第四章 串

•定义：串，即字符串，是由0个或多个字符组成的有限序列

字串：串中任意个连续的字符组成的子序列

主串：包含子串的串

字符在主串中的位置：字符在串中的序号

子串在主串中的位置：子串的第一个字符在主串中的位置

空串（长度为0）空格串

•基本操作

赋值操作 StrAssing（&T，chars）；

复制操作 StrCopy（&T，S）；

判空操作 StrEmpty（S）；

求串长 StrLength（S）；

清空操作 Clear String（&S）；

销毁串 DestroyString(&S);

连接串（&T，S1，S2）；

定位操作 Index（S，T）；

比较操作 StrCompare（S，T）；

•串的存储结构

1. 串的顺序存储

```
// 顺序存储
//静态数组（定长顺序）实现
#define  MAXLEN 255
typedefstruct{
        char ch[MAXLEN];//每个分量存储一个字符串
        int length;//串的实际长度
}SString;

//动态数组（堆分配）实现
typedefstruct{
    char *ch;//按照串长分配存储区，ch指向串的基地址
    int length;//串的长度
}HString;

HString S;

S.ch = (char *)malloc(MAXLEN *sizeof(char));//malloc按需申请，用完记得free
S.length = 0;
```

2. 串的链式存储

```
//链式存储
//低密度存储
typedef structStringNode{
   char ch;
   struct StringNode *Next;
}StringNode, *String;

//存储密度提高
typedef structStringNode{
   char ch[4];
   struct StringNode *next;
}StringNode, *String;
```

•基本操作实现

1. 求子串：用Sub返回串S的第pos个字符起长度为len的子串

```
//求子串
#define MAXLEN 255
typedef struct{
   char ch[MAXLEN];//每个分量存储一个字符
   int length;//串的实际长度
}SString;
bool SubString(SString &Sub,SString S,int pos,int len){
   //子串范围越界
   if(pos+len-1>S.length)
      return false;
   for(int i=pos;i<pos+len;i++){
      Sub.ch[i-pos+1] = S.ch[i];
   }
   Sub.length =len;
   return
}
```

2. 比较操作:比较s和T，若S>T，返回值大于0，若s=T，返回值等于0，否则小于0

```
//比较操作
int StrCompare(SString S,SString T){
   //比较s和T，若S>T，返回值大于0，若S=T，返回值等于0，否则小于0
   for(int i=1;i<=S.length&&i<=T.length;i++){
      if(S.ch[i]!=T.ch[i])
         return S.ch[i]-T.ch[i];
   }
   //如果扫描的所有字符串都一致，则长度长的串更大
   return S.length-T.length;
}
```

3. 定位操作:若主串S中存在与串T值相同的子串，则返回它在主串中第一次出现的位置，否则返回0

```
//定位操作
int Index(SString S,SString T){
    int i = 1,n=StrLength(S),m=StrLength(T);
    SString sub;//暂存子串
    while(i<=n-m+1){
        SubString(sub,S,i,m);
        if(StrCompare(sub,T)!=0) ++i;
        else return i;//返回子串在主串中的位置
    }
    return 0;
}
```

•字符串的模式匹配：在主串中找到与模式串相同的子串，并返回其所在位置

1. 朴素模式匹配算法（暴力）：将主串中所有长度为m的子串（最多对比n-m+1个子串）依次与模式串对比，直到找到一个完全匹配的子串，或者所有子串都不匹配为止

```
//朴素模式匹配算法（时间复杂度：最优O(m)，最坏O(nm)）
int Index(SString s,SString T){
    int i=1,j=1;
    while(i<=S.length && j<=T.length){
        if(S.ch[i]==T.ch[j]){
            ++i;
            ++j;
        }
        else{
            i=i-j+2;//失败后，指针后退，重新匹配
            j=1;
        }
    }
    if(j>T.length)//匹配成功，返回此时完全匹配第一个子串的位置
        return i-T.length;
    else
        return 0;
}
```

