哈尔滨工业大学计算机科学与技术学院

实验报告

|  |  |
| --- | --- |
| 课程名称： | 数据结构与算法 |
| 课程类型： | 必修 |
| 实验名称： | 树形结构与应用 |
|  |  |
| 班级： | 1703005 |
| 学号： | 1170300520 |
| 姓名： | 郭子阳 |

1. 实验目的

熟悉二叉树的建立与遍历。

1. 实验要求及实验环境

实验要求：实现递归、非递归的遍历，前序、中序和后序都要求实现

实验环境：macOS，VSCode，JDK 11

1. 设计思想（本程序中用到的所有数据类型的定义，核心算法的流程图等）

栈结构定义如下：

class Stack {

private ArrayList<TreeNode> nodes = new ArrayList<>();

public void push(TreeNode *node*) {

nodes.add(node);

}

public TreeNode pop() {

TreeNode tempNode = nodes.get(nodes.size() - 1);

nodes.remove(nodes.size() - 1);

return tempNode;

}

public boolean isEmpty() {

return nodes.isEmpty();

}

public TreeNode getTop() {

return nodes.get(nodes.size() - 1);

}

}

队列结构定义入下：

class Queue {

private ArrayList<TreeNode> nodes = new ArrayList<>();

public void enQueue(TreeNode *node*) {

nodes.add(node);

}

public TreeNode deQueue() {

return nodes.remove(0);

}

public TreeNode getFirst() {

return nodes.get(0);

}

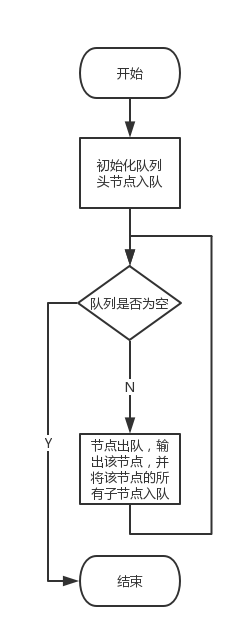
public boolean isEmpty() {

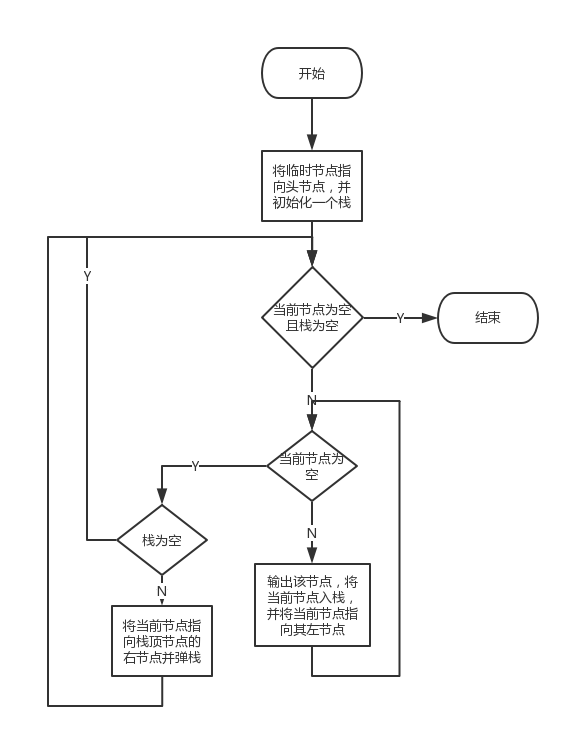
return nodes.isEmpty();

}

}

层序遍历二叉树：

非递归前序遍历二叉树：



1. 测试结果



1. 系统不足与经验体会

本次实验深刻理解了栈和队列的设计。

1. 源代码

import java.util.ArrayList;

import java.util.Scanner;

public class DSLab2 {

private static Scanner scanner = new Scanner(System.in);

private static TreeNode topNode = null;

public static void main(String[] args) {

while(true) {

menu();

}

}

private static void menu() {

System.out.println();

System.out.println();

System.out.println("1. 输入二叉树");

System.out.println("2. 遍历二叉树");

System.out.println("3. 判断完全二叉树");

System.out.println("4. 求公共祖先");

System.out.println("5. 退出");

int choice = 0;

do{

System.out.print("请输入选择：");

choice = scanner.nextInt();

}while(choice < 1 || choice > 5);

System.out.println();

System.out.println();

switch(choice) {

case 1:

try{

inputTree();

}catch(Exception e) {

System.out.println("输入数据有误！请检查后输入！");

topNode = null;

}

break;

case 2:

outputTree();

break;

case 3:

judgeTree();

break;

case 4:

findAncestor();

break;

case 5:

System.exit(0);

