



**设 计 性 实 验**

课程名称 数据结构

题目名称 抽象数据结构之B树实现

学生学院 计算机学院

专业班级 网络工程1602

学 号 3116004979

学生姓名 詹泽霖

指导教师 李杨

2017 年 12 月 22日

**一.绪论**

**1.1 B树的定义**

B树是一种平衡的多路查找树。

一颗m阶B树，或为空树，或为满足下列特性的m叉树。

1. 树中每个结点最多含有m棵子树;
2. 若根结点不是叶子结点，则至少有两颗子树;
3. 除根之外的所有非终端结点至少有[m/2];
4. 每个非终端结点中包含信息：（n,A0,K1,A1,K2,A2,...,Kn,An)。其中

①Ki（1≤i≤n）为关键字，且关键字按升序排序。

②指针Ai（0≤i≤n）指向子树的根结点。

③关键字的个数n必须满足：[m/2]-1≤n≤m-1

1. 所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部节点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）

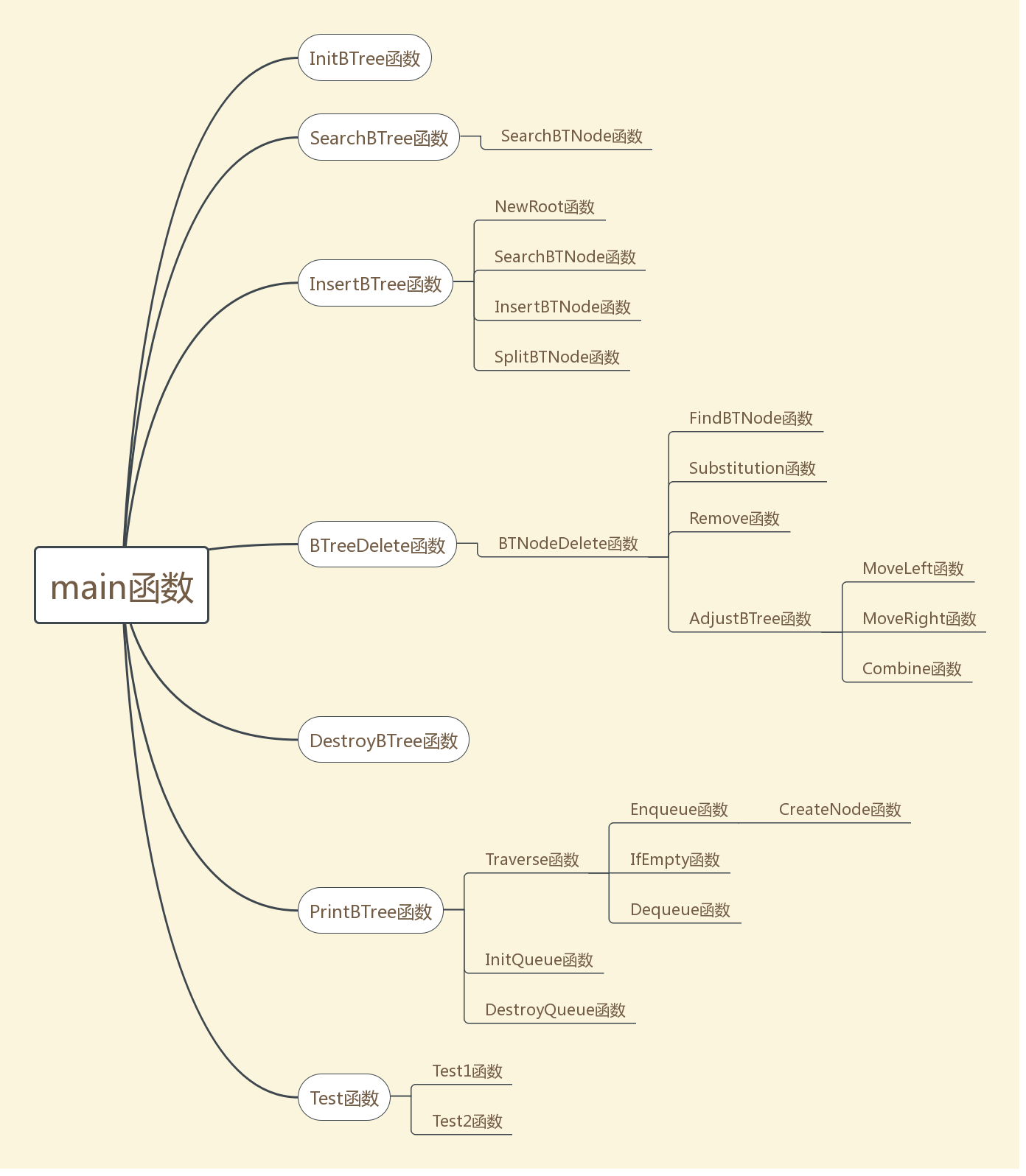
**1.2编程环境与配置**

IDE：Dev-C++ 5.11

编程语言：C

**二.B树的具体实现**

**2.1程序结构**

****

**2.2B树的抽象数据类型定义**

**ADT BTree{**

**数据对象：D是具有相同特性的数据元素的集合**

**数据关系：R1={<ai-1,ai>|ai-1,ai∈D，i=2，...,n}**

**R2={<ptr[i-1],ptr[i]>|i=1...,n}**

**约定a1|key[1]为关键字数组头，an|key[p-<keynum]为关键字数组尾**

**约定ptr[i]为结点的第i个子树**

**基本操作：**

**InitBTree(t)**

**初始条件：B树已定义**

**操作结果：初始化B树**

**SearchBTNode(BTNode \*p,KeyType k)**

**初始条件：结点p已存在**

**操作结果：在结点p中查找关键字k的插入位置i**

**Result SearchBTree(BTree t,KeyType k)**

**初始条件：B树已存在**

**操作结果：在B树查找关键字k的插入位置，返回查找结果**

**InsertBTNode(BTNode \*&p,int i,KeyType k,BTNode \*q)**

**初始条件：结点p和结点q已存在，0<i<p->keynum**

**操作结果：将关键字k和结点q分别插入到p->key[i+1]和p->ptr[i+1]中**

**SplitBTNode(BTNode \*&p,BTNode \*&q)**

**初始条件：结点p和结点q已存在**

**操作结果：将结点p分裂成两个结点,前一半保留,后一半移入结点q**

**NewRoot(BTNode \*&t,KeyType k,BTNode \*p,BTNode \*q)**

**初始条件：结点t,p,q已存在**

**操作结果：生成新的根结点t,原p和q为子树指针**

**InsertBTree(BTree &t,int i,KeyType k,BTNode \*p)**

**初始条件：结点p和结点t已存在，0<i<p->keynum**

**操作结果：在B树t中插入关键字k**

**Remove(BTNode \*p,int i)**

**初始条件：结点p已存在，0<i<p->keynum**

