@[toc]

> 龙哥说考研是最有价值的投资，学校内的一些杂事，真的是一言难尽

第九次直播

二叉排序树

- 循环队列应该是可以只表示逻辑上的，但是看了很多教材，都是默认循环队列都采用数组来实现，所以循环队列是存储结构

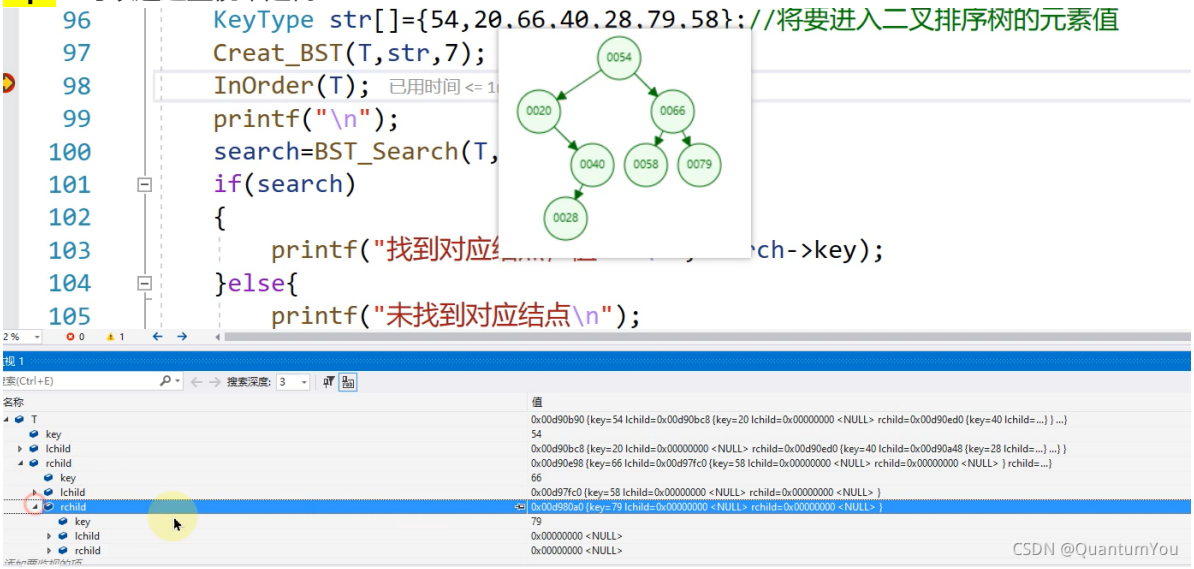
二叉排序树（也称二叉查找树）或者是一棵空树，或者是具有下列特性的二叉树：
1）若左子树非空，则左子树上所有结点的值均小于根结点的值。
2）若右子树非空，则右子树上所有结点的值均大于根结点的值。
3）左、右子树也分别是一棵二叉排序树。


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


typedef int KeyType;
typedef struct BSTNode{
    KeyType key;
    struct BSTNode *lchild,*rchild;
}BSTNode,*BiTree;
//54,20,66,40,28,79,58
int BST_Insert(BiTree &T,KeyType k)
{
    if(NULL==T)
    {   //为新节点申请空间
        T=(BiTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rchild=NULL;
        return 1;//代表插入成功
    }
    else if(k==T->key)
        return 0;//发现相同元素，就不插入
    else if(k<T->key)
        return BST_Insert(T->lchild,k);
    else
        return BST_Insert(T->rchild,k);
}
//创建二叉排序树
void Creat_BST(BiTree &T,KeyType str[],int n)
{
    T=NULL;
    int i=0;
    while(i<n)
    {
        BST_Insert(T,str[i]);
        i++;
```

