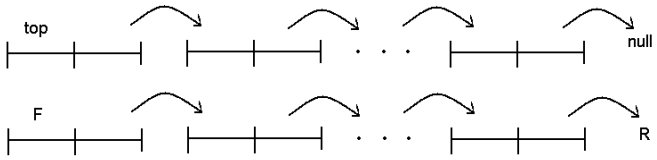

Рис. 2.9

Циклические списки

На рисунке 2.10 представлен циклический список (R – указатель на правый элемент):

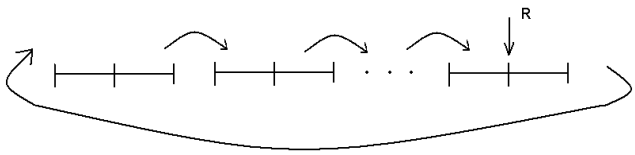

Рис. 2.10

Рассмотрим основные операции с циклическими списками:

1) включить новый элемент слева (рис. 2.11);
2) включить новый элемент справа: включить новый элемент слева, указателю R присвоить адрес нового элемента;
3) исключить левый элемент (рис. 2.12).

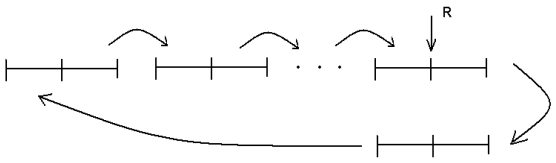

Рис. 2.11

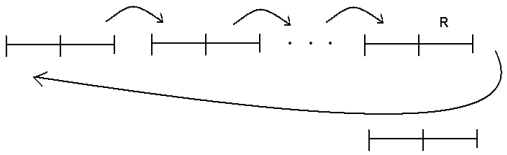

Рис. 2.12

Операции 1, 3 дают стек; операции 2, 3 дают очередь.

Списки с двумя связями

Структура в этом случае содержит информационную часть и указатели на левый и правый элементы (рис. 2.13). В списках с двумя связями можно включать и исключать элементы, зная только адрес элемента. Также возможен проход по списку вперед и назад.

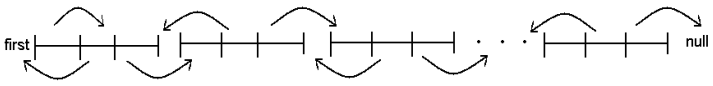

Рис. 2.13

```
struct node{char info; node *llink; node *rlink;}
```

Нестандартные методы представления связных списков

Рассмотрим два способа представления связных списков, которые позволяют проходить список в обоих направлениях, используя для представления меньший размер памяти.

- Рассмотрим представление циклического списка [1], в котором возможен проход вперед и назад, хотя есть только одно поле

связи в каждом элементе. Обозначим $link(x_i)$ – поле связи узла x_i, $loc(x_i)$ – адрес i-го узла. В рассматриваемом представлении списка в каждом узле истинно равенство $link(x_i) = loc(x_{i+1}) - -loc(x_{i-1})$ (рис. 2.14).

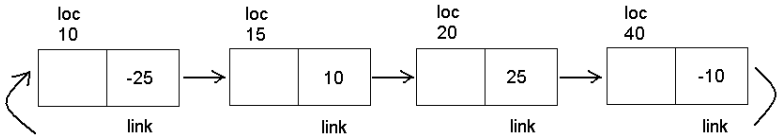

Рис. 2.14

Таким образом, если известен адрес текущего узла и адрес предшествующего узла, то можно вычислить адреса всех узлов при прохождении списка в обоих направлениях.

- Списки с 1.5 связями [7]. В этом представлении используются узлы с двумя связями (ссылки вперед и назад) и с одной связью (ссылки назад через один) (рис. 2.15). В таком списке возможен проход вперед и назад, но памяти на ссылки нужно в полтора раза меньше, чем у двухсвязных списков.

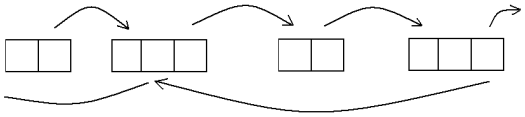

Рис. 2.15

Примеры применения связных списков

Наиболее часто используемое применение связных списков – это хранение переменного числа элементов. Такие задачи возникают в системном программировании, в задачах искусственного интеллекта и других приложениях.

Связные списки могут применяться для хранения разреженных матриц (матриц с большим количеством нулевых элементов).

Рассмотрим матрицу:

$$\begin{pmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 6 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Для хранения этой матрицы можно использовать несколько методов:

1) хранить по строкам (хранить адреса указателей на ненулевые элементы строк и списки ненулевых элементов строк, в которых хранятся элементы матрицы и номера столбцов этих элементов) (рис. 2.16);

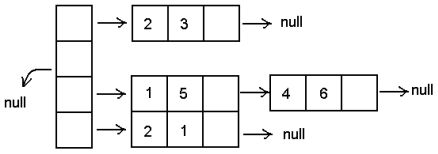

Рис. 2.16

2) хранить по столбцам (аналогично) (рис. 2.17).

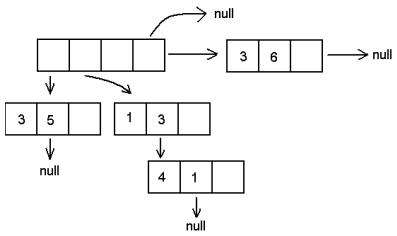

Рис. 2.17

2.4. Реализация алгоритмов работы с односвязным списком

Рассмотрим задачу: создать связный список, содержащий фамилии и номера телефонов ряда абонентов (рис. 2.18), и написать ряд

функций для работы с ним. Заметим, что в некоторых функциях реализуются только основные идеи. Для получения практических вариантов программ необходимо добавить некоторые проверки на пустоту или на окончание списка.

Функция ввода `m` элементов списка

```
# include<iostream.h>

struct node{char fam[20]; int n; node *link;};

node* create()
{
   int m, i;
   node *first, *q;
   cin>>m;         //вводим количество элементов списка
   first=NULL;     //пока нет элементов

   for(i=0; i<m; i++)   //в цикле считываем все данные
     {
         q=new node;    //выделяем память под узел
         cin>>q->fam;   //считываем фамилию
         cin>>q->n;     //считываем номер
         q->link=first; //ссылку на следующий элемент
                        //направляем туда, куда
                        //раньше указывал first,
         first = q;     //а first на новый элемент
     }
   return first; //возвращаем указатель на
                 //первый элемент в списке
}
```

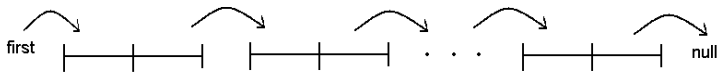

Рис. 2.18

Функция вывода всех элементов списка

```
void output(node *first)
{
   node *p;
   p=first;         //адрес первого

   while(p!=NULL)
    {
      cout<<p->fam; //выводим фамилию
      cout<<p->n;   //выводим номер
      p=p->link;    //переход на адрес следующего элемента
    }
}
```

Функция вставки элемента после элемента с адресом `q`

```
void insert(char s[20], int k, node *q, node*& first)
{
   node *p;
   p=new node;
   p->fam=s;
   p->n=k;

  if(first==NULL)
    {    p->link=NULL;
         first=p;
    }
  else
    {
         p->link=q->link;
         q->link=p;
    }
}
```

Удаление элемента, следующего после элемента с адресом `q`

```
void del(node *q)
{
    node *p;
```

```
    p=q->link;
    q->link=p->link;
    delete p;
 }
```

Обращение связного списка

Эта функция за один проход связного списка обращает порядок следования связей в списке.

```
node *reverse(node *first) //передаем указатель
{                           //на первый элемент
   node *t;
   node *y=first;
   node *r=NULL;

   while(y!=NULL)
     {
       t=y->link;
       y->link=r;
       r=y;
       y=t;
     }
  return r; //возвращаем новый указатель
            //на первый элемент
}
```

Для обращения к функциям в основной программе нужно написать, например:

```
int main()
{
      node *first;
      first=list();
      put(first);
      first=reverse(first);
      put(first);
}
```

Глава 3. Классы

3.1. Основные определения

Любой язык программирования можно мыслить себе как множество типов данных и множество операций над этими типами данных. В каждом языке программирования, в зависимости от его предназначения, выбираются те или иные стандартные (базовые) типы данных, операции над которыми реализуются в трансляторе (программе перевода с языка программирования на язык машины), а также предоставляется механизм образования новых (пользовательских) типов данных и операций. Например, двухместная операция + (сложение) определена в C++ для некоторых встроенных типов (`int`, `float` и др.), но можно ее определить и для типа `complex` (комплексные числа), который не является встроенным типом языка C++.