**操作结果：p结点删除key[i]和它的孩子指针ptr[i]**

**Substitution(BTNode \*p,int i)**

**初始条件：结点p已存在，0<i<p->keynum**

**操作结果：查找替代值**

**MoveRight(BTNode \*p,int i)**

**初始条件：结点p已存在，0<i<p->keynum**

**操作结果：结点调整右移操作**

**MoveLeft(BTNode \*p,int i)**

**初始条件：结点p已存在，0<i<p->keynum**

**操作结果：结点调整左移操作**

**Combine(BTNode \*p,int i)**

**初始条件：结点p已存在，0<i<p->keynum**

**操作结果：结点调整合并操作**

**AdjustBTree(BTNode \*p,int i)**

**初始条件：结点p已存在，0<i<p->keynum**

**操作结果：B树调整操作**

**BTNodeDelete(BTNode \*p,KeyType k)**

**初始条件：结点p已存在**

**操作结果：在结点p中删除关键字k**

**BTreeDelete(BTree &t,KeyType k)**

**初始条件：B树t已存在**

**操作结果：在B树t中删除关键字k**

**DestroyBTree(BTree &t)**

**初始条件：B树t已存在**

**操作结果：递归释放B树**

**PrintBTree(BTree t)**

**初始条件：B树t已存在**

**操作结果：遍历打印B树**

**}ADT BTree**

**2.3头文件**

定义了需要用到的数据类型，结构体类型，以及所有函数接口;

//==========ADT BTree的表示与实现==========

#ifndef \_BTREE\_H

#define \_BTREE\_H

#define MAXM 10 //定义B树的最大的阶数

const int m=4; //设定B树的阶数

const int Max=m-1; //结点的最大关键字数量

const int Min=(m-1)/2; //结点的最小关键字数量

typedef int KeyType; //KeyType为关键字类型

//===============B树存储结构==============

typedef struct node{ //B树和B树结点类型

int keynum; //结点关键字个数

KeyType key[MAXM]; //关键字数组，key[0]不使用

struct node \*parent; //双亲结点指针

struct node \*ptr[MAXM]; //孩子结点指针数组

}BTNode,\*BTree;

typedef struct{ //B树查找结果类型

BTNode \*pt; //指向找到的结点

int i; //在结点中的关键字位置;

int tag; //查找成功与否标志

}Result;

typedef struct LNode{ //链表和链表结点类型

BTree data; //数据域

struct LNode \*next; //指针域

}LNode, \*LinkList;

typedef enum status{ //枚举类型（依次递增）

TRUE,

FALSE,

OK,

ERROR,

OVERFLOW,

EMPTY

}Status;

//============基本操作的函数原型声明=============

Status InitBTree(BTree &t);

