山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：2024 级数据结构--数据智能 实验 9 二叉树		
实验学时：2	实验日期：2024/11/13	

实验目的：

1. 掌握二叉树结构的定义与实现；
2. 掌握二叉树结构的使用。

软件开发工具：

1. visual studio code 2022（使用 C/C++、C/C++ Extension Pack、C/C++ Themes 插件）
2. mingw64 工具包

1. 实验内容

完成 2024 级数据结构--数据智能 实验 9 二叉树 的 A 二叉树基础 和 B 二叉树遍历

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

本题使用二叉树数据结构，这个题 AB 题使用的算法是一致的，主要区别在于构建二叉树的方式不同。我们先声明节点结构体，存入数据、左节点指针、右节点指针、父节点指针。第一题的使用的构建信息是编号为 n 的节点的子节点的编号（数据），因为这个题的输入比较简单，输入的数据就对应其编号，因此我们可以先建立一个为节点个数大小的节点指针的数组，然后每构建一个节点，就根据这个节点的编号把节点的指针插入到这个数组里，然后我们只要遍历这个数组，就可以把每一个节点的子节点都链接好并把整个树构建好，此时我们这个树是一个强连通图，我们可以任意遍历其中的节点而不需要额外的指针来进行回溯。B 题中的输入是一个前序遍历的输出，一个中序遍历的输出，这个信息也是足够让我们构建这个树的。首先我们把这个树视为一个个子树组成的集合，其中子树的最小单位就是叶节点，最大的子树就是这个树自己。然后我们对描述每一个子树的前序遍历和中序遍历进行处理。首先根据前序遍历的遍历方式，前序遍历的第一个元素就是这个子树的根节点，然后，我们在中序遍历里搜索这个节点（这里有一个要求就是元素之间互异，根据题目描述我们可以知道这一点）当我们搜索到这个节点之后，就可以把这个中序遍历分成三部分，从左到右分别是：左子树节点的中序遍历序列、根节点、右子树节点的中序遍历序列。我们可以根据这个数据获得左子树和右子树的元素个数，再根据这个个数可以将前序遍历再分为从左到右三部分：根节点、左子树的前序遍历序列、右子树的前序遍历序列。至此，我们把这个子树分成了三部分，我们可以根据以上分出来的序列再分别对左子树和右子树进行构建，通过递归的方式直到一个子树既没有左子树也没有右子树，至此可以找到叶节点。我们每一个根节点的左右子节点指针都分别指向它的左右子树的根节点。至此，和 A 题一样，我们获得了一个强连通图，这是和 A 题相同结构的数据，因此可以复用 A 题的算法。我们再分别说明计算前序遍历序列、中序遍历序列、后序遍历序列、层次遍历序列、以第 N 个节点为根节点构成的子树的节点个数、以第 N 个节点为根节点构成的子树的深度的算法。首先是前序遍历，题目要求我们使用递归的方式来实现，我们把函数定义为每次访问一个节点，并把这个节点的值输出，然后再这个函数依次去访问这个节点的左节点、右节点，这样我们就可以按照中左右的方式以 O（n）的复杂度遍历整棵树。然后是中序遍历，中序遍历和后续遍历的算法其实是差不多的，因为中序遍历需要保持左中右的顺序，且题目让我们用循环来实现，因此我们先来描述单次循环进行的操作：

