15.二叉树的遍历，深度（栈），广度（队列）的遍历：

·基本概念：

每个节点最多有两颗子树，左子树和右子树，次序不可以颠倒

非空二叉树上第n层上至多有2^(n-1)个元素

深度为h的二叉树至多有2^h-1个节点

满二叉树：所有终端都在同一层次，且终端节点的度为2，在满二叉树中若其深度为h，则其所包含的阶段节点数必为2^h-1

完全二叉树：除了最大的层次即成为一颗满二叉树且层次最大的那层的所有节点均向左靠齐，即集中在左面的位置，不能有空位置。对于完全二叉树：设一个节点为i则其父节点为i/2，2i为左子节点，2i+1为右子节点。

二叉树的顺序存储和链式存储：顺序存储在遍历速度上有一定的优势，但因占空间较大，所以常常使用的是链式存储结构

·二叉树的遍历：

前序遍历：根节点🡪左子树🡪右子树

中序遍历：左子树🡪根节点🡪右子树

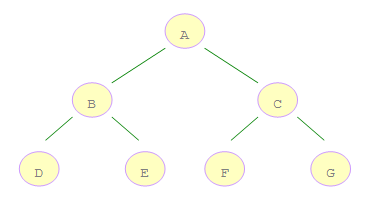
后续遍历：左子树🡪右子树🡪根节点

·如 现有以下一颗树：A     B          B1               B11          B2               B22     C          C1               C11               C12          C2     D          D1               D11  
  
第一种方式深度优先遍历 （最终返回的一棵压扁的树，依次从上往下）使用Stack，由于stack是先进后出，故实现方式为：  
首先push一个初始节点到stack中，假定为A，  
循环这个stack，只要不为空则循环不结束，从stack中pop出第一个元素，把次元素放到一个list中，作为树的返回结果显示，获取次元素的下一级子元素，如果有则把他们都push到stack中。  
首先第一次进入循环的stack中只有A，把A元素从stack中pop出来后，第一个被放到list里，然后获取到A的一级子元素（BCD），把他们push到stack中，此时stack中有三个元素（BCD），进入第二次循环。  
这 次循环从stack中pop出第一个元素B（注：这里的BCD获取的先后顺序符合先进后出原则）把B元素从stack中pop出来后，第一个被放到 list里，然后获取到A的一级子元素（B1B2），把他们push到stack中，此时stack中有元素（B1B2CD），进入第三次循环。  
这次循环从stack中pop出的应该是B1或者B2中的一个，后面和上诉的循环一致。  
获取的结果为A B B1 B11 B2 B22 C C1 C11 C12 C2 D D1 D11  
第二种方式广度优先遍历使用list，由于list是集合，集合是先进先出，故实现方式为：  
首 先add一个初始节点到list中，假定为A，循环这个list，只要不为空，则循环不结束，从这个list中取出第一个元素即A放到result（假定 也是一个list）中，并且remove这个元素。然后获取到A的一级子元素（BCD），把他们放到list中，此时list中有三个元素（BCD），进 入第二次循环。  
这次循环从list中取出第一个元素即B然后放到result中，并且remove这个元素。把B的一级子元素（B1B2）放入result中，此时list中元素为（CDB1B2）,进入第三次循环。  
这次循环和上两次一样，取出的第一个元素是C。  
获取的结果为A B C D B1 B2 C1 C2 D1 B11 B22 C11 C12 D11

·二叉树的深度优先遍历（栈）和广度优先遍历（队列）概念：

深度优先算法：是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。  
  
当节点v的所有边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。  
  
如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。如右图所示的二叉树：  
  
A 是第一个访问的，然后顺序是 B、D，然后是 E。接着再是 C、F、G。  
  
那么，怎么样才能来保证这个访问的顺序呢？  
  
分析一下，在遍历了根结点后，就开始遍历左子树，最后才是右子树。  
  
因此可以借助堆栈的[**数据结构**](http://lib.csdn.net/base/31)，由于堆栈是后进先出的顺序，由此可以先将右子树压栈，然后再对左子树压栈，  
  
这样一来，左子树结点就存在了栈顶上，因此某结点的左子树能在它的右子树遍历之前被遍历。

广度优先算法：又叫宽度优先搜索，或横向优先搜索。  
  
是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。如右图所示的二叉树，A 是第一个访问的，然后顺序是 B、C，然后再是 D、E、F、G。  
  
那么，怎样才能来保证这个访问的顺序呢？  
  
借助队列数据结构，由于队列是先进先出的顺序，因此可以先将左子树入队，然后再将右子树入队。



### B-/+Tree索引的性能分析

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系 统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B- Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

**B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。**一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

**综上所述，用B-Tree作为索引结构效率是非常高的。**

**应该花时间学习B-树和B+树数据结构**

=============================================================================================================

1）B树

B树中每个节点包含了键值和键值对于的数据对象存放地址指针，所以成功搜索一个对象可以不用到达树的叶节点。

成功搜索包括节点内搜索和沿某一路径的搜索，成功搜索时间取决于关键码所在的层次以及节点内关键码的数量。

在 B树中查找给定关键字的方法是：首先把根结点取来，在根结点所包含的关键字K1,…,kj查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定 值的关键字，则查找成功；否则，一定可以确定要查的关键字在某个Ki或Ki+1之间，于是取Pi所指的下一层索引节点块继续查找，直到找到，或指针Pi为 空时查找失败。

2）B+树

B+树非叶节点中存放的关键码并不指示数据对象的地址指针，非也节点只是索引部分。所有的叶节点在同一层上，包含了全部关键码和相应数据对象的存放地址指 针，且叶节点按关键码从小到大顺序链接。如果实际数据对象按加入的顺序存储而不是按关键码次数存储的话，叶节点的索引必须是稠密索引，若实际数据存储按关 键码次序存放的话，叶节点索引时稀疏索引。

B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

所以 B+树有两种搜索方法：

一种是按叶节点自己拉起的链表顺序搜索。

一种是从根节点开始搜索，和B树类似，不过如果非叶节点的关键码等于给定值，搜索并不停止，而是继续沿右指针，一直查到叶节点上的关键码。所以无论搜索是否成功，都将走完树的所有层。

B+ 树中，数据对象的插入和删除仅在叶节点上进行。

这两种处理索引的数据结构的不同之处：  
a，B树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而B+树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持B+树的平衡。  
b，因为B树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B+树相比来说是一种较好的折中。  
c，B树的查询效率与键在树中的位置有关，最大时间复杂度与B+树相同(在叶结点的时候)，最小时间复杂度为1(在根结点的时候)。而B+树的时候复杂度对某建成的树是固定的。

58.红黑树的五大特点：

一般的，二叉排序树，满足以下性质，即只有满足一定性质，才称之为红黑树。

    （1）每个结点要么是红色，要么是黑色；

    （2）根结点是黑的；

    （3）每个叶节点，即空结点是黑的；

    （4）如果一个结点是红的，那么的它的两个儿子都是黑的；

（5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点；

  抓住了红黑树的五个性质，事情就好办多了。

    如：

    （1）红黑红黑，要么是红，要么是黑；

    （2）根结点是黑；

    （3）每个叶结点是黑色的；

    （4）一个红结点，它的两个儿子必须都是黑的；

    （5）每一条路径上，黑结点的数目等同；

60.排序：快速排序（找第几大或第几小），堆排序（前多少后多少） 及使用场合

·http://www.codeceo.com/article/8-java-sort.html

63.完全二叉树的几大特点、b+树、b-树：

**一般二叉树性质：**

1. 在非空二叉树的k层上，至多有2k个节点(k>=0)
2. 高度为k的二叉树中,最多有2k+1-1个节点(k>=0)
3. 对于任何一棵非空的二叉树,如果叶节点个数为n0，度数为2的节点个数为n2，则有: n0 = n2 + 1