```
    }
}
//递归算法简单，但执行效率较低，实现留给大家编写
BSTNode *BST_Search(BiTree T,KeyType key,BiTree &p)
{
    p=NULL;
    while(T!=NULL&&key!=T->key)
    {
        p=T;
        if(key<T->key) T=T->lchild;//比当前节点小，就左边找
        else T=T->rchild;//比当前节点大，右边去
    }
    return T;
}
//这个书上没有二叉排序树
void DeleteNode(BiTree &root,KeyType x){
    if(root == NULL){
        return;
    }
    if(root->key>x){
        DeleteNode(root->lchild,x);
    }else if(root->key<x){
        DeleteNode(root->rchild,x);
    }else{ //查找到了删除节点
        if(root->lchild == NULL){ //左子树为空
           BiTree tempNode = root;
           root = root->rchild;
           free(tempNode);
        }else if(root->rchild == NULL){ //右子树为空
           BiTree tempNode = root;//临时指针
           root = root->lchild;
           free(tempNode);
        }else{  //左右子树都不为空
            //一般的删除策略是左子树的最大数据 或 右子树的最小数据 代替要删除的节点(这里采用查找左子树最大数据来代替)
            BiTree tempNode = root->lchild;
            if(tempNode->rchild!=NULL){
                tempNode = tempNode->rchild;
            }
            root->key = tempNode->key;
            DeleteNode(root->lchild,tempNode->key);
        }
    }
}

void InOrder(BiTree T)
{
    if(T!=NULL)
    {
        InOrder(T->lchild);
        printf("%3d",T->key);
        InOrder(T->rchild);
    }
}
//《王道C督学营》课程
//二叉排序树的创建，中序遍历，查找，删除
int main()
{
```

```
    BiTree T;
    BiTree parent;//存储父亲结点的地址值
    BiTree search;
    KeyType str[]={54,20,66,40,28,79,58};//将要进入二叉排序树的元素值
    Creat_BST(T,str,7);
    InOrder(T);
    printf("\n");
    search=BST_Search(T,40,parent);
    if(search)
    {
        printf("找到对应结点，值=%d\n",search->key);
    }else{
        printf("未找到对应结点\n");
    }
    DeleteNode(T,66);
    InOrder(T);
    printf("\n");
    system("pause");
}
```

Tips：可以通过监视来建树



- 二叉排序树的最大查找为 这棵树的高度

二叉排序树的查找代码

以下代码有可能考察大题

