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：王章瀚

学生学号：PB18111697

完成日期：2019 年 11 月 14 日

计算机实验教学中心制

2019 年 09 月

1 实验要求

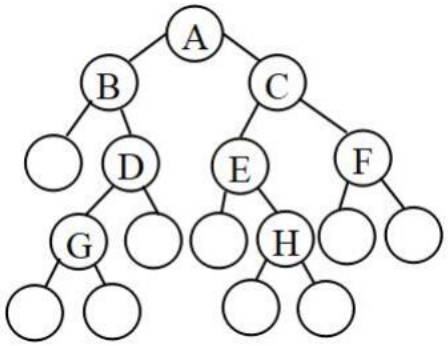
本次实验分为两个小实验。

1.1 二叉树的创建与遍历

概述

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。

例如：ABC#DEFG##H######



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

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。

输入与输出

样例如下：

```
Input:
ABC#DEFG##H######
output:
ABDGCEHF
BGDAEHCF
GDBHEFCA
```

1.2 表达式树

概述

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。

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

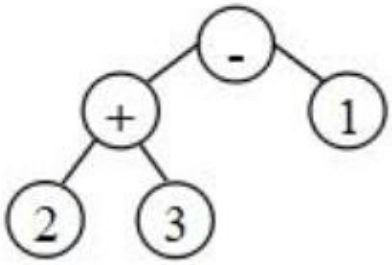
完成后序线索化树上的遍历算法，依次输出该二叉树先序遍历、中序遍历和后序遍历的结果。

输入与输出

样例如下：

```
Input:
- + 2 3 1 //波兰式
Output:
(2+3)-1 //中缀表达式
2 3 + 1 - //逆波兰式
4 //求值
```