**完全二叉树性质:**

1. 具有n个节点的完全二叉树的高度k为[log2n]
2. 对于具有n个节点的完全二叉树,如果按照从上(根节点)到下(叶节点)和从左到右的顺序对二叉树中的所有节点从0开始到n-1进行编号,则对于任意的下标为k的节点，有：

* 如果k=0,则它是根节点，它没有父节点；如果k>0,则它的父节点的下标为[(i-1)/2];
* 如果2k+1 <= n-1,则下标为k的节点的左子结点的下标为2k+1;否则,下标为k的节点没有左子结点.
* 如果2k+2 <= n-1,则下标为k的节点的右子节点的下标为2k+2;否则,下标为k的节点没有右子节点

**满二叉树性质:**

在满二叉树中，叶节点的个数比分支节点的个数多1

**扩充二叉树性质:**

1. 在扩充二叉树中，外部节点的个数比内部节点的个数多1
2. 对任意扩充二叉树，外部路径长度E和内部路径长度I之间满足以下关系：E = I + 2n, 其中n是内部节点个数

**二叉树的实现与周游**

**二叉树的顺序表示：**

二叉树的顺序表示适用于完全二叉树，根据完全二叉树的第2性质，可以建立处节点之间的关系，对于一般的二叉树，可以将其扩充为完全二叉树，然后用一维数组顺序存储.

以下是程序代码:

bintree.h



1 #include<stdio.h>

2 #include<stdlib.h>

3 #define USED 1

4 #define NOTUSED 0

5 typedef int Type;

6

7 // 二叉树节点结构体

8 typedef struct treenode

9 {

10 int nodeIndex;

11 // 标志是否为外部节点，如果是则为0，否则为1

12 int isIn;

13 // 0表示这个节点不在二叉树中，1表示在树中

14 int isUsed;

15 Type element;

16 }TreeNode;

17

18 // 二叉树结构体

19 typedef struct bintree

20 {

21 int MAXN;

22 int n;

23 TreeNode \*nodelist;

24 }\*Tree;

25

26 // 创建最大节点为m的空二叉树

27 Tree createBinTree(int m)

28 {

29 int i;

30 Tree tree = NULL;

31 TreeNode \*nodelist = NULL;

32 tree = malloc(sizeof(struct bintree));

33 if (NULL != tree)

34 {

35 nodelist = malloc(m \* sizeof(TreeNode));

36 if (NULL != nodelist)

37 {

38 tree->MAXN = m;

39 tree->n = -1;

40 for (i = 0; i < m; i++)

41 {

42 nodelist[i].isUsed = NOTUSED;

43 nodelist[i].nodeIndex = i;

44 }

45 printf("\n");

46 tree->nodelist = nodelist;

47 }

48 }

49 return tree;

50 }

51

52 // 在二叉树末尾加入元素

53 void inTree(Tree tree, Type x, int isIn)

54 {

55 void replaceTree(Tree, Type, int, int);

56 if (tree->n + 1 < tree->MAXN)

57 {

58 replaceTree(tree, x, tree->n + 1, isIn);

59 tree->n = tree->n + 1;

60 }

61 return;

62 }

63

64 // 替换二叉树中指定下标的元素

65 void replaceTree(Tree tree, Type x, int index, int isIn)

66 {

67 if (tree->n == -1 || (index >= 0 && index <= tree->n + 1))

68 {

69 tree->nodelist[index].element = x;

70 tree->nodelist[index].isIn = isIn;

71 tree->nodelist[index].isUsed = USED;

72 }

73 return;

74 }

75

76 // 得到指定下标的元素

77 Type getNode(Tree tree, int index)

78 {

79 int re;

80 if (index >= 0 && index <= tree->n)

81 re = tree->nodelist[index].element;

82 return re;

83 }

84

85 // 得到父节点的元素下标

86 int parent(Tree tree, int child)

87 {

88 if (child < 0 || child > tree->n)

89 return -1;

90 else

91 return (child - 1) / 2;

92 }

93

94 // 得到右子节点的元素下标

95 int rightChild(Tree tree, int parent)

96 {

97 if (parent < 0 || parent > tree->n)

98 return -1;

99 else

100 return 2 \* parent + 2;

101 }

102

103 // 得到左子节点的元素下标

104 int leftChild(Tree tree, int parent)

105 {

106 if (parent < 0 || parent > tree->n)

107 return -1;

108 else

109 return 2 \* parent + 1;

110 }



这里用的是非递归的周游方法，所以要用到栈，一下是自定义的栈.

stack.h



1 #include "bintree.h"

2 typedef TreeNode DataType;

3

4 typedef struct mystack

5 {

6 //数组最大长度

7 int MAXN;

8 //指定栈顶位置

9 int index;

10 DataType \*ele;

11

12 }\*MyStack, Stack;

13

14

15 //创建一个空栈

16 MyStack createStack(int num)

17 {

18 MyStack stack = NULL;

19 stack = malloc(sizeof(Stack));

20 if(stack != NULL)

21 {

22 stack->ele = malloc(sizeof(DataType) \* num);

23 if(stack ->ele != NULL)

24 {

25 stack ->MAXN = num;

26 stack ->index = -1;

27 }

28 }

29 return stack;

30 }

31

32 //判断栈是否为空

33 int isEmptyStack(MyStack stack)

34 {

35 return stack ->index == -1;

36 }

37

38 //元素入栈,如果栈已经满则返回0,否则返回1

39 int push(MyStack stack, DataType x)

40 {

41 int index = stack ->index + 1;

42 if(index >= stack ->MAXN)

43 return 0;

44 else

45 {

46 stack ->ele[index] = x;

47 stack ->index = index;

48 return 1;

49 }

50 }

51

52 //元素出栈,如果栈已经为空返回０，否则返回１

53 int pop(MyStack stack)

54 {

55 if(isEmptyStack(stack))

56 return 0;

57 else

58 {

59 stack ->index--;

60 return 1;

61 }

62 }

63

64 //取栈顶元素

65 DataType top(MyStack stack)

66 {

67 DataType get;

68 if(!isEmptyStack(stack))

69 get = stack ->ele[stack ->index];

70 return get;

71 }



bintree.c



1 #include "stack.h"

2

3 //访问节点信息

4 int visit(TreeNode node)

5 {

6 printf(" %d", node.element);

7 return node.nodeIndex;

8 }

9

10 //先根次序周游

11 void preOrder(Tree tree, MyStack stack)

12 {

13 int index = 0;

14 TreeNode node;

15 if(!tree->nodelist[index].isUsed)

16 return;

17 push(stack,tree->nodelist[index]);

18 while(!isEmptyStack(stack))

19 {

20 node = top(stack);

21 pop(stack);

22 if(node.isUsed == USED)

23 {

24 index = visit(node);

25 push(stack,tree->nodelist[rightChild(tree, index)]);

26 push(stack,tree->nodelist[leftChild(tree, index)]);

27 }

28

29 }

30 }

31

32 //中根次序周游

33 void inOrder(Tree tree, MyStack stack)

34 {

35 int index = 0;

36 TreeNode node;

37 if(tree->nodelist[index].isUsed == NOTUSED)

38 return;

39 do

40 {

41 while(tree->nodelist[index].isUsed == USED)

42 {

43 push(stack,tree->nodelist[index]);

44 index = leftChild(tree, index);

45 }

46 node = top(stack);

47 pop(stack);

48 index = visit(node);

49 index = rightChild(tree,index);

50 } while(tree->nodelist[index].isUsed == USED || !isEmptyStack(stack));

51

52 }

53

54 //后根次序周游

55 void postOrder(Tree tree, MyStack stack)

56 {

57 int index = 0, rightIndex;

58 TreeNode node;

59 if(tree->nodelist[index].isUsed == NOTUSED)

60 return;

61 while(tree->nodelist[index].isUsed == USED || !isEmptyStack(stack))

62 {

63 while(tree->nodelist[index].isUsed == USED)

64 {

65 push(stack,tree->nodelist[index]);

