

# 法律声明

---

□ 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

□ 课程详情请咨询

■ 微信公众号：小象

■ 新浪微博：ChinaHadoop



---

# 第五课 二叉树与图

林沐

# 内容概述

---

## 1.5道经典二叉树与图的相关题目

预备知识:二叉树基础知识

例1:路径之和2(medium) (二叉树深搜)

例2:最近的公共祖先(medium) (二叉树性质)

例3:二叉树转链表(medium) (二叉树与链表)

预备知识:二叉树层次遍历

例4:侧面观察二叉树(medium) (二叉树宽搜)

预备知识:图的基础知识

例5:课程安排(有向图判断环)(medium)

## 2.详细讲解题目解题方法、代码实现

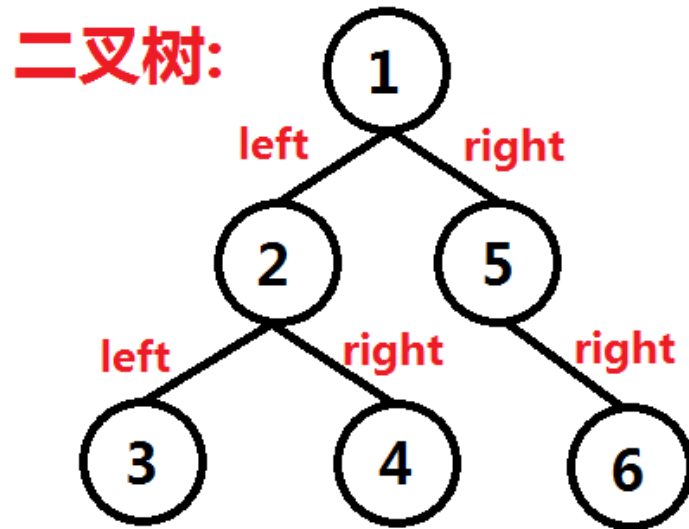
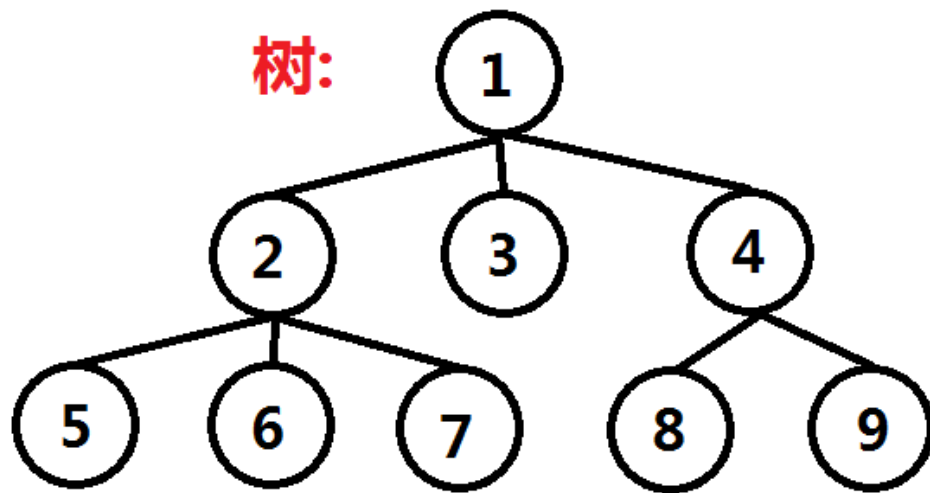
# 预备知识:二叉树定义

**树**是 $n(n \geq 0)$ 个节点的**有限集**，且这些节点**满足**如下关系：

- (1)有且仅有一个节点没有父结点，该节点称为树的**根**。
- (2)除根外，其余的每个节点都有且仅有一个**父结点**。
- (3)树中的每一个节点都构成一个**以它为根**的树。

**二叉树**在满足树的条件时，满足如下条件：

每个节点最多有**两个孩子**(子树)，这两个子树有**左右之分**，次序不可颠倒。



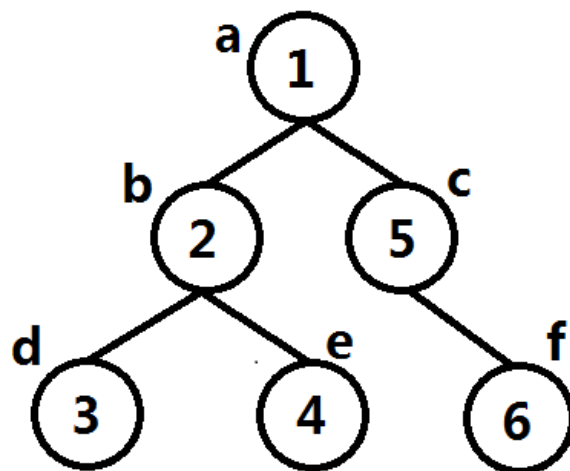
# 预备知识:二叉树构造

```
#include <stdio.h>
```

```
struct TreeNode {  
    int val;           //数据域val  
    TreeNode *left;    //left, right左右子树指针  
    TreeNode *right;  
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}  
};
```

```
void preorder_print(TreeNode *node, int layer) {  
    if (!node) {  
        return; //正在遍历的节点    //当前节点的层数  
    }  
    for (int i = 0; i < layer; i++) {  
        printf("-----"); //根据层数，打印相应数量的'-'  
    }  
    printf("[%d]\n", node->val);  
    preorder_print(node->left, layer + 1); //遍历左子树，层数+1  
    preorder_print(node->right, layer + 1); //遍历右子树，层数+1  
}
```

```
int main() {  
    TreeNode a(1);  
    TreeNode b(2);  
    TreeNode c(5);  
    TreeNode d(3);  
    TreeNode e(4);  
    TreeNode f(6);  
    a.left = &b;  
    a.right = &c;  
    b.left = &d;  
    b.right = &e;  
    c.right = &f;  
    preorder_print(&a, 0);  
    return 0;  
}
```



```
[1]  
-----[2]  
-----[3]  
-----[4]  
-----[5]  
-----[6]
```

# 预备知识:二叉树的深度遍历

```
void traversal(TreeNode *node) {  
    if (!node) {  
        return;  
    }  
}
```

**此时访问node称为前序遍历**

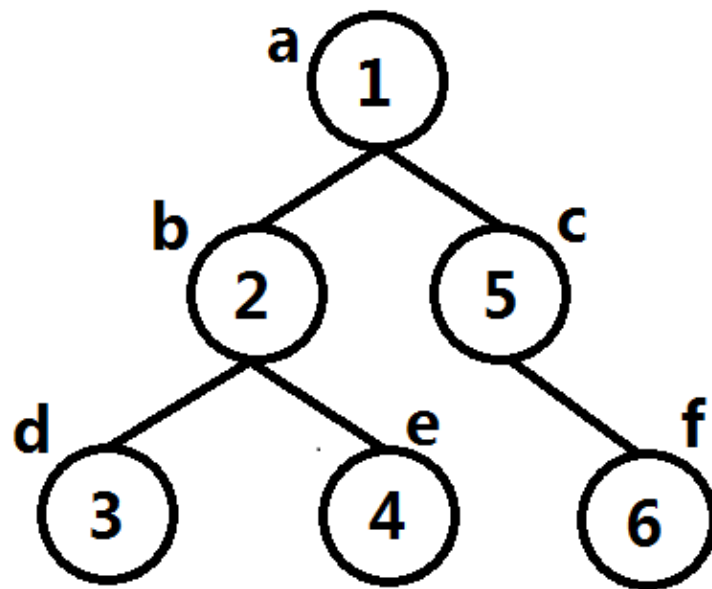
```
traversal(node->left);
```

**此时访问node称为中序遍历**

```
traversal(node->right);
```

**此时访问node称为后序遍历**

```
}
```



**前序遍历: a(1), b(2), d(3), e(4), c(5), f(6)**

**中序遍历: d(3), b(2), e(4), a(1), c(5), f(6)**

**后序遍历: d(3), e(4), b(2), f(6), c(5), a(1)**

# 预备知识:二叉树的遍历课堂练习

```
void traversal_print1(TreeNode *node, int layer) {
    if (!node) {
        return;
    }
    traversal_print1(node->left, layer + 1);
    for (int i = 0; i < layer; i++) {
        printf("-----");
    }
    printf("[%d]\n", node->val);
    traversal_print1(node->right, layer + 1);
}
```

A

X

```
[1]
-----[2]
-----[3]
-----[4]
-----[5]
-----[6]
```

```
void traversal_print2(TreeNode *node, int layer) {
    if (!node) {
        return;
    }
    traversal_print2(node->left, layer + 1);
    traversal_print2(node->right, layer + 1);
    for (int i = 0; i < layer; i++) {
        printf("-----");
    }
    printf("[%d]\n", node->val);
}
```

B

Y

```
-----[3]
-----[2]
-----[4]
[1]
-----[5]
-----[6]
```

```
void traversal_print3(TreeNode *node, int layer) {
    if (!node) {
        return;
    }
    for (int i = 0; i < layer; i++) {
        printf("-----");
    }
    printf("[%d]\n", node->val);
    traversal_print3(node->left, layer + 1);
    traversal_print3(node->right, layer + 1);
}
```

C

Z

```
-----[3]
-----[4]
-----[2]
-----[6]
-----[5]
[1]
```

A(?)

