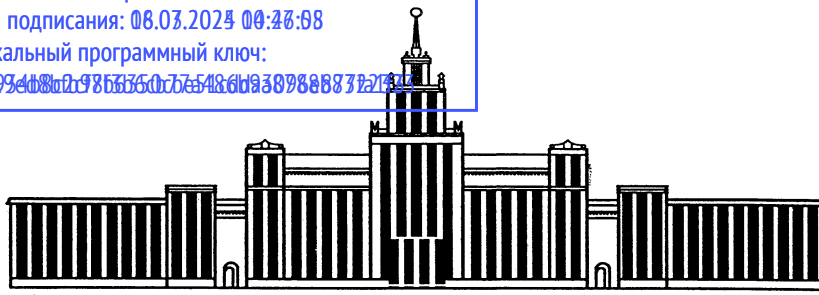
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ



ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

004(07)
К29

М.Ю. Катаргин

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

Челябинск
2015

Министерство образования и науки Российской Федерации
Южно-Уральский государственный университет
Кафедра прикладной математики

004(07)
К29

М.Ю. Катаргин

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

Челябинск
Издательский центр ЮУрГУ
2015

УДК 004.021(075.8)
К29

Одобрено
учебно-методической комиссией
факультета математики, механики и компьютерных наук

Рецензенты:
А.В. Мельников, В.Г. Попов

Катаргин, М.Ю.

К29 Алгоритмы и структуры данных: учебное пособие / М.Ю. Катаргин. – Челябинск: Издательский центр ЮУрГУ, 2015. – 101 с.

Материал учебного пособия соответствует Федеральному государственному образовательному стандарту высшего профессионального образования третьего поколения.

В пособии излагаются основные типовые структуры данных и наиболее распространенные алгоритмы, использующие эти структуры. Для некоторых алгоритмов приводятся не только описания, но и реализация их на языке С. Приводятся примеры программ с использованием изучаемых структур.

Издание предназначено для студентов направлений бакалавриата: «Прикладная математика и информатика», «Прикладная математика», «Программная инженерия».

УДК 004.021(075.8)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Основное содержание курса "Алгоритмы и структуры данных" составляет изложение типовых структур данных, методов их хранения и алгоритмов работы с ними. Изложенные методы и средства, по нашему убеждению, должны входить в инструментарий, которым должен владеть каждый профессиональный программист. К изучению курса можно приступать после того, как прослушаны курсы, посвященные алгоритмическим основам программирования.

Пособие предназначено для обучения основам программирования студентов факультета математики, механики и компьютерных наук всех направлений бакалавриата.

В пособии рассмотрены такие структуры, как стек, очередь, массив. В разделе, посвященном списочным структурам, описаны методы представления линейных и нелинейных списков, в частности, деревьев. Значительное внимание уделено методам представления таблиц. Рассмотрены последовательные, сортированные, древовидные и рассеянные таблицы. Этот раздел, по сути, является введением в курс ”Базы данных” и содержит описание их методов физической организации. Отдельный раздел посвящен методам сортировки, как внутренней, так и внешней.

Для каждой структуры данных или для алгоритма работы с ней анализируются те возможности, которые они предлагают и те недостатки, которые они в себе несут. Для алгоритмов, как правило, дана оценка порядка быстродействия. Большинство алгоритмов иллюстрируется текстами их программной реализации на языке Си. Реализация алгоритмов зачастую приводится в упрощенной трактовке, в ущерб эффективности, чтобы не загромождать изложение техническими деталями. Так, например, в разделе ”Внутренняя сортировка” рассматривается, в основном, сортировка целых чисел. Все программные тексты отлажены в среде Borland Dveloper Studio, Turbo C++.

В конце разделов приводятся контрольные вопросы, что позволяет по мере обучения проверять степень усвоения пройденного материала.

1. ТИПЫ СТРУКТУР ДАННЫХ

Любой набор знаков, рассматриваемый безотносительно к его содержательному смыслу, называют данными. Данные обычно изображают некоторую информацию, которую можно получить, если известен смысл, приписываемый данным. Однако в программировании обычно приходится иметь дело именно с данными. Например, разрабатывая систему хранения и поиска текстов, программист может и не знать их содержания. Его задача – обеспечить экономное использование памяти и быстрый поиск требуемых текстов по заданным признакам. Для решения этой задачи достаточно знать лишь количественные характеристики текстов, рассматриваемых как данные. Вообще, компьютеры выполняют только обработку данных, которая заинтересованным лицам представляется как обработка информации.

Совокупности данных, организованные некоторым образом, называются структурами данных. Структура данных определяется набором элементов и отношениями между ними. Структура данных в программе отображается в ту или иную структуру хранения. Так, например, стек может храниться в массиве или линейном списке; массив может быть представлен как последовательность элементов, ортогональный список или дерево и т.д. В настоящем пособии рассматриваются типовые структуры данных, такие как стек, очередь, массив, множество, списки – линейные и нелинейные, различные модификации деревьев; особое внимание уделено четырем базисным методам представления таблиц.

2. СТЕК

Понятие стека ввёл Алан Тьюринг в 1946г. Стек – одномерный, динамически изменяемый набор данных. Новый элемент всегда добавляют к одному и тому же концу набора, называемому вершиной стека. Удаление элемента допустимо тоже только из вершины стека. Таким образом, над стеком возможны только две операции: поместить в стек и взять из стека. Стек называют также магазином по аналогии с магазином огнестрельного оружия, в котором патрон последним вставленный в магазин, первым поступает в ствол. По этой же причине дисциплину работы со стеком называют "последний пришел, первый ушел". В английском языке используется аббревиатура LIFO (Last In, First Out). Стек может быть реализован в массиве или в списке. Реализация стека, использующего массив, приведена ниже. В примере предполагается, что стек содержит целые числа.

// Определение структуры, содержащей данные стека
Файл "stack.h"

```
#ifndef _STACK_H
#define _STACK_H
```

```
#define MAXSTACK 20 // максимальное число элементов в стеке
#include "vcl.h"

//---------------------------------------------------
struct STACK {
// элементы стека -целые числа
    int A[MAXSTACK]; // массив для элементов стека
    int v; // указатель на вершину стека
           // (индекс элемента на вершине стека)
    bool OverFlow; // индикатор переполнения стека
    bool Empty; // признак "стек пуст"
};

void StackInit(STACK *Stack);
void StackPush(STACK *Stack, int NewElement);
int StackPop(STACK *Stack);
AnsiString StackToString(STACK *Stack);
#endif

// Функции, обслуживающие стек (файл <stack.cpp>)
#include "stack.h"
void StackInit(STACK *Stack){
// инициализация стека
Stack->v=-1; // стек пуст
Stack->OverFlow=false; // стек не переполнен
Stack->Empty=true; // стек пуст
}
//---------------------------------
void StackPush(STACK *Stack, int NewElement){
// поместить в стек значение NewElement
if(Stack->v>=MAXSTACK){
     // переполнение
     Stack->OverFlow=true;
     return;
}
Stack->A[++(Stack->v)]=NewElement;
Stack->Empty=false;
}
//---------------------------------
int StackPop(STACK *Stack){
int p=-1; // -1 возвращается, если стек пуст
if(!Stack->Empty){
     p=Stack->A[v--];
     if(Stack->v<0){
         Stack->Empty=true;
     }
}
return p;
}
```

2.1. Применения стека

Ниже рассмотрены некоторые наиболее известные применения стека

2.1.1. Программный стек

Многие компиляторы языков программирования (в частности компиляторы C и C++) используют так называемый программный стек для размещения аргументов подпрограмм (функций) и локальных переменных. Программный стек – некоторая область памяти, используемая программой для размещения своих объектов. Рассмотрим в качестве примера следующую программу, состоящую из трех функций – main, f1, f2.

```
void main(void){
char x,y;
int z; // 1
x=f1(x,z); // 3
.....
z=f2(z);  // 7
}
//---------------------
char f1(char t, int u){
float q;
// 2,5
.....
}
//---------------------
int f2(int k){
char p=7;
     // 4
p=f1(1,2); // 6
.....
}
```

В процессе работы, программа проходит ряд точек, отмеченных в программе комментариями:

1 – после входа в функцию main
2 – после входа из main в f1
3 –после выхода из f1 в main
4 –после входа в f2 в main
5 – после входа в f1 из f2
6 – после выхода из f1 в функции f2
7 – после выхода из f2 в функции main

При входе в функцию программа помещает в программный стек значения аргументов в байты памяти, отводимой для параметров и локальных переменных, после чего вершина стека поднимается (Push). При выходе из функции, вершина стека опускается до уровня, предшествующего входу (Pop). В таблице 1 изображено состояние стека в

различных точках программы. По горизонтали – адреса байтов программного стека, по вертикали – номера точек.

Таблица 1

Состояние стека в процессе выполнения программы

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	x	y	z																	
2	x	y	z				t	u				q								
3	x	y	z																	
4	x	y	z				k				p									
5	x	y	z				k				p	t	u				q			
6	x	y	z				k				p									
7	x	y	z																	

Примечания:

- здесь не учитывается выравнивание адресов на границу слова, двойного слова,
- длина типа int – 4 байта,
- не учитывается что адрес точки возврата при обращении к функции тоже помещается в стек.

Использование механизма программного стека для распределения памяти для параметров и локальных переменных приводит к тому, что никаких дополнительных усилий для реализации рекурсии уже не требуется. Каждое новое обращение к рекурсивной функции помещает в стек новое поколение параметров и локальных переменных.

2.1.2. Реализация рекурсии

Некоторые языки программирования не допускают использования рекурсивных функций, то есть таких функций, которые обращаются сами к себе прямо или косвенно через другие функции. Используя стек, который функция должна обслуживать сама, программист, тем не менее, может реализовать рекурсивные алгоритмы. Типичными задачами, решаемыми с помощью рекурсии, являются задачи перебора с возвратами (back track), а также алгоритмы, в которых часть работы выполняется как вся работа. В качестве примеров рассмотрим две задачи: *обход шахматной доски ходом коня* и *вычисление определенного интеграла.*

2.1.2.1. Обход шахматной доски ходом коня

	3		2	
4				1
		X		
5				8
	6		7	

Рис. 1 Ходы коня

Конь должен побывать в каждой клетке доски размером $N \times N$ ровно один раз. Примем нумерацию возможных ходов коня из клетки как это изображено на рисунке. В стеке будем хранить координаты покидаемой клетки и номер выполняемого хода. Когда обнаружится, что никакой ход из клетки невозможен, берем координаты

предыдущей клетки из стека и пытаемся покинуть ее с помощью хода, имеющего номер на единицу больше, чем тот ход, с помощью которого ее покидали ранее (операция "*возврат*"). Если уже посетили все $N \times N$ клеток, то задача решена, и трасса обхода лежит в стеке. Если при попытке взять клетку из стека, обнаруживается, что стек пуст, то это означает, что задача не имеет решения. Ниже приведен текст программы.

```
struct STACK{
        int i,j;  // координаты клетки
        int move; // номер хода, которым покидаем клетку
}; // элемент стека
bool Horse(int n, STACK *stack);
//---------------------------------
static bool Move(int N, int i1,int j1, int n, int *i2, int
*j2){
// Ход коня. Конь ходит из клетки i1,j1 ходом номер n
// и попадает в клетку *i2,*j2
// N - размер доски
// Возвращает true при успехе, т.е. если ход допустим.
switch(n){
    case 0:
        *i2=i1+1;
        *j2=j1+2;
        break;
    case 1:
        *i2=i1+2;
        *j2=j1+1;
        break;
    case 2:
        *i2=i1+2;
        *j2=j1-1;
        break;
    case 3:
        *i2=i1+1;
        *j2=j1-2;
        break;
    case 4:
        *i2=i1-1;
        *j2=j1-2;
        break;
    case 5:
        *i2=i1-2;
        *j2=j1-1;
        break;
    case 6:
        *i2=i1-2;
        *j2=j1+1;
        break;
    case 7:
```

```
        *i2=i1-1;
        *j2=j1+2;
        break;
    default:
        // сюда можно попасть только при ошибке в программе
        *i2=INT_MAX;
        *j2=INT_MAX;
}
return (*i2>=0 && *i2<N && *j2>=0 && *j2<N);
}
//----------------------------------
bool Horse(int n, STACK *stack){
// n - размер доски.
// результат будет получен в стеке stack.
// Каждый раз, когда покидаем клетку, ее координаты
//   и номер хода помещаем в стек.
// возврат true - успех
bool  **Board=NULL;  //  в  массиве  доска  n*n  помечаем  факт
посещения клетки
int i,j; // текущие координаты клетки
int v; // указатель стека
int BegMove,k,i2,j2;
bool out=true,Good;

// выделим память для доски
Board=new bool * [n];
for(i=0;i<n;i++){
    Board[i]=new bool[n];
}
// Board[i][j]==true означает, что клетка посещалась
// Ни одна клетка еще не посещалась
for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        Board[i][j]=false;
    }
}
v=-1;       // стек пуст
i=0;j=0;    // начальная клетка
BegMove=0; // Номер хода, с которого начинается перебор
M:
// ищем допустимый ход из текущей клетки
Good=false; // предикат "Ход найден"
for(k=BegMove;k<8;k++){
    if(Move(n,i,j,k,&i2,&j2) && !Board[i2][j2]){
        Good=true;
        break;
    }
}
if(Good){
```

```
        // текущую клетку и номер хода помещаем в стек
        v++; stack[v].i=i; stack[v].j=j; stack[v].move=k;
        // пометить клетку, как посещенную
        Board[i][j]=true;
        BegMove=0;
        i=i2; j=j2;
        // может обошли все ?
        if(v==n*n-2){
            // последнюю клетку - в стек
            v++; stack[v].i=i2; stack[v].j=j2;
            goto finish;
        }
        goto M;  // все повторить для очередной клетки
    } else {      // ход не найден
        // взять из стека, если он не пуст
        if(v<0){ // стек пуст - задача не имеет решения
            out=false;
            goto finish;
        }
        // взять из стека
        i=stack[v].i; j=stack[v].j; BegMove=stack[v].move+1; v--;
        Board[i][j]=false;
        goto M;
    }
    finish:
    // освободим память
    for(i=0;i<n;i++){
        delete [] Board[i];
    }
    delete [] Board;
    return out;
    }
```

2.1.2.2. Общая схема решения задачи перебора с возвратами

В общем случае схема решения задачи перебора с возвратами имеет вид:

```
bool  BackTrack(параметры){
// опустошить стек
// задать начальные значения переменным
bool Good=false; // Good – решение найдено
M:
bool Find=Найти следующий допустимый узел в дереве решений();
if(Find){
    // данные узла поместить в стек
    if(пройдены все пути в дереве){
        Good=true;
        // решение находится в стеке
        goto finish;
```

```
    }
    goto M;  // повторить процесс исходя
             // из нового узла дерева решений
} else {     // новый узел не найден
    // взять из стека предыдущий узел, если стек не пуст
    if(v<0){ // стек пуст - задача не имеет решения
        Good=false;
        goto finish;
    }
    // взять из стека данные предшествующего узла
    // и сделать его текущим
    goto M;
}
finish:
// приберём за собой (освободим память, закроем файлы…)
return Good;
}
```

2.1.2.3. Вычисление определенного интеграла

Следующий пример иллюстрирует применение стека для решения задачи, в которой часть работы выполняется по тому же алгоритму, что и вся работа. Каждое значение интеграла $\int_a^b f(x)dx$ будем вычислять по формуле прямоугольников (хотя применяемый метод безразличен и можно было бы использовать любой).

Вычисляем два очередных приближения интеграла –

– как площадь одного прямоугольника:

$$I1 = f(a)(b-a)$$

– как площадь двух прямоугольников:

$$I2 = \frac{b-a}{2}[f(a) + f(\frac{a+b}{2})]$$

Если два приближения отличаются друг от друга более чем на назначенную меру погрешности *eps*, делим интервал интегрирования *(a,b)* пополам, и левую половину сразу обработаем тем же алгоритмом, а обработку правой отложим на потом – в стек. В противном случае , *I2* – окончательный ответ. Прибавим его к уже вычисленной части результата и поинтересуемся, не осталось ли отложенной на потом работы в стеке. Если ДА, то возьмемся за ее выполнение, если НЕТ – решение получено.

Текст функции, использующей ручное ведение стека для реализации рекурсии, приведен ниже:

```
#include <math.h>
#include <values.h>
struct INTERVAL{
    double a,b;
};
```

```
#define MAXSTACK 100
//---------------------------------------
double Integral(double a, double b, double (*f)(double),
  double eps){
// вычисление определенного интеграла от a до b с точн. eps
// от ф-ии f по формуле прямоугольников
double I1,I2; // два последовательных приближения
double d;
double Result=0;
int v=-1; // указатель стека
INTERVAL Stack[MAXSTACK];

Again:

d=b-a;
I1=(*f)(a)*d; // счет на одном интервале
d=(b-a)/2; // ширина интервала
I2=((*f)(a)+(*f)((a+b)/2))*d; // счет на 2х интервалах

if(fabs(I1-I2)<eps){
    Result+=I2;
    if(v>=0){ // если стек не пуст - взять из стека
        a=Stack[v].a;
        b=Stack[v].b;
        v--;
        goto Again;
    } else {
        // стек пуст - решение получено
        return Result;
    }
} else {
    // правую часть - в стек
    // а левой займемся немедленно
    v++;
    if(v>=MAXSTACK){
        MessageBox(0,"Переполнение стека","Ошибка",MB_OK);
        return MAXDOUBLE; // см. values.h
    } else {
        Stack[v].a=(a+b)/2;
        Stack[v].b=b;
        // a остается без изменений
        b=(a+b)/2;
        goto Again;
    }
}
}
```

2.1.3. Польская инверсная запись (ПОЛИЗ)

ПОЛИЗ, или, как её еще называют, *обратная польская запись*, это способ бесскобочного представления выражений (не только арифметических), в которых операнды предшествуют операции. Например, выражение

$$(4 \times A - 2 / X) \times (3 \times B + 2 \times Y)$$

в обратной польской записи буде выглядеть следующим образом:

$$4A \times 2X / - 3B \times 2Y \times + \times$$

Выражение, представленное в ПОЛИЗ, замечательно тем, что оно может быть вычислено за один проход слева направо без возвратов и забеганий вперед. Вычисление выражения в ПОЛИЗ выполняется следующим образом. Двигаясь слева направо по строке, операнды помещаем в стек. Когда встретится операция, то она выполняется над операндами, находящимися на вершине стека. Результат операции помещается в вершину стека вместо использованных операндов. Таблица 2, приведенная ниже, иллюстрирует процесс вычисления. В столбце "Входная строка" подчеркнута обработанная часть строки. Промежуточные результаты обозначены R_0,R_1,R_2. Окончательный результат вычисления – единственный элемент на вершине стека операндов.

Таблица 2

Вычисление значения выражения, заданного в ПОЛИЗ

Входная строка	Стек	Пояснение
4 A * 2 X / − 3 B * 2 Y * + *	4	
4 A * 2 X / − 3 B * 2 Y * + *	A4	
4 A * 2 X / − 3 B * 2 Y * + *	R_0	R_0=4*A
4 A * 2 X / − 3 B * 2 Y * + *	2R_0	
4 A * 2 X / − 3 B * 2 Y * + *	X2R_0	
4 A * 2 X / − 3 B * 2 Y * + *	R_1R_0	R_1=2/X
4 A * 2 X / − 3 B * 2 Y * + *	R_0	R_0=R_0−R_1
4 A * 2 X / − 3 B * 2 Y * + *	3R_0	
4 A * 2 X / − 3 B * 2 Y * + *	B3R_0	
4 A * 2 X / − 3 B * 2 Y * + *	R_1R_0	R=3*B
4 A * 2 X / − 3 B * 2 Y * + *	2R_1R_0	
4 A * 2 X / − 3 B * 2 Y * + *	Y2R_1R_0	
4 A * 2 X / − 3 B * 2 Y * + *	$R_2R_1R_0$	R_2=2*Y
4 A * 2 X / − 3 B * 2 Y * + *	R_1R_0	R_2=R_1+R_2
4 A * 2 X / − 3 B * 2 Y * + *	R_0	R_0=R_0*R_1

Отметим, что в обратную польскую запись может быть преобразована любая компьютерная программа. В области архитектуры ЭВМ в своё время это привело к созданию безадресных ЭВМ, в области программного обеспечения – к созданию так называемых "прямых" методов трансляции. Весьма популярным в течение некоторого времени был язык Forth (Форт), полностью базирующийся на ПОЛИЗ.

Рассмотрим алгоритм преобразования скобочного выражения в ПОЛИЗ. Для простоты ограничимся арифметическими выражениями, использующими четыре действия арифметики. Предположим также, что операндами являются только переменные с однобуквенными идентификаторами. Преобразование выполняется за один проход. Строка просматривается слева направо.

Операнды сразу помещаются в выходную строку.

Знаки операций сначала помещаются в стек. Прежде чем быть помещенным в стек, знак операции выталкивает из стека все операции, которые имеют приоритет больше или равный приоритета входной операции.

Открывающая скобка просто помещается в стек как операция с самим низким приоритетом.

Закрывающая скобка выталкивает из стека в выходную строку все операции вплоть до ближайшей открывающей скобки, которая удаляется из стека, но в выходную строку не помещается. Закрывающая скобка в стек не помещается.

После того как входная строка закончилась, остаток стека выталкивается в выходную строку.

Текст алгоритма преобразования приведен ниже.

```
struct opri{ // знаки операций и их приоритеты
    char oper; // операция
    unsigned char prior; // приоритет
};
//---------------------------------------------
static opri tpri[]={ // таблица приоритетов
    {'+',1},
    {'-',1},
    {'*',2},
    {'/',2}
};
// Приоритет операции из входной строки
static unsigned char vhpri;

//---------------------------------------------
static char Class(char z){
// Классификация символов из входной строки
// Символ может быть отнесен к одному из классов:
//   - 'б' – буква (операнд)
```

```
//    - 'o' - операция
//    - '('
//    - ')'
//    - 'н' - недопустимый символ
int i;

if(z=='(' || z==')') return z;
if(isalpha(z)) return '6'; // буква - операнд
for(i=0;i<sizeof(tpri)/sizeof(opri);i++){
    if(z==tpri[i].oper){
        vhpri=tpri[i].prior;
        return 'o'; // знак операции
    }
}
return 'н'; // недопустимый символ
}

//-------------------------------------------------
int Poliz(char *in,char *out){
// in - входная строка
// out - выходная строка
// функция возвращает 0 в случае успеха или номер ошибки
int i,j,v;
opri Stack[MAXSTACK];
j=-1; // Номер очередного выходного символа
v=-1; // указатель на вершину стека
for(i=0;i<strlen(in);i++){
    switch (Class(in[i])) {
        case '6': // буква
            // Операнды сразу помещаются в выходную строку
            out[++j]=in[i];
            break;
        case 'o': // операция
            // Операция выталкивает из стека все операции
            // с приоритетом >=
            while(v>=0 && vhpri<=Stack[v].prior){
                out[++j]=Stack[v--].oper;
            }
            // после чего сама помещается в стек
            Stack[++v].oper=in[i];
            Stack[v].prior=vhpri;
            break;
        case '(':
            // Открывающая скобка просто помещается в стек
            // как операция с самым низким приоритетом
            Stack[++v].oper=in[i];
            Stack[v].prior=0;
            break;
        case ')':
```

```
            // Закрывающая скобка выталкивает из стека всё,
            // вплоть до открывающей скобки включительно
            // но сама в стек не помещается
            for(;v>=0;v--){
                if(Stack[v].oper=='('){
                    break;
                } else {
                    out[++j]=Stack[v].oper;
                }
            }
            if(v<0){
                return 1; // Непарная открывающая скобка
            }
            v--;
            break;
        case 'н':
            return 2; // недопустимый символ
        default:
            return 9; // Ошибка в программе
    } /* switch */
    out[j+1]=0;
} /* for i */
/* Остаток стека - на выход */
for(;v>=0;v--){
    if(Stack[v].oper=='('){
        // Непарная открывающая скобка
        return 3;
    }
    out[++j]=Stack[v].oper;
}
out[++j]=0;
return 0;
} // Poliz
```

Контрольные вопросы

1. Какие объекты помещаются в программный стек при входе в функцию?
2. Как изменяется указатель программного стека при выходе из функции?
3. Что происходит с программным стеком при рекурсивном обращении функции к самой себе?
4. Напишите обратную польскую запись для выражения

$$\frac{(7q+x)(t-\frac{p}{q})}{\frac{a}{b}-\frac{b}{a}}+\frac{p-q}{8x}$$

5. Напишите последовательность операций и состояние стека в процессе вычисления значения выражения, представленного польской записью из предыдущего примера.