66 rightIndex = rightChild(tree, index);

67 index = leftChild(tree ,index);

68 if(tree->nodelist[index].isUsed == NOTUSED)

69 index = rightIndex;

70 }

71 node = top(stack);

72 pop(stack);

73 index = visit(node);

74 //如果栈不为空，且从左子树退回,进入到右子树遍历

75 if(!isEmptyStack(stack) && index == leftChild(tree,top(stack).nodeIndex))

76 index = rightChild(tree,top(stack).nodeIndex);

77 else tree->nodelist[index].isUsed = NOTUSED;

78 }

79

80 }

81

82 int main()

83 {

84 int m = 0, isIn = 1, j, buffer = 0;

85 Tree tree = NULL;

86 MyStack stack = NULL;

87

88 printf("请输入节点个数:");

89 scanf("%d",&m);

90 tree = createBinTree(m);

91 stack = createStack(m);

92 printf("请输入%d个数字:",m);

93 //将0到5六个元素依次放入二叉树中

94 for(j = 0; j < m; j++)

95 {

96 scanf(" %d", &buffer);

97 inTree(tree, buffer, isIn);

98 }

99

100 printf("先根次序周游结果为:");

101 preOrder(tree,stack);

102 printf("\n");

103

104 printf("中根次序周游结果为:");

105 inOrder(tree, stack);

106 printf("\n");

107

108 printf("后根次序周游结果为:");

109 postOrder(tree, stack);

110 printf("\n");

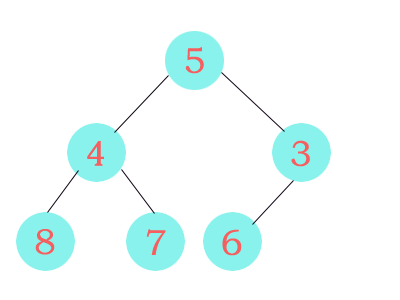
111

112 return 1;

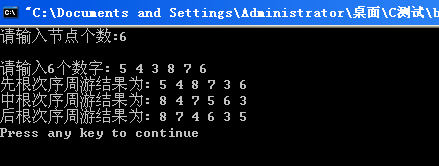
113 }



按图示二叉树输出:



输出结果为：



**二叉树连接表示:**



1 #include<stdio.h>

2 #include<stdlib.h>

3 typedef int DataType;

4 typedef struct treenode \*TreeNode;

5 typedef struct treenode \*BinTree;

6

7 struct treenode

8 {

9 DataType element;

10 TreeNode llink;

11 TreeNode rlink;

12 };

13

14 BinTree createBinTree(DataType x)

15 {

16 BinTree tree = NULL;

17 tree = malloc(sizeof(struct treenode));

18 tree->element = x;

19 return tree;

20 }

21

22 BinTree addToLeft(BinTree t, DataType x)

23 {

24 TreeNode node = NULL;

25 node = malloc(sizeof(struct treenode));

26 if (node != NULL)

27 node->element = x;

28 t -> llink = node;

29 return node;

30 }

31

32 BinTree addToRight(BinTree t, DataType x)

33 {

34 TreeNode node = NULL;

35 node = malloc(sizeof(struct treenode));

36 if (node != NULL)

37 node->element = x;

38 t-> rlink = node;

39 return node;

40 }

41

42 void visitRoot(BinTree tree)

43 {

44 printf("%d ", tree->element);

45 }

46

47 BinTree leftChild(BinTree tree)

48 {

49 return tree->llink;

50 }

51

52 BinTree rightChild(BinTree tree)

53 {

54 return tree->rlink;

55 }

56 //用递归方法

57 //先根周游

58 void preOrder(BinTree tree)

59 {

60 if(tree == NULL)

61 return;

62 visitRoot(tree);

63 preOrder(leftChild(tree));

64 preOrder(rightChild(tree));

65 }

66

67 //中根周游

68 void inOrder(BinTree tree)

69 {

70 if(NULL == tree)

71 return;

72 inOrder(leftChild(tree));

73 visitRoot(tree);

74 inOrder(rightChild(tree));

75 }

76

77 //后根周游

78 void postOrder(BinTree tree)

79 {

80 if(NULL == tree)

81 return;

82 postOrder(leftChild(tree));

83 postOrder(rightChild(tree));

84 visitRoot(tree);

85 }

86

87 int main()

88 {

89 BinTree left, right;

90 BinTree tree = createBinTree(5);

91 left = addToLeft(tree, 4);

92 right = addToRight(tree, 3);

93 addToLeft(left, 8);

94 addToRight(left, 7);

95 addToLeft(right, 6);

96

97 printf("先根周游次序：");

98 preOrder(tree);

99 printf("\n");

100 printf("中根周游次序：");

101 inOrder(tree);

102 printf("\n");

103 printf("后根周游算法：");

104 postOrder(tree);

105 printf("\n");

106 return 1;

107 }

二叉树是一种非常重要的数据结构，它同时具有数组和链表各自的特点：它可以像数组一样快速查找，也可以像链表一样快速添加。但是他也有自己的缺点：删除操作复杂。

    我们先介绍一些关于二叉树的概念名词。

    二叉树：是每个结点最多有两个子树的有序树，在使用二叉树的时候，数据并不是随便插入到节点中的，一个节点的左子节点的关键值必须小于此节点，右子节点的关键值必须大于或者是等于此节点，所以又称二叉查找树、二叉排序树、二叉搜索树。

    完全二叉树：若设二叉树的高度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第h层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉树。

    满二叉树——除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树。

    深度——二叉树的层数，就是深度。

    二叉树的特点总结：

（1）树执行查找、删除、插入的时间复杂度都是O(logN)

（2）遍历二叉树的方法包括前序、中序、后序

（3）非平衡树指的是根的左右两边的子节点的数量不一致

（4） 在非空二叉树中，第i层的结点总数不超过 , i>=1；  
（5）深度为h的二叉树最多有个结点(h>=1)，最少有h个结点；  
（6）对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1；

    二叉树的Java代码实现

    首先是树的节点的定义，下面的代码中使用的是最简单的int基本数据类型作为节点的数据，如果要使用节点带有更加复杂的数据类型，换成对应的对象即可。

**[java]** [view plaincopy](http://blog.csdn.net/zhaokaiqiang1992/article/details/41750137)



1. /\*\*
2. \*
3. \* @ClassName: com.tree.TreeNode
4. \* @Description: 节点
5. \* @author zhaokaiqiang
6. \* @date 2014-12-5 下午4:43:24
7. \*
8. \*/
9. **public** **class** TreeNode {
11. // 左节点
12. **private** TreeNode lefTreeNode;
13. // 右节点
14. **private** TreeNode rightNode;
15. // 数据
16. **private** **int** value;
18. **private** **boolean** isDelete;
20. **public** TreeNode getLefTreeNode() {
21. **return** lefTreeNode;
22. }
24. **public** **void** setLefTreeNode(TreeNode lefTreeNode) {
25. **this**.lefTreeNode = lefTreeNode;
26. }
28. **public** TreeNode getRightNode() {
29. **return** rightNode;
30. }
32. **public** **void** setRightNode(TreeNode rightNode) {
33. **this**.rightNode = rightNode;
34. }
36. **public** **int** getValue() {
37. **return** value;
38. }
40. **public** **void** setValue(**int** value) {
41. **this**.value = value;
42. }
44. **public** **boolean** isDelete() {
45. **return** isDelete;
46. }
48. **public** **void** setDelete(**boolean** isDelete) {
49. **this**.isDelete = isDelete;
50. }
52. **public** TreeNode() {
53. **super**();
54. }
56. **public** TreeNode(**int** value) {
57. **this**(**null**, **null**, value, **false**);
58. }
60. **public** TreeNode(TreeNode lefTreeNode, TreeNode rightNode, **int** value,
61. **boolean** isDelete) {
62. **super**();
63. **this**.lefTreeNode = lefTreeNode;
64. **this**.rightNode = rightNode;
65. **this**.value = value;
66. **this**.isDelete = isDelete;
67. }
69. @Override
70. **public** String toString() {
71. **return** "TreeNode [lefTreeNode=" + lefTreeNode + ", rightNode="
72. + rightNode + ", value=" + value + ", isDelete=" + isDelete
73. + "]";
74. }
76. }