//初始化B树

int SearchBTNode(BTNode \*p,KeyType k);

//在结点p中查找关键字k的插入位置i

Result SearchBTree(BTree t,KeyType k);

/\*在树t上查找关键字k,返回结果(pt,i,tag)。若查找成功,则特征值

tag=1,关键字k是指针pt所指结点中第i个关键字；否则特征值tag=0,

关键字k的插入位置为pt结点的第i个\*/

void InsertBTNode(BTNode \*&p,int i,KeyType k,BTNode \*q);

//将关键字k和结点q分别插入到p->key[i+1]和p->ptr[i+1]中

void SplitBTNode(BTNode \*&p,BTNode \*&q);

//将结点p分裂成两个结点,前一半保留,后一半移入结点q

void NewRoot(BTNode \*&t,KeyType k,BTNode \*p,BTNode \*q);

//生成新的根结点t,原结点p和结点q为子树指针

void InsertBTree(BTree &t,int i,KeyType k,BTNode \*p);

/\*在树t上结点q的key[i]与key[i+1]之间插入关键字k。若引起

结点过大,则沿双亲链进行必要的结点分裂调整,使t仍是B树\*/

void Remove(BTNode \*p,int i);

//从p结点删除key[i]和它的孩子指针ptr[i]

void Substitution(BTNode \*p,int i);

//查找被删关键字p->key[i](在非叶子结点中)的替代叶子结点(右子树中值最小的关键字)

void MoveRight(BTNode \*p,int i);

/\*将双亲结点p中的最后一个关键字移入右结点q中

将左结点aq中的最后一个关键字移入双亲结点p中\*/

void MoveLeft(BTNode \*p,int i);

/\*将双亲结点p中的第一个关键字移入结点aq中，

将结点q中的第一个关键字移入双亲结点p中\*/

void Combine(BTNode \*p,int i);

/\*将双亲结点p、右结点q合并入左结点aq，

并调整双亲结点p中的剩余关键字的位置\*/

void AdjustBTree(BTNode \*p,int i);

//删除结点p中的第i个关键字后,调整B树

int FindBTNode(BTNode \*p,KeyType k,int &i);

//反映是否在结点p中是否查找到关键字k

int BTNodeDelete(BTNode \*p,KeyType k);

//在结点p中查找并删除关键字k

void BTreeDelete(BTree &t,KeyType k);

//构建删除框架，执行删除操作

void DestroyBTree(BTree &t);

//递归释放B树

Status InitQueue(LinkList &L);

//初始化队列

LNode\* CreateNode(BTree t);

//新建一个结点

Status Enqueue(LNode \*p,BTree t);

//元素q入队列

Status Dequeue(LNode \*p,BTNode \*&q);

//出队列，并以q返回值

Status IfEmpty(LinkList L);

//队列判空

void DestroyQueue(LinkList L);

//销毁队列

Status Traverse(BTree t,LinkList L,int newline,int sum);

//用队列遍历输出B树

Status PrintBTree(BTree t);

//输出B树

void Test();

//测试B树功能函数

#endif

**2.4B树具体接口实现**

**2.4.1InitBTree函数**

**功能:**初始化B树

**代码实现:**

**Status InitBTree(BTree &t){**

**t=NULL;**

**return OK;**

**}**

**2.4.2SearchBTNode函数**

**功能：**在结点p中查找关键字k的插入位置i

**代码实现：**

int SearchBTNode(BTNode \*p,KeyType k){

int i=0;

for(i=0;i<p->keynum&&p->key[i+1]<=k;i++);

return i;

}

**2.4.3SearchBTree函数**

**功能：在树t中查找关键字k，返回查找结果类型**

**代码实现：**

Result SearchBTree(BTree t,KeyType k){

BTNode \*p=t,\*q=NULL; //初始化结点p和结点q,p指向待查结点,q指向p的双亲

int found\_tag=0; //设定查找成功与否标志

int i=0;

Result r; //设定返回的查找结果

while(p!=NULL&&found\_tag==0){

i=SearchBTNode(p,k); //在结点p中查找关键字k if(i>0&&p->key[i]==k) //找到待查关键字

found\_tag=1; //查找成功

else{ //查找失败

q=p;

p=p->ptr[i];

}

}

if(found\_tag==1){ //查找成功

r.pt=p;

r.i=i;

r.tag=1;

}

else{ //查找失败

r.pt=q;

r.i=i;

r.tag=0;

}

return r; //返回关键字k的位置(或插入位置)

}

**2.4.4InsertBTNode函数**

**功能：**关键字k和结点q分别插入到p->key[i+1]和p->ptr[i+1]中

**代码实现：**

void InsertBTNode(BTNode \*&p,int i,KeyType k,BTNode \*q){

int j;

for(j=p->keynum;j>i;j--){ //整体后移空出一个位置

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

p->ptr[j+1]=p->ptr[j];

