**第一章 绪论**

**数据结构**

相互之间存在一种或多种特定关系的数据元素的集合。

**结构**

数据元素相互之间的关系称为结构。

**4类基本结构**

1. 集合
2. 线性结构：一对一
3. 树形结构：一对多
4. 图状结构：多对多

**数据结构的形式定义**

Data\_Structure=(D,S)

D是数据元素集，S是D上的关系集

**数据类型**

一个值的集合和定义在这个值集上的一组操作的总称。

**抽象数据类型**

指一个数学模型以及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表现和实现无关。例如整数类型，尽管它们在不同处理器器上的实现方法不同，但由于其定义的数学特性相同，在用户看来都是相同的。

ADT(D,S,P)

D是数据对象，S是D上的关系集，P是D上的操作集。

ADT抽象数据类型名{

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

}ADT抽象数据类型名

**算法**

是对特定问题求解步骤的一种描述，它是指令的有序序列，其中每一条指令代表一个或多个操作。

**时间复杂度**

随问题规模n的增大，算法执行时间的增长率和f（n）的增长率相同，叫做算法的渐近时间复杂度，简称时间复杂度。

算法中基本操作重复执行的次数是问题规模n的某个函数f（n），算法的时间度量记作

T（n）=O（f（n））

**第二章 线性表**

**线性表**

一个线性表是n个数据元素的有限序列。数据元素可以称为记录，含有大量记录的线性表又称为文件。

**算法2.1 顺序表合并**

void MergeList Sq(SqList La, SqList Lb, SqList &Lc){

//已知顺序线性表La和Lb的元素按值非减排列

//归并La和Lb得到新的顺序线性表Lc，Lc的元素也按值非递减排列

pa = La.elem; pb = Lb.elem;

Lc.listsize = Lc.length = La.length+Lb.length;

pc = Lc.elem = (ElemType \*)malloc(Lc.listsize\*sizeof(ElemType)) ;

pa.last = La.elem + La.length - 1;

pb.last = Lb.elem + Lb.length - 1;

while(pa<=pa.last && pb<=pb.last){ //归并

if(\*pa<=\*pb) \*pc++ = \*pa++;

else \*pc++ = \*pb++;

}

while(pa<=pa.last) \*pc++ = \*pa++; //插入La的剩余元素

while(pb<=pb.last) \*pc++ = \*pb++; //插入Lb的剩余元素

}

**线性链表**

用一组任意的存储单元存储线性表的数据元素；除了存储其本身的信息之外，还需存储一个指示其直接后继的信息。这两部分信息称为**结点**，存储元素信息的域称为**数据域**；存储直接后继位置的域称为**指针域**。n个结点链接成一个**链表**。

**算法2.2 链表插入操作**

Status List\_Insert(LinkList &L, int i, ElemType e){

//在带头结点的单链表L中第i个位置之前插入元素e

p = L; j = 0;

while(p && j<i-1) {

p = p->next; ++j;

}

if(!p || j>i-1) return ERROR; //i小于1或者大于表长加1

s = (LinkList)malloc(sizeof(LNode)) ; //生成新结点

s->data = e; s->next = p->next; //插入L中

p->next = s;

return OK;

}

**算法2.3 链表删除操作**

Status List\_Delete(LinkList &L, int i, ElemType &e){

//在带头结点的单链表L中，删除地i个元素，并由e返回其值

p = L; j = 0;

while(p->next && j<i-1) { //寻找第i个结点，并令p指向其前驱

p = p->next; ++j;

}

if(!p-》next || j>i-1) return ERROR; //删除位置不合理

q = p->next; p->next = q->next //删除并释放结点

e = q->data; free(q);

return OK;

}

**算法2.4 有序链表合并**

void MergeList(LinkList &La, LinkList &Lb, LinkList &Lc) {

//已知链表la和Lb的元素按值非递减排列

//归并La和Lb得到新链表Lc，Lc的元素也按值非递减排列

pa = La->next; pb = Lb->next;

Lc = pc = La; //用La的头结点作为Lc的头节点

while(pa && pb) {

if(pa->data <= pb->data){

pc->next = pa; pc = pa; pa = pa->next;

}

else{

pc->next = pb; pc = pb; pb = pb->next;