对于一个节点，

①如果它有左子节点且这个左子节点没有被输出过，我们进入下一次循环处理这个节点的左

子节点
②如果它没有左子节点，或它的左子节点已经被输出过，且这个节点没有被输出过，则输出这个节点。
③如果它没有左子节点，或它的左子节点已经被输出过，且这个节点被输出过，且它有右子节点且这个右子节点没有输出过，则进入下一次循环处理这个节点的右子节点。
④如果他没有左子节点，或它的左子节点已经被输出过，且这个节点被输出过，且它有右子节点，且这个右子节点已经被输出过了或者它没有右子节点，且这个节点有父节点，则进入下一次循环的时候处理这个节点的父节点。
⑤如果他没有左子节点，或它的左子节点已经被输出过，且这个节点被输出过，且它有右子节点，且这个右子节点已经被输出过了或者它没有右子节点，且这个节点没有父节点，则结束遍历过程。
根据以上原则来进行循环，我们就可以成功以左中右的顺序完成这个树的中序遍历。
然后是后序遍历的算法，题目同样要求我们使用循环来解决这个问题，所以我们先来描述单词循环进行的操作：
对于一个节点，
①如果它有左子节点且这个左子节点没有被输出过，我们进入下一次循环处理这个节点的左子节点
②如果它没有左子节点，或者它的左子节点已经被输出过，且这个节点有右子节点，且它的右子节点没有被输出过，我们进入下一次循环处理这个节点的右子节点
③如果它没有左子节点，或者它的左子节点已经被输出过，且这个节点有右子节点且右子节点被输出过，或者它没有右子节点，且这个节点没有被输出过，则输出这个节点。
④如果它没有左子节点，或者它的左子节点已经被输出过，且这个节点有右子节点且右子节点被输出过，或者它没有右子节点，且这个节点已经被输出过了，且它有父节点，则在下一次循环的时候处理这个节点的父节点。
⑤如果它没有左子节点，或者它的左子节点已经被输出过，且这个节点有右子节点且右子节点被输出过，或者它没有右子节点，且这个节点已经被输出过了，且它没有父节点，则结束遍历过程
根据以上原则来进行循环，我们就可以成功以左右中的顺序完成这个树的后序遍历。
然后是这个树的层次遍历，我们使用队列数据结构来进行层次遍历。我们从把根节点压入队列开始处理这个队列中的每一个节点。只需遵循以下原则，即可完成层次遍历：
①取出队列中的第一个节点，输出这个节点
②对于刚才取出的节点，如果它有左子节点，则把它的左子节点压入队列，如果它有右子节点，则把它的右子节点压入队列。
然后是计算以每个节点为根构建子树，这个子树的深度。
首先，我们可以知道，以一个节点为根构建的子树的深度等于以它的左子节点为根构建的子树的深度和以它的右子节点为根构建的子树的深度的取大值加 1
以此为基础，我们从叶节点开始计算，每一个叶节点都把深度设置为 1，然后再分别去计算这些叶节点的父节点的深度，并直到计算到根节点即可。
最后是计算以每一个节点为根构建的子树的节点个数，
首先我们可以知道，以一个节点为根构建的子树的节点个数等于以它的左子节点为根构建的子树的节点个数和以它的右子节点为根构建的子树的节点个数的和加 1
以此为基础，我们从叶节点开始计算，每一个叶节点的子树节点个数都设置为 1，然后分别去计算他们的父节点，并直到计算到根节点即可。
至此我们完成了这个题要求的所有算法的设计。
3. 测试结果（测试输入，测试输出）
A 题测试输入 1：

5
2 3
4 5
-1 -1
-1 -1
-1 -1
输出：
1 2 4 5 3
4 2 5 1 3
4 5 2 3 1
1 2 3 4 5
5 3 1 1 1
3 2 1 1 1
测试输入 2：
5
3 2
-1 -1
4 5
-1 -1
-1 -1
输出：
1 3 4 5 2
4 3 5 1 2
4 5 3 2 1
1 3 2 4 5
5 1 3 1 1
3 1 2 1 1
测试输入 3：
10
2 -1
4 3
6 -1
5 8
9 7
-1 -1
-1 -1
-1 -1
10 -1
-1 -1
输出：
1 2 4 5 9 10 7 8 3 6
10 9 5 7 4 8 2 6 3 1
10 9 7 5 8 4 6 3 2 1
1 2 4 3 5 8 6 9 7 10
10 9 2 6 4 1 1 1 2 1
6 5 2 4 3 1 1 1 2 1

B 题测试输入：
5
1 2 4 5 3
4 2 5 1 3
输出：
4 5 2 3 1

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

从测试结果来看，我们的算法成功解决了这个问题。存在的问题主要是，因为这两个题的输入都比较特殊，我们实际的建树过程其实并不是较为通用的过程，实际上我也不是很清楚什么样的建树过程是比较通用常规的，但感觉这个两个题给的建树方式都比较反直觉，实际上我还实现了另一种建树方法，即按照层次遍历的顺序依次输入每个节点的左右子节点来构建树的方式，感觉这种方式比 A 题给的建树方式更符合直觉。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