Класс – это определенный пользователем тип. Объявление класса задает представление объектов (переменных) этого класса и набор операций, которые можно применять к таким объектам.

Рассмотрим несколько примеров описания различных классов. Для описания элементов связных списков можно было вместо структуры `struct` использовать следующее описание:

```
class node{node* link; char info;}.
```

Описание класса может быть пустым, при этом объекты пустого класса получают разные адреса.

```
class empty{ };

empty e1, e2;
empty *p1=&e1;
```

```
void f()
     {
        if(p1 == &e2)
           cout<< "не может быть" ;
     }
```

Пустые классы могут служить "затычками" при разработке программ. Объявление класса вводит новый тип. Например, объявим три переменные трех разных типов:

```
class X{int a;};
class Y{int a;};

X a1;
Y a2;
int a3;

a1=a2;       //ошибка: Y присваивается X
a1=a3;       //ошибка: int присваивается X
```

Члены класса не могут быть автоматическими, внешними или регистровыми. Автоматические переменные доступны только в своем блоке (функции), после выхода из блока они исчезают (по умолчанию все переменные автоматические). Внешние переменные доступны во всей программе. Если в компьютере есть быстрые регистры, то транслятор будет пытаться разместить регистровые переменные в регистрах.

```
auto int x;      //автоматическая переменная
extern int y;    //внешняя переменная
register int r;  //регистровая переменная
```

3.2. Управление доступом к членам класса

Член класса может быть:

1) приватным `private`, т. е. его имя может употребляться лишь внутри функций-членов класса и друзей класса, в котором этот член объявлен;

2) защищенным `protected`, т. е. его имя может употребляться лишь внутри функций-членов и друзей этого класса, и производных от него классов;
3) общедоступным `public`, т. е. имя может употребляться внутри любой функции.

В приведенном ниже примере класса все члены класса явно описаны как общедоступные. Функции можно реализовывать прямо в классе.

```
class X
 {
  public:
     int i;
     char name[14];
     X *ptr1;
     X *ptr2;
     void xfunc(char *data, X *left, X *right) { ... }
 };
```

Если реализация функции имеет достаточно большой размер, то более наглядный вид будет иметь представление, при котором прототип находится в классе, а реализация вынесена из класса.

```
class X
 {
    public:
       int i;
       char name[14];
       X *ptr1;
       X *ptr2;
       void xfunc(char *data, X *left, X *right);
 };

void X::xfunc(char *data, X *left, X *right) { ... }
```

Пример обращения к членам класса:

```
void f(void)
{
```

```
 X x1, x2, *xptr=&x2; //указатель на класс, инициализируемый
                      //адресом объекта x2
 x1.i = 0;            //доступ к компонентам объекта x1
 x2.i = x1.i;         //доступ к компонентам объекта x2
 xptr -> i=1;
 x1.xfunc("Stan",&x2,xptr);   //обращение к функции xfunc
 xptr->xfunc("авс",&x1,xptr); //обращение к функции xfunc
 }
```

Члены класса, объявленного с ключевым словом `class`, являются по умолчанию `private`.

```
class X
 {
            int a; // private по умолчанию
    public:
            int b; // public b
            int c; // public c
    private:
            int d; //private d
 };
```

Члены класса, объявленного с ключевым словом `struct` или `union`, являются по умолчанию `public`.

```
struct A
 {
            int a;  // по умолчанию public
   protected:
            int b;  // защищенный
   private:
            int c;  // приватный
   public:
            int d;  // публичный
 };

struct B: public A   // B - производный класс от A
 {                   //(A - базовый)
                     // B наследует все, что было у A
   int fa() {return a;} //1
```

```
   int fb() {return b;} //2
   int fc() {return c;} //3
   int fd() {return d;} //4
 };
```

В рассмотренном примере 1 и 4 верно, т. к. переменные `a` и `d` объявлены `public`, 2 верно, т. к. переменная `b` объявлена `protected`, 3 неверно, т. к. переменная `c` объявлена `private` и доступна только внутри класса или в друзьях класса.

Отметим, что контролируется доступ к членам класса, но не их видимость.

```
int i;

class X
  {
    private:
              int i;
  };

class Y: public X
  {
    void f() { i++; } // ошибка
  };
```

Переменная `i` объявлена `private` в классе `X`, поэтому возникает ошибка. Если бы `i` была невидима в классе `Y`, то она рассматривалась бы как глобальная переменная и ошибки бы не было.

3.3. Наследование

В языке C++ одни классы (производные) могут наследовать переменные и функции других классов (базовых). Также в языке C++ разрешено множественное наследование, т. е. класс может быть порожден из любого числа базовых классов.

```
class base
 {
```

```
  public:
         int a, b;
 }

class derived: public base
 {
  public:
         int b, c;
 }

void f()
 {
  derived d;
  d.a=1;    // присваивание переменной, которая объявлена в
            // классе base и унаследована в классе  derived
  d.b=2;    // присваивание переменной, которая объявлена в
            // классе derived
  d.base::b=3; //присваивание переменной, которая объявлена
            //в классе base и унаследована в классе derived
  d.c=4;    // присваивание переменной, которая объявлена в
            // классе derived
  base *bp=&d;
 }
```

Неявное преобразование от `derived*` к `base*` разрешено, если при описании наследования использовано ключевое слово `public (class derived: public base)`. Если бы использовалось `private`, то функция `f` должна была быть другом (функцией, которая не является членом этого класса, но которой разрешается использовать его приватные и защищенные члены) или членом базового класса. Из приведенного примера видно, что если члены базового класса не переопределены в производном классе, то они обозначаются и трактуются так же, как и члены производного класса.

3.4. Виртуальные функции

Если класс `base` содержит виртуальную `virtual` функцию `vf`, а производный от него класс `derived` также содержит функцию `vf` то-

го же типа, то обращение к `vf` для объекта класса `derived` вызывает `derived::vf` (даже при доступе через указатель или ссылку на `base`). В таком случае говорят, что функция производного класса подменяет (override) функцию базового класса. Если, однако, типы этих функций различны, то функции считаются различными и механизм виртуальности не включается. Ошибкой является различие виртуальной функции базового и функции производного класса лишь в типе возвращаемого значения.

```
class base
{
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
};

class derived: public base
{
   public:
     void vf1();
     void vf2(int);// скрывает  base::vf2()
     char vf3();   // ошибка, т. к. отличается только типом
                   // возвращаемого значения
     void f();
};

void g()
{
    derived d;
    base *bp=&d; // стандартное преобразование
                 // derived* к base*
    bp->vf1();   // вызов  derived::vf1
    bp->vf2();   // вызов  base ::vf2
    bp->f();     // вызов  base ::f
}
```

В этом примере для объекта `d` класса `derived` вызываются соответственно `derived::vf1`, `base::vf2` и `base::f`. Тем самым интерпретация вызова виртуальной функции зависит от типа объекта, для

которого она вызывается, в то время как интерпретация вызова невиртуальной функции-члена класса зависит лишь от типа указателя или ссылки, указывающей на этот объект. Например, `bp->vf1()` вызывает `derived::vf1`, потому что `bp` указывает на объект класса `derived`, в котором `derived::vf1()` подменила виртуальную функцию `base ::vf1()`. Это свойство делает производные классы и виртуальные функции важными понятиями при разработке многих C++ программ. Базовый класс определяет интерфейс, для которого производные классы обеспечивают набор реализаций. Указатель на объект класса может передаваться в контекст, где известен интерфейс, определенный одним из его базовых классов, но этот производный класс неизвестен. Механизм виртуальных функций гарантирует, что этот объект все равно обрабатывается функциями, определенными для него (а не функциями, определенными для базового класса). При этом будут вызываться только функции, имена которых определены в интерфейсе базового класса, и только при соответствии их фактических параметров требуемым формальным параметрам. Возможность обращения к нескольким функциям с одним общим интерфейсом, которая обеспечивается виртуальными функциями, называется полиморфизмом. Если точный тип объекта известен во время компиляции, механизм вызова виртуальных функций не нужен.