}

}

pc->next = pa?pa:pb; //插入剩余片段

free(Lb); //释放Lb的头结点

}

**第三章 栈和队列**

**栈**

是限定仅在表尾进行插入和删除操作的线性表；表为称为栈顶（top），表头称为栈底（bottom）。先进后出。

**算法3.1 汉诺塔**

void f(int n, char A, char B, char C)

{

if(1 == n)

printf("num:%d %c->%c ", n, A, C);

else{

f(n-1, A, C, B);

printf("num:%d %c->%c ", n, A, C);

f(n-1, B, A, C);

}

return;

}

**队列**

先进先出，用链表实现还是构建一个空的头结点，删除最后一个元素时令队尾指针等于队头指针。队头（front）队尾（rear）。用链表存储时还是先构造1个空的头结点。

**算法3.1 求循环队列长度**

int QueueLength(SqQueue Q) {

//返回Q是元素个数，即队列的长度

return (Q.rear - Q.front + MAXSIZE) % MAXSIZE;

}

**第四章 串**

**串（或字符串）**

由零个或多个字符组成的有限序列。字符串的0号单元可以存放串的长度。

子串的定位操作通常称作串的**匹配模式**（其中要匹配的子串成为**模式串**），被匹配的串称为**主串**。

**算法3.2 串穷举查找**

int Index\_BF(struct String \*Sst1, struct String \*Sst2){

int i ,j;

i = j = 1; //串开始的位置从1开始

while(i <= Sst1->length && j <= Sst2->length){

if(Sst1->ch[i] == Sst2->ch[j]){

i++; j++;

}

else{

i = i-j+2; //i会退到开始比对位置加1（i-j+1+1)

j = 1;

}

}

if(j > Sst2->length)

return i-Sst2->length; //这里i已经加上了1

else

return 0;

}

**第五章 数组和广义表**

**广义表**

一般记作LS = （a1, a2, a3, ···,an),ai可以是单个元素，也可以是广义表，分别称为广义表Ls 的**原子**和**子表**。当广义表非空时，称第一个元素ai为LS的**表头**，其余元素组成表是LS的**表尾**。

**第六章 树和二叉树**

**树**

n个结点的有限集，在非空树中：1.有且仅有1个根结点；2.当n>1时，其余结点可以分为m（m>0）个互不相交的有限集T1，T2···，Tm，其中每一个集合本身又是一棵树，并且成为根的**子树**。**树的度**是树内各结点度的最大值；树中结点的最大层次称为树的**深度**。

**二叉树**

每个结点至多有2棵子树，且有左右之分。

**性质1**在二叉树第i层上至多有2^（i-1）个结点（i>=1）。

**性质2**深度为k的二叉树至多有（2^k）-1个结点（k>=）。

**性质3**对任意一棵二叉树T，如果其终端结点数为n0，度为2的结点数为n2，则 n0=n2+1。

设n1为度为1的结点数，所以结点总数为

n=n0+n1+n2

设B为分支总数，则n=B+1。由于这些分支是由度为1或2的结点射出的，所 以又有B=n1+2n2，于是得

n=n1+2n2+1

所以

n0=n2+1

**性质5** 在完全二叉树中：如果i>1，则其双亲PARENT（i）是结点[i/2]。

在完全二叉树中：如果2i<=n，则其左孩子LCHILD（i）是结点[2i]。

在完全二叉树中：如果2i+1<=n，则其右孩子RCHILD（i）是结点[2i+1]。

**算法6.1 创建二叉树**

void CreateBiTree(PBITREE \*T){

char cha;

scanf("%c", &cha);

if('#' == cha)

\*T = NULL;

else{

\*T = (PBITREE)malloc(sizeof(BITREE));

(\*T)->data = cha;

CreateBiTree(&(\*T)->pLchild);

CreateBiTree(&(\*T)->pRchild);

}

}

//复制二叉树

int Copy(PBITREE T, PBITREE \*NewT){

if( NULL == T){

\*NewT = NULL;

return 0;

}

else{

\*NewT = (PBITREE)malloc(sizeof(BITREE));

(\*NewT)->data = T->data;

Copy(T->pLchild, &(\*NewT)->pLchild);

Copy(T->pRchild, &(\*NewT)->pRchild);

}

}

//求二叉树深度

int Depth(PBITREE T){

int m, n;

if( NULL == T)

return 0;

else{

m = Depth(T->pLchild);

n = Depth(T->pRchild);

if(m > n)

return m+1;

else

return n+1;