}

p->key[i+1]=k;

p->ptr[i+1]=q;

if(q!=NULL)

q->parent=p;

p->keynum++;

}

**2.4.5SplitBTNode函数**

**功能：**将结点p分裂成两个结点,前一半保留,后一半移入结点q

**代码实现：**

void SplitBTNode(BTNode \*&p,BTNode \*&q){

int i;

int s=(m+1)/2;

q=(BTNode \*)malloc(sizeof(BTNode)); //给结点q分配空间

q->ptr[0]=p->ptr[s]; //后一半移入结点q

for(i=s+1;i<=m;i++){

q->key[i-s]=p->key[i];

q->ptr[i-s]=p->ptr[i];

}

q->keynum=p->keynum-s;

q->parent=p->parent;

for(i=0;i<=p->keynum-s;i++) //修改双亲指针

if(q->ptr[i]!=NULL)

q->ptr[i]->parent=q;

p->keynum=s-1; //结点p的前一半保留,修改结点p的keynum

}

**2.4.6NewRoot函数**

**功能：**生成新的根结点t,原p和q为子树指针

**代码实现：**

void NewRoot(BTNode \*&t,KeyType k,BTNode \*p,BTNode \*q){

t=(BTNode \*)malloc(sizeof(BTNode)); //分配空间

t->keynum=1;

t->ptr[0]=p;

t->ptr[1]=q;

t->key[1]=k;

if(p!=NULL) //调整结点p和q的双亲指针

p->parent=t;

if(q!=NULL)

q->parent=t;

t->parent=NULL;

}

**2.4.7InsertBTree函数**

**功能：**在树t中插入关键字k，返回插入结果

**代码实现：**

void InsertBTree(BTree &t,int i,KeyType k,BTNode \*p){

BTNode \*q;

int finish\_tag,newroot\_tag,s; //设定需要新结点标志和插入完成标志

KeyType x;

if(p==NULL) //t是空树

NewRoot(t,k,NULL,NULL); //生成仅含关键字k的根结点t

else{

x=k;

q=NULL;

finish\_tag=0;

newroot\_tag=0;

while(finish\_tag==0&&newroot\_tag==0){

InsertBTNode(p,i,x,q); //将关键字x和结点q分别插入到p->key[i+1]和p->ptr[i+1]

if (p->keynum<=Max)

finish\_tag=1; //插入完成

else{

s=(m+1)/2;

SplitBTNode(p,q); //分裂结点

x=p->key[s];

if(p->parent){ //查找x的插入位置

p=p->parent;

i=SearchBTNode(p, x);

}

else //没找到x，需要新结点

newroot\_tag=1;

}

}

if(newroot\_tag==1) //根结点已分裂为结点p和q

NewRoot(t,x,p,q); //生成新根结点t,p和q为子树指针

}

}

**2.4.8Remove函数**

**功能：**从p结点删除key[i]和它的孩子指针ptr[i]

**代码实现：**

void Remove(BTNode \*p,int i){

int j;

for(j=i+1;j<=p->keynum;j++){ //前移删除key[i]和ptr[i]

p->key[j-1]=p->key[j];

p->ptr[j-1]=p->ptr[j];

}

p->keynum--;

}

**2.4.9Substitution函数**

**功能：**寻找替代值（右子树中最小的关键字）

**代码实现：**

void Substitution(BTNode \*p,int i){

BTNode \*q;

for(q=p->ptr[i];q->ptr[0]!=NULL;q=q->ptr[0]);

p->key[i]=q->key[1]; //复制关键字值

}

**2.4.10MoveRight函数**

**功能：**双亲结点p中的最后一个关键字移入右结点q中

将左结点aq中的最后一个关键字移入双亲结点p中

**代码实现：**

void MoveRight(BTNode \*p,int i){

int j;

BTNode \*q=p->ptr[i];

BTNode \*aq=p->ptr[i-1];

for(j=q->keynum;j>0;j--){ //将右兄弟q中所有关键字向后移动一位

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

q->ptr[j+1]=q->ptr[j];

}

q->ptr[1]=q->ptr[0]; //从双亲结点p移动关键字到右兄弟q中

q->key[1]=p->key[i];

q->keynum++;

p->key[i]=aq->key[aq->keynum]; //将左兄弟aq中最后一个关键字移动到双亲结点p中

p->ptr[i]->ptr[0]=aq->ptr[aq->keynum];

aq->keynum--;

}

**2.4.11MoveLeft函数**

**功能：**将双亲结点p中的第一个关键字移入左结点aq中，

将右结点q中的第一个关键字移入双亲结点p中

**代码实现：**

void MoveLeft(BTNode \*p,int i){

int j;

BTNode \*aq=p->ptr[i-1];