3.5. Перегрузка операций

В данном разделе на простом примере проиллюстрированы понятия перегрузки операций и конструктора.

Перегрузка операций – это придание нового смысла знакам операций.

В C++ существует ряд ограничений на перегрузку операций. Операции и символы препроцессора ., .*, ::, ?:, #, ## нельзя перегружать, нельзя менять старшинство, ассоциативность и число операндов, нельзя вводить новые знаки операций.

Конструктор превращает некоторую область памяти в объект. Деструктор превращает объект в область памяти. Деструктор автоматически вызывается при освобождении памяти, т. е. при выполнении операции `delete`, при завершении работы функции или в конце программы.

В языке C++ конструктор класса имеет такое же имя, что и класс. Например, у класса `X` конструктор имеет имя `X()`, а деструктор имеет имя `~X()`.

Рассмотрим задачу перегрузки операции сложения для комплексных чисел.

Первый способ

Объявляем бинарную операцию сложения как нестатический член класса с одним параметром.

```
class complex
 {
  public:
         int real, image;

   complex()
           { real=image=0;} // конструктор без параметров

   complex(int r, int i)
           { real=r; image=i; } // конструктор с параметрами

   complex operator+(complex c)
      { return complex(c.real+real, c.image+image); }
 };

int main()
 {
    complex c1(1,0), c2(0,1), c3;
    c3=c1+c2;             // эквивалентно c3=c1.operator+(c2)
                          // или c3=c2.operator+(c1)
    printf("re=%d im=%d", c3.real, c3.image);
 }
```

Два конструктора можно объединить в один:

```
  complex(int r=0, int i=0){ real=r; image=i; }
```

Можно оставить в классе прототип функции `operator+`, а реализацию этой функции вынести за пределы класса.

Второй способ

Реализуем перегрузку операции сложения с помощью функции с двумя параметрами, которая не является членом класса.

```
class complex
 {
   public:
         int real, image;

   complex(int r=0, int i=0) { real=r; image=i; }
 };

complex operator+(complex c1, complex c2)
 {
    return complex(c1.real+c2.real, c1.image + c2.image);
 }

int main()
 {
   complex c1(1,0), c2(1), c3;
   c3=c2+c1;           //эквивалентно c3=operator+(c1,c2);
   printf("re=%d im=%d", c3.real, c3.image);
 }
```

3.6. Шаблоны

Шаблон класса (`template`) определяет данные и операции потенциально неограниченного множества типов. Например, единый шаблон класса `list` может обеспечить общее описание для списков разных типов `int`, `float`, `char` и т. д. В качестве примера использования шаблонов рассмотрим реализацию последовательного стека с помощью параметризованного класса. Далее в ряде программ эта реализация будет использоваться.

```
template<class telem> class stack
{
    telem *x;  // массив для хранения элементов стека
    int top;   // указатель на вершину стека
    int n;     // размер выделенного для стека массива
```

```
   public:
     stack(int size);       // конструктор
    ~stack() {delete x;} // деструктор
     void push(telem y);
     telem pop();
     int empty()
         {
            if(top==0) return 1;
            else return 0;
         }
};

// Конструктор
template<class telem> stack<telem>::stack(int size)
{
   x = new telem [size];
   if(!x)
     {
       cout << "Невозможно создать стек \n"; exit (1);
     }
   n = size;           // количество элементов в стеке
   top = 0;            // пустой стек
}

// Функция включения элемента в стек
template<class telem>void stack<telem>::push(telem y)
{
   if(top==n)
     {
       cout << "Стек заполнен \n "; return;
     }
   x[top]=y; // включение элемента
   top++;
}

// Функция исключения элемента из стека
template<class telem>telem stack<telem>::pop()
 {
```

```
   if(top==0)
     {
        cout << "Стек пуст\n";
        return 0;
     }
  top--;
  return x[top];
 }

// главная программа
void main ()
{
 stack <int> x(10);   //создаем стек с элементами типа int
 stack <double> y(10);//создаем стек с элементами типа double
 stack <char> z(10);  //создаем стек с элементами типа char

 x.push(1);
 y.push(1.1);
 z.push('z');

 count << x.pop()<< endl;
 count << y.pop()<< endl;
 count << z.pop()<< endl;
 }
```

Глава 4. Нелинейные структуры данных

4.1. Бинарные деревья

Бинарным деревом называется конечное множество узлов, которое или пусто, или состоит из корня и из двух непересекающихся бинарных деревьев, называющихся левым и правым поддеревьями данного дерева.

На рисунке 4.1 представлено бинарное дерево, которое будет использоваться в ряде примеров.

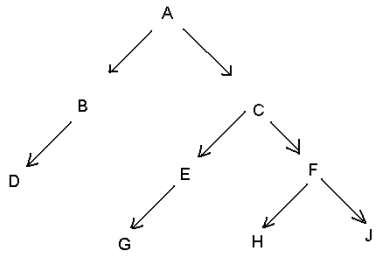

Рис. 4.1

Структура узла дерева представлена в следующем виде:

```
struct node{char info; node *llink; node *rlink;}.
```

Во многих приложениях требуется обработать узлы дерева в определенном порядке. Опишем некоторые возможные алгоритмы обхода деревьев.

Обход в прямом порядке

- Попасть в корень.
- Пройти левое поддерево в прямом порядке.
- Пройти правое поддерево в прямом порядке.

Обход в обратном порядке

- Пройти левое поддерево в обратном порядке.
- Попасть в корень.
- Пройти правое поддерево в обратном порядке.

Обход в концевом порядке

- Пройти левое поддерево в концевом порядке.
- Пройти правое поддерево в концевом порядке.
- Попасть в корень.

Обходы бинарного дерева, приведенного на рисунке 4.1, будут следующими:

1) обход в прямом порядке: ABDCEGFHJ;
2) обход в обратном порядке: DBAGECHFJ;
3) обход в концевом порядке: DBGEHJFCA.

4.2. Реализация алгоритмов работы с бинарными деревьями

Рекурсивная реализация обхода бинарного дерева в обратном порядке

```
void btree1(node *t) // t - указатель на корень дерева
{
  if(t! = NULL)
   {
      btree1 (t->llink); // обход левого поддерева
      cout << t->info;   // вывод информации
```

```
      btree1 (t->rlink); // обход правого поддерева
   }
}
```

Рекурсивная реализация ввода бинарного дерева в прямом порядке

Дерево вводится в виде *ab..c..*, где вместо NULL-связей вводим точки.

```
void input(node *&t)
{
   char c;
   cin>>c;
   if(c!='.')
     {
       t = new node;
       t->info = c;
       input(t->llink);
       input(t->rlink);
     }
   else t=NULL;
}
```

Нерекурсивный алгоритм обхода бинарного дерева в обратном порядке

Пусть есть рабочий указатель `node *p` и стек указателей на узлы дерева, `t` – указатель на корень дерева.

Алгоритм:

1) `p = t`;
2) пройти до конца самой левой ветки дерева, начинающейся в `p`, запоминая ссылки на узлы в стеке;
3) если стек непуст, то вытолкнуть из него верхний указатель в `p`, вывести информацию, которая хранится в `p`, `p = p -> rlink`, и перейти на шаг 2.