}

}

//求二叉树结点

int NodeCount(PBITREE T){

if( NULL == T)

return 0;

else{

return NodeCount(T->pLchild) + NodeCount(T->pRchild) + 1;

}

}

//求二叉树叶子结点

int LeafCount(PBITREE T){

if( NULL == T)

return 0;

else{

if(T->pLchild==NULL && T->pRchild==NULL)

return 1;

else{

return LeafCount(T->pLchild) + LeafCount(T->pRchild);

}

}

}

**算法6.2a 二叉线索树非递归遍历**

Status InOrderTraverseThr(BiThrTree T, Status( \*Visit)(TElemType)) {

//T指向头结点，头结点的左链指向根结点

//中序遍历二叉线索树的非递归算法，对每个数据元素调用函数Vist

p = t->lchild; //p指向根结点

while(p != T) { //空树或者遍历结束时，p = T

while(p->LTag == Link) p = p->lchild;

if(!Visit(p->data)) return ERROR; //访问其左子树为空的结点

while(p->RTag == Tread && p->rchild != T){

p = p->rchild; Visit(p->data); //访问后继结点

}

p = p->rchild;

}

return OK;

}

**算法6.2b二叉树线索化**

status InOrderThreading(BiThrTree &Thrt, BiThrTree T) {

//中序遍历二叉树T，并将其中序线索化，Thrt指向其头结点

//Link = 0; Thread = 1;

if(!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode)))) exit(OVERFLOW);

Thrt->LTag = Link; Thrt->RTag = Tread; //建立头结点

Thrt->rchild = Thrt; //右指针回指

if(!T) Thrt->lchild = Thrt; //若二叉树空，则左指针回指

else{

Thrt->rchild = T; pre = Thrt;

InThreading(T); //中序遍历进行中序线索化

pre->RTag = Tread; pre->rchild=Thrt; //最后一个结点线索化

Thrt->rchild = pre;

}

return OK;

}

**算法6.2c二叉树线索化**

void InThreading(BiThrTree p){

if(p){

InThreading(p->lchild); //左子树线索化

if(!p->lchild) {

p->LTag = Tread; p->lchild=pre; //前驱线索

}

if(!pre->rchild){

pre->RTag = Tread; pre->rchild=p; //后继线索

}

pre = p; //保持pre指向p的前驱

InThreading(p->rchild); //右子树线索化

}

}

**树和森林**

**树的存储结构**

**1.双亲表示法**

用数组存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点在链表中的位

置。



|  |  |  |
| --- | --- | --- |
| 0 | R | -1 |
| 1 | A | 0 |
| 2 | B | 0 |
| 3 | C | 0 |
| 4 | D | 1 |
| 5 | E | 1 |

**2.孩子表示法**

把每个结点的孩子结点排列起来，看成一个线性表，且以单链表作为存储结构，为了便于查找，可采用数组存储。

|  |  |  |
| --- | --- | --- |
| 0 | A |  |
| 1 | B | ^ |
| 2 | C |  |
| 3 | D | ^ |

5

^

3

3

0

^

**3.孩子兄弟表示法**

以二叉链表作为树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点分别命名为firstchild域和nextsibling域。

**森林**同样可以用孩子兄弟表示法存储为一棵二叉树。

**赫夫曼树及其应用**

**路径长度**

从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数目称作路径长度。**树的路径长度**是从树根到每一个结点的路径长度之和。**结点的带权路径长度**为从该结点到树根之间的路径长度与结点上权的乘积。**树的带权路径长度**为树中所有叶子结点的带权路径长度之和。

**最优二叉树或赫夫曼树**

拥有n个叶子结点二叉树其中带权路径长度最小的二叉树称作最优二叉树赫夫曼树。

**构造赫夫曼树**

1.构造森林全是根

2.选用两小造新树

3.删除两小添新人

4.重复2、3剩单根

**前缀编码**

任意一个字符编码都不是另一个字符编码的前缀。

**赫夫曼编码**

设计电文总长最短的二进制前缀编码即以n种字符出现的频率作权，设计一棵赫夫曼树的问题，由此得到的二进制前缀编码便称作**赫夫曼编码。**左分支表示0右分支表示1。

**算法6.3a 创造哈夫曼树**

void CreateHuffmanTree(HuffmanTree \*HT, int n){ //创造哈夫曼树

int m, i, t, s1, s2;

if(n < 1) return;

m = 2\*n-1; //n个叶子结点要经过n-1次合并，所以总结点为2n-1

\*HT = (HuffmanTree)malloc(sizeof(HTNode)\*(m+1)) ;

for(i=1; i<=m; i++){ //初始化哈夫曼树

(\*HT)[i].parent = 0; (\*HT)[i].lchld = 0; (\*HT)[i].rchld = 0;