    下面给出二叉树的代码实现。由于在二叉树中进行节点的删除非常繁琐，因此，下面的代码使用的是利用节点的isDelete字段对节点的状态进行标识

**[java]** [view plaincopy](http://blog.csdn.net/zhaokaiqiang1992/article/details/41750137)



1. /\*\*
2. \*
3. \* @ClassName: com.tree.Tree
4. \* @Description: 二叉树的定义
5. \* @author zhaokaiqiang
6. \* @date 2014-12-8 上午7:51:49
7. \*
8. \*/
10. **public** **class** BinaryTree {
12. // 根节点
13. **private** TreeNode root;
15. **public** TreeNode getRoot() {
16. **return** root;
17. }
19. /\*\*
20. \* 插入操作
21. \*
22. \* @param value
23. \*/
24. **public** **void** insert(**int** value) {
26. TreeNode newNode = **new** TreeNode(value);
28. **if** (root == **null**) {
29. root = newNode;
30. root.setLefTreeNode(**null**);
31. root.setRightNode(**null**);
32. } **else** {
34. TreeNode currentNode = root;
35. TreeNode parentNode;
37. **while** (**true**) {
39. parentNode = currentNode;
40. // 往右放
41. **if** (newNode.getValue() > currentNode.getValue()) {
42. currentNode = currentNode.getRightNode();
43. **if** (currentNode == **null**) {
44. parentNode.setRightNode(newNode);
45. **return**;
46. }
47. } **else** {
48. // 往左放
49. currentNode = currentNode.getLefTreeNode();
50. **if** (currentNode == **null**) {
51. parentNode.setLefTreeNode(newNode);
52. **return**;
53. }
55. }
56. }
57. }
58. }
60. /\*\*
61. \* 查找
62. \*
63. \* @param key
64. \* @return
65. \*/
66. **public** TreeNode find(**int** key) {
68. TreeNode currentNode = root;
70. **if** (currentNode != **null**) {
72. **while** (currentNode.getValue() != key) {
74. **if** (currentNode.getValue() > key) {
75. currentNode = currentNode.getLefTreeNode();
76. } **else** {
77. currentNode = currentNode.getRightNode();
78. }
80. **if** (currentNode == **null**) {
81. **return** **null**;
82. }
84. }
86. **if** (currentNode.isDelete()) {
87. **return** **null**;
88. } **else** {
89. **return** currentNode;
90. }
92. } **else** {
93. **return** **null**;
94. }
96. }
98. /\*\*
99. \* 中序遍历
100. \*
101. \* @param treeNode
102. \*/
103. **public** **void** inOrder(TreeNode treeNode) {
104. **if** (treeNode != **null** && treeNode.isDelete() == **false**) {
105. inOrder(treeNode.getLefTreeNode());
106. System.out.println("--" + treeNode.getValue());
107. inOrder(treeNode.getRightNode());
108. }
109. }
111. }

    在上面对二叉树的遍历操作中，使用的是中序遍历，这样遍历出来的数据是增序的。

    下面是测试代码:

**[java]** [view plaincopy](http://blog.csdn.net/zhaokaiqiang1992/article/details/41750137)



1. **public** **class** Main {
3. **public** **static** **void** main(String[] args) {
5. BinaryTree tree = **new** BinaryTree();
6. // 添加数据测试
7. tree.insert(10);
8. tree.insert(40);
9. tree.insert(20);
10. tree.insert(3);
11. tree.insert(49);
12. tree.insert(13);
13. tree.insert(123);
15. System.out.println("root=" + tree.getRoot().getValue());
16. // 排序测试
17. tree.inOrder(tree.getRoot());
18. // 查找测试
19. **if** (tree.find(10) != **null**) {
20. System.out.println("找到了");
21. } **else** {
22. System.out.println("没找到");
23. }
24. // 删除测试
25. tree.find(40).setDelete(**true**);
27. **if** (tree.find(40) != **null**) {
28. System.out.println("找到了");
29. } **else** {
30. System.out.println("没找到");
31. }
33. }
35. }

import java.util.ArrayList;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

import java.util.Queue;

import java.util.Stack;

/\*\*

\* http://blog.csdn.net/luckyxiaoqiang/article/details/7518888 轻松搞定面试中的二叉树题目

\* http://www.cnblogs.com/Jax/archive/2009/12/28/1633691.html 算法大全（3） 二叉树

\*

\* TODO: 一定要能熟练地写出所有问题的递归和非递归做法！

\*

\* 1. 求二叉树中的节点个数: getNodeNumRec（递归），getNodeNum（迭代）

\* 2. 求二叉树的深度: getDepthRec（递归），getDepth

\* 3. 前序遍历，中序遍历，后序遍历: preorderTraversalRec, preorderTraversal, inorderTraversalRec, postorderTraversalRec

\* (https://en.wikipedia.org/wiki/Tree\_traversal#Pre-order\_2)

\* 4.分层遍历二叉树（按层次从上往下，从左往右）: levelTraversal, levelTraversalRec（递归解法！）

\* 5. 将二叉查找树变为有序的双向链表: convertBST2DLLRec, convertBST2DLL

\* 6. 求二叉树第K层的节点个数：getNodeNumKthLevelRec, getNodeNumKthLevel

\* 7. 求二叉树中叶子节点的个数：getNodeNumLeafRec, getNodeNumLeaf

\* 8. 判断两棵二叉树是否相同的树：isSameRec, isSame

\* 9. 判断二叉树是不是平衡二叉树：isAVLRec

\* 10. 求二叉树的镜像（破坏和不破坏原来的树两种情况）：mirrorRec, mirrorCopyRec

\* 10.1 判断两个树是否互相镜像：isMirrorRec

\* 11. 求二叉树中两个节点的最低公共祖先节点：getLastCommonParent, getLastCommonParentRec, getLastCommonParentRec2

\* 12. 求二叉树中节点的最大距离：getMaxDistanceRec

\* 13. 由前序遍历序列和中序遍历序列重建二叉树：rebuildBinaryTreeRec

\* 14.判断二叉树是不是完全二叉树：isCompleteBinaryTree, isCompleteBinaryTreeRec

\*

\*/

public class Demo {

/\*

1

/ \

2 3

/ \ \

4 5 6

\*/

public static void main(String[] args) {

TreeNode r1 = new TreeNode(1);

TreeNode r2 = new TreeNode(2);

TreeNode r3 = new TreeNode(3);

TreeNode r4 = new TreeNode(4);

TreeNode r5 = new TreeNode(5);

TreeNode r6 = new TreeNode(6);

r1.left = r2;

r1.right = r3;

r2.left = r4;

r2.right = r5;

r3.right = r6;

// System.out.println(getNodeNumRec(r1));

// System.out.println(getNodeNum(r1));

// System.out.println(getDepthRec(r1));

// System.out.println(getDepth(r1));

// preorderTraversalRec(r1);

// System.out.println();

// preorderTraversal(r1);

// System.out.println();

// inorderTraversalRec(r1);

// System.out.println();

// inorderTraversal(r1);

// System.out.println();

// postorderTraversalRec(r1);

// System.out.println();

// postorderTraversal(r1);

// System.out.println();

// levelTraversal(r1);

// System.out.println();

// levelTraversalRec(r1);

// System.out.println();

// TreeNode tmp = convertBSTRec(r1);

// while(true){

// if(tmp == null){

// break;

// }

// System.out.print(tmp.val + " ");

// if(tmp.right == null){

// break;

// }

// tmp = tmp.right;

// }

// System.out.println();