2. KMP算法

注意next数据求解

```
//KMP算法
int Index_KMP(SString S,SString T,int next[]){
    int i=1,j=1;
    while(i<=S.length&&j<=T.length){
        if(j==0||S.ch[i]==T.ch[j]){
            ++i;
            ++j;//继续比较后继字符
        }
        else
            j=next[j];//模式串向右移动
    }
    if(j>T.length)
        return i-T.length;//匹配成功
    else
        return 0;
}
```

第五章 树和二叉树

•树基本概念

空树——节点数为0的树

树的度：各结点的度的最大值

n叉树：每个结点最多只能有n棵子树的树

非空树的特点

有且仅有一个根节点

没有后继的结点称为“叶子节点”

有后继的结点称为“分支结点”

除了根结点，任何一个结有且仅有一个前驱

每个结点可以有0个或多个后继

树的常见考点

常见考点1:结点数=总度数+1

常见考点2:度为m的树、m叉树的区别

常见考点3:度为m的树第i层至多有m^{i-1}个结点；m叉树第i层至多有m^{i-1}个结点

常见考点4:高度为h的m叉树至多有$(m^h-1)/(m-1)$个结点

常见考点5:高度为h的m叉树至少有h个结点，高度为h，度为m的树至少有h+m-1个结点

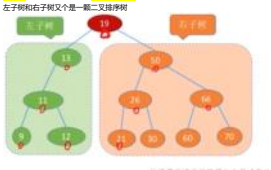
•二叉树（有序树）的基本概念

二叉树是n(>=0)个结点的有限集合:

1. 或者为空二叉树，即n=0;
2. 或者由一个根结点和两个互不相交的被称为根的左子树和右子树组成：左右子树分别也是一颗二叉树
3. tips：a.每个结点至多两颗子树b.左右子树不能颠倒

几个特殊的二叉树

1. 满二叉树：一颗高度为h，且含有2^h-1个结点的二叉树
   a. 只有最后一层叶子结点
   b. 不存在度为1点的结点
   c. 按层序从1开始编号，结点i的左孩子为2i，有孩子为2i+1
2. 完全二叉树：当且仅当其每个结点都与高度为h的满二叉树中的编号为1 – n的结点一一对应
   a. 只有可能在最后两层有叶子结点
   b. 最多只有一个度为1的结点
   c. 按层序从1开始编号，结点i的左孩子为2i，有孩子为2i+1
   d. 如果i<=n/2，则结点i为分支结点，否则为叶子结点
3. 二叉排序树：一颗二叉树或者是空二叉树，或者具有如下性质的树：
   a. 左子树上所有结点的关键字均小于根结点的关键字
   b. 右子树上所有结点的关键字均大于根结点的关键字
   c. 左子树和右子树又个是一颗二叉排序树



4. 平衡二叉树：树上任一结点的左子树和右子树的深度之差不超过1

二叉树常考性质：

1. 设非空二叉树中度为0，1和2的结点个树分别为n0，n1和n2，则n0=n2+1
2. 二叉树第i层之多有2^(i-1)个结点，m叉树第i层最多有m^(i-1)个结点
3. 高度为h个二叉树至多有2^h-1个结点(满二叉树)高度为h的m叉树至多有(m^h-1)/(m-1)个结点

二叉树的存储结构

1. 顺序存储

```
//顺序存储
#define MaxSize 100
struct TreeNode{
      ElemType value;//结点中的数据元素
      bool isEmpty;//结点是否为空
};
TreeNode t[MaxSize];
```

2. 链式存储

```
//链式存储
typedef struct BiTNode{
      ElemType data;
      struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;
```

3. 链式存储，初始化，构建二叉树

```
//链式存储，初始化，构建二叉树

struct ElemType{
      int value;
};


typedef struct BiTNode{
      ElemType data;
      struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

//定义一颗空树
Bitree root = NULL;


//插入根节点
root = (BiTree)malloc(sizeof(BiTNode));
root->data=(1);
root->lchild = NULL;
root->rchild = NULL;


//插入新节点
BiTNode *p = (biTNode*)malloc(sizeof(biTNode));
p->data = (2);
p->lchild = NULL;
p->rchild = NULL;
root->lchild = p;//作为根节点的左孩子
```