}

for(i = 1; i <= n;i++){ //输入前n个叶子结点的权重

printf("请输入第%d个权重数据：", i);

scanf("%d", &t);

(\*HT)[i].weight = t;

}

for(i= n+1; i<=m; i++){ //对后面m-n个结点赋值

Select(\*HT, i-1, &s1, &s2); //每次在前i-1项parent=0的元素选择权重最小的两个

(\*HT)[s1].parent = i; (\*HT)[s2].parent = i; //把i的下标赋值给s1和s2

(\*HT)[i].lchld = s1;(\*HT)[i].rchld = s2;

(\*HT)[i].weight = (\*HT)[s1].weight + (\*HT)[s2].weight;

}

//创造哈夫曼编码

char \*\* HC = (char \*\*)malloc((n+1)\*sizeof(char \*));

char \*cd = (char \*)malloc(n\*sizeof(char));

cd[n-1] = '\0';

int start, c, f;

for(i=1; i<=n; i++){

start = n-1;

for(c = i, f = (\*HT)[i].parent; f != 0;c = f, f = (\*HT)[f].parent){

if((\*HT)[f].lchld == c){

start--; cd[start] = '0';

}

else{

start--; cd[start] = '1';

}

HC[i] = (char \*)malloc((n-start)\*sizeof(char));

strcpy(HC[i], &cd[start]);

}

}

free(cd);

for(i=1; i<=n; i++)

printf("%s\n", HC[i]);

}

**算法6.3b 选择函数**

void Select(HuffmanTree HT,int end,int \*s1,int \*s2)

{

int i, m, n, count, tmp;

for(i=1,count=1;i<=end;i++)

{

//结点成员parent值为0表明该结点还未被安排

//将第一个，第二个未被安排的结点序号赋给m,n

if(HT[i].parent ==0)

{

if(count==1)

m=i;

if(count==2)

n=i;

count++;

}

if(count>2)

break;

}

//令m为结点较小权值的序号，令n为较大的序号

if(HT[m].weight>HT[n].weight)

{

tmp=n;

n=m;

m=tmp;

}

//i从下一个开始，一直扫描到end

i=(m>n?m:n)+1;

while(i<=end)

{

//同样寻找未被安排的结点

if(HT[i].parent==0)

{

//i的权重比m的小,则用m替n, i替m

if(HT[i].weight<HT[m].weight)

{

n=m;

m=i;

}

//i的权重介于m和n之间，则用i替n

//还剩一种情况，当i比n大时，不做任何改变

//故在此分为了两类

else

{

if(HT[i].weight>=HT[m].weight&&

HT[i].weight<HT[n].weight)

n=i;

}

}

i++;

}

//用s1返回最小权重序号

//用s2返回次小权重序号

\*s1=HT[m].weight<=HT[n].weight?m:n;

\*s2=HT[m].weight>HT[n].weight?m:n;

return;

}

**第七章 图**

**顶点（Vertex）**

在图中的数据元素通常称作顶点。

**弧（Arc）**

VR是两个顶点之间的关系的集合；若<v, w>属于VR，则<v, w>表示从v到w的一条**弧**，且称v为**弧尾**，称w为**弧头**，此时图称为**有向图**。无序对（v, w）表示v和w之间的一条**边**，此时的图称为**无向图**。有1/2n（n-1）条边的无向图称为**完全图**。具有n（n-1）条弧的有向图称为**完全有向图**。有很少条边（如e<nlogn）的图称为**稀疏图**，反之称为**稠密图。**