BTNode \*q=p->ptr[i];

aq->keynum++; //把双亲结点p中的关键字移动到左兄弟aq中

aq->key[aq->keynum]=p->key[i];

aq->ptr[aq->keynum]=p->ptr[i]->ptr[0];

p->key[i]=q->key[1]; //把右兄弟q中的关键字移动到双亲节点p中

q->ptr[0]=q->ptr[1];

q->keynum--;

for(j=1;j<=aq->keynum;j++){ //将右兄弟q中所有关键字向前移动一位

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

aq->ptr[j]=aq->ptr[j+1];

}

}

**2.4.12Combine函数**

**功能：**双亲结点p、右结点q合并入左结点aq，

并调整双亲结点p中的剩余关键字的位置

**代码实现：**

void Combine(BTNode \*p,int i){

int j;

BTNode \*q=p->ptr[i];

BTNode \*aq=p->ptr[i-1];

aq->keynum++; //将双亲结点的关键字p->key[i]插入到左结点aq

aq->key[aq->keynum]=p->key[i];

aq->ptr[aq->keynum]=q->ptr[0];

for(j=1;j<=q->keynum;j++){ //将右结点q中的所有关键字插入到左结点aq

aq->keynum++;

aq->key[aq->keynum]=q->key[j];

aq->ptr[aq->keynum]=q->ptr[j];

}

for(j=i;j<p->keynum;j++){ //将双亲结点p中的p->key[i]后的所有关键字向前移动一位

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

p->ptr[j]=p->ptr[j+1];

}

p->keynum--; //修改双亲结点p的keynum值

free(q); //释放空右结点q的空间

}

**2.4.13AdjustBTree函数**

**功能：**删除结点p中的第i个关键字后,调整B树

**代码实现：**

void AdjustBTree(BTNode \*p,int i){

if(i==0) //删除的是最左边关键字

if(p->ptr[1]->keynum>Min) //右结点可以借

MoveLeft(p,1);

else //右兄弟不够借

Combine(p,1);

else if(i==p->keynum) //删除的是最右边关键字

if(p->ptr[i-1]->keynum>Min) //左结点可以借

MoveRight(p,i);

else //左结点不够借

Combine(p,i);

else if(p->ptr[i-1]->keynum>Min) //删除关键字在中部且左结点够借

MoveRight(p,i);

else if(p->ptr[i+1]->keynum>Min) //删除关键字在中部且右结点够借

MoveLeft(p,i+1);

else //删除关键字在中部且左右结点都不够借

Combine(p,i);

}

**2.4.14BTNodeDelete函数**

**功能：**在结点p中查找并删除关键字k

**代码实现：**

int BTNodeDelete(BTNode \*p,KeyType k){

int i;

int found\_tag; //查找标志

if(p==NULL)

return 0;

else{

found\_tag=FindBTNode(p,k,i); //返回查找结果

if(found\_tag==1){ //查找成功

if(p->ptr[i-1]!=NULL){ //删除的是非叶子结点

Substitution(p,i); //寻找相邻关键字(右子树中最小的关键字)

BTNodeDelete(p->ptr[i],p->key[i]); //执行删除操作

}

else

Remove(p,i); //从结点p中位置i处删除关键字

}

else

found\_tag=BTNodeDelete(p->ptr[i],k); //沿孩子结点递归查找并删除关键字k

if(p->ptr[i]!=NULL)

if(p->ptr[i]->keynum<Min) //删除后关键字个数小于MIN

AdjustBTree(p,i); //调整B树

return found\_tag;

}

}

**2.4.15BTreeDelete函数**

**功能：**构建删除框架，执行删除操作

**代码实现：**

void BTreeDelete(BTree &t,KeyType k){

BTNode \*p;

int a=BTNodeDelete(t,k); //删除关键字k

if(a==0) //查找失败

printf(" 关键字%d不在B树中\n",k);

else if(t->keynum==0){ //调整

p=t;

t=t->ptr[0];

free(p);

}

}

**2.4.16DestroyBTree函数**

**功能：**递归释放B树

**代码实现：**

void DestroyBTree(BTree &t){

int i;

BTNode\* p=t;

if(p!=NULL){ //B树不为空

for(i=0;i<=p->keynum;i++){ //递归释放每一个结点

DestroyBTree(\*&p->ptr[i]);

}

free(p);

}

t=NULL;

}

**2.4.17InitQueue函数**

**功能：**初始化队列

**代码实现：**

Status InitQueue(LinkList &L){

L=(LNode\*)malloc(sizeof(LNode)); //分配结点空间

if(L==NULL) //分配失败

return OVERFLOW;

L->next=NULL;

return OK;

}

**2.4.18CreateNode函数**

**功能：**新建一个结点

**代码实现：**

LNode\* CreateNode(BTNode \*p){

LNode \*q;

q=(LNode\*)malloc(sizeof(LNode)); //分配结点空间

if(q!=NULL){ //分配成功

q->data=p;

q->next=NULL;