对应表达式树如下：

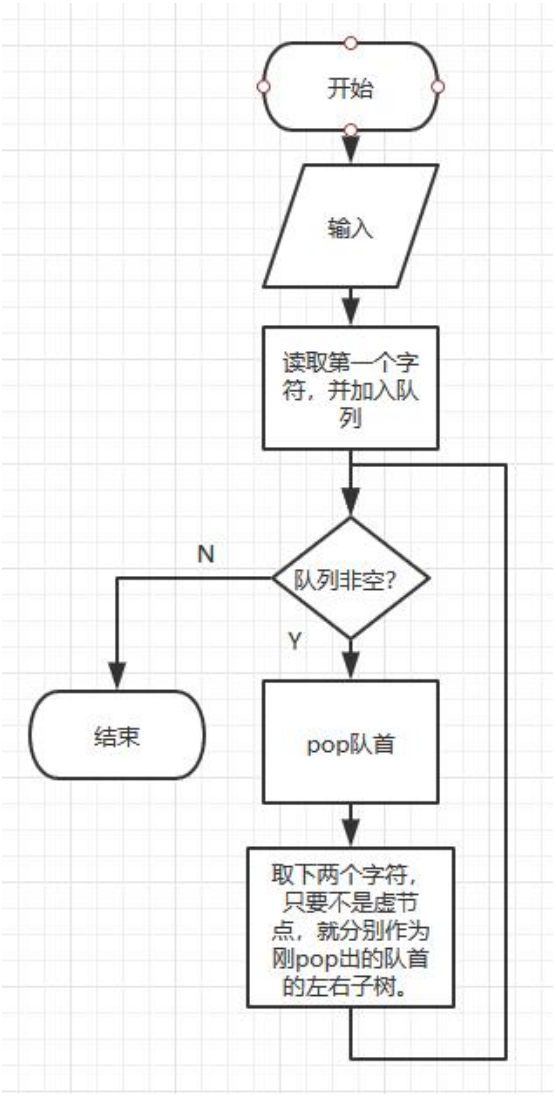


2 设计思路

2.1 二叉树的创建与遍历

1. 构建

对于这样给出的层序遍历输入，可以按照如下算法来完成建树。为方便理解，使用流程图来描述：



2. 后序线索化

后序线索化的方法主要是按以下几点来进行的：

• 用 pre 来记录上一个遍历的结点

• 递归过程中，首先对左子树递归，再对右子树递归。

• 其后若本节点左子树指向空，则将本结点的左子树设置为 pre；若 pre 的右子树指向空，则将 pre 的右子树置为当前节点。

3. 后序线索化后的后序遍历

对于二叉树的后序线索化的遍历，需要添加双亲节点指针，构建三叉链表才能完成。

后序线索化遍历的过程主要以下面几个准则来进行：

- 若结点的右子树指向后继结点，则可以直接进行访问
- 若结点的右子树不指向后继结点，则后继结点应为“对其双亲节点的右子树作后序遍历的第一个结点”

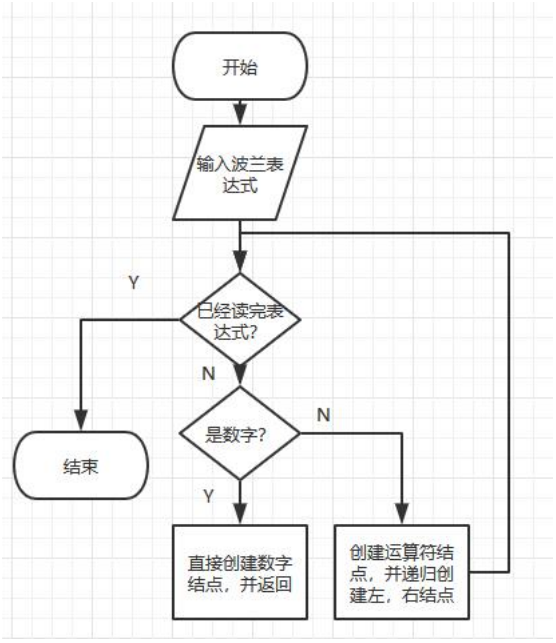
4. 先序遍历与中序遍历

采用递归遍历，较为简单，故不做赘述。

2.2 表达式树

1. 构建

可以按照如下算法来完成建树。为方便理解，使用流程图来描述：



2. 输出波兰表达式或逆波兰表达式

只需要对表达式树进行先序遍历或后序遍历即可完成，较为简单，故不赘述。

3. 输出中缀表达式

主要思路是中序遍历表达式。但值得注意的是怎么确保不加多余的括号。

如果当前结点是'*' 或'/'，则查看左右结点是否为'+' 或'-'，如果是，则输出时加上括号；如果不是则不必。

4. 由逆波兰表达式构建表达式树

对输入字符串的遍历改为从后到前，并且先递归构建当前节点，再右子树，再左子树，即可完成。和 **1. 构建**中的描述比较类似。

3 关键代码讲解

3.1 二叉树的创建与遍历

1. 数据类型的定义

```
class BTreeNode {
private:
    BTreeNode* pre = nullptr; //前一个遍历的结点
public:
    static const char IMAGE_NODE = '#'; //表示虚节点的字符
    char data;
    BTreeNode* left = nullptr, * right = nullptr, * parent = nullptr;
    bool LTag, RTag;

    //构建一个结点，其中数据为data，左、右、双亲节点为left，right，
parent
    BTreeNode(char data, BTreeNode* left, BTreeNode* right, BTreeNode*
parent);
    //参数为题述字符串，构建对应树
    BTreeNode(string layerIter);
    //先序遍历
    static void PreOrderTraverse(BTreeNode* t, void function(BTreeNode*)
);
    //中序遍历
    static void InOrderTraverse(BTreeNode* t, void function(BTreeNode*))
;
    //后序遍历
    static void PostOrderTraverse(BTreeNode* t, void function(BTreeNode
*));
    //后序线索化
    void PostOrderThreading(BTreeNode* t);
    //利用后序线索化的遍历
    void PostOrderThreadingTraverse(void function(BTreeNode*));
};
```

```

