# 数据结构实验报告三

## 题目： 二叉树及其应用

班级：计科一班　　姓名：范文　　学号： PB18111679　　完成日期：2019.11.14

### 实验一： 二叉树的创建和遍历

1. **实验要求：**
   通过添加虚结点，为二叉树的每一实结点补足其孩子，再对补足虚结点后二叉树按层次遍历的次序输入，构建这颗二叉树（不包含图中的虚结点），并增加左右标志域，将二叉树后序线索化。

2. **设计思路：**
   使用一个队列来层次序创建一个二叉树，首先用递归的方法输出其前序，中序和后序。接下来，用递归的方法确定每个结点的左右标志符和左右结点的指向，从而完成中序线索化。

3. **核心代码：**
   首先，建立线索二叉树的结点

```
// the threading binary tree for the char

enum PointerTag

{

    link,

    thread

};

struct BiThrTNode

{

    char data;

    BiThrTNode *lchild, *rchild;

    PointerTag Ltag, Rtag;

};
```

接着创建线索二叉树的 ADT

```
class MyBiThrTree

{

    private:
```

```
        BiThrTNode *root;
        BiThrTNode *thrt;

    public:
        /*  since the constructor can not be in a recursive process,
        we need to put another recursive function in it. */
         MyBiThrTree()
        {
            root = new BiThrTNode;
            thrt = new BiThrTNode;
        }
        MyBiThrTree(const string &inputString)
        {
            root = levelOrderCreate(inputString,'#');
            thrt = new BiThrTNode;
        }
        void pre_Order_Traverse() const   { preOrderTraverse(root);cout << 
endl;}
        void in_Order_Traverse() const  { inOrderTraverse(root);cout << end
l;}
        void post_Order_Traverse() const  { postOrderTraverse(root);cout <<
 endl;}
        void level_Order_Traverse() const { levelOrderTraverse(root);cout <
< endl;}
        void inOrderThrTraverse();
        BiThrTNode *get_root() const { return root; }
        BiThrTNode *get_thrt() const { return thrt; }
        ~MyBiThrTree()    { deleteBiThrTNode(root); }
};
```

其中十分重要的就是创建函数，遍历函数以及线索化函数，下面一一介绍。

（1） **创建树的函数**。由于是使用层序创建，因此可以使用队列的非递归方法来实现。

```
/* using recursion to create a binary tree in a levelOrder sequence.
a node can store a char, and using the invalidchar(such as '#') as a separate character.*/
BiThrTNode* levelOrderCreate(const string & inputString,const char & invalidChar = '#')
{
    int index = 0;
    queue<BiThrTNode*> nodeQueue;
    if ( inputString[index]  == invalidChar)
        return nullptr;
    BiThrTNode *tempNode1 = new BiThrTNode;
    tempNode1->Ltag = link;
    tempNode1->Rtag = link;
    BiThrTNode *rootNode = tempNode1;
    tempNode1->data = inputString[index];
    nodeQueue.push(tempNode1);
    while( !nodeQueue.empty() )
    {
        tempNode1 = nodeQueue.front();
        nodeQueue.pop();
        //consider the left child
        ++index;
        if( inputString[index] == invalidChar )
        {
            tempNode1->lchild = nullptr;
        }
```

```
        else
        {
            BiThrTNode *tempNode2 = new BiThrTNode;
            tempNode2->Ltag = link;
            tempNode2->Rtag = link;
            tempNode2->data = inputString[index];
            tempNode1->lchild = tempNode2;
            nodeQueue.push(tempNode2);
        }
        //consider the right child
        ++index;
        if( inputString[index] == invalidChar )
        {
            tempNode1->rchild = nullptr;
        }
        else
        {
            BiThrTNode *tempNode2 = new BiThrTNode;
            tempNode2->Ltag = link;
            tempNode2->Rtag = link;
            tempNode2->data = inputString[index];
            tempNode1->rchild = tempNode2;
            nodeQueue.push(tempNode2);
        }
    }//while( !nodeQueue.empty() )
    return rootNode;
}
```

