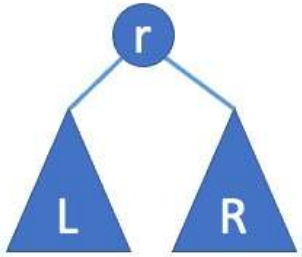
5.4.2 二叉树遍历的递归算法



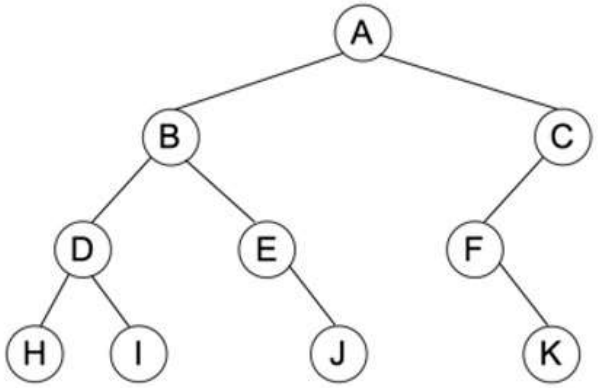
前序遍历方案
(1) 访问根结点r
(2) 前序遍历左子树L
(3) 前序遍历右子树R

中序遍历方案
(1) 中序遍历左子树L
(2) 访问根结点r
(3) 中序遍历右子树R

后序遍历方案
(1) 后序遍历左子树L
(2) 后序遍历右子树R
(3) 访问根结点r

5.4.2 二叉树遍历的递归算法

遍历二叉树并按序输出结点数据



前序遍历：<A, B, D, H, I, E, J, C, F, K>

中序遍历：<H, D, I, B, E, J, A, F, K, C>

后序遍历：<H, I, D, J, E, B, K, F, C, A>

根结点位置：

- 前序遍历的最前
- 后序遍历的最后
- 中序遍历出现在左子树和右子树遍历结果之间

单选题 1分

设二叉树各结点的数据互异，遍历二叉树并输出结点数据，什么情况下，前序遍历与中序遍历的结果一定相同。

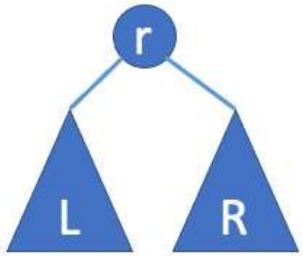
A 二叉树是满树

B 二叉树没有度为2的结点

C 二叉树所有结点没有左子结点

D 二叉树所有结点没有右子结点

问题：设二叉树各结点的数据互异，遍历二叉树并输出结点数据，什么情况下，前序遍历与中序遍历的结果一定相同。



- **前序遍历**：$r\ L^{pre}\ R^{pre}$

  ||

- **中序遍历**：$L^{in}\ r\ R^{in}$

- $L^{in} = \epsilon$，即根结点的左子树 L 为空树，所以 $L^{pre} = \epsilon$
- $R^{pre} = R^{in}$，因此可递归证明右子树 R 中各节点的左子树为空

递归算法

算法5-2. 前序遍历二叉树 PreOrder(*tree*)

```
if tree ≠ NIL then  //空树不做处理，直接返回
| Visit(tree)              //访问根结点
| PreOrder(tree.left)   //前序遍历左子树
| PreOrder(tree.right)  //前序遍历右子树
end
```

算法5-3. 中序遍历二叉树 InOrder(*tree*)

```
if tree ≠ NIL then
| Inorder(tree.left)   //中序遍历左子树
| Visit(tree)              //访问根结点
| InOrder(tree.right)  //中序遍历右子树
end
```

算法5-4. 后序遍历二叉树 PostOrder(*tree*)

```
if tree ≠ NIL then
| Postorder(tree.left)   //后序遍历左子树
| PostOrder(tree.right)  //后序遍历右子树
| Visit(tree)              //访问根结点
end
```

后序遍历递归算法的应用

算法5-5. 计算二叉树高度 Height(*tree*)

输入：二叉树*tree*

输出：二叉树的高度值

```
if tree = NIL then  //空树
| return 0    //空树的高度为0
else
| h_left ← Height(tree.left)          //遍历左子树，求子树的高度
| h_right ← Height(tree.right)        //遍历右子树，求子树的高度
| return Max(h_left, h_right) + 1     //树的高度等于左右子树高度
                                      // 的较大值加1
end
```

表达式树

算术表达式：由常数、运算符和括号组成，有前缀、中缀、后缀三种表示法

- 中缀表示法：9 + (6 + 3 * 2) / (8 / (5 - 3))
- 前缀表示法：+ 9 / + 6 * 3 2 / 8 - 5 3
- 后缀表示法：9 6 3 2 * + 8 5 3 - / / +