二叉树的先/中/后/层序遍历

```
typedef struct BiTNode{
      ElemType value;
      struct BiTNode *lchild,*rchild;
}BiTNode,BiTree;
```

- **先序遍历：根左右（NLR）**

```
void PreOrder(BiTree T){
      if(T!=NULL){
            visit(T);//访问根节点
            PreOrder(T->lchild0);//递归遍历左子树
            PreOrder(T->rchild);//递归遍历右子树
      }
}
```

- **中序遍历：左根右（LNR）**

```
void InOrder(BiTree T){
      if(T!=NULL){
            InOrder(T->lchild);//递归遍历左子树
            visit(T);//访问根节点
            InOrder(T->rchild);//递归遍历右子树
      }
   }
```

- **后序遍历：左右根（LRN）**

```
void PostOrder(BiTree T){
      if(T!=NULL){
            PoxrOrder(T->lchild);//递归遍历左子树
            PostOrder(T->rchild);//递归遍历右子树
            visit(T);
      }
}
```

•求树的深度

```
//求树的深度
int treeDepth(BiTree T){
      if(T == NULL){
            return 0;
      }
      else{
            int l = treeDepth(T->lchild);
            int r = treeDepth(T->rchild);
            //树的深度=Max（左子树深度，右子树深度）+1;
            return l>r?l+1:r+1;
      }
}
```

•层序遍历

```
//层序遍历
//二叉树的结点
typedef struct BiTNode{
      char data;
      struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

//链式队列结点
typedef struct LinkNode{
      BiTNode *data;
      struct LinkNode *next;
}LinkNode;

typedef struct{
      LinkNode *front,*rear;//队头队尾
}LinkQueue;

//初始化队列操作（带头结点）
void InitQueue(LinkQueue &Q){
      Q.front = Q.rear = (LinkNode*)malloc(sizeof(LinkNode));
      Q.front->next=NULL;
}

//判断队列是否为空（带头节点）
bool isEmpty(LinkQueue Q){
      if(Q.front == Q.rear)
            return true;
      else
            return false;
}

//入队（带头节点）
void EnQueue(LinkQueue &Q,ElemType x){
      LinkNode *s=(LinkNode*)malloc(sizeof(LinkNode));
      s->data = x;
      s->next = NULL;
      Q.rear->next = s;//新结点插入到rear之后
      Q.rear = s;//修改表尾指针
}

void LevelOrder(BiTree){
      LinkQueue Q;//初始化辅助队列
      InitQueue(Q);
      BiTree p;
      EnQueue(Q,T);//根节点入队
      while(!isEmpty(Q)){//队列不空则循环
            DeQueue(Q,p);//队头结点出队
            visit(p);//访问出队结点
            if(p->lchild!=NULL)
                  EnQueue(Q,p->lchild);//左孩子入队
            if(p->rchild!=NULL)
                  EnQueue(Q,p->rchild);//右孩子入队
      }
}
```

·线索二叉树

1. 线索二叉树的存储结构

```
//二叉树的结点（链式存储）
typedef struct BiTNode{
      ElemType data;
      struct BiTNode,*lchild,*rchild;
}

//线索二叉树结点
typedef struct ThreadNode{
      ElemType data;
      struct ThreadNode *lchild,*rchild;
      int ltag,rtag;//左右线索标志
}ThreadNode,*ThreadTree;
```

2. 二叉树的中序/先序/后续线索化

```
//访问结点e
void visit(BiTNode *q){
      if(q==p)//当前访问结点刚好是结点p
            final = pre;//找到p的前驱
      else
            pre = q;//pre指向当前结点
}

//中序递归遍历
void InOrder(BiTNode T){
      if(T!=NULL){
            inOrder(T->lchild);
            visit(T);
            inOrder(T->rchild);
      }
}

//辅助全局变量，用于查找结点p的前驱
BiTNode *p;//指向目标结点
BiTNode *pre = NULL;//指向当前访问结点的前驱
BiTNode *final = NULL;//记录最终结果
```

