结构：

采用的是三叉链表还存储每个二叉树的节点。

1.节点结构：

每个二叉树的节点包括：数据域、左孩子域、右孩子域。

public class BinaryTreeNode{

private Object date; //数据域

private BinaryTreeNode lchild; //左孩子域

private BinaryTreeNode rchild;//右孩子域

public BinaryTreeNode(){ //无参构造

date=null;

lchild=null;

rchild=null;

}

public BinaryTreeNode(Object date){ //有参构造

this.date=date;

lchild=null;

rchild=null;

}

}

类似链表，链表只不过是内部只有一个指针域，指向下一个节点。

2.二叉树结构

每棵二叉树内部结构只有一个根节点，通过根节点就能找到其他节点。

类似链表，链表内部只有头节点，通过头节点就可以找到其他节点

private BinaryTreeNode root;//根节点

public BinaryTree(){ //构造空二叉树

root=null;

}

public BinaryTree(BinaryTreeNode root){//构造以root为节点的二叉树

this.root=root;

}

遍历：

1.递归方法

（1）先序遍历（先根）

public void preRootTraverse(BiTreeNode T){ //先根遍历,递归

if(T!=null){

System.out.print(T.date); //先访问节点数据

preRootTraverse(T.lchild); //递归遍历左孩子

preRootTraverse(T.rchild); //递归遍历右孩子

}

}

（2）中序遍历（中根）

public void inRootTraverse(BiTreeNode T){ //中根遍历，递归

if(T!=null){

inRootTraverse(T.lchild); //先递归遍历左孩子，遍历完再访问节点

System.out.print(T.date);

inRootTraverse(T.rchild);//再递归遍历右孩子

}

}

（3）后序遍历（后根）

public void postRootTraverse(BiTreeNode T){ //后根遍历，递归

if(T!=null){

postRootTraverse(T.lchild);

postRootTraverse(T.rchild);

System.out.print(T.date); //最后访问数据

}

}

2.非递归方法

非递归算法实现需要借助栈才能实现。

我们这里栈用的是LinkedList。

（1）前序遍历

访问顺序是先访问根节点，再沿着该节点的左子树向下依次访问其左子树的根节点，直到最后访问的节点都无左子树为止，再继续向上先根遍历根节点的右子树。

访问过程中，我们需要借助栈来记载当前被访问节点的右孩子节点，以便先遍历完一个节点的左子树后能顺利进入这个节点的右子树进行遍历。

思路：先从根节点出发，沿着该节点的左子树向下搜索，在搜索构成中没遇到一个节点就先访问该节点（因为是先根，有孩子节点就是相应根节点就访问），并将相应的非空右孩子节点压入栈中。当左子树访问完，就从栈中弹出右子树用上述同样方法进行访问。

步骤：

1.创建一个栈对象，根节点入栈

2.栈非空，将栈顶点弹出并访问其数据

3.当前元素左孩子非空则访问数据，右孩子非空则压入栈中

4.重复2,3直到栈空

代码；

public void preRootTraverse(){ //先根遍历，非递归

BiTreeNode T = root;

if(T!=null){

LinkedList<BiTreeNode> stack = new LinkedList(); //构造栈

stack.push(T); //根节点入栈

while(!stack.isEmpty()){

T=stack.pop(); //取出栈顶元素

System.out.print(T.date);//先根遍历，先访问根节点数据

while (T!=null){ //出栈节点不为空

if(T.lchild!=null) //访问出栈节点的左孩子

System.out.print(T.lchild.date); //访问左孩子

if(T.rchild!=null) //访问出栈的节点的右孩子

stack.push(T.rchild); //非空入栈

T=T.lchild; //继续向下遍历当前栈顶元素的左子树。

}

}

}

}

（2）中序遍历

需要借助栈来记载遍历过程中所经历的而未被访问的所有节点，以便遍历完一个节点的左子树后能顺利的返回它的父节点。

思想：从二叉树的根节点出发，沿着该节点的左子树向下搜索，在搜索过程中将每个节点都依次压入栈中，直到二叉树的最左下节点压入栈中为止，最左下节点的左孩子为null，然后从栈中弹出栈顶节点并进行访问，访问完再次进入该节点的右子树并用上述方法遍历该节点的右子树。

步骤：

1.创建一个空栈，根节点压入栈中

2.栈非空，将栈顶元素的非空左孩子节点依次进栈

3.此时栈顶元素是最左下的节点的左孩子为null，需要pop()空节点

4.栈顶元素出栈并访问栈顶元素，该元素的非空右孩子压入栈中

5.重复3,4直到栈空为止

代码：

public void inRootTraverse(){ //中根遍历，非递归

BiTreeNode T = root;

LinkedList<BiTreeNode> stack = new LinkedList<>();

stack.push(T); //根节点入栈

while(!stack.isEmpty()){

while(stack.peek()!=null){ //将栈顶点的左孩子节点相继入栈

stack.push(stack.peek().lchild);