Далее приведена реализация некоторых алгоритмов работы с бинарными деревьями с использованием шаблонов.

```
template<class DataT> class tree
 {
      DataT info;
      tree *llink;
      tree *rlink;
   public:
      tree *root;                  // корень дерева
      tree(){root=NULL;};          // конструктор
      void in(tree<DataT> *&t);    // ввод дерева
      void btree1(tree<DataT> *t);
      void btree2(tree<DataT> *t);
      void btree3(tree<DataT>*t);
 };

// Ввод дерева в прямом порядке
template<class DataT>void tree<DataT>::in(tree<DataT> *&t)
 {
   DataT c;
   cin>>c;
   if(c!='.')
     {
       t=new tree<DataT>;
       t->info=c;
       in(t->llink);
       in(t->rlink);
     )
   else  t=NULL;
 }

// Рекурсивная реализация обхода в обратном порядке
template<class DataT>void tree<DataT>::btree1(tree<DataT> *t)
 {
   if(t!=NULL)
      {
        btree1(t->llink);
        cout<<t->info;
        btree1(t->rlink);
      }
 }
```

```
// Нерекурсивная реализация обхода в обратном порядке
template <class DataT>void tree<DataT>::btree2(tree<DataT> *t)
 {
   stack <tree<DataT>*> x(100);
   tree<DataT> *p;
   p=t;
   m: while(p!=NULL)
        {
           x.push(p);
           p=p->llink;
        }

      if(!x.empty())       // если стек непуст
        {
           p=x.pop();      // выталкиваем p
           cout<<p->info;
           p=p->rlink;
           goto m;
        }
 }

// Нерекурсивная реализация обхода в прямом порядке
template<class DataT>void tree<DataT>::btree3(tree<DataT> *t)
 {
    stack<tree<DataT>*> s(100);
    tree<DataT> *p;
    p=t;
    s.push(p);
    while(!s.empty())
       {
         p=s.pop();
         cout<<p->info;
         cout<<"\n";
         if (p->rlink!=NULL)
                     s.push(p->rlink);
         if (p->llink!=NULL)
                     s.push(p->llink);
       }
```

```
 }

//главная программа
int main()
 {
   tree<char> t; // задаем дерево с элементами типа char
   cout<<endl;
   t.in(t.root);
   t.btree1(t.root);
   t.btree2(t.root);
   t.btree3(t.root);
 }
```

Нерекурсивная реализация обхода бинарного дерева по уровням с помощью FIFO-очереди

Теперь рассмотрим новый обход бинарного дерева, который называют обходом в ширину. Этот метод использует FIFO-очередь и осуществляет обход дерева по уровням слева направо, в отличие от стекового обхода в глубину, когда алгоритм пытается пройти как можно дальше вглубь, пока не дойдет до конца какой-либо ветки. В этой программе приведена параметризованная реализация последовательной циклической FIFO-очереди и снова описан параметризованный класс "бинарное дерево". Другой вариант программы мог бы включать в одну программу реализацию и стека, и очереди, и описание одного класса "бинарное дерево".

Этот принцип применения стека для обхода в глубину и очереди для обхода в ширину сохраняется и при обходе других структур данных, например графов. Отметим, что в функции ввода бинарного дерева для обозначения нулевых связей используется символ точка ".". Это накладывает ограничение на тип информационного узла дерева. Если необходимо работать не только с типом `char`, то надо вводить нулевые связи каким-то другим способом. Такая модификация программы оставлена читателю в качестве упражнения.

```
#include <iostream.h>

template<class telem> class queue{
    telem *x;
    int head;
```

```
    int tail;
    int n;
    int elements;
public:
    queue(int size);
   ~queue(){delete x;}
    void push(telem y);
    telem pop();
    int empty()
        {
          if(elements==0) return 1;
          else            return 0;
        }
};

template<class telem>queue<telem>::queue(int size)
{
    x=new telem[size];
    if(!x)
       {
         cout<<"Невозможно создать очередь\n";
         exit(1);
       }
    n=size;
    head=0;
    elements=0;
    tail=head;
}

template<class telem>void queue<telem>::push(telem y)
{
    if(elements==n)
        {
           cout<<"Очередь заполнена\n";
           return;
        }
    x[tail]=y;
    if((tail++)>(n-1))
                     tail=0;
```

```
    elements++;
}

template<class telem>telem queue<telem>::pop()
{
    telem nif;
    if(!elements)
         {
            cout<<"Очередь пуста\n";
            return 0;
         }
    nif=x[head];
    if((head++)>(n-1))
                      head=0;
    elements--;
    return nif;
}

template<class DataT>class tree
{
    DataT info;
    tree *llink;
    tree *rlink;
public:
    tree *root;
    tree(){ root=NULL; };
    void in(tree<DataT>*&t);
    void btree4(tree<DataT> *t);
};

template<class DataT>void btree<DataT>::vv(tree<DataT>*& t)
{
    DataT c;
    cin>>c;
    if (c!='.')
      {
        t=new (tree<DataT>);
        t->info=c;
        in(t->llink);
```

```
        in(t->rlink);
      }
    else t=NULL;
}

template<class DataT>void btree<DataT>::btree4(tree<DataT> *t)
{
    queue<tree<DataT>*> q(15);
    btree<DataT> *p;
    p=t;
    q.push(p);
    while(!q.empty()
      {
        p=q.pop();
        cout<<(p->info);
        cout<<"\n";
        if (p->llink!=NULL)
                          q.push(p->llink);
        if (p->rlink!=NULL)
                          q.push(p->rlink);
     }
}

int main()
{
    tree<char> t;
    t.in(t.root);
    t.btree4(t.root);
}
```

4.3. Представление лесов деревьев в виде бинарных деревьев

Деревом называется непустое конечное множество узлов, состоящее из выделенного узла, который называется корнем дерева, и из $T_1, T_2, ..., T_n$ деревьев, которые называются поддеревьями данного дерева.

Лесом называется конечное (возможно, пустое) множество деревьев. На рисунке 4.2 представлен лес.

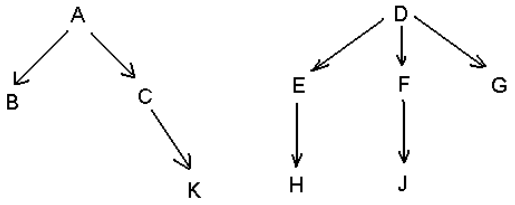

Рис. 4.2

Между лесами деревьев и бинарными деревьями можно установить взаимно однозначное соответствие (см. рис. 4.3).

Пусть дан лес $F = (T_1, T_2, ..., T_n)$. Поставим в соответствие ему бинарное дерево $B(F)$, которое определяется следующим образом:

1) если $n = 0$, то $B(F)$ пусто;
2) если $n > 0$, то корнем дерева $B(F)$ является корень дерева T_1, левым поддеревом $B(F)$ является $B(T_{11}, T_{12}, ..., T_{1m})$, где T_{11}, T_{12},..., T_{1m} – поддеревья корня первого дерева, правым поддеревом $B(F)$ является $B(T_2, ..., T_n)$.

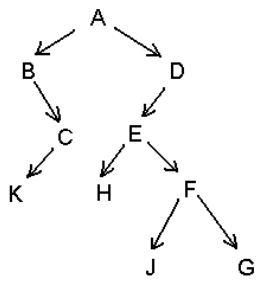

Рис. 4.3

Можно определить алгоритм обхода лесов деревьев в прямом порядке (попасть в корень первого дерева, пройти поддеревья первого

дерева в прямом порядке, пройти оставшиеся деревья в прямом порядке). Обратный, концевой, другие порядки определяются аналогично обходам бинарных деревьев.

4.4. Другие представления деревьев

Последовательное представление в обратном порядке

Записываем узлы леса в обратном порядке и задаем степени (число выходящих дуг) всех узлов.

B	K	C	A	H	E	J	F	G	D
0	0	1	2	0	1	0	1	0	3

Такое представление удобно для вычисления функции, заданной на узлах дерева.

Правая скобочная запись

Если корнем дерева T является узел a, с поддеревьями $(T_1, T_2, ..., T_n)$, то правая скобочная запись определяется следующим образом:

1) $r(a) = a$;
2) $r(T) = (r(T_1), ..., r(T_n))a$.

Пусть дано дерево T, изображенное на рисунке 4.4. Его правая скобочная запись имеет вид $r(T) = ((3, 4)2, ((7, 8, 9)6)5)1)$.

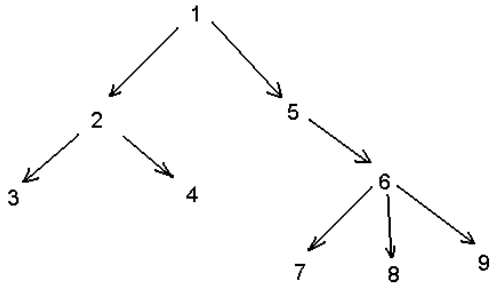

Рис. 4.4

Левая скобочная запись

Если корнем дерева T является узел a, с поддеревьями $(T_1, T_2, ..., T_n)$, то правая скобочная запись определяется следующим образом:

1) $l(a) = a$;
2) $l(T) = a(l(T_1), ..., l(T_n))$.