**权**

弧和边具有与它相关的数。

**网**

带权的图通常称为网。

**邻接点**

如果边（v， v~）属于E，则称顶点v和v~互为邻接点。

**度**

顶点v的度是和v相关联的边的数目。以顶点v为头的弧的数目称为v的**入度**；以顶点v为尾的弧的数目称为v的**出度**。

**回路或环**

第一个顶点和最后一个顶点相同的路径。序列中顶点不重复出现的路径称为简单路径。

**连通分量**

无向图的极大连通子图，再多一个顶点就不连通了。

**邻接矩阵**

typedef struct{ //邻接矩阵图

char vexs[MVNum]; //顶点

int arcs[MVNum][MVNum]; //矩阵

int vexnum, arcnum; //顶点和边的数量

bool visited[MVNum]; //遍历记号数组

}AMGraph;

**算法7.1找位置**

int LocateVex(AMGraph \*G, char u){

int i;

for(i=0; i<G->vexnum; i++)

if( u == G->vexs[i])

return i;

return -1;

}

**算法7.2 创建矩阵**

void Create(AMGraph \*G){

int i,j;

printf("输入总顶点数：");

scanf("%d", &G->vexnum) ;

for(i=0; i<G->vexnum; i++)

G->visited[i] = false;

printf("输入总边数：");

scanf("%d", &G->arcnum) ;

for(i = 0; i<G->vexnum; i++){

printf("输入第%d个点的信息：", i);

getchar();

scanf("%c", &G->vexs[i]);

}

for(i=0; i<G->vexnum; i++)

for(j=0; j<G->vexnum; j++)

G->arcs[i][j] = Maxint;

char v1, v2;

int w, k;

for(k=0; k<G->arcnum; k++){

printf("输入2个顶点v1、v2和权值weight：");

getchar();

scanf("%c %c %d", &v1, &v2, &w) ;

i = LocateVex(G, v1);

j = LocateVex(G, v2);

G->arcs[i][j] = w;

G->arcs[j][i] = G->arcs[i][j];

}

}

**算法7.3深度优先搜索**

void DFS(AMGraph \*G, int v){

printf("%d->", v);

G->visited[v] = true;

int w;

for(w = 0; w<G->vexnum; w++){

if((G->arcs[v][w] != 0) && (!G->visited[w]))

DFS(G, w);

}

}

**算法7.4 广度优先搜索**

void BFSTraverse(Gaph G, status (\* Visit)(int v)){

//按广度优先非递归遍历图G。使用辅助队列Q和访问标志数组visited

for(v=0; v<G.vexnum; v++) visited[v] = FALSE;

InitQueue(Q); //置空的辅助队列

for(v=0; v<G.vexnum; v++)

if(!visited[v]){

visited[v] = TRUE; Visited(v); //v尚未访问

EnQueue(Q, v); //v入队

while(!QueueEmpty(Q)){

DeQueue(Q, u); //队头元素出队并置为u

for(w=FirstAdjVex(G, u); w>=0; w=NextAdjVex(G, u, w))

if(!Visited){ //w为u的尚未访问的邻接点

Visited[w] = TRUE; Visit(w);

EnQueue(Q, w);

}

}

}

}

**最小生成树（MST）**

**普里姆（prim）算法**

顶点各边中取最小的边，然后重复下去，直到全部顶点连通。

**克鲁斯卡尔（Kruskal）算法**

图中每个顶点自成一个连通分量，选择代价最小的边，若该边依附的顶点在不同的连通分量上，则将该边加入进去。

**有向无环图**

一个无环的有向图称作有向无环图。通常用来描述一个工程或系统的进行过程，一个工程可以分为若干个子工程，只有完成这些子工程才能导致整个工程完成。

**AOV网**

用一个有向图表示一个工程的各个子工程及其相互制约的关系，其中顶点表示活动，弧表示活动之间的相互制约关系，称这种有向图为顶点表示活动的网，简称AOV网（Activity on Verte network）。

**AOE网**

用一个有向图表示一个工程的各个子工程及其相互制约的关系，以弧表示活动，以顶点表示活动的开始或结束事件，称边表示活动的网，简称AOE网（Activity on Edge）。

**拓扑排序**

在AOV网没有回路的前提下，我们将全部活动排成一个线性序列，使得AOV网中有弧<i, j>存在，在这个序列中，i一定排在j的前面，具有这种性质的线性序列称为**拓扑有序序列**，相应的拓扑有序排序的算法成为称为**拓扑排序**。