```

2. 主要算法

```
//建树函数
BTreeNode::BTreeNode(string layerIter) : LTag(false), RTag(false) {
    if (layerIter[0] == IMAGE_NODE) {
        data = IMAGE_NODE;
        cout << "Not a tree!" << endl;
    }
    else {
        //先做好根节点
        data = layerIter[0];
        queue<BTreeNode*> lastLayer;
        lastLayer.push(this);
        int nodeNum = 2;
        int index = 1;
        int strlen = layerIter.length();
        //用index逐一访问字符串layerIter
        while (index < strlen) {
            int qSize = lastLayer.size();
            for (int i = 0; i < qSize; i++) {
                BTreeNode* curNode = lastLayer.front();
                lastLayer.pop();
                if (layerIter[index] != IMAGE_NODE) {
                    //如果下一个字符不表示虚节点
                    curNode->left =
                    new BTreeNode(layerIter[index], nullptr, nullptr,
curNode);
                    lastLayer.push(curNode->left);
                }
                else //否则是虚节点
                curNode->left = nullptr;
                index++;
                if (layerIter[index] != IMAGE_NODE) {
                    //如果下一个字符不表示虚节点
                    curNode->right =
                    new BTreeNode(layerIter[index], nullptr, nullptr,
curNode);
                    lastLayer.push(curNode->right);
                }
                else //否则是虚节点
                curNode->right = nullptr;
                index++;
            }
```

```
            }
        }
    };

```

建树函数

```
    //后序线索化函数
    void BTreeNode::PostOrderThreading(BTreeNode* t) {
        if (t == nullptr)
        return;
        //对左子树线索化
        PostOrderThreading(t->left);
        //对右子树线索化
        PostOrderThreading(t->right);
        //对本结点和pre线索化
        if (t->left == nullptr) {
            t->left = pre;
            t->LTag = true;
        }
        if (pre != nullptr && pre->right == nullptr) {
            pre->RTag = true;
            pre->right = t;
        }
        pre = t;
    }

```

后序线索化函数

```
    //后序线索化后的遍历函数
    void BTreeNode::PostThreadingTraverse(void function(BTreeNode*)) {
        BTreeNode* temp = this;
        // 找后序遍历的第一个结点
        while (!temp->LTag || !temp->RTag) {
            while (!temp->LTag)
            temp = temp->left;
            while (!temp->RTag)
            temp = temp->right;
        }
        function(temp);
```

```
12
13          while (temp != nullptr && temp->parent != nullptr) {
14              //如果该结点是双亲结点的右孩子或双亲节点没有右孩子
15              if (temp->parent->right == temp
16              || temp->parent->RTag) {
17                  temp = temp->parent;
18              }
19              else {
20                  //否则后继结点是双亲节点的右子树中后序遍历第一个结点
21                  temp = temp->parent->right;
22                  // 找后序遍历的第一个结点
23                  while (!temp->LTag || !temp->RTag) {
24                      while (!temp->LTag)
25                      temp = temp->left;
26                      while (!temp->RTag)
27                      temp = temp->right;
28                  }
29              }
30              function(temp);
31          }
32      }
33
34
35      //启动函数
36      void BTreeNode::PostOrderThreading(BTreeNode* t) {
37          PostThreading(t);
38          if (pre != nullptr && pre->right == nullptr) {
39              pre->RTag = true;
40              pre->right = t;
41          }
42      }
43
```

后序线索化后的遍历函数及启动函数

3.2 表达式树

1. 数据类型的定义

```
1       class ExpressionTree {
2           private:
3           //由波兰表达式建树
4           ExpressionTree* createTree_P(stringstream& ss, ExpressionTree* t);
5           //由逆波兰表达式建树
6           ExpressionTree* createTree_IP(stringstream& ss, ExpressionTree* t);
7           public:
```

```
        string op = IMAGE_NODE; //结点的运算符（若为运算符）
        string operand; //结点的数值（若为数）
        bool isOperand = false;
        ExpressionTree* left = nullptr, * right = nullptr;
        const string IMAGE_NODE = "#";
        bool isValueGetted = false;
        int value; //子树值

        //通过data和是否是操作数来创建一个结点
        ExpressionTree(string data, bool isOperand);
        //by为1则通过波兰表达式建树，by为0则通过逆波兰表达式建树
        ExpressionTree(string polish, int by);

        //打印中缀表达式
        static void printInfixExpression(ExpressionTree* t);
        //打印波兰表达式
        static void printPolishExpression(ExpressionTree* t);
        //打印逆波兰表达式
        static void printInversePolishExpression(ExpressionTree* t);
        //计算各子树的值
        static int getValue(ExpressionTree* t);
    };