Левая скобочная запись дерева, изображенного на рисунке 4.4, имеет вид $l(T) = 1(2(3, 4), 5(6(7, 8, 9)))$.

Представление арифметических выражений в виде деревьев

Пусть дано арифметическое выражение $(a + b) * c$. Его представление в виде дерева изображено на рисунке 4.5. Правая и левая скобочные записи этого выражения: $r(T) = ((a, b)+, c)*), l(T) = (*(+(a, b), c))$.

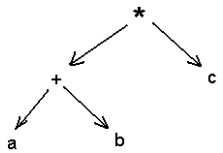

Рис. 4.5

Обратная польская запись получается из правой скобочной записи выбрасыванием всех скобок и запятых $ab + c*$.

Прямая скобочная запись получается из левой скобочной записи выбрасыванием всех скобок и запятых $* + abc$.

Обратная польская запись используется в языке Fort, а прямая польская запись – в языке Lisp.

Далее приведен пример реализации простейшего стекового калькулятора, вычисляющего значение выражения в обратной польской записи. В программе будем использовать параметризованную реализацию стека, приведенную в главе 2.

```
int main()
 {
    calculator();
     return 0;
 }
```

```
void calculator()
 {
     stack<double> calc (100);
     double a, b;
     char str[1];
     cout <<"Простейший калькулятор\n";
     cout <<"Для выхода введите "q"\n";
    do
      {
        cout<<":";
        cin>>str;
    switch(*str)
        {
           case '+':
                 a=calc.pop();
                 b=calc.pop();
                 cout << a+b << endl;
                 calc.push (a+b);
                 break;

           case '/':
                 a=calc.pop();
                 b=calc.pop();
                 if(!a)
                    {
                       cout <<"Деление на 0\n";
                       break;
                    }
                 cout<<b/a<<endl;
                 calc.push(b/a);
                 break;

            case '.': //содержимое вершины стека
                 a=calc.pop();
                 calc.push(a);
                 cout<<"Текущее значение в вершине стека:";
                 cout<<a<<endl;
                 break;
```

```
            default:
                  calc.push(atof(str));
        }
    } while (*str!='q');
}
```

Глава 5. Сортировка

5.1. Постановка задачи

Пусть дано конечное множество записей $R_1, R_2, ..., R_n$ и линейно упорядоченное множество ключей $K_1, K_2, ..., K_n$, которое взаимно однозначно отображается на множество записей. Задача сортировки – найти такую перестановку записей $R_{i_1}, R_{i_2}, ..., R_{i_n}$, чтобы выполнялось неравенство $K_{i_1} \leq K_{i_2} \leq ... \leq K_{i_n}$.

В качестве примера можно привести задачу сортировки телефонного справочника. Тогда $R_1, R_2, ..., R_n$ – это номера телефонов, адреса и фамилии абонентов, а $K_1, K_2, ..., K_n$ – это лексикографически упорядоченное множество фамилий абонентов.

Сортировка называется *устойчивой*, если записи с равными ключами остаются на месте.

Если записи и/или ключи занимают память большого размера, то лучше составить новую таблицу адресов и работать только с ней, не перемещая записи (на рис. 5.1 сегменты). Такой метод называется *сортировкой таблицы адресов*.

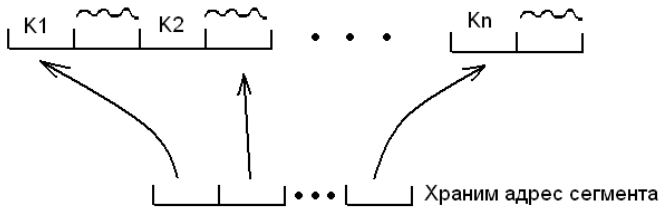

Рис. 5.1

Если в каждой записи организовать вспомогательное поле связи, то можно работать только со связями, не перемещая записи. Такой метод называется *сортировкой списка*.

В дальнейшем, будем считать, что мы сортируем массив целых чисел $K_1, K_2, ..., K_n$.

5.2. Сортировка вставками

Сортировка простыми вставками

Пусть ключи $K_1, K_2, ..., K_{j-1}$ уже упорядочены, т. е. $K_1 \leq K_2 \leq ... \leq K_{j-1}$. Ключ K_j последовательно сравниваем с ключами K_{j-1}, K_{j-2}, ... , K_1, сдвигая при необходимости ключи на одну позицию вверх до тех пор, пока не найдем место ключа K_j в упорядоченном массиве.

Пример: 7, 5, 1, 10, 4
5, 7, 1, 10, 4
5, 1, 7, 10, 4
1, 5, 7, 10, 4
1, 5, 7, 4, 10
1, 5, 4, 7, 10
1, 4, 5, 7, 10

Для поиска места для ключа K_j необходимо произвести в среднем $\frac{j-1}{2}$ сравнений. Тогда среднее число сравнений вычисляется по формуле $T = 1 + 2 + ... + \frac{j-1}{2} + ... + \frac{n-1}{2}$. Среднее время работы алгоритма равно $T = C \cdot n^2$.

Метод Шелла

Пусть даны ключи: $K_1, K_2, ..., K_8$ и задана последовательность целых чисел (шагов) $h = (4, 2, 1)$.

1) сортируем ключи, отстоящие друг от друга на четыре позиции, т. е. пары (K_1, K_5), (K_2, K_6), (K_3, K_7), (K_4, K_8) методом простых вставок;

2) сортируем ключи, отстающие друг от друга на 2 позиции, т. е. четверки (K_1, K_3, K_5, K_7), (K_2, K_4, K_6, K_8) методом простых вставок;

3) сортируем весь массив методом простых вставок.

На каждом шаге сортируются частично отсортированные массивы. Среднее число сравнений в этом методе $T = 1.66n^{1.25}$. Кнут рекомендует выбирать последовательность шагов по формуле $h_{s+1} = 3h_s + 1$, где $h_1 = 1$. Остановиться следует на значении h_t, когда $h_{t+2} \geq n$.

Сортировка выбором

В этом методе сначала находим наибольший элемент массива и меняем местами этот элемент с последним элементом массива. Затем делаем то же самое с предпоследним элементом и т. д. Среднее время работы этого метода, как и у всех простейших методов сортировки, пропорционально n^2. Усовершенствованный метод, основанный на использовании бинарных деревьев, называется пирамидальной сортировкой.

5.3. Обменная сортировка

Сортировка методом пузырька

Сравниваем K_1 и K_2 и, если надо, меняем местами. Затем сравниваем K_2 и K_3 и, если надо, меняем местами. Таким образом доходим до K_{n-1} и K_n и, если надо, меняем местами. Затем повторяем этот алгоритм для $n - 1$ элементов (т. к. самый большой уже поднят наверх) и т. д.

Быстрая сортировка Хоара

Дан массив K_1, ..., K_n. Сначала $i = 1, j = n$. Сравниваем K_i и K_j. Если обмен не нужен, то уменьшаем j на 1 и продолжаем. После первого обмена продолжаем сравнение, увеличивая i на 1. После очередного обмена уменьшаем j и т. д., до тех пор пока $i \neq j$. К этому моменту ключ K_1 займет свою позицию, т. е. все ключи слева меньше K_1, а справа – больше (рис. 5.2).

Заносим границы правого массива в стек и сортируем левый массив тем же способом. Возможна также и рекурсивная реализация.

Метод требует память размера $M = C_1(n+2 \cdot log_2 n)$ и среднее время $T = C_2 \cdot n \cdot ln(n)$. В среднем метод очень хороший, но в худшем случае (когда массив почти упорядочен) оценка становится n^2. Чтобы избежать этого случая, ключ разделения выбирают не K_1, а случайным

Рис. 5.2

образом. Иногда задают некоторую константу C так, чтобы массивы длины меньше C сортировались простыми вставками.

