中国科学技术大学计算机学院
《数据结构》报告

实验题目：二叉树及其应用

学生姓名：高楚晴

学生学号：PB18111688

完成日期：2019.12.5

实验目的

1. 以二叉树的链式表示、建立和应用为基础，深入了解二叉树的存储表示特征以及遍历次序与二叉树的存储结构之间的关系，进一步掌握利用遍历思想解决二叉树中相关问题的方法。
2. 通过思考、上机实践与分析总结，理解计算机进行算术表达式解析、计算的可能方法，初步涉及一些编译技术，增加今后学习编译原理的兴趣，并奠定一些学习的基础。

设计思路

二叉树的创建与遍历

本实验目的为通过添加虚结点，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。构建不含虚节点的二叉树，并增加左右标志域，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。

由于层序遍历，普通先序遍历、中序遍历以及二叉树的创建算法均为课内所学，在此不再赘述，核心部分为后序线索化以及后序线索树上的后序遍历。基本思路为预置一个前继节点，即每次节点后移前保留当前节点，通过后序遍历的递归方式串联其左右标签域。

后序线索树上的后序遍历基本思路为：

①若该节点无左孩子，则前驱节点存放在其左孩子的虚节点处；若该节点有左孩子且有右孩子，则其前驱节点为其右孩子；若该节点有左孩子但没有右孩子，前驱节点为其左孩子。

②若该节点是二叉树的根，则其后继节点为NULL；若该节点是其双亲的右孩子或是双亲的左孩子但没有右子树，则其猴急为其双亲。若该节点是双亲的左孩子且其双亲有右子树，则后继几点为双亲右子树的后序遍历列出的第一个结点。

表达式树

实验目的为输入合法的波兰式(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。

构建的基本思路为按照先序遍历递归的基本原理，输入为字符则生成左右子树，直至读到数字，将其置为树叶。中缀表达式即为按照中序遍历的基本原理，对每个递归入口附加左右括号，逆波兰式即以后序遍历将表达式树输出，表达式求值即为检测双亲节点操作符的种类，递归的方式计算左右子树的值，最终得到结果。

特别地，由于表达书操作数可能存在多位，节点数据类型为字符数组，本程序中仅考虑三位以内数字输入，若需要更多位数直接修改字符数组长度即可。表达式求时一个一个字符处理即可。

选做

实验目的为输出不含多余括号的中缀表达式。

只需在必做部分中缀表达式输出括号的时候检测其优先级，若内层运算优先级低于外层，则补充括号，不然不添加括号。

关键代码讲解

二叉树的创建和遍历

层序遍历输入

按照层序遍历的基本原理读入该二叉树。

```
int LevelCreate(Bitree &T, int(*Crea)(Bitree &)) {
    Bitnode *p;
    InitQueue(Q);                    //队列内存放已读入但还没建立左右子树的节点
    Crea(T);
    while (!Isempty(Q)) {
        p = DeQueue(Q);
        if (tree[i] == '*') {        //读到特殊字符即创建一个对应空节点
            p->lchild = NULL;
            i++;
        }
        else {
            Crea(p->lchild);
            p->lchild->parent = p;
        }
        if (tree[i] == '*') {
            p->rchild = NULL;
            i++;
        }
        else {
            Crea(p->rchild);
            p->rchild->parent = p;
        }
    }
    return 0;
}
```

后序线索化

```
int PostThreading(Bitnode*& t)
{
    if (t)
    {
        PostThreading(t->lchild);
        PostThreading(t->rchild);       //按照后序遍历的方式访问节点并线索化
        if (t->lchild == NULL)          //将空节点依次线索化
        {
            t->lchild = Prev;           //前驱置为左标志域
            t->ltag = Thread;
        }
        if (Prev && Prev->rchild == NULL)    //处理右侧标志域
        {
            Prev->rchild = t;
            Prev->rtag = Thread;
        }
        Prev = t;
    }
    return 0;
}
```

后序遍历

```
int Postorder(Bitnode *t){
    if (!t)
        return 0;
```

```
    else {
        Bitnode* temp = t;
        Prev = NULL;
        while (temp)
        {
            while (temp ->lchild != Prev && temp->ltag == Link) //有左子树
                temp = temp ->lchild;
            while (temp && temp->rtag == Thread)    //右子树为空
            {
                printf("%c", temp->data);
                Prev = temp;
                temp = temp->rchild;
            }
            if (temp == t)                  //输出该节点
            {
                printf("%c", temp->data);
                return 0;
            }
            while (temp && temp->rchild == Prev)  //节点是左孩子且双亲无右子树
            {
                printf("%c", temp->data);
                Prev = temp;
                temp = temp->parent;
            }
            if (temp && temp->rtag == Link)     //节点是双亲右孩子
            {
                temp = temp->rchild;
            }
        }
    }
}
```