3. ОЧЕРЕДЬ

Очередью называют одномерную структуру данных с дисциплиной обслуживания "первый пришел – первый ушел". Такую дисциплину обслуживания также называют FIFO (First In – First Out), что означает то же самое. Все включения в очередь делаются на одном конце, а все исключения – на другом. Очередь может быть представлена массивом или списком. В этом разделе рассматривается представление очереди массивом (рис. 2).

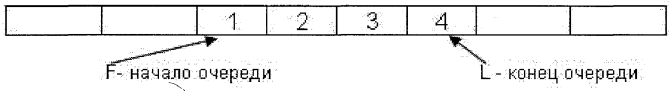


Рис. 2. Очередь в массиве

Простейшее решение состоит в том, чтобы иметь два указателя для начала и конца очереди – F и R. F указывает место на начало очереди; R указывает на последний элемент очереди.

При самом простодушном подходе к выполнению операций включение и исключение выполняются следующим образом:

```
/* Поместить в очередь */ R=R+1; X[R]=Y;
/* Взять из очереди */    F=F+1; Y=X[F];
```

Заметим, что в процессе выполнения серии включений и исключений, очередь будет "ползти" вдоль памяти, так как F и R непрерывно увеличиваются. Это означает чрезвычайно расточительное использование памяти. Проблему можно решить, отводя под очередь M позиций и неявно замыкая её в кольцо, так, что за последней позицией следует первая. Пусть очередь представлена структурой:

```
#define M 8 // длина массива, отводимого под очередь
struct QUEUE{
    int Q[M]; // массив, используемый для хранения очереди
    int F; // ИНДЕКС первого элемента очереди
    int L; // ИНДЕКС ПОСЛЕДНЕГО
    bool IsEmpty; // признак пустой очереди
};

QUEUE T; // экземпляр очереди над которой выполняются операции
// Функции «поместить в очередь» и «взять из очереди»:

bool PutInQueue(int x){
// поместить элемент x в очередь
// возвращает истину в случае успеха
```

```
int N=(T.L+1)%M;
if(!T.IsEmpty && T.F==N){
    return false;
} else {
    T.L=N;
}
T.Q[T.L]=x;
T.IsEmpty=false;
return true;
}

int TakeFromQueue(){
// Взять элемент из очереди T
// функция возвращает взятый из очереди элемент
// если очередь пуста, то возвращается INT_MAX
//  (константа, определённая в limits.h)
int z;
bool fl=T.F==T.L;
if(!T.IsEmpty){
    z=T.Q[T.F];
    T.F=T.F==M-1 ? 0 : T.F+1;
    T.IsEmpty=fl;
} else {
    z=INT_MAX;
}
return z;
}
```

Очередь в программировании используется, как и в реальной жизни, когда нужно совершить какие-то действия в порядке их поступления, выполнив их последовательно. Примером может служить организация обработки событий в Windows. Когда пользователь оказывает какое-то действие на приложение, то в приложении не вызывается соответствующая процедура (ведь в этот момент приложение может совершать другие действия), а ему присылается сообщение, содержащее информацию о совершенном действии, это сообщение ставится в очередь, и только когда будут обработаны сообщения, пришедшие ранее, приложение выполнит необходимое действие.

Аналогично, очереди могут выстраиваться к любым ресурсам общего пользования – процессору, принтеру…

Контрольные вопросы

1. По каким правилам элементы помещаются в очередь и удаляются из очереди?
2. Как избежать перемещения очереди по памяти?
3. Вычислите значение $U_{1000000}$, если $U_n=U_{n-1}+2U_{n-2}-3U_{n-3}$ и $U_0=U_1=U_2=U_3=1$

4. МАССИВЫ

Массив – набор однотипных компонентов (элементов), доступ к которым осуществляется по индексу (индексам).

4.1. Размещение прямоугольных массивов

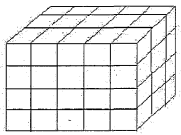


Рис.3 Трёхмерный массив

В качестве примера рассмотрим трёхмерный прямоугольный массив $3 \times 4 \times 5$ (3 слоя по 4 строки по 5 элементов в строке). Такой массив можно представить в виде параллелепипеда (рис. 3.). Предположим, что элементы массива следуют в памяти в лексико-графическом порядке, то есть так, что первым изменяется последний индекс. Такой порядок используется в большинстве алгоритмических языков, но есть и исключения, например, Фортран. Порядковый номер элемента массива *A[i][j][k]* можно вычислить по формуле:

Порядковый номер элемента массива *A[i][j][k]* можно вычислить по формуле:

Порядковый номер=i*объём слоя+j*размер строки+k

В общем случае описание массива может иметь вид:

$$A[i_1:j_1][i_2:j_2]\ldots[i_n:j_n]$$

где i_m, j_m – нижняя и верхняя граница индекса в m-м измерении. Каждый раз, когда программа обращается к элементу массива, она должна вычислить его адрес по заданным индексам $k_1,k_2\ldots k_n$. Функция упорядочения, которая это делает, имеет вид:

$$L = b + \sum_{m=1}^{n} (k_m - i_m) \times D_m$$

где L – искомый адрес; b – начальный адрес массива (адрес его 1-го элемента); D_m – объем m-го сечения массива (слоя, строки…) в байтах. В общем случае функцию упорядочения можно записать в виде:

$$L = c + \sum_{m=1}^{n} k_m \times D_m \text{, где}$$

$$c = b - \sum_{m=1}^{n} i_m \times D_m$$

Константу c называют базовым адресом массива, который определяется как адрес возможно несуществующего элемента массива, у которого все индексы равны нулю.

Пример: для массива с границами индексов 3:5,1:4,0:1 имеем:

$D_3=1$

$D_2=(j_3-i_3+1)*D_3=2$

$D_1=(j_2-i_2+1)*D_2=8$

$c=b-(i_1*D_1+i_2*D_2+i_3*D_3)=26*(b-1)$

И функция упорядочения имеет вид:

$L=25*(b-1)+8*k_1+2*k_2+k_3$

Функция упорядочения строится компиляторами автоматически. Исходные данные о массиве представляются обычно в виде информационного вектора массива:
I_A=(n, число элементов, C, D_1, D_2, ...D_n, i_1,j_1, i_2, j_2, ...i_n, j_n)

4.2. Метод Айлиффа

Метод Айлиффа доступа к элементам массива пригоден для работы, как с прямоугольными, так и непрямоугольными массивами. В некоторых случаях он может оказаться более быстродействующим, так как не использует операций умножения, однако метод требует дополнительной памяти. В качестве примера рассмотрим представление двумерного массива, в котором длина i-й строки равна $i+1$, как это представлено на рис. 4. Выделение памяти для такого массива выглядит следующим образом:

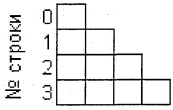


Рис.4

```
// выделим память для массива из 4-х указателей на строки
double  **a=new double *[4];
// выделим память для каждой строки
for(int i=0; i<4; i++){
  a[i]=new double[i+1];
}
```

Теперь имеем право обращаться к элементам массива как обычно: *a[i][j]*

Заметим, что запись *a[i][j]* эквивалентна записи **(*(a+i)+j)*, что в действительности и происходит при обращении к элементу массива *a[i][j]*:

1. в переменной *a* находится адрес начала массива указателей на строки.
2. прибавив к нему *i*, получим адрес *a+i* указателя на i-ю строку извлечем из него адрес начала i-й строки: **(a+i)*
3. прибавим к нему *j* и получим адрес *j*-го элемента *i*-й строки: **(a+i)+j*
4. и, наконец, извлекаем по этому адресу значение элемента массива: **(*(a+i)+j)*

Как видно, операция умножения действительно не используется. Поскольку память для массива была взята из кучи, то впоследствии ее необходимо освободить. Освобождение выполняется в порядке обратном выделению:

```
for(int i=0; i<4; i++){
  delete [] a[i];
}
delete a;
```

Контрольные вопросы

1. Каким образом вычисляется адрес элемента многомерного массива по его индексам?

2. Вычислите адрес элемента массива *A[4][3][2]*, если его описание имеет вид *int A[2:5][1:7][0:3];* (здесь предполагается, что нижние границы изменения индекса могут быть заданы).
3. Напишите текст фрагмента программы, создающей непрямоугольный массив из 10 строк, в котором четные строки имеют по 5 элементов, а нечетные по 8 элементов.

5. СПИСОЧНЫЕ СТРУКТУРЫ

При выборе структуры хранения для некоторых данных, необходимо помнить, что помимо самих данных требуется хранить структурные связи каждого элемента данных с другими элементами. Так, например, для строки символов существенными являются не только сами символы, но и порядок их следования. Таким образом, элемент данных (символ) структурно связан с предыдущим и последующим элементом. Элемент числовой матрицы имеет соседей слева, справа, сверху и снизу. При последовательном распределении памяти структурные связи отображаются на физическое расположение элементов структуры данных в памяти. Так, для строки следующий символ располагается вслед за текущим.

В случае связанного распределения памяти, адреса элементов, структурно связанных с элементом данных, хранятся вместе с этим элементом.

5.1. Односвязный линейный список

Односвязный линейный список состоит из элементов (узлов), в каждом из которых помимо данных, содержится указатель на следующий элемент списка.

5.1.1. Представление односвязного списка

В качестве примера рассмотрим представления символьной строки в виде линейного списка. Каждый элемент данных хранит один символ и указатель на следующий элемент списка:

```
struct NODE{
    char Letter;          // символ
    NODE *Next;    // указатель на следующий элемент
};
```

Слово "КОРА" будет представлено списком, изображенном на рис. 5.

Рис. 5. Представление строки линейным списком

5.1.2. Операции над односвязным списком

В случае представления строки массивом вставка строки S1 в строку S2, начиная с позиции N, может быть выполнена следующим образом:

1. часть строки S2, начиная с символа N, сдвигается вправо на длину строки S1
2. строка S1 копируется в освободившееся пространство

Таким образом, операция вставки связана с физическим перемещением данных, объем которых может быть значительным. В случае использования списочной организации, операция вставки сводится к изменению нескольких указателей. На рис. 6 изображена операция вставки строки "ИЦ" в слово "КОРА" так, чтобы получилось слово "КОРИЦА". Пунктиром изображены связи после выполнения операции.

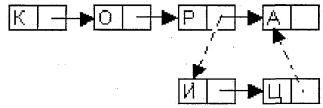


Рис. 6. Вставка в список

Рассмотрим операции вставки и удаления узлов.

```
NODE *InsertNode(NODE *p){
// функция выполняет вставку нового узла вслед за
// узлом p и возвращает указатель на новый узел
NODE *x;
x=new NODE;
x->Next=p->Next;
p->Next=x;
return x;
}
```

Отметим, что предложенная функция не может выполнить вставку узла в начало списка, так как для первого узла не существует предшественника. Следующая модификация функции вставки свободна от этого недостатка. Она отличается от предыдущей тем, что имеет аргументом указатель на указатель на узел и, следовательно, способна изменить указатель на начало списка.

```
NODE *InsertNode(NODE **p){
// функция выполняет вставку нового узла вслед за
// *p и возвращает указатель на новый узел
NODE *x;
x=new NODE;
x->Next=*p;
*p=x;
return x;
}
```

Теперь, если требуется выполнить вставку узла вслед за узлом, на который указывает указатель z, следует обратиться к функции:

```
S=InsertNode(&(z->Next));
```

А для вставки узла в начало списка, на который указывает указатель q, следует использовать обращение:

```
q=InsertNode(&q);
```

Другие, более безопасные методы, позволяющие избавиться от особенностей обработки крайних узлов списка, будут рассмотрены позже. Для удаления узла односвязного списка необходимо знать указатель на его предшественника:

```
bool DeleteNode(NODE *p){
// функция удаляет узел, следующий за p
// и возвращает true в случае успеха
NODE *t;
if(p->Next==NULL)return false;
t=p->Next;
p->Next=t->Next;
delete t;
return true;
}
```

Рассмотрим следующую простую задачу, иллюстрирующую технику работы с односвязным линейным списком. Проходя список:

1. удалить все вхождения символа ‘Ы‘,
2. заменить всюду '?' на '!'
3. после каждого вхождения символа 'А' добавить символ 'Б'.

Поскольку предстоит выполнение удалений, в алгоритме будут использованы два рабочих указателя – на текущий элемент и на его предшественника. На начало списка указывает *NODE *h*. Первый узел списка не обрабатывается (см. далее "голова списка").

```
NODE *p1, *p2, *x;
for(p1=h,p2=p1->Next; p2!=NULL;p1=p1->Next,p2=p1->Next){
  switch(p2->Letter){
        case 'Ы':
              p1->Next=p2->Next;
              delete p2;
              break;
        case '?':
              p2->Letter='!'
              break;
        case 'А':
              x=new NODE;
              x->Letter='Б';
              x->Next=p2->Next;
  } // switch
} // for
```

5.1.3. Голова списка

Для того, чтобы избавиться от особенностей вставки и удаления в начале списка и для того, чтобы иметь возможность отличить пустой список от несуществующего, в начало списка можно поместить специальный узел, называемый головой списка. Таким образом, даже пустой список содержит хотя бы один узел — голову. В таком случае, все

узлы списка, в тои числе и первый имеют предшественника и операции вставки и удаления в начале списка не имеют особенностей. В поле данных головы обычно помещают данные, которые не может иметь ни один рабочий узел списка.

5.1.4 Циклический список

В циклическом списке поле связи последнего узла указывает на начало списка, как это изображено на рис. 7.

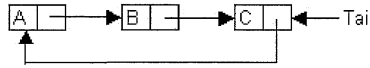


Рис.7. Циклический список

Циклический список обычно представлен в программе указателем на хвост (Tail), так как получить в этом случае указатель на начало не представляет труда. Для циклических списков становятся простыми операции сцепления и расщепления. Рис. 8 иллюстрирует операцию расщепления по узлу P. Пунктиром изображено изменение связей после операции.

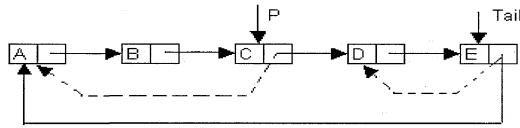


Рис.8. Расщепление списка

На языке С это может быть записано следующим образом:

```
NODE *x;
x=P->Next;
P->Next=Tail->Next;
Tail->Next=x;
```

Нетрудно заметить, что операция выполняет обмен местами указателей `P->Next` и `Tail->Next`. Следовательно, повторное применение той же операции приведет к сцеплению списков. Операция работает как кнопочный выключатель.

5.1.5. Представление стека линейным списком

Ранее рассматривалось представление стека массивом. Зачастую более удобно представлять стек в виде односвязного линейного списка как на рис. 9:

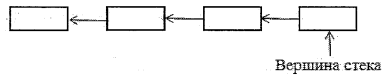


Рис. 9. Представление стека списком

Пусть структура узла списка:

```
struct NODE {
      <тип> Data; // данные
      NODE *Next; // указатель на следующий элемент списка
}
```

Операции над стеком принимают вид:

```
// операция «поместить в стек
// v – указатель на вершину стека
NODE *x=new NODE;
x->Next=v;
v=x;
// операция «взять из стека
NODE *x=v;  // элемент, снимаемый с вершины стека
v=v->Next;
```

5.1.6. Пример. Сложение многочленов

В этом примере исходными данными являются два многочлена от трех переменных:

$$P(x,y,z) = \sum_{i=0}^{n}\sum_{j=0}^{m}\sum_{k=0}^{l} A_{ijk}\, x^i y^j z^k$$

$$Q(x,y,z) = \sum_{i=0}^{n}\sum_{j=0}^{m}\sum_{k=0}^{l} B_{ijk}\, x^i y^j z^k$$

Априорно известно, что лишь немногие из коэффициентов этих полиномов отличны от нуля. Если воспользоваться прямоугольными массивами для хранения коэффициентов многочленов, то, во-первых, основной объем памяти будет потрачен на хранение нулей, а, во-вторых, основное время работы алгоритма будет потрачено на сложение с нулем и умножение на нуль.

В данном примере полином представлен линейным односвязным циклическим списком с головой. Структура узла имеет вид :

```
struct NODE{
      float A; // коэффициент при одночлене
      int px,py,pz; // степени x,y,z
      NODE *next;
};
```

Многочлен $3x^2y^2z+8x^2y^3z^2-7x^3yz^4$ будет представлен списком, изображенным на рис.9.

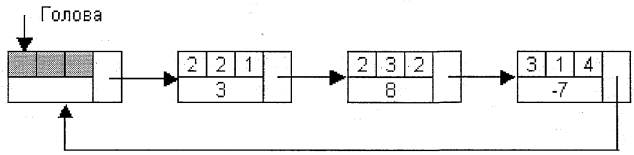


Рис.10. Многочлен в списке

Будем полагать, что показатели степеней *ijk* переменных следуют в списке в порядке возрастания (лексикографический порядок). Для сравнения степеней используется функция, возвращающая –1, 0,+1:

```
int PowerCmp(NODE *p, NODE *q){
if(p->px > q->px) return 1;
if(p->px < q->px) return -1;
if(p->py > q->py) return 1;
if(p->py < q->py) return -1;
if(p->pz > q->pz) return 1;
if(p->pz < q->pz) return -1;
return 0;
}
```

В поля px,py,pz головы списка поместим максимально возможные значения (INT_MAX (см. файл limits.h)). Зачем это нужно будет ясно позднее. Алгоритм выполняет операцию $P=P+Q$, то есть результат сложения будет получен на месте первого многочлена.

```
#define ZERO 1.0e-7 /* величины меньшие чем ZERO считаем равными нулю */
void PoliAdd(NODE *p, NODE *q){
// p,q - головы списков многочленов-слагаемых
// p=p+q
NODE *p1,*p2,*q1;
NODE *x;

for(p1=p,p2=p1->next,q1=q->next; q1!=q; ){
      switch(PowerCmp(p2,q1)){
          case 1:
               // p2>q1
               x=new NODE;
               memcpy(x,q1,sizeof(NODE));
               // вставка вслед за p1
               x->next=p2;
               p1->next=x;
               p1=x;
               p2=p1->next;
               q1=q1->next;
               break;
          case -1:
               // p2<q1
               p1=p2;
               p2=p2->next;
               break;
          case 0:
               // p2==q1
               p2->A+=q1->A;
               if(fabs(p2->A)<ZERO){
                    // удалить p2
                    p1->next=p2->next;
                    delete p2;
                    p2=p1->next;
               } else {
```

```
                    p1=p2;
                    p2=p2->next;
                }
                q1=q1->next;
                break;
        } //switch
    } // for
    }
```

Отметим, что приведенном алгоритме нет необходимости специально обрабатывать ситуацию, когда список p закончился, а список q нет. В головы списков помещены максимально возможные значения степеней, поэтому, когда закончится список p, его голова "пропустит" вперед себя весь остаток списка q.

5.2. Двусвязный линейный список

Основным недостатком односвязного списка является невозможность удаления узла, когда известен только указатель на него и неизвестен указатель на предшественника. Этого недостатка лишен двусвязный список, в котором каждый элемент имеет указатели на предшественника и на последователя (рис.10).

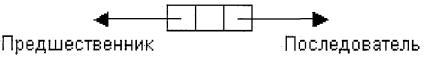


Рис.11. Узел двусвязного списка

Точно так же, как и односвязный список, двусвязный может иметь голову и быть кольцевым. На рис. 11 изображен кольцевой двусвязный список с головой.

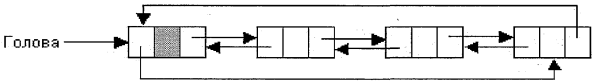


Рис.12. Двусвязный циклический список с головой

Узел двусвязного списка может быть представлен структурой:

```
struct UZEL{
    <тип> Info;          // данные узла
    UZEL *Left;  // указатель на предшественника
    UZEL *Right; // указатель на последователя
};
```

Операция вставки нового узла представлена на рис. 13.

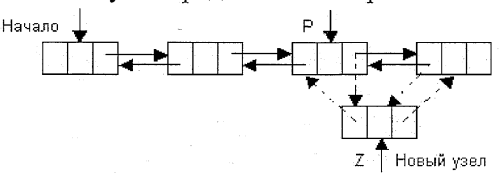


Рис.13. Вставка узла в двусвязный список вслед за узлом P

Как и прежде, пунктиром обозначены связи, изменившиеся после выполнения операции. Функция, выполняющая операцию вставки, приведена ниже.

```
UZEL *InsertUzel(UZEL *p){
// функция вставляет новый узел справа от P
// и возвращает указатель на новый узел
UZEL *Z;
Z=new UZEL;
If(Z==NULL) return NULL; // нет памяти в куче
Z->Left=p;
Z->Right=p->Right;
p->Right->Left=X;
p->Right=X;
return X;
}
```

Операция удаления узла изображена на рис.14.

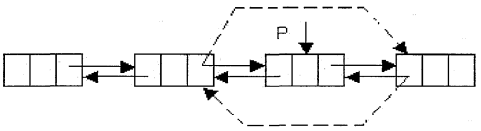


Рис. 14. Удаление узла из двусвязного списка

Функция, выполняющая удаление:

```
void DeleteUzel(UZEL *p){
p->Left->Right=p->Right;
p->Right->Left=p->Left;
delete p;
}
```

5.2.1. Представление очереди двусвязным линейным списком

Очередь удобно представлять двусвязным линейным циклическим списком с головой как на рис. 15.

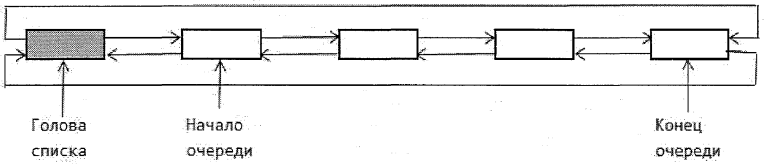


Рис.15. Представление очереди двусвязным списком

Приняты следующие соглашения:

– начало очереди находится справа от головы

– конец очереди – слева от головы

Теперь операции над очередью выполнятся следующим образом:

– поместить в очередь – вставить новый узел справа от головы

– взять из очереди – удалить узел слева от головы

5.3. ОРТОГОНАЛЬНЫЕ СПИСКИ

Линейные списки могут рассматриваться как некая альтернатива одномерным массивам. Для многомерных массивов такой альтернативой являются ортогональные списки. В качестве примера рассмотрим представление матрицы. Пусть имеется разреженная матрица, то есть такая матрица, в которой большинство элементов тривиально (обычно это значения 0 или ∞), и мы не хотим тратить память на их хранение и стремимся избежать многочисленных сложений с нулём и умножений на нуль при выполнении операций с такой матрицей. Пример такой матрицы приведен на рис.16.