**拓扑排序的方法**

在有向图中选择一个没有前驱的顶点且输出。

从图中删除该顶点和所有以它为尾的弧。

重复上述两步，直至全部顶点均已输出。

若网中所有顶点都在拓扑有序序列中，则该AOV网必定不存在环。

**关键路径**

路径长度最长的路径叫做**关键路径**。把工程计划表示为边表示为活动的网络，即AOE网，用顶点表示事件，弧表示活动，弧的权表示活动的持续时间。事件表示在它之前的活动已经完成，在它之后的活动可以开始。

设活动ai用弧<j, k>表示，其持续时间为：Wj,k

则有：（1）e（i）=ve（j） （2）l（i）=vl（k）-Wj,k //k的最迟发生减去弧ai的权重

关键路径：e（i）=l（i）

（1）从ve（0）=0开始，往后推

ve（j）=Max{ve（i）+Wi,j} //找出权重最大的弧

<i, j>为弧

1. 从最后一个点向前推

vl（i）=Min{vl（j）-Wi,j}

<i, j>为弧

**最短路径**

**第九章 查找**

**查找表**

由同一类型的数据元素构成的集合。只做查找操作的表成为**静态查找表**，若在查找表中同时插入或删除数据元素，则称此表为**动态查找表**。

**关键字**

数据元素中某个数据项的值，用它可以标识一个数据元素；若此关键字可以唯一的标识一个记录，则称此关键字为**主关键字**。反之，称用以识别若干记录的关键字为**次关键字**。

**静态查找表**

**算法9.1 顺序查找**

int sequentialsearch(list Tbl, ElementType k){

int i;

Tbl->Element[0] = k; //在0位置设置哨兵，可减少一半的时间

for(i = Tbl->Length; Tbl->Element[i] != k; i--);

return i;

}

**算法9.2 折半查找（查找表必须有序）**

int SerchBin(SSTable ST, KeyType key) {

//在有序表ST中折半查找关键字等于key的数据元素

low = 1; high = ST.length;

while(low <= high) {

mid = (low + high) / 2;

if(ST->data[mid] == key)

return mid; //找到待查元素

else if(ST->data[mid] > key)

high = mid - 1; //继续在前半区查找

else

low = mid + 1; //继续在后半区查找

}

return 0;

}

**索引顺序表（分块查找）**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 索引表 | | |
| 最大关键字 | 22 | 48 | 85 |
| 起始地址 | 1 | 7 | 13 |

表中包含如上述的索引表。

**动态查找表**

**二叉排序树**

1. 若它的左子树不空，则左子树上的所有结点均小于它的根结点。

（2）若它的右子树不空，则右子树上的所有结点均大于它的根结点。

（3）它的左右子树也分别为二叉排序树。

**算法9.3a 二叉排序树查找**

bool SearchBST(BiTree T, int key, BiTree f, BiTree \*p){ //查找key，找到后返回其地址\*p

//在根指针T所指的二叉排序树中递归的查找关键字等于key的数据元素，若查找成功，

//则指针p指向该数据元素结点，并返回true，否则指针p指向查找路径上访问的

//最后一个结点并返回false，指针f指向T的双亲，其初始调用值为NULL

if(T == NULL){

\*p = f; //查找不成功

return false;

}

else if(key == T->data){

\*p = T; //查找成功

return true;

}

else if(key < T->data)

return SearchBST(T->lchild, key, T, p); //在左子树中继续查找

else

return SearchBST(T->rchild, key, T, p); //在右子树中继续查找

}

**算法9.3b 二叉排序树插入**

bool InsertBST(BiTree \*T, int key){ //插入值

BiTree p = NULL;

if(SearchBST(\*T, key, NULL, &p) == false){ //查找不成功

BiTree s = (BiTree)malloc(sizeof(BiTNode));

s->data = key;

s->lchild = s->rchild = NULL;

if(p == NULL)

\*T = s; //被插\*s结点为新的根结点

else if(key < p->data)

p->lchild = s; //被插结点为\*s的左孩子

else

p->rchild = s; //被插结点为\*s的右孩子

return true;

}

else

return false;

}

**算法9.3c 二叉排序树删除元素**

bool DeleteBST(BiTree \*T, int key){ //找到后删除值

if(\*T == NULL)

return false; //不存在关键字等于key的数据元素

else{

if((\*T)->data == key)

return Delete(T); //找到关键字等于key的数据元素

else if(key < (\*T)->data)

return DeleteBST(&((\*T)->lchild), key);

else

return DeleteBST(&((\*T)->rchild), key);