表达式树：表示算术表达式（非线性）逻辑结构的二叉树

中缀表达式的递归定义

(1) 单个常数是表达式

(2) 若exp1、exp2是表达式，则 (exp1) op (exp2) 也是表达式 （op表示运算符）

转换

表达式树

(1) 单个常数用单根树表示

(2) 若表达式包含多个常数及运算符，将表达式分解成(expr_left) op (expr_right)

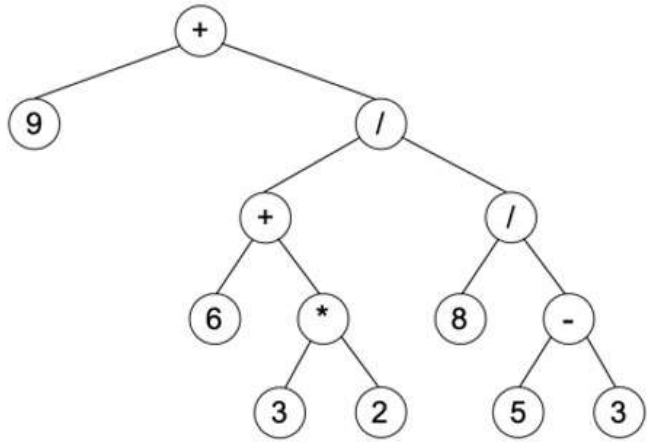
(3) 用根结点表示运算符op，并用左右子树分别表示expr_left和expr_right（递归）

表达式树

$9 + (6 + 3 * 2) / (8 / (5 - 3))$

中缀表达式

转换

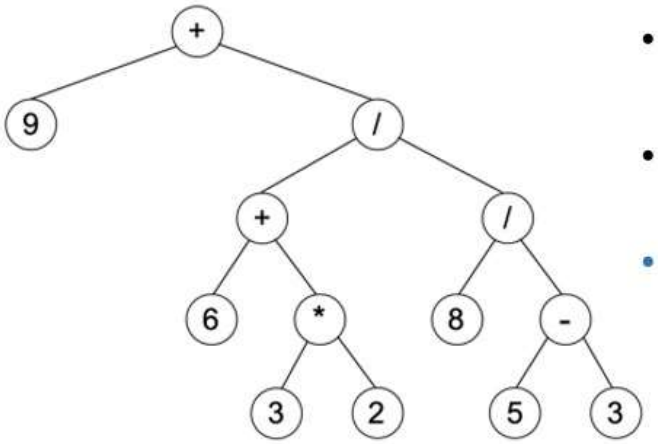


表达式树

- 叶结点都对应常数
- 每个中间结点都存放一个运算符
- 表达式树是满树
- 根结点是表达式中优先级最低的运算符

表达式树

表达式树



表达式：9 + (6 + 3 * 2) / (8 / (5 - 3))

- 前序遍历结果：+ 9 / + 6 * 3 2 / 8 - 5 3 = 前缀表示法
- 后序遍历结果：9 6 3 2 * + 8 5 3 - / / + = 后缀表示法
- 中序遍历结果：9 + 6 + 3 * 2 / 8 / 5 – 3 ≠ 中缀表示法

没有括号，不是正确的中缀表达式！

需要对左右子树生成的表达式加上括号

将表达式树转换成中缀表达式的算法

算法5-6. PrintInfixExpression(*tree*)

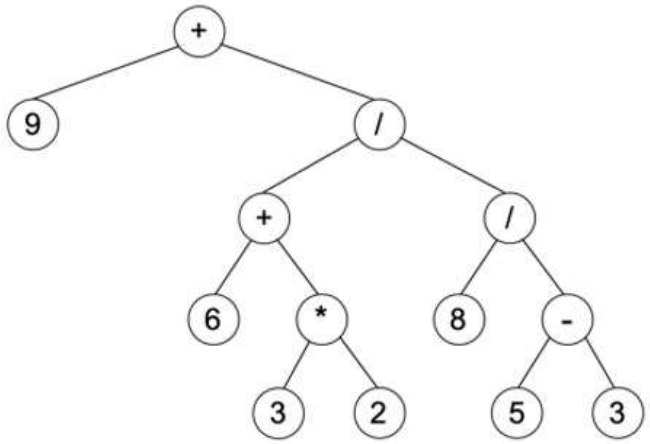
输入：非空二叉树*tree*

输出：中缀表达式（含冗余括号）

```
if tree.left ≠ NIL then     //左子树非空
| print（                   //输出左括号
| PrintInfixExpression(tree.left)        //中序遍历左子树并生成表达式
| print ）                  //输出右括号，与前面左括号一起将表达式加上括号
end
print  tree.data            //若结点是叶结点，输出常数；否则，输出运算符
if tree.right ≠ NIL then   //右子树非空
| print（                   //输出左括号
| PrintInfixExpression(tree.right)       //中序遍历右子树并生成表达式
| print ）                  //输出右括号，与前面左括号一起将表达式加上括号
end
```