**(2)三种递归遍历：**递归方法很简单，只是要在递归前先判断一下标志符，只有为 link 说明这个节点的儿子结点是正常的；如果是 thread，说明儿子结点是经过线索化之后的前驱或者后继，就不予考虑。

如先序遍历:

```
//a recursive function to traverse the tree in a preorder.
//sequence is 1.root    2.left child    3.right child
void preOrderTraverse(BiThrTNode *T)
{
    if(T)
    {
        visit(T->data);
        if(T->Ltag == link )
            preOrderTraverse(T->lchild);
        if(T->Rtag == link)
            preOrderTraverse(T->rchild);
    }
}
```

中序遍历:

```
//a recursive function to traverse the tree in an inorder.
//sequence is 1.left child    2.root    3.right child
void inOrderTraverse(BiThrTNode *T)
{
    if(T)
    {
        if(T->Ltag == link)
            inOrderTraverse(T->lchild);
        visit(T->data);
        if(T->Rtag == link)
            inOrderTraverse(T->rchild);
    }
}
```

后序遍历:

```
    //a recursive function to traverse the tree in a postorder.
```

```
//sequence is 1.left child    2.right child      3.root
void postOrderTraverse(BiThrTNode *T)
{
    if(T)
    {
        if(T->Ltag == link)
            postOrderTraverse(T->lchild);
        if(T->Rtag == link)
            postOrderTraverse(T->rchild);
        visit(T->data);
    }
}
```

**（3）线索化二叉树的实现**

递归函数 inthreading，用来将 orig_Ptr 结点的子树给线索化。

```
// the main recursive function to build the threading
//binary tree.
void inThreading(BiThrTNode *origPtr)
{
    if(origPtr)
    {
        //firstly, build the threading for the left subtree.
        inThreading(origPtr -> lchild);
        // we can put the prior thread into its leftchild.
        if( origPtr -> lchild == nullptr)
        {
            origPtr -> Ltag = thread;
            origPtr -> lchild = pre;
        }
        // be careful, the initialized value for the Tag may be thread,
        //so we must change it to link.
```

```
        else
        {
            origPtr->Ltag = link;
        }
        //we can put the next thread into its rightchild.
        if( pre->rchild == nullptr)
        {
            pre->Rtag = thread;
            pre->rchild = origPtr;
        }
        pre = origPtr;
        //thirdly, build the threading for the right subtree.
        inThreading(origPtr->rchild);
    }//if(origPtr)
}
```

创建二叉树的头结点，并利用之前的 inthreading 函数将一个二叉树线索化。

```
//input: a binary tree without being threaded,(seems like a normal one)
//output: a binary tree that has kept its prior and next nodes in inorder t
raverse
BiThrTNode* inOrderThreading(BiThrTNode *rootNode,BiThrTNode *headNode)
{
    //setting a head node. its left child points to the root,
    //right child points to the last node in inorder traverse.
    //ALSO, the first node and the last node in inorder traverse will point
 to the head.
    headNode->Ltag = link;
    headNode->Rtag = thread;
    headNode->rchild = headNode;
    if(rootNode == nullptr)
```

```
    {
        headNode->lchild = headNode;
    }
    else
    {
        //using pre as the global previous node for
        headNode->lchild = rootNode;
        pre = headNode;
        inThreading(rootNode);
        // a little revise for the last node.
        pre->rchild = headNode;
        pre->Rtag = thread;
        headNode->rchild = pre;
    }
    return headNode;
}
```

最后就是中序线索化遍历函数，由于之前的线索化，因此这个函数是非递归的。

```
//using the thread binary tree to inOrder traverse in
// - a non recursive way.
//once we have let it with thread, it is not ok to use the recursive
// traverse. since the lchild  and rchild have both changed.
void MyBiThrTree::inOrderThrTraverse()
{
    thrt = inOrderThreading(root,thrt);
    BiThrTNode *tempPtr = thrt -> lchild;
    while( tempPtr != thrt)
    {
        while( tempPtr -> Ltag == link)
        {
```

```
            tempPtr = tempPtr -> lchild;
        }
        // go to the bottom of the left child
        visit(tempPtr -> data);
        while(tempPtr -> Rtag == thread && tempPtr -> rchild != thrt)
        {
            tempPtr = tempPtr -> rchild;
            visit(tempPtr->data);
        }
        tempPtr = tempPtr -> rchild;
    }
}
```