5.4. Реализация быстрой сортировки

Рекурсивная реализация быстрой сортировки

Рассмотрим сначала параметризованную рекурсивную реализацию быстрой сортировки. Тип элемента сортируемого массива `stype` передается в качестве параметра в функцию сортировки `quick`, которая производит рекурсивный вызов функции `qs(item,0,count-1)`, сортирующей массив `item`, начиная с элемента с номером 0 до элемента с номером `count-1`. В функции `main` сначала с помощью функции сортировки `quick` сортируется массив символов и массив целых чисел, а затем сортируется случайный массив целых чисел с помощью реализованной параметризованной функции `quick` и с помощью стандартной функции `qsort`. Функция `comp` задает отношение порядка на множестве сортируемых элементов массива. Сравнение времен сортировки одного и того же массива чисел показывает, что параметризованная функция выполняется намного быстрее, т. к. при вызове стандартной функции `qsort` ей передается адрес функции `comp`, которая вызывается большое количество раз. При параметризованной реализации компилятор встраивает (`inline`) сравнение, получая выигрыш в скорости.

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

```
#include <time.h>

template<class stype>void quick(stype *item,int count);
template<class stype>void qs(stype *item,int left,int right);
int comp(const void *a,const void *t);

int main()
{
 char str []="zxscz";
 quick (str, (int) strlen (str));
 cout <<"Отсортированная строка:" <<str <<endl;
 int nums[]={1,6,3,10,4,6};
 int i;
 quick(nums,6);
 cout <<"Отсортированный массив:";

 for(i=0;i<6;i++)
          cout<<nums[i]<<" ";

 int nums1[100000],nums2[100000];
 time_t start, end;

 for(i=0;i<100000;i++)
          nums1[i]=nums2[i]=rand();

 start=clock();
 quick(nums1, 100000);
 end=clock();
 cout <<endl<<"Quick sort:"<<end-start;
 start=clock();
 qsort(nums2,unsigned(100000),sizeof(int),comp);
 end=clock();
 cout <<endl<<"Qsort time:" <<end-start;
 system("PAUSE");
 return 0;
}

template<class stype>void quick(stype *item,int count)
{
```

```
  qs (item, 0, count-1);
}

template<class stype>void qs(stype *item,int left,int right)
{
  register int i, j;
  stype x, y;
  i=left;
  j=right;
  x=item[(left+right)/2];

 do
  {
       while (item[i]<x && i<right) i++;
       while (x<item[j] && j>left) j--;

    if(i<=j)
         {
           y=item [i];
           item [i]=item[j];
           item [j]=y;
           i++; j--;
         }

  }  while (i<=j);

   if(left<j)  qs(item, left, j);
   if(i<right) qs(item, i, right);
}

int comp (const void *a, const void *b)
{
  return *(int*) a - *(int*) b;
}
```

Нерекурсивная реализация быстрой сортировки

Теперь приведем пример нерекурсивной реализации быстрой сортировки с использованием ранее реализованного параметризованного

стека. Здесь в функции `main` решаются те же задачи, что и в рекурсивном варианте, но без использования стандартной функции `qsort`.

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

template<class item>void quicksort(item*a,intl,int r);
template<class item>int partition(item *a,int l,int r);
inline void push2(stack<int> &s,int A, int B);

int main()
{
 char str []="bead";
 quicksort (str,0,(int)strlen(str)-1);
 cout <<"Отсортированная строка:" <<str <<endl;
 int nums[]={1,6,3,10,4,6};
 int i;
 quicksort(nums,0,5);
 cout <<"Отсортированный массив:";

 for(i=0;i<6;i++)
         cout<<nums[i]<<" ";

 int nums1 [100];
 time_t start, end;

 for(i=0;i<100;i++)
         nums1[i]=rand();

 start=clock();
 quicksort(nums1,0,99);
 end=clock();
 cout<<endl<<"Quick sort:"<<end-start;
 system("PAUSE");
 return 0;
}
```

```
inline void push2(stack<int> &s, int A, int B)
{
   s.push(B);
   s.push(A);
}

template<class item>void quicksort(item *a,int l,int r)
{
  stack<int> s(50);
  push2(s,l,r);

  while(!s.empty())
   {
     l=s.pop();
     r=s.pop();

     if(r<=l)
            continue;

     int i=partition(a,l,r);

     if(i-1>r-i)
     {
        push2(s,l,i-1);
        push2(s,i+1,r);
     }

     else
     {
        push2 (s, i+1, r);
        push2 (s, l, i-1);
     }
   }
 }

template<class item>int partition(item*a,int l,int r)
{
   int i=l-1, j=r;
   item v=a[r],c;
```

```
   for (;;)
   {
       while(a[++i]<v);

       while(v<a[--j])
            if(j==l) break;

       if (i>=j) break;
       c=a[i];
       a[i]=a[j];
       a[j]=c;
   }

   c=a[i];
   a[i]=a[r];
   a[r]=c;
   return i;
}
```

Сортировка типов, определенных пользователем

Здесь рассмотрен пример использования стековой быстрой сортировки для сортировки массива, элементы которого являются объектами типа `address`, определенного пользователем. Для этого перегружены операция отношения порядка для сортируемого типа данных и еще ряд необходимых операций.

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

template<class item>void quicksort(item *a,int l,int r);
template<class item>int partition(item *a,int l,int r);

template<class Stype>class stack
{
   Stype *stck;
```

```
   int top;
   int length;

 public:
        stack(int size);
       ~stack() {delete[] stck;}
        void push(Stype i);
        Stype pop();
        int empty()
          {
              if(top==0)  return 1;
              else        return 0;
          }
};

template<class Stype>stack<Stype>::stack(int size)
{
  stck=new Stype[size];

  if(!stck)
   {
        cout<< "Невозможно создать стек\n";
        exit(1);
   }

  length=size; top=0;
}

template<class Stype>void stack<Stype>::push(Stype i)
{
  if(top==length)
     {
          cout  << "Стек заполнен\n";
          return ;
     }

  stck[top]=i; top++;
}

template<class Stype>Stype stack<Stype>::pop(
```

```
{
  if(top==0)
   {
     cout << "Стек пуст\n";
     return 0;
   }

  top--;
  return stck[top];
}

inline void push2(stack <int> &s, int A, int B);

class address
{
      char name[40];
      char street[40];
      char city[10];
      char zip[11];

 public:
       address(char *n,char *s,char *c, char *z);
       address(){};
       int operator<(address &ob)
        {
           return strcmp (zip, ob.zip)<0;
        }
 friend ostream &operator<<(ostream &stream, address &ob);
};

address::address(char *n,char *s,char *c, char *z)
{
    strncpy(name, n);
    strncpy(street, s);
    strncpy(city, c);
    strncpy(zip, z);
}

ostream &operator<<(ostream &stream, address &ob)
```

```
{
    cout << ob.name <<endl;
    cout << ob.street <<endl;
    cout << ob.city <<endl;
    cout << ob.zip <<endl;
    return stream;
}

address addrs [4]={
address("A.C. Pushkin",   "ул.Пушкина 12", "Москва","d"),
address("A.C. Griboedov", "ул.Грибоедова 13", "Москва","c"),
address("A.V. Sokolov ",  "ул.Мира 14", "Петрозаводск","b"),
address("E.A. Aksenova",  "ул.Дружбы 15", "Петрозаводск","a"),
                 };

int main ()
{
    int i;
    cout<<"Unsorted Addresses:\n";

    for(i=0;i<4;i++)
               cout<<addrs[i];

    quicksort(addrs,0,3);
    cout<<"Addresses sorted by ZIP code:\n";

    for(i=0;i<4;i++)
               cout<<addrs [i];

    system("PAUSE");
    return 0;
}

inline void push2(stack<int> &s, int A, int B)
{
     s.push(B);
     s.push(A);
}

template <class item> void quicksort(item *a,int l,int r)
```

```
{
   stack <int> s(50);
   push2(s,l,r);

   while(!s.empty())
    {
        l=s.pop();
        r=s.pop();

      if (r<=l)  continue;

      int i=partition(a,l,r);

      if(i-1>r-i)
        {
            push2(s,l,i-1);
            push2(s,i+1,r);
        }

      else
        {
            push2(s,i+1,r);
            push2(s,l,i-1);
        }
    }
}

template <class item> int partition(item *a,int l,int r)
{
    int i=l-1, j=r;
    item v=a[r], c;

    for(;;)
     {
         while(a[++i]<v);
         while(v<a[--j])
              if(j==l) break;

         if(i>=j) break;
```

```
            c=a[i];
            a[i]=a[j];
            a[j]=c;
        }

        c=a[i];
        a[i]=a[r];
        a[r]=c;
        return i;
}
```

5.5. Сортировка слиянием фон Неймана

Этот метод основан на последовательном слиянии упорядоченных массивов в большие упорядоченные массивы.