```

表达式树数据类型定义

2. 主要算法

思路和前述流程图一样，故不赘述。

```
ExpressionTree* ExpressionTree::createTree_P(stringstream& ss,
ExpressionTree* t) {
    string polish;
    ss >> polish;
    if (polish[0] == '+' || polish[0] == '-'
    || polish[0] == '*' || polish[0] == '/') {
        if (t != this)
        t = new ExpressionTree(polish, false);
        else {
            t->isOperand = false;
            t->isValueGetted = false;
            t->op = polish[0];
        }
        if (!ss.eof()) {
            t->left = createTree_P(ss, t->left);
            t->right = createTree_P(ss, t->right);
        }
    }
```

```
        else {
            t = new ExpressionTree(polish, true);
        }
        return t;
    }

```

通过波兰表达式建树

```
    void ExpressionTree::printInfixExpression(ExpressionTree* t) {
        if (t != nullptr) {
            //处理左子树
            if (t->left->isOperand) {
                cout << t->left->operand;
            }
            else {
                if ((t->left->op == "+" || t->left->op == "-")
                && (t->op == "*" || t->op == "/")) {
                    cout << '(';
                    printInfixExpression(t->left);
                    cout << ')';
                }
                else
                printInfixExpression(t->left);
            }

            //处理当前结点
            if (t->isOperand) {
                cout << t->operand;
            }
            else {
                cout << t->op;
            }

            //处理右子树
            if (t->right->isOperand) {
                cout << t->right->operand;
            }
            else {
                if ((t->right->op == "+" || t->right->op == "-")
                && (t->op == "*" || t->op == "/")) {
                    cout << '(';
                    printInfixExpression(t->right);
                    cout << ')';
```

```
            }
            else
            printInfixExpression(t->right);
        }
    }
}

```

输出中缀表达式的函数

```
int ExpressionTree::getValue(ExpressionTree* t) {
    if (t->isValueGetted)
    return t->value;
    else {
        if (t->op == "+") {
            t->value = getValue(t->left) + getValue(t->right);
            t->isValueGetted = true;
            return t->value;
        }
        else if (t->op == "-") {
            t->value = getValue(t->left) - getValue(t->right);
            t->isValueGetted = true;
            return t->value;
        }
        else if (t->op == "*") {
            t->value = getValue(t->left) * getValue(t->right);
            t->isValueGetted = true;
            return t->value;
        }
        else if (t->op == "/") {
            t->value = getValue(t->left) / getValue(t->right);
            t->isValueGetted = true;
            return t->value;
        }
    }
}

```

计算表达式树值

4 调试分析

4.1 二叉树的创建与遍历

问题发现与解决

实验过程中发现自己对于线索二叉树不甚熟悉，翻查各方资料后方能较好地完成线索化。

算法的时间复杂度分析

不论是建树还是遍历，均几乎只访问各个结点一次，时间复杂度接近 o(n).

4.2 表达式树

问题发现与解决

有了前一个实验的基础，本实验做起来比较简单，没有什么大问题。

算法的时间复杂度分析

同样的，各个结点均几乎只访问一次，时间复杂度接近 o(n).