// while(true){

// if(tmp == null){

// break;

// }

// System.out.print(tmp.val + " ");

// if(tmp.left == null){

// break;

// }

// tmp = tmp.left;

// }

// TreeNode tmp = convertBST2DLL(r1);

// while(true){

// if(tmp == null){

// break;

// }

// System.out.print(tmp.val + " ");

// if(tmp.right == null){

// break;

// }

// tmp = tmp.right;

// }

// System.out.println(getNodeNumKthLevelRec(r1, 2));

// System.out.println(getNodeNumKthLevel(r1, 2));

// System.out.println(getNodeNumLeafRec(r1));

// System.out.println(getNodeNumLeaf(r1));

// System.out.println(isSame(r1, r1));

// inorderTraversal(r1);

// System.out.println();

// mirror(r1);

// TreeNode mirrorRoot = mirrorCopy(r1);

// inorderTraversal(mirrorRoot);

System.out.println(isCompleteBinaryTree(r1));

System.out.println(isCompleteBinaryTreeRec(r1));

}

private static class TreeNode {

int val;

TreeNode left;

TreeNode right;

public TreeNode(int val) {

this.val = val;

}

}

/\*\*

\* 求二叉树中的节点个数递归解法： O(n)

\* （1）如果二叉树为空，节点个数为0

\* （2）如果二叉树不为空，二叉树节点个数 = 左子树节点个数 +

\* 右子树节点个数 + 1

\*/

public static int getNodeNumRec(TreeNode root) {

if (root == null) {

return 0;

} else {

return getNodeNumRec(root.left) + getNodeNumRec(root.right) + 1;

}

}

/\*\*

\* 求二叉树中的节点个数迭代解法O(n)：基本思想同LevelOrderTraversal，

\* 即用一个Queue，在[Java](http://www.2cto.com/kf/ware/Java/)里面可以用LinkedList来模拟

\*/

public static int getNodeNum(TreeNode root) {

if(root == null){

return 0;

}

int count = 1;

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

while(!queue.isEmpty()){

TreeNode cur = queue.remove(); // 从队头位置移除

if(cur.left != null){ // 如果有左孩子，加到队尾

queue.add(cur.left);

count++;

}

if(cur.right != null){ // 如果有右孩子，加到队尾

queue.add(cur.right);

count++;

}

}

return count;

}

/\*\*

\* 求二叉树的深度（高度） 递归解法： O(n)

\* （1）如果二叉树为空，二叉树的深度为0

\* （2）如果二叉树不为空，二叉树的深度 = max(左子树深度， 右子树深度) + 1

\*/

public static int getDepthRec(TreeNode root) {

if (root == null) {

return 0;

}

int leftDepth = getDepthRec(root.left);

int rightDepth = getDepthRec(root.right);

return Math.max(leftDepth, rightDepth) + 1;

}

/\*\*

\* 求二叉树的深度（高度） 迭代解法： O(n)

\* 基本思想同LevelOrderTraversal，还是用一个Queue

\*/

public static int getDepth(TreeNode root) {

if(root == null){

return 0;

}

int depth = 0; // 深度

int currentLevelNodes = 1; // 当前Level，node的数量

int nextLevelNodes = 0; // 下一层Level，node的数量

LinkedList<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

while( !queue.isEmpty() ){

TreeNode cur = queue.remove(); // 从队头位置移除

currentLevelNodes--; // 减少当前Level node的数量

if(cur.left != null){ // 如果有左孩子，加到队尾

queue.add(cur.left);

nextLevelNodes++; // 并增加下一层Level node的数量

}

if(cur.right != null){ // 如果有右孩子，加到队尾

queue.add(cur.right);

nextLevelNodes++;

}

if(currentLevelNodes == 0){ // 说明已经遍历完当前层的所有节点

depth++; // 增加高度

currentLevelNodes = nextLevelNodes; // 初始化下一层的遍历

nextLevelNodes = 0;

}

}

return depth;

}

/\*\*

\* 前序遍历，中序遍历，后序遍历 前序遍历递归解法：

\* （1）如果二叉树为空，空操作

\* （2）如果二叉树不为空，访问根节点，前序遍历左子树，前序遍历右子树

\*/

public static void preorderTraversalRec(TreeNode root) {

if (root == null) {

return;

}

System.out.print(root.val + " ");

preorderTraversalRec(root.left);

preorderTraversalRec(root.right);

}

/\*\*

\* 前序遍历迭代解法：用一个辅助stack，总是把右孩子放进栈

\* http://www.youtube.com/watch?v=uPTCbdHSFg4

\*/

public static void preorderTraversal(TreeNode root) {

if(root == null){

return;

}

Stack<TreeNode> stack = new Stack<TreeNode>(); // 辅助stack

stack.push(root);

while( !stack.isEmpty() ){

TreeNode cur = stack.pop(); // 出栈栈顶元素

System.out.print(cur.val + " ");

// 关键点：要先压入右孩子，再压入左孩子，这样在出栈时会先打印左孩子再打印右孩子

if(cur.right != null){

stack.push(cur.right);

}

if(cur.left != null){

stack.push(cur.left);

}

}

}

/\*\*

\* 中序遍历递归解法

\* （1）如果二叉树为空，空操作。

\* （2）如果二叉树不为空，中序遍历左子树，访问根节点，中序遍历右子树

\*/

public static void inorderTraversalRec(TreeNode root) {

if (root == null) {

return;

}

inorderTraversalRec(root.left);

System.out.print(root.val + " ");

inorderTraversalRec(root.right);

}

/\*\*

\* 中序遍历迭代解法 ，用栈先把根节点的所有左孩子都添加到栈内，

\* 然后输出栈顶元素，再处理栈顶元素的右子树

\* http://www.youtube.com/watch?v=50v1sJkjxoc

\*

\* 还有一种方法能不用递归和栈，基于线索二叉树的方法，较麻烦以后补上

\* http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/

\*/

public static void inorderTraversal(TreeNode root){

if(root == null){

return;

}

Stack<TreeNode> stack = new Stack<TreeNode>();

TreeNode cur = root;

while( true ){

while(cur != null){ // 先添加一个非空节点所有的左孩子到栈

stack.push(cur);

cur = cur.left;

}

if(stack.isEmpty()){

break;

}

// 因为此时已经没有左孩子了，所以输出栈顶元素

cur = stack.pop();

System.out.print(cur.val + " ");

cur = cur.right; // 准备处理右子树

}

}

/\*\*

\* 后序遍历递归解法

\* （1）如果二叉树为空，空操作

\* （2）如果二叉树不为空，后序遍历左子树，后序遍历右子树，访问根节点

\*/

public static void postorderTraversalRec(TreeNode root) {

if (root == null) {

return;

}

postorderTraversalRec(root.left);

postorderTraversalRec(root.right);

System.out.print(root.val + " ");

}

/\*\*

\* 后序遍历迭代解法

\* http://www.youtube.com/watch?v=hv-mJUs5mvU

\*

\*/

public static void postorderTraversal(TreeNode root) {

if (root == null) {

return;

}

Stack<TreeNode> s = new Stack<TreeNode>(); // 第一个stack用于添加node和它的左右孩子

Stack<TreeNode> output = new Stack<TreeNode>();// 第二个stack用于翻转第一个stack输出

s.push(root);

while( !s.isEmpty() ){ // 确保所有元素都被翻转转移到第二个stack

TreeNode cur = s.pop(); // 把栈顶元素添加到第二个stack

output.push(cur);

if(cur.left != null){ // 把栈顶元素的左孩子和右孩子分别添加入第一个stack

s.push(cur.left);

}

if(cur.right != null){

s.push(cur.right);

}

}

while( !output.isEmpty() ){ // 遍历输出第二个stack，即为后序遍历

System.out.print(output.pop().val + " ");