表达式树

表达式树



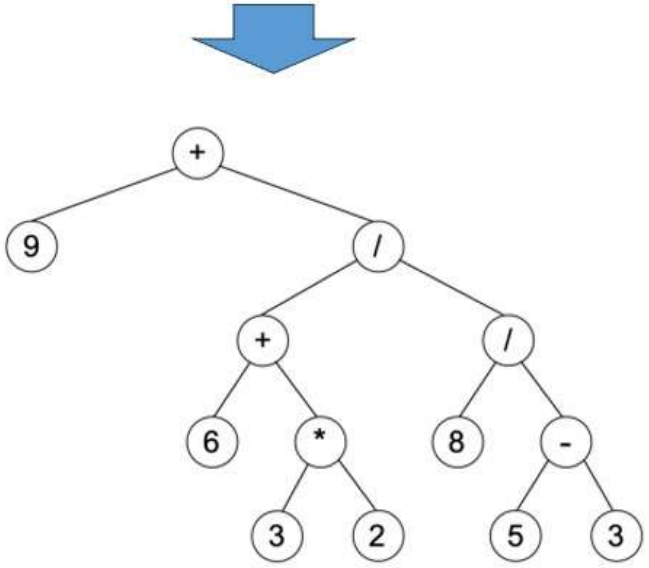
表达式：9 + (6 + 3 * 2) / (8 / (5 - 3))

- **前序遍历结果**：+ 9 / + 6 * 3 2 / 8 - 5 3　　前缀表示法
- **后序遍历结果**：9 6 3 2 * + 8 5 3 - / / +　　后缀表示法
- **算法5-6**：(9)+(((6)+((3)*(2)))/((8)/((5)-(3))))　中缀表示法

- 通过遍历将表达式树转换为算术表达式的时间复杂度是O(n), n表示结点数目
- 思考：如何将算术（中缀）表达式转换为表达式树，分析需要使用的关键数据结构

*表达式树的构建

表达式：9 + (6 + 3 * 2) / (8 / (5 - 3))



表达式树

算法1：由中缀表达式直接建树，每次从表达式中查找优先级最低的运算符生成树（子树）根结点，然后递归转换运算符两边的表达式为根结点的左右子树

- 难度大，时间效率低

算法2：先将中缀表达式转换成后缀表达式，然后使用后缀表达式建树。建树过程与从后缀表达式生成中缀表达式的过程一致！

- 时间效率高（线性时间O(n)）

思考：算法2需要两次转换，每次都使用栈存放中间结果，可否将两次转换合并成一次？这种情况下，需要多少栈？栈中存放什么数据？

5.4.3 二叉树遍历的非递归算法

递归算法的问题：

(1) 存在不支持递归算法的程序设计语言

(2) 递归算法在运行中，需要系统在内存栈中分配空间保存函数的参数、返回地址以及局部变量等，运行效率较低

(3)系统对每个进程分配的栈容量有限，如果二叉树的深度太大造成递归调用的层次太高，容易导致栈溢出

非递归算法实现的关键：使用栈结构模拟函数调用中系统栈的工作原理

算法5-7：非递归前序遍历 PreOrder(*tree*)

```
InitStack(stack)    //初始化栈stack，用于存放结点
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do        //当前结点不是空结点
| | Visit(tree)              //访问结点
| | Push(stack, tree)        //结点压入栈
| | tree ← tree.left         //沿左分支下移
| end
| if IsEmpty(stack) = false then   //如果栈不为空
| | tree ← Top(stack)
| | Pop(stack)               //弹出栈顶结点
| | tree ← tree.right        //移到栈顶结点的右子树
| end
end
DestroyStack(stack)
```

沿左分支下移，并将经过的结点压入栈

tree=NIL表示左子树为空，或者左子树遍历结束，则从栈中弹出结点，开始遍历结点的右子树

算法5-8：非递归中序遍历 InOrder(*tree*)