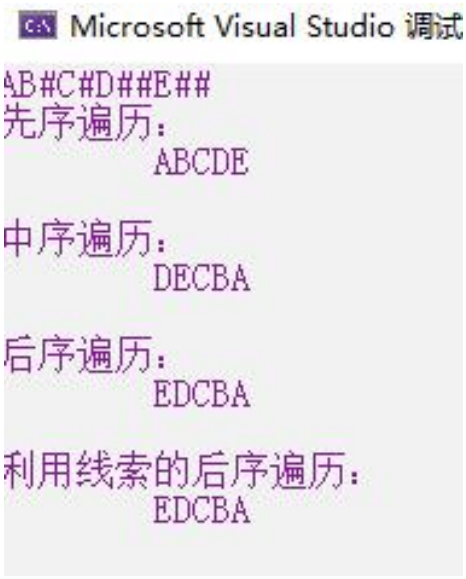
5 代码测试

5.1 二叉树的创建与遍历

输入 1：

```
AB#C#D##E##
```

输出截图 1：



输入 2：

A#BC##DE##F##

输出截图 2：

输入 3：

```
ABCD##E##F###
```

输出截图 3：

5.2 表达式树

输入 1：

```
+ 2 * 30 5
```

输出截图 1：

输入 2：

```
/ + 15 * 5 + 2 18 5
```

输出截图 2：

输入 3：

```
- / 15 - 2 7 10
```

输出截图 3：

6 实验总结

首先回答几个思考题。

1. 给定先序和中序序列、中序和后序序列、先序和后序序列均可以唯一确定一棵树。

• 先序 + 中序：逐一遍历先序序列的结点，对当前遍历的结点，在中序序列中可以知道其左右子树中的结点（在该结点左边就是左子树结点，右边就是右子树结点），如此递归即可确定下一棵树。

• 中序 + 后序：逐一遍历后序序列的结点，类似于“先序 + 中序”的情况，也可以解决。

• 先序 + 后序：逐一遍历先序序列中的结点，可以其剩余序列分为左子树和右子树，同理在后序序列中找到该结点，其前面的序列也分为该节点的左子树和右子树。只需找到对应长度的序列，即可确定该节点的左右子树，从而递归地完成建树。

2. 表达式树的先序序列对应波兰表达式；中序序列不考虑括号则对应中缀表达式是；后序序列对应逆波兰表达式。

3. 给定波兰式、中缀表达式或逆波兰式中的任意一种，可以唯一确定一颗表达式树。造成这种差异的原因是：表达式树的叶子只能是数值，数值也只能在叶子上，而符号则在分支上。这样就给出了足够多的约束条件。