B(?)

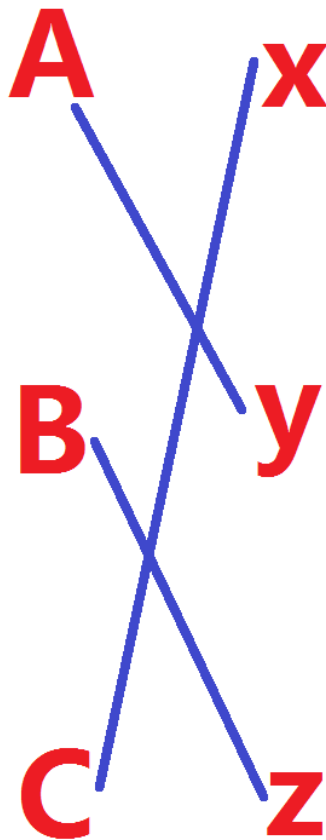
C(?)

# 预备知识:二叉树的遍历讲解

```
void traversal_print1(TreeNode *node, int layer) {
    if (!node) {
        return;
    }
    traversal_print1(node->left, layer + 1);
    for (int i = 0; i < layer; i++) {
        printf("-----");
    }
    printf("[%d]\n", node->val);
    traversal_print1(node->right, layer + 1);
}

void traversal_print2(TreeNode *node, int layer) {
    if (!node) {
        return;
    }
    traversal_print2(node->left, layer + 1);
    traversal_print2(node->right, layer + 1);
    for (int i = 0; i < layer; i++) {
        printf("-----");
    }
    printf("[%d]\n", node->val);
}

void traversal_print3(TreeNode *node, int layer) {
    if (!node) {
        return;
    }
    for (int i = 0; i < layer; i++) {
        printf("-----");
    }
    printf("[%d]\n", node->val);
    traversal_print3(node->left, layer + 1);
    traversal_print3(node->right, layer + 1);
}
```



```
[1]
-----[2]
-----[3]
-----[4]
-----[5]
-----[6]
```

```
-----[3]
-----[2]
-----[4]
[1]
-----[5]
-----[6]
```

```
-----[3]
-----[4]
-----[2]
-----[6]
-----[5]
[1]
```

A(y)

B(z)

C(x)



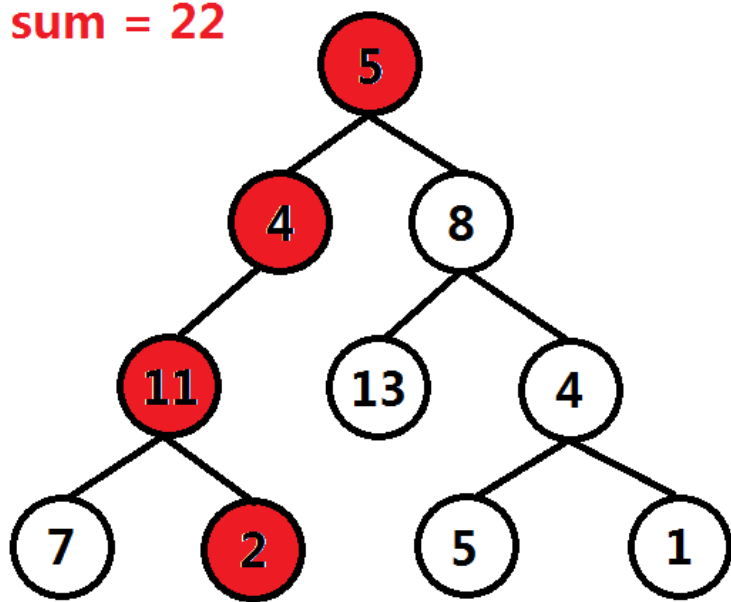
# 例1:路径之和2

给定一个**二叉树**与整数sum，找出**所有**从根节点到叶结点的**路径**，这些路径上的节点值**累加和为sum**。

```
#include <vector>
struct TreeNode {    //树节点数据结构
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) :
        val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    std::vector<std::vector<int>> >
        pathSum(TreeNode* root, int sum) {
    }    [[5, 4, 11, 2], [5, 8, 4, 5]]
};
```

sum = 22



选自 **LeetCode 113. Path Sum II**

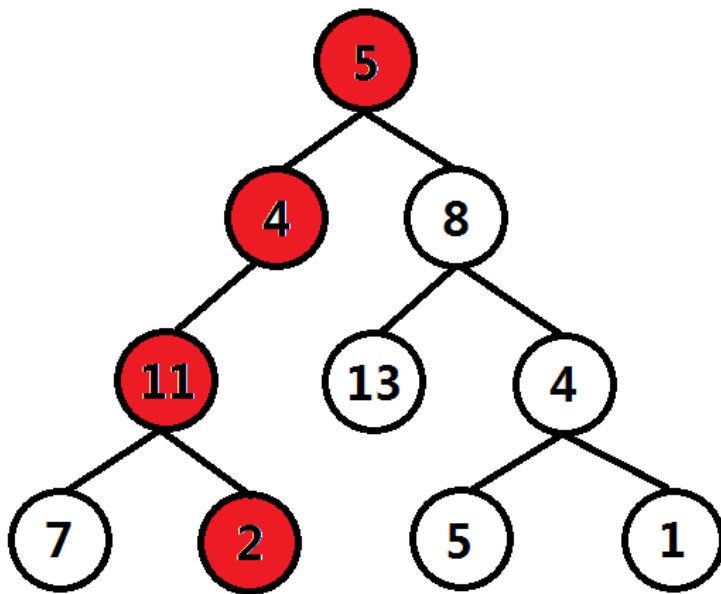
<https://leetcode.com/problems/path-sum-ii/description/>

难度:**Medium**

# 例1:思考

深度搜索所有从**根节点**到**叶结点**的**路径**，检查**各路径**上所有节点的**值的和**是否为sum。

sum = 22    [5, 4, 11, 2]    [5, 8, 4, 5]

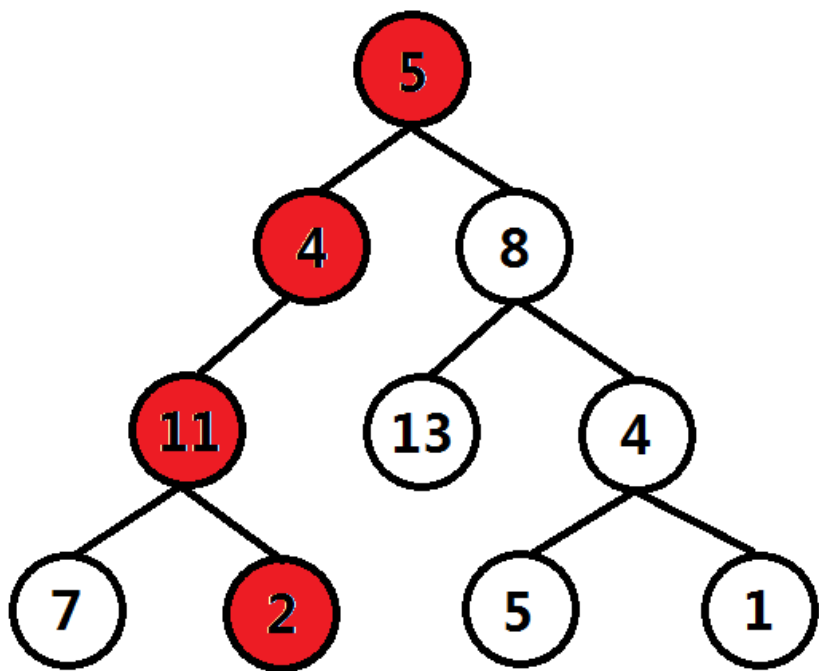


## 思考:

- 1.使用何种数据结构存储**遍历路径**上的节点?
- 2.在树的**前序遍历**时做什么?**后序遍历**时做什么?
- 3.如何判断一个节点为**叶结点**?当遍历到叶结点时应该做什么?