Пусть надо упорядочить массив $(7, 1, 3, 2, 5, 6, 8, 4)$. Сливая одиночные элементы в пары и записывая пары в новый массив, получим массив упорядоченных пар: $(1, 7)$, $(2, 3)$, $(5, 6)$, $(4, 8)$. Теперь сливаем упорядоченные пары в упорядоченные четверки и, записывая их на место первоначального массива, получаем $(1, 2, 3, 7)$ и $(4, 5, 6, 8)$. И, наконец, сливаем четверки в упорядоченную восьмерку чисел и получаем $(1, 2, 3, 4, 5, 6, 7, 8)$.

Алгоритм фон Неймана имеет среднее время работы $T = C \cdot n \cdot log_2 n$ и требует двойную память.

В процессе работы алгоритму приходится решать следующую задачу: имея два упорядоченных массива $x_1 \le x_2 \le ... \le x_m$ и $y_1 \le y_2 \le ... \le y_n$, надо сделать из них один упорядоченный массив $z_1 \le z_2 \le ... \le z_{m+n}$. Пусть i, j, k – счетчики массивов x, y, z соответственно.

Алгоритм двухпутевого слияния в пошаговой форме:

D1: $i = 1; j = 1; k = 1;$
D2: если $x_i \le y_j$, то перейти к шагу D5;
D3: $z_k = y_j$; $k = k + 1$; $j = j + 1$;
    если $j \le n$, то перейти к шагу D2;
D4: $z_k ... z_{m+n} = x_i ... x_m$;
D5: $z_k = x_i$; $k = k + 1$; $i = i + 1$;
    если $i \le m$, то перейти к шагу D2;
D6: $z_k ... z_{m+n} = y_j ... y_n$.

В этой главе кратко рассмотрены основные методы внутренней сортировки, т. е. сортировки в оперативной памяти. Сортировка файлов, размер которых превышает объем оперативной памяти, называется внешней сортировкой. Она основана на сортировке подфайлов, которые помещаются в оперативную память одним из методов внутренней сортировки, и последующем слиянии их методом k-путевого слияния. Методы внутренней сортировки можно применять и в системах с виртуальной памятью, но надо выбирать методы, которые имеют минимальный размер рабочего множества, т. к. в противном случае можно получить очень большие затраты на алгоритмы замещения страниц.

Глава 6. Поиск

6.1. Последовательный поиск

Задача: пусть задана таблица записей с ключами $K_1, K_2, ..., K_n$. Нужно найти в таблице запись с ключом K (или убедиться в его отсутствии).

Алгоритм последовательного поиска:

R1: $i = 1$;
R2: если $K_i = K$, то УДАЧА;
R3: $i = i + 1$;
R4: если $i \leq n$, то перейти к шагу R2;
иначе НЕУДАЧА.

Оказывается, что данный алгоритм можно ускорить при помощи вставки ключа поиска в конец массива ключей.

Быстрый последовательный поиск:

B1: $i = 1, K_{n+1} = K$;
B2: если $K_i = K$, то перейти к шагу B4;
B3: $i = i + 1$; перейти к шагу B2;
B4: если $i \leq n$, то УДАЧА;
иначе НЕУДАЧА.

В этом алгоритме быстрее выполняется самый внутренний цикл за счет того, что не надо выполнять проверку $i \leq n$.

Пусть теперь известны вероятности: $p_1, p_2, \ldots, p_n$, где $\sum p_i = 1$. Возникает задача: как расположить ключи, чтобы среднее время поиска было минимально. Среднее время вычисляется по формуле $T = C(p_1 \cdot 1 + p_2 \cdot 2 + p_3 \cdot 3 + ... + p_i \cdot i + ... + p_n \cdot n)$. Очевидно, что эта функция достигает минимума, если $p_1 \geq p_2 \geq ... \geq p_n$.

Если вероятности неизвестны, то можно завести счетчики числа обращений и на их основании переупорядочивать записи. Того же эффекта можно достичь, перемещая записи, которые стали нужны, в начало файла. Это удобно делать, если записи связаны в линейный список. Такой метод называется самоорганизующимся файлом.

6.2. Бинарный поиск

Этот метод используется при поиске в упорядоченной таблице. Пусть $K_1 \le K_2 \le ... \le K_n$. Нужно найти в таблице ключ K.

Алгоритм бинарного поиска:

B1: $l = 1; u = n$;
B2: если $u < l$, то НЕУДАЧА;
иначе $i = \lfloor \frac{u+l}{2} \rfloor$;
B3: если $K > K_i$, то перейти к шагу B4;
если $K = K_i$, то УДАЧА;
$u = i - 1$; перейти к шагу B2;
B4: $l = i + 1$; перейти к шагу B2;

Среднее время вычисляется по формуле $T = C \cdot log_2 n$.

6.3. Поиск по бинарному дереву

Бинарным деревом поиска называется такое бинарное дерево ключей, в котором все ключи левого поддерева каждого узла меньше ключа данного узла, а правого – больше (рис. 6.1).

Пусть узел бинарного дерева поиска состоит из ключа, информационной части (массива символов) и указателей на левое и правое поддеревья:

```
struct node{int key; char info[20]; node *llink; node *rlink;}.
```

Следующая функция осуществляет поиск ключа `K` в дереве, на корень которого указывает указатель `t`. Если ключа в данном дереве нет, то он вставляется в дерево поиска.

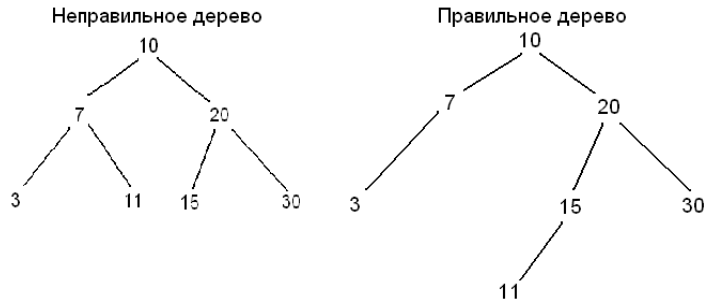

Рис. 6.1

```
void bstree(node *t, int K)
{
   node *p, *q;
   p=t;
   M1:
        if(K>p->key) goto M2;

        if(K==p->key)
          {
                cout<<p->info;
                 return;
          }

        if(p->llink!=NULL)
          {
               p=p->llink;
               goto M1;
          }
        else goto M3;
  M2:
       if(p->rlink!=NULL)
         {
               p=p->rlink;
               goto M1;
         }
  M3:
```

```
        q=new node;
        q->key=K;
        cin>>q->info;
        q->llink=NULL;
        q->rlink=NULL;

        if(K>p->key)
                p->rlink=q;
        else
                p->llink=q;
}
```

Среднее время поиска вычисляется по формуле $T = 2 \cdot ln(n)$.

6.4. Оптимальные и сбалансированные деревья

Пусть есть три ключа K_1, K_2, K_3, где $K_1 \leq K_2 \leq K_3$, и известны вероятности обращений к этим ключам p_1, p_2, p_3, где $p_1 + p_2 + p_3 = 1$. Возможны 5 разных деревьев поиска (рис. 6.2).

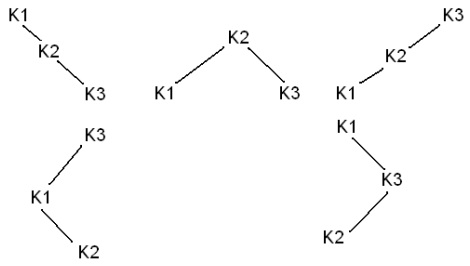

Рис. 6.2

Напишем формулу, которая вычисляет среднее время поиска для каждого дерева:

I: $1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3$,
II: $2 \cdot p_1 + 1 \cdot p_2 + 2 \cdot p_3$,
... и т. д. ...

