上 海 大 学

2021-2022 冬季学期

《数据结构（1）》实验报告

实 验 组 号：08

上 课 老 师：沈　俊

小 组 成 绩：

小组成员成绩表

序号	学号	姓名	贡献因子	成绩
1	20120500	王静颐	20	
2	20120796	康高熙	20	
3	20121034	胡才郁	20	
4	20121076	刘元	20	
5	20124633	金之谦	20	

注：小组所有成员的贡献因子之和为 100.

计算机工程与科学学院

2021 年 02 月 28 日

实验五 二叉树

1 设计性实验

1.1 二叉树的顺序存储

1.1.1 基本功能介绍

参考二叉树的二叉链表类模板，设计并实现二叉树的顺序存储表示。增加函数成员，求离两个元素（编号为 i 和 j）最近的共同祖先。

1.1.2 主要算法设计

对于有根树 T 的两个结点 u、v，最近公共祖先 LCA(T,u,v)表示一个结点 x，满足 x 是 u 和 v 的祖先且 x 的深度尽可能大。在这里，一个节点也可以是它自己的祖先。

对于 u、v 节点最近公共祖先的查询，我们先将 u、v 跳到同一高度，即只把深度高的跳到其父节点，这时 u、v 可能跳到同一节点，这种情况他们的最近公共祖先就是当前节点，直接返回 u 节点即可。还有一种情况是 u、v 变成了堂兄弟节点，这时我们让 u、v 一直往上跳直到他们的父节点相同，此时他们的最近公共祖先就是他们的父节点，返回 u -> father 即可。

1.1.3 主要数据组织

在此定义了结构体模板 BinTreeNode 与类模板 BinTree,并给出了它们的主要数据成员以及成员函数，如下图所示：

```
结构体模板BinTreeNode:
T data;//该节点的数据
int number;//该节点的编号
int dep;//该节点的深度
BinTreeNode * father;//该节点的父节点
BinTreeNode* leftchild;//该节点的左儿子
BinTreeNode* rightchild;//该节点的右儿子
BinTreeNode(const int Number,const T data,BinTreeNode *lChild=NULL,BinTreeNode
*rChild=NULL,const int Dep=0,BinTreeNode<T> *Father=NULL);//构造函数

类模板BinTree:
private:
    BinTreeNode<T> * root;//这棵树的根节点
    T data[maxn];//这棵树每个节点的数据
public:
    BinTreeNode<T> * pos[maxn];//每个标号对应的节点地址
    void Insert(int cur,int lChild,int rChild);//节点插入
    void SetRoot(BinTreeNode<T> *node=NULL);//设置这棵树的根节点
    void SetData(const T *a=NULL,const int len=0);//设置每个节点的数据
    void GetDep();//遍历求出每个节点的深度
    BinTreeNode<T>* GetRoot();//返回根节点地址
    BinTreeNode<T>* GetLca(const int x,const int y);//返回两个节点的最近公共祖先的地址
```

1.1.4 测试分析

编写如下的测试函数，并进行测试：

```cpp
int main()
{
    printf("请输入节点数:");
    scanf("%d",&n),getchar();
    printf("请输入每个节点的数据:");
    for(int i=1;i<=n;++i) scanf("%c",&a[i]),getchar();
    tree.SetData(a,n);
    printf("请输入每个节点及其左右儿子:");
    for(int i=1;i<=n;++i)
    {
        scanf("%d%d%d",&cur,&lchild,&rchild);
        tree.Insert(cur,lchild,rchild);
    }
    tree.SetRoot(tree.pos[1]);
    tree.GetDep();
    printf("请输入询问组数:");
    scanf("%d",&m);
    for(int i=1;i<=m;++i)
    {
        printf("请输入询问节点:");
        scanf("%d%d",&x,&y);
        BinTreeNode<char>* lca=tree.GetLca(x,y);
        cout<<"节点"<<x<<"和节点"<<y<<"的最近公共祖先是"
            <<lca->number<<",数据为"<<lca->data<<endl;
    }
    return 0;
}
```

```
8
A B C D a b c d
1 2 3
2 4 0
3 5 6
4 0 0
5 7 8
6 0 0
7 0 0
8 0 0
5
2 3
5 6
4 6
2 4
7 8
```