0	4	0	5
3	0	0	7
6	0	9	2

Рис. 16. Разреженная матрица

Для представления элемента матрицы воспользуемся структурой:

```
struct EM {
    int Row,Col;  // строка и столбец элемента
    double Value; // значение элемента матрицы
    EM *Right;// указатель на последователя в строке
    EM *Down; // указатель на последователя в столбце
};
```

Ортогональный список, соответствующий матрице рис.16 изображен на рис. 17.

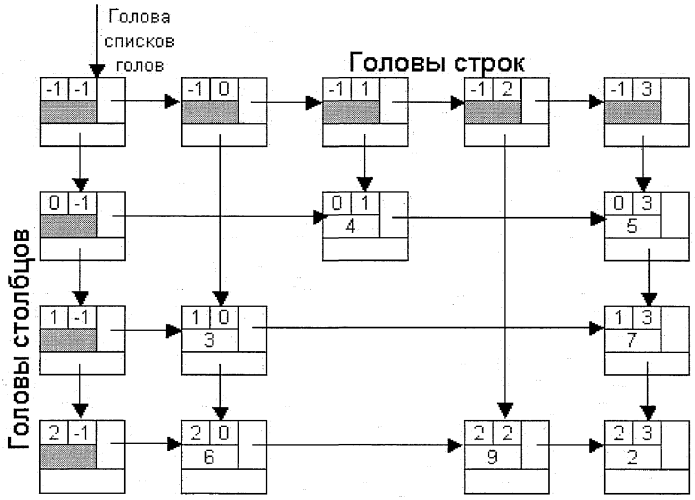


Рис. 17. Представление разреженной матрицы ортогональным списком

5.4. Списки общего вида

Списки общего вида отличаются от обычных линейных списков только одной деталью – элементом списка общего вида может быть список. Список общего вида является простым обобщением линейного списка. В случае односвязного списка в узел добавляется указатель на подсписок, как на рис. 18.

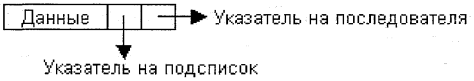


Рис. 18. Узел списка общего вида.

Такое определение рекурсивно, и список может содержать в качестве элемента самого себя. Рассмотрим списочную структуру:

L=(a:N,b,c:(d:N),e:L)

N=(f, g:(h:L, j:N))

Запись вида x:Y означает, что узел x содержит подсписок Y. Узлы списка перечисляются через запятую. Графически списочная структура изображена на рис. 19.

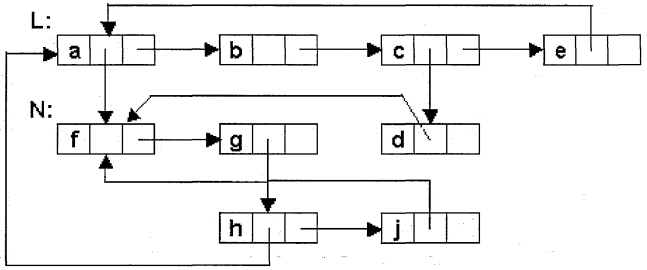


Рис. 19. Представление списка общего вида

Такое представление несёт с собой одну проблему. На рис. 19 видно, что на узел f имеется три ссылки. В действительности, это ссылки на список N, в котором узел f является начальным. Если потребуется удалить узел f из списка N, то придется регулярным образом обходить всю списочную структуру с целью обнаружения всех ссылок на узел f, что, конечно, неприемлемо.

Если потребовать, чтобы каждый список имел голову, то проблема исчезает – на каждый узел, не являющийся головой, имеется одна ссылка. Рассмотренная выше структура примет вид как на рис. 20.

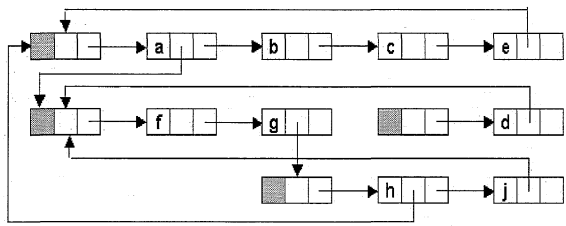


Рис. 20. Список общего вида с головами списков.

Списки, составляющие списочную структуру общего вида могут иметь любую организацию из рассмотренных выше – односвязные или двусвязные, кольцевые или не кольцевые, с головой или без, в зависимости от операций, которые необходимо выполнять над структурой.

5.4.1. Пример. Топологическая сортировка.

Топологической называют сортировку на множестве элементов в случае, когда на множестве элементов задано отношение *частичного порядка*. Порядок является частичным, если отношение предшествования задано не для всех пар элементов. Множество целых чисел является полностью упорядоченным, так как для любой пары целых чисел a,b определено отношение ' > ' (больше) или '< ' (меньше).

В качестве примера ситуации, когда порядок является частичным, рассмотрим производство следующих работ:

1) сделать болт;
2) сделать гайку;
3) навернуть гайку на болт.

Ясно, что работы 1 и 2 должны предшествовать работе 3. Запишем этот факт как $1 \prec 2$ (читается: 1 предшествует 2) и $2 \prec 3$. Работы 1 и 2 не связаны между собой отношением предшествования.

Другим примером, в котором имеется частичный порядок, является учебный план подготовки специалистов в ВУЗе. Курс "Основы алгоритмизации и языки программирования" должен предшествовать курсу "Алгоритмы и структуры данных", но ни тот, ни другой никак не связаны с курсом "Отечественная история".

Третий пример: пусть требуется написать библиотеку подпрограмм, в которой некоторые из подпрограмм вызывают другие подпрограммы из той же библиотеки. Очевидно, что приступать к отладке вызывающих подпрограмм следует после завершения отладки вызываемых.

Топологическая сортировка на основе имеющегося отношения частичного порядка строит линейную последовательность элементов сортируемого множества, в которой для любой пары X_i, X_j не может быть выполнено условие $X_i \prec X_j$ при i>j. Иными словами, для пары $a \prec b$, а не может появиться в выходной последовательности позже b.

Исходными данными для работы алгоритма является массив сортируемых элементов и массив пар элементов first, second, таких, что first ≺ second. Одна такая пара представляется структурой:

```
struct PAIR{
   char *first;
   char *second;
};
```

Для реализации алгоритма используются списочная структура, содержащая узлы двух типов: узлы основного списка (MAINS) и узлы подсписков (POD). Основной список двусвязный, с головой, циклический. Подсписок – односвязный, без головы и нециклический.

```
struct MAINS{ // структура узла основного списка
   int count; // счетчик числа элементов, предшествующих данному
   char name[40]; // имя элемента
   MAINS *Left;  // левая связь узла
   MAINS *Right;  // правая связь
   struct POD *pod;  // указатель на подсписок
};

struct POD { // структура узла подсписка
   POD *next;// следующий узел подсписка
   MAINS *m; // указатель на узел основного списка
};
```

Алгоритм имеет три фазы:

Первая фаза – первоначальное построение списочной структуры. Проходим массив пар, и для каждого элемента множества, который ещё не включен в список, создаем узел и помещаем его в хвост основного списка. Для каждой пары first,second в подсписок узла first помещаем узел, в поле m которого помещаем ссылку на узел основного списка second. Счетчик count узла second увеличиваем на единицу. После завершения первой фазы узел каждого элемента содержит число элементов, предшествующих данному. На рис.21 изображено состояние основного списка после завершения первой фазы для массива пар предшествования A≺B, A≺C,A≺D, C≺B, C≺D, D≺B

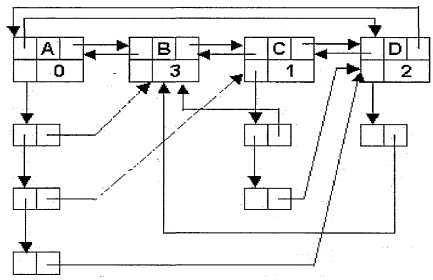


Рис. 21. Топологическая сортировка. 1 фаза.

Вторая фаза – создание списка ведущих элементов. Список ведущих имеет такую же структуру, что и основной список. Проходя основной список слева направо, узлы, имеющие поле счетчика, равное нулю, переносим в список ведущих элементов. Этим элементам ничто не предшествует, и они первыми будут выведены в выходную последовательность.

Третья фаза – построение выходной последовательности. Проходим список ведущих и его элементы помещаем в выходную последовательность и удаляем из списка ведущих. Проходя подсписок выводимого в результат элемента, уменьшаем на единицу поле счетчика того элемента основного списка, на который ссылается поле m подсписка. Действительно, после вывода элемента списка ведущих в результирующую последовательность, он уже не предшествует оставшимся в основном списке элементам. Если поле счетчика окажется равным нулю, переносим его в конец списка ведущих.

Если отношение предшествования было задано корректно, то все элементы будут выведены и основной список опустеет. Если же основной список по завершении третьей фазы не пуст, то это говорит о противоречивом задании частичного порядка. Например, частичный порядок $a \backslash b$; $b \backslash c$; $c \backslash a$ противоречив.

Ниже приведен текст функции, реализующей топологическую сортировку (для упрощения алгоритм предполагает, что не существует элементов, не входящих ни в какое отношение предшествования):

```
void TopSort(PAIR *p, int n_pair, FILE *result){
// p - массив пар указателей на имена элементов
// Наличие пары s1,s2 означает,
// что s1\s2
// n_pair - число таких пар
// result - файл, в который помещаются результаты

MAINS *mhead; // голова основного списка
MAINS *vhead; // голова списка ведущих элементов
MAINS *u1,*u2,*v,*w;
int i;
POD *x,*y;

// создание основного списка
mhead=new MAINS;
mhead->Left=mhead;
mhead->Right=mhead;

// проход по всем парам
for(i=0; i<n_pair; i++){
    // найдем или создадим элемент по имени p[i].first
    u1=FindU(mhead,p[i].first);
    // найдем или создадим элемент по имени p[i].second
```

```
    u2=FindU(mhead,p[i].second);
    // u1 предшествует u2, поэтому к счетчику u2 прибавим 1
    u2->count++;

    // в подсписок u1 добавим узел, указывающий на u2
    x=new POD;
    x->next=u1->pod;
    u1->pod=x;
    x->m=u2;
}

// создание списка ведущих
vhead=new MAINS; // голова
vhead->Left=vhead->Right=vhead;

// проходим по основному списку и узлы с полем счетчика ==0
// переносим в список ведущих
for(v=mhead->Right; v!=mhead; v=w){
    w=v->Right;
    if(v->count==0){
        // переносим в хвост списка ведущих
        AddToTail(vhead,v);
    }
}

// проход по списку ведущих
for(v=vhead->Right; v!=vhead; v=v->Right){
    // имя элемента печатаем в файл результата
    fprintf(result,"    \"%s\",\n",v->name);
    // проходим по подсписку и уменьшаем на 1 поле счетчика
    // в узлах основного списка, на которые ссылаются узлы
    // подсписка
    for(x=v->pod,y=x->next; x!=NULL; x=y){
        y=x->next;
        x->m->count--;
        if(x->m->count==0){
          // если счетчик==0, то переносим узел из основного
          // списка в конец списка ведущих
            AddToTail(vhead,x->m);
        }
        // узлы подсписка больше не понадобятся
        delete x;
    }
    // узлы списка ведущих тоже больше не нужны
    delete v;
}
delete vhead;

// осталось ли что-нибудь в основном списке ?
```

```
// если осталось, то это говорит о противоречивом
// задании предшествования и список содержит кольца
// элементов множества
// в любом случае надо все освободить
if(mhead->Right!=mhead){
    fprintf(result,"Список функций, образующих рекурсивные
цепи\n");
    for(v=mhead->Right; v!=mhead; v=v->Right){
        fprintf(result,"%3d %s\n",v->count, v->name);
        for(x=v->pod,y=x->next; x!=NULL; x=y){
            y=x->next;
            delete x;
        }
        delete v;
    }
}
delete mhead;
}
```

Рассмотренный алгоритм использует две вспомогательные функции:

```
MAINS *FindU(MAINS *head, char *WhatToFind){
MAINS *v;
// Функция либо находит в основном списке с головой head
// узел с элементом по имени WhatToFind, либо создает его
// В любом случае возвращается указатель на узел с
// элементом по имени WhatToFind

// ищем, и если найдем, вернем указатель
for(v=head->Right; v!=head; v=v->Right){
    if(strcmp(WhatToFind,v->name)==0){
        return v;
    }
}

// не нашли - создаем
v=new MAINS;
strcpy(v->name,WhatToFind);
v->count=0;
v->pod=NULL;
v->Right=head;
v->Left=head->Left;
head->Left->Right=v;
head->Left=v;
return v;
}

//-------------------------------------
void AddToTail(MAINS *head, MAINS *v){
// изъять узел v из основного списка и вставить
```

```
// в хвост списка ведущих с головой head

// удаляем из того списка, в котором он сейчас находится
v->Left->Right=v->Right;
v->Right->Left=v->Left;

// помещаем слева от head
v->Right=head;
v->Left=head->Left;
head->Left->Right=v;
head->Left=v;
}
```

Приведённое решение предполагает, что каждый элемент входит массив Pair (аргумент функции TopSort) по крайней мере 1 раз.

5.5. Стек свободного пространства

До сих пор мы не заботились о том, куда деваются освобождаемые узлы списков, перекладывая это на алгоритмы работы с "кучей". Это было возможно, потому, что наши списки располагались в оперативной памяти и в нашем распоряжении имелись операторы new и delete (или функции calloc и free). Существуют ситуации, когда программа сама должна заботиться об "утилизации" освобождаемых узлов списков. Такая ситуация возникнет, если отказаться от кучи как поставщика узлов или список находится на внешнем носителе. Просто бросать освободившиеся узлы было бы слишком расточительно. Мы будем помещать их в стек свободного пространства, организованный как линейный односвязный список. Когда нам потребуется новый узел, то возьмем его из вершины стека. Операция вставки узла в некоторый список при наличии стека свободного пространства, иллюстрируется рис. 22.

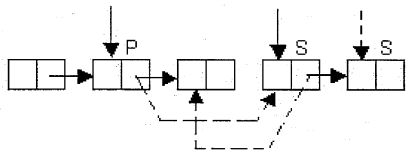


Рис. 22. Вставка узла в список.

На рисунке приняты обозначения:

– P –узел вслед за которым выполняется вставка

– S – указатель на вершину стека свободного пространства

Текст функции приведен ниже.

```
NODE *InsertNode(NODE *p, NODE *s){
// Вставка узла вслед за p
// s - указатель на вершину стека свободного пространства
// возвращает указатель на новый узел
```

```
NODE *X;
X=s;
s=s->Next;
X->Next=p->Next;
p->Next=X;
return X;
}
```

Операция удаления изображена на рис. 23. Удалённый узел помещается в вершину стека свободного пространства.

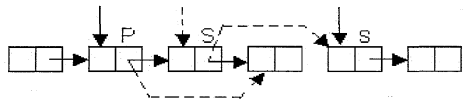


Рис. 23. Удаление узла из списка.

На рисунках, как и ранее, сплошные линии относятся к состоянию списков *до* операции, а пунктирные относятся к состоянию *после* операции.

Текст функции, выполняющей операцию удаления, приведен ниже.

```
void DeleteNode(NODE *p, NODE *s){
// узел, следующий за p удаляется и помещается
// в вершину стека свободного пространства s
NODE *X;
X=p->Next; // X-удаляемый узел
p->Next=X->Next;
X->Next=s;
s=X;
}
```

Для списочных структур на внешнем носителе роль указателя играет номер узла или смещение узла относительно начала файла в байтах.

5.6. Обслуживание свободного пространства

Для списков общего вида удаляемый из одного списка узел не может быть просто отправлен в стек свободного пространства, так как он может входить более чем в один список. Для обслуживания свободного пространства могут быть использованы методы *счетчиков ссылок* и *сбор мусора.*

5.6.1. Счетчик ссылок

В методе счетчиков ссылок каждый узел содержит поле счетчика int count, равное числу узлов, ссылающихся на данный. При удалении узла из списка поле count уменьшается на единицу. Когда счетчик окажется равным нулю, узел можно отправить в свободное пространство. Метод не работает для рекурсивных списков, в которых узел может прямо или косвенно ссылаться на самого себя. Такой узел никогда не будет

освобожден. Метод счетчиков ссылок употребляется, например, как встроенный механизм в Visual Basic 6.0.

5.6.2. Сбор мусора

При использовании метода "сбор мусора" программы беззаботно работают до тех пор, пока не будет исчерпано свободное пространство. Брошенные и нигде более не используемые узлы называют *мусором*. Когда свободное пространство исчерпано, вызывается *мусорщик*, целью которого является собрать мусор в стек свободного пространства. Каждый узел имеет бит маркировки. Метод работает в три фазы. В *первой фазе* обходится вся память, занятая списками и биты маркировки устанавливаются в "0". *Вторая фаза* систематически обходит все узлы списков и устанавливает в них бит маркировки в "1". Таким образом, маркированными нулём остаются только узлы, относящиеся к мусору. *Третья фаза* вновь обходит всю память и собирает узлы, помеченные нулем в стек свободного пространства.

Время работы мусорщика выражается формулой

$$T=C_1N+C_2M, \text{ где}$$

C_1, C_2 – константы,

N – число всех узлов в списках,

M – число всех узлов в памяти, отведенной под списки.

Таким образом, M-N – количество узлов, возвращаемых мусорщиком в стек свободного пространства. Время, расходуемое на возврат одного узла, составляет

$$\tau=(C_1N+C_2M)/(M-N)$$

Обозначим $\rho=N/M$ - коэффициент загрузки памяти, тогда

$$\tau=(C_1\rho+C_2)/(1-\rho)$$

Как видно из последней формулы, недостатком метода является то, что он медленно работает, когда загрузка памяти велика. В таких случаях число узлов, возвращаемых в стек свободного пространства не окупает затраченных усилий. Те программы, которым не хватает памяти, впустую расходуют массу времени, многократно и бесплодно вызывая мусорщика перед тем, как окончательно исчерпать память. Метод сбора мусора малопригоден для работы в реальном времени, так как он вызывается в непредвиденные моменты времени и может долго работать.

Контрольные вопросы

1. Чем отличается связанное распределение памяти под данные от последовательного?
2. Какие преимущества даёт представление строки линейным списком по сравнению с представлением массивом символов?
3. Какие операции могут быть выполнены над линейным списком?
4. Что такое голова списка и зачем она нужна?

5. Чем отличается кольцевой список от обычного и какие дополнительные возможности он имеет?
6. Какие дополнительные возможности предоставляет двусвязный список по сравнению с односвязным ?
7. В каких случаях целесообразно применять ортогональные списки вместо многомерных массивов?
8. Чем отличается список общего вида от обычного линейного списка?
9. Дайте определение понятия "топологическая сортировка"
10. Что такое стек свободного пространства?
11. Каковы недостатки метода счетчика ссылок при обслуживании свободного пространства для списочных структур?
12. Аналогичный вопрос для метода "сбора мусора"

6. МНОЖЕСТВА

Для представления конечных множеств могут быть использованы битовые строки, массивы, линейные списки или деревья.

При использовании битовых строк, каждое множество представляется последовательностью битов, длина которой равна числу элементов в универсальном множестве, то есть в множестве всех возможных элементов. Равенство единице j-го бита строки означает, что элемент с номером j входит в множество. Теоретико-множественные операции над множествами, представленными последовательностями битов, выполняются как побитовые булевы операции. Так, пересечению множеств соответствует конъюнкция, объединению – дизъюнкция, разности $x-y$, где x,y – множества, соответствует побитовая операция $x_i \wedge \overline{y_i}$.

Структура данных для множества, хранящегося в массиве, может быть следующей:

```
#define MAXSIZE 100
struct SETINARRAY {
      int nElem; // число элементов в множестве
      int Elem[MAXSIZE]; // элементы множества
};
```

Здесь предполагается, что элементы множества – целые числа, однако они могут быть любого типа. Элементы в множестве сортированы – это позволяет реализовать теоретико-множественные операции как однопроходные, что означает, что время выполнения операций пропорционально $n+m$, где n и m –количество элементов в множествах-операндах. Если же они не сортированы, время выполнения операций пропорционально $n \times m$. Два этих представления имеют существенные недостатки:

– в первом случае фиксируется объем универсального множества

– во втором – максимально возможное количество элементов в множестве.

Кроме того, если количество элементов в множестве в среднем невелико, то память используется нерационально. От этих недостатков свободно представление множества линейным списком. Ниже приведен текст операции «пересечение множеств». Другие операции пишутся аналогично. Множества представлены односвязными циклическими списками с головой. В приводимом примере в линейном списке элементами множеств являются числа, хранимые в сортированном порядке, что позволяет выполнять операции над множествами за линейное время. В поле данных головы хранится значение INT_MAX.

```
struct NODE{
     int El; // номер элемента множества
     NODE *Next;
};
/* прототипы функций*/
NODE *CrossSet(NODE *Head1, NODE *Head2);
NODE *CreateEmptySet();

//-----------------------------------
NODE *CreateEmptySet(){
// создание пустого множества
NODE *Head;
Head=new NODE;
Head->El=INT_MAX;
Head->Next=Head;
return Head;
}

//-----------------------------------
NODE *CrossSet(NODE *Head1, NODE *Head2){
// пересечение множеств с головами Head1,Head2
// возвращает указатель на голову результата
NODE *p,*q, *r /* указатель на хвост результата */,*n;

NODE *y=CreateEmptySet();
r=y;
for(p=Head1->Next,q=Head2->Next;  p!=Head1 && q!=Head2;){
     if(p->El>q->El){
          q=q->Next;
          continue;
     }
     if(p->El<q->El){
          p=p->Next;
          continue;
     }
     if(p->El==q->El){
```

```
            n=new NODE;
            n->Next=y;
            n->El=p->El;
            r->Next=n;
            r=n;
            p=p->Next;
            q=q->Next;
            continue;
        }
    } // for
    return y;
    }
```

Контрольные вопросы

1. Какие методы представления множеств вы знаете?
2. Почему целесообразно хранить элементы множества в массиве или в списке в сортированном порядке?
3. Дайте сравнительную оценку скорости выполнения теоретико-множественных операций для представления множества битовыми строками и списками.

7. ДЕРЕВЬЯ

Дерево есть конечное множество T узлов, такое, что:

1. имеется один специально обозначенный узел, называемый корнем дерева
2. остальные узлы содержатся в $m \geq 0$ попарно непересекающихся множествах $T_1, T_2, \ldots T_m$, каждое из которых является деревом

Деревья $T_1, T_2, \ldots T_m$ называются поддеревьями данного корня. Строго говоря, приведенное определение относится к специальному случаю дерева, называемому *корневым* деревом. Деревья, в которых корень не выделен, называют *висячими*. Здесь рассматриваются исключительно корневые деревья.

Число поддеревьев узла называют его степенью. Узел нулевой степени называют *листом*. Уровень узла в дереве определяется следующим образом:

1. корень имеет уровень 1
2. корни поддеревьев узла имею уровень на 1 больший, чем уровень узла.