}

}

/\*\*

\* 分层遍历二叉树（按层次从上往下，从左往右）迭代

\* 相当于广度优先搜索，使用队列实现。队列初始化，将根节点压入队列。当队列不为空，进行如下操作：弹出一个节点

\* ，访问，若左子节点或右子节点不为空，将其压入队列

\*/

public static void levelTraversal(TreeNode root) {

if (root == null) {

return;

}

LinkedList<TreeNode> queue = new LinkedList<TreeNode>();

queue.push(root);

while (!queue.isEmpty()) {

TreeNode cur = queue.removeFirst();

System.out.print(cur.val + " ");

if (cur.left != null) {

queue.add(cur.left);

}

if (cur.right != null) {

queue.add(cur.right);

}

}

}

/\*\*

\* 分层遍历二叉树（递归）

\* 很少有人会用递归去做level traversal

\* 基本思想是用一个大的ArrayList，里面包含了每一层的ArrayList。

\* 大的ArrayList的size和level有关系

\*

\* 这是我目前见到的最好的递归解法！

\* http://discuss.leetcode.com/questions/49/binary-tree-level-order-traversal#answer-container-2543

\*/

public static void levelTraversalRec(TreeNode root) {

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();

dfs(root, 0, ret);

System.out.println(ret);

}

private static void dfs(TreeNode root, int level, ArrayList<ArrayList<Integer>> ret){

if(root == null){

return;

}

// 添加一个新的ArrayList表示新的一层

if(level >= ret.size()){

ret.add(new ArrayList<Integer>());

}

ret.get(level).add(root.val); // 把节点添加到表示那一层的ArrayList里

dfs(root.left, level+1, ret); // 递归处理下一层的左子树和右子树

dfs(root.right, level+1, ret);

}

/\*\*

\* 将二叉查找树变为有序的双向链表 要求不能创建新节点，只调整指针。

\* 递归解法：

\* 参考了http://stackoverflow.com/questions/11511898/converting-a-binary-search-tree-to-doubly-linked-list#answer-11530016

\* 感觉是最清晰的递归解法，但要注意递归完，root会在链表的中间位置，因此要手动

\* 把root移到链表头或链表尾

\*/

public static TreeNode convertBST2DLLRec(TreeNode root) {

root = convertBST2DLLSubRec(root);

// root会在链表的中间位置，因此要手动把root移到链表头

while(root.left != null){

root = root.left;

}

return root;

}

/\*\*

\* 递归转换BST为双向链表(DLL)

\*/

public static TreeNode convertBST2DLLSubRec(TreeNode root){

if(root==null || (root.left==null && root.right==null)){

return root;

}

TreeNode tmp = null;

if(root.left != null){ // 处理左子树

tmp = convertBST2DLLSubRec(root.left);

while(tmp.right != null){ // 寻找最右节点

tmp = tmp.right;

}

tmp.right = root; // 把左子树处理后结果和root连接

root.left = tmp;

}

if(root.right != null){ // 处理右子树

tmp = convertBST2DLLSubRec(root.right);

while(tmp.left != null){ // 寻找最左节点

tmp = tmp.left;

}

tmp.left = root; // 把右子树处理后结果和root连接

root.right = tmp;

}

return root;

}

/\*\*

\* 将二叉查找树变为有序的双向链表 迭代解法

// \* 类似inorder traversal的做法

\*/

public static TreeNode convertBST2DLL(TreeNode root) {

if(root == null){

return null;

}

Stack<TreeNode> stack = new Stack<TreeNode>();

TreeNode cur = root; // 指向当前处理节点

TreeNode old = null; // 指向前一个处理的节点

TreeNode head = null; // 链表头

while( true ){

while(cur != null){ // 先添加一个非空节点所有的左孩子到栈

stack.push(cur);

cur = cur.left;

}

if(stack.isEmpty()){

break;

}

// 因为此时已经没有左孩子了，所以输出栈顶元素

cur = stack.pop();

if(old != null){

old.right = cur;

}

if(head == null){ // /第一个节点为双向链表头节点

head = cur;

}

old = cur; // 更新old

cur = cur.right; // 准备处理右子树

}

return head;

}

/\*\*

\* 求二叉树第K层的节点个数 递归解法：

\* （1）如果二叉树为空或者k<1返回0

\* （2）如果二叉树不为空并且k==1，返回1

\* （3）如果二叉树不为空且k>1，返回root左子树中k-1层的节点个数与root右子树k-1层节点个数之和

\*

\* 求以root为根的k层节点数目 等价于 求以root左孩子为根的k-1层（因为少了root那一层）节点数目 加上

\* 以root右孩子为根的k-1层（因为少了root那一层）节点数目

\*

\* 所以遇到树，先把它拆成左子树和右子树，把问题降解

\*

\*/

public static int getNodeNumKthLevelRec(TreeNode root, int k) {

if (root == null || k < 1) {

return 0;

}

if (k == 1) {

return 1;

}

int numLeft = getNodeNumKthLevelRec(root.left, k - 1); // 求root左子树的k-1层节点数

int numRight = getNodeNumKthLevelRec(root.right, k - 1); // 求root右子树的k-1层节点数

return numLeft + numRight;

}

/\*\*

\* 求二叉树第K层的节点个数 迭代解法：

\* 同getDepth的迭代解法

\*/

public static int getNodeNumKthLevel(TreeNode root, int k){

if(root == null){

return 0;

}

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

int i = 1;

int currentLevelNodes = 1; // 当前Level，node的数量

int nextLevelNodes = 0; // 下一层Level，node的数量

while( !queue.isEmpty() && i<k){

TreeNode cur = queue.remove(); // 从队头位置移除

currentLevelNodes--; // 减少当前Level node的数量

if(cur.left != null){ // 如果有左孩子，加到队尾

queue.add(cur.left);

nextLevelNodes++; // 并增加下一层Level node的数量

}

if(cur.right != null){ // 如果有右孩子，加到队尾

queue.add(cur.right);

nextLevelNodes++;

}

if(currentLevelNodes == 0){ // 说明已经遍历完当前层的所有节点

currentLevelNodes = nextLevelNodes; // 初始化下一层的遍历

nextLevelNodes = 0;

i++; // 进入到下一层

}

}

return currentLevelNodes;

}

/\*\*

\* 求二叉树中叶子节点的个数（递归）

\*/

public static int getNodeNumLeafRec(TreeNode root) {

// 当root不存在，返回空

if (root == null) {

return 0;

}

// 当为叶子节点时返回1

if (root.left == null && root.right == null) {

return 1;

}

// 把一个树拆成左子树和右子树之和，原理同上一题

return getNodeNumLeafRec(root.left) + getNodeNumLeafRec(root.right);

}

/\*\*

\* 求二叉树中叶子节点的个数（迭代）

\* 还是基于Level order traversal

\*/

public static int getNodeNumLeaf(TreeNode root) {

if(root == null){

return 0;

}

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

int leafNodes = 0; // 记录上一个Level，node的数量

while( !queue.isEmpty() ){

TreeNode cur = queue.remove(); // 从队头位置移除

if(cur.left != null){ // 如果有左孩子，加到队尾

queue.add(cur.left);

}

if(cur.right != null){ // 如果有右孩子，加到队尾

queue.add(cur.right);

}

if(cur.left==null && cur.right==null){ // 叶子节点

leafNodes++;

}

}

return leafNodes;

}

/\*\*

\* 判断两棵二叉树是否相同的树。

\* 递归解法：

\* （1）如果两棵二叉树都为空，返回真

\* （2）如果两棵二叉树一棵为空，另一棵不为空，返回假

\* （3）如果两棵二叉树都不为空，如果对应的左子树和右子树都同构返回真，其他返回假

\*/

public static boolean isSameRec(TreeNode r1, TreeNode r2) {

// 如果两棵二叉树都为空，返回真

if (r1 == null && r2 == null) {

return true;

}

// 如果两棵二叉树一棵为空，另一棵不为空，返回假

else if (r1 == null || r2 == null) {

return false;

}

if(r1.val != r2.val){

return false;

