树

1. 二叉树

```
#include <stdio.h>
#include <malloc.h>

#define MaxSize 100
typedef char ElemType;

typedef struct node //二叉链中结点类型BTNode
{
    ElemType data;          //数据元素
    struct node *lchild;    //指向左孩子节点
    struct node *rchild;    //指向右孩子节点
} BTNode;


void CreateBTree(BTNode * &b,char *str) //创建二叉树：A(B(D,E),C(,F))
{
    BTNode *St[MaxSize],*p=NULL;
    int top=-1,k,j=0;
    char ch;
    b=NULL;             //建立的二叉树初始时为空
    ch=str[j];
    while (ch!='\0')    //str未扫描完时循环
    {
        switch(ch)
        {
        case '(':top++;St[top]=p;k=1; break;        //为左孩子节点
        case ')':top--;break;
        case ',':k=2; break;                            //为孩子节点右节点
        default:p=(BTNode *)malloc(sizeof(BTNode));
                p->data=ch;p->lchild=p->rchild=NULL;
                if (b==NULL)                            //*p为二叉树的根节点
                    b=p;
                else                                    //已建立二叉树根节点
                {
                    switch(k)
                    {
                    case 1:St[top]->lchild=p;break;
                    case 2:St[top]->rchild=p;break;
                    }
                }
        }
        j++;
        ch=str[j];
    }
}

void DestroyBTree(BTNode *&b) //销毁二叉树
{   if (b!=NULL)  //若b==null
```

```
    {   DestroyBTree(b->lchild); //递归调用
        DestroyBTree(b->rchild);
        free(b);
    }
}

BTNode *FindNode(BTNode *b,ElemType x) //查找结点
{
    BTNode *p;
    if (b==NULL)  //若b==null
        return NULL;
    else if (b->data==x) //找到b了
        return b;
    else
    {
        p=FindNode(b->lchild,x);  //p=f(b->lchild,x)且p!=null
        if (p!=NULL)
            return p;
        else //p==null,f(b->rchild)
            return FindNode(b->rchild,x);
    }
}

BTNode *LchildNode(BTNode *p)  //返回结点p的左孩子指针
{
    return p->lchild;
}

BTNode *RchildNode(BTNode *p)  //返回结点p的右孩子指针
{
    return p->rchild;
}

int BTHeight(BTNode *b)   //求树高
{
    int lchildh,rchildh;
    if (b==NULL) return(0);                  //空树的高度为0
    else
    {
        lchildh=BTHeight(b->lchild);    //求左子树的高度为lchildh
        rchildh=BTHeight(b->rchild);    //求右子树的高度为rchildh
        return (lchildh>rchildh)? (lchildh+1):(rchildh+1);
    }
}

void DispBTree(BTNode *b) //A(B(D,E),C(,F))
{
    if (b!=NULL)
    {   printf("%c",b->data);
        if (b->lchild!=NULL || b->rchild!=NULL)
        {   printf("(");                          //有孩子节点时才输出(
            DispBTree(b->lchild);                //递归处理左子树
            if (b->rchild!=NULL) printf(",");    //有右孩子节点时才输出,
            DispBTree(b->rchild);                //递归处理右子树
            printf(")");                         //有孩子节点时才输出)
```

```
        }
    }
}

// /*以下主函数用做调试
void main()
{
    BTNode *b;
    CreateBTree(b,"A(B(D,E),C(,F))");
    DispBTree(b);
    printf("\n");
}
```

```
//树的队列层次遍历算法：A(B(D(,G)),C(E,F))
#include "btree.cpp"
#define MaxSize 100

void LevelOrder(BTNode *b)
{
    BTNode *p;
    SqQueue *qu;
    InitQueue(qu);                  //初始化队列
    enQueue(qu,b);                  //根结点指针进入队列
    while (!QueueEmpty(qu))         //队不为空循环
    {
        deQueue(qu,p);              //出队节点p
        printf("%c ",p->data);      //访问节点p
        if (p->lchild!=NULL)        //有左孩子时将其进队
            enQueue(qu,p->lchild);
        if (p->rchild!=NULL)        //有右孩子时将其进队
            enQueue(qu,p->rchild);
    }
}
```

```
//先序、中序和后序递归遍历算法：A(B(D(,G)),C(E,F))
#include "btree.cpp"

void PreOrder(BTNode *b)        //先序遍历的递归算法
{
    if (b!=NULL)
    {
        printf("%c ",b->data);  //访问根结点
        PreOrder(b->lchild);    //先序遍历左子树
        PreOrder(b->rchild);    //先序遍历右子树
    }
}

void InOrder(BTNode *b)         //中序遍历的递归算法
{
    if (b!=NULL)
    {
        InOrder(b->lchild);     //中序遍历左子树
        printf("%c ",b->data);  //访问根结点
        InOrder(b->rchild);     //中序遍历右子树
```

```
    }
}

void PostOrder(BTNode *b)       //后序遍历的递归算法
{
    if (b!=NULL)
    {
        PostOrder(b->lchild);   //后序遍历左子树
        PostOrder(b->rchild);   //后序遍历右子树
        printf("%c ",b->data);  //访问根结点
    }
}
```

```
//先序、中序和后序非递归遍历算法
typedef struct
{   BTNode *data[MaxSize];          //存放栈中的数据元素