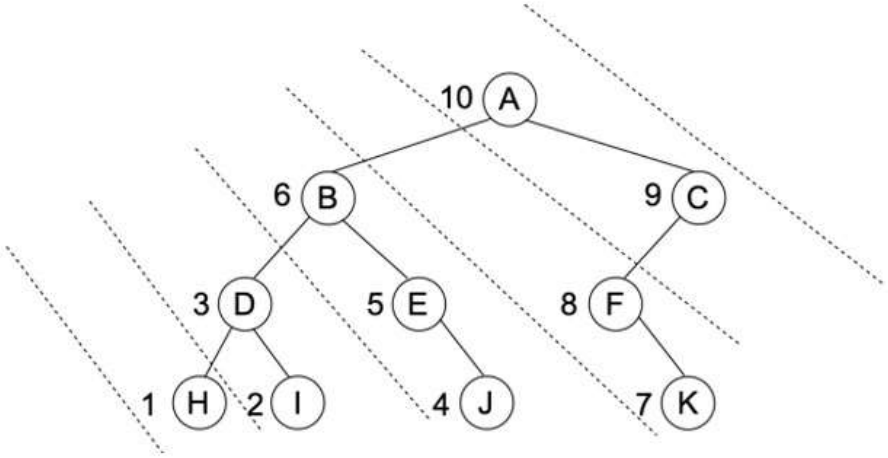
```
InitStack(stack)    //初始化栈stack，用于存放结点
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do        //当前结点不是空结点
| | Push(stack, tree)   //结点压入栈
| | tree ← tree.left    //沿左分支下移
| end
| if IsEmpty(stack) = false then   //如果栈不为空
| | tree ← Top(stack)
| | Visit(tree)         //访问栈顶结点
| | Pop(stack)          //弹出栈顶结点
| | tree ← tree.right   //移到栈顶结点的右子树
| end
end
DestroyStack(stack)
```

沿左分支下移，并将经过的结点压入栈

tree=NIL表示左子树为空，或者左子树遍历结束，则从栈中弹出结点，开始遍历结点的右子树

5.4.3 二叉树遍历的非递归算法

后序遍历算法的非递归化



二叉树所有结点的后序遍历次序，用各结点左边的数字表示

后序遍历次序的结点序列可分为多个段，用虚线分开，每段中的结点具有以下性质：

(1) 段中各结点的访问次序是连续的，并且最先访问（次序最小）的结点没有右子结点

(2) 段中若有多个结点，则次序相邻的任意两个结点，次序小的结点是次序大的结点的右子结点

(3) 段中次序最大的结点如果不是根，则是其父结点的左子结点

算法5-9：非递归后序遍历 PostOrder(*tree*)

```
InitStack(stack)
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do  //当前结点不是空结点
| | Push(stack, tree)  //结点压入栈
| | tree ← tree.left  //沿左分支下移
| end
|
| top ← Top(stack)     //stack非空，top指向栈顶结点
| pre_top ← NIL        //初始化 pre_top
|                      //如果栈顶结点的右子树为空，开始从栈弹出结点
```

沿左分支下移，并将经过的结点压入栈

开始时，*pre_top* = NIL

(continue)

```
| while IsEmpty(stack) = false 且 top.right = pre_top do
| | Visit(top)          //访问当前栈顶结点
| | pre_top ← top       //栈顶结点传给pre_top
| | Pop(stack)          //弹出栈顶结点
| | if IsEmpty(stack) = false then
| | | top ← Top(stack)  //栈非空，top指向新的栈顶结点
| | else
| | | top ← NIL         //空栈，top赋值NIL，结束遍历
| | end
| end
| if top ≠ NIL then
| | tree ← top.right    //移到栈顶结点的右子树并开始遍历
| end
end
DestroyStack(stack)
```

从右子树为空的结点开始。依次弹出各段中的结点

若弹出的结点是新栈顶结点的右子结点，继续弹出

5.4.3 二叉树遍历的非递归算法

递归算法的问题：

(1) 存在不支持递归算法的程序设计语言

(2) 递归算法在运行中，需要系统在内存栈中分配空间保存函数的参数、返回地址以及局部变量等，运行效率较低

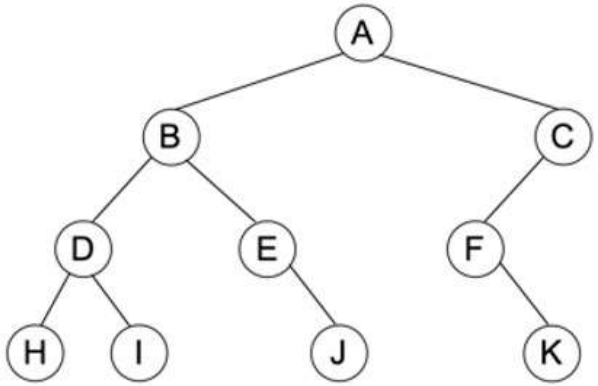
(3)系统对每个进程分配的栈容量有限，如果二叉树的深度太大造成递归调用的层次太高，容易导致栈溢出

