说明

1.此文件是814数据结构内容文档；

2.此文件包含以下几个部分：

(1)数据结构知识点

(2)数据结构基础代码

(3)知识点的应用(考法)

3.此资料内部使用，请勿外传，谢谢！

4.有任何问题请联系QQ：1932824470

5.西南科技大学814考研学习交流群：873339498

开始之前的必要准备工作

1.请将代码文件新建为.hpp结尾的文件(方便用C++的方法来测试C代码是否合格)，例如"Code.hpp"；

2.请在"Code.hpp"文件的最顶上写入如下内容后再开始写后续代码:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <string.h>
#include <string>
#include <map>

using namespace std;

typedef int DataType;
```

至此，欢迎你进入数据结构的强化阶段。

第一章 顺序表

顺序表的基本操作

```
/*顺序表的操作*/
//1.插入
//2.删除
//3.查找
//4.遍历

/*顺序表应用*/
//1.排序算法(在排序部分单独来做)
//2.二分查找:在长度为n的数组arr中查找val值的下标,未找到返回-1
//3.模拟双端栈(节省空间)、循环队列 -- 可以不用管
//4.反转顺序表的前x位和后n-x位,或反转整个顺序表
//5.删除有序表中的重复元素，并返回最后数组的长度
```

代码部分从下页开始

1.1.二分查找

```
//2.二分查找:在长度为n的数组arr中查找val值的下标,未找到返回-1
//C语言版:
DataType binarySearch_C(DataType*arr,DataType n,DataType val){
    DataType lchild=0,rchild=n-1;   //定左右边界
    while(lchild<=rchild){     //判断左右边界是否相等
        DataType mid=(lchild+rchild)/2; //计算中间值
        if(arr[mid]==val){  //找到val
            return mid;
        }else if(arr[mid]>val){ //中间值大于val,向左边查找
            rchild=mid-1;
        }else{              //中间值小于val,向右边查找
            lchild=mid+1;
        }
    }
    return -1;              //没找到,返回-1
}
//C++版:
DataType binarySearch_CPP(vector<DataType>arr,DataType val){
    DataType lchild=0,rchild=arr.size()-1;  //定左右边界
    while(lchild<=rchild){              //判断左右边界是否相等
        DataType mid=(lchild+rchild)/2;     //计算中间值
        if(arr[mid]==val){          //找到val
            return mid;
        }else if(arr[mid]>val){     //中间值大于val,向左边查找
            rchild=mid-1;
        }else{                      //中间值小于val,向右边查找
            lchild=mid+1;
        }
    }
    return -1;                      //没找到,返回-1
}
```

1.2.反转顺序表的前x位和后n-x位,或反转整个顺序表

```c
void reverseI2J(DataType*arr,int i,int j){  //反转顺序表的第i-j位元素,j是最右边元素的下标
    int L = i,R = j;
    while(L<R){ //互换i和j位置的元素即可
        DataType tmp = arr[L];
        arr[L++] = arr[R];
        arr[R--] = tmp;
    }
}
void reverseHeadX(DataType*arr,int n,int x,int mode){
    //其中mode = 1:反转前x位，mode = 2:反转前x位和后n-x位，mode = 3:反转整个顺序表
    //1.反转前x位
    reverseI2J(arr,0,x-1);
    if(mode == 1)return;
    //2.反转后n-x位
    reverseI2J(arr,x,n-1);
    if(mode == 2)return ;
    //3.反转整个顺序表
    reverseI2J(arr,0,n-1);
}
```

1.3.删除有序表中的重复元素，并返回最后数组的长度

```
int delDuplication(DataType*arr,int n){
    int i = 0,j = 1;    //i+1做最后的返回值，j代表当前待比较的最左边的元素
    for(;i < n;i++){
        while(arr[i] == arr[j])j++;//跳过重复元素
        arr[++i] = arr[j++];
    }
    return i+1;     //新数组的长度
}
```

第二章 链表

开始之前的准备工作

> 请在开始代码之前，将如下代码复制到"Code.hpp"中。

```
typedef struct linknode{    //单链表
    int data;               //数据域
    struct linknode*next;   //指针域
}LinkList,LinkListHead,CycleLinkList,CycleLinkListHead,LinkNode,CycleLinkNode;
typedef struct doublelinknode{      //双链表
    int data;               //数据域
    struct doublelinknode*next,*pre;//指针域
}DoubleLinkList,DoubleLinkListHead,DoubleLinkNode;
```

链表的基本操作

```
/*以下链表的插入分为头插和尾插*/
/*带\不带 头结点的 单\双\循环 链表的操作*/
//1.插入
//2.删除
//3.查找
//4.遍历

/*链表的应用*/
//1.反转单链表、反转双链表、反转循环单链表、反转循环双链表;
//2.递归/非递归删除值为val的元素
//3.打印两个有序单链表中的共同部分;
//4.判断单链表是否是回文单链表
//5.判断单链表是否有环,若有环则找出环的入口
//6.找两个单链表的公共后缀
//7.将单链表按某值划分成左边小,中间相等,右边大的形式;
//8.单链表设计循环队列
//9.找链表的中间结点
```

2.1.1.反转不带头结点的单链表

```
LinkList* reverseLinkList(LinkList*head){   //使用头插法
    if(!head)return NULL;     //空链表返回NULL
    LinkNode*cur = head,*next = cur->next;  //cur指向第一个结点,next指向下一个结点防止断链
    head = NULL;
    while(cur){               //当前未走到最后一个结点
        cur->next = head;     //头插,cur作为新的头结点,与head及其后边的结点建立连接
        head = cur;           //更新head为cur
        cur = next;           //cur继续往后走
        if(next)next = next->next;  //next仍然不是NULL时,就继续往后走防止断链
    }
    return head;              //返回新的head结点(因为要换头,所以必须返回head)
}
```

2.1.2.反转带头结点的单链表

```
bool reverseLinkListHead(LinkListHead*head){
    if(!head->next)return true; //空链表返回true,也可以返回false;
    LinkNode*cur = head->next,*next = cur->next;  //cur指向第一个结点,next指向下一个结点防止断链
    head->next = NULL;  //重置head的next指针,保持链表为空,进行头插
    while(cur){         //当cur不是最后一个结点时
        cur->next = head->next; //头插更新头结点head连接的第一个结点
        head->next = cur;       //更新head的next指针,指向新头插的结点cur
        cur = next;             //cur继续往后走,进行头插法
        if(next)next = next->next;  //若next未结束则继续指向后边结点,防止断链
    }
    return true;        //反转完毕,返回true
}
```

2.1.3.反转带头结点的单链表

```
CycleLinkList*reverseCyclyLinkList(CycleLinkList*head){ //使用头插法
    if(!head)return NULL;    //空链表返回NULL
    CycleLinkNode*cur_head = head;  //记录当前头结点,会变成尾结点
    CycleLinkNode*cur = head,*next = cur->next;  //cur指向第一个结点,next指向下一个结点防止断链
    head = NULL;
    while(1){
        cur->next = head;    //头插,cur作为新的头结点,与head及其后边的结点建立连接
        head = cur;          //更新head为cur
        if(next == cur_head) break; //若next指向头结点则跳出循环,此时cur指向原链表的尾结点
        cur = next;          //cur继续往后走
        next = next->next;   //next继续往后走防止断链
    }
    cur_head->next = cur;    //cur指向原链表的尾结点,此时cur_head指向原链表的头结点,next反转
    return head;             //返回新的head结点(因为要换头,所以必须返回head)
}
```

2.1.4.反转带头结点的单链表

```
bool reverseCyclyLinkListHead(CycleLinkListHead*head){
    if(!head->next)return true; //空链表返回true,也可以返回false
    CycleLinkNode*cur = head->next->next,*next = cur->next;  //cur指向第二个结点,next指向下一个结点防止断链
    head->next->next = head;  //重置head->next的next指针,保持链表为空,进行头插

    return true;        //反转完毕,返回true
}
```

2.1.5.反转带头结点的单链表

```
DoubleLinkList*reverseDoubleLinkList(DoubleLinkList*head){
    if(!head)return NULL;     //空链表返回NULL
    DoubleLinkNode*cur = head->next,*next = cur->next;  //cur指向第二个结点,next指向下一个结点防止断链
    head->next= NULL;         //调整head的next指针和pre指针
    head->pre = NULL;
    while(cur){               //当前未走到最后一个结点
        cur->next = head;     //头插,cur作为新的头结点,与head及其后边的结点建立连接
        cur->pre = NULL;      //更新cur的pre指针
        head->pre = cur;      //更新head的pre指针,指向新头插的结点cur
        head = cur;           //更新head为cur
        cur = next;           //cur继续往后走
        if(next)next = next->next;  //next仍然不是NULL时,就继续往后走防止断链
    }
    return head;              //返回新的head结点(因为要换头,所以必须返回head)
}
```

2.1.6.反转带头结点的单链表

```
bool reverseDoubleLinkListHead(DoubleLinkListHead*head){
    if(!head->next)return true; //空链表返回true,也可以返回false
    DoubleLinkNode*cur = head->next,*next = cur->next;  //cur指向第一个结点,next指向下一个结点防止断链
    head->next = NULL;  //重置head的next指针,保持链表为空,进行头插
    while(cur){         //开始头插
        cur->next = head->next; //修改cur指针的next
        cur->pre = head;        //修改cur指针的pre,先让其指向head
        if(head->next)          //若head->next不为空
            head->next->pre = cur;  //则让head->next的pre指向新的头插结点cur
        head->next = cur;       //将头插结点放入head->next
        cur = next;             //cur继续往后走
        if(next)next = next->next;      //next继续往后走防止断链
    }
    return true;        //反转完毕,返回true
}
```

2.2.1.递归删除不带头结点的单链表中值为val的元素