因此，当输入测试数据 ABC#DEFG##H###### 时，得到了结果：

```
please input a string.
ABC#DEFG##H######
finish constructing.
now traversing the tree in a preorder way.
A B D G C E H F

now traversing the tree in a inorder way.
B G D A E H C F

now traversing the tree in a postorder way.
G D B H E F C A

now travsering the tree in a threading inorder sequence.
B G D A E H C F
```

符合预期。

## 实验二：表达式树

1. **实验要求：**
   建立一个二叉树来储存表达式，并可以求值以及输出一个不带多余括号的中缀表达式，因此需要判定什么情况下应该输出括号。

2. **设计思路：**
   首先构建 ExpreTree 类并让其继承 MyBiTree 类。另外，修改读入数据的方式为以空格为分界符，用 string 来储存数字和操作符。其次，用递归的方法将表达式拆分为若干项，计算每个项的值。判断是否要加括号时（例如 A B C 三个运算符），应该是：当优先级 A < B 时，给 A 连接的项加括号；当优先级 C <= B 时，给 C 连接

的项加括号,其余情况都不加括号。

3. **核心代码**:

**构造派生类 ExpreTree 的函数**: 每一项用 string 保存，分隔符为空格。

```
//input a prefix expression and create the expressionTree
//supposed that the prefix expression is valid
//- and that the operators in the prefix expression are all binary.
// the seperation sign is the blank space.
BiTNode *pre_order_create( string &inputString)
{
    BiTNode *tempNode = nullptr;
    //the traverse of the inputstring is over, tree is built.
    if(inputString == "")
      return nullptr;
    int sepeIndex = inputString.find(' ', 0);
    if(sepeIndex < 0)
    {
      sepeIndex = inputString.length();
    }
    //s is the content between the start and the seperation sign.
    string s = inputString.substr(0, sepeIndex);
    tempNode = new BiTNode;
    tempNode->data = s;
    // s is a number,it must be a leaf node
    if( operator_level(s) == 0 )
    {
      tempNode->lchild = nullptr;
      tempNode->rchild = nullptr;
    }
    // ch is an operator
    else
    {
```

```
      //let inputSTring be the string cutting off the previous content.
      // if the previous content is the end , the new inputString is ""
      inputString = inputString.substr(sepeIndex + 1);
      tempNode->lchild = pre_order_create(inputString);
      //later,consider the right child in the same way.
      sepeIndex = inputString.find(' ', 0);
      if( sepeIndex < 0 )
        sepeIndex = inputString.length()-1;
      inputString = inputString.substr(sepeIndex + 1);
      tempNode->rchild = pre_order_create(inputString);
    }
    return tempNode;
}
```