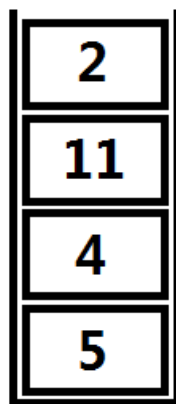
# 例1:算法思路

- 1.从根节点**深度遍历**二叉树，**先序遍历**时，将该节点值存储至**path栈**中(vector实现)，使用 path\_value**累加**节点值。
- 2.当遍历至**叶结点**时，检查**path\_value值**是否为sum，若为sum，则将path **push进入**result结果中。
- 3.在**后续遍历**时，将该节点值从path栈中**弹出**，path\_value**减去**节点值。

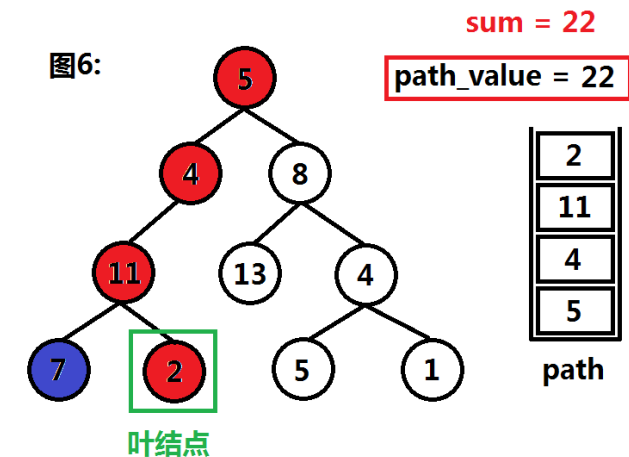
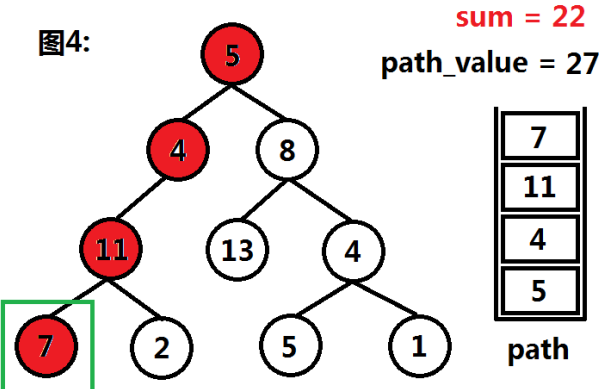
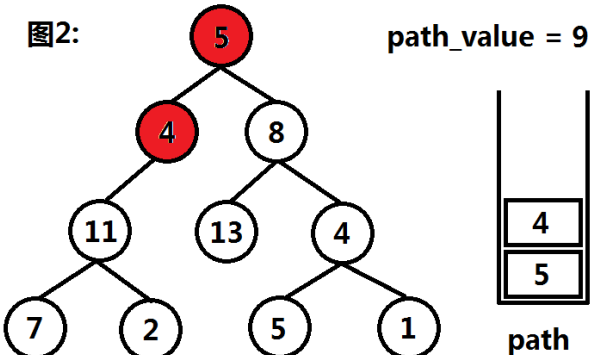
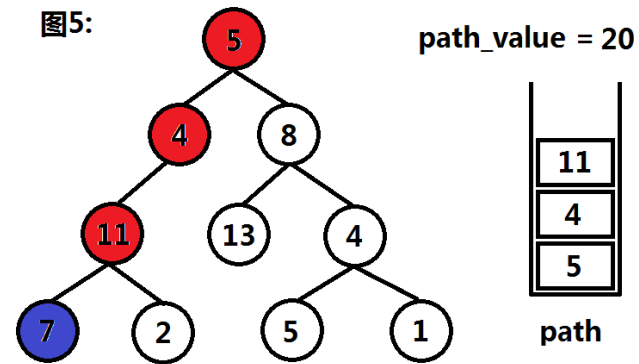
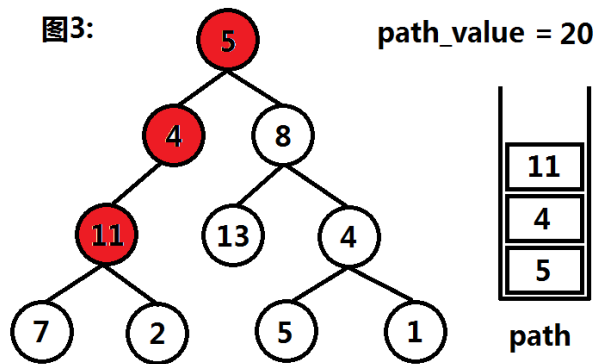
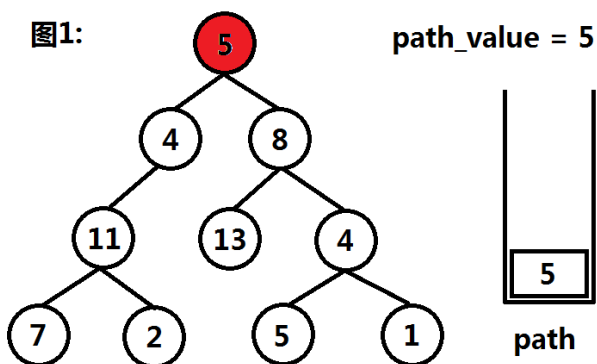


path\_value = 22

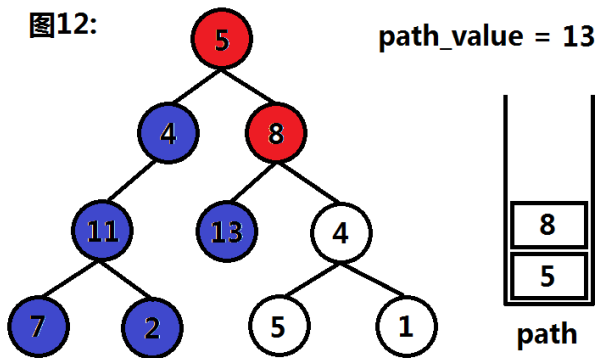
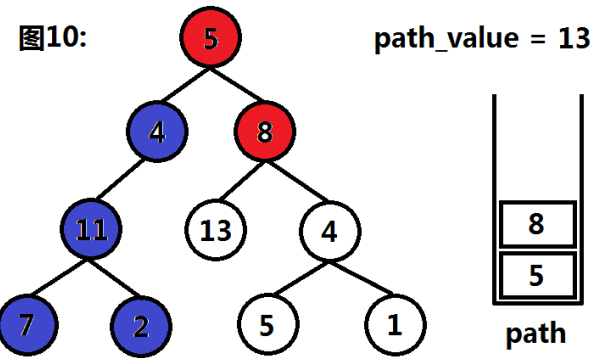
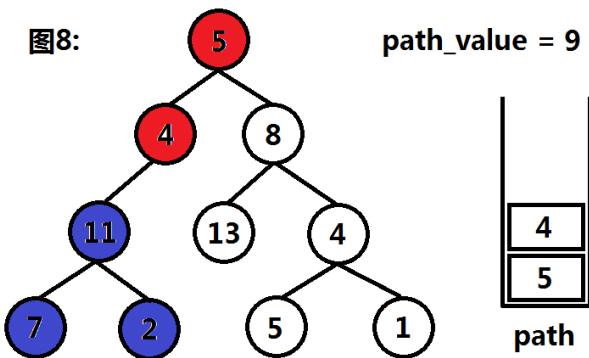
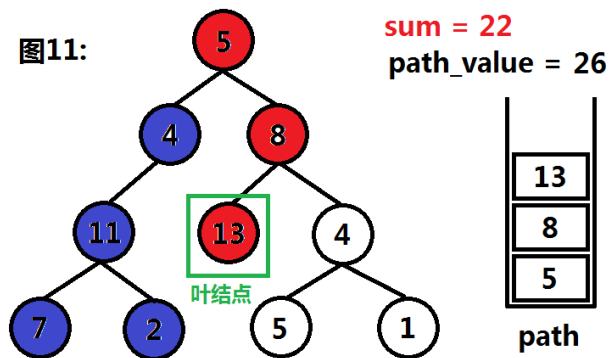
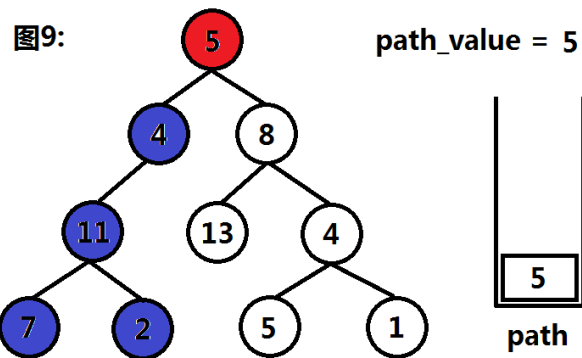
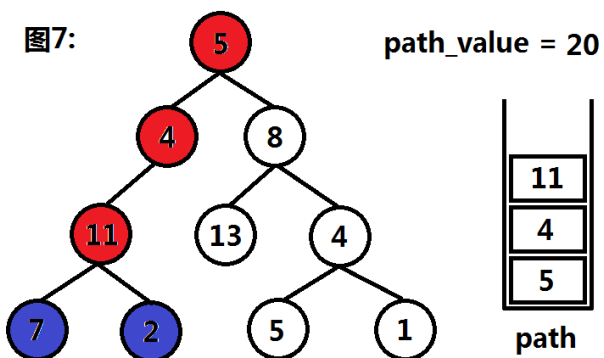
path\_value ?? sum