}

return q;

}

**2.4.19Enqueue函数**

**功能：**元素q入队列

**代码实现：**

Status Enqueue(LNode \*p,BTNode \*q){

if(p==NULL)

return ERROR;

while(p->next!=NULL) //调至队列最后

p=p->next;

p->next=CreateNode(q); //生成结点让q进入队列

return OK;

}

**2.4.20Dequeue函数**

**功能：**出队列，并以q返回值

**代码实现：**

Status Dequeue(LNode \*p,BTNode \*&q){

LNode \*aq;

if(p==NULL||p->next==NULL) //删除位置不合理

return ERROR;

aq=p->next; //修改被删结点aq的指针域

p->next=aq->next;

q=aq->data;

free(aq); //释放结点aq

return OK;

}

**2.4.21IfEmpty函数**

**功能：**队列判空

**代码实现：**

Status IfEmpty(LinkList L){

if(L==NULL) //队列不存在

return ERROR;

if(L->next==NULL) //队列为空

return TRUE;

return FALSE; //队列非空

}

**2.4.22DestroyQueue函数**

**功能：**销毁队列

**代码实现：**

void DestroyQueue(LinkList L){

LinkList p;

if(L!=NULL){

p=L;

L=L->next;

free(p); //逐一释放

DestroyQueue(L);

}

}

**2.4.23Traverse函数**

**功能：**用队列遍历输出B树

**代码实现：**

Status Traverse(BTree t,LinkList L,int newline,int sum){

int i;

BTree p;

if(t!=NULL){

printf(" [ ");

Enqueue(L,t->ptr[0]); //入队

for(i=1;i<=t->keynum;i++){

printf(" %d ",t->key[i]);

Enqueue(L,t->ptr[i]); //子结点入队

}

sum+=t->keynum+1;

printf("]");

if(newline==0){ //需要另起一行

printf("\n");

newline=sum-1;

sum=0;

}

else

newline--;

}

if(IfEmpty(L)==FALSE){ //l不为空

Dequeue(L,p); //出队，以p返回

Traverse(p,L,newline,sum); //遍历出队结点

}

return OK;

}

**2.4.24PrintBTree函数**

**功能：**输出B树

**代码实现：**

Status PrintBTree(BTree t){

LinkList L;

if(t==NULL){

printf(" B树为空树");

return OK;

}

InitQueue(L); //初始化队列

Traverse(t,L,0,0); //利用队列输出

DestroyQueue(L); //销毁队列

return OK;

}

**2.4.25Test1函数**

**功能：**测试B树功能

**代码实现：**

void Test1(){

system("color 70");

BTNode \*t=NULL;

Result s; //设定查找结果

int j,n=15;

KeyType k;

KeyType a[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

printf("创建一棵%d阶B树:\n",m);

for(j=0;j<n;j++){ //逐一插入元素

s=SearchBTree(t,a[j]);

if(s.tag==0)

InsertBTree(t,s.i,a[j],s.pt);

printf(" 第%d步,插入元素%d:\n ",j+1,a[j]);

PrintBTree(t);

}

printf("\n");

printf("删除操作:\n"); //删除操作

k=9;

BTreeDelete(t,k);

printf(" 删除%d:\n ",k);

printf(" 删除后的B树: \n");

PrintBTree(t);

printf("\n");

k=1;

BTreeDelete(t,k);

printf(" 删除%d:\n ",k);

printf(" 删除后的B树: \n");

PrintBTree(t);

printf("\n");

printf(" 递归释放B树\n"); //递归释放B树

DestroyBTree(t);

PrintBTree(t);

}

**2.4.26Test2函数**

**功能：**测试B树功能

**代码实现：**

void Test2(){

int i,k;

system("color 70");

BTree t=NULL;

Result s; //设定查找结果

while(1){

printf("此时的B树：\n");

PrintBTree(t);

printf("\n");

printf("=============Operation Table=============\n");

printf(" 1.Init 2.Insert 3.Delete \n");

printf(" 4.Destroy 5.Exit \n");

printf("=========================================\n");

printf("Enter number to choose operation:\_\_\_\_\_\b\b\b");

scanf("%d",&i);

switch(i){

case 1:{

InitBTree(t);

printf("InitBTree successfully.\n");

break;

}

case 2:{

printf("Enter number to InsertBTree:\_\_\_\_\_\b\b\b");

scanf("%d",&k);

s=SearchBTree(t,k);

InsertBTree(t,s.i,k,s.pt);

printf("InsertBTree successfully.\n");

break;

}

case 3:{

printf("Enter number to DeleteBTree:\_\_\_\_\_\b\b\b");

scanf("%d",&k);