使用右图的数据进行测试，并获得测试结果：

```
请输入节点数:8
请输入每个节点的数据:A B C D a b c d
请输入每个节点及其左右儿子:1 2 3
2 4 0
3 5 6
4 0 0
5 7 8
6 0 0
7 0 0
8 0 0
请输入询问组数:5
请输入询问节点:2 3
节点2和节点3的最近公共祖先是1,数据为A
请输入询问节点:5 6
节点5和节点6的最近公共祖先是3,数据为C
请输入询问节点:4 6
节点4和节点6的最近公共祖先是1,数据为A
请输入询问节点:2 4
节点2和节点4的最近公共祖先是2,数据为B
请输入询问节点:7 8
节点7和节点8的最近公共祖先是5,数据为a

--------------------------------
Process exited after 107.8 seconds with return value 0
请按任意键继续. . .
```

经过多组数据测试表明，此程序能够稳定完成实验要求。

2 综合性实验

2.1 表达式二叉树

2.1.1实验内容

表达式可以用一颗二叉树表示，叶子结点代表操作数，分支节点代表操作符。对于表示简单四则运算表达式的表达式二叉树，要求实现：（1）输入表达式，生成其二叉树表示。（2）对于一颗构造好的表达式二叉树，输出相应的中缀表达式（不允许有冗余的括号）。（3）对于一颗构造好的表达式二叉树，输出相应的后缀表达式。（4）对于操作数都是正数的表达式二叉树，计算该表达式的值。（5）输出表达式二叉树的树形结构

2.1.2 算法分析

对于一个构造好的二叉树，输出相应的中缀表达式，我们直接按照二叉树的中序遍历即可。输出后缀表达式其实直接后序遍历即可。只不过输出相应的中缀表达式时，我们要考虑添加括号的情况。分析可知，只有当父亲节点的运算符优先级高于左儿子或者右儿子运算符的优先级时，我们需要在相应的位置两端添加括号。

对于问题（4），表达式二叉树已经构建好的情况下，求表达式的值，我们只需要按照后序遍历的顺序，递归的先计算出左儿子的值，再计算出右儿子的值，再求得整颗子树的值即可。

对于问题（1），我们仍然采取递归的方式。假如当前递归层所处理的表达式长度为 m，我们 for 循环遍历直到找到第一个不在括号里的+或-。找到以后，我们把符号左边的去递归处理当做左子树，把符号右边的去递归处理当做右子树。当前符号当做父亲节点。递归结束后，自然构造好了表达式二叉树。

2.1.3 主要数据组织

在建立表达式二叉树时，主要使用到了栈数据结构辅助非递归遍历。使用到栈的主要成员函数如下：

```
int GetLength() const;                          // 求栈长度
bool IsEmpty() const;                           // 判断栈是否为空
Status Push(const ElemType e);                  // 入栈
Status Top(ElemType &e) const;                  // 返回栈顶元素
Status Pop(ElemType &e);                        // 出栈
```

并且在以下条件内，对于第一操作数与第二操作数进行构建：

```
while (stack_ope.Top(temp),
       !stack_ope.IsEmpty() &&
       temp != '(' &&
       ops[temp] >= ops[expression[i]])
```

遇到右括号时，进行处理：

```
if (expression[i] == ')') {
    char temp;        //to restore the top of stack_ope

    while (!stack_ope.IsEmpty() && (stack_ope.Top(temp), temp != '(')) {
        char stack_ops_pop;                       //to restore the pop of stack_ope
        string temp_str;                        //convert stack_ops_pop to string
        stack_ope.Pop(stack_ops_pop);            //pop stack_ope
        temp_str = stack_ops_pop;                //convert it
        t = new BinTreeNode<string>(temp_str);    //struct a new node from temp_str
        stack_node.Pop(t1);                       //get right child
        stack_node.Pop(t2);                       //get left child
        t->leftChild = t2;                       //link left child
        t->rightChild = t1;                       //link right child
        stack_node.Push(t);                       //push the new node to the stack
    }
    stack_ope.Pop(
        temp);                //delete the unnecessary '(', otherwise loop won't run next time
}
```