A 题代码：

```cpp
/*2024 级数据结构--数据智能 实验 9 二叉树 A 二叉树基础.cpp*/
#include <iostream>
#define debug cout << __LINE__ << " " << __FUNCTION__ << " " << endl;
using namespace std;


template<class T>
class queue
{
private:
    T* data;
    int front; //当前的头
    int tile;  //当前的最后一个的下一个
    int capacity;
public:
    queue(int capacity) {
        this->capacity = capacity;
        data = new T[capacity];
        front = 0;
        tile = 0;
    }
    ~queue() {
        delete[] data;
    }
    void push(T&& a_data) {
        if (tile == capacity) {
            T* newData = new T[capacity * 2];

            for (int i = front; i < tile; i++) {
                newData[i - front] = this->data[i];
            }
```

```
32.                 delete[] data;
33.                 tile -= front;
34.                 data = newData;
35.                 front = 0;
36.                 capacity *= 2;
37.             }
38.             data[tile++] = a_data;
39.         }
40.         void push(T& a_data)
41.         {
42.             if (tile == capacity) {
43.             T* newData = new T[capacity * 2];
44.
45.             for (int i = front; i < tile; i++) {
46.                 newData[i - front] = this->data[i];
47.             }
48.             delete[] data;
49.             tile -= front;
50.             data = newData;
51.             front = 0;
52.             capacity *= 2;
53.         }
54.             data[tile++] = a_data;
55.         }
56.         T pop() {
57.             return data[front++];
58.         }
59.
60.         int size() {
61.             return tile - front;
62.         }
63.
64.         bool empty() {
65.             return tile == front;
66.         }
67.     };
68.
69.     size_t pow(size_t base, size_t exp)
70.     {
71.         size_t result = 1;
72.         while (exp > 0)
73.         {
74.             if (exp & 1)
75.             {
76.                 result *= base;
77.             }
```

```
78.             base *= base;
79.             exp >>= 1;
80.         }
81.         return result;
82.     }
83.
84.     template<class T>
85.     inline T& max(T& a, T& b)
86.     {
87.         return a > b? a : b;
88.     }
89.     template<class T>
90.     inline T& min(T& a, T& b)
91.     {
92.         return a < b? a : b;
93.     }
94.
95.
96.     template<class T>
97.     struct Node
98.     {
99.         T data;
100.        Node<T>* left, * right, * parent;
101.        size_t depth;
102.        size_t sons;
103.        bool touched;
104.        Node(Node<T>* parent, const T& data) : data(data), left(nullptr
    ), right(nullptr), parent(parent), depth(0), sons(0), touched(false)
    {}
105.    };
106.
107.    template<class T>
108.    class binaryTree
109.    {
110.    public:
111.        enum inputType { preOrder, inOrder, postOrder };
112.    private:
113.        Node<T>* root;
114.        size_t count;
115.        size_t depth;
116.        Node<T>** index;
117.        Node<T>** index4level;
118.    public:
119.        Node<T>* Root() { return root; };
120.        binaryTree();
121.        binaryTree(size_t size, inputType type = inputType::preOrder);
```

```
122.      ~binaryTree();
123.      size_t size() const;
124.      size_t height() const;
125.      size_t subTreeSize(const size_t subscript) const;
126.      size_t subTreeHeight(const size_t subscript) const;
127.      size_t subTreeSize(Node<T>* node) const;
128.      size_t subTreeHeight(Node<T>* node) const;
129.      void preOrderOutput(Node<T>* node);
130.      void inOrderOutput();
131.      void postOrderOutput();
132.      void levelOrderOutput();
133.      void subTreeSizeOutput();
134.      void subTreeHeightOutput();
135.      void iniDepth(Node<T>* node);
136.      void iniSons(Node<T>* node);
137.  };
138.  template<class T>
139.  binaryTree<T>::binaryTree()
140.  {
141.      root = nullptr;
142.      count = 0;
143.      depth = 0;
144.      index = nullptr;
145.  }
146.
147.
148.  template<class T>
149.  binaryTree<T>::binaryTree(size_t size, binaryTree<T>::inputType typ
     e)
150.  {
151.      if (size == 0)
152.      {
153.          return;
154.      }
155.      Node<T>* node = new Node<T>(nullptr, 1);
156.      root = node;
157.      count = 1;
158.      depth = 1;
159.      index = new Node<T>*[size];
160.      index4level = new Node<T>*[size];
161.      node->depth = 1;
162.      index[0] = node;
163.      index4level[0] = node;
164.      if (size == 1)
165.      {
```

```
166.             return;
167.         }
168.         if (type == binaryTree<T>::inputType::preOrder)
169.         {
170.             return;
171.         }
172.         else if (type == binaryTree<T>::inputType::inOrder)
173.         {
174.             // size_t nullptr_count = 0;
175.             for (size_t i = 0; i < size; i++)
176.             {
177.                 Node<T>* operatedNode = index[i];
178.                 // cout << operatedNode->data << " ";
179.                 T left, right;
180.                 cin >> left >> right;
181.                 if (left != -1)
182.                 {
183.                     Node<T>* leftNode = new Node<T>(operatedNode, left)
    ;
184.                     operatedNode->left = leftNode;
185.                     leftNode->depth = operatedNode->depth + 1;
186.                     index[left - 1] = leftNode;
187.                     index4level[count] = leftNode;
188.                     count++;
189.                 }
190.                 else
191.                 {
192.                     operatedNode->left = nullptr;
193.                     // nullptr_count++;
194.                 }
195.                 if (right != -1)
196.                 {
197.                     Node<T>* rightNode = new Node<T>(operatedNode, righ
    t);
198.                     operatedNode->right = rightNode;
199.                     rightNode->depth = operatedNode->depth + 1;
200.                     index[right - 1] = rightNode;
201.                     index4level[count] = rightNode;
202.                     count++;
