**《数据结构》实验报告（第五章）**

**学号：\_\_\_\_\_\_58122231\_\_\_\_ 姓名 ：\_\_\_\_\_陆文韬\_\_\_\_\_\_**

**实验题号： P267 10 P272-273 1 2\_ 实验日期：\_2023.12.12\_\_\_\_**

**实验一**

**1．问题描述：**描述实验内容和要求以及需要解决的问题。

为二叉树实现一个完整的C + +模板类。要求包括构造函数、复制构造两数和析构两数,实现本节讨论的4种遍历方法以及实现这4个方法的前向迭代器。另外，还需包括A D T 5 . 1中的其余方法 。

**2．算法思想：**详细描述解决相应问题所需要的算法设计思想。

该程序首先需要建立一个二叉树的类，这个类包含一个节点类，在实例化的过程中，我们采用完全二叉树的顺序去建立一棵树，再使用迭代器的方法分别实现前序遍历、中序遍历、后序遍历和层级遍历。

**3．功能函数：**描述所设计的功能函数。如果有多个函数，需要描述它们之间的关系。

重载\*函数：使得用迭代器输出数据时，能使用\*it访问数据

重载++函数：使得迭代器能够正常进行迭代

重载!=函数：判断两个迭代器是否相等

pushLeft：将当前节点压入栈内，将指针指向当前节点的左子树

isEmpty：判断是否是空树

RootData：返回根节点的数据

LeftSubtree：返回左子树

RightSubtree：返回右子树

Insert：构建二叉树的插入节点函数

Clear：删除节点

**4．测试数据：**设计测试数据，或具体给出测试数据。

提示：要求测试数据能全面地测试所设计程序的功能。

tree.insert(1);

tree.insert(2);

tree.insert(3);

tree.insert(4);

tree.insert(5);

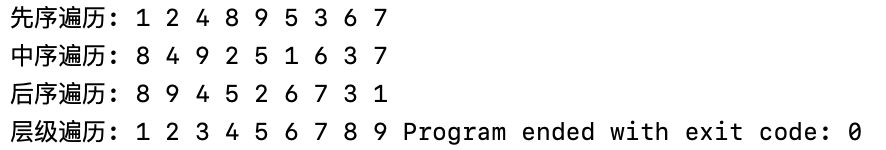
tree.insert(6);

tree.insert(7);

tree.insert(8);

tree.insert(9);

**5．测试情况：**给出程序的测试情况，分析运行结果，显示实验结果截图。

****

**6．实验总结：**写出实验过程中遇到的问题，以及问题的解决过程。分析算法的时间复杂度和空间复杂度，总结实验心得体会。

前序遍历、中序遍历、后序遍历都采用栈作为数据成员，层级遍历采用队列作为数据成员，前序和中序的思想较为简单，后序遍历需要记录下前一个输出的节点

**7. 源代码：**

#include <iostream>

#include <stack>

#include <queue>

#include <stdexcept>

**using** **namespace** std;

**template** <**typename** T>

**class** BinaryTree;

**template** <**typename** T>

**class** PreOrderIterator //先序遍历

{

**public**:

PreOrderIterator(**typename** BinaryTree<T>::Node\* root)

{

**if** (root)

{

stack.push(root);//先输出根结点

}

}

T& **operator**\*() **const**

{

**return** stack.top()->value;

}

PreOrderIterator& **operator**++()

{

**if** (stack.empty())

{

**throw** out\_of\_range("栈为空");

}

**typename** BinaryTree<T>::Node\* current = stack.top();

stack.pop();

// 先压入右孩子，然后压入左孩子，保证左孩子在栈顶

**if** (current->right)

{

stack.push(current->right);

}

**if** (current->left)

{

stack.push(current->left);

}

**return** \***this**;

}

**bool** **operator**!=(**const** PreOrderIterator& other) **const**

{

// 如果两个迭代器的栈都是空的，那么它们是相等的

**if** (stack.empty() && other.stack.empty()) **return** **false**;

// 如果一个栈是空的而另一个不是，那么它们不相等

**if** (stack.empty() || other.stack.empty()) **return** **true**;

// 如果栈顶相同，则认为迭代器相同

**return** stack.top() != other.stack.top();

}

**private**:

stack<**typename** BinaryTree<T>::Node\*> stack;