本次实验让我深刻了解了二叉树的创建，线索化，及线索化遍历。此外，还以此为基础，完成了表达式树的一些操作。虽然实验任务量相对较大，但学到的东西也更多。

7 附录

7.1 附录 A. 二叉树的创建与遍历

```
#include <iostream>
#include <string>
#include <queue>
#pragma once

using namespace std;

class BTreeNode {
    private:
    BTreeNode* pre = nullptr; //前一个遍历的结点
    public:
    static const char IMAGE_NODE = '#'; //表示虚节点的字符
    char data;
    BTreeNode* left = nullptr, * right = nullptr, * parent = nullptr;
    bool LTag, RTag;

    //构建一个结点，其中数据为data，左、右、双亲节点为left，right，
parent
    BTreeNode(char data, BTreeNode* left, BTreeNode* right, BTreeNode*
parent);
```

```
19          //参数为题述字符串，构建对应树
20          BTreeNode(string layerIter);
21          //先序遍历
22          static void PreOrderTraverse(BTreeNode* t, void function(BTreeNode*)
        );
23          //中序遍历
24          static void InOrderTraverse(BTreeNode* t, void function(BTreeNode*))
        ;
25          //后序遍历
26          static void PostOrderTraverse(BTreeNode* t, void function(BTreeNode
        *));
27          //后序线索化子程序
28          void PostThreading(BTreeNode* t);
29          //后序线索化
30          void PostOrderThreading(BTreeNode* t);
31          //利用后序线索化的遍历
32          void PostOrderThreadingTraverse(void function(BTreeNode*));
33      };
34
35
36      BTreeNode::BTreeNode(char data, BTreeNode* left, BTreeNode* right,
        BTreeNode* parent)
37      : data(data), left(left), right(right), parent(parent),
38      LTag(false), RTag(false) {};
39
40      BTreeNode::BTreeNode(string layerIter) : LTag(false), RTag(false) {
41          if (layerIter[0] == IMAGE_NODE) {
42              data = IMAGE_NODE;
43              cout << "Not a tree!" << endl;
44          }
45          else {
46              //先做好根节点
47              data = layerIter[0];
48              queue<BTreeNode*> lastLayer;
49              lastLayer.push(this);
50              int nodeNum = 2;
51              int index = 1;
52              int strlen = layerIter.length();
53              //用index逐一访问字符串layerIter
54              while (index < strlen) {
55                  int qSize = lastLayer.size();
56                  for (int i = 0; i < qSize; i++) {
57                      BTreeNode* curNode = lastLayer.front();
58                      lastLayer.pop();
59                      if (layerIter[index] != IMAGE_NODE) {
60                          //如果下一个字符不表示虚节点
```

```
61                          curNode->left =
62                          new BTreeNode(layerIter[index], nullptr, nullptr,
    curNode);
63                          lastLayer.push(curNode->left);
64                      }
65                      else //否则是虚节点
66                      curNode->left = nullptr;
67                      index++;
68                      if (layerIter[index] != IMAGE_NODE) {
69                          //如果下一个字符不表示虚节点
70                          curNode->right =
71                          new BTreeNode(layerIter[index], nullptr, nullptr,
    curNode);
72                          lastLayer.push(curNode->right);
73                      }
74                      else //否则是虚节点
75                      curNode->right = nullptr;
76                      index++;
77                  }
78              }
79          }
80      };
81
82      void BTreeNode::PreOrderTraverse(BTreeNode* t, void function(BTreeNode*)
    ) {
83          if (t != nullptr)
84          {
85              function(t);
86              if (t->LTag == 0)
87              PreOrderTraverse(t->left, function);
88              if (t->RTag == 0)
89              PreOrderTraverse(t->right, function);
90          }
91      }
92
93      void BTreeNode::InOrderTraverse(BTreeNode* t, void function(BTreeNode*))
     {
94          if (t != nullptr)
95          {
96              if (t->LTag == 0)
97              InOrderTraverse(t->left, function);
98              function(t);
99              if (t->RTag == 0)
100             InOrderTraverse(t->right, function);
101         }
102     }
```

```
103
104     void BTreeNode::PostOrderTraverse(BTreeNode* t, void function(BTreeNode
        *)) {
105         if (t != nullptr)
106         {
107             if (t->LTag == 0)
108             PostOrderTraverse(t->left, function);
109             if (t->RTag == 0)
110             PostOrderTraverse(t->right, function);
111             function(t);
112         }
113     }
114
115     void BTreeNode::PostThreading(BTreeNode* t) {
116         if (t == nullptr)
117         return;
118         //对左子树线索化
119         PostThreading(t->left);
120         //对右子树线索化
121         PostThreading(t->right);
122         //对本结点和pre线索化
123         if (t->left == nullptr) {
124             t->left = pre;
125             t->LTag = true;
126         }
127         if (pre != nullptr && pre->right == nullptr) {
128             pre->RTag = true;
129             pre->right = t;
130         }
131         pre = t;
132     }
133
134     void BTreeNode::PostOrderThreading(BTreeNode* t) {
135         PostThreading(t);
136         if (pre != nullptr && pre->right == nullptr) {
137             pre->RTag = true;
138             pre->right = t;
139         }
140     }
141
142     void BTreeNode::PostOrderThreadingTraverse(void function(BTreeNode*)) {
143         BTreeNode* temp = this;
144         // 找后序遍历的第一个结点
145         while (!temp->LTag || !temp->RTag) {
146             while (!temp->LTag)
147             temp = temp->left;
```

```
148             while (!temp->RTag)
149                 temp = temp->right;
150         }
151         function(temp);
152 
153         while (temp != nullptr && temp->parent != nullptr) {
154             //如果该结点是双亲结点的右孩子或双亲节点没有右孩子
155             if (temp->parent->right == temp
156             || temp->parent->RTag) {
157                 temp = temp->parent;
158             }
159             else {
160                 //否则后继结点是双亲节点的右子树中后序遍历第一个结点
161                 temp = temp->parent->right;
162                 // 找后序遍历的第一个结点
163                 while (!temp->LTag || !temp->RTag) {
164                     while (!temp->LTag)
165                     temp = temp->left;
166                     while (!temp->RTag)
167                     temp = temp->right;
168                 }
169             }
170             function(temp);
171         }
172     }
173 
```