203.                 }
204.                 else
205.                 {
206.                     operatedNode->right = nullptr;
207.                     // nullptr_count++;
208.                 }
209.                 // debug
```

```
210.                 if (i == size - 1)
211.                 {
212.                     this->depth = operatedNode->depth;
213.                 }
214.             }
215.             return;
216.         }
217.         else if (type == binaryTree<T>::inputType::postOrder)
218.         {
219.             return;
220.         }
221.     }
222.
223.     template<class T>
224.     binaryTree<T>::~binaryTree()
225.     {
226.         for (size_t i = 0; i < count; i++)
227.         {
228.             delete index[i];
229.         }
230.         delete[] index;
231.     }
232.
233.     template<class T>
234.     size_t binaryTree<T>::size() const
235.     {
236.         return count;
237.     }
238.
239.     template<class T>
240.     size_t binaryTree<T>::height() const
241.     {
242.         return depth;
243.     }
244.
245.
246.     template<class T>
247.     size_t binaryTree<T>::subTreeSize(size_t subscript) const
248.     {
249.         return index[subscript]->sons;
250.     }
251.
252.     template<class T>
253.     size_t binaryTree<T>::subTreeSize(Node<T>* node) const
254.     {
255.         if (node == nullptr)
```

```
256.     {
257.         return 0;
258.     }
259.     return node->sons;
260. }
261.
262. template<class T>
263. size_t binaryTree<T>::subTreeHeight(size_t subscript) const
264. {
265.     return index[subscript]->depth;
266. }
267.
268. template<class T>
269. size_t binaryTree<T>::subTreeHeight(Node<T>* node) const
270. {
271.     if (node == nullptr)
272.     {
273.         return 0;
274.     }
275.     return node->depth;
276. }
277.
278. template<class T>
279. void binaryTree<T>::preOrderOutput(Node<T>* node)
280. {
281.     if (node == nullptr)
282.     {
283.         return;
284.     }
285.     cout << node->data << " ";
286.     preOrderOutput(node->left);
287.     preOrderOutput(node->right);
288.     return;
289. }
290.
291. template<class T>
292. void binaryTree<T>::inOrderOutput()
293. {
294.     Node<T>* node = root;
295.     for (size_t i = 0; i < this->size(); i++)
296.     {
297.         index[i]->touched = false;
298.     }
299.     while (1)
300.     {
301.         if (node->left != nullptr && node->left->touched == false)
```

```
302.         {
303.             node = node->left;
304.         }
305.         else
306.         {
307.             if (node->touched == false)
308.             {
309.                 cout << node->data << " ";
310.                 node->touched = true;
311.                 if (node->right != nullptr)
312.                 {
313.                     node = node->right;
314.                 }
315.                 else
316.                 {
317.                     node = node->parent;
318.                 }
319.             }
320.             else
321.             {
322.                 if (node->parent != nullptr)
323.                 {
324.                     node = node->parent;
325.                 }
326.                 else if (node->right != nullptr && node->right->tou
    ched == true)
327.                 {
328.                     break;
329.                 }
330.             }
331.
332.         }
333.         if (node == nullptr)
334.         {
335.             break;
336.         }
337.         if (node->parent == nullptr && node->right != nullptr && no
    de->right->touched == true)
338.         {
339.             break;
340.         }
341.     }
342.
343.     cout << endl;
344.     return;
```

```
345.  }
346.
347.  template<class T>
348.  void binaryTree<T>::postOrderOutput()
349.  {
350.      Node<T>* node = root;
351.      for (size_t i = 0; i < this->size(); i++)
352.      {
353.          index[i]->touched = false;
354.      }
355.      while (1)
356.      {
357.          if (node == nullptr)
358.          {
359.              break;
360.          }
361.          if (node->left != nullptr && node->left->touched == false)

362.          {
363.              node = node->left;
364.          }
365.          else
366.          {
367.              if (node->right != nullptr && node->right->touched == f
    alse)
368.              {
369.                  node = node->right;
370.              }
371.              else
372.              {
373.                  if (node->touched == false)
374.                  {
375.                      cout << node->data << " ";
376.                      node->touched = true;
377.                      node = node->parent;
378.                  }
379.                  else
380.                  {
381.                      if (node->parent != nullptr)
382.                      {
383.                          node = node->parent;
384.                      }
385.                      else if (node->right != nullptr && node->right-
    >touched == true && node->touched == true)
386.                      {
387.                          break;
```

```
                    }
                }
            }
        }
        if (node == nullptr)
        {
            break;
        }
        if (node->parent == nullptr && node->right != nullptr && node->right->touched == true && node->touched == true)
        {
            break;
        }
    }
    cout << endl;
    return;
}

template<class T>
void binaryTree<T>::levelOrderOutput()
{
    // for (size_t i = 0; i < this->size(); i++)
    // {
    //     cout << index4level[i]->data << " ";
    // }
    // cout << endl;
    Node<T>* node = root;
    queue<Node<T>*> q(this->size());
    q.push(node);
    while (q.size() > 0)
    {
        node = q.pop();
        if (node->left != nullptr)
        {
            q.push(node->left);
        }
        if (node->right != nullptr)
        {
            q.push(node->right);
        }
        cout << node->data << " ";
    }
    cout << endl;
}