    int top;                        //存放栈顶指针，即栈顶元素在data数组中的下标
} SqStack;

void PreOrder1(BTNode *b)           //先序非递归遍历算法1
{
    BTNode *p;
    SqStack *st;                    //定义一个顺序栈指针st
    InitStack(st);                  //初始化栈st
    Push(st,b);                 //根节点进栈
    while (!StackEmpty(st))     //栈不为空时循环
    {
        Pop(st,p);              //退栈节点p并访问它
        printf("%c ",p->data);  //访问节点p
        if (p->rchild!=NULL)    //有右孩子时将其进栈
            Push(st,p->rchild);
        if (p->lchild!=NULL)    //有左孩子时将其进栈
            Push(st,p->lchild);
    }
    printf("\n");
    DestroyStack(st);               //销毁栈
}
void PreOrder2(BTNode *b)           //先序非递归遍历算法2
{
    BTNode *p;
    SqStack *st;                    //定义一个顺序栈指针st
    InitStack(st);                  //初始化栈st
    p=b;
    while (!StackEmpty(st) || p!=NULL)
    {
        while (p!=NULL)             //访问节点p及其所有左下节点并进栈
        {
            printf("%c ",p->data);  //访问节点p
            Push(st,p);             //节点p进栈
            p=p->lchild;            //移动到左孩子
        }
        if (!StackEmpty(st))        //若栈不空
        {
            Pop(st,p);              //出栈节点p
            p=p->rchild;            //转向处理其右子树
```

```
        }
    }
    printf("\n");
    DestroyStack(st);                //销毁栈
}
void InOrder1(BTNode *b)                //中序非递归遍历算法
{
    BTNode *p;
    SqStack *st;                        //定义一个顺序栈指针st
    InitStack(st);                      //初始化栈st
    if (b!=NULL)
    {
        p=b;
        while (!StackEmpty(st) || p!=NULL)
        {
            while (p!=NULL)             //扫描节点p的所有左下节点并进栈
            {
                Push(st,p);             //节点p进栈
                p=p->lchild;            //移动到左孩子
            }
            if (!StackEmpty(st))        //若栈不空
            {
                Pop(st,p);              //出栈节点p
                printf("%c ",p->data);  //访问节点p
                p=p->rchild;            //转向处理其右子树
            }
        }
        printf("\n");
    }
    DestroyStack(st);                //销毁栈
}

void PostOrder1(BTNode *b)              //后序非递归遍历算法
{
    BTNode *p,*r;
    bool flag;
    SqStack *st;                        //定义一个顺序栈指针st
    InitStack(st);                      //初始化栈st
    p=b;
    do
    {
        while (p!=NULL)                 //扫描节点p的所有左下节点并进栈
        {
            Push(st,p);                 //节点p进栈
            p=p->lchild;                //移动到左孩子
        }
        r=NULL;                         //r指向刚刚访问的节点，初始时为空
        flag=true;                      //flag为真表示正在处理栈顶节点
        while (!StackEmpty(st) && flag)
        {
            GetTop(st,p);               //取出当前的栈顶节点p
            if (p->rchild==r)           //若节点p的右孩子为空或者为刚刚访问过的节点
            {
                printf("%c ",p->data);  //访问节点p
                Pop(st,p);
```

```
                r=p;                        //r指向刚访问过的节点
            }
            else
            {
                p=p->rchild;                //转向处理其右子树
                flag=false;                 //表示当前不是处理栈顶节点
            }
        }
    } while (!StackEmpty(st));              //栈不空循环
    printf("\n");
    DestroyStack(st);                   //销毁栈
}
```

```
typedef char ElemType;
typedef struct node //线索二叉树结点类型声明
{
    ElemType data;
    int ltag,rtag;       //增加的线索标记
    struct node *lchild; //左孩子或线索指针
    struct node *rchild; //右孩子或线索指针
} ThreadNode;

//中序线索二叉树的算法
void InThread(TBTNode *&p,ThreadNode *&pre)
{
    if (p!=NULL)
    {
        InThread(p->lchild);                //左子树线索化
        if (p->lchild==NULL)            //前驱线索
        {
            p->lchild=pre;              //建立当前节点的前驱线索
            p->ltag=1;
        }
        else p->ltag=0;
        if (pre->rchild==NULL)          //后继线索
        {
            pre->rchild=p;              //建立前驱节点的后继线索
            pre->rtag=1;
        }
        else pre->rtag=0;
        pre=p;
        InThread(p->rchild);            //右子树线索化
    }
}

void CreatInThread(ThreadNode *T)
{
    ThreadNode *pre=Null;
    if(T!=Null){              //对非空二叉树线索化
        InThread(T,pre);    //线索化二叉树
        pre->rchild=Null;   //处理遍历结束后的最后一个结点
        pre->rtag=1;
    }
}
```