非递归算法实现的关键：使用栈结构模拟函数调用中系统栈的工作原理

非递归算法分析：前序、中序和后序的非递归算法均使用一个存放结点的栈实现，并且在遍历过程中每个结点都有且仅有1次入栈和1次出栈的机会，因此算法的时间复杂度和空间复杂度都是O(n)，其中n表示二叉树的结点数

层序遍历

广度优先遍历：从根结点开始，从上至下按层访问每个结点，并且每层的结点按照从左向右的顺序进行处理



层序遍历：<A, B, C, D, E, F, H, I, J, K >

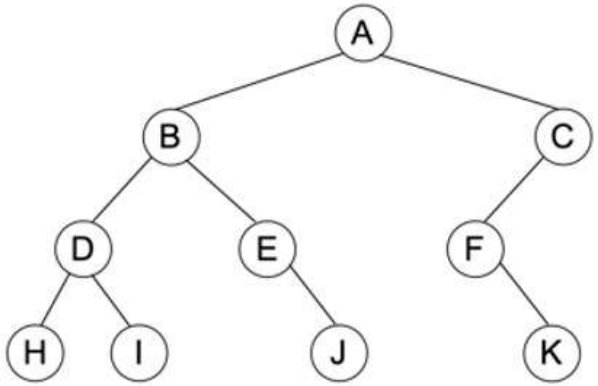
层序遍历方案通常设计成非递归的算法，可以用队列结构来实现

算法5-10：层序遍历二叉树 LevelOrder(*tree*)

```
InitQueue(queue)            //初始化队列queue，用于存放结点
EnQueue(queue, tree)        //树根结点入队
while IsEmpty(queue) = false do
| node_ptr ← GetFront(queue)  //取出队首结点
| DeQueue(queue)             //队首出队
| if node_ptr ≠ NIL then      //结点非空
| | Visit(node_ptr)           //访问结点
| | EnQueue(queue, node_ptr.left)   //左子结点入队
| | EnQueue(queue, node_ptr.right)  //右子结点入队
| end
end
DestroyQueue(queue)
```

层序遍历

广度优先遍历：从根结点开始，从上至下按层访问每个结点，并且每层的结点按照从左向右的顺序进行处理

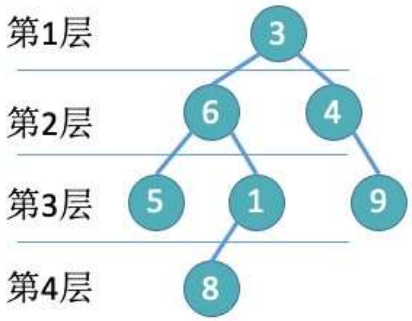


层序遍历：<A, B, C, D, E, F, H, I, J, K >

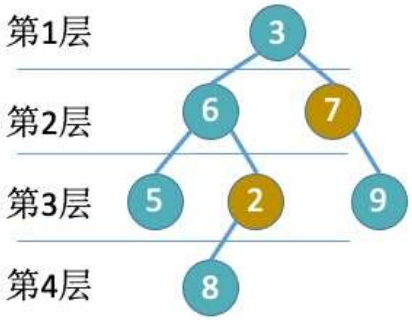
算法分析：二叉树的每个结点都会入队1次，然后出队1次，因此层序遍历的时间复杂度为O(n)，与深度优先遍历相同

二叉树遍历的应用：奇偶树I

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，而在偶数层的结点元素都是偶数，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。



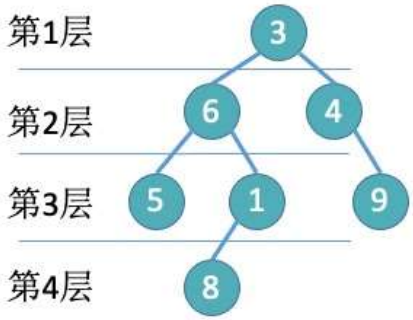
奇偶树



非奇偶树

二叉树遍历的应用：奇偶树I

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，而在偶数层的结点元素都是偶数，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。