```

```
433. template<class T>
434. void binaryTree<T>::subTreeSizeOutput()
435. {
436.     for (size_t i = 0; i < this->size(); i++)
437.     {
438.         cout << subTreeSize(i) << " ";
439.     }
440.     cout << endl;
441. }
442.
443. template<class T>
444. void binaryTree<T>::subTreeHeightOutput()
445. {
446.     for (size_t i = 0; i < this->size(); i++)
447.     {
448.         cout << subTreeHeight(i) << " ";
449.     }
450.     cout << endl;
451. }
452.
453. template<class T>
454. void binaryTree<T>::iniDepth(Node<T>* node)
455. {
456.     if (node == nullptr)
457.     {
458.         return;
459.     }
460.     if (node->left == nullptr && node->right == nullptr)
461.     {
462.         node->depth = 1;
463.         return;
464.     }
465.     iniDepth(node->left);
466.     iniDepth(node->right);
467.     if (node->left != nullptr && node->right != nullptr)
468.     {
469.         node->depth = max(node->left->depth, node->right->depth) +
    1;
470.     }
471.     else if (node->left != nullptr && node->right == nullptr)
472.     {
473.         node->depth = node->left->depth + 1;
474.     }
475.     else if (node->left == nullptr && node->right != nullptr)
476.     {
477.         node->depth = node->right->depth + 1;
```

```
478.     }
479.     return;
480. }
481. 
482. template<class T>
483. void binaryTree<T>::iniSons(Node<T>* node)
484. {
485.     if (node == nullptr)
486.     {
487.         return;
488.     }
489.     if (node->left == nullptr && node->right == nullptr)
490.     {
491.         node->sons = 1;
492.         return;
493.     }
494.     iniSons(node->left);
495.     iniSons(node->right);
496.     if (node->left != nullptr && node->right != nullptr)
497.     {
498.         node->sons = node->left->sons + node->right->sons + 1;
499.     }
500.     else if (node->left != nullptr && node->right == nullptr)
501.     {
502.         node->sons = node->left->sons + 1;
503.     }
504.     else if (node->left == nullptr && node->right != nullptr)
505.     {
506.         node->sons = node->right->sons + 1;
507.     }
508.     return;
509. }
510. class Solution
511. {
512. public:
513.     void solve();
514. };
515. 
516. 
517. void Solution::solve()
518. {
519.     size_t size;
520.     cin >> size;
521.     binaryTree<int> tree(size, binaryTree<int>::inputType::inOrder)
    ;
522.     tree.preOrderOutput(tree.Root());
```

```
    cout << endl;
    tree.inOrderOutput();
    tree.postOrderOutput();
    tree.levelOrderOutput();
    tree.iniDepth(tree.Root());
    tree.iniSons(tree.Root());
    tree.subTreeSizeOutput();
    tree.subTreeHeightOutput();
}


int main()
{
    Solution solution;

    solution.solve();

    return 0;
}
```