}

}

bool Delete(BiTree \*p){ // 删除值，并且重建树

BiTree q, s;

if((\*p)->rchild == NULL){ //右子树空则只需要重接它的左子树

q = \*p;

\*p = (\*p)->lchild;

free(q);

}

else if((\*p)->lchild == NULL){ //只需要重接它的右子树

q = \*p;

\*p = (\*p)->rchild;

free(q);

}

else{

q = \*p; s = (\*p)->lchild;

while(s->rchild != NULL){ //转左，然后向右到尽头

q = s; s = s->rchild; //s指向被删除结点的前驱

}

(\*p)->data = s->data;

if(q != \*p) //重接\*q的右子树

q->rchild = s->lchild;

else //重接\*q的左子树

q->lchild = s->lchild;

free(s);

}

return true;

}

**平衡二叉树（AVL树）**

它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。结点的**平衡因子**定义为左子树的深度减去右子树的深度，则平衡二叉树上所有的结点的平衡因子只可能是-1、0和1。

**散列表（哈希表）**

在存储位置和它的关键字之间建立一个确定的对应关系f，使每个关键字和结构中一个唯一的存储位置相对应。称这个对应关系f为**哈希（Hash）**函数。根据哈希函数和处理冲突的方法将一组关键字映像到一个有限的连续的地址集上，这种表成为**哈希表**。

**冲突**

对不同的关键字可能得到同一哈希地址，这种现象称为**冲突**。

**哈希函数构造方法**

**1.直接定址法**

H（key）=key或H（key）=a·key+b

优点没有冲突，缺点存储空间大

1. **除留取余法**

取关键字被某个不大于哈希表长m的数p除后所得余数为哈希地址。即

H（key）=key MOD p，p<=m

一般情况下，可以选p为质数。

**处理冲突的方法**

**1.开放定址法**

Hi=（H（key）+di）MOD m

其中：H（key）为哈希函数，m为哈希表长，di为增量序列，可有下列3种取法

1. di=1，2，3…，m-1称线性探测再散列；
2. di=1^2，-1^2，2^2，-2^2，…，+-k^2称二次探测再散列；
3. di=伪随机数列，称伪随机探测再散列；

**2.链地址法**

将所有关键字为同义词的记录存储在同一线性链表中。设立一个指针型数组，然后把记录插入到这个数组指针后。

**第十章 内部排序**

**插入排序**

**算法10.1 直接插入排序**

void InsertSort(Sqlist \*L){//插入排序

int i, j;

for(i=2; i<=L->length; i++){

if(L->key[i]<L->key[i-1]){ //将key[i]插入到有序子表

L->key[0] = L->key[i]; //复制哨兵

for(j=i-1; L->key[j]>L->key[0]; j--){

L->key[j+1] = L->key[j]; //记录后移

}

L->key[j+1] = L->key[0]; //插入到正确位置

}

}

}

**算法10.2 折半插入排序**

void BInsertSort(Sqlist \*L){//折半插入排序

int low, high, mid, i, j;

for( i=2; i<=L->length; i++){

if(L->key[i-1] > L->key[i]){

L->key[0] = L->key[i];

low = 1; high = i-1;

while(low <= high){

mid = (low +high) / 2;

if(L->key[mid] > L->key[0])

high = mid - 1;

else

low = mid + 1;

}

high = high + 1;

for(j=i-1; j>=high; j--)

L->key[j+1] = L->key[j];

L->key[high] = L->key[0];

}

}

}

**算法10.2 希尔排序**

先将整个待排序的记录分割成若干子序列分别进行直接插入排序，待整个序列中的记录基本有序时，再对全体记录进行一次直接插入排序。

void ShellSrot(Sqlist \*L, int dlta[], int t){//希尔排序

int k, i, j; //dlta[]是排序增量数组，t是取多少个数组元素

for(k=t-1; k>=0; k--){

for(i=dlta[k]+1; i<=L->length; i++){

if(L->key[i]<L->key[i-dlta[k]]){

L->key[0] = L->key[i];

for(j=i-dlta[k]; j>=0 && L->key[j]>L->key[0]; j = j-dlta[k]){

L->key[j+dlta[k]] = L->key[j];