```
BSTNode *BST_Search(BiTree T,KeyType key,BiTree &p)
{
    p=NULL;
    while(T!=NULL&&key!=T->key)
    {
        p=T;
        if(key<T->key) T=T->lchild;//比当前节点小，就左边找
        else T=T->rchild;//比当前节点大，右边去
    }
    return T;
}
```

二叉排序树的删除代码

- 当要删除的子树左右都有节点的时候，考虑的策略为删除左子树的最大值（即为左子树的最右节点）或右子树的最小值（即为右子树的最左节点）

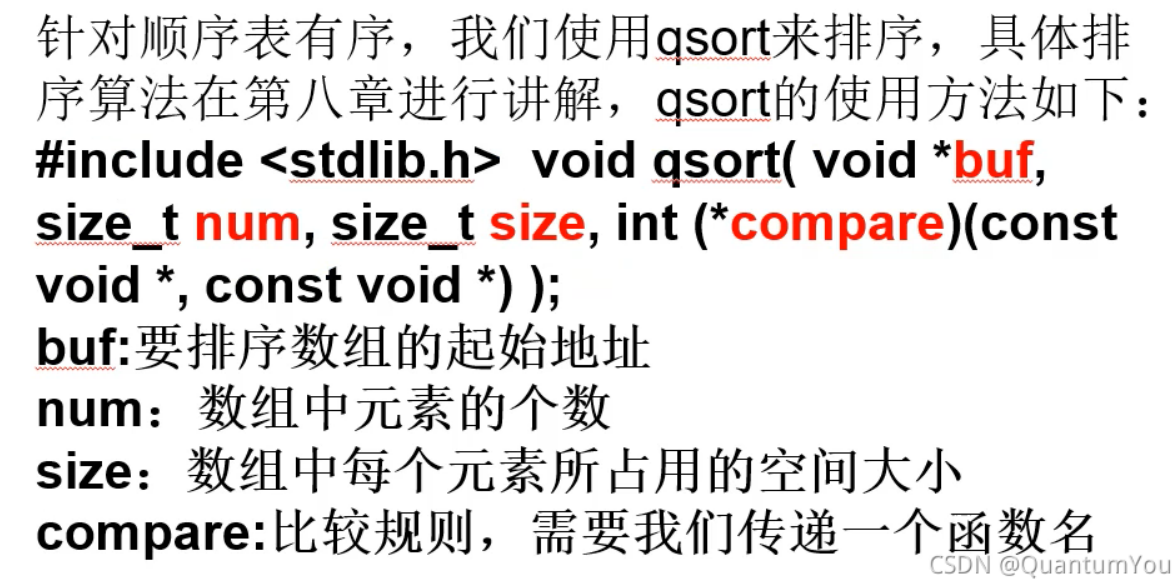
```
void DeleteNode(BiTree &root,KeyType x){
    if(root == NULL){
        return;
    }
    if(root->key>x){
        DeleteNode(root->lchild,x);
    }else if(root->key<x){
        DeleteNode(root->rchild,x);
    }else{ //查找到了删除节点
        if(root->lchild == NULL){ //左子树为空
           BiTree tempNode = root;
           root = root->rchild;
           free(tempNode);
        }else if(root->rchild == NULL){ //右子树为空
           BiTree tempNode = root;//临时指针
           root = root->lchild;
           free(tempNode);
        }else{  //左右子树都不为空
            //一般的删除策略是左子树的最大数据 或 右子树的最小数据 代替要删除的节点(这里采用查找左子树最大数据来代替)
            BiTree tempNode = root->lchild;
            if(tempNode->rchild!=NULL){
                tempNode = tempNode->rchild;
            }
            root->key = tempNode->key;
            DeleteNode(root->lchild,tempNode->key);
        }
    }
}
```

查找

顺序查找

- 顺序查找又称线性查找，它对于顺序表和链表都是适用的。对于顺序表，可通过数组下标递增来顺序扫描每个元素；对于链表，则通过指针next来依次扫描每个元素

qsort 排序接口介绍



二分查找网站

第十次直播

- 关于 OJ 逻辑正确，提交错误的描述

```
bool DeQueue(LinkQueue& Q, LinkDataType& x) {
    if (Q.front == Q.rear) {
        return false;
    }
    else
    {
        LinkNode* p = Q.front->next;//头结点什么都没存，所以头结点的下一个节点才有数据
        x = p->data;
        Q.front->next = p->next;
        if (Q.rear == p) {
            Q.rear = Q.front;
            free(p);
            return true;
        }
    }
    return true; //这里要注意，不加会wrong answer
}
```

由测试知识可知，OJ 是通过分支进行测试，所以注意不同分支的测试

哈希查找

哈希就是散列

- **散列函数**：一个把查找表中的关键字映射成该关键字对应的地址的函数，记为 `Hash(key)=Addr`(这里的地址可以是数组下标、索引或内存地址等)
  散列函数可能会把两个或两个以上的不同关键字映射到同一地址称这种情况为冲突

- **散列表**：根据关键字而直接进行访问的数据结构。也就是说，散列表建立了关键字和存储地址之间的一种直接映射关系。
  理想情况下，对散列表进行查找的时间复杂度为O(1),即与表中元素的个数无关

以下代码理解原理即可，不需要掌握编写

```
#define MaxKey 1000
#include <stdio.h>
//这就是哈希函数
int hash(const char* key)
{
    int h = 0, g;
    while (*key)
    {
        h = (h << 4) + *key++;
        g = h & 0xf0000000;
        if (g)
        {
            h ^= g >> 24;
        }
        h &= ~g;
    }
    return h % MaxKey;//算出下标要取余
}