}

boolean leftRes = isSameRec(r1.left, r2.left); // 比较对应左子树

boolean rightRes = isSameRec(r1.right, r2.right); // 比较对应右子树

return leftRes && rightRes;

}

/\*\*

\* 判断两棵二叉树是否相同的树（迭代）

\* 遍历一遍即可，这里用preorder

\*/

public static boolean isSame(TreeNode r1, TreeNode r2) {

// 如果两个树都是空树，则返回true

if(r1==null && r2==null){

return true;

}

// 如果有一棵树是空树，另一颗不是，则返回false

if(r1==null || r2==null){

return false;

}

Stack<TreeNode> s1 = new Stack<TreeNode>();

Stack<TreeNode> s2 = new Stack<TreeNode>();

s1.push(r1);

s2.push(r2);

while(!s1.isEmpty() && !s2.isEmpty()){

TreeNode n1 = s1.pop();

TreeNode n2 = s2.pop();

if(n1==null && n2==null){

continue;

}else if(n1!=null && n2!=null && n1.val==n2.val){

s1.push(n1.right);

s1.push(n1.left);

s2.push(n2.right);

s2.push(n2.left);

}else{

return false;

}

}

return true;

}

/\*\*

\* 判断二叉树是不是平衡二叉树 递归解法：

\* （1）如果二叉树为空，返回真

\* （2）如果二叉树不为空，如果左子树和右子树都是AVL树并且左子树和右子树高度相差不大于1，返回真，其他返回假

\*/

public static boolean isAVLRec(TreeNode root) {

if(root == null){ // 如果二叉树为空，返回真

return true;

}

// 如果左子树和右子树高度相差大于1，则非平衡二叉树, getDepthRec()是前面实现过的求树高度的方法

if(Math.abs(getDepthRec(root.left) - getDepthRec(root.right)) > 1){

return false;

}

// 递归判断左子树和右子树是否为平衡二叉树

return isAVLRec(root.left) && isAVLRec(root.right);

}

/\*\*

\* 求二叉树的镜像 递归解法：

\* （1）如果二叉树为空，返回空

\* （2）如果二叉树不为空，求左子树和右子树的镜像，然后交换左子树和右子树

\*/

// 1. 破坏原来的树，把原来的树改成其镜像

public static TreeNode mirrorRec(TreeNode root) {

if (root == null) {

return null;

}

TreeNode left = mirrorRec(root.left);

TreeNode right = mirrorRec(root.right);

root.left = right;

root.right = left;

return root;

}

// 2. 不能破坏原来的树，返回一个新的镜像树

public static TreeNode mirrorCopyRec(TreeNode root){

if(root == null){

return null;

}

TreeNode newNode = new TreeNode(root.val);

newNode.left = mirrorCopyRec(root.right);

newNode.right = mirrorCopyRec(root.left);

return newNode;

}

// 3. 判断两个树是否互相镜像

public static boolean isMirrorRec(TreeNode r1, TreeNode r2){

// 如果两个树都是空树，则返回true

if(r1==null && r2==null){

return true;

}

// 如果有一棵树是空树，另一颗不是，则返回false

if(r1==null || r2==null){

return false;

}

// 如果两个树都非空树，则先比较根节点

if(r1.val != r2.val){

return false;

}

// 递归比较r1的左子树的镜像是不是r2右子树 和

// r1的右子树的镜像是不是r2左子树

return isMirrorRec(r1.left, r2.right) && isMirrorRec(r1.right, r2.left);

}

// 1. 破坏原来的树，把原来的树改成其镜像

public static void mirror(TreeNode root) {

if(root == null){

return;

}

Stack<TreeNode> stack = new Stack<TreeNode>();

stack.push(root);

while( !stack.isEmpty() ){

TreeNode cur = stack.pop();

// 交换左右孩子

TreeNode tmp = cur.right;

cur.right = cur.left;

cur.left = tmp;

if(cur.right != null){

stack.push(cur.right);

}

if(cur.left != null){

stack.push(cur.left);

}

}

}

// 2. 不能破坏原来的树，返回一个新的镜像树

public static TreeNode mirrorCopy(TreeNode root){

if(root == null){

return null;

}

Stack<TreeNode> stack = new Stack<TreeNode>();

Stack<TreeNode> newStack = new Stack<TreeNode>();

stack.push(root);

TreeNode newRoot = new TreeNode(root.val);

newStack.push(newRoot);

while( !stack.isEmpty() ){

TreeNode cur = stack.pop();

TreeNode newCur = newStack.pop();

if(cur.right != null){

stack.push(cur.right);

newCur.left = new TreeNode(cur.right.val);

newStack.push(newCur.left);

}

if(cur.left != null){

stack.push(cur.left);

newCur.right = new TreeNode(cur.left.val);

newStack.push(newCur.right);

}

}

return newRoot;

}

/\*\*

\* 求二叉树中两个节点的最低公共祖先节点

\* 递归解法：

\* （1）如果两个节点分别在根节点的左子树和右子树，则返回根节点

\* （2）如果两个节点都在左子树，则递归处理左子树；如果两个节点都在右子树，则递归处理右子树

\*/

public static TreeNode getLastCommonParentRec(TreeNode root, TreeNode n1, TreeNode n2) {

if (findNodeRec(root.left, n1)) { // 如果n1在树的左子树

if (findNodeRec(root.right, n2)) { // 如果n2在树的右子树

return root; // 返回根节点

} else { // 如果n2也在树的左子树

return getLastCommonParentRec(root.left, n1, n2); // 递归处理

}

} else { // 如果n1在树的右子树

if (findNodeRec(root.left, n2)) { // 如果n2在左子树

return root;

} else { // 如果n2在右子树

return getLastCommonParentRec(root.right, n1, n2); // 递归处理

}

}

}

// 帮助方法，递归判断一个点是否在树里

private static boolean findNodeRec(TreeNode root, TreeNode node) {

if (root == null || node == null) {

return false;

}

if (root == node) {

return true;

}

// 先尝试在左子树中查找

boolean found = findNodeRec(root.left, node);

if (!found) { // 如果查找不到，再在右子树中查找

found = findNodeRec(root.right, node);

}

return found;

}

// 求二叉树中两个节点的最低公共祖先节点 （更加简洁版的递归）

public static TreeNode getLastCommonParentRec2(TreeNode root, TreeNode n1, TreeNode n2) {

if(root == null){

return null;

}

// 如果有一个match，则说明当前node就是要找的最低公共祖先

if(root.equals(n1) || root.equals(n2)){

return root;

}

TreeNode commonInLeft = getLastCommonParentRec2(root.left, n1, n2);

TreeNode commonInRight = getLastCommonParentRec2(root.right, n1, n2);

// 如果一个左子树找到，一个在右子树找到，则说明root是唯一可能的最低公共祖先

if(commonInLeft!=null && commonInRight!=null){

return root;

}

// 其他情况是要不然在左子树要不然在右子树

if(commonInLeft != null){

return commonInLeft;

}

return commonInRight;

}

/\*\*

\* 非递归解法：

\* 先求从根节点到两个节点的路径，然后再比较对应路径的节点就行，最后一个相同的节点也就是他们在二叉树中的最低公共祖先节点

\*/

public static TreeNode getLastCommonParent(TreeNode root, TreeNode n1, TreeNode n2) {

if (root == null || n1 == null || n2 == null) {

return null;

}

ArrayList<TreeNode> p1 = new ArrayList<TreeNode>();

boolean res1 = getNodePath(root, n1, p1);

ArrayList<TreeNode> p2 = new ArrayList<TreeNode>();

boolean res2 = getNodePath(root, n2, p2);

if (!res1 || !res2) {

return null;

}

TreeNode last = null;

Iterator<TreeNode> iter1 = p1.iterator();

Iterator<TreeNode> iter2 = p2.iterator();

while (iter1.hasNext() && iter2.hasNext()) {

TreeNode tmp1 = iter1.next();