};

**template** <**typename** T>

**class** InOrderIterator//中序遍历

{

**public**:

**void** pushLeft(**typename** BinaryTree<T>::Node\* node)

{

**while** (node)

{

stack.push(node);

node = node->left;

}

}

InOrderIterator(**typename** BinaryTree<T>::Node\* root)

{

pushLeft(root);//先将左子树压入栈

}

T& **operator**\*() **const**

{

**return** stack.top()->value;

}

InOrderIterator& **operator**++()

{

**if** (stack.empty())

{

**throw** out\_of\_range("栈为空");

}

**typename** BinaryTree<T>::Node\* current = stack.top();

stack.pop();

// 根结点出栈后压入右孩子

pushLeft(current->right);

**return** \***this**;

}

**bool** **operator**!=(**const** InOrderIterator& other) **const**

{

// 如果两个迭代器的栈都是空的，那么它们是相等的

**if** (stack.empty() && other.stack.empty()) **return** **false**;

// 如果一个栈是空的而另一个不是，那么它们不相等

**if** (stack.empty() || other.stack.empty()) **return** **true**;

// 如果栈顶相同，则认为迭代器相同

**return** stack.top() != other.stack.top();

}

**private**:

stack<**typename** BinaryTree<T>::Node\*> stack;

};

**template** <**typename** T>

**class** PostOrderIterator//后序遍历

{

**public**:

**void** pushLeft(**typename** BinaryTree<T>::Node\* node)

{

**while** (node)

{

stack.push(node);

node = node->left;

}

}

PostOrderIterator(**typename** BinaryTree<T>::Node\* root)

{

pushLeft(root);//先将左子树压入栈

}

T& **operator**\*() **const**

{

**return** stack.top()->value;

}

**bool** WhetherOutput = **true**;//防止重复输出栈顶元素

PostOrderIterator& **operator**++()

{

**if** (stack.empty())

{

**throw** std::out\_of\_range("栈为空");

}

**typename** BinaryTree<T>::Node\* node = stack.top();

**if** (!node->right || prev == node->right)//如果栈顶元素没有右孩子或右孩子是上一个输出的元素就出栈

{

stack.pop();

prev = node;//记录刚刚出栈的元素，避免重复压入栈内

//若此时栈顶元素的左孩子已经输出而右孩子还没有输出，就不输出栈顶元素的值

**if** (!stack.empty() && prev == stack.top()->left && stack.top()->right)

WhetherOutput = **false**;

}

**else**//如果栈顶元素有右孩子且还没有输出过就将其压入栈内

{

pushLeft(node->right);

prev = **nullptr**;

WhetherOutput = **true**;

}

**return** \***this**;

}

**bool** **operator**!=(**const** PostOrderIterator& other) **const**

{

// 如果两个迭代器的栈都是空的，那么它们是相等的

**if** (stack.empty() && other.stack.empty()) **return** **false**;

// 如果一个栈是空的而另一个不是，那么它们不相等

**if** (stack.empty() || other.stack.empty()) **return** **true**;

// 如果栈顶相同，则认为迭代器相同

**return** stack.top() != other.stack.top();

}

**private**:

stack<**typename** BinaryTree<T>::Node\*> stack;

**typename** BinaryTree<T>::Node\* prev = **nullptr**;

};

**template** <**typename** T>

**class** LevelOrderIterator//层级遍历

{

**public**:

LevelOrderIterator(**typename** BinaryTree<T>::Node\* root)

{

**if** (root)

{

queue.push(root);//先压入根结点

}

}

T& **operator**\*() **const**

{

**return** queue.front()->value;

}

LevelOrderIterator& **operator**++()

{

**if** (queue.empty())

{

**throw** out\_of\_range("队列为空");

}

//依次压入每一层的元素

**typename** BinaryTree<T>::Node\* current = queue.front();

queue.pop();