Высотой дерева называют наибольший уровень узла в нём.

Если в п.2 определения дерева имеет значение относительный порядок следования поддеревьев $T_1, T_2, \ldots T_m$, то дерево называют упорядоченным. Лес – это множество, быть может, пустое, состоящее из некоторого числа непересекающихся деревьев. Определение дерева рекурсивно и также рекурсивными являются большинство методов обработки деревьев.

Дерево можно изобразить различными способами. Изображения деревьев на рис. 24 эквивалентны.

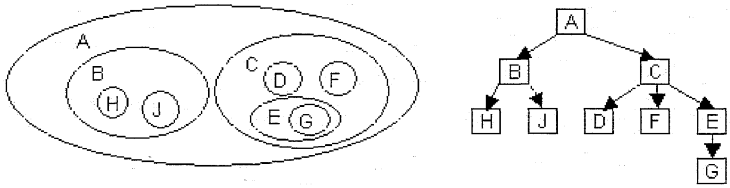


Рис.24. Различные изображения деревьев

Примерами древовидной структуры являются генеалогические деревья, оглавления, формулы, классификации. На рис.25 изображены деревья, изображающие формулу a-b*(c/d+e/f) и классификации товаров.

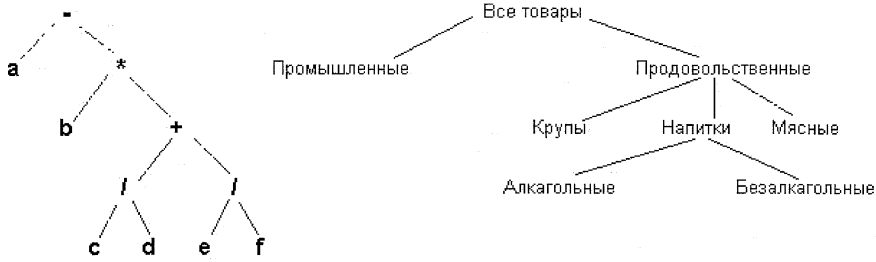
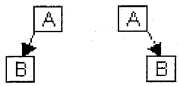


Рис. 25. Примеры деревьев

Когда говорят о деревьях, часто используют такие термины, как "отец", "сын", "брат", "предок", "потомок". Каждый узел является корнем своих поддеревьев. Последние являются братьями между собой и сыновьями своего отца. В упорядоченном дереве левый брат считается старшим. По аналогии можно ввести термины "дядя", "племянник" и другие. Термины "предок" и "потомок" употребляются для обозначения родства, простирающегося на несколько уровней дерева.

7.1. Бинарные деревья



Бинарное дерево определяется как конечное множество узлов, которое или пусто, или состоит из корня и двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями корня. Отметим, что деревья на рисунке слева различны, так как в одном случае пусто левое поддерево, а в другом правое.

Узел бинарного дерева может быть представлен структурой:

```
struct NODE{
    <тип> <поле данных>; // далее в примерах мы полагаем int x
    NODE *Left; // указатель на левого сына
    NODE *Right; // указатель на правого сына
```

```
};
```

7.2. Обход бинарного дерева

Для работы с древовидными структурами имеется множество алгоритмов, и многие из них используют одну и ту же идею, а именно идею прохождения или обхода дерева. Обход дерева подразумевает такой порядок работы с его узлами, для которого каждый из них посещается точно один раз. Для обхода бинарного дерева могут быть использованы три способа, определяемых рекурсивно.

Прямой обход.

1. Обработать корень
2. Обойти левое поддерево
3. Обойти правое поддерево

Обратный обход.

1. Обойти левое поддерево
2. Обработать корень
3. Обойти правое поддерево

Концевой обход.

1. Обойти левое поддерево
2. Обойти правое поддерево
3. Обработать корень

На рис. 26. изображены все варианты обходов бинарного дерева.

Рис. 26. Варианты обхода бинарного дерева.

На рис. 27. изображено бинарное дерево и порядок следования его узлов для различных методов обхода.

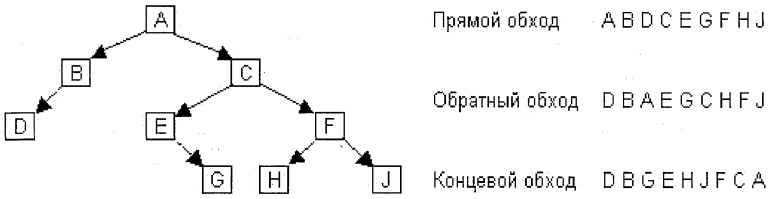


Рис. 27. Бинарное дерево и варианты его обхода

Ниже приводится текст рекурсивной функции, выполняющей обход дерева в прямом порядке.

```
void DirectBypass(NODE *Root){
// Root - указатель на корень дерева
if(Root==NULL) return; // дерево пусто
```

```
    Обработка(Root); // делаем то, что нужно с узлом
    DirectBypass(Root->Left); // пройти левое поддерево
    DirectBypass(Root->Right); // пройти левое поддерево
    }
```

Обратный и концевой обходы отличаются только местоположением оператора Обработка(Root) среди остальных.

Идея реализации алгоритма с "ручным" ведения стека (это может потребоваться, если язык не допускает рекурсии) заключается в том, что мы запоминаем в стеке историю спуска по левым ветвям дерева для того, чтобы впоследствии можно было вернуться и обработать оставшиеся узлы. Ниже приведен текст функции, выполняющей обход в обратном порядке.

```
#define MAXSTACK 50
void InverseBypass(NODE *Root){
// Нерекурсивный обход бинарного дерева
NODE *stack[MAXSTACK];
NODE *s;
int v=0; // указатель на вершину стека

s=Root;
again:
if(s!=NULL){
    // спускаемся по левым ветвям, запоминая историю спуска в
стеке
    stack[v++]=s;
    s=s->Left;
    goto again;
} else {
    if(v==0){ // стек пуст
        return;
    }
    // взять узел из стека
    s=stack[--v];
    Обработка(s);
    // переходим к правому поддереву
    s=s->Right;
    goto again;
}
}
```

Рассмотрим две конкретные задачи, решаемые с помощью обхода дерева.

Задача 1. Копирование бинарного дерева

Для решения задачи естественно использовать прямой обход с тем, чтобы узел – отец создавался раньше, чем сыновья.

```
NODE *CopyBinTree(NODE *Root){
// функция имеет указатель на корень
// дерева – оригинала в качестве аргумента
```

```
// и возвращает указатель на корень дерева – копии
if(Root==NULL) return NULL; // копия пустого дерева – пустое
дерево
// создадим корень в копии
NODE *RootCopy=new NODE;
RootCopy->x=Root->x; // копируем данные узла
// левый и правый сыновья в дереве – копии являются
// копиями левого и правого поддеревьев корня в оригинале
RootCopy->Left=CopyBinTree(Root->Left);
RootCopy->Right=CopyBinTree(Root->Right);
Return RootCopy;
}
```

Задача 2. Вычисление значения выражения, заданного деревом.

В качестве примера рассмотрим выражение ((2+3)*(7−4))/3. Порядок вычисления выражения можно изобразить в виде дерева (рис. 28.).

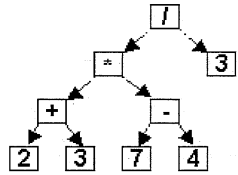


Рис.28. Выражение в дереве

Узел дерева в поле данных содержит либо число, либо символ операции. Если узел содержит число, то это операнд, а если операцию, то значения левого и правого поддеревьев суть её операнды. Вычисление естественно выполнять в порядке концевого обхода, поскольку для того, чтобы выполнить операцию, надо знать её операнды. Структура узла имеет вид:

```
#define OPERATION 0 // признак: узел содержит операцию
#define NUMBER 1    // признак: узел содержит число
struct UZEL{
  union {
       char Operation; // символ операции
       float Number;   // число
  };
  int Tag; // может принимать значения OPERATION или NUMBER
  UZEL *Left, *Right; // указатели на сыновей
};
```

Приведенная ниже функция вычисляет значение выражения, заданного деревом.

```
float TreeValue(UZEL *Root){
float Result;
if(Root->Tag==NUMBER) return Root->Number;
// в узле операция. Найдем её операнды
float x=TreeValue(Root->Left); // левый операнд
float y=TreeValue(Root->Right); // правый операнд
```

```
// выполним операцию
switch(Root->Operation){
      case '+':
            Result=x+y;
            break;
      case '-':
            Result=x-y;
            break;
      case '*':
            Result=x*y;
            break;
      case '/':
            Result=x/y;
            break;
}
return Result;
}
```

Голова дерева.

Дерево, как и линейный список, может иметь голову. В таких случаях, дерево, как правило, делают левым поддеревом головы. При обратном обходе, повторный выход на голову означает завершение алгоритма. Если дерево имеет голову, то каждый узел имеет отца, что позволяет избавиться от особенностей обработки корня, кроме тог, как и в случае линейных списков, наличие головы дерева позволяет избавиться от проблем, связанных различением пустого и несуществующего дерева.

7.3. Прошитые деревья

В бинарном дереве, содержащем N узлов, на каждый узел, кроме корня указывает ровно одна связь. Всего связей $2*N$; непустых – $N-1$, следовательно, $N+1$ связь пуста. Пустые связи существуют только для того, чтобы обозначить, что дальше в этом направлении пути нет, для чего достаточно одного бита. Возникает вопрос: а нельзя ли более рационально использовать пространство, занимаемое пустыми связями. *Прошитые деревья* используют место, занимаемое пустыми связями для хранения указателей, упрощающих прохождение дерева. Эти дополнительные связи называют *нитями*, откуда и появился термин *прошитые*. Введем обозначения:

1. *P – предшественник узла P в обратном порядке,
2. P* - последователь узла P в обратном порядке,
3. $^{+}$P – предшественник в прямом порядке,
4. P^{+} - последователь в прямом порядке.

Дерево может быть прошито для обхода в одном из порядков. Сопоставим обычное дерево и дерево, прошитое для обхода в обратном порядке. Вместо пустых левых связей будем хранить указатель на

предшественника в обратном порядке, вместо пустых правых связей – указатель на последователя. Эти связи будем называть "нитями" в отличие от основных связей, которые такие же, как в обычном дереве. Для того, чтобы отличать основные связи от нитей, в каждом узле заведем два поля L и R, которые будут иметь значения THREAD, если связь – нить и MAINLINK, если связь – основная (THREAD и MAINLINK – константы). Таким образом, структура узла прошитого дерева имеет вид:

```
#define THREAD 0
#define MAINLINK 1
struct NODE{
      <поля данных>;
      NODE *Left, *Right;
      BYTE L,R;
};
```

На рис.29. изображено прошитое дерево. Пунктиром изображены нити.

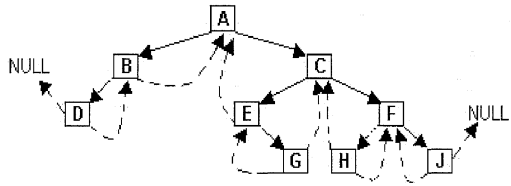


Рис.29. Прошитое дерево

Преимущество прошитых деревьев заключается в том, что упрощаются алгоритмы обхода. Ниже приведена функция, возвращающая указатель на последователя p в обратном порядке.

```
NODE *NextUzel(NODE *p){
NODE *q=p->Right;
if(p->R==THREAD) return q; // если это нить, то q-результат
// в противном случае спуститься до упора по левым связям
while(q->L==MAINLINK) q=q->Left;
return q;
}
```

При наличии алгоритма определения последователя отпадает необходимость в стеке (явном или порождаемом механизмом реализации рекурсии). Функция, выполняющая обход дерева в обратном порядке принимает вид:

```
void InverseBypass(NODE *Root){
NODE *q=Root;
// найдем первый в обратном порядке узел
// он находится в конце спуска по основным левым связям
while(q->L==MAINLINK)q=q->Left;
// проходим все узлы, используя функцию NextUzel
for(;q!=NULL;q=NextUzel(q)){
  Обработка(q);
```

```
}
}
```

7.4. Другие представления бинарных деревьев

Подходящий выбор представления дерева в первую очередь определяется видом операций, выполняемым над деревьями. В частности, можно использовать методы последовательного распределения памяти, отображающие связи на физическое размещение данных. Такой способ годится, когда требуется компактное представление дерева, и оно не будет подвергаться радикальным динамическим изменениям в процессе работы программы. Он заключается в том, что опускается поле Left или Right, а последовательное расположение узлов замещает опущенную связь. Структура узла имеет вид:

```
struct NODE{
      <поля данных>;
      NODE *Right;
      bool HaveLeftSon;
};
```

Хранится только правая связь. Левый сын узла, если он есть, расположен в памяти сразу за отцом. Поле HaveLeftSon имеет значение true, если узел имеет левого сына. Узлы в памяти хранятся в порядке прямого обхода. На рис. 30 изображено дерево и его представление в последовательной памяти. Угловая скобка справа внизу от узла означает отсутствие левого сына.

Возможно также последовательное размещение узлов дерева в концевом и обратном порядке.

Можно также предложить чисто последовательное размещение узлов дерева в памяти, когда сыновья узла с адресом *X* имеют адреса *2*X* и *2*X+1*. Память при этом может расходоваться крайне непроизводительно.

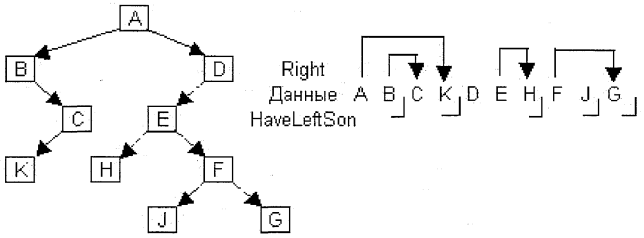


Рис. 30. Альтернативное представление бинарного дерева

7.5. Представление деревьев общего вида

Рассмотрим два варианта представления дерева общего вида. В первом варианте для хранения указателей на сыновей используется массив фиксированной длины:

```
#define MAXSON 10 // максимально возможное число сыновей
struct NODE {
      <поля данных>;
      int nSon; // действительное число сыновей узла
      NODE *Sons[MAXSON];
};
```

В некоторых узлах память оказывается недоиспользованной, а также возможна ситуация, когда число сыновей узла окажется больше максимально допустимого.

От этого недостатка свободно представление, когда указатели на сыновей находятся в линейном списке. Такая списочная структура содержит два типа узлов – узлы дерева и узлы линейного списка сыновей.

```
struct NODE { // узел дерева
      <поля данных>;
      SON *Sons; // указатель на начало списка сыновей
};

struct SON { // узел списка сыновей
      struct NODE *Son; // указатель на сына
      struct SON *Next; // указатель на следующий узел
                        // списка сыновей
};
```

На рис. 31 изображен пример такого дерева.

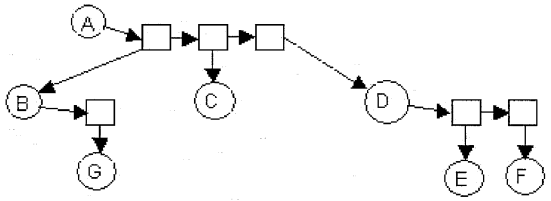


Рис.31. Дерево с линейным списком сыновей

7.5.1. Представление деревьев общего вида бинарными деревьями

Всякое дерево можно представить в виде бинарного дерева. При преобразовании сохраняется связь отца с самым левым (старшим) сыном. Они соединяются левой связью. Сыновья одного отца соединяются правыми связями. Рис. 32 поясняет преобразование.

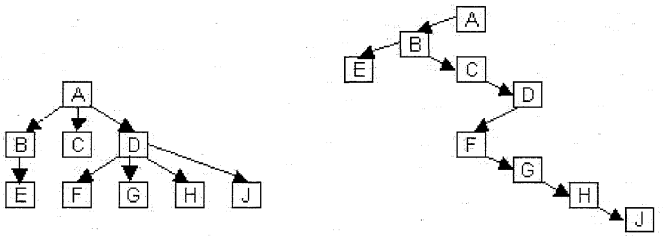


Рис. 32. Преобразование дерева общего вида к бинарному дереву

Отметим, что преобразование является взаимно однозначным.

Контрольные вопросы

1. Дайте определение дерева.
2. Дайте определение бинарного дерева.
3. Применительно к деревьям определите понятия: корень, лист, степень узла, отец, сын, предок, потомок, брат.
4. Определите прямой, обратный и концевой порядок обхода бинарного дерева.
5. Напишите функцию, вычисляющую высоту бинарного дерева.
6. Какие преимущества даёт прошивка бинарного дерева?
7. Какие способы представления деревьев общего вида вы знаете?

8. КОНЕЧНЫЙ АВТОМАТ

Конечным автоматом называется абстрактное устройство, которое в каждый момент времени находится в одном из конечного множества состояний. На вход автомата поступают символы из некоторого алфавита. Под воздействием символа, автомат переходит в следующее состояние и выполняет некоторую функцию. Выше дано нестрогое определение одной из разновидностей конечного автомата – автомата Мили. Автомат Мура здесь не рассматривается. Таким образом, автомат может быть задан таблицей переходов, одна строка которой определяется следующим образом:

```
struct PEREHOD {
    int State; // текущее состояние
    int Class; // класс входного символа
    int Next;  // следующее состояние
    void (*f)(); // указатель функции, вызываемой на переходе
};
```

Рассмотрим применение конечного автомата на примере сборки вещественной константы, заданной своим символьным представлением вида “–234.456E-12”. Эту задачу должен решать любой транслятор в фазе лексического анализа. Граф конечного автомата для решения этой задачи представлен на рис.33.

Вершины графа соответствуют состояниям, а ребра помечены классом возможного входного символа и функцией, выполняемой на переходе. Начальным является состояние 1.

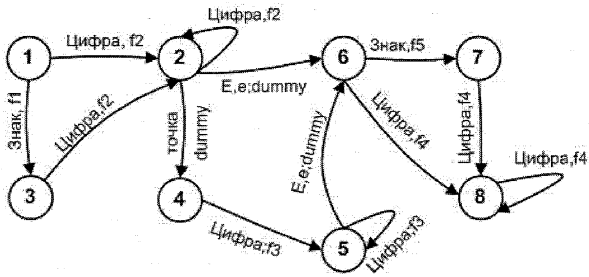


Рис. 33. Граф конечного автомата

Таблица, приведенная ниже, поясняет функционирование автомата.

Состояние	Входной символ	Следующее состояние	Пояснение
1	цифра	2	Цифра относится к целой части мантиссы
	знак	3	Это знак числа
2	цифра	2	Продолжают поступать цифры целой части мантиссы
	точка	4	Точка, отделяющая целую часть мантиссы от дробной
	E,e	6	Целая часть мантиссы закончилась, дробная часть отсутствует, но имеется порядок числа
3	цифра	2	Это цифра целой части мантиссы
4	цифра	5	Цифра дробной части мантиссы
5	цифра	5	Продолжают поступать цифры дробной части мантиссы
	E,e	6	Дробная часть мантиссы закончилась, начинается порядок
6	знак	7	Знак порядка
	цифра	8	Цифра порядка
7	цифра	8	Цифра порядка
8	цифра	8	Продолжают поступать цифры порядка

Текст программы приведён ниже.

```
// Алгоритм работает с объектами:
```

```
static int ZnakN; // знак числа
static double m;  // мантисса
static int nd;    // число цифр в дробной части
static int ZnakP; // знак порядка
static int p;     // порядок
static char z;    // очередной символ из входной строки
// Функции, выполняемые на переходах:
static void f1(){ZnakN=z=='+' ? 1:(-1);} // знак числа
static void f2(){m=10*m+z-'0';} // цифра целой части мантиссы
static void dummy(){} // пустышка
static void f3(){f2();nd++;} // цифра дробной части мантиссы
static void f5(){ZnakP=z=='+' ? 1:(-1);} // знак порядка
static void f4(){p=10*p+z-'0';} // цифра порядка
// Константы для определения классов символов
#define ZNAK=0  // знак + или -
#define DIGIT=1 // цифра
#define E=2     // символ e или E (признак порядка)
#define DOT=3   // точка, отделяющая дробную часть от целой
#define OTHER=4 // все прочие символы. Их появление - ошибка
// Таблица переходов автомата:
static PEREHOD tp[]={
     {1, ZNAK, 3, f1},
     {1, DIGIT, 2, f2},
     {2, DIGIT, 2, f2},
     {3, DIGIT, 2, f2},
     {2, DOT,   4, dummy},
     {2, E,     6, dummy},
     {4, DIGIT, 5, f3},
     {5, DIGIT, 5, f3},
     {5, E,     6, dummy},
     {6, ZNAK,  7, f5},
     {6, DIGIT, 8, f4},
     {7, DIGIT, 8, f4},
     {8, DIGIT, 8, f4}
};
// Функция, классифицирующая символы:
static int Classify(char z){
if(z=='+' || z=='-') return ZNAK;
if(z>='0' && z <='9') return DIGIT;
if(z=='E' || z=='e') return E;
if(z=='.')return DOT;return OTHER;
}
/* И, наконец, собственно функция преобразования строки в
вещественную константу */
double AutomatForDoubleConst(char *s, char *error){
// s - входная строка
// error - 0-успех, 1-не найдена строка в таблице переходов
// возвращает double значение преобразлванной строки
int State; // текущее состояние
```

```
int Class;
int i;
bool Found;
char *x;
*error=0;

// инициирование
ZnakN=1;m=0;nd=0;ZnakP=1;p=0;
State=1;

for(x=s;*x!='\0';x++){
    z=*x;
    Class=Classify(z); // найдем класс очередного символа
    Found=false;
    for(i=0;i<sizeof(tp)/sizeof(PEREHOD);i++){
        if(State==tp[i].State && Class==tp[i].Class){
            Found=true;
            break;
        }
    }
    if(!Found){
        *error=1;
        return 0;
    }
    (*tp[i].f)();
    State=tp[i].Next;
}
double r=ZnakN*m*pow(10,(ZnakP*p-nd));
return r;
}
```

Контрольные вопросы

1. Разработайте схему конечного автомата, преобразующего символьные строки вида "123", "-123", "-0xFF17", "0377" в целые числа в соответствии с правилами языка Си. Напишите функцию, реализующую преобразование.
2. Напишите функцию, удаляющую все пробелы, стоящие подряд между графическим символом и запятой из любого текста.

9. ТАБЛИЦЫ

В данном разделе рассмотрены плоские таблицы с фиксированным числом столбцов. Примером такой таблицы может служить телефонный справочник:

Улица	Дом	Кв.	Фамилия И.О.	Телефон
Цвиллинга	31	39	Петров П.П.	2634444
Советская	65	12	Сидоров С.С.	2651118

Таблицу идентификаторов строит любой компилятор:

Идентификатор	Тип	Длина	Адрес
Alpha	int	4	0xea45345
Beta	double	8	0xabc6543a

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Рис.34.Таблица умножения

Мы не рассматриваем таблиц с двумя входами типа таблицы умножения (рис.34.), в которых искомая величина лежит на перекрестии строки и столбца. Строку таблицы называют также записью или кортежем. Столбец также называют полем или атрибутом. Поле или группа полей, по которым выполняется поиск данных в таблице, называется ключом. Если в таблице не может существовать более одной записи с данным значением ключа, то такой ключ называют *первичным*. Другие ключи называют *непервичными* или *вторичными*. В телефонном справочнике роль первичного ключа может играть, например, номер телефона. Фамилия не может быть первичным ключом, так как возможно существование однофамильцев. Все операции в таблице задаются по отношению к ключу, а выполняются над всей записью. К типичным операциям над таблицей относятся:

- *включение*: дана пара ключ - данные. Требуется включить запись в таблицу так, что впоследствии ее можно было найти по ключу.
- *поиск*. Дан ключ, требуется найти запись.
- *модификация*: дан ключ, и новые значения изменяемых полей. Требуется найти запись и изменить данные.
- *удаление*. Дан ключ, требуется найти и удалить запись.