奇偶树

算法1：层序遍历

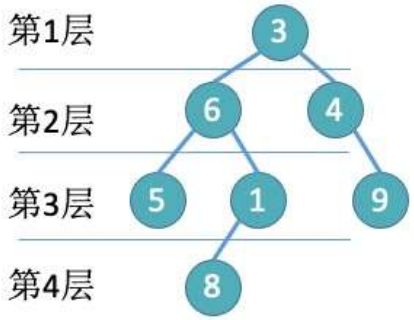
- 从上至下依层遍历所有结点，同时记录结点所在层数

  (1)结点队列：记录结点
  (2)层数队列：记录各结点所在层

- 对出队结点判断其元素及层数的奇偶性是否一致

二叉树遍历的应用：奇偶树I

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，而在偶数层的结点元素都是偶数，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。

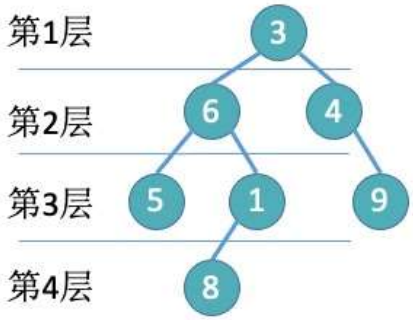


奇偶树

```
1.  EnQueue(node_queue, tree)    //根结点入队
2.  EnQueue(level_queue, 1)      //根结点所在层
3.  is_parity_tree ← true        //判定结果的初始值
4.  while is_parity_tree = true 且 IsEmpty(node_queue) = false do
5.  |  node_ptr ← DeQueue(node_queue)
6.  |  level ← DeQueue(level_queue)
7.  |  if node_ptr ≠ NIL then      //非空结点
8.  |  |  if node_ptr.data % 2 ≠ level % 2 then  //奇偶性不一致
9.  |  |  |  is_parity_tree ← false  //非奇偶树
10. |  |  end
11. |  |  EnQueue(node_queue, node_ptr.left)  //左子结点及层数入队
12. |  |  EnQueue(level_queue, level+1)
13. |  |  EnQueue(node_queue, node_ptr.right) //右子结点及层数入队
14. |  |  EnQueue(level_queue, level+1)
15. |  end
16. end
17. return is_parity_tree
```

二叉树遍历的应用：奇偶树I

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，而在偶数层的结点元素都是偶数，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。



奇偶树

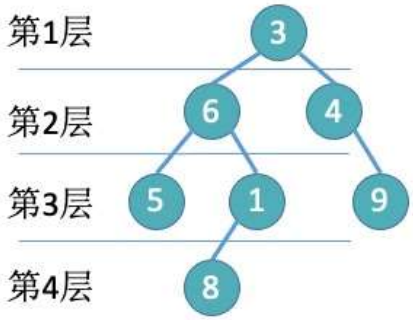
- 二叉树性质：根结点在第1层，所有非根结点的层数是其父结点的层数加1

算法2：前序遍历

- 深度优先遍历二叉树，把结点所在层数作为参数传递给子结点
- 判断当前结点的元素及层数的奇偶性是否一致

二叉树遍历的应用：奇偶树I

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，而在偶数层的结点元素都是偶数，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。



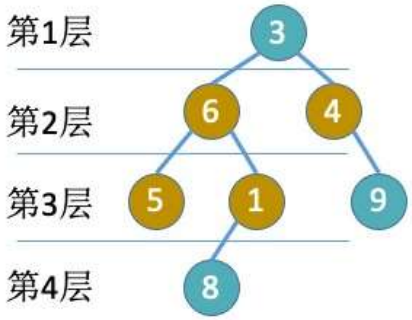
奇偶树

算法：IsParityTree(tree, level)
输入：二叉树tree, 树根结点所在层数level
输出：是否奇偶树的判定结果

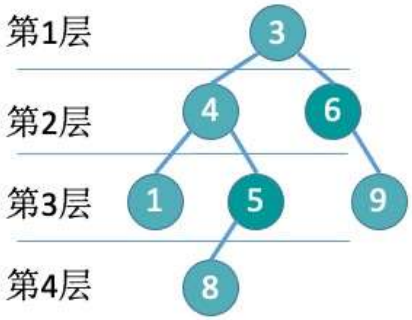
```
1. if tree = NIL then   //空树树奇偶树
2. |  return true
3. end
4. if tree.data % 2 ≠ level % 2 then
5. |  return false
6. else
7. |  return IsParityTree(tree.left, level+1) and
8.              IsParityTree(tree.right, level+1)  //判断左右子树
9. end
```

二叉树遍历的应用：奇偶树II（LeetCode1609）

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，在偶数层的结点元素都是偶数，并且各层结点从左向右按递增序排列，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。