在中序遍历时，设计了如下函数，通过比较优先级正确顺序输出括号：

```
//print the tree as in order
void inOrder(BinTreeNode<string> *p) {
    if (p == NULL)return;
    if (p->leftChild != NULL) {
        BinTreeNode<string> *le = p->leftChild;
        bool flag = 0;
        if (p->data == "*" || p->data == "/") {
            if (le->data == "+" || le->data == "-")
                flag = 1;
        }
        if (flag) {
            cout << '(';
            inOrder(p->leftChild);
            cout << ')';
        } else inOrder(p->leftChild);
    }
    cout << p->data;
    if (p->rightChild != NULL) {
        BinTreeNode<string> *ri = p->rightChild;
        bool flag = 0;
        if (p->data == "*" || p->data == "/") {
            if (ri->data == "+" || ri->data == "-")
                flag = 1;
        }
        if (p->data == "-") {
            if (ri->data == "+" || ri->data == "-")
                flag = 1;
        }
        if (p->data == "/") {
            if (ri->data == "*" || ri->data == "/")
                flag = 1;
        }
        if (flag) {
            cout << '(';
            inOrder(p->rightChild);
            cout << ')';
        } else inOrder(p->rightChild);
    }
}
```

2.1.4 数据测试

对于题目给定的表达式 a-(b-c)/(e*(f+g)) 进行处理，如下图分别输出其后缀表达式形式(postOrder)、中缀表达式形式（inOrder），并且按照树形结构进行答应，并且此程序对于表达式形式进行了兼容性处理，使其在最外层有多层括号时也可以得到正确的结果。如右侧下图对于表达式 (((a-(b-c)/(e*(f+g))))) 的处理

```
==================work()===============
==================getString()===============
Input the expression:a-(b-c)/(e*(f+g))
 ==================getString()===============

==================work finish===============

print as post order: a b c - e f g + * / -
print as in order: a-(b-c)/(e*(f+g))

        g
      +
        f
    *
      e
  /
      c
    -
      b
-
  a
```

```
==================work()===============
==================getString()===============
Input the expression:(((a-(b-c)/(e*(f+g)))))
 ==================getString()===============

==================work finish===============

print as post order: a b c - e f g + * / -
print as in order: a-(b-c)/(e*(f+g))

        g
      +
        f
    *
      e
  /
      c
    -
      b
-
  a
```

并且，此程序不仅仅按照实验题目要求，可以处理整数表达式的运算，并且此程序还可以对于负数表达式进行处理。（如右下图）

```
==================work()===============
==================getString()===============
Input the expression:3 / (4 - 2)
 ==================getString()===============

==================work finish===============

print as post order: 3 4 2 - /
print as in order: 3/(4-2)
answer: 1.5

    2
  -
    4
/
  3
```

```
==================work()===============
==================getString()===============
Input the expression:((-3) / (4 - 2))
 ==================getString()===============

==================work finish===============

print as post order: (-3) 4 2 - /
print as in order: (-3)/(4-2)
answer: -1.5

    2
  -
    4
/
  (-3)
```

3.1 课程设计中遇到的问题和解决方法

在中缀表达式建立表达数二叉树时时，我们小组进行了分工合作，经过了小组讨论与搜集资料，将任务分别处理成读取表达式、建立二叉树、表达式输出几个模块。我们对算法的时间复杂度进行了优化，特殊位置、临界位置进行了特殊处理，保证了程序的健壮性。

3.2 实验总结

验证性实验是有助于夯实基础，对于综合性实验，我们不仅仅应当注意是否能够解决问题，更要注重优化算法的时间复杂度。严谨性也同样重要，代码实现以后，一定要通过多组数据检验，确保其正确性。