Как видно, во всех случаях, прежде чем выполнить операцию, требуется сначала найти запись. Это относится и к операции включения, так как для того, чтобы добавить запись в таблицу, требуется найти место вставки. Следует различать удачный и неудачный поиск (поиск того, чего в таблице нет), а также поиск единственной записи (по первичному ключу) или многих записей, удовлетворяющих критерию поиска. Эффективность этих операций, как правило, различается. Мы будем рассматривать четыре основных способа организации таблиц:

1. последовательные
2. сортированные
3. древовидные
4. рассеянные (они же рандомизированные или hash – таблицы)

9.1. Последовательные таблицы

Новые записи помещаются в конец таблицы в порядке поступления. Поиск осуществляется перебором записей. Пусть таблица содержит N

записей. Для успешного поиска единственной записи требуется просмотреть в среднем *N/2* записей; в случае неудачного поиска придется просмотреть всю таблицу, то есть *N* записей. Поиск по вторичному ключу требует просмотра *N* записей. В любом случае время поиска пропорционально размеру таблицы.

Для того, чтобы физически удалить запись из таблицы, требуется выполнить сдвиг на одну позицию вверх всех нижележащих записей, что требует значительного времени. Как правило, так не поступают. Вместо этого запись помечают как удаленную в специально отводимом для этого байте. Операция упаковки, то есть физического удаления помеченных на удаление записей, производится изредка. Такова, например, дисциплина работы с одним из старейших форматов файлов баз данных – dbf – форматом. Удаленные записи можно "сшить" в стек свободного пространства и использовать для вставки новых записей.

Последовательные таблицы есть смысл использовать, когда таблица мала или скорость поиска несущественна.

Контрольные вопросы

1. Напишите функцию, выполняющую физическое удаление записей из последовательной таблицы, помеченных символом '*' в 1-м байте.
2. Напишите функцию поиска записи по фамилии в телефонном справочнике, который хранится как последовательная таблица в бинарном файле.
3. Напишите функцию, помещающую новую запись в первую удалённую позицию последовательной таблицы, или в конец таблицы, если удалённые позиции отсутствуют.

9.2. Сортированные таблицы

В сортированных таблицах записи упорядочиваются по ключу, состоящему из одного или более полей записи. Для такой таблицы определяется правило сравнения ключей, позволяющее определить предшествование одного ключа другому.

9.2.1. Алгоритмы поиска в сортированной таблице

В обсуждаемых ниже алгоритмах для упрощения предполагается, что ключи – это целые числа, что никак не сказывается на сути дела. Отметим, что для любого метода организации таблицы решение задачи поиска может быть сформулировано как решение уравнения *Ключ=F(Адрес)* относительно адреса при заданном ключе.

9.2.1.1. Бинарный поиск (дихотомия)

Бинарный поиск в сортированной таблице – это разновидность метода *деления интервала пополам* для решения уравнения *f(x)=0*.

Действительно, если ключи интерпретировать как числа, то в сортированной таблице они следуют в неубывающем порядке, например, как это изображено на рис. 35.

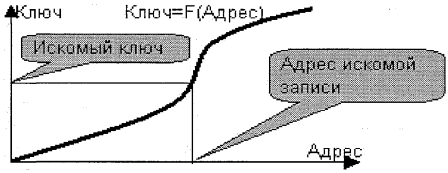


Рис.35. Постановка задачи поиска в сортированной таблице

Приведенный ниже алгоритм отыскивает индекс заданного целого числа в массиве сортированных целых чисел.

```
int BinSearch(int Key, int n, int t[]){
// int t[] - массив длиной n, в котором производится поиск
// ключа Key
// в случае успеха функция возвращает индекс найденного
// элемента, а в случае неудачи - тот индекс, на котором
// находился бы ключ, если бы он был в таблице
int i,j,k;
i=0;j=n-1;
while(j>i+1){
     k=(i+j)/2; // середина интервала
     if(t[k]==Key) return k;
     if(t[k]<Key){
          i=k;
     } else {
          j=k;
     }
}
if(t[i]<Key){
    k=j;
} else {
    k=i;
}
return k;
}
```

Очевидно, что число делений интервала пополам не превышает $\log_2 N$. Это и есть оценка быстродействия бинарного поиска. Функция `bsearch`, выполняющая бинарный поиск, входит состав библиотек, поставляемых практически с любым компилятором, и имеет прототип:

```
void *bsearch(const void *key, const void *base, size_t nelem,
size_t width, int (*fcmp)(const void *a, const void *b ));
```

Аргументы:

- key – адрес записи, имеющей структуру такую же, как и запись таблицы. В этой записи заполнены только поля, составляющие ключ.
- base – адрес начала таблицы

– nelem – число записей в таблице
– width – длина записи в байтах
– fcmp – функция, определяемая пользователем, которая сравнивает ключи записей, имеющих адреса a и b и возвращает значение:

< 0 при (*a<*b),
0 при (*a==*b)
>0 при (*a>*b).

Функция *bsearch* возвращает адрес найденной записи или *NULL*.

Для поиска в сортированной таблице могут быть использованы любые методы решения уравнения *Ключ=F(Адрес),* в частности метод золотого сечения и интерполяционные методы.

9.2.2. Вставка и удаление в сортированной таблице

При вставке новой записи в сортированной таблице, требуется найти место в которое её следует поместить (время $\sim\log_2 N$) и сдвинуть все записи от места вставки и ниже на одну позицию вниз.

Аналогично, при удалении требуется найти удаляемую запись и все записи ниже удаляемой сдвинуть на одну позицию вверх. Обе операции требуют физического перемещения в среднем *N/2* записей. Таким образом, при таком простодушном поведении, мы проигрываем все преимущества сортированной таблицы для операции поиска. Поэтому при удалении запись не удаляют физически, а лишь помечают как удаленную в специально отведенном байте. При вставке находят ту позицию, в которую следовало бы поместить новую запись, чтобы она не нарушала порядка, но не помещают ее туда, а помещают в линейный список (а можно и в дерево), начинающийся с позиции вставки, как на рис.35.

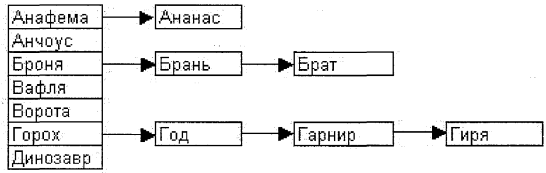


Рис.35. Сортированная таблица с цепочками переполнения

При поиске записи теперь следует найти позицию, в которой должна была бы находиться запись и, отправляясь от нее пройти линейный список, исходящий из этой позиции. Очевидно, что после большого количества вставок, поиск будет все больше походить на простой перебор записей. Рано или поздно, придется переписать таблицу как сортированную заново, таким образом, чтобы в ней не было удаленных позиций и цепочек переполнения. Отложенные операции обслуживания характерны для всех способов организации таблиц, кроме древовидных.

9.2.3. Оценка трудоемкости сортировки

Оценим трудоемкость процесса упорядочения массива из N ключей. В исходном состоянии эти ключи могут образовывать любую из $N!$ перестановок. Энтропия массива ключей, определяемая по Шеннону:

$$H = -\sum_{i=1}^{N!} p_i \log_2 p_i$$

где p_i – вероятность перестановки с номером i.

Наибольшей энтропией обладает система, для которой все состояния равновероятны: $H_{max}=\log_2 N!$. Для упорядоченной таблицы, все ключи которой различны, энтропия равна нулю. Изменение энтропии равно количеству информации, получаемой в процессе сортировки. Для случайного массива необходимо получить $\log_2 N!$ бит информации. При сравнении ключей K_i,K_j можем получить два исхода: $K_i<K_j$ *и* $K_i>K_j$. (случаем $K_i=K_j$ пренебрегаем, как маловероятным).

Если исходы равновероятны, то сравнение дает ровно 1 бит информации. Такие ключи называют статистически эквивалентными. Если исходы неравновероятны, то будет получено меньше одного бита информации. Кстати, именно из-за того, что некоторые сортировки сравнивают статистически неэквивалентные ключи, они имеют низкую эффективность.

Таким образом, число сравнений ключей при сортировке удовлетворяет неравенству

$$M \geq \log_2 N!$$

Для вычисления $N!$ воспользуемся формулой Стирлинга

$$N! = \sqrt{2\pi N}\left(\frac{N}{e}\right)^N$$

С учетом этого после преобразований получим

$$M \geq N(\log_2 N - 1.43)$$

Найденная оценка $N \log_2 N$ будет служить ориентиром при оценке эффективности различных методов сортировки.

9.2.4. Внутренняя сортировка

Различают внутреннюю и внешнюю сортировку. Под внутренней сортировкой понимают сортировку в условиях, когда вся таблица помещается в оперативную память. Внешняя сортировка – это сортировка данных на внешнем носителе, причем доступный объем оперативной памяти недостаточен для того, что считать сразу всю таблицу в оперативную память.

9.2.4.1. Сортировка подсчетом

Этот простой метод основан на том, что j-й ключ в сортированной последовательности превышает ровно $j-1$ ключ. В первой фазе алгоритм

подсчитывает для каждого ключа число ключей, которые меньше его. Значение счетчика для ключа и есть тот номер позиции в сортированной таблице, который он должен занять.

```
void CountSort(int n, int t[]){
// t - сортируемый массив целых чисел длиной n
int i,j;
int *c=new int[n]; // массив счетчиков
// обнулим счетчики
for(i=0; i<n; i++) c[i]=0;
// фаза подсчета
for(i=0; i<n; i+){
     for(j=0; j<i; j++){
          if(t[i]>t[j]){
               c[i]++;
          } else {
               c[j]++;
          }
     }
}
// фаза расстановки
for(i=0; i<n; i++){
while(i!=c[i]){ // до тех пор, пока t[i] не займет
     // окончательного положения
          swap(t[i],t[c[i]]); // обмен местами эл-тов таблицы
          swap(c[i],c[c[i]]); // обмен местами счетчиков
     }
}
}
```

Ключ t_i в исходной последовательности сравнивается с i предшествующими ключами и, следовательно, общее число сравнений равно

$$\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$$

то есть пропорционально N^2.

9.2.4.2. Сортировка простым выбором

Просматривая исходную последовательность ключей t, начиная с t_0, находим среди них наименьший и меняем местами с t_0, затем повторяем процесс начиная с t_1, находим наименьший и меняем местами с t_1, и так далее до конца таблицы. Ниже представлен текст функции, выполняющей сортировку простым выбором.

```
void SimpleChoice(int n, int t[]){
int i,j,k;
for(i=0; i<n; i++){
     k=i;
     // к моменту завершения цикла по j, k будет индексом
     // наименьшего ключа на отрезке i…N-1
     for(j=i+1;j<n;j++){
```

```
            if(t[j]<t[k]){
                k=j;
            }
        }
        swap(t[i], t[k]);
    }
    }
```

В этом случае время также пропорционально N^2, так как при поиске наименьшего ключа на отрезке *i...N-1*, потребуется выполнить *N-i-1* сравнений. Суммируя по *i*, получим величину, пропорциональную N^2.

9.2.4.3. Квадратичный выбор

Быстродействие сортировки выбором можно улучшить, если сделать процесс выбора многоступенчатым. Пусть размер таблицы N=16. Разобьем таблицу на 4 группы по 4 элемента в каждой, как на рис. 36.

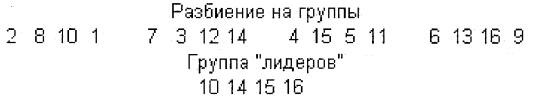


Рис. 36. Квадратичный выбор

Определим наибольший элемент в каждой группе и поместим его в отдельную группу "лидеров". Наибольший среди них и есть наибольший в таблице. Поместим его в область вывода. Последующие элементы отыскиваются так: среди элементов группы, из которой происходил выбранный на предыдущем шаге элемент, выбираем наибольший и помещаем вместо выбывшего в группу лидеров. Затем очередной элемент выводится из группы лидеров. Если N – точный квадрат, то можно разделить таблицу на $\sqrt{N}$ групп по $\sqrt{N}$ элементов. Любой выбор, кроме первого потребует не более $\sqrt{N}-1$ сравнений внутри группы ранее выбранного элемента плюс $\sqrt{N}-1$ сравнений внутри группы лидеров. Таким образом, время такой сортировки пропорционально $N\sqrt{N}$, что гораздо лучше N^2. Эту идею можно обобщить, получив метод кубического выбора, выбора четвертой степени и т.д.

Кубический выбор состоит в том, чтобы разделить таблицу на $\sqrt[3]{N}$ больших групп, каждая из которых состоит из $\sqrt[3]{N}$ малых групп по $\sqrt[3]{N}$ элементов. Время такой сортировки пропорционально $N\sqrt[3]{N}$. Если развить эту идею до ее полного завершения, когда в группе на каждом уровне содержится только два элемента, то придем к методу, основанному на выборе из бинарного дерева.

9.2.4.4. Выбор из дерева

На рис.37. изображена схема "турнира с выбыванием" для ключей.

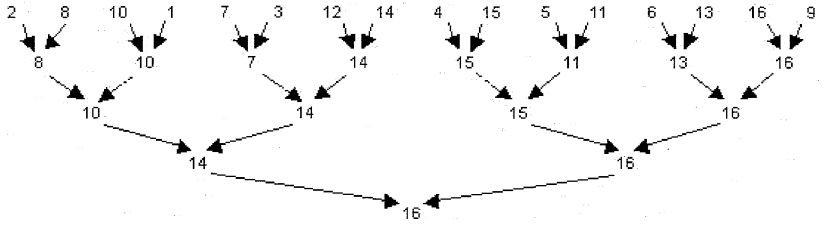


Рис.37. Турнир с выбыванием

В корень попадает наибольший ключ. Чтобы выявить второй по величине ключ, нужно выведенный ключ заменить на -∞, тогда в корень попадает второй по величине ключ. Для этого следует изменить лишь один путь в дереве, для чего потребуется не более $\log_2 N + 1$ сравнений. Таким образом, время сортировки пропорционально $N \log_2 N$.

9.2.4.5. Сортировка простыми вставками

Допустим, что ключи $t_{0...}t_{k-1}$ уже упорядочены. В упорядоченной части последовательности найдем место для t_k, на котором ключ t_k не нарушает порядка, и вставим его туда, предварительно сдвинув вправо на одну позицию ключи от места вставки до t_k. Также поступим с последующими ключами.

```
void  InsertSort(int n, int t[]){
int i,j,k,Key;

for(k=0; k<n; k++){
     Key=t[k]; // Key - очередной вставляемый ключ
     // от k-1 ключа двигаемся влево в поисках первого
     // ключа, который меньше Key
     // совмещаем поиск места вставки со сдвигом ключей вправо
for(j=k-1; j>=0; j--){
     if(t[j]<Key){
          break; // нашли
     }
     t[j+1]=t[j];
}
// новый ключ должен быть установлен в позицию t[j+1]
t[j+1]=Key;
}
}
```

При поиске места вставки *k*-го ключа необходимо пройти в среднем *k/2* ключей, выполняя сравнения и сдвиги. Таким образом, порядок времени работы алгоритма опять составляет величину порядка N^2. Время поиска места вставки можно уменьшить до ~$\log_2 N$, используя дихотомию, однако количество сдвигов от этого не уменьшится.

9.2.4.6. Сортировка методом "пузырька"

Проходим массив ключей слева направо, сравнивая соседние ключи. Если они стоят не по порядку, меняем их местами. Если в процессе просмотра не было обменов местами, то работа закончена, в противном случае повторяем процесс.

```
void BubbleSort(int n, int t[]){
bool b; // признак того, что в просмотре были обмены
do {
     b=false;
     for(int k=0; k<n-1; k++){
          if(t[k]>t[k+1]){
               // обмен местами t[k] и t[k+1]
               swap(t[k],t[k+1]);
               b=true;
          }
     }
} while(b);
}
```

Метод "пузырька" носит такое название, потому что в результате одного просмотра таблицы, запись с наибольшим значением ключа перемещается вверх (если считать, что в верхней части таблицы располагаются наибольшие значения ключей), то есть всплывает как пузырек воздуха в жидкости. Время работы алгоритма пропорционально N^2. Анализ алгоритма довольно сложен и здесь не приводится.

Быстродействие можно несколько улучшить, если просматривать массив не далее места последнего обмена местами в предыдущем просмотре, поскольку все ключи правее места последнего обмена уже стоят на своих окончательных местах ("всплыли"). Еще одно усовершенствование – просматривать массив ключей поочередно слева направо, а затем справа налево. Такой метод называется *шейкер – сортировкой*. Предложенные усовершенствования улучшают быстродействие, однако время работы остается пропорциональным N^2.

9.2.4.7. Сортировка слиянием

Слияние означает объединение двух или более упорядоченных таблиц в одну. Например, можно слить две таблицы с ключами:

3 5 6
2 4 7

Будем каждый раз переносить в область вывода меньший из двух наименьших элементов (рис. 38).

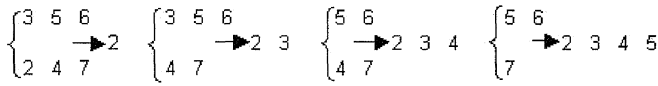


Рис. 38. Процесс слияния двух последовательностей

Для того, чтобы избавиться от обработки ситуации когда одна из последовательность закончилась, а другая нет, добавим в конец каждой из них искусственных "стражей" - ключи, равные ∞. Ниже представлен текст функции, выполняющей слияние двух сортированных целых массивов в один сортированный массив.

```
void Merge(int n, int t[], int m, int v[], int r[]){
// функция выполняет слияние массива t длиной n и
// массива v длиной m. Результат помещается в массив r
int i=0,j=0,k=0; // индексы для t,v,r
while(i<n && j<m){
     if(t[i]<v[j]){
          r[k++]=t[i++];
     } else {
          r[k++]=v[j++];
     }
}
}
```

Простейшая форма сортировки слиянием начинает с подмножеств, состоящих из единичных элементов, и объединяет их попарно в *~N/2* сортированных подмножеств, содержащих по два элемента. Затем эти подмножества попарно сливаются, и число их уменьшается вдвое и т.д., пока таблица не будет отсортирована полностью.

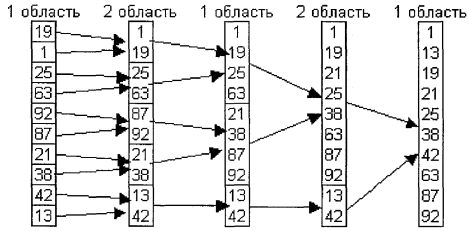


Рис.39. Сортировка слиянием

Рис. 39 поясняет этот процесс. Данные поочередно переписываются из одной области памяти в другую. Все слияния, выполняемые при переходе из одной области в другую, требуют не более N сравнений и необходимо $\sim\log_2 N$ таких переходов, следовательно, время сортировки пропорционально $N \log_2 N$.

9.2.4.8. Сортировка фон Неймана

Метод фон Неймана (1945г.) использует тот факт, что некоторые группы элементов уже упорядочены в исходной последовательности. Используются две области памяти, назовем их A и B. В исходном состоянии таблица находится в области A. Сначала выполняем слияние

отрезков упорядоченности, начинающихся на разных концах таблицы (рис. 40).

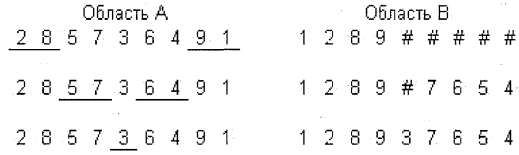


Рис. 40. Сортировка фон Неймана

Результат слияния помещается в начало области *B*. Затем от достигнутых позиций продолжаем движение по области *A* (вторая строка рис. 40), помещая результат, начиная с последней позиции в области *B*. Следующий процесс слияния вновь помещает результат в левую часть области *B*, начиная от уже заполненных позиций. Таким образом, двигаемся по области *A* слева направо и справа налево, помещая результаты слияния поочередно в левую и правую часть области B до тех пор, пока перенос всех данных из области *A* в область *B* не будет завершен. В результате такого переноса число отрезков упорядоченности уменьшится вдвое. Затем тот же процесс повторяется для переноса из области *B* в область *A*. Алгоритм заканчивает свою работу, когда в результате очередного переноса из области в область, будет получен единственный отрезок. Текст алгоритма приведен ниже.

```
void Join(int A[], int B[], int *left, int *right,
int *kl, int *kr, int Step){
// Выполняется слияние отрезков массива ключей A:
// левого, начиная с *left и правого, начиная с *right
// ( по ходу дела *left, *right изменяются)
// результат слияния помещается в массив B, начиная с
// *kl или *kr, которые изменяются с шагом Step. Step=1,
// если результат слияния помещается в b слева направо и
// Step=-1, если справа налево
bool l=true,r=true; // признаки того, что участок
// упорядоченности (left,right) еще не кончился
int v;
while(l || r){
    // найдем v - меньший ключ в сливаемых отрезках
    if(l && (!r || A[*left] <= A[*right])){
        v=A[(*left)++];
        l=(*left<*right && A[*left] >= A[*left-1]);
    } else {
        v=A[(*right)--];
        r=(*right>*left && A[*right] >= A[*right+1]);
    }
    if(Step==1){
        B[(*kl)++]=v;
    } else {
```

```
        B[(*kr)--]=v;
    }
}
}

//-----------------------------------------------
int Prohod(int A[], int B[], int N){
// функция выполняет перекачку из области A в область B
// и возвращает число отрезков упорядоченности в области B
int left,right,kl,kr;
int Count,Step;

left=0; right=N-1; kl=0; kr=N-1; step=1;Count=0;

while(left<right){
    Join(A,B,&left,&right,&kl,&kr,Step);
    if(right==left){
        B[kl]=A[left];
        Count++;
    }
    Count++;
    Step=-Step;
}
return Count;
}

//-----------------------------------------------
void Neumann(int N, int A[]){
// собственно сортировка массива A длиной N
int *B;
bool flag=false; // если flag=false, то выполняем
                 // перекачку из A в B иначе из B в A
int *from,*to;

B=new int[N]; // выделяем память для вспомогательной области
do {
    if(flag){
        from=B;
        to=A;
    } else {
        from=A;
        to=B;
    }
    flag=!flag;
} while(Prohod(from,to,N) > 1);
if(flag){
    memcpy(A,B,N*sizeof(int)); /*если результат получен в области
B, копируем в область A */
}
```

```
delete [] B; // освобождаем память области B
}
```

Время работы сортировки пропорционально $N \log_2 N$, так как потребуется не более $\log_2 N$ переходов из области в область, и при каждом переходе проходятся все данные.

9.2.4.9. Сортировка Хоара (1962 г.)