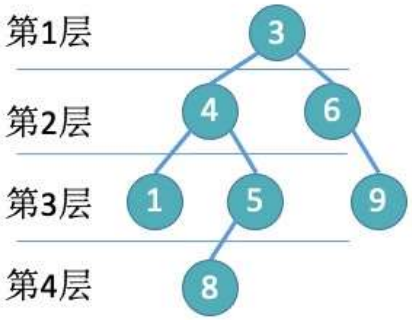
非奇偶树



奇偶树

二叉树遍历的应用：奇偶树II

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，并且各层结点从左向右按递增序排列在偶数层的结点元素都是偶数，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。



层序遍历

```
1.  EnQueue(node_queue, tree)
2.  EnQueue(level_queue, 1)
3.  pre_node ← NIL        //记录前一个出队的非空结点
4.  pre_level ← 0         //记录前一个出队的非空结点层数，初始值0
5.  is_parity_tree ← true        //判定结果的初始值
6.  while is_parity_tree = true 且 IsEmpty(node_queue) = false do
7.  | node_ptr ← DeQueue(node_queue)
8.  | level ← DeQueue(level_queue)
9.  | if node_ptr ≠ NIL then       //非空结点
10. | | if node_ptr.data % 2 ≠ level % 2 then  //奇偶性不一致
11. | | | is_parity_tree ← false    //结点与层数奇偶性不一致，非奇偶树
12. | | end          //如果前个结点与当前结点同层，比较数据
13. | | if pre_level = level 且 pre_node.data ≥ node_ptr.data then
14. | | | is_parity_tree ← false          //不满足递增条件，非奇偶树
15. | | end
16. | | pre_node ← node  //记录当前结点
17. | | pre_level ← level  //记录当前结点层数
18. | | EnQueue(node_queue, node_ptr.left)  //左子结点及层数入队
19. | | EnQueue(level_queue, level+1)
20. | | EnQueue(node_queue, node_ptr.right) //右子结点及层数入队
21. | | EnQueue(level_queue, level+1)
22. | end
23. end
24. return is_parity_tree
```

单选题 1分

层序遍历是按层从上至下，然后在同一层从左向右连续访问结点。那么前序遍历对同一层结点是按什么顺序进行访问？

A 与层序遍历一样，从左向右连续访问

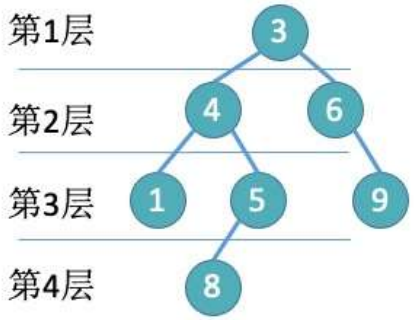
B 一定从左向右访问，但不一定连续

C 一定从左向右访问，且一定不连续（设有多个结点）

D 遍历方案不同，顺序可以从左向右，也可以从右向左

二叉树遍历的应用：奇偶树II

问题描述：设二叉树所有结点的元素都是正整数，如果在奇数层的结点元素都是奇数，并且各层结点从左向右按递增序排列在偶数层的结点元素都是偶数，则该二叉树称作奇偶树。设计判断二叉树是否奇偶树的算法。



奇偶树

前序遍历

算法：IsParityTree(tree, level)
输入：二叉树tree, 树根结点所在层数level
输出：是否奇偶树的判定结果
辅助空间：顺序表pre_nodes, 记录各层前一个访问结点，初始值为NIL

```
1.  if tree = NIL then
2.  |  return true
3.  end
4.  if tree.data % 2 ≠ level % 2 then
5.  |  return false
6.  end
7.  if pre_nodes[level] ≠ NIL 且 pre_nodes[level].data ≥ tree.data then
8.  |  return false
9.  end
10. pre_nodes[level] ← tree  //记录当前结点
11. return IsParityTree(tree.left, level+1) and
12.          IsParityTree(tree.right, level+1)  //判断左右子树
```

5.4.4 二叉树的序列化与反序列化

二叉树的序列化：按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

序列化的作用：将树的非线性结构转换成线性结构，便于使用线性表或字符串等存储

二叉树的反序列化：根据线性序列重构原始的二叉树

问题：

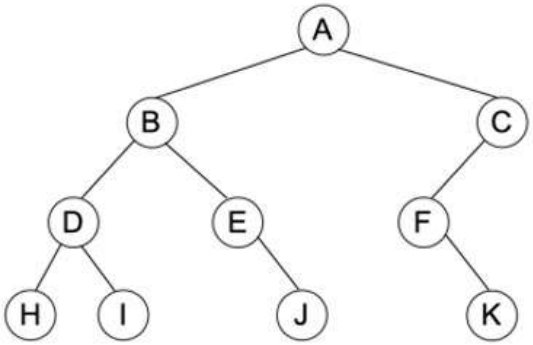
- 完全二叉树的顺序存储是一种序列化方案，并且可以根据结点间的相对位置确定它们之间的逻辑关系，重构出二叉树
- 但该方法对于一般的二叉树可能造成空间浪费，在最坏情况下，空间复杂度达到$O(2^n)$