B 题代码：

```
/*2024 级数据结构--数据智能 实验 9 二叉树 B 二叉树遍历.cpp*/
#include <iostream>

using namespace std;


template<class T>
class queue
{
private:
    T* data;
    int front; //当前的头
    int tile;  //当前的最后一个的下一个
    int capacity;
public:
    queue(int capacity) {
        this->capacity = capacity;
        data = new T[capacity];
        front = 0;
        tile = 0;
    }
    ~queue() {
        delete[] data;
    }
    void push(T&& a_data) {
        if (tile == capacity) {
```

```
27.                 T* newData = new T[capacity * 2];
28.
29.                 for (int i = front; i < tile; i++) {
30.                     newData[i - front] = this->data[i];
31.                 }
32.                 delete[] data;
33.                 tile -= front;
34.                 data = newData;
35.                 front = 0;
36.                 capacity *= 2;
37.             }
38.             data[tile++] = a_data;
39.         }
40.         void push(T& a_data)
41.         {
42.             if (tile == capacity) {
43.             T* newData = new T[capacity * 2];
44.
45.             for (int i = front; i < tile; i++) {
46.                 newData[i - front] = this->data[i];
47.             }
48.             delete[] data;
49.             tile -= front;
50.             data = newData;
51.             front = 0;
52.             capacity *= 2;
53.         }
54.             data[tile++] = a_data;
55.         }
56.         T pop() {
57.             return data[front++];
58.         }
59.
60.         int size() {
61.             return tile - front;
62.         }
63.
64.         bool empty() {
65.             return tile == front;
66.         }
67.     };
68.
69.     template<class T>
70.     struct Node
71.     {
72.         T data;
```

```
73.       Node<T>* left, * right, * parent;
74.       size_t depth;
75.       size_t sons;
76.       bool touched;
77.       Node(Node<T>* parent, const T& data) : data(data), left(nullptr
   ), right(nullptr), parent(parent), depth(0), sons(0), touched(false)
   {}
78.       Node(): left(nullptr), right(nullptr), parent(nullptr), depth(0
   ), sons(0), touched(false) {}
79.   };
80.
81.   template<class T>
82.   class binaryTree
83.   {
84.   public:
85.       enum inputType { preOrder, inOrder, postOrder, minxOrder };
86.   private:
87.       Node<T>* root;
88.       size_t count;
89.       size_t depth;
90.       Node<T>** index;
91.       Node<T>** index4level;
92.   public:
93.       Node<T>* Root() { return root; };
94.       binaryTree();
95.       binaryTree(size_t size, inputType type = inputType::preOrder);

96.       ~binaryTree();
97.       size_t size() const;
98.       size_t height() const;
99.       size_t subTreeSize(const size_t subscript) const;
100.      size_t subTreeHeight(const size_t subscript) const;
101.      size_t subTreeSize(Node<T>* node) const;
102.      size_t subTreeHeight(Node<T>* node) const;
103.      void preOrderOutput(Node<T>* node);
104.      void inOrderOutput();
105.      void postOrderOutput();
106.      void levelOrderOutput();
107.      void subTreeSizeOutput();
108.      void subTreeHeightOutput();
109.      void iniDepth(Node<T>* node);
110.      void iniSons(Node<T>* node);
111.      void buildTree(Node<T>* root,T preOrder[], T inOrder[], size_t
   size,bool direction = true);
112.  };
113.  template<class T>
```

```
114. binaryTree<T>::binaryTree()
115. {
116.     root = nullptr;
117.     count = 0;
118.     depth = 0;
119.     index = nullptr;
120. }
121.
122.
123. template<class T>
124. binaryTree<T>::binaryTree(size_t size, binaryTree<T>::inputType typ
   e)
125. {
126.     if (size == 0)
127.     {
128.         return;
129.     }
130.     Node<T>* node = new Node<T>(nullptr, 1);
131.     root = node;
132.     count = 1;
133.     depth = 1;
134.     index = new Node<T>*[size];
135.     index4level = new Node<T>*[size];
136.     node->depth = 1;
137.     index[0] = node;
138.     index4level[0] = node;
139.     if (size == 1)
140.     {
141.         return;
142.     }
143.     if (type == binaryTree<T>::inputType::preOrder)
144.     {
145.         return;
146.     }
147.     else if (type == binaryTree<T>::inputType::inOrder)
148.     {
149.         // size_t nullptr_count = 0;
150.         for (size_t i = 0; i < size; i++)
151.         {
152.             Node<T>* operatedNode = index[i];
153.             // cout << operatedNode->data << " ";
154.             T left, right;
155.             cin >> left >> right;
156.             if (left != -1)
157.             {
158.                 Node<T>* leftNode = new Node<T>(operatedNode, left)
```

```
    ;
                operatedNode->left = leftNode;
                leftNode->depth = operatedNode->depth + 1;
                index[left - 1] = leftNode;
                index4level[count] = leftNode;
                count++;
            }
            else
            {
                operatedNode->left = nullptr;
                // nullptr_count++;
            }
            if (right != -1)
            {
                Node<T>* rightNode = new Node<T>(operatedNode, righ
    t);
                operatedNode->right = rightNode;
                rightNode->depth = operatedNode->depth + 1;
                index[right - 1] = rightNode;
                index4level[count] = rightNode;
                count++;
            }
            else
            {
                operatedNode->right = nullptr;
                // nullptr_count++;
            }
            // debug