3. 中序线索化

```
//中序线索化
//全局变量pre，指向当前访问结点的前驱
ThreadNode *pre = NULL;

//线索二叉树结点
typedef struct ThreadNode{
	ElemType data;
	struct ThreadNode *lchild,*rchild;
	int ltag,rtag;//左右线索标志
}ThreadNode,*ThreadTree;

//中序遍历二叉树，一边遍历一边线索化
void InThread(ThreadTree T){
	if(T!=NULL){
		InThread(T->lchild);//中序遍历左子树
		visit(T);//访问根节点
		InThread(T->rchild);//中序遍历右子树
	}
}

void visit(ThreadNode *q){
	if(q->lchild==NULL){//左子树为空，建立前驱线索
		q->lchild=pre;
		q->ltag = 1;
	}
	if(pre!=NULL&&pre->rchild==NULL){
		pre->rchild=q;//建立前驱结点的后继线索
		pre->rtag = 1;
	}
	pre = q;
}

//中序线索化二叉树：
void CreateInThread(ThreadTree T){
	pre = NULL;
	if(T!=NULL){//非空二叉树才能线索化
		InThread(T);
		if(pre->rchild==NULL)
			pre->rtag=1;//处理遍历后的最后一个结点
	}
}
```

4. 先序遍历线索化

```
//先序线索化
//全局变量pre，指向当前访问结点的前驱
ThreadNode *pre = NULL;

//线索二叉树结点
typedef struct ThreadNode{
	ElemType data;
	struct ThreadNode *lchild,*rchild;
	int ltag,rtag;//左右线索标志
}ThreadNode,*ThreadTree;

//先序遍历二叉树，一边遍历一边线索化
void PreThread(ThreadTree T){
	if(T!=NULL){
		visit(T);//访问根节点
		if(T->ltag==0)//lchild不是先驱线索
			PreThread(T->lchild);//先序遍历左子树
		PreThread(T->rchild);//先序遍历右子树
	}
}

void visit(ThreadNode *q){
	if(q->lchild==NULL){//左子树为空，建立前驱线索
		q->lchild=pre;
		q->ltag = 1;
	}
	if(pre!=NULL&&pre->rchild==NULL){
		pre->rchild=q;//建立前驱结点的后继线索
		pre->rtag = 1;
	}
	pre = q;
}

//先序线索化二叉树：
void CreatePreThread(ThreadTree T){
	pre = NULL;
	if(T!=NULL){//非空二叉树才能线索化
		PreThread(T);
		if(pre->rchild==NULL)
			pre->rtag=1;//处理遍历后的最后一个结点
	}
}
```

5. 后序线索化

```
//后序线索化
void PostThread(ThreadTree p,ThreadTree &pre){
	if(p!=NULL){
		PostThread(p->lchild,pre);//递归，线索化左子树
		PostThread(p->rchild,pre);//递归，线索化右子树
		if(p->lchild==NULL){//左子树为空，建立前驱线索
			p->lchild=pre;
			p->ltag=1;
		}
		if(pre!=NULL&&pre->rchild==NULL){
			pre->rchild=p;
			pre->rtag=1;
		}
		pre=p;
	}
}

void CreatePostThread(ThreadTree T){
	ThreadTree pre = NULL;
	if(T!=NULL){
		PostThread(T,pre);//线索化二叉树
		if(pre->rchild==NULL)//处理遍历最后一个结点
			pre->rtag=1;
	}
}
```

数据结构

2021年9月22日 星期三　　下午3:39

第五章 树和二叉树

•基本概念

空树——结点数为0的树

非空树

有且仅有一个根节点：c

没有后继的结点称为“叶子节点”（或终端结点）；

有后继的结点称为“分支节点”（或非终端结点）；

除了根节点外，任何一个结点都有且仅有一个前驱

每个结点可以有0个或多个后继

•基本术语