**if** (current->left != **nullptr**)

{

queue.push(current->left);

}

**if** (current->right != **nullptr**)

{

queue.push(current->right);

}

**return** \***this**;

}

**bool** **operator**!=(**const** LevelOrderIterator& other) **const**

{

// 如果两个迭代器的队列都是空的，那么它们是相等的

**if** (queue.empty() && other.queue.empty()) **return** **false**;

// 如果一个队列是空的而另一个不是，那么它们不相等

**if** (queue.empty() || other.queue.empty()) **return** **true**;

**return** **true**;

}

**private**:

queue<**typename** BinaryTree<T>::Node\*> queue;

};

**template** <**typename** T>

**class** BinaryTree

{

**public**:

**struct** Node

{

T value;

Node\* left;

Node\* right;

Node(T val) : value(val), left(**nullptr**), right(**nullptr**) {}

};

**bool** isEmpty()

{

**if** (root) **return** **false**;

**return** **true**;

}

T RootData()//返回根结点的数据

{

**return** root->value;

}

BinaryTree(Node\* subtreeRoot)

: root(copy(subtreeRoot)) {}

BinaryTree<T>LeftSubtree()//返回左子树

{

**if** (root == **nullptr**)

{

**return** BinaryTree(); // 返回一个空的二叉树

}

**return** BinaryTree(root->left);

}

BinaryTree<T>RightSubtree()//返回右子树

{

**if** (root == **nullptr**)

{

**return** BinaryTree(); // 返回一个空的二叉树

}

**return** BinaryTree(root->right);

}

// 使用根节点、左子树和右子树创建新的二叉树

BinaryTree(T& item, BinaryTree<T>& bt1, BinaryTree<T>& bt2)

{

BinaryTree newTree;

newTree.root = **new** Node(item); // 创建新的根节点

newTree.root->left = newTree.copy(bt1.root); // 复制左子树

newTree.root->right = newTree.copy(bt2.root); // 复制右子树

**return** newTree;

}

**void** insert(**const** T& value)//为二叉树插入结点

{

Node\* newNode = **new** Node(value);

**if** (!root)

{

root = newNode;

**return**;

}

queue<Node\*> queue;

queue.push(root);

**while** (!queue.empty())

{

Node\* current = queue.front();

queue.pop();

//从左到右依次插入结点

**if** (current->left == **nullptr**)

{

current->left = newNode;

**return**;

}

**else**

{

queue.push(current->left);

}

**if** (current->right == **nullptr**)

{

current->right = newNode;

**return**;

}

**else**

{

queue.push(current->right);

}

}

}

// 构造函数

BinaryTree()

: root(**nullptr**) {}

// 进行拷贝

Node\* copy(Node\* otherRoot)

{

**if** (!otherRoot)

{

**return** **nullptr**;

}

Node\* newNode = **new** Node(otherRoot->value);

newNode->left = copy(otherRoot->left);

newNode->right = copy(otherRoot->right);

**return** newNode;

}

// 拷贝构造

BinaryTree(**const** BinaryTree& other)

: root(copy(other.root)) {}

// 删除结点

**void** clear(Node\* node)

{

**if** (node)

{

clear(node->left);

clear(node->right);

**delete** node;

}

}

// 析构函数

~BinaryTree()

{

clear(root);

}

//先序遍历的开始与结束标志

**using** iteratorPreOrder = PreOrderIterator<T>;

iteratorPreOrder beginPreOrder() { **return** iteratorPreOrder(root); }

iteratorPreOrder endPreOrder() { **return** iteratorPreOrder(**nullptr**); }

//中序遍历的开始与结束标志

**using** iteratorInOrder = InOrderIterator<T>;

iteratorInOrder beginInOrder() { **return** iteratorInOrder(root); }

iteratorInOrder endInOrder() { **return** iteratorInOrder(**nullptr**); }

//后序遍历的开始与结束标志

**using** iteratorPostOrder = PostOrderIterator<T>;

iteratorPostOrder beginPostOrder() { **return** iteratorPostOrder(root); }

iteratorPostOrder endPostOrder() { **return** iteratorPostOrder(**nullptr**); }

//层级遍历的开始与结束标志

**using** iteratorLevelOrder = LevelOrderIterator<T>;

iteratorLevelOrder beginLevelOrder() { **return** iteratorLevelOrder(root); }

iteratorLevelOrder endLevelOrder() { **return** iteratorLevelOrder(**nullptr**); }