```
LinkList* deleteDuplication(LinkList*head,int val){
    if(!head)return NULL;   //结束条件
    LinkList*next = head->next; //next用于指向下一个结点,防止断链
    if(next->data == val) free(head);   //如果下一个结点的值等于val,则释放该结点
    return deleteDuplication(next,next->data);  //返回后续结点的递归检查结果
}
```

2.2.2. 非递归删除不带头结点的单链表中值为val的元素

```
LinkList* deleteDuplication2(LinkList*head,int val){
    if(!head)return NULL;   //空链表直接结束
    while(head && head->data==val){ //先固定头结点,找值不为val的结点做为新的头结点
        LinkList*tmp = head->next;  //tmp指向head->next防止断链
        free(head);                 //释放head
        head = tmp;
    }
    LinkList*pre = head;            //pre从head开始
    LinkList*cur = head?head->next:NULL; //cur指向已经固定的头结点的第二个结点
    LinkList*next = cur?cur->next:NULL;  //next用于指向下一个结点,防止断链
    while(cur){
        if(cur->data == val){       //cur结点值等于val
            pre->next = next;       //将pre->next指向next
            free(cur);              //释放cur
            cur = next;             //cur继续往后走
            if(next)next = next->next;
        }else{                      //cur结点值不等于val
            pre = cur;              //pre继续往后走
            cur = cur->next;        //cur继续往后走
            next = next?next->next:NULL;
        }
    }
    return head;                    //返回新的head
}
```

2.3.打印两个不带头结点的有序单链表中的共同部分

```
void getCommonList(LinkList*head1,LinkList*head2){
    if(!head1 || !head2)return ;     //空链表直接返回
    LinkList *L = head1,*R = head2; //L和R分别指向两个链表
    while(L && R){                   //开始循环判断
        if(L->data == R->data)       //若两个结点的值相等
            printf("%d ",L->data);
        if(L->data < R->data)        //若L结点的值小于R结点的值
            L = L->next;
            printf("%d ",L->data);
        if(L->data<R->data)L = L->next;
        if(L->data>R->data)R = R->next;
    }
    printf("\n");    //加个换行符,也可以不加,看题目要求
}
```

2.4.判断不带头结点的单链表是否是回文单链表

```c
bool isPalindrome(LinkList*head){   //使用快慢指针找中点,逆序后比较
    if(!head || !head->next)return false;  //空链表返回false
    //1.快慢指针找后续链表的中点
    LinkNode*fast = head,*slow = head;
    do{          //快指针走两步,慢指针走一步
        if(fast->next && fast->next->next)fast = fast->next->next;
        else break;
        slow = slow->next;
    }while(1);  //退出循环后slow->next指向的就是后半部分链表的第一个结点
    //2.反转slow->next开始的所有结点 - 为省空间,可将该过程写成函数进行调用
    LinkNode*cur = slow->next,*next = cur->next;
    slow->next = NULL;
    while(cur){ //开始翻转,使用头插法
        cur->next = slow->next;
        slow->next = cur;
        cur = next;
        if(next)next = next->next;
    }
    //3.开始比较,cur和next用来指向前后半部分的结点
    cur = head;          //指向前半部分
    next = slow->next;  //指向后半部分
    while(cur && next){ //任意一个走到NULL时停止
        if(cur->data != next->data)return false;
        cur = cur->next;
        next = next->next;
    }
    //4.比较完成,将后半部分链表重新反转回来
    cur = slow->next;
    next = cur->next;
    slow->next = NULL;
    while(cur){ //开始翻转,使用头插法
        cur->next = slow->next;
        slow->next = cur;
        cur = next;
        if(next)next = next->next;
    }
    return true;//比较结束,返回true
}
```

2.5.判断不带头结点的单链表是否有环,若有环则找出环的入口

```
LinkNode* isLoopList(LinkList*head){    //使用快慢指针
    if(!head)return NULL;  //空链表,无环,返回NULL
    LinkNode*slow = head,*fast = head;  //slow指向第一个结点,fast指向第一个结点
    do{
        slow = slow->next;  //慢指针走一步
        if(fast->next && fast->next->next)fast = fast->next->next;//快指针走两步
        else return NULL;   //无环直接返回NULL,只用判断快指针,因为快指针始终在慢指针前边
    }while(slow!=fast);     //快慢指针相遇,有环
    //快指针回到起点,慢指针不变,同时开始一步一步地走,相遇的结点即入口
    fast = head;
    while(fast != slow){    //fast和slow指针开始同步走
        fast = fast->next;
        slow = slow->next;
    }
    return fast;    //返回slow也可以,此时fast == slow
}
```

2.6.找两个不带头结点的单链表的公共后缀,返回起始节点

```
int getLinkListLength(LinkList*head){   //求链表长度
    int count = 0;
    LinkList*cur = head;
    while(cur){     //遍历一遍计数
        count++;
        cur = cur->next;
    }
    return count;
}
LinkNode*getLinkListRearNode(LinkList*head){    //找单链表的尾结点
    LinkList*p = head;
    while(p->next) p = p->next;
    return p;
}
LinkNode*getCommonFinalNode(LinkList*head1,LinkList*head2){
    //1.求两个链表的长度
    int len1 = getLinkListLength(head1);
    int len2 = getLinkListLength(head2);
    //2.求两个链表的尾结点
    LinkNode*tail1 = getLinkListRearNode(head1);
    LinkNode*tail2 = getLinkListRearNode(head2);
    //3.求公共部分
    if(tail1!=tail2)return NULL;    //尾结点不一样,说明没有公共后缀
    LinkNode*LongList = len1>len2?head1:head2;  //LongList指向更长的链表
    LinkNode*ShortList = LongList==head1?head2:head1;//ShortList指向更短的链表
    int delta = len1>len2?len1-len2:len2-len1;  //求长度差值
    while(delta--)  //长链表先走差值步
        LongList = LongList->next;
    while(LongList!=ShortList){ //两个链表同时走,直到走到公共起始节点
        LongList = LongList->next;
        ShortList= ShortList->next;
    }
    return LongList;//返回ShortList也可以,此时它们都指向公共后缀的第一个结点
}
```

2.7.将不带头结点的单链表按某值划分成左边小,中间相等,右边大的形式

```
LinkNode* partition(LinkList*head,int val){
    //六个变量,记录三个区间的头结点和尾结点,并初始化为NULL
    LinkNode*BigHead,*BigEnd,*EquHead,*EquEnd,*SmlHead,*SmlEnd;
    BigHead=BigEnd=EquHead=EquEnd=SmlHead=SmlEnd = NULL;
    LinkNode*cur = head;     //从head开始判断
    while(cur){              //将所有结点放到对应的区
        if(cur->data<val){   //放到小于区
            if(!SmlHead)     //小于区还未放任何结点
                SmlHead = SmlEnd = cur;
            else{            //否则放到最后
                SmlEnd->next = cur;
                SmlEnd = SmlEnd->next;
            }
        }else if(cur->data == val){ //放到等于区
            if(!EquHead)     //等于区还未放任何结点
                EquHead = EquEnd = cur;
            else{            //否则放到最后
                EquEnd->next = cur;
                EquEnd = EquEnd->next;
            }
        }else{               //放到大于区
            if(!BigHead)     //大于区还未放任何结点
                BigHead = BigEnd = cur;
            else{            //否则放到最后
                BigEnd->next = cur;
                BigEnd = BigEnd->next;
            }
        }
        cur = cur->next;     //cur继续往后走
    }
    head = SmlHead;          //head指向大于区的头结点
    SmlEnd->next = EquHead; //SmlEnd指向等于区的头结点
    EquEnd->next = BigHead; //EquEnd指向大于区的头结点
    return head;             //因为涉及到"换头"操作,所以要返回大于区的头结点作为新的头结点
}
```

2.8.1.单链表设计循环队列——入队操作

```c
//思路:使用循环单链表来做,为简单插入删除操作,只使用尾指针
//先设计数据结点及结构体
typedef struct queueNode{   //数据结点
    DataType data;          //数据域
    struct queueNode*next;  //指针域
}myQueueNode;
```

> 入队操作 - 队列尾进头出:

```c
bool EnQueue(myQueueNode**q,DataType data){ //此时的queue是尾结点
    //可能涉及"换尾"操作,因此queue的传入需要用二级指针
    if(!(*q)){  //空队,直接将元素插入即可
        *q = (myQueueNode*)malloc(sizeof(myQueueNode));
        (*q)->data = data;
        (*q)->next= (*q);   //单个节点时指向自己
        return true;
    }
    //队非空,则插入q->next即可
    myQueueNode*tmp = (myQueueNode*)malloc(sizeof(myQueueNode));
    tmp->data = data;
    tmp->next = (*q)->next; //tmp接上q的头结点
    (*q)->next = tmp;       //尾结点q指向tmp
    (*q) = tmp;             //更新tmp成为新的尾结点
    return true;
}
```

2.8.2.出队操作,值放到e中 - 队列头进尾出

```
bool DeQueue(myQueueNode**q,DataType*e){ //此时的queue是尾结点
    if(!(*q)) return false; //空队,出队失败
    *e = (*q)->next->data;  //队头元素出队
    if((*q)->next == (*q)){  //只有一个元素时
        free(*q);
        *q = NULL;
        return true;
    }
    //还有其余元素,则将队头出队
    myQueueNode*tmp = (*q)->next;    //将出队元素拿出来
    *e = tmp->data;          //取值放入e
    (*q)->next = tmp->next; //重新链接原链表
    free(tmp);               //释放内存空间
    return true;
}
```

2.8.3.取队头队尾及判空/清空队列 - 队列头进尾出