- 常用的前序遍历或层序遍历算法，产生的结点序列只包含了树结构的部分信息，通常无法重构二叉树

5.4.4 二叉树的序列化与反序列化

二叉树的序列化：按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

二叉树的反序列化：根据线性序列重构原始的二叉树



前序遍历：<A, B, D, H, I, E, J, C, F, K>

从序列中，最多只能确定A是根结点，其它信息，如左子树包含哪些结点等无法确定

问题：如何使前序遍历的结果能够重构二叉树?

5.4.4 二叉树的序列化与反序列化

二叉树的序列化：按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

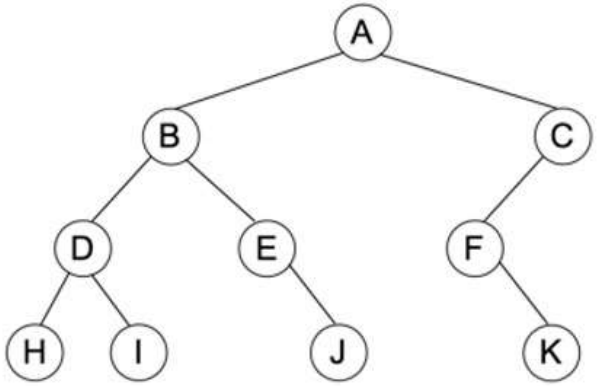
二叉树的反序列化：根据线性序列重构原始的二叉树

序列化方法

- 用特殊符号#表示空结点
- 当在前序遍历过程中遇到空结点或空子树时，不是直接返回，而是输出符号#，从而将空结点也标记在序列中

前序序列

二叉树前序序列化：前序遍历二叉树，如果结点非空，输出结点数据，否则输出#



通过在结点序列中插入空记号，记录二叉树的非线性结构

<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>

前序序列

算法5-11：二叉树前序序列化 PreOrderSerialize(*tree*)

输入：二叉树*tree*
输出：二叉树的前序序列

```
if tree = NIL then  //空树
| print #    //输出特殊符号，代表空结点
else
| print  tree.data   //输出结点数据
| PreOrderSerialize (tree.left)     //对左子树前序序列化
| PreOrderSerialize (tree.right)    //对右子树前序序列化
end
```

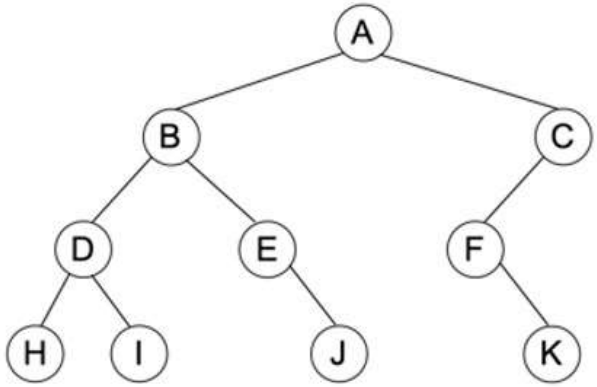
基于前序遍历算法

前序序列

前序序列的反序列化

从前序序列先端依次读取数据，执行下面的操作：

- 如果读取的数据是#，则返回NIL，表示空结点或空树
- 否则新建二叉树结点，把数据代入结点并递归地重构结点的左子树和右子树，然后返回结点。



<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>

前序序列

算法5-12：二叉树前序序列的反序列化 PreOrderDeSerialize(*preorder*, *n*)

输入：存放二叉树前序序列的线性表*preorder*，表中元素个数*n* (n>0)
输出：二叉树
全局变量：k，初始值为-1

```
k ← k + 1
tree ← NIL  //初始化一个空树
if k < n then  //k是线性表的有效序号
| data ← Get(preorder, k)      //读出线性表第k个元素
| if data ≠ # then     //非空记号
| | tree ← new BinaryTreeNode( )  //新建二叉树结点
| | tree.data ← data              //代入数据
| | tree.left ← PreOrderDeSerialize(preorder, n)  //重构左子树
| | tree.right ← PreOrderDeSerialize(preorder, n)  //重构右子树
| end
end
return tree  //返回新建的二叉树或空树
```