}

stack.pop(); //最左节点的左孩子也加入到栈中，取出来。

if(!stack.isEmpty()){

T=stack.pop();//取出栈顶节点

System.out.print(T.date);//访问栈顶节点数据

stack.push(T.rchild);//把右孩子压入栈中

//因为是从下面遍历的右孩子

//当前的栈顶元素是从下到上的对应根节点

}

}

}

（3）后序遍历

后根遍历是先处理左子树，再处理右子树，然后才访问根节点，所以遍历搜索过程也冲根节点出发，沿着左子树向下搜索，但是在搜索过程中需要判断遇到的每一个节点是否是访问过，若是，则不立即访问，将该节点压入栈中保存，然后遍历该节点的左子树。当该节点的左子树遍历完毕，返回该节点，但是不能访问该节点，应继续访问该节点的右子树。当左右子树全部访问完毕后才能够弹出该节点，访问该节点。我们需要一个boolean变量flag来记录当前栈顶元素是否被访问过，用一个节点指针p来记录当前遍历过程中最后一个被访问的节点，如果当前栈顶节点的右孩子节点为空，或者就是p指向的节点，则表明当前节点的右子树已经遍历完毕，此时应该访问栈顶元素。

步骤：

1.创建空栈，保存根节点

2.栈非空，则栈顶节点的非空左孩子进栈

3.若栈非空，查看栈顶节点，若该节点的右孩子为空，或者与p相等，则弹出栈顶节点并访问它，同时将p指向刚才访问的弹出的栈顶元素，并设置flag为true，否则将栈顶节点的右孩子入栈，并设置flag为false

4.若flag为true执行3，否则重复2，3，直到栈为空。

代码：

public void postRootTraverse(){ //后根遍历，非递归

BiTreeNode T =root;

LinkedList<BiTreeNode> stack = new LinkedList<>();

stack.push(T);

BiTreeNode p=null;//p指向刚才被访问的节点

boolean flag; //访问标记

while(!stack.isEmpty()){

while(stack.peek()!=null){ //找到当前栈顶节点最下边的左子孙

stack.push(stack.peek().lchild);

}//因为最下边左子孙为叶子节点，它的左孩子也进栈了，为null空节点

stack.pop();//空节点退栈

while(!stack.isEmpty()){

T=stack.peek();//获取当前栈顶元素

if(T.rchild==null||T.rchild==p){ //当前栈顶元素右子树访问完毕

System.out.print(T.date);//访问

stack.pop();//出栈

p=T; //p指向当前被访问的节点

flag=true; //设置访问过的标志

}

else { //如果还有右孩子，继续遍历

stack.push(T.rchild);//把右孩子入栈

flag=false;//未访问

}

if(!flag) //当flag为true的时候说明此时的栈顶的下面没有需要继续入栈的元素，输出即可，然后下一个栈顶元素判断

//当为false的时候说明当前栈顶元素还有右子树没有访问完需要继续访问，就不跳出循环

break;

}

}

}

（4）层次遍历

层次遍历过程中需要一个队列来作为辅助的存储结构。在遍历开始的时候，首先根节点入队，然后从队列总取出队首节点进行处理，取出的节点的左孩子入队，右孩子入队。在处理每个节点的时候，都是先访问该节点。这样就能一层一层的访问了。

步骤：

1.创建一个空的队列，根节点不为空，入队

2.如果队列不为空，则队首节点出队，访问该节点，其节点的左孩子不为空入队，右孩子不为空，入队

3.重复2，直到队列为空

代码：

public void levelTraverse(){ //层次遍历，从左到右

LinkedList<BiTreeNode> queue = new LinkedList<>(); //层次遍历用队列

BiTreeNode T = root;

if(T!=null){

queue.offer(T); //根节点进队

while (!queue.isEmpty()){

T=queue.poll(); //输出当前队首

System.out.print(T.date);

if(T.lchild!=null) //左孩子入队

queue.offer(T.lchild);

if(T.rchild!=null) //右孩子入队

queue.offer(T.rchild);

}

}

}

3.二叉树统计节点

（1）层次遍历输出的时候进行计数

（2）递归

Public int countNode(BiTreeNode T){

if(T==null) //根节点为空

return 0;

else

return countNode(T.lchild)+countNode(T.rchild)+1;

}

4.二叉树深度

Public int getDepth(BiTreeNode T){

if(T!=null){

int lDepth = getDepth(T.lchild);//当前节点T的左子树深度

int rDepth = getDepth(T.rchild);//当前节点T的右子树深度

return 1+(lDepth>rDepth?lDepth:rDepth);//返回左右子树深度最大值加1

}

return 0；

}

5.判断二叉树是否相等

Public Boolean isEqual(BiTreeNode T1, BiTreeNode T2){

if(T1==null&&T2==null)//同时为空

return false;

if(T1!=null&&T2!=null){

if(T1.date.equals(T2.date)) //当前节点相同的时候

if(isEqual(T1.lchild,T2.rchild)) //根节点相同，比较当前节点的左子树

if(isEqual(T1.rchild,T2.rchild))//左子树相同，比较当前节点的右子树

return true;

}

return false;

}