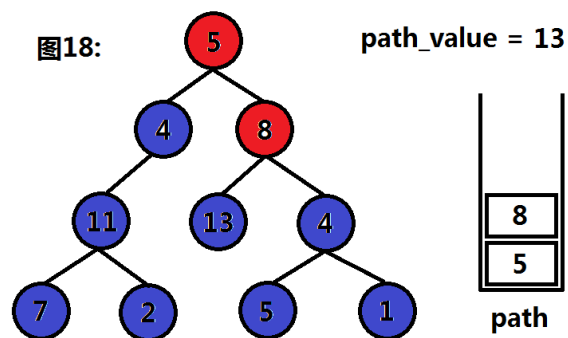
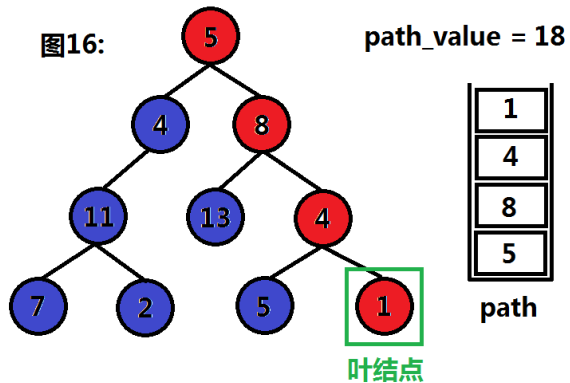
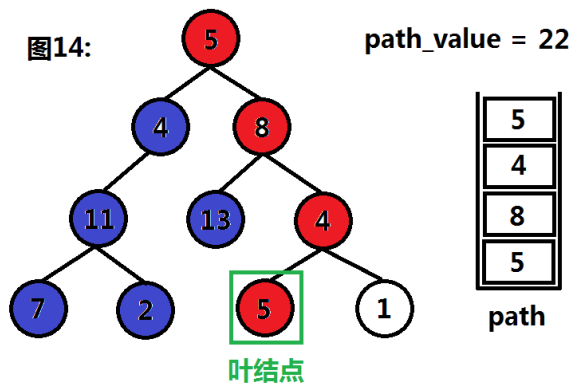
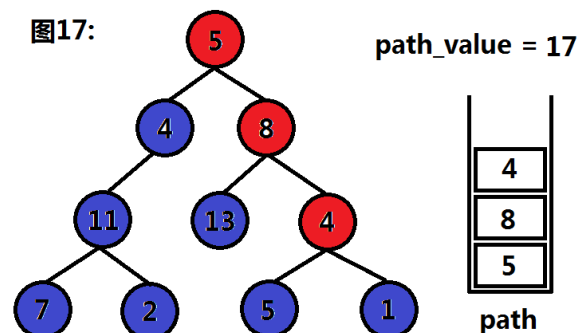
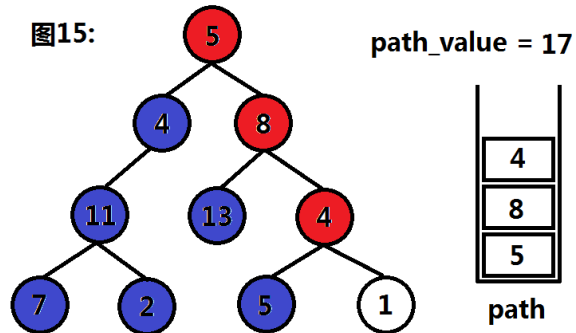
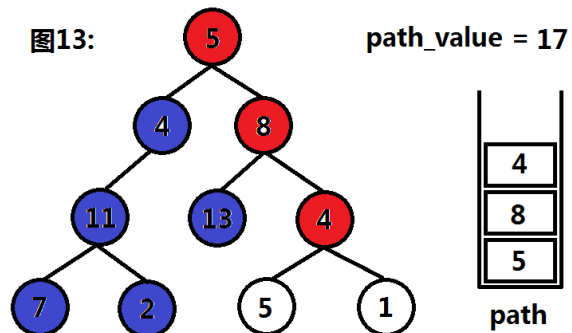
# 例1:算法思路



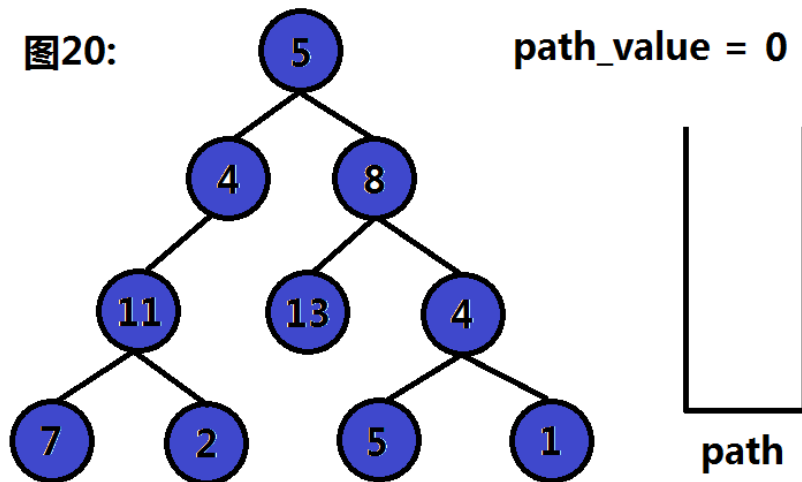
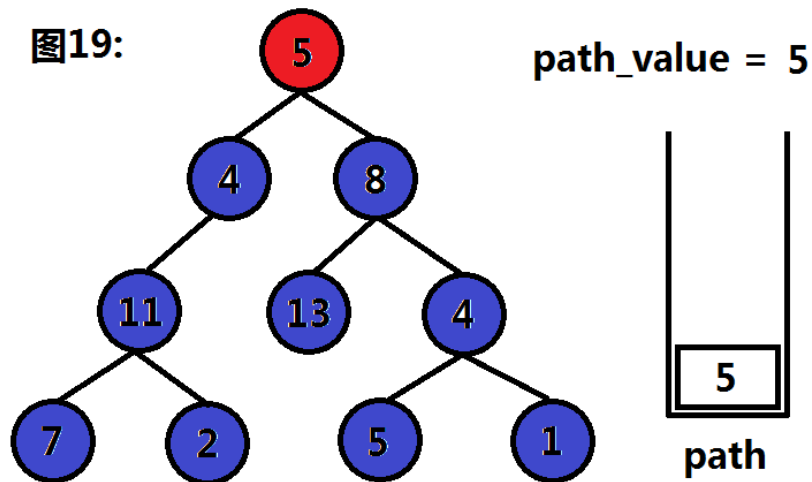
# 例1:算法思路



# 例1:算法思路



# 例1:算法思路



所有路径:

[5, 4, 11, 7]  $\text{path\_value} = 27$

[5, 4, 11, 2]  $\text{path\_value} = 22$

[5, 8, 13]  $\text{path\_value} = 26$

[5, 8, 4, 5]  $\text{path\_value} = 22$

[5, 4, 11, 1]  $\text{path\_value} = 21$

```
class Solution {
public:
    std::vector<std::vector<int>> > pathSum(TreeNode* root, int sum) {
        std::vector<std::vector<int>> > result; //存储满足条件路径的数组
        std::vector<int> path; //路径栈与路径值
        int path_value = 0;
        preorder(root, path_value, sum, path, result);
        return result;
    }
private:
    void preorder(TreeNode *node, int &path_value, int sum,
        std::vector<int> &path,
        std::vector<std::vector<int>> > &result){
        if (!node){
            return;
        } //遍历一个节点即更新一次路径值
        path_value += node->val;
        1
        if ( 2 ) {
            result.push_back(path); //满足 ?? 条件时，将path添加至结果数组
        }
        preorder(node->left, path_value, sum, path, result);
        preorder(node->right, path_value, sum, path, result);
        3
        path.pop_back(); //遍历完成后，将该节点送路径栈中弹出
    }
};
```