```
//取队头元素 - 队列头进尾出
bool GetHead(myQueueNode*q,DataType*e){
    if(!q) return false;
    *e = q->next->data;
    return true;
}
//取队尾元素 - 队列尾进头出
bool GetTail(myQueueNode*q,DataType*e){
    if(!q) return false;
    *e = q->data;
    return true;
}
//判断队列是否为空
bool isEmpty(myQueueNode*q){
    return !q;
}
//清空队列
void ClearQueue(myQueueNode**q){    //q指向尾结点
    if(!(*q)) return;
    myQueueNode*tmp = (*q)->next;   //从头结点开始删除
    while(tmp != (*q)){             //当未删到尾结点时
        (*q)->next = tmp->next;
        free(tmp);
        tmp = (*q)->next;
    }
    free(*q);       //单独删除尾结点即可
    *q = NULL;
}
```

2.8.4.找链表的中间结点 - 快慢指针

```
LinkNode* FindMid(LinkNode*head){
    if(!head)return NULL;
    LinkNode*fast = head,*slow = head;
    do{          //快指针走两步,慢指针走一步
        if(fast->next && fast->next->next)fast = fast->next->next;
        else break;
        slow = slow->next;
    }while(1);  //slow指向中间结点(奇数结点时)或中间结点的前一个结点(偶数结点时)
    return fast->next?slow->next:slow;  //当链表个数奇、偶数时的不同处理方法
}
```

第三章 栈

栈的基本操作

```
/*顺序栈、链栈的操作*/
//1.判空判满
//2.Push&Pop
//3.取栈顶Top

/*栈的应用*/
//注:树的非递归遍历等需要用到栈,但归结到树的知识点中去
//1.括号匹配·
//2.表达式求值
//3.中缀转后缀
```

3.1.括号匹配,思路：读到左括号入栈,读到右括号出栈并判断是否匹配

```
bool isMatch(char* str,int len){    //串的长度为len,不考虑字符串结尾的'\0'
    char*arr = (char*)malloc(sizeof(char)*len); //模拟栈
    for(int i = 0;i < len; i++)arr[i] = '\0';   //初始化栈元素
    int top = 0;    //标示栈顶
    for(int i = 0;i<len;i++){   //遍历串
        if(str[i] == '(' || str[i] == '[' || str[i] == '{') //左括号入栈
            arr[top++] = str[i];
        else if(str[i] == ')' || str[i] == ']' || str[i] == '}'){   //右括号则进行匹配
            switch(str[i]){     //匹配左括号与右括号
                case ')':       //左圆括号
                    if(arr[top-1] == '(')top--; //匹配成功,继续匹配
                    else return false;          //匹配失败,直接结束
                    break;
                case ']':       //左方括号
                    if(arr[top-1] == '[')top--; //匹配成功,继续匹配
                    else return false;          //匹配失败,直接结束
                    break;
                case '}':       //左花括号
                    if(arr[top-1] == '{')top--; //匹配成功,继续匹配
                    else return false;          //匹配失败,直接结束
                    break;
                default:        //其他情况,直接返回false
                    return false;
            }
        }
    }
    free(arr);                  //释放内存空间
    return top == 0?true:false; //遍历结束,且栈空,则匹配
}
```

3.2.表达式求值

```
int ExpressionEvaluate(char* str, int len) {
    char* sym = (char*)malloc(sizeof(char) * len); //符号栈
    int* num = (int*)malloc(sizeof(int) * len); //操作数栈
    for (int i = 0; i < len; i++) {//初始化栈元素
        sym[i] = '\0';
        num[i] = 0;
    }
    int top_sym = 0, top_num = 0;    //符号栈和操作数栈的栈顶
    for (int i = 0; i < len; i++) {     //遍历串,将所有符号和操作数入栈
        if (str[i] >= '0' && str[i] <= '9')        //数字进num栈
            num[top_num++] = str[i] - '0';
        else if (str[i] == '(')      //左括号直接进sym栈
            sym[top_sym++] = str[i];
        else if (str[i] == '+' || str[i] == '-') {//加减号要比较sym栈元素优先级
            if (top_sym == 0 || sym[top_sym - 1] == '(' || sym[top_sym - 1] == '\0')
                sym[top_sym++] = str[i];//优先级高,直接入栈即可
            else {   //优先级相等或更低,则一直出栈或走到'('处
                while (sym[top_sym - 1] != '('&& top_sym>0) {
                    int b = num[--top_num], a = num[--top_num];
                    switch (sym[--top_sym]) { //对a、b进行计算
                    case '*':num[top_num++] = a * b; break;
                    case '/':num[top_num++] = a / b; break;
                    case '+':num[top_num++] = a + b; break;
                    case '-':num[top_num++] = a - b; break;
                    }
                }
                sym[top_sym++] = str[i]; //当前符号入栈
            }
        }
        else if (str[i] == ')') {    //右括号直接出栈到左括号为止
            while (sym[top_sym - 1] != '(') {
                int b = num[--top_num], a = num[--top_num];
                switch (sym[--top_sym]) { //对a、b进行计算
                case '*':num[top_num++] = a * b; break;
                case '/':num[top_num++] = a / b; break;
                case '+':num[top_num++] = a + b; break;
                case '-':num[top_num++] = a - b; break;
                }
            }
            top_sym--; //左括号出栈
        }
        else if (str[i] == '*' || str[i] == '/')   //直接往栈里扔
            sym[top_sym++] = str[i];
    }
    while (top_sym > 0) {   //遍历串后对sym栈进行出栈
        int b = num[--top_num], a = num[--top_num];
        switch (sym[--top_sym]) { //对a、b进行计算
        case '*':num[top_num++] = a * b; break;
        case '/':num[top_num++] = a / b; break;
        case '+':num[top_num++] = a + b; break;
        case '-':num[top_num++] = a - b; break;
        }
    }
    free(sym);  //释放栈
```

```
    top_num = num[0];
    free(num);  //释放栈
    return top_num;
}
```

3.3.中缀转后缀

```c
//一种思路:只用一个栈,遇到数字就输出,遇到符号看情况输出
void InToPost(char* str,int len){
    char*sym = (char*)malloc(sizeof(char)*len); //符号栈
    for(int i=0;i<len;i++)sym[i]='\0';  //初始化栈
    int top = 0;    //标示栈顶
    for(int i = 0; i < len; i++){   //开始遍历字符串
        if(str[i]>='0' && str[i] <='9') printf("%c",str[i]);    //数字直接输出
        else if(str[i] =='+'||str[i] == '-'){   //加减则输出栈中其余符号,直到空或'('
            while(top>0 && sym[top-1]!='(')
                printf("%c",sym[--top]);
            sym[top++] = str[i];    //当前符号入栈
        }else if(str[i] == '*' || str[i] == '/' || str[i] == '('){//乘除括号直接入栈
            sym[top++] = str[i];
        }else if(str[i] == ')'){        //出栈直到左括号
            while(sym[top-1]!='(')
                printf("%c",sym[--top]);
            top--;  //左括号出栈
        }
    }
    while(top>0)    //剩余元素出栈
        printf("%c",sym[--top]);
}
```

第四章 队列

队列的基本操作

```
/*普通队列、双端队列、循环队列的操作*/
//1.入队出队
//2.判空判满
//3.取队头队尾元素

/*队列的应用*/
//基本上都是在二叉树层序遍历等算法中应用。
```

第五章 二叉树

开始之前的准备工作

> 请在开始代码之前，将如下代码复制到"Code.hpp"中。

```
typedef struct binarytreenode{  //二叉树结构体
    int data;           //数据域
    struct binarytreenode* lchild,*rchild;    //左孩子
    struct binarytreenode* left,*right;   //右孩子
    //以下是一些特殊的树用得上的东西,为了方便起见,就一起定义到这里边了
    struct binarytreenode* parent;   //父结点
    int ltag;       //左标志
    int rtag;       //右标志
    int level;      //层
    int height;     //高度
    int count;      //结点数
    bool balance;  //是否平衡
}BTree,BTNode;
#define MaxSize 1000
```

二叉树的基本操作

```
/*二叉树的操作*/
//1.递归遍历:前序、中序、后序;
//2.非递归遍历:前序、中序、后序、层序;
//3.实现二叉排序树
//4.实现平衡二叉树
//5.实现线索二叉树(非递归的Morris遍历)
//6.构造B树
//7.构造B+树
//8.构造哈夫曼树

/*二叉树的应用*/
//1.查找值为val的结点;
//2.求给定二叉树T的深度;
//3.求给定二叉树T的宽度(结点数最多的一层);
//4.找给定结点Node的双亲结点;
//5.翻转二叉树(所有左子树变右子树);
//6.判断二叉树是否为完全二叉树;
//7.判定给定的二叉树T是否是平衡二叉树;
//8.判定给定的二叉树T是否是二叉排序树;
//9.判定给定的二叉树T是否是满二叉树;
//10.找值为val的第一个结点
//11.统计二叉树结点个数
```

5.0.写一个构造二叉树的测试算法

```c
//用于测试后边的其他算法 - 将按层序给的数组arr转换成二叉树返回
BTree* CreateBTree(int*arr,int n){  //没有结点的位置值为-1
    if(!arr || !n)return NULL;    //空数组返回NULL
    BTNode*root = (BTNode*)malloc(sizeof(BTNode));
    root->data = arr[0];
    root->lchild = root->rchild = root->lchild = root->rchild = NULL;
    //构造队列层序添加结点
    BTNode**tmp = (BTNode**)malloc(sizeof(BTNode*)*MaxSize);
    int rear = 0,head = 0,top = 1;   //队列头尾浮标
    tmp[rear++] = root;
    while(top < n){ //所有数放好后结束
        BTNode*cur = tmp[head++];   //队头结点出队
        if(arr[top++] !=-1){//左结点
            cur->lchild = (BTNode*)malloc(sizeof(BTNode));
            cur->lchild->data = arr[top-1];
            cur->lchild->lchild = cur->lchild->rchild = NULL;
            tmp[rear++] = cur->lchild;  //左孩子入队
        }
        if(arr[top++] != -1){//右节点
            cur->rchild = (BTNode*)malloc(sizeof(BTNode));
            cur->rchild->data = arr[top-1];
            cur->rchild->lchild = cur->rchild->rchild = NULL;
            tmp[rear++] = cur->rchild;  //右孩子入队
        }
    }
    free(tmp);      //释放内存
    return root;    //返回构造好的头结点
}
```

5.1.1.递归遍历(先中后序)