int main()
{
    const char* pStr[5] = { "xiongda","lele","hanmeimei","wangdao","fenghua"
};
    int i;
    const char* pHash_table[MaxKey] = {NULL};//哈希表，散列表
    for (i = 0; i < 5; i++)
    {
        printf("%s is key=%d\n", pStr[i], hash(pStr[i]));//算哈希值并打印
        pHash_table[hash(pStr[i])] = pStr[i];//存入哈希表
    }
    return 0;
}
```

串的匹配

字符串匹配算法，暴力匹配算法代码如下

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef char* SString;
//暴力比对
//S  abcabaaabaabcac
//T  abaabcac
//为什么从1位置开始比较，因为0号位置存储了字符串的长度
int Index(SString S,SString T)
{
    int i=1,j=1;
    while(i<=S[0]&&j<=T[0])
```

```
    {
        if(S[i]==T[j])
        {
            ++i,++j;//继续比较后续字符
        }else{
            i=i-j+2;j=1;//指针后退重新开始匹配
        }
    }
    if(j>T[0]) return i-T[0];//匹配成功
    else return 0;
}
//i游标，遍历T，现在直播不懂那么别去调试get_next代码
void get_next(char T[],int next[])
{
    int i=1;
    next[1]=0;//恒为零
    int j=0;
    //abaabcac
    while(i<T[0])//T[0]中记录了字符串的长度
    {
        if(j==0||T[i]==T[j])//j==0，说明再次回到了开头
        {
            ++i;++j;
            next[i]=j;//记录出现重复的位置
        }else{
            j=next[j];//不相同，找个位置重新比较
        }
    }
}
//S  abcabaaabaabcac
//T  abaabcac
int KMP(char S[],char T[],int next[],int pos)
{
    int i=pos;//开始查找的起始位置
    int j=1;
    while(i<=S[0]&&j<=T[0])
    {
        if(j==0||S[i]==T[j]){//相等各自加加，往后走
            ++i;
            ++j;
        }
        else//不等，就回退next[j]的位置
            j=next[j];
    }
    if(j>T[0])//说明比对成功
        return i-T[0];
    else
        return 0;
}

//简单模式匹配 与  KMP（KMP考的概率极低）
int main()
{
    //字符串进行初始化
    char S[256];
    char T[10];
    int next[10];
    int pos;
```

```
    S[0]=strlen("abcabaaabaabcac");//strlen里有多少个字符
    strcpy(S+1,"abcabaaabaabcac");
    T[0]=strlen("abaabcac");
    strcpy(T+1,"abaabcac");
    pos=Index(S,T);//暴力匹配
    if(pos)
    {
        printf("匹配成功，位置为%d\n",pos);
    }else{
        printf("未匹配\n");
    }
    get_next(T,next);//算出next数组
    pos=KMP(S,T,next,1);
    if(pos)
    {
        printf("匹配成功，位置为%d\n",pos);
    }else{
        printf("未匹配\n");
    }
    system("pause");
}
```

KMP原理

（如图2所示）当c与b不匹配时，已匹配'abca'的前缀a和后缀a为最长公共元素。已知前缀a与b、c均不同，与后缀a相同，故无须比较，直接将子串移动“已匹配的字符数 - 对应的部分匹配值”，用子串前缀后面的元素与主串匹配失败的元素开始比较即可

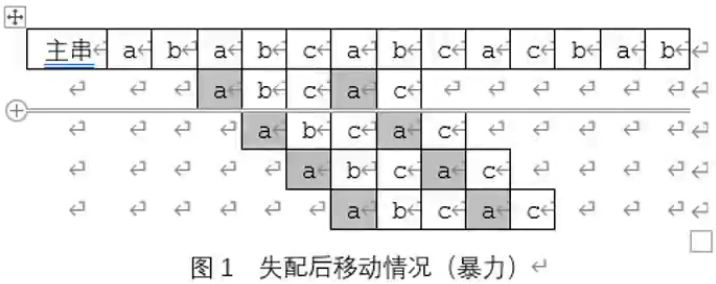


图 1　失配后移动情况（暴力）

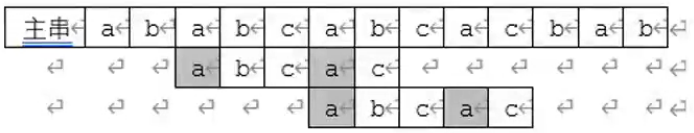


图 2　直接移动到合适位置