**private**:

Node\* root;

};

**int** main()

{

// 构建二叉树

BinaryTree<**int**> tree;

tree.insert(1);

tree.insert(2);

tree.insert(3);

tree.insert(4);

tree.insert(5);

tree.insert(6);

tree.insert(7);

tree.insert(8);

tree.insert(9);

cout << "先序遍历: ";

**for** (**auto** it = tree.beginPreOrder(); it != tree.endPreOrder(); ++it)

{

cout << \*it << " ";

}

cout << endl;

cout << "中序遍历: ";

**for** (**auto** it = tree.beginInOrder(); it != tree.endInOrder(); ++it)

{

cout << \*it << " ";

}

cout << endl;

cout << "后序遍历: ";

**for** (**auto** it = tree.beginPostOrder(); it != tree.endPostOrder(); ++it)

{

**if** (it.WhetherOutput == **true**)

cout << \*it << " ";

}

cout << endl;

cout << "层级遍历: ";

**for** (**auto** it = tree.beginLevelOrder(); it != tree.endLevelOrder(); ++it)

{

cout << \*it << " ";

}

**return** 0;

}

**实验二**

**1．问题描述：**描述实验内容和要求以及需要解决的问题。

实现一个C++两数用来计算一棵二叉树中叶结点的数目。给出该函数的时间复杂度。

**2．算法思想：**详细描述解决相应问题所需要的算法设计思想。

采用层级遍历的思想，不断迭代遍历树，如果一个节点没有左孩子和右孩子即叶节点，就将个数加一

**3．功能函数：**描述所设计的功能函数。如果有多个函数，需要描述它们之间的关系。

getNumberOfLeaves：获取叶节点的数量

**4．测试数据：**设计测试数据，或具体给出测试数据。

提示：要求测试数据能全面地测试所设计程序的功能。

tree.insert(1);

tree.insert(2);

tree.insert(3);

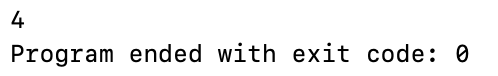
tree.insert(4);

tree.insert(5);

tree.insert(6);

tree.insert(7);

**5．测试情况：**给出程序的测试情况，分析运行结果，显示实验结果截图。

****

**6．实验总结：**写出实验过程中遇到的问题，以及问题的解决过程。分析算法的时间复杂度和空间复杂度，总结实验心得体会。

采用层级遍历的思想，因此时间复杂度为O(n)，n为节点个数

**7. 源代码：**

#include<iostream>

#include<queue>

**using** **namespace** std;

**class** BinaryTree

{

**public**:

**struct** Node

{

**int** value;

Node\* left;

Node\* right;

Node(**int** val) : value(val), left(**nullptr**), right(**nullptr**) {}

};

**void** insert(**const** **int**& value)//为二叉树插入结点

{

Node\* newNode = **new** Node(value);

**if** (!root)

{

root = newNode;

**return**;

}

queue<Node\*> queue;

queue.push(root);

**while** (!queue.empty())

{

Node\* current = queue.front();

queue.pop();

//从左到右依次插入结点

**if** (current->left == **nullptr**)

{

current->left = newNode;

**return**;

}

**else**

{

queue.push(current->left);

}

**if** (current->right == **nullptr**)

{

current->right = newNode;

**return**;

}

**else**

{

queue.push(current->right);

}

}

}

// 构造函数

BinaryTree()

: root(**nullptr**) {}

// 删除结点

**void** clear(Node\* node)

{

**if** (node)

{

clear(node->left);

clear(node->right);

**delete** node;

}

}

// 析构函数

~BinaryTree()

{

clear(root);

}

**int** getNumberOfLeaves()

{

**int** NumberOfLeaves = 0;

queue<Node\*>queue;

queue.push(root);