}

L->key[j+dlta[k]] = L->key[0];

}

}

}

}

**快速排序（交换排序）**

**算法10.3 冒泡排序**

void Bubsort(Sqlist \*L){//冒泡排序

int i, j, tmp, flag;

for(i=1; i<=L->length-1; i++){

flag = 0;

for(j=1; j<=L->length-i; j++){

if(L->key[j+1] < L->key[j]){

flag = 1;

tmp = L->key[j+1];

L->key[j+1] = L->key[j];

L->key[j] = tmp;

}

}

}

}

**算法10.4快速排序**

通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

int Partion(Sqlist \*L, int low, int high){//查找中心位置

L->key[0] = L->key[low];

while(low < high && L->key[high] > L->key[0]) high--;

L->key[low] = L->key[high];

while(low < high && L->key[low] < L->key[0]) low++;

L->key[high] = L->key[low];

return low;

}

void QSort(Sqlist \*L, int low, int high){//快速排序

int pivotloc;

if(low < high){

pivotloc = Partion(L, low, high);

QSort(L, low, pivotloc-1);

QSort(L, pivotloc+1, high);

}

}

**选择排序**

**算法10.5 选择排序**

void SelectSort(Sqlist \*L){//选择排序

int i, j, k, tmp;

for(i=1; i<L->length; i++) {

k = i;

for(j=i; j<=L->length; j++){

if(L->key[j] < L->key[k]){

tmp = j;

j = k;

k = tmp;

}

}

if(k != i){

tmp = L->key[i];

L->key[i] = L->key[k];

L->key[k] = tmp;

}

}

}

**算法10.6 堆排序**

堆的定义如下：n个元素序列{K1，K2，···Kn}当且仅当满足以下关系时，称之为堆。

{Ki<=K2i，Ki<=K2i+1} 或 {Ki>=K2i，Ki>=K2i+1}

若将此序列看成相应的一维数组，则可以看成是一个完全是二叉树；则堆的含义表明，完全二叉树中所有的非终端结点的值均不大于（或不小于）其左右孩子结点的值。

void HeapAdjust(Sqlist \*L, int s, int n){//堆排序调整函数 ，调整为大根堆

int j, rc; //L包含一个一维数组和它的长度Length，s是根节点编号，n是数组长度

rc = L->key[s];

for(j=2\*s; j<=n; j=j\*2) {

if(j<n && L->key[j+1]>L->key[j]) j++; //j为左右孩子较大的下标

if(rc >= L->key[j]) break;

L->key[s] = L->key[j]; //把大于rc的孩子上移

s = j;

}

L->key[s] = rc; //最后移到根结点或不动

}

void Swap(int \*i, int \*j){

int t;

t = \*i;

\*i = \*j;

\*j = t;

}

void HeapSort(Sqlist \*L) {//堆排序函数

int i;

int n = L->length;

for( i=n/2; i>=1; i--){

HeapAdjust(L, i, n); //把L里的数组建成大根堆

}

for(i=n; i>1; i--){

Swap(&L->key[1], &L->key[i]); //将最后一个记录和第一个记录互换

HeapAdjust(L, 1, i-1); //将i-1个记录重新调整为大根堆

}

}

**各种排序方法比较**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 类别 | 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 |
| 最好情况 | 最坏情况 | 平均情况 | 辅助存储 |
| 插入排序 | 直接插入排序 | O（n） | O（n^2） | O（n^2） | O（1） | 稳定 |
| 希尔排序 | O（n） | O（n^2） | O（n^1.3） | O（1） | 不稳定 |
| 交换排序 | 冒泡排序 | O（n） | O（n^2） | O（n^2） | O（1） | 稳定 |
| 快速排序 | O（nlogn） | O（n^2） | O（nlogn） | O（nlogn） | 不稳定 |
| 选择排序 | 直接选择排序 | O（n^2） | O（n^2） | O（n^2） | O（1） | 不稳定 |
| 堆排序 | O（nlogn） | O（nlogn） | O（nlogn） | O（1） | 不稳定 |
| 归并排序 | | O（nlogn） | O（nlogn） | O（nlogn） | O（n） | 稳定 |
| 基数排序  k为待排元素的维数，m为基数的个数 | | O（n+m） | O（k\*(n+m)） | O（k\*(n+m)） | O（n+m） | 稳定 |