            if (i == size - 1)
            {
                this->depth = operatedNode->depth;
            }
        }
        return;
    }
    else if (type == binaryTree<T>::inputType::postOrder)
    {
        return;
    }
    else if (type == binaryTree<T>::inputType::minxOrder)
    {
        T* preOrder = new T[size];
        T* inOrder = new T[size];
        for (size_t i = 0; i < size; i++)
        {
            cin >> preOrder[i];
```

```
203.         }
204.         for (size_t i = 0; i < size; i++)
205.         {
206.             cin >> inOrder[i];
207.         }
208.
209.         this -> root = new Node<T>(nullptr, preOrder[0]);
210.
211.
212.         buildTree(this->root, preOrder, inOrder, size);
213.
214.         return;
215.     }
216.
217. }
218.
219. template<class T>
220. void binaryTree<T>::buildTree(Node<T>* root, T preOrder[], T inOrde
   r[], size_t size , bool direction)
221. {
222.     if (root == nullptr)
223.     {
224.         return;
225.     }
226.     if (size == 0)
227.     {
228.         return;
229.     }
230.     if (size == 1)
231.     {
232.         // if (direction)
233.         // {
234.         //     root->left = new Node<T>(root, preOrder[0]);

235.         // }
236.         // if (!direction)
237.         // {
238.         //     root->right = new Node<T>(root, preOrder[0]);
239.         // }
240.         return;
241.     }
242.     size_t rootIndex = 0;
243.     for (size_t i = 0; i < size; i++)
244.     {
245.         if (inOrder[i] == preOrder[0])
246.         {
```

```
247.                rootIndex = i;
248.                break;
249.            }
250.        }
251.
252.        size_t leftSize = rootIndex;
253.        size_t rightSize = size - leftSize - 1;
254.
255.        T* leftPreOrderHead = &preOrder[1];
256.
257.        T* rightPreOrderHead = &preOrder[rootIndex + 1];
258.
259.        T* leftInOrderHead = &inOrder[0];
260.
261.        T* rightInOrderHead = &inOrder[rootIndex + 1];
262.
263.
264.        if (leftSize > 0)
265.        {
266.            root->left = new Node<T>(root, preOrder[1]);
267.            buildTree(root->left, leftPreOrderHead, leftInOrderHead, le
    ftSize, true);
268.        }
269.        if (rightSize > 0)
270.        {
271.            root->right = new Node<T>(root, preOrder[rootIndex + 1]);
272.            buildTree(root->right, rightPreOrderHead, rightInOrderHead,
     rightSize, false);
273.        }
274.    }
275.
276.
277.
278.
279.    template<class T>
280.    binaryTree<T>::~binaryTree()
281.    {
282.        for (size_t i = 0; i < count; i++)
283.        {
284.            delete index[i];
285.        }
286.        delete[] index;
287.    }
288.
289.    template<class T>
290.    size_t binaryTree<T>::size() const
```

```
291. {
292.     return count;
293. }
294.
295. template<class T>
296. size_t binaryTree<T>::height() const
297. {
298.     return depth;
299. }
300.
301.
302. template<class T>
303. size_t binaryTree<T>::subTreeSize(size_t subscript) const
304. {
305.     return index[subscript]->sons;
306. }
307.
308. template<class T>
309. size_t binaryTree<T>::subTreeSize(Node<T>* node) const
310. {
311.     if (node == nullptr)
312.     {
313.         return 0;
314.     }
315.     return node->sons;
316. }
317.
318. template<class T>
319. size_t binaryTree<T>::subTreeHeight(size_t subscript) const
320. {
321.     return index[subscript]->depth;
322. }
323.
324. template<class T>
325. size_t binaryTree<T>::subTreeHeight(Node<T>* node) const
326. {
327.     if (node == nullptr)
328.     {
329.         return 0;
330.     }
331.     return node->depth;
332. }
333.
334. template<class T>
335. void binaryTree<T>::preOrderOutput(Node<T>* node)
336. {
```

```
337.     if (node == nullptr)
338.     {
339.         return;
340.     }
341.     cout << node->data << " ";
342.     preOrderOutput(node->left);
343.     preOrderOutput(node->right);
344.     return;
345. }
346. 
347. template<class T>
348. void binaryTree<T>::inOrderOutput()
349. {
350.     Node<T>* node = root;
351.     for (size_t i = 0; i < this->size(); i++)
352.     {
353.         index[i]->touched = false;
354.     }
355.     while (1)
356.     {
357.         if (node->left != nullptr && node->left->touched == false)

358.         {
359.             node = node->left;
360.         }
361.         else
362.         {
363.             if (node->touched == false)
364.             {
365.                 cout << node->data << " ";
366.                 node->touched = true;
367.                 if (node->right != nullptr)
368.                 {
369.                     node = node->right;
370.                 }
371.                 else
372.                 {
373.                     node = node->parent;
374.                 }
375.             }
376.             else
377.             {
378.                 if (node->parent != nullptr)
379.                 {
380.                     node = node->parent;
381.                 }
```

```
382.                 else if (node->right != nullptr && node->right->tou
    ched == true)
383.                 {
384.                     break;
385.                 }
386.             }
387.
388.         }
389.         if (node == nullptr)
390.         {
391.             break;
392.         }
393.         if (node->parent == nullptr && node->right != nullptr && no
    de->right->touched == true)
394.         {
395.             break;
396.         }
397.     }
398.
399.     cout << endl;