```
void PreOrder_Iterator(BTree*root){  //先序
    if(!root)return;
    printf("%d ",root->data);       //打印根节点
    PreOrder_Iterator(root->lchild);  //打印左子树
    PreOrder_Iterator(root->rchild); //打印右子树
}
void InOrder_Iterator(BTree*root){  //中序
    if(!root)return;
    InOrder_Iterator(root->lchild);   //打印左子树
    printf("%d ",root->data);       //打印根节点
    InOrder_Iterator(root->rchild);  //打印右子树
}
void PostOrder_Iterator(BTree*root){//后序
    if(!root)return;
    PostOrder_Iterator(root->lchild); //打印左子树
    PostOrder_Iterator(root->rchild);//打印右子树
    printf("%d ",root->data);       //打印根节点
}
```

5.1.2.1.递归遍历一(层序)

```c
//层序遍历的两种实现思路:
//a.使用单链表模仿链队列:
typedef struct LevelNode{   //存储二叉树结点的结构
    BTree*node;             //存储二叉树节点
    struct LevelNode*next;  //存储下一个二叉树结点
}LevelNodeList;
void LevelOrder_IteratorList(BTree*root){//层序
    if(!root)return ;   //判误
    LevelNodeList*head = (LevelNodeList*)malloc(sizeof(LevelNodeList));//头结点
    head->node = root;
    head->next = NULL;
    LevelNodeList*cur = head,*rear = head;  //头尾节点,模仿队列
    while(cur){ //队列不空
        printf("%d\n",cur->node->data);     //输出队列结点
        if(cur->node->lchild){//左孩子有的话就入队
            LevelNode*n = (LevelNode*)malloc(sizeof(LevelNode));
            n->next = NULL;
            n->node = cur->node->lchild;
            rear->next = n;
            rear = rear->next;
        }
        if(cur->node->rchild){//右孩子有的话就入队
            LevelNode*n = (LevelNode*)malloc(sizeof(LevelNode));
            n->next = NULL;
            n->node = cur->node->rchild;
            rear->next = n;
            rear = rear->next;
        }
        cur = cur->next;    //继续往下一个结点去
    }
    //记得要销毁动态分配的堆区内存
    cur = head;
    rear = cur->next;
    while(cur){ //销毁LevelNodeList释放内存
        free(cur);
        cur = rear;
        if(rear)rear = rear->next;
    }
}
```

5.1.2.2.递归遍历二(层序)

```c
//b.使用指针数组实现队列:
int countTree(BTree*root);  //函数原型,声明是定义在下边应用题里的,这里要使用,就要提前定义
//考试的时候记得要把统计结点个数的代码写出来
void LevelOrder_IteratorArray(BTree*root){//层序
    if(!root)return ;   //判误
    //1.求二叉树的结点个数
    int num = countTree(root);
    BTree**arr = (BTree**)malloc(sizeof(BTree*)*num);  //申请二维数组
    int rear = 0,top = 0;
    arr[rear++] = root; //头结点入队
    while(top<rear){
        BTNode*cur = arr[top++];
        printf("%d ",cur->data);     //输出队列结点
        if(cur->lchild) arr[rear++] = cur->lchild;  //左孩子入队
        if(cur->rchild)arr[rear++] = cur->rchild; //右孩子入队
    }
    free(arr);  //记得释放内存
}
```

5.2.1.非递归遍历-总结及Morris本体

```
*  Morris算法-在时间复杂度O(n),空间复杂度O(1)的条件下遍历二叉树
    Morris还有个叫法->线索二叉树
    步骤:cur从root结点开始,当cur非空时:
    输出其值(进行访问);
    (1)cur无左子树时,cur = cur->rchild;
    (2)cur有左子树时找其左子树的最右结点R:
        (a)R->rchild==NULL,表示第一次访问cur结点,执行以下操作:
            R->rchild = cur;
            cur = cur->lchild;
        (b)R->rchild==root,表示第二次访问cur结点,执行以下操作:
            cur = cur->rchild;
    (3)cur==NULL时停止

    普通循环遍历-使用栈、队列等方式遍历二叉树
*/
```

> Morris算法本体如下:

```
void Morris(BTree*root){
    if(!root)return;    //空树返回
    BTNode*cur = root;  //cur从root开始
    while(cur){         //cur空则停止
        printf("%d\n",cur->data);   //输出cur的值-访问
        if(!cur->lchild)cur=cur->rchild;   //只能到一次的结点
        else{           //上边的情况(2)
            BTNode*R = cur->lchild;   //找左子树最右结点
            while(R->rchild && R->rchild!=cur)R=R->rchild;
            if(!R->rchild){  //情况(a),第一次访问
                R->rchild = cur;
                cur = cur->lchild;
            }else{          //情况(b),第二次访问
                R->rchild = NULL;
                cur = cur -> rchild;
            }
        }
    }
}
```

5.2.2.Morris实现先序和中序遍历

> 1.先序遍历:

```
//思路：输出第一次到达或只能到达一次的结点值
//与5.2.1相比，不同之处用*标出
void Morris_PreOrder(BTree*root){
    if(!root)return;     //空树返回
    BTNode*cur = root;  //cur从root开始
    while(cur){         //cur空则停止
        if(!cur->lchild){ //只能到一次的结点
            printf("%d\n",cur->data);    //*输出cur的值-访问
            cur = cur->rchild;
        }else{          //上边的情况(2)
            BTNode*R = cur->lchild;    //找左子树最右结点
            while(R->rchild && R->rchild != cur)R = R->rchild;
            if(!R->rchild){  //情况(a)，第一次访问
                R->rchild = cur;
                printf("%d\n",cur->data);    //*输出cur的值-访问
                cur = cur->lchild;
            }else{          //情况(b)，第二次访问
                R->rchild = NULL;
                cur = cur->rchild;
            }
        }
    }
}
```

> 2.中序遍历:

```
//思路：输出第二次到达或只能到达一次的结点值
//与2.1相比，不同之处用*标出
void Morris_MidOrder(BTree*root){
    if(!root)return;     //空树返回
    BTNode*cur = root;  //cur从root开始
    while(cur){         //cur空则停止
        if(!cur->lchild){ //只能到一次的结点
            printf("%d\n",cur->data);    //*输出cur的值-访问
            cur = cur->rchild;
        }else{
            BTNode*R = cur->lchild;    //找左子树最右结点
            while(R->rchild && R->rchild != cur)R = R->rchild;
            if(!R->rchild){  //情况(a)，第一次访问
                R->rchild = cur;
                cur = cur->lchild;
            }else{          //情况(b)，第二次访问
                R->rchild = NULL;
                printf("%d\n",cur->data);    //*输出cur的值-访问
                cur = cur->rchild;
            }
        }
    }
}
```

5.2.3.Morris实现后续遍历

```c
//思路:只在第二次到达的结点处逆序输出其左子树所有右结点，最后逆序输出整棵树右节点
//为保证空间复杂度为O(1)，逆序输出时采用反转单链表的方式进行
//2.4.1.反转结点cur的左子树所有右节点
void reverseRoot(BTNode*cur){
    if(!cur)return;
    BTNode*lchild = cur->lchild;//lchild作为头结点
    BTNode*R = lchild->rchild; //R指向右节点，防止断链
    cur->lchild = NULL;        //cur->lchild先置空
    while(lchild){             //lchild不空就继续执行
        lchild->rchild = cur->lchild;
        cur->lchild = lchild;
        lchild = R;
        if(R && R->rchild && R->rchild !=cur) R = R->rchild;
        else R = NULL;       //当R->rchild指向cur时保证R置空
    }
}
//2.4.2.Morris后续遍历算法
void Morris_PostOrder(BTree*root){
    if(!root)return;    //空树返回
    BTNode*cur = root;  //cur从root开始
    while(cur){
        if(!cur->lchild)cur = cur->rchild; //只访问一次的结点
        else{
            BTNode*R = cur->lchild;        //找左子树最右结点
            while(R->rchild && R->rchild!=cur) R=R->rchild;
            if(!R->rchild){  //第一次访问到cur结点
                R->rchild = cur;
                cur = cur->lchild;
            }else{           //第二次访问到cur结点，要逆序输出所有
                R->rchild = NULL;     //R->rchild先置空，方便反转
                reverseRoot(cur);   //反转cur的左子树
                R = cur->lchild;  //R作为临时指针用于访问所有右结点
                while(R){
                    printf("%d\n",R->data);
                    R = R->rchild;
                }
                reverseRoot(cur);   //再将cur的左子树反转回来
                cur = cur->rchild;
            }
        }
    }
    //逆序输出整棵树的右节点
    BTNode* tmp = (BTNode*)malloc(sizeof(BTNode));
    tmp->lchild = root;
    reverseRoot(tmp);   //先反转
    cur = tmp->lchild;
    while (cur) {
        printf("%d\n", cur->data);
        cur = cur->rchild;
    }
    reverseRoot(tmp);
    free(tmp);
}
```

5.3.1.常规非递归先序遍历二叉树 - 根 左 右