Это, по-видимому, наиболее популярный в мире метод внутренней сортировки. Установим указатели i и j на начало и конец таблицы. После сравнения ключей t_i, t_j изменяется один из индексов – i или j (увеличивается i или уменьшается j). Первым изменяется j. Если $t_i > t_j$, то ключи меняются местами и переключаемся на изменение противоположного указателя. Процесс продолжается до тех пор, пока указатели не "встретятся". Пример выполнения процесса изображен на рис.41.

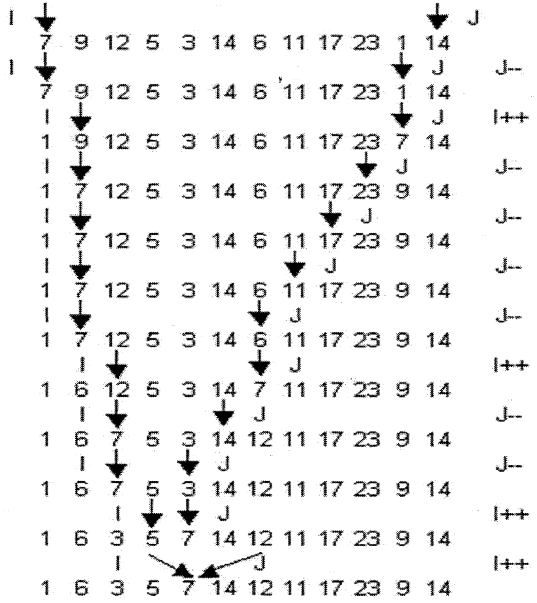


Рис. 41. Сортировка Хоара

Заметим, что ключ 7 участвовал во всех сравнениях. Все ключи слева от него меньше его, все ключи справа – больше его, а сам ключ 7 занял свое окончательное положение. Рассмотренный процесс назовем *разделением*. Теперь можно применить тот же самый процесс к левой и правой части разделенной таблицы.

```
int Partition(int m, int n, int t[]){
// функция выполняет разделение участка массива ключей
// от m до n, возвращает точку расщепления
bool b=true; // если b==true, то перемещается правый указатель
```

```
                // в противном случае - левый
    while(m<n){
        if(t[m]>t[n]){
            swap(t[m],t[n]);
            b=!b;   //   переключаемся   на   изменение   другого
    указателя
        }
        if(b){
            n--;
        } else {
            m++;
        }
    }
    return m;
    }
    //-----------------------------------------------------
    void Hoar(int m, int n, int t[]){
    // функция сортирует отрезок таблицы t на участке m…n
    if(m>=n) return;
    int k=Partition(m,n,t);
    Hoar(m,k-1,t);
    Hoar(k+1,n,t);
    }
```

Оценим быстродействие алгоритма в наилучшем и наихудшем случае. В лучшем случае каждый отрезок разделяется точно посередине и всего разделений $\log_2 N$. В каждом отрезке проходятся все элементы, то есть для всех отрезков – N элементов. Таким образом, время оказывается пропорциональным $N\log_2 N$. В худшем случае массив ключей уже упорядочен. Каждое разделение создает два отрезка, в один из них входит 1 ключ, в другой – все остальные, следовательно, число разделений равно N и время сортировки пропорционально N^2. Ситуацию можно поправить выбором разделяющего ключа одним из двух способов:

1. выбираем ключ со случайным номером на отрезке $m...n$
2. выбираем медиану из трех ключей t_m, $t_{(m+n)/2}$, t_n.

Сортировка Хоара включена в библиотеки практически для всех компиляторов. Ниже приведён прототип функции *qsort*, которая ее реализует.

```
void qsort(void *base, size_t nelem, size_t width,
  int (*fcmp)(const void *, const void *));
```

Она имеет те же параметры, что и ранее описанная функция бинарного поиска (`bsearch`).

9.2.4.10. Двоичная поразрядная сортировка

Алгоритм рассматривает ключи как последовательности битов. Таблица сортируется сначала по старшему значащему биту так, чтобы все ключи со старшим битом '0' предшествовали ключам со старшим битом '1'. Затем та же процедура выполняется для второго бита и полученных двух частей таблицы.

```
int BitPart(unsigned int Left, unsigned int Right,
  unsigned int Tab[], unsigned int Bit){
// разделение отрезка таблицы Tab от Left до Right
// по биту Bit.
// В двоичном представлении переменной Bit только один
// бит равен единице, все остальные - нули
int i,j,k;
bool l,r; // l=true, если бит Bit в ключе = 1
          // r=true, если бит Bit в ключе = 0
unsigned int x;

i=Left;
j=Right;
k=-1;
while(j>=i){
    // перемещая i слева направо, находим ключ с битом Bit = 1
    // перемещая j справа налево, находим ключ с битом Bit = 0
    l=((Tab[i] & Bit) != 0); // l=true, если бит Bit в i-м
ключе = 1
    r=((Tab[j] & Bit) == 0); // r=true, если бит Bit в l-м
ключе = 0
    if(!l)  k=i++;
    if(!r)  j--;
    if(l && r ){
        // бит в ключе слева равен 1
        // бит в ключе справа равен 0
        // меняем их местами
        swap(Tab[i],Tab[j]);
        k=i; // k - индекс самого правого ключа с битом 0
        i++;
        j--;
    }
}
if(i>Right){
    return Right;
} else {
    if(j<Left){
        return Left-1;
    } else {
        return k;
    }
```

```
}
}

//-----------------------------------------------------
void BitSort(unsigned int Left, unsigned int Right,
  unsigned int Tab[],unsigned int Bit){
// сортировка таблицы Tab на отрезке Left - Right
// по битам, начиная с бита Bit
int k;
if(Left>=Right) return;
k=BitPart(Left,Right,Tab,Bit);
if((Bit>>1)!=0){
    BitSort(Left,k,Tab,Bit>>1);
    BitSort(k+1,Right,Tab,Bit>>1);
}
}

//-----------------------------------------------------
void BinarySort(int n,unsigned Tab[], unsigned int keylen){
// собственно сортировка массива ключей Tab длиной n
// и числом бит в ключе keylen.
// Функция является прокладкой к BitSort,
// скрывающая непонятные пользователю аргументы BitSort
unsigned Bit;
Bit=(1<<(keylen-1));
BitSort(0,n-1,Tab,Bit);
}
```

Рассмотренная сортировка так же, как и следующая, не основана на сравнении ключей. Приблизительную оценку времени работы можно получить из следующих соображений. Пусть длина ключа M битов. Если таблица содержит все возможные ключи, которые можно составить из M битов, то число ключей $N=2^M$ или $M=\log_2 N$. Таким образом, для сортировки нужно пройти все данные столько раз, сколько битов в ключе и время работы пропорционально $N \log_2 N$.

9.2.4.11. Цифровая поразрядная сортировка

Цифровая поразрядная сортировка рассматривает ключ как последовательность цифр в некоторой системе счисления с основанием P. Подсчитаем, сколько имеется в таблице ключей с младшей цифрой d ($d=0...P-1$). Для этого потребуется P счетчиков C_0, C_1,... C_{P-1} и дополнительная область памяти для вывода в нее записей или указателей на записи. После подсчета ясно, что все ключи с младшей цифрой 0 должны размещаться с позиции 0, ключи с цифрой 1 – с позиции C_0, с цифрой 2 – с позиции C_0+C_1 и так далее. Затем та же самая процедура выполняется для последующих цифр. Рис. 42 поясняет процесс сортировки.

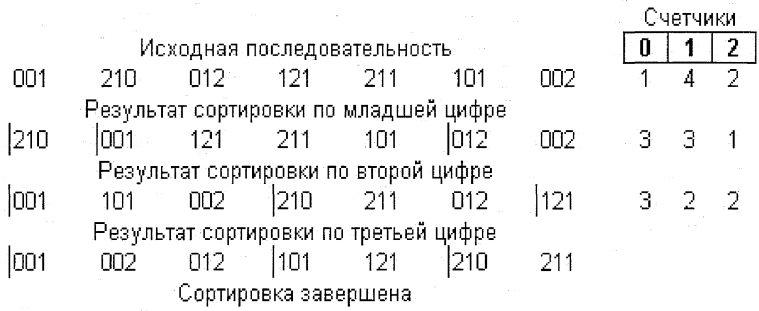


Рис.42. Цифровая поразрядная сортировка

Сравнение ключей не производится. Фактически каждый просмотр состоит из 3-х фаз – подсчет, распределение памяти, перемещение. Две из них можно совместить – накапливать значения счетчиков для *k+1* просмотра одновременно с перемещением *k*-го просмотра. В качестве системы счисления естественно выбрать 16-ричную (байт ключа содержит 2 цифры) или систему счисления с основанием 256 (1 байт ключа – 1 цифра). Поскольку алгоритм допускает распараллеливание, он весьма эффективен для многопроцессорных компьютеров. Ниже приведен текст функции, реализующей метод.

```
#define NDIGIT 256 // основание системы счисления
//-------------------------------------------
void DigitalSort(BYTE *t, int N, int KeyLen){
// цифровая поразрядная сортировка
// (основание системы счисления=256)
// t - сортируемый массив ключей
// N - число ключей в массиве
// KeyLen - число цифр в сортируемом ключе
int *Count;
int i,j,k;
BYTE Dig;
BYTE *b; // буферная область
int *Pos; // позиции расстановки

Count=new int[NDIGIT]; // память для счетчиков
Pos=new int[NDIGIT];   // текущие позиции при расстановке
b=new BYTE [N*KeyLen]; // буферная область

for(i=0; i<KeyLen; i++){
    for(k=0; k<NDIGIT; k++)Count[k]=0; // обнулим счетчики
    for(j=0; j<N; j++){ // подсчет
        // Dig - i-я цифра в j-м ключе
        Dig=*(t+j*KeyLen+KeyLen-i-1);
        Count[Dig]++;
    }
    // расчет позиций
```

```
        Pos[0]=0;
        for(j=1; j<NDIGIT; j++){
            Pos[j]=Pos[j-1]+Count[j-1];
        }
        // расстановка
        for(j=0; j<N; j++){
            Dig=*(t+j*KeyLen+KeyLen-i-1);
            memcpy(b+KeyLen*Pos[Dig]++, t+j*KeyLen, KeyLen);
        }
        // копируем то, что получилось в исходную область
        // хотя можно было бы перекачивать туда-обратно
        memcpy(t,b,KeyLen*N);
    }
    delete [] b;
    delete [] Count;
    delete [] Pos;
    }
```

9.2.1. Внешняя сортировка

Рассмотренные методы сортировки были рассчитаны на то, что таблица целиком помещается в оперативной памяти и в любой момент открыт доступ к любому элементу таблицы. Внешняя сортировка отличается тем, что время доступа к данным на внешних носителях чрезвычайно велико по сравнению с временем доступа к оперативной памяти.

Предположим, что нужно отсортировать таблицу из 5000 R_1-R_{5000} записей, а в оперативную память помещается не более 1000 записей. Решение может быть следующим – начать с внутренней сортировки каждого из 5 отрезков по 1000 записей, а затем слить полученные сортированные отрезки. Такой процесс – внутренняя сортировка с последующим внешним слиянием – является основой многих методов внешней сортировки.

В качестве введения рассмотрим метод, который можно назвать *сбалансированным двухпутевым* слиянием. Метод использует 4 файла. В первой фазе упорядоченные отрезки, получаемые внутренней сортировкой, помещаются поочередно в файлы 1 и 2 до тех пор, пока не исчерпаются входные данные. Начальная фаза распределения отрезков размещает пять отрезков следующим образом:

Файл 1: $R_1 - R_{1000}$; $R_{2001} - R_{3000}$; $R_{4001} - R_{5000}$

Файл 2: $R_{1001} - R_{2000}$; $R_{3001} - R_{4000}$

Файл 3: *пусто*

Файл 4: *пусто*

Затем файлы 1 и 2 вновь позиционируем на начало и сливаем отрезки с этих файлов, получая отрезки вдвое более длинные, чем исходные. Эти отрезки попеременно записываются в файлы 3,4. Отрезок $R_{4001} - R_{5000}$ в файле 1 не имеет пары для слияния в файле 2. Неявно добавим для

создания такой пары в файл 2 фиктивный отрезок длины 0. После первого прохода слияния получим:

Файл 1: *пусто*

Файл 2: *пусто*

Файл 3: $R_1 \_ R_{2000}$; $R_{4001} \_ R_{5000}$

Файл 4: $R_{2001} \_ R_{4000}$

Применив тот же алгоритм к файлам 3,4, в файлах 1,2 получим результат:

Файл 1: $R_1 \_ R_{4000}$

Файл 2: $R_{4001} \_ R_{5000}$

Наконец, после последнего прохода в файле 3 получим $R_1 - R_{5000}$, и сортировка завершена.

Аналогично можно рассмотреть 3-х и более – путевое слияние. Пусть для сортировки выделено N файлов. Разделим их на 2 части – в одной M файлов, в другой – N-M файлов. Распределим исходные отрезки по возможности более равномерно по M файлам первой части и выполним M-путевое слияние, помещая результат слияния поочередно в оставшиеся N-M файлов. Затем выполним N-M – путевое слияние, помещая результаты слияния в файлы первой части.

9.2.1.1. Многопутевое слияние и выбор с замещением

Рассмотрим процесс порождения начальных отрезков, которые затем подвергаются слиянию. Пусть имеется P возрастающих отрезков. Очевидным способом их слияния является следующий: из первых ключей каждого отрезка выбрать минимальный, соответствующую запись передать на выход и исключить из входных данных, затем этот процесс повторяется. Если P велико, то целесообразно использовать дерево выбора из P элементов. Пример 4-х путевого слияния изображен на рис. 43.

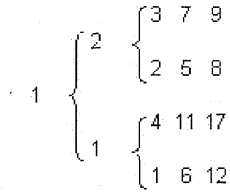


Рис.43. 4х-путевое слияние

Техника выбора с замещением может использоваться на первой фазе внешней сортировки. В этом случае P выбирается достаточно большим. Каждая запись при выводе замещается очередной записью из исходных данных. Если у новой записи ключ меньше, чем у только что выведенной, то она помечается как не участвующая в дальнейшем выборе, а значение P уменьшается на единицу и процесс продолжается.

Пример. Пусть исходная последовательность ключей: 2 7 12 3 8 9 4 1 5 11 10 6 и P=4

Процесс вывода начальных сортированных отрезков иллюстрируется таблицей:

Содержимое памяти				Вывод
2	7	12	3	2
8	7	12	3	3
8	7	12	9	7
(4)	8	12	9	8
(4)	(1)	12	9	9
(4)	(1)	12	(5)	12
(4)	(1)	(11)	(5)	Конец отрезка
4	1	11	5	1
4	10	11	5	4
6	10	11	5	…

В скобки заключены ключи, не участвующие в дальнейшем выборе. Таким путем удается получить упорядоченные последовательности длиннее P, что важно, так как время внешней сортировки слиянием в значительной мере зависит от числа начальных отрезков. Э.Ф.Мур предложил изящный способ доказательства того, что средняя длина порождаемого отрезка равна $2P$, проведя аналогию со снегоочистителем, движущимся по кругу. Пусть на кольцевую дорогу непрерывно падают снежинки (записи). Снежинки исчезают из системы (записи выводятся), когда они сбрасываются за пределы дороги. Точки дороги обозначаются вещественными числами x ($0 \le x \le 1$). Снежинка, падающая в точку x, соответствует входной записи с ключом x, а снегоочиститель имитирует процесс вывода. В установившемся режиме общее число снежинок на дороге в точности равно P. Каждый раз, когда снегоочиститель проходит точку 0, заканчивается формирование нового отрезка. На рис. 44 изображено поперечное сечение дороги, когда система находится в установившемся режиме.

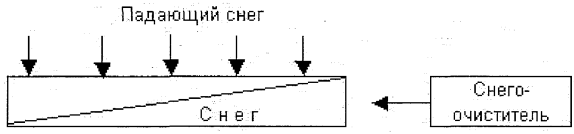


Рис. 44. Аналогия со снегоочистителем

Из рисунка видно, что количество снега, удаляемого за один оборот, вдвое превосходит количество снега, присутствующего на дороге в любой момент.

9.2.1.2. Многофазное слияние

Рассмотрим различные методы распределения отрезков по файлам. Предположим сначала, что у нас имеется 3 файла: *F1, F2, F3*. Можно воспользоваться сбалансированным слиянием:

1. Распределить начальные отрезки на *F1* и *F2*.
2. Слить отрезки с *F1*, *F2* на *F3* и остановиться, если *F3* содержит только 1 отрезок
3. Скопировать с *F3* отрезки попеременно на *F1, F2* и вернуться к шагу 2

Если начальное распределение дало S отрезков, то первый проход слияния даст S/2 отрезков на *F3*, второй – S/4 и т.д. Если, например, $17 \le S \le 32$, то произойдет:

- 1 проход распределения,
- 5 проходов слияния,
- 4 прохода копирования,

всего $2\log_2 S$ проходов по данным.

Можно обойтись половиной копирований, если использовать двухфазную процедуру:

1. Распределить начальные отрезки попеременно на *F1, F2*.
2. Слить отрезки с *F1, F2* на *F3* и остановиться, если *F3* содержит один отрезок
3. Скопировать половину отрезков с *F3* на *F1*
4. Слить отрезки с *F1*, *F3* на *F2*, остановиться, если получен единственный отрезок
5. Скопировать половину отрезков с *F2* на *F1* и вернуться к шагу 2

Заметим, что много времени тратится на копирование, которое не улучшает структуру данных.

9.2.1.3. Фибоначчиево слияние

В действительности, можно полностью устранить копирование, если начать с f_n отрезков в файле *F1* и f_{n-1} отрезков в *F2,* где f_n и f_{n-1} – последовательные числа Фибоначчи. Таблица, приведенная ниже,

поясняет процесс слияния. Элемент таблицы вида *A*B* обозначает *A* отрезков относительной длины *B*.

Фаза	F1	F2	F3	Число обрабатываемых отрезков
1	13*1	8*1	Пусто	21
2	5*1	Пусто	8*2	16
3	Пусто	5*3	3*2	15
4	3*5	2*3	Пусто	15
5	1*5	Пусто	2*8	16
6	Пусто	1*13	1*8	13
7	1*21	Пусто	Пусто	21

Суммарное число проходов по данным составит $5\frac{4}{7}$, в то время, как двухфазная процедура затратила бы 8 проходов. Эту идею можно обобщить на *N* входных файлов, используя *N-1* –путевое слияние. В качестве примера рассмотрим случай *N=6* при числе начальных отрезков 129.

Фаза	F1	F2	F3	F4	F5	F6	Число отрезков
1	31*1	30*1	28*1	24*1	16*1	-	129
2	15*1	14*1	12*1	8*1	-	16*5	80
3	7*1	6*1	4*1	-	8*9	8*5	72
4	3*1	2*1	-	4*17	4*9	4*5	68
5	1*1	-	2*33	3*17	2*9	2*5	66
6	-	1*65	1*33	1*17	1*9	1*5	65
7	1*129	-	-	-	-	-	129

Чтобы метод работал, необходимо после каждой фазы иметь точное "фибоначчиево" распределение отрезков по файлам. Для этого некоторые файлы можно дополнить фиктивными отрезками нулевой длины. Точное фибоначчиево распределение можно получить, прокручивая рассмотренную схему в обратном порядке, циклически переставляя содержимое файлов и игнорируя выходной файл. При *N=6* имеем следующее распределение отрезков:

Уровень	F1	F2	F3	F4	F5	Сумма
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65
6	31	30	28	24	16	129
.	.	.	.	.	.	.
n	a_n	b_n	c_n	d_n	e_n	t_n
n+1	a_n+b_n	a_n+c_n	a_n+d_n	a_n+e_n	a_n	t_n+4a_n

Из приведенной таблицы следует, что

$e_n=a_{n-1}$

$d_n=a_{n-1}+e_{n-1}=a_{n-1}+a_{n-2}$

$c_n=a_{n-1}+d_{n-1}=a_{n-1}+a_{n-2}+a_{n-3}$

$b_n=a_{n-1}+c_{n-1}=a_{n-1}+a_{n-2}+a_{n-3}+a_{n-4}$

$a_n=a_{n-1}+b_{n-1}=a_{n-1}+a_{n-2}+a_{n-3}+a_{n-4}+a_{n-5}$,

где $a_0=1$ и $a_n=0$ при $n<0$.

Числа Фибоначчи порядка p F_n^p определяются правилами:

$$F_n^p = F_{n-1}^p + F_{n-2}^p + \ldots + F_{n-p}^p \text{ при n} \geq \text{p},$$

$$F_n^p = 0 \text{ при } 0 \leq n \leq p\text{-}2, \qquad F_{p-1}^p = 1.$$

В общем случае, если положить $p=N-1$, распределения отрезков по лентам в многофазном слиянии для N файлов аналогичным образом соответствуют числам Фибоначчи порядка p. В точном распределении n-го уровня в k-ом файле будет

$$F_{n+p-2}^p + F_{n+p-3}^p + \ldots + F_{n+k-2}^p$$

начальных отрезков для $1 \leq k \leq p$, а общее количество начальных отрезков во всех файлах составит

$$t_n = pF_{n+p-2}^p + (p-1)F_{n+p-3}^p + \ldots + F_{n-1}^p$$

9.2.1.4. Каскадное слияние

Каскадное слияние, подобно многофазному, начинается с точного распределения отрезков по лентам, хотя правила распределения другие. В качестве примера рассмотрим слияние, использующее 6 файлов. Каждая строка в таблице, приведенной ниже, представляет полный проход по всем данным.

№ прохода	F1	F2	F3	F4	F5	F6	Всего отрезков
1	55*1	50*1	41*1	29*1	15*1	-	190
2	-	5*1	9*2	12*3	14*4	15*5	190
3	5*15	4*14	3*12	2*9	1*5	-	190
4	-	1*15	1*29	1*41	1*50	1*55	190
5	1*190	-	-	-	-	-	190

Проход 2, например, поучается выполнением 5-путевого слияния с F1…F5 на F6, пока не опустеет F5, затем 4-путевого слияния с F1…F4 на F5, 3-путевого с F1, F2, F3 на F4, 2-путевого с F1, F2 на F3, и, наконец, однопутевого (копирования) – с F1 на F2. Подробно второй проход представлен в таблице:

Слияние	F1	F2	F3	F4	F5	F6
исходно	55*1	50*1	41*1	29*1	15*1	-
5-путевое	40*1	35*1	26*1	14*1	-	15*5
4-путевое	26*1	21*1	12*1	-	14*4	
3-путевое	14*1	9*1	-	12*3		
3-путевое	5*1	-	9*2			
копирование	-	5*1				

Ясно, что операция копирования излишня и оставлена в описании алгоритма только для сохранения единообразия процесса. Рассматривая процесс в обратном порядке, и игнорируя выходной файл, можно вывести точные распределения отрезков по файлам на любом этапе:

Уровень	F1	F2	F3	F4	F5
0	1	0	0	0	0
1	1	1	1	1	1
2	5	4	3	2	1
3	15	14	12	9	5
4	55	50	41	29	15
.	.	.	.	.	.
n	a_n	b_n	c_n	d_n	e_n
N+1	$a_n+b_n+c_n$ d_n+e_n	a_n+b_n $+c_n+d_n$	a_n+b_n+ c_n	a_n+b_n	a_n

Числа в распределении носят название *каскадных*.

9.2.1.5. Сортировка в одном файле

Алгоритм упорядочения данных в одном файле (то есть на месте) получается естественным обобщением метода пузырька. Роль такого

пузырька играет порция данных, в которой постепенно накапливаются записи с наибольшими (или наименьшими) значениями ключа. Величина порции составляет половину выделенного для сортировки объема оперативной памяти. Допустим, что файл разбивается на *N* порций (зон).

