

# 实验三 二叉树及其应用

---

题目：二叉树及其应用

班级：少年班学院少年班 姓名：邱子悦 学号：PB18000028 完成日期：2019.11.10

## 一. 实验要求

---

### 1. 二叉树的创建与遍历

基本要求：

- 通过添加虚结点，为二叉树的每一实结点补足其孩子，再对补足虚结点后的二叉树按层次遍历的次序输入。
- 增加左右标志域，将二叉树后序线索化。
- 完成后序线索化树上的遍历算法，依次输出该二叉树先序遍历、中序遍历和后序遍历的结果。

### 2. 表达式树

基本要求：

- 输入合法的波兰式(仅考虑运算符为双目运算符的情况)，构建表达式树
- 分别输出对应的中缀表达式(可含有多余的括号)、逆波兰式和表达式的值
- 输入的运算符与操作数之间会用空格隔开。

选做要求(二选一即可)：

1. 输出的中缀表达式中不含有多余的括号。
2. 输入逆波兰式，输出波兰式、中缀表达式(可含有多余的括号)和表达式的值。

## 二. 设计思路

---

### 1. 二叉树的创建与遍历

- 利用c++的queue函数，在CreateTree()函数中，将输入的按层次的字符串转化为二叉树
- 递归实现了PreOrder(), InOrder(), 先序、中序遍历
- 后序线索化PostOrderThreading()函数：左孩子为NULL，左标志域置为Thread，并指向前序结点；右孩子为NULL且前序结点不为空，右标志域置为Thread，并指向根结点；Prev指针更新。
- 后序线索化遍历PostOrder()：具体见“三. 关键代码讲解”。

### 2. 表达式树

- 判断函数：isOperator()判断是不是操作符，Lower()判断优先级，Calculate()根据符号计算数值。
- RevPolish() 将表达式二叉树输出为逆波兰表达式形式
- CreateExpression() 将输入的波兰表达式转化为二叉树，递归实现

- ArExp() 将表达式二叉树输出为表达式形式，并通过判断优先级，去掉了多余括号。

## 三.关键代码讲解

### 1. 二叉树的创建与遍历

- 数据类型:

```
typedef enum{ Link, Thread} PointerTag;
typedef struct BiThrNode{
    char data;
    struct BiThrNode *lchild,*rchild,*parent;
    PointerTag ltag,rtag;
}BiThrNode, *BiThrTree;
```

- 全局变量

```
int n;//the total number of knots
char save[MAXLEN];
queue<BiThrTree> que;
```

- 利用了c++自带的queue函数，辅助建立二叉树，并且进行了初始化。

```
BiThrTree CreateTree(){
    int tn=n;
    int point=0;//pointer of the save[]
    BiThrTree p,t,q;
    //p for root, q for create a new knot, t for exchange

    if(save[point]=='#')
        return NULL;//if root even not exist, just return NULL
    else{
        p=(BiThrTree)malloc(sizeof(BiThrNode));
        p->data=save[point];
        que.push(p);
        tn--;
    }
    while(!que.empty()){
        t=que.front();
        que.pop();
        if(tn==0)
            break;
        point++;
        if(save[point]!='#'){
            q=(BiThrTree)malloc(sizeof(BiThrNode));
            q->data=save[point];
            t->lchild=q;
            t->lchild->parent=t;
        }
    }
}
```

```

        que.push(q);
        tn--;
    }
    else{
        t->lchild=NULL;
        tn--;
    }
    point++;
    if(save[point]!='#'){
        q=(BiThrTree)malloc(sizeof(BiThrNode));
        q->data=save[point];
        t->rchild=q;
        t->rchild->parent=t;
        que.push(q);
        tn--;
    }
    else {
        t->rchild = NULL;
        tn--;
    }
}
return p;
}

```

- 递归实现先序、中序遍历

```

void PreOrder(BiThrTree Root){
    if(!Root)
        return;
    printf("%c",Root->data);
    PreOrder(Root->lchild);
    PreOrder(Root->rchild);
}

void InOrder(BiThrTree Root){
    if(!Root)
        return;
    InOrder(Root->lchild);
    printf("%c",Root->data);
    InOrder(Root->rchild);
}

```

- 后序线索化：左孩子为NULL，左标志域置为Thread，并指向前序结点；右孩子为NULL且前序结点不为空，右标志域置为Thread，并指向根结点；Prev指针更新。

```

void PostOrderThreading(BiThrTree& Root,BiThrTree& Prev){
    if(Root==NULL)
        return;

```

```

PostOrderThreading(Root->lchild, Prev);
PostOrderThreading(Root->rchild, Prev);

if(Root->lchild==NULL){
    Root->lchild=Prev;
    Root->ltag=Thread;
}
if(Prev!=NULL && Prev->rchild==NULL){
    Prev->rchild=Root;
    Prev->rtag=Thread;
}
Prev=Root;
}

```

- 后序线索化遍历：从最左的结点开始遍历，一直往后续结点访问，若无后续结点，继续沿着Link往上走，直到遇到根（因为Root->parent==NULL），开始遍历右子树。

```

void PostOrder(BiThrTree pRoot)
{
    BiThrTree pCur = pRoot;
    BiThrTree prev = NULL;

    while (pCur)
    {
        while (pCur->lchild!=prev && pCur->ltag == Link) //找到最左边的结点
            pCur = pCur->lchild;

        while (pCur->rtag == Thread) //访问连在一起的后续结点
        {
            printf("%c", pCur->data);
            prev = pCur;
            pCur = pCur->rchild;
        }
        while (pCur && pCur->rchild == prev) //访问当前结点
        {
            printf("%c", pCur->data);
            prev = pCur;
            pCur = pCur->parent;
        }
        if (pCur && pCur->rtag == Link) //开始访问右子树
        {
            pCur = pCur->rchild;
        }
    }
}