**中序输出，添加括号的函数：**每个结点要为它的左右儿子结点考虑是否添加括号的问题。

```
/*  only add the necessary bracket for the infix expression. */
// if flagOfAddBracket is true, the expression connected by the T optr is
//- required to add bracket.
// also ,we can figure out the flagofAddBracket for the next subExpression
by
//-comparing the priority.
void inOrderTraverse_addBracket_concise(BiTNode *T,bool flagOfAddBracket)
{
  // such as the case A--B--C
  if (T)
  {
    //consider for the A(left subexpression.)
    bool tempFlagOfAddBracket = false;
    //1. B is a number, don't add bracket for A
```

```
if (operator_level(T->data) == 0 )
    tempFlagOfAddBracket = false;
else if ( operator_level(T->data) > 0 )
{
  //2. B is an optr but its priority <= A's priority(A is alos an optr)
  // don't add bracket for A
    if( operator_level(T->lchild->data) >= operator_level(T->data) )
         tempFlagOfAddBracket = false;
  // 3. B is an optr and A is a number, don't add bracket for A.
    else if( operator_level(T->lchild->data) == 0 )
        tempFlagOfAddBracket = false;
  //4. B is optr , A is optr and priority : B > A
  // add bracket for AA.
    else
      tempFlagOfAddBracket = true;
}
// 1,2,3 are all the cases for add bracket for left subexpression,
// for the rest, we need to add bracket.
else
{
    tempFlagOfAddBracket = true;
}

if( flagOfAddBracket)
    cout << "( ";
inOrderTraverse_addBracket_concise(T->lchild,tempFlagOfAddBracket);
visit(T->data);
//considering the right subexpression,similar to the left subexpression
if (operator_level(T->data) == 0 )
    tempFlagOfAddBracket = false;
else if ( operator_level(T->data) > 0 )
```

```
    {
      // the only difference: B and C are both optr and
      // -we DON'T add bracket only when the priority: C > B
      // if priority: C = B, we had better add bracket
      //- because it shows the computing sequence.
        if( operator_level(T->rchild->data) > operator_level(T->data) )
             tempFlagOfAddBracket = false;
        else if(operator_level(T->rchild->data) == 0)
            tempFlagOfAddBracket = false;
        else
            tempFlagOfAddBracket = true;
    }
    else
    {
        tempFlagOfAddBracket = true;
    }

    inOrderTraverse_addBracket_concise(T->rchild,tempFlagOfAddBracket);

    if(flagOfAddBracket == true)
      cout << ") ";
  }
}
```

**表达式求值的函数**如下：注意要把字符串转化为对应的实数。我这里调用了 stringstream 头文件实现的。

```
//calculate the value of that expression.
//using the tree and the recursion.(from root to leaves)
double calculate_value(BiTNode *T)
{
    if(T)
```

```
    {
        if( T->data == "+" )
          return calculate_value(T->lchild) + calculate_value(T->rchild);
        else if ( T->data == "-")
          return calculate_value(T->lchild) - calculate_value(T->rchild);
        else if( T->data == "*")
          return calculate_value(T->lchild) * calculate_value(T->rchild);
        else if( T->data == "/")
          return calculate_value(T->lchild) / calculate_value(T->rchild);
        // the data is a number, so we need to transform it from the string
to double.
        else
        {
            double number;
            stringstream ss;
            ss << T->data;
            ss >> number;
            return number;
        }
    }//if(T)
    else
      return 0;
}
```

然后输入测试数据 / + 15 * 5 + 2 18 5 得到结果:

```
please input a string.
/ + 15 * 5 + 2 18 5
the inputString is / + 15 * 5 + 2 18 5
finish constructing.
now getting the prefix expression.
/ + 15 * 5 + 2 18 5
now getting the infix expression.
( 15 + ( 5 * ( 2 + 18 ) ) ) / 5
now getting the postfix expression.
15 5 2 18 + * + 5 /
the value of the expression is 23
now getting the infix expression with concise brackets.
( 15 + 5 * ( 2 + 18 ) ) / 5
```

符合预期。

## 三．实验总结

本次实验我尝试用递归构造数据结构二叉树，并且将其遍历出来和线索化，因此对树的结构有着更加深刻的理解。在树的应用上，我发现前缀、中缀和后缀表达式在二叉树中更加直观易懂，而且用二叉树递归进行表达式求值比用栈迭代方便不少。同时，从线索二叉树的非递归遍历实现也可以看出递归与非递归的显著区别：递归节省空间却浪费时间，非递归更快速但需要更多内存储存相应的中间值（比如 Ltag 和 Rtag）。

## 四．源代码附件

thdBiTree.h 存放线索二叉树 ADT 的头文件

thdBiTree_test.cpp 线索化二叉树的测试文件

myBiTree.h 存放一般二叉树 ADT 的头文件

expression.h 存放派生类 ExpreTree 的头文件

expre_test 表达式树的测试文件