400.     return;
401. }
402.
403. template<class T>
404. void binaryTree<T>::postOrderOutput()
405. {
406.     Node<T>* node = root;
407.     // for (size_t i = 0; i < this->size(); i++)
408.     // {
409.     //     index[i]->touched = false;
410.     // }
411.     while (1)
412.     {
413.         if (node == nullptr)
414.         {
415.             break;
416.         }
417.         if (node->left != nullptr && node->left->touched == false)

418.         {
419.             node = node->left;
420.         }
421.         else
422.         {
423.             if (node->right != nullptr && node->right->touched == f
    alse)
```

```
424.                 {
425.                     node = node->right;
426.                 }
427.                 else
428.                 {
429.                     if (node->touched == false)
430.                     {
431.                         cout << node->data << " ";
432.                         node->touched = true;
433.                         node = node->parent;
434.                     }
435.                     else
436.                     {
437.                         if (node->parent != nullptr)
438.                         {
439.                             node = node->parent;
440.                         }
441.                         else if (node->right != nullptr && node->right->touched == true && node->touched == true)
442.                         {
443.                             break;
444.                         }
445.                     }
446.                 }
447.             }
448.             if (node == nullptr)
449.             {
450.                 break;
451.             }
452.             if (node->parent == nullptr && node->right != nullptr && node->right->touched == true && node->touched == true)
453.             {
454.                 break;
455.             }
456.         }
457.         cout << endl;
458.         return;
459.     }
460.
461.     template<class T>
462.     void binaryTree<T>::levelOrderOutput()
463.     {
464.         // for (size_t i = 0; i < this->size(); i++)
465.         // {
466.         //     cout << index4level[i]->data << " ";
467.         // }
```

```
468.     // cout << endl;
469.     Node<T>* node = root;
470.     queue<Node<T>*> q(this->size());
471.     q.push(node);
472.     while (q.size() > 0)
473.     {
474.         node = q.pop();
475.         if (node->left != nullptr)
476.         {
477.             q.push(node->left);
478.         }
479.         if (node->right != nullptr)
480.         {
481.             q.push(node->right);
482.         }
483.         cout << node->data << " ";
484.     }
485.     cout << endl;
486. }
487.
488.
489. template<class T>
490. void binaryTree<T>::subTreeSizeOutput()
491. {
492.     for (size_t i = 0; i < this->size(); i++)
493.     {
494.         cout << subTreeSize(i) << " ";
495.     }
496.     cout << endl;
497. }
498.
499. template<class T>
500. void binaryTree<T>::subTreeHeightOutput()
501. {
502.     for (size_t i = 0; i < this->size(); i++)
503.     {
504.         cout << subTreeHeight(i) << " ";
505.     }
506.     cout << endl;
507. }
508.
509. template<class T>
510. void binaryTree<T>::iniDepth(Node<T>* node)
511. {
512.     if (node == nullptr)
513.     {
```

```
514.             return;
515.         }
516.         if (node->left == nullptr && node->right == nullptr)
517.         {
518.             node->depth = 1;
519.             return;
520.         }
521.         iniDepth(node->left);
522.         iniDepth(node->right);
523.         if (node->left != nullptr && node->right != nullptr)
524.         {
525.             node->depth = max(node->left->depth, node->right->depth) +
    1;
526.         }
527.         else if (node->left != nullptr && node->right == nullptr)
528.         {
529.             node->depth = node->left->depth + 1;
530.         }
531.         else if (node->left == nullptr && node->right != nullptr)
532.         {
533.             node->depth = node->right->depth + 1;
534.         }
535.         return;
536.     }
537.
538.     template<class T>
539.     void binaryTree<T>::iniSons(Node<T>* node)
540.     {
541.         if (node == nullptr)
542.         {
543.             return;
544.         }
545.         if (node->left == nullptr && node->right == nullptr)
546.         {
547.             node->sons = 1;
548.             return;
549.         }
550.         iniSons(node->left);
551.         iniSons(node->right);
552.         if (node->left != nullptr && node->right != nullptr)
553.         {
554.             node->sons = node->left->sons + node->right->sons + 1;
555.         }
556.         else if (node->left != nullptr && node->right == nullptr)
557.         {
558.             node->sons = node->left->sons + 1;
```

```
559.      }
560.      else if (node->left == nullptr && node->right != nullptr)
561.      {
562.          node->sons = node->right->sons + 1;
563.      }
564.      return;
565.  }
566.  class Solution
567.  {
568.  public:
569.      void solve();
570.  };
571.
572.
573.  void Solution::solve()
574.  {
575.      size_t size;
576.      cin >> size;
577.
578.      binaryTree<int> tree(size, binaryTree<int>::inputType::minxOrde
      r);
579.      // tree.preOrderOutput(tree.Root());
580.      // cout << endl;
581.      tree.postOrderOutput();
582.  }
583.
584.
585.  int main()
586.  {
587.      Solution solution;
588.
589.      solution.solve();
590.
591.      return 0;
592.  }
```