```

## 2. 表达式树

## 数据类型：

```
typedef struct
{
    int num;    //存储整形数字
    char ch;    //存储运算符
} ElemType;

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

- 判断函数：isOperator()判断是不是操作符，Lower()判断优先级，Calculate()根据符号计算数值。

```
int isOperator(char s[]){
    if(strcmp(s,"+")==0)
        return 1;
    if(strcmp(s,"-")==0)
        return 1;
    if(strcmp(s,"*")==0)
        return 1;
    if(strcmp(s,"/")==0)
        return 1;
    return 0;
}
```

```
int Lower(char a,char b){
    if((a=='+' || a=='-' ) && (b=='*' || b=='/'))
        return 1;
    return 0;
}
```

```
double Calculate(BiTree T){
    if(T->data.ch=='#')
        return T->data.num;

    switch(T->data.ch){
        case '+': return Calculate(T->lchild)+Calculate(T->rchild);
        case '-': return Calculate(T->lchild)-Calculate(T->rchild);
        case '*': return Calculate(T->lchild)*Calculate(T->rchild);
        case '/': return Calculate(T->lchild)/Calculate(T->rchild);
        default: printf("Error.");
    }
}
```

- RevPolish() 将表达式二叉树输出为逆波兰表达式形式，要注意检查data是操作符还是数字

```

void RevPolish(BiTree Root){
    if(!Root)
        return;
    RevPolish(Root->lchild);
    RevPolish(Root->rchild);
    if(Root->data.num==-1)
        printf("%c ",Root->data.ch);
    else if(Root->data.ch=='#')
        printf("%d ",Root->data.num);
    else{
        printf("num and char error!\n");
    }
}

```

- CreateExpression() 将输入的波兰表达式转化为二叉树，递归实现

```

BiTree CreateExpression(){
    char s[MAXLEN];
    scanf("%s",s);
    BiTree p=(BiTree)malloc(sizeof(BiNode));
    if(!isOperator(s)){
        p->lchild=p->rchild=NULL;
        p->data.num=atof(s);
        p->data.ch='#';//initial
    }
    else{
        p->data.ch=s[0];
        p->data.num=-1;//initial
        p->lchild=CreateExpression();
        p->rchild=CreateExpression();
    }
    return p;
}

```

- ArExp() 将表达式二叉树输出为表达式形式，并通过判断优先级，去掉了多余括号。

```

void ArExp(BiTree& T){
    if(T){
        if(T->lchild){
            if(Lower(T->lchild->data.ch,T->data.ch)){
                printf("(");
                ArExp(T->lchild);
                printf(")");
            }
            else
            {
                ArExp(T->lchild);
            }
        }
    }
}

```

```

    }
    if(T->data.num==-1)
        printf("%c",T->data.ch);
    else if(T->data.ch=='#')
        printf("%d",T->data.num);
    else{
        printf("num and char error!\n");
    }
    if(T->rchild){
        if(Lower(T->rchild->data.ch,T->data.ch)){
            printf("(");
            ArExp(T->rchild);
            printf(")");
        }
        else
        {
            ArExp(T->rchild);
        }
    }
}
return;
}

```

## 四.调试分析&代码测试

- 因为过于顺利，代码没有遇到太多调试的问题，出错也总能立刻找到，所以合并了两个模块。

### 1. 输入输出情况：

Input: AB#C#D##E##

Output:

ABCDE

DECBA

EDCBA

```

AB#C#D##E##
ABCDE
DECBA
EDCBA

```

Input: a#bc##de##f#####

Output:

abcdef acefdb fedcba

```

a#bc##de##f#####
abcdef
acefdb
fedcba

```

Input: abcd##e##f###

Output:

abdcef dbacfe dbfeca

```
abcd##e##f###
abdcef
dbacfe
dbfeca
```

## 2. 输入输出情况：

Input: / + 15 \* 5 + 2 18 5

Output:

(15+5\*(2+18))/5 15 5 2 18 + \* + 5 / 23

```
/ + 15 * 5 + 2 18 5
(15+5*(2+18))/5
15 5 2 18 + * + 5 /
23
```

Input: - / 15 - 2 7 10

Output:

15/(2-7)-10 15 2 7 - / 10 - -13

```
- / 15 - 2 7 10
15/(2-7)-10
15 2 7 - / 10 -
-13
```

Input: - 20 + 11 13

Output:

20-11+13 20 11 13 + - -4

```
- 20 + 11 13
20-11+13
20 11 13 + -
-4
```

Input: / 32 \* 8 2

Output:

32/8\*2 32 8 2 \* / 2

```
/ 32 * 8 2
32/8*2
32 8 2 * /
2
```

## 五.实验总结

第一次尝试较大的二叉树实现，使用CLion书写和执行。



涉及到了按层次输入的二叉树建立、线索化、线索化遍历、递归遍历、表达式树（应用）等等，极大地锻炼了能力。

从最初的手忙脚乱到后来功能清晰、细节处理到位，收获很大，体会到了思考全面以及迅速找到出错地方的方法。

实验进行较为顺利，按时完成实验。

## 六.附录

---

main.c //主程序