Вычисляя средние времена поиска для всех возможных деревьев, мы находим дерево, которое минимизирует время поиска, т. е. опти-

мальное дерево. Построение оптимальных деревьев сложно и требует знания вероятностей обращения к ключам. Поэтому на практике обычно ограничиваются построением в некотором смысле "хороших" деревьев. Один из таких методов был предложен в 1962 году советскими математиками Е. М. Ландисом и Г. М. Адельсоном-Вельским. Они предложили AVL-деревья, или иначе – сбалансированные деревья. В таком методе требуется не более $C \cdot log_2 n$ операций для поиска и вставки. Но платить за это надо расходами памяти по 2 бита в каждом узле для хранения разности высот левого и правого поддеревьев.

Высотой бинарного дерева называется число вершин на самом длинном пути от корня.

Бинарное дерево называется *сбалансированным*, если высота левого поддерева каждого узла отличается от высоты правого поддерева не более чем на 1 (рис. 6.3). Существуют и другие определения сбалансированности.

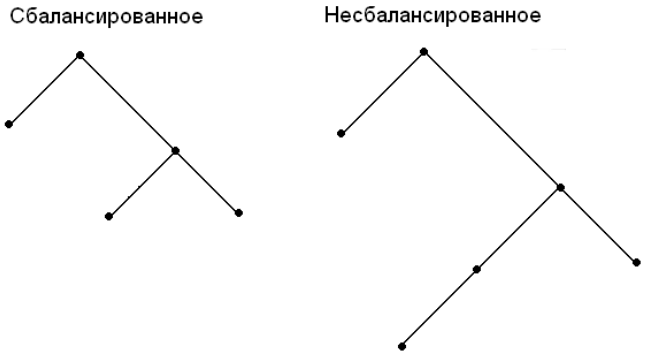

Рис. 6.3

6.5. Хеширование

В этом методе поиска считается, что на множестве ключей задана некоторая функция $h(K)$, которая называется хеш-функцией, так что по ключу K можно сразу вычислить его порядковый номер в таблице поиска. Например, если ключи – это числа $0, 1, 2, ..., n$, то существует простая хеш-функция $h(K) = K$ (рис. 6.4). При хеш-поиске может

возникнуть коллизия, т. е. ситуация, когда $K_1 \neq K_2$, а $h(K_1) = h(K_2)$. Существует несколько алгоритмов разрешения коллизий с использованием хеш-функции.

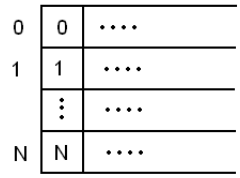

Рис. 6.4

Хеш-функцию надо выбирать таким образом:

1) хеш-функция должна быстро вычисляться;
2) хеш-функция должна минимизировать число коллизий.

Пусть элементы таблицы пронумерованы от 0 до $n-1$. Можно выбрать в качестве хеш-функции $h(K) = K\%n$, где n – размер таблицы.

Поиск со вставкой по рассеянной таблице с цепочками

В этом методе для разрешения коллизий используются связные списки, в которые помещаются элементы таблицы с равными значениями хеш-функции. Пусть элементы таблицы поиска имеют следующий вид:

```
struct node{int p; int key; int link;},
```

где `p` – признак занятости узла (`p = 0` – узел свободен, `p = 1` – узел занят), `key` – поле ключ, `link` – указатель на следующий элемент связного списка (этот элемент в данной реализации имеет целый тип, т. к. в качестве адресов мы используем индексы элементов массива `node table[n+1]`).

Узел `table[0]` в таблице всегда свободен. Для облегчения поиска свободных мест в таблице будет использоваться переменная R. В начале работы $R = n + 1$. В процессе работы `table[i]` заняты для всех $R \leq i \leq n$.

Алгоритм:

C1: $i = h(K) + 1$;
C2: если `table[i].p`=0, то перейти к шагу C6;
C3: если `table[i].key`= K, то УДАЧА;
C4: если `table[i].link`$\neq 0$,
то i =`table[i].link` и перейти к шагу C3;
C5: уменьшаем R до тех пор, пока `table[R].p`$\neq 0$;
если $R = 0$, то ПЕРЕПОЛНЕНИЕ;
иначе `table[i].link`= $R, i = R$;
C6: `table[i].p`=1; `table[i].key`= K; `table[i].link`=0.

Обозначим $C(n)$ – среднее время поиска в случае, если алгоритм заканчивается удачей, а $C'(n)$ – среднее время поиска, если алгоритм заканчивается неудачей. Тогда для приведенного алгоритма $C(n) = = 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{1}{4}\alpha$, $C'(n) = 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$, где $\alpha = \frac{M}{n}$ – коэффициент заполнения таблицы размера n, содержащей M ключей.

Алгоритм поиска по рассеянной таблице с цепочками можно модифицировать – хранить несколько связных списков, по одному на каждый хеш-адрес. При такой модификации необходимо хранить начальные адреса списков. После хеширования выполняется последовательный поиск в соответствующем списке. Можно ускорить поиск, если для каждого хеш-адреса хранить не связный список, а дерево поиска.

Поиск со вставкой по открытой рассеянной таблице

Рассмотрим другой алгоритм организации хеш-таблицы, в котором коллизии разрешаются простым перебором соседних мест в таблице. Пусть элементы таблицы поиска имеют следующий вид:

```
struct node{int p; int key;},
```

где `p` – признак занятости узла (`p=0` – узел свободен, `p=1` – узел занят), `key` – ключ. Таблица элементов хранится в массиве `node table[n]`. Пусть M – число занятых элементов в таблице. В начале работы $M = 0$. Таблица будет считаться переполненной, если $M = n - 1$. Пожертвуем одной ячейкой для того, чтобы внутренний цикл был быстрее. В противном случае необходимо будет заводить счетчик для числа повторений шага L2.

Алгоритм:

L1: $i = h(K)$;
L2: если `table[i].p`$= 0$, то перейти к шагу L4;
если `table[i].key`$= K$, то УДАЧА;
L3: $i = i - 1$;
если $i < 0$, то $i = i + M$ и перейти к шагу L2;
L4: если $M = n - 1$, то ПЕРЕПОЛНЕНИЕ;
$M = M + 1$; `table[i].p`$= 1$; `table[i].key`$= K$.

Оценки среднего времени работы этого алгоритма имеют вид:
$C(n) = \frac{1}{2}(1 + \frac{1}{1-\alpha})$, $C'(n) = \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$.

Существует много других способов организации поиска при помощи хеш-функций. Например, алгоритм поиска со вставкой по открытой рассеянной таблице можно модифицировать введением второй $h_2(K)$ хеш-функции, которая задает порядок перебора ключей. Шаг L3 будет теперь иметь вид $i = i - h_2(K)$. Такой метод называется "открытая адресация с двойным хешированием".

Список использованной литературы

1. *Кнут Д.* Искусство программирования для ЭВМ. Т. 1. М.: Мир, 1976.
2. *Кнут Д.* Искусство программирования для ЭВМ. Т. 3. М.: Вильямс, 2001.
3. *Шилдт Г.* Теория и практика C++. СПб.: BHV – Санкт-Петербург, 1996.
4. *Эллис М., Строуструп Б.* Справочное руководство по языку программирования C++ с комментариями. М.: Мир, 1992.
5. *Седжвик Р.* Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. Киев: Изд-во "ДиаСофт", 2001.
6. *Вирт Н.* Алгоритмы + структуры данных = программы. М.: Мир, 1977.
7. *Соколов А. В.* Математические модели и алгоритмы оптимального управления динамическими структурами данных. Петрозаводск: Изд-во ПетрГУ, 2002.
8. *Кормен Е., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.
9. *Шилдт Г.* Полный справочник по C. 4-е изд. М.: Вильямс, 2002.

Учебное издание

Аксёнова Елена Алексеевна,
Соколов Андрей Владимирович

Алгоритмы и структуры данных на C++

Учебное пособие

Редактор Т. А. Каракан
Компьютерная верстка Е. А. Аксёновой

Подписано в печать 20.06.08. Формат 60x84 1/16.
Офсетная печать. Уч.-изд. л. 4. Тираж 120 экз. Изд. № 115.

Государственное образовательное учреждение
высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Отпечатано в типографии Издательства ПетрГУ
185910, Петрозаводск, пр. Ленина, 33