}

}

private static void inputTree() throws Exception {

System.out.println("请按照先序序列输入二叉树（回车间隔，空节点输入#）：");

String value = scanner.next();

if("#".equals(value)) {

topNode = null;

} else {

if(topNode == null) {

topNode = new TreeNode(value);

} else {

topNode.setValue(value);

}

topNode.setLeftNode(new TreeNode());

createTree(topNode.getLeftNode());

topNode.setRightNode(new TreeNode());

createTree(topNode.getRightNode());

}

System.out.println("二叉树建立完成！");

}

private static void createTree(TreeNode node) throws Exception {

String value = scanner.next();

if("#".equals(value)) {

node = null;

} else {

node.setValue(value);

node.setLeftNode(new TreeNode());

createTree(node.getLeftNode());

node.setRightNode(new TreeNode());

createTree(node.getRightNode());

}

}

private static void outputTree() {

System.out.println("1.先序遍历");

System.out.println("2.中序遍历");

System.out.println("3.后序遍历");

System.out.println("4.层序遍历");

System.out.print("请输入你的选择：");

int firstChoice = scanner.nextInt();

if(firstChoice == 4) {

levelTraverse();

return;

}

System.out.println("1.递归");

System.out.println("2.迭代");

System.out.print("请输入你的选择：");

int secondChoice = scanner.nextInt();

switch(firstChoice) {

case 1:

if(secondChoice == 1) {

if(topNode == null || topNode.getValue() == null) {

System.out.println("空二叉树！请输入数据！");

return;

}

recursivePreOrder(topNode);

}

if(secondChoice == 2) {

iteratedPreOrder();

}

break;

case 2:

if(secondChoice == 1) {

if(topNode == null || topNode.getValue() == null) {

System.out.println("空二叉树！请输入数据！");

return;

}

recursiveInOrder(topNode);

}

if(secondChoice == 2) {

iteratedInOrder();

}

break;

case 3:

if(secondChoice == 1) {

if(topNode == null || topNode.getValue() == null) {

System.out.println("空二叉树！请输入数据！");

return;

}

recursivePostOrder(topNode);

}

if(secondChoice == 2) {

iteratedPostOrder();

}

break;

default:

System.out.println("输入错误！");

}

}

private static void levelTraverse() {

Queue queue = new Queue();

queue.enQueue(topNode);

TreeNode tempNode = null;

while(!queue.isEmpty()) {

tempNode = queue.deQueue();

if(tempNode != null && tempNode.getValue() != null) {

System.out.print(tempNode.getValue() + " ");

}

if(tempNode.getLeftNode() != null && tempNode.getLeftNode().getValue() != null) {

queue.enQueue(tempNode.getLeftNode());

}

if(tempNode.getRightNode() != null && tempNode.getRightNode().getValue() != null) {

queue.enQueue(tempNode.getRightNode());

}

}

}

private static void recursivePreOrder(TreeNode node) {

if(node != null && node.getValue() != null) {

System.out.print(node.getValue() + " ");

recursivePreOrder(node.getLeftNode());

recursivePreOrder(node.getRightNode());

}

}

private static void recursiveInOrder(TreeNode node) {

if(node != null && node.getValue() != null) {

recursiveInOrder(node.getLeftNode());

System.out.print(node.getValue() + " ");

recursiveInOrder(node.getRightNode());

}

}

private static void recursivePostOrder(TreeNode node) {

if(node != null && node.getValue() != null) {

recursivePostOrder(node.getLeftNode());

recursivePostOrder(node.getRightNode());

System.out.print(node.getValue() + " ");

}

}

private static void iteratedPreOrder() {

if(topNode == null || topNode.getValue() == null) {

System.out.println("空二叉树！请输入数据！");

return;

}

Stack stack = new Stack();

TreeNode currentNode = topNode;

while(currentNode != null || !stack.isEmpty()) {

while(currentNode != null) {

if(currentNode.getValue() != null){

System.out.print(currentNode.getValue() + " ");

}

stack.push(currentNode);

currentNode = currentNode.getLeftNode();

}

if(!stack.isEmpty()) {

currentNode = stack.pop();

currentNode = currentNode.getRightNode();

}

}

}

private static void iteratedInOrder() {

if(topNode == null || topNode.getValue() == null) {

System.out.println("空二叉树！请输入数据！");

return;

}

Stack stack = new Stack();

TreeNode currentNode = topNode;

while(currentNode != null || !stack.isEmpty()) {

while(currentNode != null) {

stack.push(currentNode);

currentNode = currentNode.getLeftNode();

}

if(!stack.isEmpty()) {

currentNode = stack.pop();

if(currentNode.getValue() != null) {

System.out.print(currentNode.getValue() + " ");

}

currentNode = currentNode.getRightNode();