## 例1:课堂练习

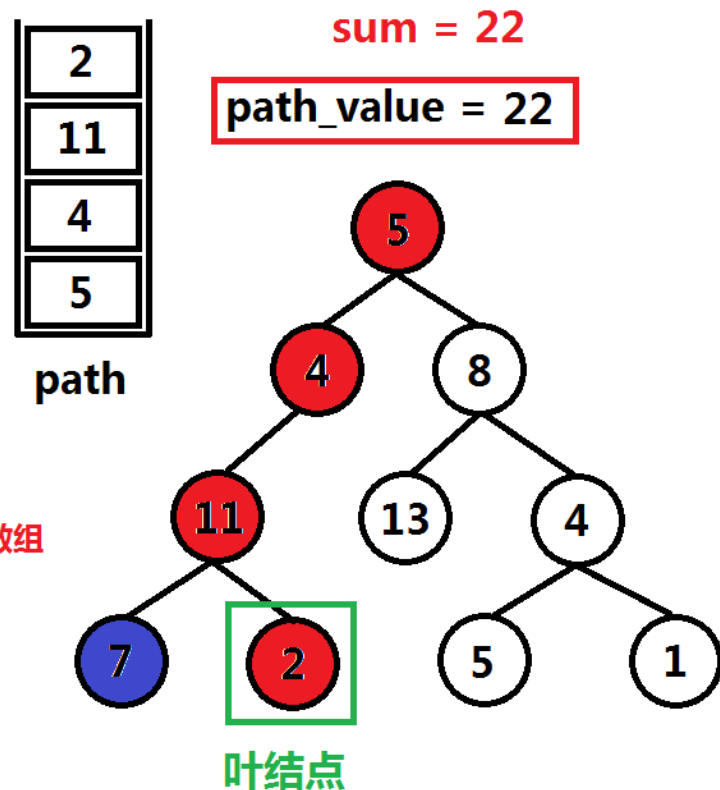
3分钟填写代码，  
有问题随时提出！



# 例1:实现

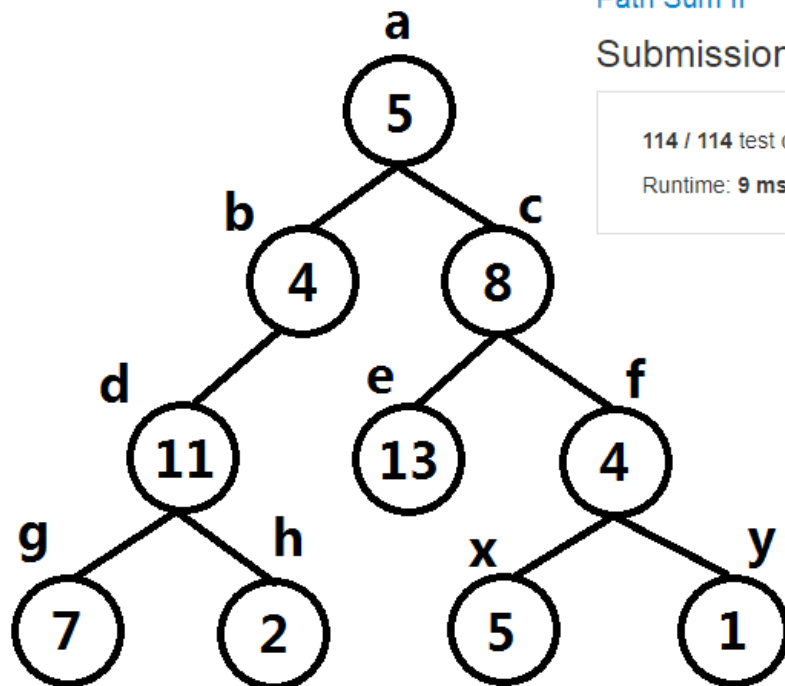
```
void preorder(TreeNode *node, int &path_value, int sum,
               std::vector<int> &path,
               std::vector<std::vector<int> > &result) {
    if (!node) {
        return;
    }
    //遍历一个节点即更新一次路径值
    path_value += node->val;
    path.push_back(node->val);

    if (!node->left && !node->right && path_value == sum) {
        result.push_back(path); //满足 ?? 条件时，将path添加至结果数组
    }
    preorder(node->left, path_value, sum, path, result);
    preorder(node->right, path_value, sum, path, result);
    path_value -= node->val;
    path.pop_back(); //遍历完成后，将该节点送路径栈中弹出
}
```



# 例1:测试与leetcode提交结果

```
int main() {
    TreeNode a(5);
    TreeNode b(4);
    TreeNode c(8);
    TreeNode d(11);
    TreeNode e(13);
    TreeNode f(4);
    TreeNode g(7);
    TreeNode h(2);
    TreeNode x(5);
    TreeNode y(1);
    a.left = &b;
    a.right = &c;
    b.left = &d;
    c.left = &e;
    c.right = &f;
    d.left = &g;
    d.right = &h;
    f.left = &x;
    f.right = &y;
    Solution solve;
    std::vector<std::vector<int>> result = solve.pathSum(&a, 22);
    for (int i = 0; i < result.size(); i++){
        for (int j = 0; j < result[i].size(); j++){
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```



Path Sum II

Submission Details

114 / 114 test cases passed.

Status: Accepted

Runtime: 9 ms

Submitted: 0 minutes ago

```
[5][4][11][2]
[5][8][4][5]
```

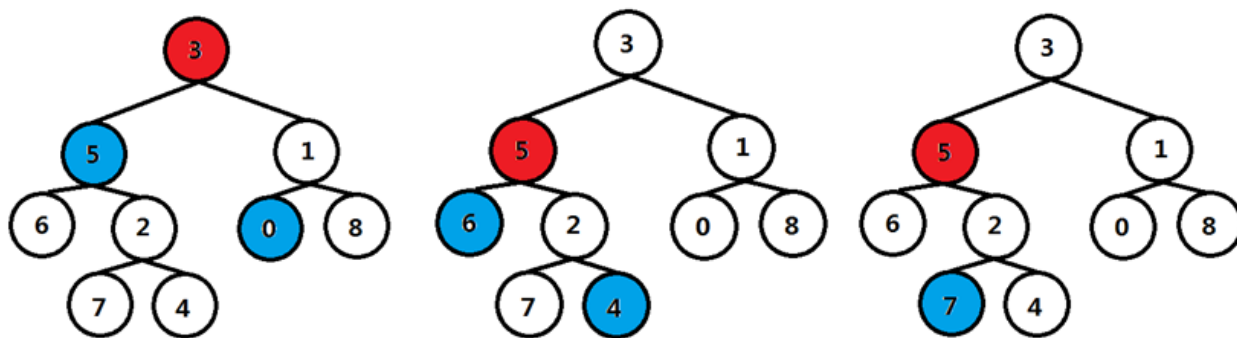


小象学院  
ChinaHadoop.cn

# 例2:最近的公共祖先

已知二叉树，求二叉树中给定的两个节点的**最近公共祖先**。

最近公共祖先: 两节点v与w的最近公共祖先u，满足在**树上最低(离根最远)**，且v,w两个节点都是u的**子孙**。



```
class Solution {  
public:  
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {  
        返回p,q最近公共祖先节点  
    }  
};
```

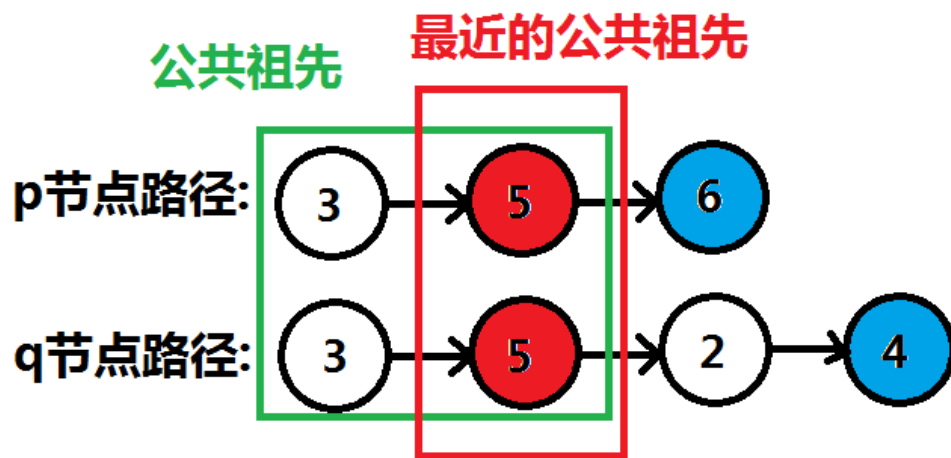
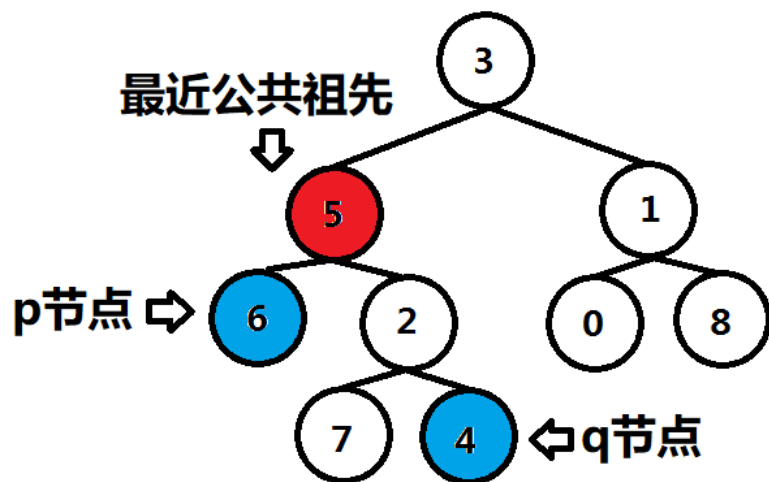
选自 **LeetCode 236. Lowest Common Ancestor of a Binary Tree**

<https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/description/>

难度:**Medium**

# 例2:思考与分析

- 1.两个节点的**公共祖先**一定在从根节点，至这两个节点的路径上。
- 2.由于求公共祖先中的**最近**公共祖先，那么即**同时出现**在这两条路径上的**离根节点最远**的节点(或离两个最近)。
- 3.最终算法即:求p节点路径，q节点路径，两路径上**最后一个相同**的节点。



# 例2:求根节点至某节点路径(深度搜索)

- 1.从根节点**遍历(搜索)**至该节点，**找到**该节点后就**结束搜索**。
- 2.将遍历过程中遇到的节点**按照顺序**存储起来，这些节点即路径节点。

前序遍历(深度优先遍历)

正在遍历的节点