```
void PreOrderTree(BTree*root){
    if(!root)return ;
    BTNode**q = (BTNode**)malloc(sizeof(BTNode*)*MaxSize);    //结点栈
    int top = 0;    //栈顶
    q[top++] = root;
    while(top){     //栈空则结束
        BTNode*cur = q[--top];  //栈顶元素出栈
        printf("%d\n",cur->data);   //输出栈顶元素
        if(cur->rchild)q[top++] = cur->rchild;    //右结点入栈
        if(cur->lchild)q[top++] = cur->lchild;      //左结点入栈
    }
    free(q);    //释放内存
}
```

5.3.2.常规非递归中序遍历二叉树 - 左 根 右

```
/*
    思路:
    1.cur从root开始
    2.cur非空或栈非空时，执行以下操作
        (1)cur非空时入栈cur，cur=cur->left;
        (2)cur为空时出栈，并访问结点，接着cur=cur->right
*/
void InOrderTree(BTree*root){
    if(!root)return ;
    BTNode**q = (BTNode**)malloc(sizeof(BTNode*)*MaxSize);    //结点栈
    int top = 0;    //栈顶
    BTNode*cur = root;  //1.cur从root开始
    while(top || cur){  //cur非空或栈非空
        while(cur){     //(1).cur非空时入栈cur，并继续往左树走
            q[top++] = cur;
            cur = cur->lchild;
        }
        cur = q[--top];  //(2).cur为空时出栈，并访问结点
        printf("%d\n",cur->data);
        cur = cur->rchild;
    }
    free(q);    //释放内存
}
```

5.3.3.常规非递归后序遍历二叉树 - 左 右 根

```
/*
    思路:
    1.cur从root开始;
    2.cur非空或栈非空时，执行以下操作
        (1)cur非空时入栈cur，cur=cur->left;
        (2)cur为空时出栈，根据以下条件判断:
            a.若无右节点或右节点已访问过，则访问cur结点，并置cur = NULL;
            b.否则cur重新压栈，并继续往右树走
*/
void PostOrderTree(BTree*root){
    if(!root)return ;
    BTNode**q = (BTNode**)malloc(sizeof(BTNode*)*MaxSize);  //结点栈
    int*arr = (int*)malloc(sizeof(int)*MaxSize);    //记录结点值是否访问过，假设结点值<MaxSize
    memset(arr,0,MaxSize);  //将arr初始化为0
    int top = 0;    //栈顶
    BTNode*cur = root;  //1.cur从root开始
    while(top || cur){  //cur非空或栈非空
        while(cur){     //(1).cur非空时入栈cur，并继续往左树走
            q[top++] = cur;
            cur = cur->lchild;
        }
        cur = q[--top];  //(2).cur为空时出栈并进行下边的判断
        if(!cur->rchild || arr[cur->rchild->data]){  //a.若无右节点或右节点已访问过，则访问cur结点，并置cur = NULL
            printf("%d\n",cur->data);
            arr[cur->data] = 1;
            cur = NULL;
        }else{
            q[top++] = cur;  //b.否则cur重新压栈并继续往右树走
            cur = cur->rchild;
        }
    }
}
```

5.4.实现二叉排序树

```c
BTree* CreateBST(int*arr,int n){
    if(!arr || !n)return NULL;  //空数组返回NULL
    int cur = 0;    //当前正在构造的结点
    BTree*root = (BTNode*)malloc(sizeof(BTNode));   //根节点
    root->data = arr[cur++];
    root->lchild = root->rchild = NULL;
    BTNode*p = root;  //p从root开始找合适的位置插入结点
    while(cur < n){
        p = root;   //每轮循环都从root开始找
        BTNode*tmp = (BTNode*)malloc(sizeof(BTNode));   //创建新结点
        tmp->data = arr[cur++];
        tmp->lchild = tmp->rchild = NULL;
        while(p){
            if(tmp->data < p->data){    //新结点值更小，往左走
                if(p->lchild)p = p->lchild;
                else{
                    p->lchild = tmp;
                    break;
                }
            }
            if(p->rchild)p = p->rchild; //新结点值更大，往右走
            else{
                p->rchild = tmp;
                break;
            }
        }
    }
    return root;
}
```

5.n_说明

> 一些比较难的暂时没放代码，基本上814也考不到那么难，所以主要重点照顾这些给出来的代码。
>
> 二叉树的平衡二叉树、线索二叉树、构造B树、B+树及哈夫曼树都不做代码要求。
>
> 【下边从标号5.5开始，这些跳过的代码补在最后】

5.5.查找值为val的结点

```
int countTree(BTree*root);  //函数原型(因为定义在后边,所以要先声明一下)
BTNode* FindNode(BTree*root,int val){
    if(!root)return NULL;   //空结点返回NULL
    if(root->data == val)return root;       //当前结点值等于val,返回
    BTNode*lchild = FindNode(root->lchild,val); //递归查找左子树
    BTNode*rchild= FindNode(root->rchild,val);//递归查找右子树
    return lchild?lchild:rchild?rchild:NULL;      //返回左子树或右子树
}
```

5.6.求给定二叉树T的深度

```
int Depth(BTree*root){
    if(!root)return 0;   //空树深度为0
    int lchild = Depth(root->lchild);   //递归求左子树深度
    int rchild = Depth(root->rchild); //递归求右子树深度
    return lchild>rchild?lchild+1:rchild+1;
}
```

5.7.求给定二叉树T的宽度(结点数最多的一层)

```c
//思路：先求树T高度,再使用数组来统计每一层的结点数,这就需要使用辅助函数
void levelCount(BTree*root,int *depthArr,int level){
    //depthArr是统计的数组,level是当前所在层
    if(!root)return;   //空树返回
    depthArr[level]++; //统计当前层的结点数
    levelCount(root->lchild,depthArr,level+1); //递归左子树
    levelCount(root->rchild,depthArr,level+1); //递归右子树
}
int Width(BTree*root){
    //1.求树高度(深度)
    int depth = Depth(root);    //直接用上边现成代码了,考试就把上边Depth的代码抄到这儿
    //2.用数组记录每一层的结点数
    int *arr = (int*)malloc(sizeof(int)*depth);
    memset(arr, 0, depth);      //初始化为0
    //3.用辅助函数统计每一层的结点数 - root结点假设在第0层
    levelCount(root,arr,0);
    //4.找最大宽度
    int max = arr[0];
    for(int i = 1;i < depth;i++)
        if(arr[i]>max) max = arr[i];
    return max; //返回最大宽度
}
```

5.8.找给定结点Node的双亲结点

```
BTNode* FindParent(BTree*root,BTNode*node){
    if(!root)return NULL;   //空树返回NULL
    if(root == node)return node;    //当前结点是待查找node,则返回当前结点
    BTNode*lchild = FindParent(root->lchild,node);  //递归查找左子树
    if(lchild)return lchild == node?root:lchild;      //左子树是node,则root是其父结点,父结点肯定不是node
    BTNode*rchild = FindParent(root->rchild,node);//递归查找右子树
    if(rchild)return rchild == node?root:rchild;   //右子树是node,同上,直接返回root
    return NULL;
}
```

5.9.翻转二叉树(所有左子树变右子树)

```
void ReverseTree(BTree*root){
    if(!root)return;            //空树返回
    BTNode*lchild = root->lchild;   //接住左子树
    root->lchild = root->rchild;   //右子树换到左子树去
    root->rchild = lchild;          //左子树换到右子树去
    ReverseTree(root->lchild);    //左子树上继续转换
    ReverseTree(root->rchild);   //右子树上继续转换
}
```

5.10.判断二叉树是否为完全二叉树

```
bool IsCompleteTree(BTree*root){
    int flag = 0;   //表示不是完全二叉树
    //思路:新建辅助队列,用来层序遍历树,当遍历到第一个小于1个孩子的结点时,其余结点都必须是叶节点
    int num = countTree(root);  //先求结点个数
    BTree**arr = (BTree**)malloc(sizeof(BTree*)*num);
    for(int i = 0;i < num;i++)arr[i] = NULL;    //初始化队列
    int top = 0,rear = 0;   //表示队列的头尾
    arr[rear++] = root;     //根结点入队
    while(top < rear){      //队列非空时
        BTNode*cur = arr[top++];    //取队列头结点
        if(flag)            //此节点只能是叶节点
            if(cur->lchild||cur->rchild)return false;  //若有孩子则不是完全二叉树
        else {              //还未走到最后一个结点
            if(!cur->lchild || !cur->rchild)flag = 1;    //此结点是第一个小于1个孩子的结点
        }
        if(cur->lchild)arr[rear++] = cur->lchild; //左孩子入队
        if(cur->rchild)arr[rear++] = cur->rchild; //右孩子入队
    }
    return true;            //比较完成,该二叉树是完全二叉树
}
```

5.11.判定给定的二叉树T是否是平衡二叉树

```
bool IsBalanceTree(BTree*root){
    if(!root)return true;           //空树返回true
    int lchild = Depth(root->lchild);   //求左子树深度
    int rchild = Depth(root->rchild); //求右子树深度
    int delta = lchild>rchild?lchild-rchild:rchild-lchild;   //平衡因子
    return delta <= 1 && IsBalanceTree(root->lchild) && IsBalanceTree(root->rchild);
}
```

5.12.判定给定的二叉树T是否是二叉排序树

```
bool IsSortTree(BTree*root){
    if(!root)return true;   //空树返回true
    if(root->lchild)if(root->lchild->data>root->data)return false;//左子树大于根结点返回false
    if(root->rchild)if(root->rchild->data<root->data)return false;//右子树小于根结点返回false
    return IsSortTree(root->lchild)&&IsSortTree(root->rchild); //继续递归左右子树
}
```

5.13.判定给定的二叉树T是否是满二叉树-两种方法实现