}

}

}

private static void iteratedPostOrder() {

if(topNode == null || topNode.getValue() == null) {

System.out.println("空二叉树！请输入数据！");

return;

}

Stack stack = new Stack();

TreeNode currentNode = topNode;

while(currentNode != null || !stack.isEmpty()) {

while(currentNode != null) {

currentNode.setFlag(1);

stack.push(currentNode);

currentNode = currentNode.getLeftNode();

}

while(!stack.isEmpty() && stack.getTop().getFlag() == 2) {

TreeNode tempNode = stack.pop();

if(tempNode.getValue() != null) {

System.out.print(tempNode.getValue() + " ");

}

}

if(!stack.isEmpty()) {

stack.getTop().setFlag(2);

currentNode = stack.getTop().getRightNode();

}

}

}

private static void judgeTree() {

if(topNode == null || topNode.getValue() == null) {

System.out.println("空二叉树！请输入数据！");

return;

}

Queue queue = new Queue();

queue.enQueue(topNode);

while(!queue.isEmpty()) {

TreeNode top = queue.getFirst();

if(top.getLeftNode() != null && top.getLeftNode().getValue() != null &&

top.getRightNode() != null && top.getRightNode().getValue() != null) {

queue.deQueue();

queue.enQueue(top.getLeftNode());

queue.enQueue(top.getRightNode());

continue;

}

if((top.getLeftNode() == null || top.getLeftNode().getValue() == null) &&

top.getRightNode() != null && top.getRightNode().getValue() != null) {

System.out.println("该树不是完全二叉树！");

return;

}

if(((top.getLeftNode() != null && top.getLeftNode().getValue() != null) &&

(top.getRightNode() == null || top.getRightNode().getValue() == null))||

((top.getLeftNode() == null || top.getLeftNode().getValue() == null) &&

(top.getRightNode() == null || top.getRightNode().getValue() == null))) {

queue.deQueue();

while(!queue.isEmpty()) {

top = queue.getFirst();

if((top.getLeftNode() == null || top.getLeftNode().getValue() == null) &&

(top.getRightNode() == null || top.getRightNode().getValue() == null)) {

queue.deQueue();

} else {

System.out.println("该树不是完全二叉树！");

return;

}

}

System.out.println("该树是完全二叉树！");

return;

}

}

System.out.println("该树是完全二叉树！");

return;

}

private static void findAncestor() {

System.out.println("请输入要查找祖先的两个节点的值（回车或空格间隔）：");

String one = scanner.next();

String another = scanner.next();

TreeNode target = findNearsetAncestor(topNode, one, another);

System.out.println("两个节点最近的祖先节点的值是" + target.getValue());

}

private static TreeNode findNearsetAncestor(TreeNode root, String one, String another) {

if((root == null || root.getValue() == null) ||

one.equals(root.getValue()) || another.equals(root.getValue())) {

return root;

}

TreeNode leftNode = findNearsetAncestor(root.getLeftNode(), one, another);

TreeNode rightNode = findNearsetAncestor(root.getRightNode(), one, another);

if(leftNode != null && leftNode.getValue() != null &&

rightNode != null && rightNode.getValue() != null) {

return root;

}

return (leftNode == null || leftNode.getValue() == null) ? rightNode : leftNode;

}

}

class TreeNode {

private String value;

private TreeNode leftNode;

private TreeNode rightNode;

private int flag;

public TreeNode() {}

public TreeNode(String value) {

this.value = value;

}

public void setValue(String value) {

this.value = value;

}

public String getValue() {

return value;

}

public void setLeftNode(TreeNode leftNode) {

this.leftNode = leftNode;

}

public TreeNode getLeftNode() {

return leftNode;

}

public void setRightNode(TreeNode rightNode) {

this.rightNode = rightNode;

}

public TreeNode getRightNode() {

return rightNode;

}

public void setFlag(int flag) {

this.flag = flag;

}

public int getFlag() {

return flag;

}

}

class Stack {

private ArrayList<TreeNode> nodes = new ArrayList<>();

public void push(TreeNode node) {

nodes.add(node);

}

public TreeNode pop() {

TreeNode tempNode = nodes.get(nodes.size() - 1);

nodes.remove(nodes.size() - 1);

return tempNode;

}

public boolean isEmpty() {

return nodes.isEmpty();

}

public TreeNode getTop() {

return nodes.get(nodes.size() - 1);

}

}

class Queue {

private ArrayList<TreeNode> nodes = new ArrayList<>();

public void enQueue(TreeNode node) {

nodes.add(node);

}

public TreeNode deQueue() {

return nodes.remove(0);

}

public TreeNode getFirst() {

return nodes.get(0);

}

public boolean isEmpty() {

return nodes.isEmpty();

}

}