```
void preorder(TreeNode *node,  
               TreeNode *search) {  
    if (!node) {  
        return;  
    } //如果遍历至空指针(叶结点的孩子), 结束
```

```
    if (node == search) {  
        //找到了待搜索节点, 做一些操作  
    }
```

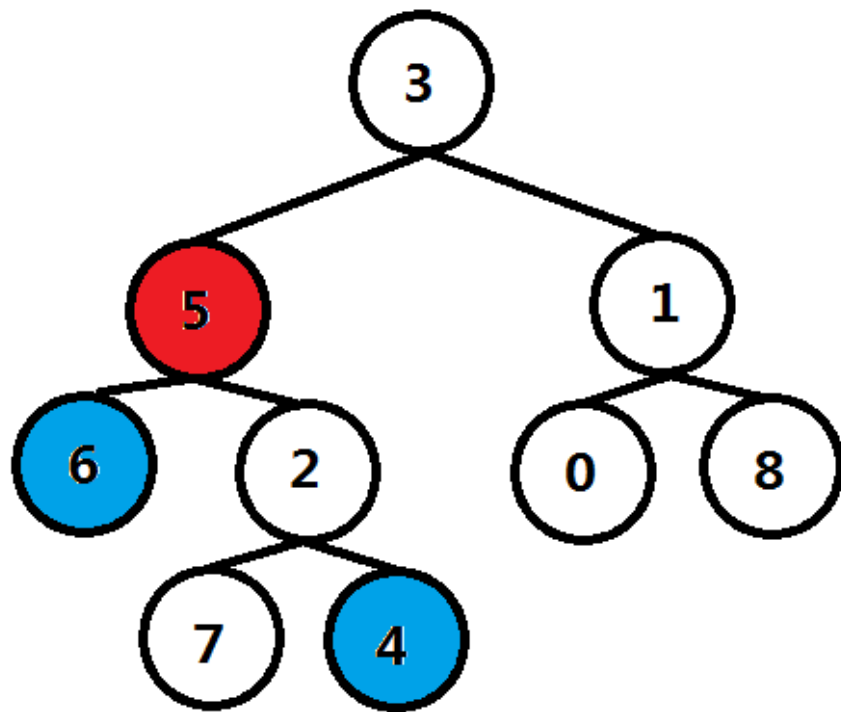
前序

```
    preorder(node->left, search);
```

中序

```
    preorder(node->right, search);
```

后续



# 例2:求根节点至某节点路径(栈存储路径)

记录节点p路径的过程:

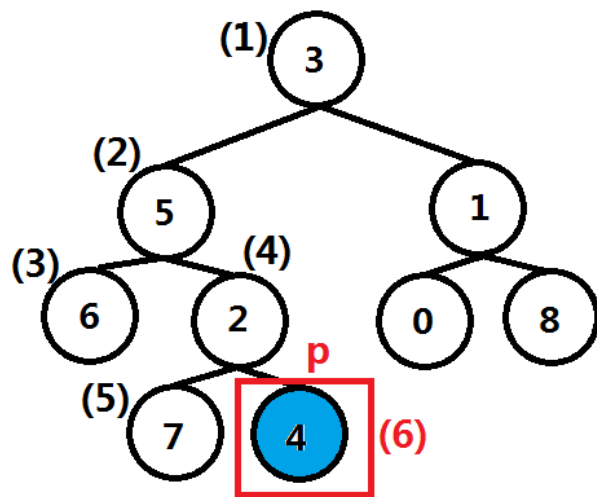
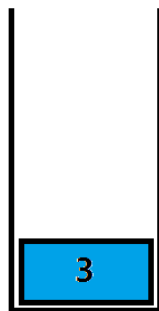
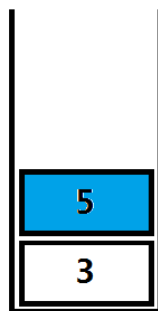


图1



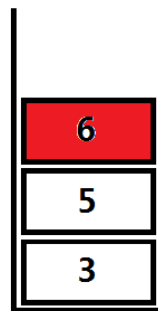
深度遍历节点(1)  
前序

图2



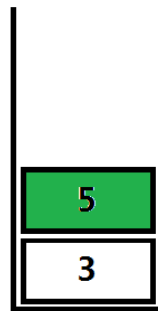
深度遍历节点(2)  
前序

图3



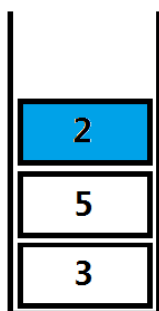
深度遍历节点(3)  
前序(后序)

图4



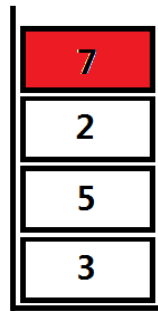
深度遍历节点(2)  
中序

图5



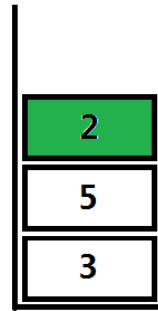
深度遍历节点(4)  
前序

图6



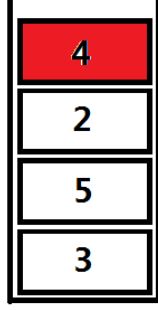
深度遍历节点(5)  
前序(后序)

图7



深度遍历节点(4)  
中序

图8



深度遍历节点(6)  
前序(后序)

# 例2:求根节点至某节点路径(实现, 课堂练习)

```
void preorder(TreeNode* node, //正在遍历的节点
              TreeNode* search, //待搜索节点
              std::vector<TreeNode*> &path, //遍历时的节点路径栈
              std::vector<TreeNode*> &result, //最终搜索到节点search的路径结果
              int &finish) { //记录是否找到节点search的变量, 未找到时是0, 找到为1

    if (!node || 1) { //当node为空或已找到search节点直接返回, 结束搜索
        return;
    }

    path.push_back(node); //先序遍历时, 将节点压入path栈

    if (node == search) {
        finish = 1; //当找到search节点后, 标记finish变量
        2
    }

    preorder(node->left, search, path, result, finish); //深度遍历node左孩子
    preorder(node->right, search, path, result, finish); //深度遍历node右孩子
    3
}
```

**3分钟**时间填写代码,  
**有问题随时提出!**

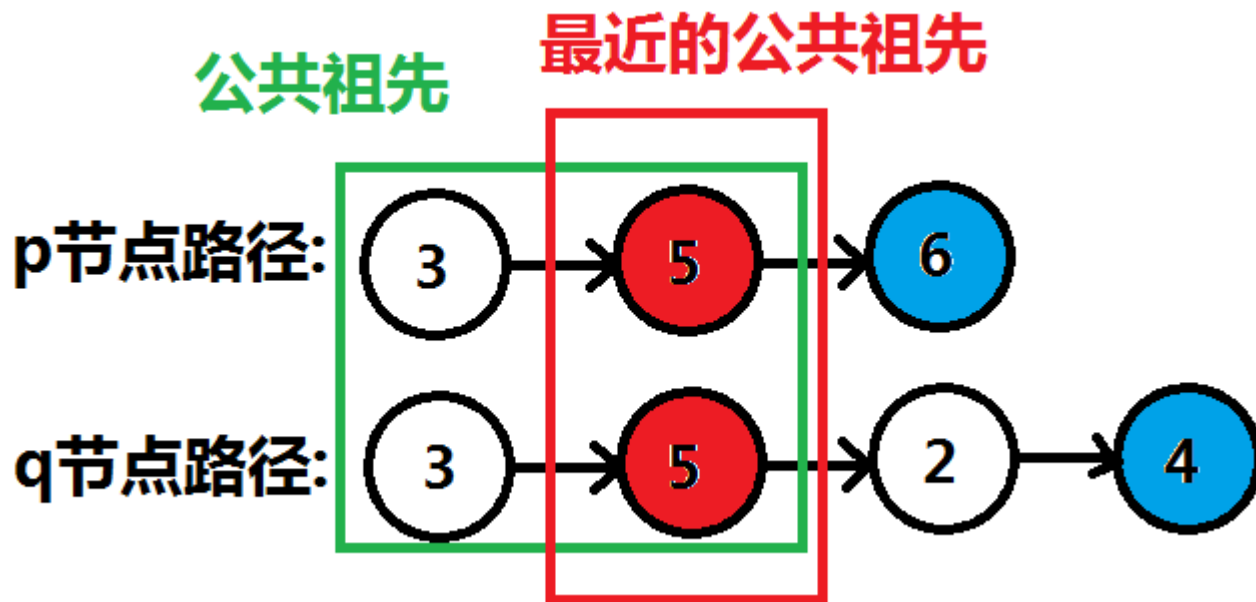
# 例2:求根节点至某节点路径(实现)