```
//1.通过递归深度实现
bool IsFullTree(BTree*root){
    if(!root)return false;   //空树返回false
    int lchild = Depth(root->lchild);     //求左子树深度
    int rchild = Depth(root->rchild); //求右子树深度
    return lchild == rchild;
}
//2.通过统计高度实现
bool IsFullTree2(BTree*root){
    //1.求树深度
    int depth = Depth(root);
    //2.求深度为depth时的结点个数
    int num = 1;
    for(int i = 0;i < depth;i++)num*=2;
    num-=1;
    //3.判断是否满二叉树
    return num == countTree(root);
}
```

5.14.找值为val的第一个结点

```
BTNode* FindFirstNode(BTree*root,int val){
    if(!root)return NULL;   //空树返回NULL
    if(root->data == val)return root;       //当前结点值等于val,返回
    BTNode*lchild = FindFirstNode(root->lchild,val); //递归查找左子树
    BTNode*rchild= FindFirstNode(root->rchild,val);//递归查找右子树
    return lchild?lchild:rchild?rchild:NULL;      //返回左子树或右子树
}
```

5.15.统计二叉树结点个数

```
int countTree(BTree*root){
    if(!root)return 0;  //空树返回0
    int lchild = countTree(root->lchild);   //递归左子树
    int rchild= countTree(root->rchild);  //递归右子树
    return lchild+rchild+1;//返回左子树个数+右子树个数+当前结点
}
```

第六章 图

开始之前

> 本章一样只先给出最可能会考代码的部分，其余部分基本不会涉及代码问题，所以重点仍然是这些给了的代码。

图的基本操作

```
/*图的基本操作*/
//1.构造图的邻接矩阵
//2.构造图的邻接链表
//3.十字链表法构造图
//4.邻接多重表的构造
//5.图的深度优先遍历(DFS)和广度优先遍历(BFS)
//6.求图某一个结点的入度
//7.求图某一个结点的出度
```

图的数据结构定义

```
#define MAX_VERTICES 100
typedef struct {    //邻接矩阵的结构
    int vertices;   //图中顶点的数量
    int adjacencyMatrix[MAX_VERTICES][MAX_VERTICES];   //邻接矩阵
}adJacencyGraph;
```

6.1.1构造图的邻接矩阵

```
void initAdJacencyGraph(adJacencyGraph*graph,int vertices){
    graph->vertices = vertices; //初始化图的顶点数量
    for(int i = 0;i < vertices;i++)
        for(int j = 0;j < vertices;j++)
            graph->adjacencyMatrix[i][j] = 0;
    //2.用户输入邻接矩阵的值
    puts("图顶点一律从1开始.");
    for(int i = 0;i < graph->vertices;i++){
        for(int j = 0;j < graph->vertices;j++){//i==j时权值为0
            if(i == j) graph->adjacencyMatrix[i][j] = 0;
            else{
                printf("请输入顶点%d到顶点%d的权值:(没有路径填-1)",i+1,j+1);
                scanf("%d",&graph->adjacencyMatrix[i][j]);
            }
        }
    }
    puts("图的邻接矩阵构造完毕.");
}
```

6.1.2.输出邻接矩阵

```
void displayAdJacencyMatrix(const adJacencyGraph*const graph){
    if(!graph->vertices)return ;
    //1.输出表头
    printf("  ");
    for(int i = 0;i < graph->vertices;i++)
        printf("%d ",i+1);
    printf("\n");
    //2.输出数据
    for(int i = 0;i < graph->vertices;i++){
        printf("%d ",i+1);
        for(int j = 0;j < graph->vertices;j++){
            if(graph->adjacencyMatrix[i][j] == -1)printf("∞ ");
            else printf("%d ",graph->adjacencyMatrix[i][j]);
        }
        printf("\n");
    }
}
```

6.2.1.构造图的邻接链表

邻接链表的结构定义

```c
typedef struct adjacencyNode{   //顶点结点
    int vertex;     //顶点编号
    struct adjacencyNode*next;  //指向下一个顶点的指针
}AdjNode;           //图顶点的结构
typedef struct {    //邻接链表的结构
    int vertices;   //图中顶点的数量
    struct adjacencyNode*adjacencyList[MAX_VERTICES];   //邻接链表
    //传统邻接链表是用来记录无权图的,因此通常不具备查询权值的功能,但可以如下方式扩充:
    int EdgeArray[MAX_VERTICES][MAX_VERTICES];  //权重表-用来记录每两个顶点之间的权值
}AdjListGraph;      //邻接链表的结构
```

构造邻接链表

```c
void initAdjListGraph(AdjListGraph*graph,int vertices){ //初始化邻接链表
    graph->vertices = vertices; //初始化图的顶点数量
    for(int i = 0;i < vertices;i++) //初始化图的每个顶点
        graph->adjacencyList[i] = NULL;
    //2.用户输入邻接链表的值
    AdjNode*rear = NULL;        //用于找当前顶点的最后一个顶点
    puts("图顶点一律从1开始.");
    for(int i = 0;i < graph->vertices;i++){
        rear = graph->adjacencyList[i];
        while(rear->next)rear = rear->next;
        for(int j = 0;j < graph->vertices;j++){//i==j时权值为0
            if(i != j) {    //当结点不同时才在第i个顶点后边新增顶点
                printf("请输入顶点%d到顶点%d的权值:(没有路径填-1)",i+1,j+1);
                int weight = -1;
                scanf("%d",&weight);
                if(weight != -1){   //权值不为-1,则有路径相连,接入顶点
                    rear->next = (AdjNode*)malloc(sizeof(AdjNode));
                    rear->next->vertex = j; //添加新顶点的编号
                    rear->next->next = NULL;//新顶点的指针置为空
                    rear = rear->next;      //rear重新指向下一个顶点
                    graph->EdgeArray[i][j] = weight;//更新权重表
                }else graph->EdgeArray[i][j] = -1;   //权值为-1,只更新权重表
            }else graph->EdgeArray[i][j] = 0;   //i==j时也只更新权重表
        }
    }
    puts("图的邻接链表构造完毕.");
}
```

6.2.2.输出邻接链表

```c
void displayAdjListGraph(const AdjListGraph*const graph){
    if(!graph->vertices)return ;
    //1.输出邻接链表结构:
    for(int i = 0;i < graph->vertices;i++){      //一共有多个顶点
        printf("%d:",i+1);  //输出第i+1个顶点的链
        AdjNode*rear = graph->adjacencyList[i]; //rear从连接到第i个顶点的第一个顶点开始
        while(rear){         //依次往后输出每个顶点
            printf("%d->",rear->vertex);     //输出顶点
            rear = rear->next;
        }
        printf("\n");
    }
    //2.输出表头
    printf("   ");
    for(int i = 0;i < graph->vertices;i++)
        printf("%d ",i+1);
    printf("\n");
    //3.输出权重表的数据-本质就是遍历一个二维数组
    for(int i = 0;i < graph->vertices;i++){
        printf("%d ",i+1);
        for(int j = 0;j < graph->vertices;j++){
            if(graph->EdgeArray[i][j] == -1)printf("∞ ");
            else printf("%d ",graph->EdgeArray[i][j]);
        }
        printf("\n");
    }
}
```

第七章 串

串的考点如下

```
/*串的存储结构*/
//1.串的定长存储
//2.串的堆存储

/*串的基本应用*/
//1.给定子串s1,求s1的长度
//2.给定子串s1、s2,求s1在s2中第一次出现的下标
//3.给定子串s1、s2,求s1在s2中最后一次出现的下标
//4.KMP算法求s1在s2中第一次出现的位置
//5.KMP算法求s1在s2中最后一次出现的位置
```

7.1.给定子串s1,求s1的长度 - 用暴力破解法

```
int getStrLength(char*str){ //字符串以'\0'结尾
    int count = 0;
    while(str[count]!='\0')count++;
    return count;
}
```

7.2.给定串s1、s2,求s2在s1中第一次出现的下标

```
int locateStrFirst(char*s1,int n,char*s2,int m){ //在s2中找s1
    if(!s1 |||s2||n-m<0)return -1;  //判误
    for(int i = 0;i < n-m+1;i++){   //i是主串s1的当前位置
        int p = i,flag = 0; //flag表示是否找到子串s2
        for(int j = 0;j < m;j++){   //开始匹配主串与子串
            if(s1[p]!=s2[j]){       //不匹配
                flag = 1;
                break;
            }
            p++;            //匹配则p++
        }
        if(!flag)return i;  //找到子串,起始位置为i
    }
    return -1;  //查找完都没找到,返回-1
}
```

7.3.给定子串s1、s2,求s1在s2中最后一次出现的下标

```
int locateStrLast(char*s1,int n,char*s2,int m){
    if(!s1||!s2||n-m<0)return -1;   //判误
    int last = -1;  //用于记录最后一次匹配的位置
    for(int i = 0;i < n-m+1;i++){   //i是主串s1的当前位置
        int p = i,flag = 0; //flag表示是否找到子串s2
        for(int j = 0;j < m;j++){   //开始匹配主串与子串
            if(s1[p]!=s2[j]){       //不匹配
                flag = 1;
                break;
            }
            p++;            //匹配则p++
        }
        if(!flag) last = i; //记录当前i位置
    }
    return last;    //返回最后记录的匹配位置即可;
}
```

7.4.KMP算法求s1在s2中第一次出现的位置