BTreeDelete(t,k);

printf("\n");

printf("DeleteBTree successfully.\n");

break;

}

case 4:{

DestroyBTree(t);

break;

printf("DestroyBTree successfully.\n");

}

case 5:{

exit(-1);

break;

}

}

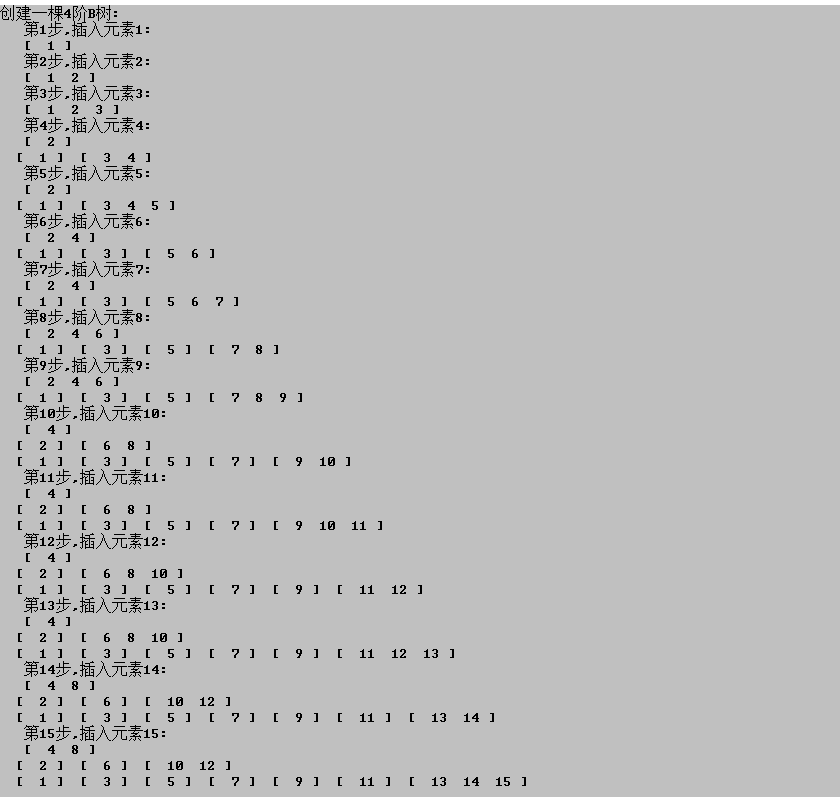
}

}

1. **功能测试**

**3.1插入功能测试/遍历功能测试**

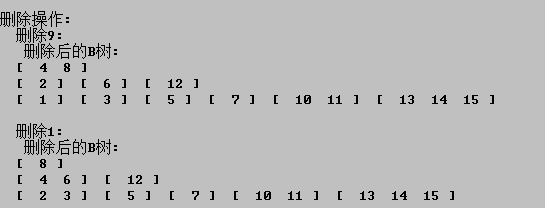
**依次插入1-15进行测试输出，结果如下：**



由输出的B树可知，插入功能正常并且遍历功能正常

**3.2删除功能测试**

在之前插入1-15后进行删除关键字的功能测试，选取9和1依次进行删除测试，结果如下：



根据B树的定义和该B树的输出，删除功能正常

**3.3释放功能测试**

**在之前的基础上进行递归释放B树功能测试，结果如下：**



遍历输出的结果为B数为空树，说明释放功能正常