```
void preorder(TreeNode* node, //正在遍历的节点
               TreeNode* search, //待搜索节点
               std::vector<TreeNode*> &path, //遍历时的节点路径栈
               std::vector<TreeNode*> &result, //最终搜索到节点search的路径结果
               int &finish) { //记录是否找到节点search的变量，未找到时是0，找到为1
    1 if (!node || finish) { //当node为空或已找到search节点直接返回，结束搜索
        return;
    }
    path.push_back(node); //先序遍历时，将节点压入path栈
    if (node == search) {
        finish = 1; //当找到search节点后，标记finish变量
        2 result=path //将当前的path存储到result中
    }
    preorder(node->left, search, path, result, finish); //深度遍历node左孩子
    preorder(node->right, search, path, result, finish); //深度遍历node右孩子
    3 path.pop_back(); //结束遍历node时，将node节点弹出path栈
}
```



# 例2:求两路径上最后一个相同的节点

1. 求出**较短**路径的**长度n**。
2. 同时遍历p节点的路径与q节点的路径，遍历n个节点，**最后一个**发现的相同节点，即**最近公共祖先**。



# 例2:整体代码(实现, 课堂练习)

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {

        std::vector<TreeNode*> path; //声明遍历用的临时栈
        std::vector<TreeNode*> node_p_path; //存储p节点路径
        std::vector<TreeNode*> node_q_path; //存储q节点路径
        int finish = 0; //记录是否完成搜索的变量finish

        1

        path.clear();
        finish = 0; //清空path、finish, 计算q节点路径
        preorder(root, q, path, node_q_path, finish);
        int path_len = 0; //较短路径的长度

        if ( 2 ) {
            path_len = node_p_path.size();
        }
        else {
            path_len = node_q_path.size();
        }
        TreeNode *result = 0; 同时遍历根到p,q两个节点的的路径上的节点
        for (int i = 0; i < path_len; i++) {
            if ( 3 ) {
                result = node_p_path[i];
            }
        }
        return result;
    }
};
```

3分钟时间填写代码,  
有问题随时提出!

# 例2:整体代码(实现)

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {

        std::vector<TreeNode*> path; //声明遍历用的临时栈
        std::vector<TreeNode*> node_p_path; //存储p节点路径
        std::vector<TreeNode*> node_q_path; //存储q节点路径
        int finish = 0; //记录是否完成搜索的变量finish

        preorder(root, p, path, node_p_path, finish);

        path.clear(); //清空path、finish，计算q节点路径
        finish = 0;
        preorder(root, q, path, node_q_path, finish);
        int path_len = 0; //较短路径的长度
        if ( node_p_path.size() < node_q_path.size() ) {
            path_len = node_p_path.size();
        }
        else{
            path_len = node_q_path.size();
        }
        TreeNode *result = 0; 同时遍历根到p,q两个节点的的路径上的节点
        for (int i = 0; i < path_len; i++) {
            if ( node_p_path[i] == node_q_path[i] ) {
                result = node_p_path[i]; //找到了最近公共祖先
            }
        }
        return result;
    }
};
```

# 例2:测试与leetcode提交结果

```
int main() {
```

```
TreeNode a(3);  
TreeNode b(5);  
TreeNode c(1);  
TreeNode d(6);  
TreeNode e(2);  
TreeNode f(0);  
TreeNode x(8);  
TreeNode y(7);  
TreeNode z(4);  
a.left = &b;  
a.right = &c;  
b.left = &d;  
b.right = &e;  
c.left = &f;  
c.right = &x;  
e.left = &y;  
e.right = &z;
```

构造样例中的树

```
Solution solve;
```

```
TreeNode *result = solve.lowestCommonAncestor(&a, &b, &f);  
printf("lowestCommonAncestor = %d\n", result->val);  
result = solve.lowestCommonAncestor(&a, &d, &z);  
printf("lowestCommonAncestor = %d\n", result->val);  
result = solve.lowestCommonAncestor(&a, &b, &y);  
printf("lowestCommonAncestor = %d\n", result->val);
```

```
return 0;
```

```
}
```

```
lowestCommonAncestor = 3  
lowestCommonAncestor = 5  
lowestCommonAncestor = 5  
请按任意键继续. . .
```

Lowest Common Ancestor of a Binary Tree

## Submission Details

31 / 31 test cases passed.

Status: **Accepted**

Runtime: 19 ms

Submitted: 1 minute ago

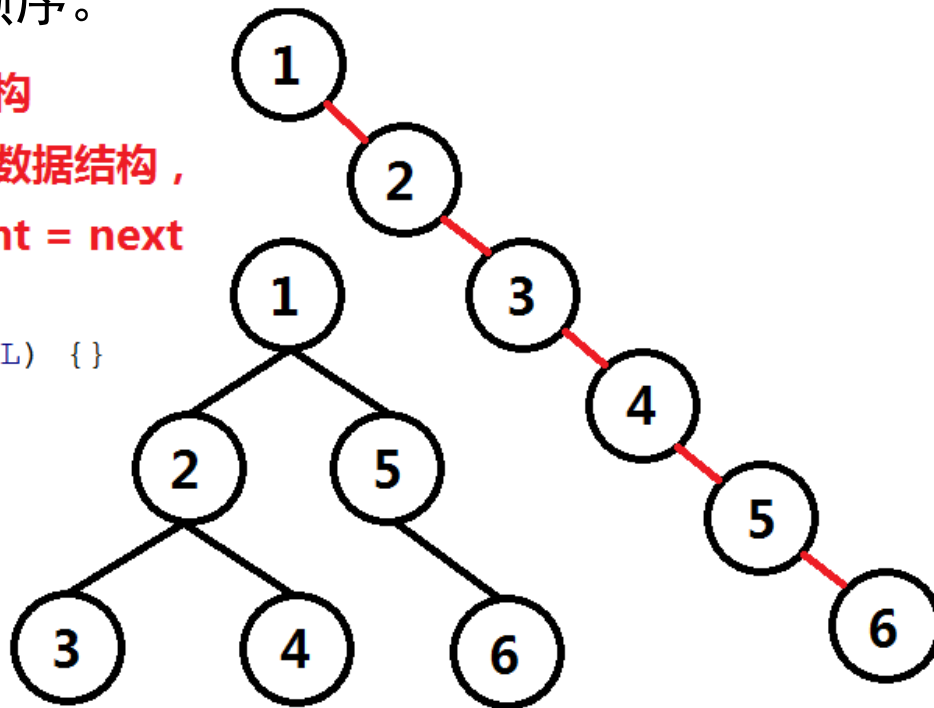
# 例3:二叉树转链表

给定一个**二叉树**，将该二叉树**就地(in-place)**转换为**单链表**。单链表中节点顺序为二叉树**前序遍历**顺序。

**//树节点的数据结构**

```
struct TreeNode {  
    int val;  
    TreeNode *left;  
    TreeNode *right;  
    TreeNode(int x) :  
        val(x), left(NULL), right(NULL) {}  
};  
  