```c
//1.求next数组
int* getNext(char*str,int n){   //KMP算法求next数组
    int *next = (int*)malloc(sizeof(int)*n);    //next数组
    //for(int i = 0;i<n;i++)next[i] = -1;   //初始化next数组,可有可无,反正下边会更新
    next[0] = -1;           //0位置的最长公共前后缀无意义,人为定义成-1
    next[1] = 0;            //1位置的最长公共前后缀只有一个字母,人为定义成0
    int i = 2,cmp = 0;      //从第i个位置开始求最长公共前后缀,cmp是与i位置进行比较的位置
    while(i<n && cmp<n){    //开始求next
        if(str[i-1] == str[cmp])//当str[i-1]与cmp位置字符相同时
            next[i++] = ++cmp;  //next值是cmp+1,因为cmp是前缀的最后一个位置,因此+1表示前缀的长度
        else if(cmp>0)cmp = next[cmp];  //当还能回溯时进行回溯
        else next[i++] = 0;     //最长公共前后缀是0
    }
    return next;
}
//2.KMP算法
int KMP(char*str,int n,char*sub_str,int m){
    if(!str || !sub_str)return -1;  //判误
    //1.拿next数组加速
    int *next = getNext(sub_str,m);
    //2.开始KMP:
    int i = 0,j = 0;    //分别是主串与子串的指针
    while(i<n && j<m){
        if(str[i] == sub_str[j]){   //相同往后走
            i++;
            j++;
        }else if(j>=0) j = next[j]; //回溯
        else {          //已经不能回溯了(j==-1),则从下一个位置开始比较
            i++;
            j++;
        }
    }
    return j==m?i-j:-1;
}
```

7.5.KMP算法求串最后一次出现的下标

```c
int KMPLast(char*str,int n,char*sub_str,int m){
    if(!str || !sub_str)return -1;  //判误
    //1.拿next数组加速
    int *next = getNext(sub_str,m);
    //2.开始KMP:
    int i = 0,j = 0;     //分别是主串与子串的指针
    int index = -1;      //记录最后结点
    while(i<n){
        if(str[i] == sub_str[j]){   //相同往后走
            i++;
            j++;
        }else if(j>=0) j = next[j]; //回溯
        else {          //已经不能回溯了(j==-1),则从下一个位置开始比较
            i++;
            j++;
        }
        if(j>m)break;   //子串已经比主串长,返回
        else if(j == m){
            index = i-j;//记录当前匹配的串的起始下标
            i = i-j+1;  //防止已匹配串里有后续匹配串中的字符
            j = 0;      //j回到开头
        }
    }
    return index;
}
```

第八章 散列表(Hash)

开始之前

> 本章不会考代码，只会考简答题，所以把简答题复习好就行

散列表的操作

```
/*哈希表的应用*/
//1.实现hash表-使用线性探测法(归属于开放定址法)
//2.实现hash表-使用平方探测法(归属于开放定址法)
//3.实现hash表-使用再散列法(归属于开放定址法)
//4.实现hash表-链地址法
```

第九章 排序算法

排序算法考点

```
//1.冒泡排序
//2.插入排序
//3.选择排序
//4.快速排序
//5.折半插入排序
//6.希尔排序
//7.归并排序
//8.堆排序
//9.计数排序
```

9.1.冒泡排序

```
bool BubbleSort(DataType*arr,int n){    //默认升序
    if(!arr)return false;
    DataType temp;          //临时变量
    for(int i = n-1;i>0;i--){       //外层循环
        for(int j = 0;j < i;j++){   //内层循环
            if(arr[j]>arr[j+1]){    //比较并交换
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    return true;
}
```

9.2.插入排序

```
bool InsertSort(DataType*arr,int n){    //默认升序
    if(!arr)return false;
    DataType temp;          //临时变量
    for(int i = 1;i<n;i++){         //外层循环
        for(int j = i;j>0;j--){     //内层循环
            if(arr[j]<arr[j-1]){    //比较并交换
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }else break;            //已排好序,退出循环
        }
    }
    return true;
}
```

9.3.选择排序

```
bool SelectSort(DataType*arr,int n){     //默认升序
    if(!arr)return false;
    for(int i = 0;i<n-1;i++){           //外层循环
        int min = i;                    //记录最小值的下标
        for(int j = i+1;j<n;j++){       //内层循环
            if(arr[j]<arr[min]){        //比较并交换
                min = j;
            }
        }
        if(min!=i){                     //交换
            DataType temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
    }
    return true;
}
```

9.4.快速排序

```
void partition(DataType*arr,int L,int R){//快排主过程
    if(L>=R)return;                 //已经有序,直接返回-递归结束条件
    int min = L-1,max = R+1;        //定义小于部分、大于部分
    int cur = L;                    //当前指针
    int baseVal = arr[L+(R-L)/2];   //取分类的基准值,该轮partition会将基准值放到合适的位置
    while(cur<max){
        if(arr[cur]<baseVal){       //小于基准值,放到小于部分
            DataType tmp = arr[++min];
            arr[min] = arr[cur];
            arr[cur++] = tmp;       //这里的cur++是因为cur前边换过来的值已经比较过了
        }else if(arr[cur]>baseVal){     //大于基准值,放到大于部分
            DataType tmp = arr[--max];
            arr[max] = arr[cur];
            arr[cur] = tmp;
            //这里不会cur++,因为从cur后边换过来的值还未进行比较
        }else cur++;                //等于基准值,直接cur++,即min到cur之间的值都等于baseVal
    }
    //通过上述步骤,将基准值放到合适的位置,接着再左右开弓,递归将小于区和大于区都分别进行partition
    partition(arr,L,min);           //让小于区也有序
    partition(arr,max,R);           //让大于区也有序
}
void quickSort(DataType*arr,int n){ //快排主函数
    partition(arr,0,n-1);   //调用快排主过程进行快排
}
```

9.5.折半插入排序

```c
int BinarySort(DataType*arr,int n,int val){    //默认升序-折半插入排序的折半查找,返回插入位置的下标
    if(n == 1) return arr[0]<val?1:0;  //只有一个元素时
    int L = 0,R = n-1;
    while(L<=R){    //二分查找
        int mid = (L+R)/2;
        if(arr[mid]<val) L = mid+1;     //小于就往右更大的地方找
        else if(arr[mid]>val) R = mid-1;//大于就往左更小的地方找
        else{       //相等就找第一个适合插入的位置-会破坏稳定性
            while(arr[mid--]==val);
            return ++mid;
        }
    }
    return L;   //返回L,L会始终指向正确的插入位置,可以手动模拟一下各种情况
}
void BinaryInsertSort(DataType*arr,int n){    //默认升序-折半插入排序
    for(int i = 1;i<n;i++){
        DataType val = arr[i];  //待查找值
        int index = BinarySort(arr,i,val); //折半查找前i个元素,找到插入位置的下标
        for(int j = i;j>index;j--) //移动元素
            arr[j] = arr[j-1];
        arr[index] = val;       //放置val
    }
}
```

9.6.希尔排序

```
void ShellSort(DataType*arr,int n){
    int gap = n/2;  //分组的间隔
    while(gap>0){
        for(int i=0;i<gap;i++){ //分gap组进行插入排序
            for(int j = i+gap;j<n;j+=gap){  //开始直接插入排序
                for(int k = j;k>0;k-=gap){
                    if(arr[k]<arr[k-gap]){  //比较并交换,默认升序
                        DataType temp = arr[k];
                        arr[k] = arr[k-gap];
                        arr[k-gap] = temp;
                    }else break;
                }
            }
        }
        gap/=2; //每次减少间隔
    }
}
```

9.7.归并排序

```c
void merge(DataType*arr,int lchild,int rchild){
    if(lchild>=rchild)return;       //递归结束条件
    int mid = (lchild+rchild)/2;    //中点
    merge(arr,lchild,mid);         //左边归并有序
    merge(arr,mid+1,rchild);      //右边归并有序
    int i = lchild,j = mid+1,k = 0;     //左右部分开始并的过程
    DataType *temp_arr = (DataType*)malloc(sizeof(DataType)*(rchild-lchild+1));//临时数组
    while(i<=mid&&j<=rchild){     //谁小谁先写
        if(arr[i]<arr[j]) temp_arr[k++] = arr[i++];
        else temp_arr[k++] = arr[j++];
    }
    while(i<=mid) temp_arr[k++] = arr[i++]; //将剩余元素补齐
    while(j<=rchild) temp_arr[k++] = arr[j++];//将剩余元素补齐
    for(int i = lchild;i<=rchild;i++) arr[i] = temp_arr[i-lchild];   //元素拷贝回arr中
}
void MergeSort(DataType*arr,int n){ //归并排序主函数
    merge(arr,0,n-1);   //调用merge函数进行归并
}
```

9.8.堆排序

```c
void HeapSort(DataType*arr,int n){//默认升序
    DataType*heap = (DataType*)malloc(sizeof(DataType)*n); //建立堆
    //1.先将用户数据转换成小根堆
    heap[0] = arr[0];
    for(int i=1;i<n;i++){   //从第二个元素开始往堆里扔
        heap[i] = arr[i];
        int fa = (i-1)/2;   //开始和父节点做比较
        for(int j = i;j>0;){
            if(heap[fa]>heap[j]){  //父结点更大就换下来
                DataType temp = heap[fa];
                heap[fa] = heap[j];
                heap[j] = temp;
                j = fa;
                fa = (fa-1)/2;
            }else break;            //否则比较结束
        }
    }
    //2.开始堆排序
    int count = n;  //用于记录当前堆个数
    int cur = 0;    //用于记录当前arr中元素的位置
    while(count>0){   //堆个数大于1时
        arr[cur++]=heap[0]; //堆顶最小元素放入arr中
        heap[0] = heap[--count];  //最后一个元素放入堆顶
        if(count == 0)break;
        //开始重新将堆调整为小根堆,向下比较
        int h_i = 0;  //h_i为堆顶,此时是待调整结点
        int L_child = 2*h_i+1,R_child=2*h_i+2;  //左右孩子
        while(h_i<count){       //当前结点仍在堆里
            if(L_child<count){  //当有左孩子时
                if(R_child>=count){  //当右孩子不存在时
                    if(heap[h_i]>heap[L_child]){  //左孩子更小时交换
                        DataType tmp = heap[h_i];
                        heap[h_i] = heap[L_child];
                        heap[L_child] = tmp;
                        h_i = L_child;
                    }
                }else{  //当heap[h_i]比左孩子或右孩子小时,更小的孩子上来
                    if(heap[L_child]<heap[R_child]){    //左孩子更小时
                        if(heap[h_i]>heap[L_child]){    //若heap[h_i]更大则交换
                            DataType tmp = heap[h_i];
                            heap[h_i] = heap[L_child];
                            heap[L_child] = tmp;
                            h_i = L_child;
                        }//否则就不管
                    }else{  //右孩子更小或相等时,统一换右孩子
                        if(heap[h_i]>heap[R_child]){    //若heap[h_i]更大则交换
                            DataType tmp = heap[h_i];
                            heap[h_i] = heap[R_child];
                            heap[R_child] = tmp;
                            h_i = R_child;
                        }
                    }
                }
            }
```