В начале процесса в память считываются две первые порции (зоны) данных и упорядочиваются там как единый массив, после чего половина массива с меньшими значениями ключа (назовем ее "нижней") записывается во вторую зону. На освободившееся место считывается третья зона, содержимое памяти вновь сортируется каким либо методом внутренней сортировки, и нижняя половина записывается в третью зону. Процесс повторяется для всех зон до *N-1* включительно. После считывания *N*-й зоны и внутренней сортировки уже верхняя половина (пузырек) записывается в *N*-ю зону. Таким образом, ключи последней зоны уже заняли свое окончательное положение в файле. Процесс повторяется для оставшихся *N-1* зоны, затем для оставшихся *N-2* зоны и т.д.

Ниже приведен текст функции, реализующей алгоритм.

```
int BubbleExternSort(char *FileName, int RecLen,
  int (*cmp)(const void *r1, const void *r2), int MemSize){
// Внешняя сортировка методом пузырька
// FileName - имя бинарного файла, содержащего таблицу
// RecLen - длина записи
// cmp - функция, сравнивающая ключи записей
//   возвращает <0,0,>0, как это принято для функций сравнения
// MemSize - размер оперативной памяти, который разрешено
//     использовать для сортировки
// при нормальном завершении возвращает 0
// в противном случае - код ошибки

int h; // дескриптор файла
long FileSize; // размер файла
unsigned long nZon; // число зон
int ZonSize; // размер зоны в байтах
int i;
char *Buffer; // память для сортировки
unsigned long izon, jzon;
int nByte;

h=_rtl_open(FileName, O_RDWR); // открытие бинарного файла
if(h<0) return 1; // не удалось открыть файл
FileSize=lseek(h,0L,SEEK_END);
if(RecLen*(FileSize/RecLen)!=FileSize) {
     close(h); // размер файла не кратен длине записи
     return 2;
}
ZonSize=(MemSize/RecLen/2)*RecLen;
nZon=FileSize/ZonSize;
```

```
if(nZon*ZonSize!=FileSize){ // последняя зона неполная
     nZon++;
}
if(ZonSize<RecLen){
     close(h);
     return 3; // мало дали памяти
}
Buffer=new char[2*ZonSize+1];
if(Buffer==NULL) {
     close(h);
     return 4; // нет памяти
}
Buffer[2*ZonSize]=0;

for(izon=nZon-1; izon>1; izon--){ // проходы по файлу
     // izon - число зон в данном проходе
     // читать первые две зоны
     lseek(h, 0L, SEEK_SET);
     nByte=_rtl_read(h, Buffer, 2*ZonSize);

     for(jzon=0; jzon < izon-1; jzon++){
          // сортировать
          qsort(Buffer, nByte/RecLen, RecLen, cmp);
          // нижнюю половину - в зону jzon
          lseek(h, jzon*ZonSize, SEEK_SET);
          _rtl_write(h, Buffer, ZonSize);
          // подкачать следующую зону
          lseek(h, (jzon+2)*ZonSize, SEEK_SET);
          nByte=_rtl_read(h, Buffer, ZonSize);
          // если прочитано меньше ZonSize,
          // то верхнюю половину сдвинуть вниз
          // на ZonSize-nByte вплотную к нижней
          if(nByte < ZonSize){
               for(i=0; i<ZonSize; i++){
                    Buffer[nByte+i]=Buffer[ZonSize+i];
               }
          }
          nByte+=ZonSize;
     }
     // переписать пузырек
     qsort(Buffer, nByte/RecLen, RecLen, cmp);
     lseek(h, (izon-1)*ZonSize, SEEK_SET);
     _rtl_write(h, Buffer, nByte);
}

// две первых зоны
lseek(h, 0L, SEEK_SET);
nByte=_rtl_read(h, Buffer, 2*ZonSize);
qsort(Buffer, nByte/RecLen, RecLen, cmp);
```

```
lseek(h,0L,SEEK_SET);
_rtl_write(h, Buffer, nByte);

delete [] Buffer;
close(h);
return 0;
}
```

Аналогично методу пузырька для внутренней сортировки, число проходов по данным $\sim N^2$, где N – число зон.

Контрольные вопросы

1. Какова оценка времени поиска записи в сортированной таблице?
2. Какие проблемы вызывают операции вставки и удаления для сортированной таблицы?
3. Какова нижняя граница времени работы сортировки, основанной на сравнениях ключей?
4. Перечислите известные вам методы внутренней сортировки.
5. Дайте описание алгоритмов внутренней сортировки, имеющих время работы порядка $N\log_2 N$
6. В чем заключается принципиальное отличие внешней сортировки от внутренней?
7. Опишите метод, позволяющий порождать сортированные отрезки длины большей, чем объем оперативной памяти, отведенной для этого.
8. За счет чего удается избежать операций копирования при использовании Фибоначчиева слияния?
9. Каким методом может быть выполнена внешняя сортировка данных без использования дополнительных файлов?

9.3. Древовидные таблицы

Явное использование структуры бинарного дерева позволяет быстро вставлять и удалять записи и производить эффективный поиск, и особенно целесообразно для организации динамических таблиц, подверженных частым вставкам и удалениям.

Рассмотрим простой метод построения дерева. Первую запись с ключом K_1 поместим в корень дерева. Второй ключ K_2 сравним с K_1. Если $K_2<K_2$, поместим его в левое поддерево, а если $K_2 \geq K_1$, то в правое поддерево. В общем случае, когда требуется поместить в дерево i-й ключ, поступаем так: сравниваем K_i с ключами, начиная с корня. Если K_i меньше очередного ключа, то переходим к левому сыну, в противном случае – к правому, а если соответствующий сын отсутствует, то это и есть место, куда нужно вставить ключ K_i. Пусть структура узла дерева:

```
struct NODE {
      void *Record; // указатель на запись таблицы
      NODE *Left, *Right;
```

```
};
```

Пусть `int Cmp(const void *Record1, const void *Record2);` – функция, выполняющая сравнение ключей записей `*Record1` и `*Record2`. Функция традиционно возвращает значения <0,0,>0 соответственно в случаях *ключ1<ключ2, ключ1=ключ2* и *ключ1>ключ2*.

Функция, выполняющая вставку новой записи в древовидную таблицу, приведена ниже:

```
#define LEFT 0
#define RIGHT 1
//---------------------------------
NODE *InsertRecord(NODE *Root, void *NewRecord){
NODE *x,*Cur,*Next;
int WhatSon; // каким сыном устанавливать новую запись -
             // - левым (LEFT) или правым (RIGHT)
int CmpKeys; // результат сравнения ключей

x=new NODE;
x->Record=NewRecord;
Cur=Root;
// поиск места вставки
for(;;){
     CmpKeys=Cmp(NewRecord,Cur->Record);
     switch(CmpKeys){
          case -1:
               WhatSon=LEFT;
               Next=Cur->Left;
               break;
          case 0:
          case 1:
               WhatSon=RIGHT;
               Next=Cur->Right;
               break;
     } // switch
     if(Next==NULL){
          break;
     }
     Cur=Next;
}
if(WhatSon==LEFT){
     Cur->Left=x;
} else {
     Cur->Right=x;
}
return x;
}
```

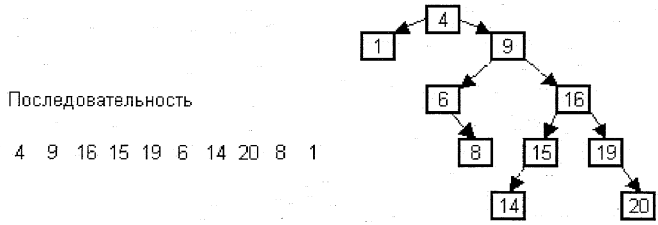


Рис. 45. Результат вставки ключей в бинарное дерево

На рис. 45 изображен результат работы алгоритма для заданной последовательности ключей.

Операция удаления узла:

Сначала заметим, что обход бинарного дерева в обратном порядке даёт сортированную последовательность. Все замечательные свойства древовидной таблицы связаны с этим обстоятельством. При удалении узла необходимо сохранить порядок следования узлов в обратном порядке. Легко удалить лист или узел, у которого пуста правая или левая связь. В общем случае одно из возможных решений таково: удалить следующий в обратном порядке узел, левая связь которого пуста, а затем вернуть его на место узла, который действительно требуется удалить. Ниже приведен текст функции, удаляющей узел, содержащий запись с целым ключом *Key*. Здесь предполагается, что дерево имеет голову, и, что само дерево является левым поддеревом головы. Голова содержит максимально возможное значение ключа. В случае успеха функция возвращает *true*.

```
struct NODE {
     int Info;
     NODE *Left;
     NODE *Right;
     int Rank; // для доступа по индексу
};
//--------------------------------------
bool DeleteKey(NODE *Head, int Key){
// удалить Key из дерева с головой Head
NODE *Cur, *Next=NULL,*father=NULL;

// ищем узел, запоминая каждый раз отца
for(Cur=Head; Cur!=NULL && Cur->Info!=Key; Cur=Next){
     if(Key<=Cur->Info){
          Next=Cur->Left;
     } else {
          Next=Cur->Right;
     }
     father=Cur;
}
```

```
if(Cur==NULL){ // не нашли Key
     return false;
}
// узел cur надо удалить
// может у Cur пуста правая связь ?
if(Cur->Right==NULL){
     if(father->Left==Cur){
          // он левый сын своего отца
          father->Left=Cur->Left;
     } else {
          // он правый сын своего отца
          father->Right=Cur->Left;
     }
     delete Cur;
     return true;
}
// поищем последователя с пустой левой связью
// шаг вправо и вниз до упора по левым связям
Next=Cur->Right;
father=Cur;
while(Next->Left!=NULL){
     father=Next;
     Next=Next->Left;
}

// Next - удаляемый узел
if(father->Left==Next){
     // удаляемый узел - левый сын своего отца
     father->Left=Next->Right;
} else {
     // удаляемый узел - правый сын своего отца
     father->Right=Next->Right;
}
Cur->Info=Next->Info;
delete Next;
return true;
}
```

9.3.1. Оценка трудоемкости поиска в случайном дереве

Обозначим значения ключей, следующих в порядке возрастания $k_1,k_2,...,k_n$. Вероятность того, что в корень попадет ключ k_i равна $1/n$. Тогда в левом поддереве будет $i-1$ узлов, а в правом $n-i$ узлов. Пусть a_n – средний уровень узла в дереве, содержащем n узлов. Тогда при условии, что ключ k_i находится в корне,

$$a_n^{(i)} = (a_{i-1}+1)\frac{i-1}{n} + \frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n}$$

Выражение в правой части содержит три слагаемых:

1. искомый узел находится в левом поддереве с вероятностью *(i-1)/n* и средний уровень его равен $a_{i-1}+1$
2. с вероятностью *1/n* это корень и его уровень 1.
3. с вероятностью *(n-i)/n* искомый узел находится в правом поддереве и средний уровень его $a_{n-i}+1$

Искомая величина a_n представляет собой среднее по i для всех $a_n^{(i)}$

$$a_n = \frac{1}{n}\sum_{i=1}^{n} a_n^{(i)} = \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\frac{i-1}{n} + \frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n}\right] =$$

$$1 + \frac{1}{n^2}\sum_{i=1}^{n}[(i-1)a_{i-1} + (n-i)a_{n-i}] = 1 + \frac{2}{n^2}\sum_{i=1}^{n}(i-1)a_{i-1} = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i a_i \qquad (1)$$

Отсюда следует:

$$a_n = 1 + \frac{2}{n^2}(n-1)a_{n-1} + \frac{2}{n^2}\sum_{i=1}^{n-2} i a_i \qquad (2)$$

А из (1) можно получить:

$$a_{n-1} = 1 + \frac{2}{(n-1)^2}\sum_{i=1}^{n-2} i a_i \qquad (3)$$

Из (3) можно найти $\sum_{i=1}^{n-2} i a_i$, и, подставив это в (2), получим:

$$a_n = \frac{1}{n^2}((n^2-1)a_{n-1} + 2n - 1) \qquad (4)$$

a_n можно представить в виде:

$$a_n = 2\frac{n+1}{n}H_n - 3, \qquad (5)$$

где $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$. Справедливость (5) можно проверить непосредственной подстановкой (5) в (4).

По формуле Эйлера $H_n = \gamma + \ln n + \frac{1}{12n^2} + \ldots$, где γ- постоянная Эйлера, равная 0.571… Для больших n получим:

$$a_n = (2\ln n + \gamma) - 3 \approx 1.386 \log_2 n$$

В заключение отметим, что, хотя среднее время поиска в случайном дереве пропорционально логарифму числа узлов, тем не менее, существует досадная возмож

ность получения вырожденного дерева, время поиска в котором пропорционально числу узлов. Примеры таких деревьев приведены на рис.46.

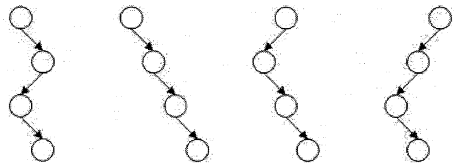


Рис.46. Примеры вырожденных деревьев

9.3.2. Оптимальные деревья

Оптимальным назовем дерево, высота которого минимальна. Это означает, что у него заполнены все уровни, кроме, быть может, последнего. Алгоритм построения оптимального дерева, позволяющего выполнить поиск по ключу, заключается в следующем:

1. отсортируем ключи и средний из них поместим в корень
2. левым сыном сделаем ключ, являющийся средним среди ключей слева от корня
3. правым сыном сделаем ключ, являющийся средним среди ключей справа от корня
4. точно также поступим при выборе сыновей каждого из узлов.

Ниже представлен текст функции, формирующей оптимальное дерево из массива ключей.

```
#define LEFT 0
#define RIGHT 1

struct NODE {
     int Info;
     struct NODE *Left;
     struct NODE *Right;
};
//-----------------------------------
NODE *CreateOptimalTree(int KeyArray[], int m, int n){
/*создать оптимальное дерево из ключей отрезка от m до n
 сортированного массива ключей KeyArray */
int k; NODE *Root;
if(n<m) return NULL;
Root=new NODE;
k=(n+m)/2;
Root->Info=KeyArray[k];
if(n==m){
    Root->Left=NULL;
    Root->Right=NULL;
} else {
    Root->Left=CreateOptimalTree(KeyArray,m,k-1);
    Root->Right=CreateOptimalTree(KeyArray,k+1,n);
}
return Root;
```

}

Построение оптимальных деревьев имеет смысл для статических таблиц, то есть для таких, для которых операции вставки и удаления отсутствуют. Восстановление оптимальности после вставки или удаления требует полного перестроения дерева.

9.3.3. Сбалансированные деревья

Несмотря на оптимистический прогноз для средней высоты случайного дерева поиска $O(\log_2 N)$, существует досадная возможность получить вырожденное дерево. Построение оптимального дерева и поддержание оптимальности обходится слишком дорого. Существуют компромиссные решения, которые рассматриваются ниже.

Сбалансированные деревья представляют собой компромисс между оптимальными и случайными деревьями. Авторы идеи сбалансированного дерева - Адельсон-Вельский и Ландис (1962г.), поэтому такие деревья называют также AVL-деревьями. AVL-деревья используют дополнительно 2 бита на узел и позволяют за время $\sim\log_2 N$, выполнять операции вставки, поиска и удаления.

Определим высоту дерева как максимальную длину пути от корня до листа. Будем называть дерево сбалансированным, если в каждом узле высота правого и левого поддеревьев различаются не более чем на ±1.

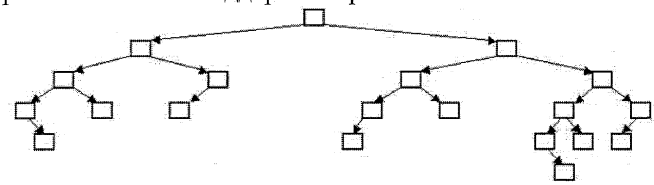


Рис.47. Пример сбалансированного дерева

На рис. 47 изображен пример сбалансированного дерева. Оценим максимальную высоту сбалансированного дерева. Обозначим минимальное число узлов в AVL-дереве высоты h через $N(h)$. В таком дереве с минимальным числом узлов одна из ветвей, исходящих из корня, должна содержать $N(h-1)$ узлов, а другая - $N(h-2)$ узлов. Таким образом,

$$N(h)=N(h-1)+N(h-2)+1 \qquad (6)$$

Очевидно, что $N(0)=1$, $N(1)=2$. Далее по формуле (6) находим , что $N(2)=4$. Для $h=3$ и $h=4$ можно непосредственно проверить, а затем по индукции доказать, что при $h\geq 3$ справедливо неравенство

$$N(h) > \alpha^{h+1} \qquad (7)$$

где $\alpha = \frac{\sqrt{5}+1}{2}$ – положительный корень уравнения x^2-x-1=0, которое является характеристическим для разностного уравнения (6). Действительно, допустим, что неравенство (7) выполняется для всех $k<N$. Тогда, подставив в правую часть равенства (6) нижние границы $N(h-1)$, $N(h-2)$ в соответствии с неравенством (7), получим:

$$N(h) > \alpha^h+\alpha^{h-1}+1 \qquad (8)$$

Подставим в левую часть неравенства (7) нижнюю грань для $N(h)$ из неравенства (8). В результате будет получено более сильное неравенство:

$$\alpha^h+\alpha^{h-1}+1 > \alpha^{h+1} \qquad (9)$$

Очевидно, что если (9) справедливо, то тем более справедливо (7). Преобразуем неравенство (9) к виду:

$$\alpha^{h-1}(\alpha^2-\alpha-1)-1 < 0$$

Трехчлен в скобках тождественно равен нулю, так как α - его корень. Остается $-1<0$. Таким образом, неравенство (7) доказано. Следовательно, если сбалансированное дерево содержит ветвь длины $h>3$, то число его вершин n удовлетворяет неравенству $n > N(h) > \alpha^{h+1}$, откуда

$$h+1 < \log_{\alpha}2 * \log_2 n \approx 1.44 \log_2 n \qquad (10)$$

Величина $h+1$ – это число сравнений ключей при поиске записи, расположенной в конце пути длиной h, исходящем из корня. Окончательный вывод: число сравнений ключей при поиске в сбалансированном дереве из n узлов не превышает $1.44 \log_2 n$.

9.3.4.1. Поддержание балансировки

Для работы алгоритма поддержания балансировки требуется в каждом узле хранить показатель сбалансированности, принимающий значения –1, 0, 1, и равный разности высот правого и левого поддеревьев узла.

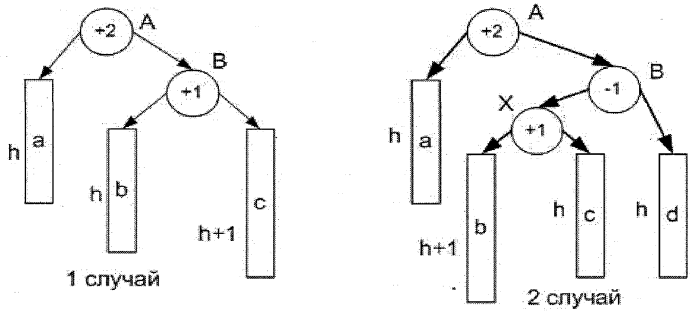


Рис. 48. Два случая нарушения балансировки

В принципе возможны только два случая нарушения балансировки, изображенные на рис. 48. Другие "плохие" случаи можно получить вращением вокруг вертикальных осей, проходящих через узлы A и B.

Первый случай имеет место, когда вставка увеличивает высоту правого поддерева узла B с *h* до *h+1*. Во втором случае либо *h=0* и X – новый узел, либо узел X имеет поддеревья с высотами *h, h+1*. Преобразования, изображенные на рис.49 восстанавливают балансировку, сохраняя правильность построения дерева.

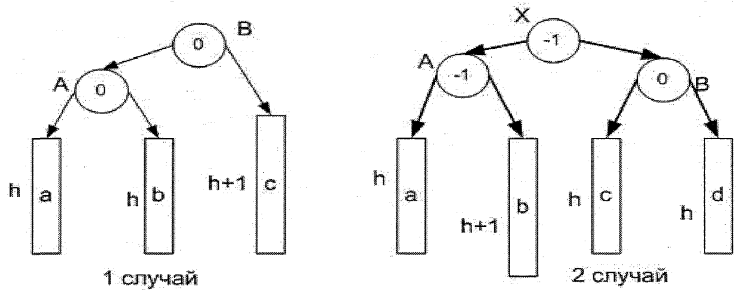


Рис. 49. Восстановление балансировки

В обоих случаях в дереве нужно изменить лишь несколько связей. Преобразованные деревья имеют высоту *h+2*, как до включения нового узла. Следовательно, часть дерева, расположенная над узлам A не изменится. Обратите внимание на тот факт, что порядок следования узлов в обратном порядке до и после преобразования не изменился.

9.3.5. Представление линейных списков деревьями

Такое представление позволяет за логарифмическое время иметь доступ к узлу дерева, как по ключу, так и по порядковому номеру. Для решения этой задачи в каждый узел дерева добавим новое поле по имени Rank. Это поле содержит порядковый номер узла при обратном обходе в дереве, которое из него исходит. На рис.50 вместе со значениями ключей в скобках указаны значения поля Rank.

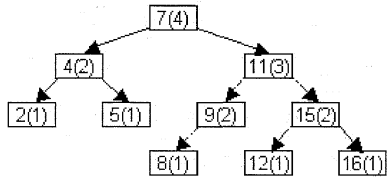


Рис. 50. Представление массива бинарным деревом

Ниже представлен текст функции, выполняющей поиск элемента массива по индексу. Алгоритм предполагает наличие головы у дерева. Собственно дерево является левым поддеревом головы.

```
struct NODE {
      int Info;
      NODE *Left;
      NODE *Right;
```

```
     int Rank;
};
//-----------------------------------------
NODE *FindByIndex(NODE *Head, int Index){
// найти узел по индексу (отсчет от 0)
int k;
NODE *Cur;// указатель на текущий узел

k=Index+1;
Cur=Head;
while(Cur!=NULL && k!=Cur->Rank){
     if(k < Cur->Rank){
          Cur=Cur->Left;
     } else {
          k-=Cur->Rank;
          Cur=Cur->Right;
     }
}
return Cur;
}
```

Таким образом, поскольку мы теперь имеем возможность доступа к узлу по индексу, то мы получили ещё один способ представления массивов. Причем массив может быть индексирован как ключом, так и индексом. Такие массивы называют ассоциативными. Роль ключа могут играть строки или любые другие типы данных.

Ассоциативный массив (словарь) – абстрактный тип данных, позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами. В паре k,v значение v называется значением, ассоциированным с ключом k.

Операция поиска возвращает значение, ассоциированное с заданным ключом, или признак того что значение не найдено. Операции вставки и удаления возвращают признак успешности операции.

Наиболее популярны реализации, основанные на различных деревьях поиска. Так, например, в стандартной библиотеке STL языка C++ реализация выполнена в виде *красно-чёрного дерева* (см. ниже). У каждой реализации есть свои достоинства и недостатки. Важно, чтобы три операции выполнялись как в среднем, так и в худшем случае за время $O(\log\ n)$, где n – количество хранимых пар. Для сбалансированных деревьев поиска (в том числе для красно-чёрных деревьев) это условие выполнено.