**3.4其他功能测试**

其他接口在以上功能中已经有所体现，均正常，不再一一调用测试。

**四.思考与小结**

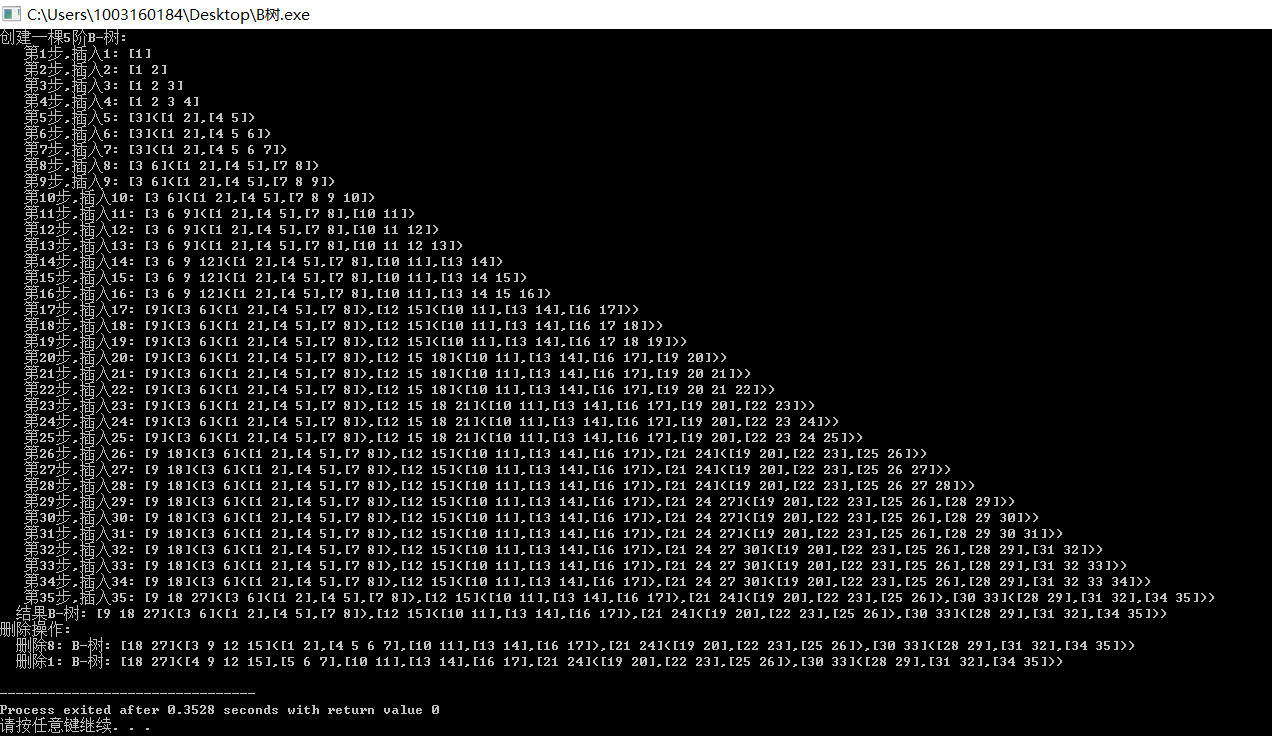
**4.1错误总结**

1. 在部分需要判空的地方没有判空
2. 递归实现的时候多次爆栈
3. 插入分裂的SplitBTNode函数一开始写的时候分裂成两个
4. 删除操作中的Combine函数的指针忘记调整

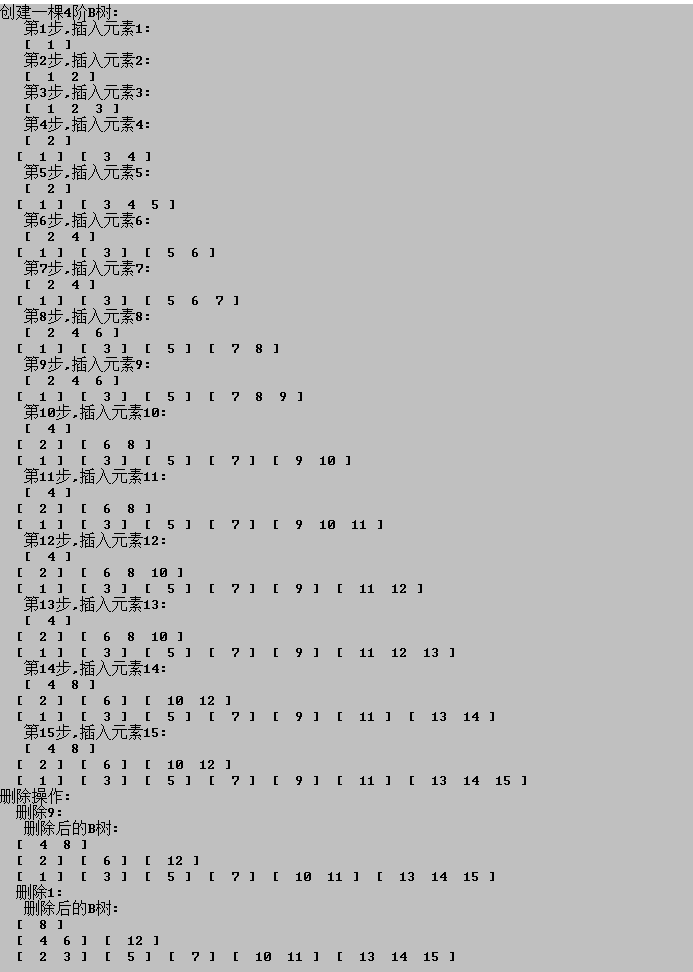
**4.2部分优化**

**4.2.1输出优化**

在一开始输出的时候选用的是括号输出法(测试功能选用的值略有不同),在直观上比较难的去分辨哪些是双亲结点的左右结点，因此在输出函数上进行了优化



通过队列遍历，在每一次遍历的过程中能够，模拟层次遍历，在B树的结构上更加美观，而且更容易看清楚B树的结构



**4.2.2测试界面优化**

**在保持原本接口不变的情况下，写了Test2函数，自行创建和进行各种B树的操作.**