```
                break;//没有左孩子时直接进行下一轮循环
            }
        }
}
```

9.9.计数排序

```c
void CountSort(DataType*arr,int n){
    DataType max = arr[0];
    //1.找最大值
    for(int i=1;i<n;i++) if(arr[i]>max) max = arr[i];
    //2.申请辅助数组空间
    DataType *temp = (DataType*)malloc(sizeof(DataType)*(max+1));
    for(int i=0;i<max+1;i++) temp[i] = 0;   //初始化为0
    //3.统计数组
    for(int i=0;i<n;i++) temp[arr[i]]++;
    //4.拷贝回原数组
    int i = 0,j = 0;     //i标记arr当前长度,j标记当前temp的下标
    while(j<max+1){      //j从0到max+1依次查看temp数组
        while(temp[j]-->0)   //temp数组值非空时拷贝下标到arr中
            arr[i++] = j;
        j++;
    }
}
```

第十章 广义表

开始之前的说明

> 这章几乎不会考代码，主要考Head,Tail等操作，熟悉这些操作即可。

广义表的基本操作

```
/*广义表的基本操作*/
//1.广义表的顺序存储
//2.广义表的链式存储
//3.求表头
//4.求表尾
```

第十一章 查找

开始之前的说明

> 查找基本上就只考折半查找，诸如哈希等查找方法在"第八章 散列表中会涉及。

查找的基本操作

```
/*查找算法*/
//1.折半查找
//2.其他应用如哈希查找等,不在此处涉及
```

折半查找

```
//折半查找 - 只能找有序表
int BinarySearch(DataType*arr,int n,DataType val){
    DataType lchild=0,rchild=n-1;       //定左右边界
    while(lchild<=rchild){              //判断左右边界是否相等
        DataType mid=(lchild+rchild)/2;     //计算中间值
        if(arr[mid]==val){          //找到val
            return mid;
        }else if(arr[mid]>val){     //中间值大于val,向左边查找
            rchild=mid-1;
        }else{                      //中间值小于val,向右边查找
            lchild=mid+1;
        }
    }
    return arr[lchild]==val?lchild:-1;                  //没找到,返回-1
}
```

第十二章 扩展部分 - 可不学

说明

1.该部分是数据结构的扩展部分，可以不用学；

2.若时间足够，可做部分了解；

3.若时间充足，可学习以进一步增强自己的数据结构水平。

内容概要(目前暂时有的)

```
//1.并查集
//2.Manacher-回文子串算法 - 马拉车算法
```

12.1.并查集

并查集操作及必要数据结构定义

```c
/*并查集操作*/
//1.Union过程 - 将两个元素合并到同一个集合
//2.isSame查询过程 - 查询两个元素是否在同一个集合中
typedef struct UFSNode{ //并查集的结构
    DataType data;
    struct UFSNode *parent; //父结点
}UFS;
```

代码

```c
//先决条件,找结点node的最顶上的父结点
UFSNode*find(UFSNode*node){ //在找的过程中集成优化(路径压缩)过程
    //1.先找最顶上的父结点
    if(!node)return NULL;   //空结点返回NULL
    UFSNode*f = node;       //f从node开始
    while(f->parent!=f) f=f->parent;    //f往上继续找
    //2.优化guoc
    UFSNode*cur = node,*p = node->parent;
    while(cur!=f){          //扁平化,路径上所有node的parent都指向f
        cur->parent = f;
        cur = p;
        p = p->parent;
    }
    return f;
}
//1.Union过程
bool isSame(UFSNode*node1,UFSNode*node2);
bool Union(UFSNode*node1,UFSNode*node2){    //将两个结点node1与node2合并到同一个集合
    //1.求node1与node2是否同一个集合:
    if(isSame(node1,node2))return true;
    //2.合并 - 由于在查找过程中会进行优化(路径压缩)操作,因此将一个父结点并到另一个父结点上即可
    node2->parent = node1;
    //3.检查是否合并成功
    return isSame(node1,node2);
}
//2.isSame查询过程
bool isSame(UFSNode*node1,UFSNode*node2){
    //1.找两个结点的父结点 - 集成了优化过程
    UFSNode*f1 = find(node1),*f2 = find(node2);
    //2.比较两个结点的parent是否相同
    return f1 == f2?true:false;
}
```

12.2.1.Manacher-找最长回文子串-说明

```
/*
    说明：Manacher(马拉车)算法用于在字符串str中找出最长的回文子串
    1.i>R:暴力扩,narr[i] = 2j-1,c=i,R=i+(j-1)//j从1开始
    2.i≤R:找i关于c的对称点i1,L=2c-R,L1=i1-narr[i1]/2;
        2.1.i1的范围在L-R内:narr[i]=narr[i1];
        2.2.i1的左边界等于L:找R关于i的对称点R1,R1=2i-R;
          a.R+j与R1-j相等,继续扩(j从1开始)
          b.不相等或出界时:narr[i]=narr[i1]+(j-1)*2
          R=R+j-1,c=i
        2.3.i1左边界<L:
        j=R-i+1,当i+j<n时：从j开始比较str[i-j]与str[i+j]
        相等就j++,否则narr[i] = 1+(j-1)*2;
*/
```

12.2.2.Manacher-代码

```
bool Manacher(char*str,int n,int*index,int*len){
    //index表示每个回文串的中点下标,len表示回文串长度
    if(!str || n<=0)return false;
    //处理字符串,如将1221变成#1#2#2#1#,即在开始、结束和每个字符之间加入分隔符#
    char *s = (char*)malloc(sizeof(char)*(n*2+1));
    for(int i = 0;i < 2*n+1;i++){
        if(i%2==0)s[i] = '#';   //偶数位放'#'
        else s[i] = str[i/2];   //奇数位放str中的值
    }
    // //调试
    // for(int i = 0;i < 2*n+1;i++)cout<<s[i]<<" ";
    // cout<<endl;
    // //调试
    int *narr = (int*)malloc(sizeof(int)*n);   //存储每个字符的回文半径
    int i = 0,j = 0,c = -1,R = -1;  //含义见上方注释
    while(i<2*n+1){ //开始求narr数组
        // //调试
        // cout<<"当前走到"<<s[i]<<":"<<endl;
        // cout<<"i = "<<i<<",j = "<<j<<",c = "<<c<<",R = "<<R<<"narr[i] = "<<narr[i]
<<endl;
        // //调试
        if(i>R){//1.看i是否大于R,大于R则暴力扩展
            j = 1;
            while(i-j>=0 && i+j<2*n+1 && s[i-j]==s[i+j])
                j++;
            narr[i] = 2*j-1;
            if(R<i+j-1){
                R = i+j-1;
                c = i;
            }
        }else{  //2.i在R内,开始找对称点并做判断:
            int i1 = 2*c-i; //i关于c的对称点
            int L = 2*c-R;  //当前最长子串的左边界
            int L1 = i1-narr[i1]/2;  //i1的左边界
            int R1 = 2*i-R;  //找R关于i的对称点
            if(L1>L)    //2.1.i1的范围在L-R内
                narr[i] = narr[i1];
            else if(L1==L){ //2.2.i1的左边界等于L
                j = 1;
                while(R1-j>=0 && (R+j<i+j)&&(i+j<2*n+1) && s[R1-j]==s[R+j])  //相等继续扩
                    j++;
                narr[i] = 2*(j-1)+narr[i1];   //否则结束扩展
                if(R<R+j-1){//若右边界变化,则更新所有相关内容
                    R = R+j-1;
                    c = i;
                }
            }else{      //2.3.i1的左边界大于L
                j = 1;
                while(R1-j>=0 && (R+j<i+j)&&(i+j<2*n+1) && s[R1-j]==s[R+j])  //相等继续扩
                    j++;
                narr[i] = 2*(j-1)+narr[i1];   //否则结束扩展
                if(R<R+j-1){//若右边界变化,则更新所有相关内容
                    R = R+j-1;
                    c = i;
```

```
                }
            }
        }
        // //调试
        // cout<<"当前结尾"<<s[i]<<":"<<endl;
        // cout<<"i = "<<i<<",j = "<<j<<",c = "<<c<<",R = "<<R<<"narr[i] = "<<narr[i]
<<endl;
        // //调试
        i++;    //往后继续比较
    }
    //找出最长回文串
    *index = -1;
    *len = -1;
    //调试
    // cout<<"***************"<<endl;
    // for(int i = 0;i < 2*n+1;i++)cout<<narr[i]<<" ";
    // cout<<"***************"<<endl;
    //调试
    for(int i = 1;i < 2*n+1;i+=2){
        if(narr[i]>*len){
            *index = i/2;
            *len = narr[i]/2;
        }
    }
    return true;
}
```