**while** (!queue.empty())

{

//采用层级遍历的思想实现

Node\* current = queue.front();

**if** (!current->left && !current->right)

NumberOfLeaves++;

queue.pop();

**if** (current->left != **nullptr**)

{

queue.push(current->left);

}

**if** (current->right != **nullptr**)

{

queue.push(current->right);

}

}

**return** NumberOfLeaves;

}

**private**:

Node\* root;

};

**int** main()

{

BinaryTree tree;

tree.insert(1);

tree.insert(2);

tree.insert(3);

tree.insert(4);

tree.insert(5);

tree.insert(6);

tree.insert(7);

cout << tree.getNumberOfLeaves() << endl;

}

**实验三**

1. **问题描述：**描述实验内容和要求以及需要解决的问题。

实现一个C++函数SwapTree()，用来交换二叉树中每一个结点的左右孩子。

**2．算法思想：**详细描述解决相应问题所需要的算法设计思想。

采用递归的思想，先交换根节点的左右子树，再对左右子树进行函数的递归调用

**3．功能函数：**描述所设计的功能函数。如果有多个函数，需要描述它们之间的关系。

SwapTree：交换节点的左右子树

**4．测试数据：**设计测试数据，或具体给出测试数据。

提示：要求测试数据能全面地测试所设计程序的功能。

tree.insert(1);

tree.insert(2);

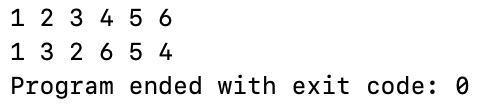
tree.insert(3);

tree.insert(4);

tree.insert(5);

tree.insert(6);

**5．测试情况：**给出程序的测试情况，分析运行结果，显示实验结果截图。

****

**6．实验总结：**写出实验过程中遇到的问题，以及问题的解决过程。分析算法的时间复杂度和空间复杂度，总结实验心得体会。

采用递归调用，因此时间复杂度为O(n)，n为节点个数

**7. 源代码：**

#include<iostream>

#include<queue>

**using** **namespace** std;

**class** BinaryTree

{

**public**:

**struct** Node

{

**int** value;

Node\* left;

Node\* right;

Node(**int** val) : value(val), left(**nullptr**), right(**nullptr**) {}

};

**void** insert(**const** **int**& value)//为二叉树插入结点

{

Node\* newNode = **new** Node(value);

**if** (!root)

{

root = newNode;

**return**;

}

queue<Node\*> queue;

queue.push(root);

**while** (!queue.empty())

{

Node\* current = queue.front();

queue.pop();

//从左到右依次插入结点

**if** (current->left == **nullptr**)

{

current->left = newNode;

**return**;

}

**else**

{

queue.push(current->left);

}

**if** (current->right == **nullptr**)

{

current->right = newNode;

**return**;

}

**else**

{

queue.push(current->right);

}

}

}

// 构造函数

BinaryTree()

: root(**nullptr**) {}

// 删除结点

**void** clear(Node\* node)

{

**if** (node)

{

clear(node->left);

clear(node->right);

**delete** node;

}

}

// 析构函数

~BinaryTree()

{

clear(root);

}

//层级打印树的内容

**void** print() **const**

{

queue<Node\*> queue;

queue.push(root);

**while** (!queue.empty())

{

Node\* current = queue.front();

queue.pop();

cout << current->value << " ";

**if** (current->left)

queue.push(current->left);

**if** (current->right)

queue.push(current->right);

}

cout << endl;

}

**void** SwapTree(Node\* node)

{

//采用递归的思想实现

**if** (node)

{

Node\* temp = node->left;

node->left = node->right;

node->right = temp;

SwapTree(node->left);

SwapTree(node->right);

}

}

Node\* getRoot()

{

**return** root;

}

**private**:

Node\* root;

};

**int** main()

{

BinaryTree tree;

tree.insert(1);

tree.insert(2);

tree.insert(3);

tree.insert(4);

tree.insert(5);

tree.insert(6);

tree.print();

tree.SwapTree(tree.getRoot());

tree.print();

}