BTreeNode.h

```
1   #include <iostream>
2   #include "BTreeNode.h"
3   using namespace std;
4 
5   void printTree(BTreeNode* t)
6   {
7       if(t != nullptr)
8       cout << t->data;
9   }
10 
11  int main()
12  {
13      BTreeNode* t = &BTreeNode("ABC##DEFG##H#######");
14      BTreeNode::PreOrderTraverse(t, printTree);
15      cout << endl;
```

```
16          BTreeNode::InOrderTraverse(t, printTree);
17          cout << endl;
18          BTreeNode::PostOrderTraverse(t, printTree);
19          cout << endl;
20          t->PostOrderThreading(t);
21          t->PostOrderThreadingTraverse(printTree);
22      }
23
```

二叉树的创建与遍历 main.c

7.2 附录 B. 表达式树

```
1   #include <iostream>
2   #include <string>
3   #include <queue>
4   #include <cstring>
5   #pragma once
6
7   using namespace std;
8
9   class ExpressionTree {
10      private:
11      //由波兰表达式建树
12      ExpressionTree* createTree_P(stringstream& ss, ExpressionTree* t);
13      //由逆波兰表达式建树
14      ExpressionTree* createTree_IP(stringstream& ss, ExpressionTree* t);
15      public:
16      string op = IMAGE_NODE; //结点的运算符（若为运算符）
17      string operand; //结点的数值（若为数）
18      bool isOperand = false;
19      ExpressionTree* left = nullptr, * right = nullptr;
20      const string IMAGE_NODE = "#";
21      bool isValueGetted = false;
22      int value; //子树值
23
24      //通过data和是否是操作数来创建一个结点
25      ExpressionTree(string data, bool isOperand);
26      //by为1则通过波兰表达式建树，by为0则通过逆波兰表达式建树
27      ExpressionTree(string polish, int by);
28
29      //打印中缀表达式
30      static void printInfixExpression(ExpressionTree* t);
31      //打印波兰表达式
32      static void printPolishExpression(ExpressionTree* t);
```

```
33          //打印逆波兰表达式
34          static void printInversePolishExpression(ExpressionTree* t);
35          //计算各子树的值
36          static int getValue(ExpressionTree* t);
37      };
38
39      ExpressionTree::ExpressionTree(string data, bool isOperand) {
40          this->isOperand = isOperand;
41          if (isOperand) {
42              operand = data;
43              stringstream ss(data);
44              ss >> value;
45              isValueGetted = true;
46          }
47          else {
48              isValueGetted = false;
49              op = data[0];
50          }
51      }
52
53      ExpressionTree* ExpressionTree::createTree_P(stringstream& ss,
        ExpressionTree* t) {
54          string polish;
55          ss >> polish;
56          if (polish[0] == '+' || polish[0] == '-'
57          || polish[0] == '*' || polish[0] == '/') {
58              if (t != this)
59              t = new ExpressionTree(polish, false);
60              else {
61                  t->isOperand = false;
62                  t->isValueGetted = false;
63                  t->op = polish[0];
64              }
65              if (!ss.eof()) {
66                  t->left = createTree_P(ss, t->left);
67                  t->right = createTree_P(ss, t->right);
68              }
69          }
70          else {
71              t = new ExpressionTree(polish, true);
72          }
73          return t;
74      }
75
76      ExpressionTree* ExpressionTree::createTree_IP(stringstream& ss,
        ExpressionTree* t) {
```

```
77          string polish;
78          ss >> polish;
79          if (polish[0] == '+' || polish[0] == '-'
80          || polish[0] == '*' || polish[0] == '/') {
81              if (t != this)
82              t = new ExpressionTree(polish, false);
83              else {
84                  t->isOperand = false;
85                  t->isValueGetted = false;
86                  t->op = polish[0];
87              }
88              if (!ss.eof()) {
89                  t->right = createTree_IP(ss, t->right);
90                  t->left = createTree_IP(ss, t->left);
91              }
92          }
93          else {
94              t = new ExpressionTree(polish, true);