TreeNode tmp2 = iter2.next();

if (tmp1 == tmp2) {

last = tmp1;

} else { // 直到遇到非公共节点

break;

}

}

return last;

}

// 把从根节点到node路径上所有的点都添加到path中

private static boolean getNodePath(TreeNode root, TreeNode node, ArrayList<TreeNode> path) {

if (root == null) {

return false;

}

path.add(root); // 把这个节点加到路径中

if (root == node) {

return true;

}

boolean found = false;

found = getNodePath(root.left, node, path); // 先在左子树中找

if (!found) { // 如果没找到，再在右子树找

found = getNodePath(root.right, node, path);

}

if (!found) { // 如果实在没找到证明这个节点不在路径中，说明刚才添加进去的不是路径上的节点，删掉！

path.remove(root);

}

return found;

}

/\*\*

\* 求二叉树中节点的最大距离 即二叉树中相距最远的两个节点之间的距离。 (distance / diameter)

\* 递归解法：

\* （1）如果二叉树为空，返回0，同时记录左子树和右子树的深度，都为0

\* （2）如果二叉树不为空，最大距离要么是左子树中的最大距离，要么是右子树中的最大距离，

\* 要么是左子树节点中到根节点的最大距离+右子树节点中到根节点的最大距离，

\* 同时记录左子树和右子树节点中到根节点的最大距离。

\*

\* http://www.cnblogs.com/miloyip/archive/2010/02/25/1673114.[html](http://www.2cto.com/kf/qianduan/css/)

\*

\* 计算一个二叉树的最大距离有两个情况:

情况A: 路径经过左子树的最深节点，通过根节点，再到右子树的最深节点。

情况B: 路径不穿过根节点，而是左子树或右子树的最大距离路径，取其大者。

只需要计算这两个情况的路径距离，并取其大者，就是该二叉树的最大距离

\*/

public static Result getMaxDistanceRec(TreeNode root){

if(root == null){

Result empty = new Result(0, -1); // 目的是让调用方 +1 后，把当前的不存在的 (NULL) 子树当成最大深度为 0

return empty;

}

// 计算出左右子树分别最大距离

Result lmd = getMaxDistanceRec(root.left);

Result rmd = getMaxDistanceRec(root.right);

Result res = new Result();

res.maxDepth = Math.max(lmd.maxDepth, rmd.maxDepth) + 1; // 当前最大深度

// 取情况A和情况B中较大值

res.maxDistance = Math.max( lmd.maxDepth+rmd.maxDepth, Math.max(lmd.maxDistance, rmd.maxDistance) );

return res;

}

private static class Result{

int maxDistance;

int maxDepth;

public Result() {

}

public Result(int maxDistance, int maxDepth) {

this.maxDistance = maxDistance;

this.maxDepth = maxDepth;

}

}

/\*\*

\* 13. 由前序遍历序列和中序遍历序列重建二叉树（递归）

\* 感觉这篇是讲的最为清晰的:

\* http://crackinterviewtoday.wordpress.com/2010/03/15/rebuild-a-binary-tree-from-inorder-and-preorder-traversals/

\* 文中还提到一种避免开额外空间的方法，等下次补上

\*/

public static TreeNode rebuildBinaryTreeRec(List<Integer> preOrder, List<Integer> inOrder){

TreeNode root = null;

List<Integer> leftPreOrder;

List<Integer> rightPreOrder;

List<Integer> leftInorder;

List<Integer> rightInorder;

int inorderPos;

int preorderPos;

if ((preOrder.size() != 0) && (inOrder.size() != 0))

{

// 把preorder的第一个元素作为root

root = new TreeNode(preOrder.get(0));

// Based upon the current node data seperate the traversals into leftPreorder, rightPreorder,

// leftInorder, rightInorder lists

// 因为知道root节点了，所以根据root节点位置，把preorder，inorder分别划分为 root左侧 和 右侧 的两个子区间

inorderPos = inOrder.indexOf(preOrder.get(0)); // inorder序列的分割点

leftInorder = inOrder.subList(0, inorderPos);

rightInorder = inOrder.subList(inorderPos + 1, inOrder.size());

preorderPos = leftInorder.size(); // preorder序列的分割点

leftPreOrder = preOrder.subList(1, preorderPos + 1);

rightPreOrder = preOrder.subList(preorderPos + 1, preOrder.size());

root.left = rebuildBinaryTreeRec(leftPreOrder, leftInorder); // root的左子树就是preorder和inorder的左侧区间而形成的树

root.right = rebuildBinaryTreeRec(rightPreOrder, rightInorder); // root的右子树就是preorder和inorder的右侧区间而形成的树

}

return root;

}

/\*\*

14. 判断二叉树是不是完全二叉树（迭代）

若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，

第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。

有如下算法，按层次（从上到下，从左到右）遍历二叉树，当遇到一个节点的左子树为空时，

则该节点右子树必须为空，且后面遍历的节点左右子树都必须为空，否则不是完全二叉树。

\*/

public static boolean isCompleteBinaryTree(TreeNode root){

if(root == null){

return false;

}

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

boolean mustHaveNoChild = false;

boolean result = true;

while( !queue.isEmpty() ){

TreeNode cur = queue.remove();

if(mustHaveNoChild){ // 已经出现了有空子树的节点了，后面出现的必须为叶节点（左右子树都为空）

if(cur.left!=null || cur.right!=null){

result = false;

break;

}

} else {

if(cur.left!=null && cur.right!=null){ // 如果左子树和右子树都非空，则继续遍历

queue.add(cur.left);

queue.add(cur.right);

}else if(cur.left!=null && cur.right==null){ // 如果左子树非空但右子树为空，说明已经出现空节点，之后必须都为空子树

mustHaveNoChild = true;

queue.add(cur.left);

}else if(cur.left==null && cur.right!=null){ // 如果左子树为空但右子树非空，说明这棵树已经不是完全二叉完全树！

result = false;

break;

}else{ // 如果左右子树都为空，则后面的必须也都为空子树

mustHaveNoChild = true;

}

}

}

return result;

}

/\*\*

\* 14. 判断二叉树是不是完全二叉树（递归）

\* http://stackoverflow.com/questions/1442674/how-to-determine-whether-a-binary-tree-is-complete

\*

\*/

public static boolean isCompleteBinaryTreeRec(TreeNode root){

// Pair notComplete = new Pair(-1, false);

// return !isCompleteBinaryTreeSubRec(root).equalsTo(notComplete);

return isCompleteBinaryTreeSubRec(root).height != -1;

}

// 递归判断是否满树（完美）

public static boolean isPerfectBinaryTreeRec(TreeNode root){

return isCompleteBinaryTreeSubRec(root).isFull;

}

// 递归，要创建一个Pair class来保存树的高度和是否已满的信息

public static Pair isCompleteBinaryTreeSubRec(TreeNode root){

if(root == null){

return new Pair(0, true);

}

Pair left = isCompleteBinaryTreeSubRec(root.left);

Pair right = isCompleteBinaryTreeSubRec(root.right);

// 左树满节点，而且左右树相同高度，则是唯一可能形成满树（若右树也是满节点）的情况

if(left.isFull && left.height==right.height){

return new Pair(1+left.height, right.isFull);

}

// 左树非满，但右树是满节点，且左树高度比右树高一

// 注意到如果其左树为非完全树，则它的高度已经被设置成-1，

// 因此不可能满足第二个条件！

if(right.isFull && left.height==right.height+1){

return new Pair(1+left.height, false);

}

// 其他情况都是非完全树，直接设置高度为-1

return new Pair(-1, false);

}

private static class Pair{

int height; // 树的高度

boolean isFull; // 是否是个满树

public Pair(int height, boolean isFull) {

this.height = height;

this.isFull = isFull;

}

public boolean equalsTo(Pair obj){

return this.height==obj.height && this.isFull==obj.isFull;

}

}

}