先序及中序均为基本的递归算法，在此不做赘述。

表达式树

创建

该函数是建立在事先将全部输入以gets方式存入字符数组str的基础上。

```
int create(Bitree &T) {               //创建表达式树
    int j;
    if (!str[i])
        return 0;
    T = (Bitree)malloc(sizeof(Bitnode));
    if(str[i]<'0'){                   //如果当前字符为操作符
        T->data[0] = str[i++];
        T->data[1] = '\0';            //由于以字符串形式存放，依次赋值后需要在末端补'\0'
    }
    else{
        j = 0;
        while(str[i]!=' '&&str[i]!='\0')    //直至读到空格分隔符或者读到字符数组末尾
            T->data[j++] = str[i++];        //以字符为单位复制
        T->data[j] = '\0';                  //补充结束符\0
    }
    T->lchild = NULL;
    T->rchild = NULL;
```

```
    i++;                                //跳过空格
    if (str[i - 2] < '0' )              //是操作符，则需访问其左右子树
    {
        create(T->lchild);
        create(T->rchild);
    }
    return 0;
}
```

逆波兰式

逆波兰式即将表达式树以后序遍历的方式访问节点并输出。

```
int postorder(Bitree T, int(*prin)(Bitree T)) {       //逆波兰
    if (T) {
        postorder(T->lchild, prin);
        postorder(T->rchild, prin);
        prin(T);                          //prin实现功能为printf(T->data)，含空格
    }
    else    return 0;
}
```

可含有多余括号的中缀表达式

实现方式为在每一个非根节点的左右子树访问前后添加左右括号

```
int inorder(Bitree T, int(*pri)(Bitree T)) {          //有多余括号的中缀
    if (T) {
        if (T->lchild->data[0] < '0') {
            printf("(");
            inorder(T->lchild, pri);
            printf(")");
        }
        else    pri(T->lchild);
        pri(T);                                 //pri实现功能为printf(T->data),不含空格
        if (T->rchild->data[0] < '0') {
            printf("(");
            inorder(T->rchild, pri);
            printf(")");
        }
        else    pri(T->rchild);
    }
    else    return 0;
}
```

表达式求值

```
int calc(Bitree T){                   //计算某个字符串对应数字值
    int k = 0,num = 0;
    while(T->data[k]!='\0')
        num = num*10+(T->data[k++]-'0');
    return num;
}
int count(Bitree T) {
    if (T->data[0] >= '0')
        return calc(T);
```

```
    else {                                  //依次递归计算左右子树的值
        switch (T->data[0]) {
        case '+': return count(T->lchild) + count(T->rchild);
        case '-': return count(T->lchild) - count(T->rchild);
        case '*': return count(T->lchild) * count(T->rchild);
        case '/': return count(T->lchild) / count(T->rchild);
        }
    }
    return 0;
}
```

不含多余括号的中缀表达式

只需在对应位置判断操作符的优先级，对于左子树，由于先输出，只需在内部加减外部乘除的情况加括号；右子树后输出，除了内部乘除外部加减的情况，其余均需要加括号。

```
int inorderp(Bitree T, int(*pri)(Bitree T)) {          //去多余括号的中缀
    if (T) {
        if (T->lchild->data[0] < '0') {
            if ((T->lchild->data[0] == '+' || T->lchild->data[0] == '-') && (T-
>data[0] == '*' || T->data[0] == '/')) {
                printf("(");
                inorderp(T->lchild, pri);
                printf(")");
            }
            else    inorderp(T->lchild, pri);
        }
        else    pri(T->lchild);
        pri(T);
        if (T->rchild->data[0] < '0') {
            if ((T->rchild->data[0] == '*' || T->rchild->data[0] == '/') && (T-
>data[0] == '+' || T->data[0] == '-'))
                inorderp(T->rchild, pri);
            else {
                printf("(");
                inorderp(T->rchild, pri);
                printf(")");
            }
        }
        else
            pri(T->rchild);
    }
    else    return 0;
}
```

调试分析

初次完成代码后进行调试，发现运行失败，经检查为误在出栈pop操作之后再进行了p = p->next操作，调试后运行正常。

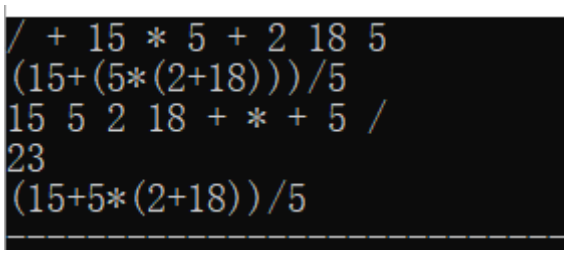
由于本次实验均为二叉树遍历相关算法的应用，他们的时间复杂度均为$O(n)$，n为结点数。

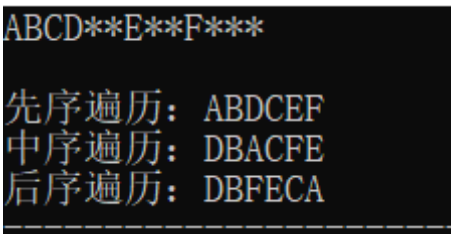
因为它们只访问每个节点一次，不存在多余的访问。 层次遍历的时间复杂度是O(w)，其中w是二叉树的宽度（拥有最多节点的层的节点数）。第二个实验的数值转换复杂度也为$O(n)$，综上，两个实验时间复杂度均为$O(n)$。n表示输入长度。

此外，第一个实验不产生额外存储空间，因此空间复杂度为$O(1)$,第二个实验需要字符串来存放输入，空间复杂度为$O(n)$。

代码测试

执行结果如下。





附录

PB18111688_高楚晴_3_1.cpp

PB18111688_高楚晴_3_2.cpp