95          }
96          return t;
97      }
98
99      ExpressionTree::ExpressionTree(string polish, int by) {
100         if (by) {
101             stringstream ss(polish);
102             createTree_P(ss, this);
103         }
104         else {
105             int len = polish.length();
106             stringstream ss(polish);
107             string inversed;
108             while (!ss.eof()) {
109                 string temp;
110                 ss >> temp;
111                 inversed = temp + " " + inversed;
112             }
113             stringstream iss(inversed);
114             createTree_IP(iss, this);
115         }
116     }
117
118     void ExpressionTree::printInfixExpression(ExpressionTree* t) {
119         if (t != nullptr) {
120             //处理左子树
121             if (t->left->isOperand) {
122                 cout << t->left->operand;
```

```
123             }
124             else {
125                 if ((t->left->op == "+" || t->left->op == "-")
126                 && (t->op == "*" || t->op == "/")) {
127                     cout << '(';
128                     printInfixExpression(t->left);
129                     cout << ')';
130                 }
131                 else
132                 printInfixExpression(t->left);
133             }
134 
135             //处理当前结点
136             if (t->isOperand) {
137                 cout << t->operand;
138             }
139             else {
140                 cout << t->op;
141             }
142 
143             //处理右子树
144             if (t->right->isOperand) {
145                 cout << t->right->operand;
146             }
147             else {
148                 if ((t->right->op == "+" || t->right->op == "-")
149                 && (t->op == "*" || t->op == "/")) {
150                     cout << '(';
151                     printInfixExpression(t->right);
152                     cout << ')';
153                 }
154                 else
155                 printInfixExpression(t->right);
156             }
157         }
158     }
159 
160     void ExpressionTree::printPolishExpression(ExpressionTree* t)
161     {
162         if (t != nullptr) {
163             if (t->isOperand)
164             cout << t->operand << ' ';
165             else
166             cout << t->op << ' ';
167             printPolishExpression(t->left);
168             printPolishExpression(t->right);
```

```
169         }
170     }
171
172     void ExpressionTree::printInversePolishExpression(ExpressionTree* t)
173     {
174         if (t != nullptr) {
175             printInversePolishExpression(t->left);
176             printInversePolishExpression(t->right);
177             if (t->isOperand)
178             cout << t->operand << ' ';
179             else
180             cout << t->op << ' ';
181         }
182     }
183
184     int ExpressionTree::getValue(ExpressionTree* t) {
185         if (t->isValueGetted)
186         return t->value;
187         else {
188             if (t->op == "+") {
189                 t->value = getValue(t->left) + getValue(t->right);
190                 t->isValueGetted = true;
191                 return t->value;
192             }
193             else if (t->op == "-") {
194                 t->value = getValue(t->left) - getValue(t->right);
195                 t->isValueGetted = true;
196                 return t->value;
197             }
198             else if (t->op == "*") {
199                 t->value = getValue(t->left) * getValue(t->right);
200                 t->isValueGetted = true;
201                 return t->value;
202             }
203             else if (t->op == "/") {
204                 t->value = getValue(t->left) / getValue(t->right);
205                 t->isValueGetted = true;
206                 return t->value;
207             }
208         }
209     }
210
```

ExpressionTree.h