9.3.6. Красно-черные деревья

Красно-чёрное дерево – это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

Листья красно-черных деревьев не содержат данных. Такие листья не нуждаются в явном выделении памяти — нулевой указатель на потомка может фактически означать, что этот потомок — листовой узел, но в некоторых случаях работы с красно-черными деревьями использование явных листовых узлов может упростить реализацию алгоритм.

Каждый узел имеет атрибут *цвет*, принимающий значения *красный* или *черный*. В дополнение к обычным требованиям, налагаемым на двоичные деревья поиска, к красно-чёрным деревьям применяются следующие требования:

1. Узел либо красный, либо чёрный.
2. Корень – чёрный.
3. Все листья (NULL) – черные.
4. Оба потомка каждого красного узла – черные.
5. Всякий путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число черных узлов.

Пример красно-черного дерева изображён на рис.51 (вместо красного цвета использован белый).

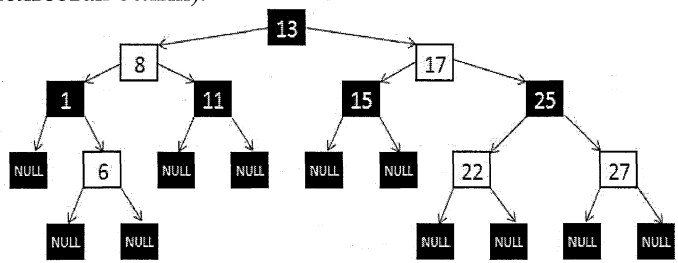


Рис. 51. Красно-черное дерево

Эти ограничения реализуют главное свойство красно-черных деревьев: путь от корня до самого дальнего листа не более чем в два раза длиннее пути от корня до ближайшего листа. Результатом является то, что дерево примерно сбалансировано.

Чтобы понять, почему это гарантируется, достаточно рассмотреть эффект свойств 4 и 5 вместе. Пусть для красно-чёрного дерева T число черных узлов в свойстве 5 равно B. Тогда кратчайший возможный путь от

корня дерева T до любого листового узла содержит B черных узлов. Более длинный возможный путь может быть построен путем включения красных узлов. Однако, свойство 4 не позволяет вставить несколько красных узлов подряд. Поэтому самый длинный возможный путь состоит из 2B узлов, попеременно красных и черных. Любой максимальный путь имеет одинаковое число черных узлов (по свойству 5), следовательно, не существует пути, более чем вдвое длинного, чем любой другой путь.

По этой причине для описания красно-черного дерева использованы «фиктивные листовые узлы», которые не содержат данных и просто служат для указания, где дерево заканчивается. Следствием этого является то, что все не листовые узлы имеют два потомка, один из них может быть пустым. Свойство 5 гарантирует, что красный узел обязан иметь в качестве потомков либо два черных нулевых листа, либо два черных внутренних узла. Для чёрного узла с одним потомком – листом и другим потомком, не являющимся листом, свойства 3, 4 и 5 гарантируют, что последний должен быть красным узлом с двумя черными листьями в качестве потомков.

Красно-чёрные деревья являются одними из наиболее активно используемых на практике самобалансирующихся деревьев поиска. В частности, контейнеры set и map в большинстве реализаций библиотеки STL языка C++, так же, как и многие другие реализации ассоциативных массивов в различных библиотеках, основаны на красно-чёрных деревьях.

9.3.7. B-деревья

B-дерево - это структура, очень широко применяемая для поиска данных на внешнем носителе. Ее используют практически все существующие системы управления базами данных.

B-деревом порядка m называется дерево, обладающее следующими свойствами:

1. каждый узел имеет не более m ключей
2. каждый узел имеет не менее m/2 ключей
3. корень, если он не лист, имеет от 1 до m ключей
4. все листья расположены на одном уровне
5. узел, имеющий *n* ключей, имеет *n+1* указателей на сыновей.
6. ключи расположены в узле в возрастающем порядке.

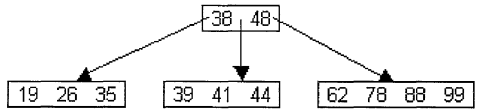


Рис.52. B-дерево

На рис. 52. изображено B-дерево порядка 4. Связи P_i как бы вставлены между ключами так, что указатель P_i указывает на поддерево с ключами K,

такими, что $K_{i-1} < K < K_i$. Узел, содержащий n ключей и $n+1$ указателей можно представить как на рис. 53.

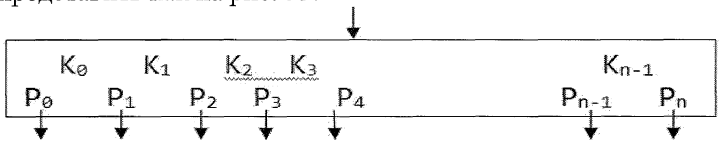


Рис.53. Узел B-дерева

Для B-дерева существует обобщение обратного обхода: сначала проходим поддерево, исходящее из P_0, затем ключ K_0, разделяющий поддеревья, исходящие из P_0 и P_1, затем поддерево, исходящее из P_1, и так далее. Для дерева на рис. 52 обратный обход дает последовательность:

19 26 35 38 39 41 44 48 62 78 88 99.

Таким образом, обратный обход B-дерева дает *сортированную* последовательность. Рассмотрим операции поиска, вставки и удаления.

Поиск

Начиная с корня, ищем ключ K среди ключей узла. Если поиск удачен, то операция завершается. Если же поиск неудачен, и нужный ключ имеет значение между K_i и K_{i+1}, спускаемся вниз по связи P_i и повторяем процесс. В конце концов, алгоритм либо найдет ключ, либо выйдет на пустую связь, что означает, что искомого ключа в дереве нет.

Вставка

Вставка всегда выполняется в лист. Место для вставки обнаруживается описанным выше процессом поиска. Если узел, в который должна быть выполнена вставка, содержит меньше m ключей, то помещаем новый ключ в узел и на этом операция заканчивается. Если же узел уже содержит максимально возможное число ключей, то выберем соседний неполный узел. Допустим, что это сосед слева, и рассмотрим множество ключей, состоящее из ключей левого соседа, разделяющего ключа в отце и ключей переполнившегося узла как на рис. 54.

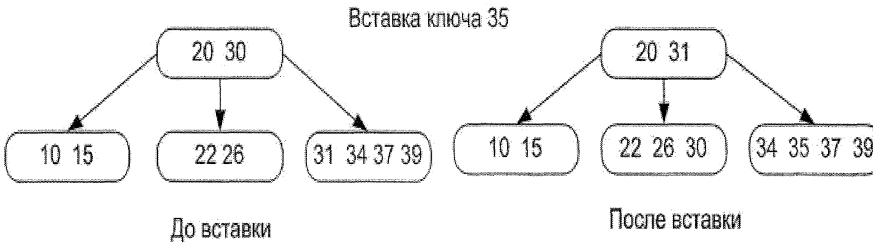


Рис. 54. Вставка ключа в B-дерево

Средний ключ поместим на место разделяющего, ключи слева от него передадим левому соседу, ключи справа от него поместим в узел, в котором произошло переполнение.

Если же не найдется соседа, которому можно было бы передать лишний ключ, то узел, в котором произошло переполнение, расщепляется на два, средний ключ поднимается к отцу, а оставшиеся ключи делятся пополам между двумя узлами. Рис. 55 иллюстрирует операцию. Передача ключа в узел отца может вызвать его расщепление. Процесс расщеплений может рекурсивно подняться до корня и расщепить его. В этом случае создается новый корень, содержащий единственный ключ.

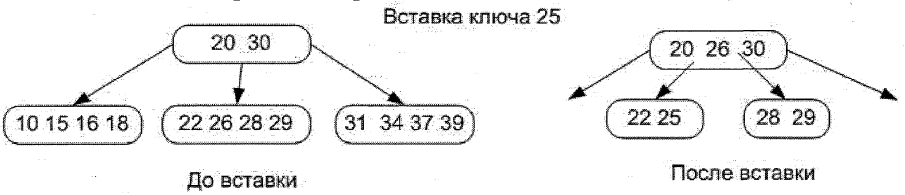


Рис. 55. Расщепление узла при вставке ключа в B-дерево

Удаление

Если удаляемый ключ находится в листе и после удаления число ключей в узле не становится меньше *m/2,* то ключ просто удаляется из узла и на этом операция заканчивается.

Если ключ находится не в листе, то вместо него удалим последователя в обратном порядке (а он всегда в листе), а затем вернем на место ключа, который действительно надо было удалить. Алгоритм определения последователя ключа из нелистового узла следующий:

1. спуститься вниз по указателю справа от ключа
2. далее спускаться по самым левым связям до листа

Так или иначе, в некотором листе станет на один ключ меньше. Ситуацию, когда число ключей становится меньше *m/2*, назовем *антипереполнением*. Антипереполнение должно быть ликвидировано. Для этого поищем "богатого" соседа, у которого можно было бы "занять" ключи. Если такой сосед найдется, то поступим так же, как и при ликвидации переполнения. Рассмотрим множество ключей, состоящее из ключей соседа, разделяющего ключа в отце и ключей узла, в котором произошло антипереполнение. Средний ключ поместим в отца, а оставшиеся ключи поделим пополам между узлом, в котором произошло антипереполнение и соседом. Рис. 56 иллюстрирует ситуацию.

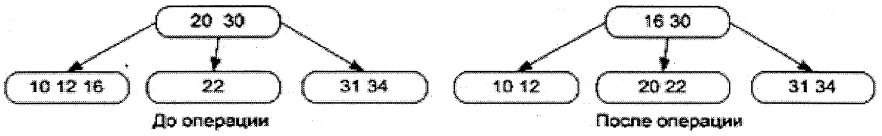


Рис.56. Ликвидация антипереполнения

Если же подходящего соседа не найдется, то выбирается любой из соседей, и удаляется, а ключи, составляющие описанное выше множество, помещаются в узел, вызвавший антипереполнение (рис.57).

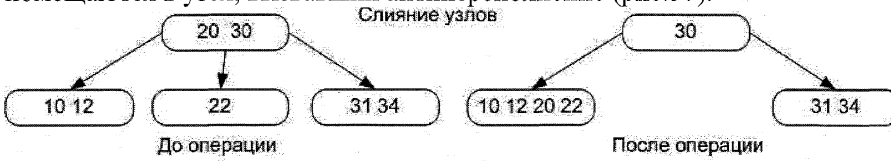


Рис.57. Слияние узлов при ликвидации антипереполнения

В результате слияния число ключей в отце уменьшается на единицу и может вызвать его антипереполнение, которое обрабатывается точно также. Процесс может рекурсивно подняться до корня и уничтожить его. В этом случае корневым узлом становится результат последнего слияния.

Оценка быстродействия

Оценим время поиска в B-дереве. Рассмотрим, к скольким узлам в наихудшем случае будет происходить обращение при поиске в B-дереве порядка *m*. Пусть B-дерево содержит *N* ключей и имеет высоту *h*. Число ключей на уровнях 1,2,3…h не менее 1, 2(m/2), $2(m/2)^2$, $2(m/2)^3$…$2(m/2)^{h-1}$, следовательно

$$N+1 \geq 2(1+m/2+(m/2)^2+(m/2)^3+...+(m/2)^{h-1}) = 2\frac{(m/2)^h-1}{m/2-1} \approx 2(\frac{m}{2})^{h-1} \quad (11)$$

или

$$h \leq 1+\log_{m/2}(\frac{N+1}{2}) \quad (12)$$

Так как при поиске происходит обращение не более чем к *h* узлам, то эффективность весьма высока, например, при m=200 и $N=10^6$, имеем *h=4*. При вставке может потребоваться расщепить *h* узлов. Однако среднее число расщеплений невелико. Поскольку любой новый узел, кроме корня, создается как результат расщепления, то суммарное число расщеплений, выполненное при построении дерева из *P* узлов, равно *P-1*. В дереве не менее чем $1+(P-1)\frac{m}{2}$ ключей, следовательно, $P \leq 1+\frac{N-1}{m/2}$. Это означает, что среднее число расщеплений, приходящееся на вставку одного ключа, меньше чем $\frac{1}{m/2}$.

Реально фиксируемой для B-дерева величиной является размер узла (страницы), а порядок m вычисляется исходя из размера записи.

Контрольные вопросы

1. Дайте описание алгоритма вставки записей в таблицу, организованную как бинарное дерево.

2. Как удалить запись из таблицы, организованной как бинарное дерево?
3. Дайте оценку трудоёмкости поиска в случайном бинарном дереве.
4. Дайте описание алгоритма построения оптимального дерева.
5. Приведите определение сбалансированного дерева.
6. Выведите выражение для максимальной высоты сбалансированного дерева.
7. Как выполняется восстановление балансировки?
8. Опишите алгоритм поиска записи по индексу в древовидной таблице.
9. Дайте определение красно-черного дерева
10. Приведите определение B-дерева.
11. Опишите алгоритмы вставки, поиска и удаления для B-дерева.
12. Оцените время, необходимое для выполнения операций поиска и вставки в B-дерево.

9.4. Таблицы с прямым доступом

Пусть в таблице из m записей все записи имеют разные значения ключа $k_1, k_2, \ldots, k_m$ и таблица отображается в вектор $a_1, a_2, \ldots a_n$, где $n \geq m$. Если определена функция $f(k)$ такая, что для любого значения k_i $(i=1 \ldots m)$ $1 \leq f(k_i) \leq n$, причем $f(k_i) \neq f(k_j)$ при $i \neq j$, то табличная запись с ключом k взаимно однозначно отображается в элемент $a_{f(k)}$. Функцию $f(k)$ называют *функцией расстановки*. Доступ к записи по ключу k производится непосредственно путем вычисления $f(k)$. Таблица, для которой существует и известна функция расстановки, называют таблицей с прямым доступом. Подбор функции расстановки, обеспечивающей взаимно однозначное преобразование ключа записи в адрес хранения, на практике можно решить только для постоянных таблиц с заранее известным набором значений ключа.

9.5. Рассеянные таблицы (Hash)

Таблицы с прямым доступом имеют очень ограниченное применение в первую очередь из-за ограничения $f(k_i) \neq f(k_j)$ при $i \neq j$, то есть требования взаимной однозначности преобразования ключа в адрес. Можно получить более гибкий метод, если отказаться от этого ограничения. Тогда сделается возможной ситуация $f(k_i)=f(k_j)$ при $i \neq j$, то есть более чем одна запись претендует на один и тот же адрес хранения. В этом случае ключи называют *синонимами*, а событие – *коллизией*. Чтобы таких ситуаций было меньше, функцию расстановки подбирают из соображений случайного и возможно более равномерного распределения ключей по памяти, отведенной для таблицы. Таблицы, построенные по такому принципу, называют *рассеянными, рандомизированными* или *хешированными* (*hash*)

таблицами. Метод вычисления адреса называют хешированием. Английский глагол *to hash* означает нарезать, искрошить, сделать месиво. Функцию расстановки *h(k)* называют хеш-функцией и стремятся сделать такой, чтобы она равномерно рассеивала ключи по памяти.

Будем считать, что хеш-функция имеет не более *m* различных значений $0 \le h(k) < m$. Хорошая хеш-функция должна удовлетворять двум требованиям: ее вычисление должно быть достаточно быстрым, а число коллизий минимальным. Как правило, хеш-функции используют ту же идею, что и линейные конгруэнтные генераторы псевдослучайных чисел, а именно:

$$h(k)=(a \times k) \bmod m,$$

где *a* и *b* – тщательно подобранные константы, m – число позиций в таблице, которое постоянно и назначается при создании таблицы. Возможное переполнение при выполнении операций умножения и сложения игнорируется.

Для разрешения коллизий существует два метода: метод цепочек переполнения и метод открытой адресации.

В методе цепочек переполнения поддерживается *m* линейных списков (можно деревьев) – по одному на каждый возможный хеш-адрес. После хеширования алгоритм получает адрес головы списка и производит в нем поиск простым перебором, если требуется поиск, или вставляет новую запись вслед за головой, если требуется вставка. Если имеется n ключей и *m* списков, то средняя длина списка *m/n*. На рис. 58 изображен массив списков для метода цепочек переполнения.

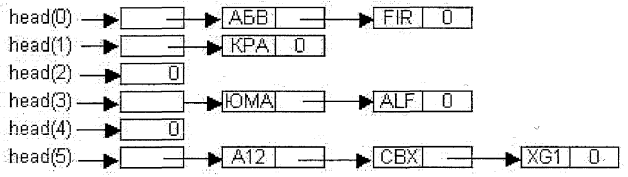


Рис.58. Метод цепочек переполнения

Метод открытой адресации состоит в том, чтобы, отправляясь от вычисленного хеш-адреса, просматривать записи до тех пор, пока не будет найден искомый ключ при поиске или свободная позиция при вставке. Если в процессе поиска алгоритм встретит свободную позицию, то это означает, что искомого ключа в таблице нет, так как механизм вставки такой, что запись вставляется в первую найденную свободную позицию. Простейшая схема открытой адресации, известная как *линейное опробование*, последовательно перебирает записи, отправляясь от хеш-адреса, и использует циклическую последовательность проб:

$$h(k),\ h(k)+1,\ h(k)+2,\ldots,m-2,\ m-1,\ 0,\ 1,\ 2,\ \ldots,h(k)-1$$

Эксперименты с линейным опробованием показали, что метод хорошо работает, пока таблица не слишком заполнена. Для линейного опробования

характерно явление "скучивания" – скопление записей группами. Действительно, если некоторый отрезок позиций таблицы полностью занят записями, то вероятность хеш-адреса новой записи попасть в этот отрезок и тем самым увеличить его, тем больше, чем больше длина отрезка. Таким образом, куча имеет склонность к экспоненциальному росту. Изменение шага просмотра с единицы на некоторую константу *C* не решает проблемы – куча все равно образуется, хотя элементы кучи и не являются физическими соседями в памяти.

В методе вторичного хеширования константа *C* зависит от ключа k. Алгоритм вычисляет две хеш-функции: $h_1(k)$ и $h_2(k)$. Как и прежде, $h_1(k)$ это хеш-адрес, а $h_2(k)$ – это шаг опробования. Значение $h_2(k)$ должно лежать в диапазоне от *1* до *m-1* и быть взаимно простым с *m* для того, чтобы с этим шагом можно было пройти все позиции таблицы.

Удаление из рассеянной таблицы несет с собой некоторые проблемы. Просто удалить запись, пометив занимаемую ей позицию как свободную, нельзя, так как при этом нарушится работа алгоритмов вставки и поиска. Действительно, при удалении записи с хеш-адресом *L* одновременно сделаются недоступными все синонимы, которые были включены в таблицу позднее. Можно выйти из положения, имея три типа позиций: свободные, занятые и удаленные. При поиске удаленные позиции трактуются как занятые, а при вставке как свободные. Однако это не решает проблему полностью, так как после длинной серии вставок и удалений, в таблице не останется свободных позиций – останутся только занятые и удаленные. В этом случае любой неудачный поиск (поиск ключа, которого нет в таблице) будет приводить к полному перебору всей таблицы. Единственное известное решение этой проблемы заключается в рехешировании, то есть построении таблицы заново. После рехеширования в таблице останутся только свободные и занятые позиции.

Рассмотрим трудоемкость операций над рассеянной таблицей.

Пусть для хранения таблицы отведено *m* позиций, и в таблице имеется *n* записей, тогда $\alpha=n/m \le 1$ – коэффициент загруженности памяти. Вероятность того, что хеш-адрес попадет в занятую позицию равна α, а в свободную $1-\alpha$. При вставке нам удастся найти место для новой записи:

- с 1-й попытки с вероятностью $1-\alpha$,
- со 2-й попытки – с вероятностью $\alpha(1-\alpha)$ (первая попытка неудачная, вторая успешна),
- с 3-й попытки – с вероятностью $\alpha^2(1-\alpha)$ (две первых неудачны, третья успешна) и т.д.

Математическое ожидание числа попыток *N* при вставке:

$$E(N) = 1\times(1-\alpha) + 2\times\alpha\times(1-\alpha) + 3\times\alpha^2\times(1-\alpha) + ... = (1-\alpha)(1+2\alpha+3\alpha^2+4\alpha^3+...) \quad (13)$$

Выражение $(1+2\alpha+3\alpha^2+4\alpha^3+...)$ является производной по α от суммы геометрической прогрессии $(\alpha+\alpha^2+\alpha^3+...)$, которая может быть вычислена по формуле

$$(\alpha+\alpha^2+\alpha^3+...)=\alpha/(1-\alpha) \qquad (14).$$

Подставляя производную правой части (14) в (13), получим:

$$E(N)=1/(1-\alpha) \qquad (15)$$

Для определения среднего числа проб при поиске заметим, что запись будет найдена в точности за столько же проб, сколько потребовалось при ее вставке. Каждая запись вставляется при своем значении α, так, например, при вставке первой записи $\alpha=0$. Усредняя (14) по α, получим:

$$E(R)=\frac{1}{\alpha}\int_0^{\alpha}\frac{1}{1-\alpha}d\alpha=\frac{1}{\alpha}\ln\frac{1}{1-\alpha} \qquad (16)$$

Таблица, приведенная ниже, дает представление о среднем числе проб при вставке и поиске.

α	$N(\alpha)$	$R(\alpha)$
0,20	1,25	1,12
0,50	2,00	1,39
0,80	5,00	2,01
0,90	10,00	2,56
0,95	20,00	3,15
0,99	100,00	4,65

Как видно из таблицы, не следует рассчитывать на полное использование памяти. Для динамичных таблиц коэффициент загрузки памяти не должен превышать 0,7 – 0,8.

Контрольные вопросы

1. В чем заключается основное отличие рассеянных таблиц от таблиц с прямым доступом?
2. Каким требованиям должны удовлетворять первичная и вторичная хеш-функции?
3. Какие методы разрешения коллизий в рассеянных таблицах вы знаете?
4. В чем заключается явление скучивания?
5. Опишите алгоритмы поиска, вставки и удаления для метода открытой адресации.
6. Дайте оценку трудоёмкости операций поиска и вставки для рассеянной таблицы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Вирт, Н. Алгоритмы и структуры данных. / Н. Вирт — СПб.: Невский диалект, 2013. — 272 с.

2. Ахо, А. Структуры данных и алгоритмы./ А. Ахо, Д. Хопкрофт, Дж.Ульман. — СПб.: Издательский дом "Вильямс", 2010. — 400 с.

3. Алгоритмы. Построение и анализ. / Т.Кормен [и др.]; — 3-е изд. — СПб.: Издательский дом "Вильямс". 2013. — 1328 с.

4. Кнут, Д.Э. Искусство программирования. Том 1. Основные алгоритмы. / Д.Э.Кнут 3-е изд. СПб.:Издательский дом "Вильямс". 2010. — 720 с.

5. Кнут, Д.Э. Искусство программирования. Том 3. Сортировка и поиск. / Д.Э.Кнут 2-е изд. СПб.:Издательский дом "Вильямс". 2010. — 824 с.

Учебное издание

Катаргин Михаил Юрьевич

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

Техн. редактор *А.В. Миних*

Издательский центр Южно-Уральского государственного университета

Подписано в печать 27.02.2015. Формат 60x84 1/16. Печать цифровая.
Усл. печ. л. 6,04. Тираж 30 экз. Заказ 90/111.

Отпечатано в типографии Издательского центра ЮУрГУ.
454080. г. Челябинск, пр. им. В.И. Ленина, 76.