class Solution {  
public:  
    void flatten(TreeNode *root)  
};
```

**//单链表仍使用该数据结构，  
即left=NULL,right = next**



选自 **LeetCode 114. Flatten Binary Tree to Linked List**

<https://leetcode.com/problems/flatten-binary-tree-to-linked-list/description/>

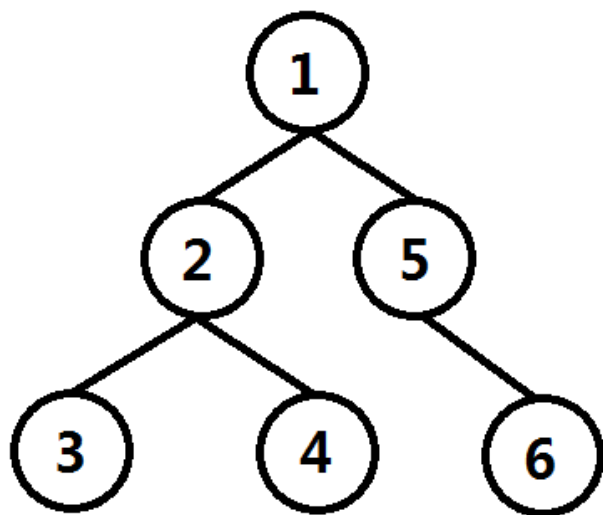
难度:**Medium**

# 例3:思考

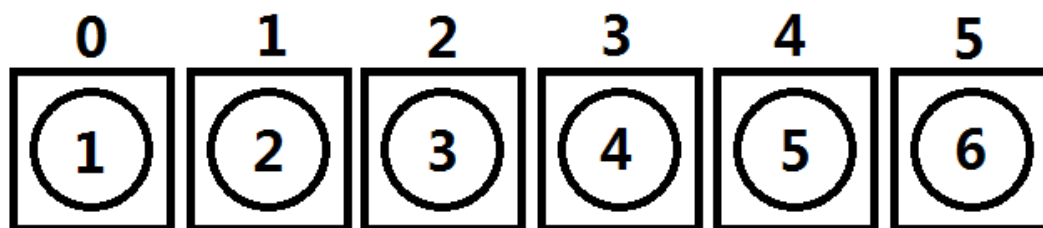
**前序遍历**二叉树，将节点指针push进入vector，顺序**遍历**vector中的节点，链接相邻两节点，形成链单链表。**(投机取巧)**

该方法虽然可通过题目，但不满足**就地(in-place)转换**的条件。

若**就地(in-place)转换**应该如何做？



vector:



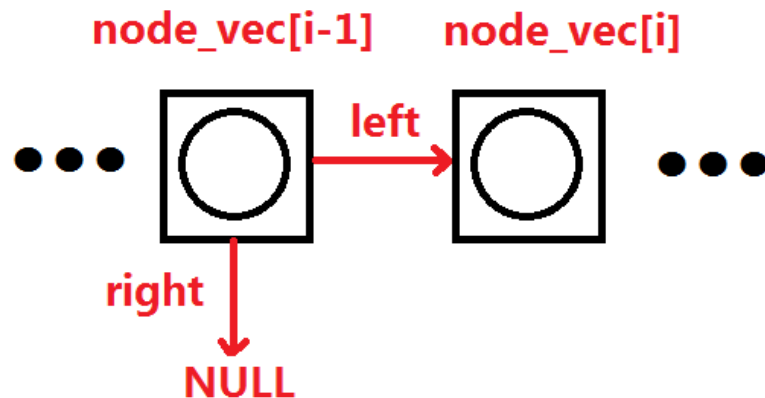
按顺序遍历vector，将前面的节点左指针置空，右指针与后面的节点相连。

# 例3:方法1课堂练习

```
#include <vector>
class Solution {
public:
    void flatten(TreeNode *root) {
        std::vector<TreeNode *> node_vec;
        preorder(root, node_vec);
        for (int i = 1; i < node_vec.size(); i++){
            1
            2
        }
    }
private:
    void preorder(TreeNode *node, std::vector<TreeNode *> &node_vec) {
        if (!node) {
            return;
        }
        3
        preorder(node->left, node_vec);
        preorder(node->right, node_vec);
    }
};
```

3分钟填写代码，  
有问题随时提出！

# 例3:方法1实现

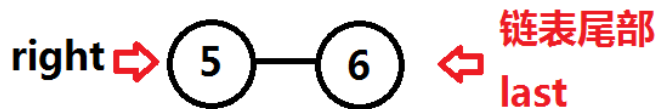
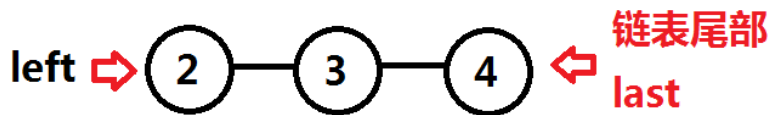
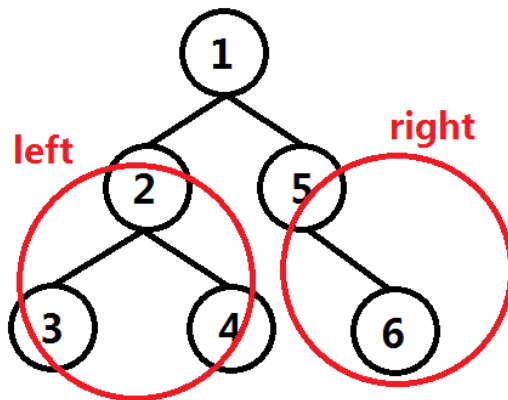


```
#include <vector>
class Solution {
public:
    void flatten(TreeNode *root) {
        std::vector<TreeNode *> node_vec;
        preorder(root, node_vec);
        for (int i = 1; i < node_vec.size(); i++) {
            node_vec[i-1]->left = NULL;
            node_vec[i-1]->right = node_vec[i];
        }
    }
private:
    void preorder(TreeNode *node, std::vector<TreeNode *> &node_vec) {
        if (!node) {
            return;
        }
        node_vec.push_back(node);
        preorder(node->left, node_vec);
        preorder(node->right, node_vec);
    }
};
```



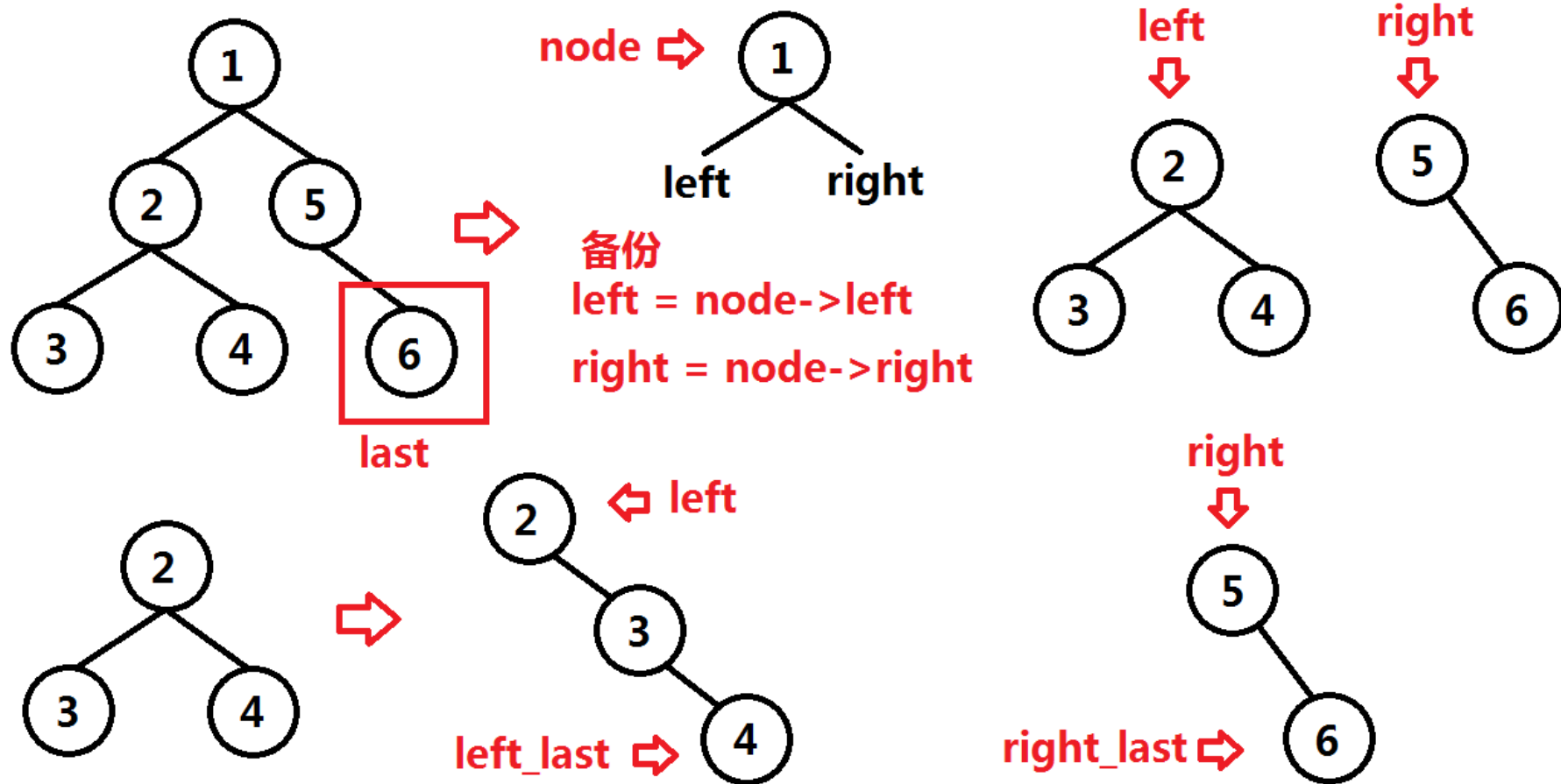
# 例3:算法思路(方法2整体)

```
//将以node为根的树转为链表(拉直), 并传出必要信息
void preorder(TreeNode *node,      ●●●      ){
    if (!node) {
        return;
    }
    if ( 页节点      ){
        do something
    }
    前序, 访问node时
    do something
    if (有左子树){      必要信息
        preorder(left,      ●●●      );
        中序, 完成左孩子的访问
    }
    if (有右子树){      必要信息
        preorder(right,      ●●●      );
        后序, 完成右孩子的访问
    }
}
```

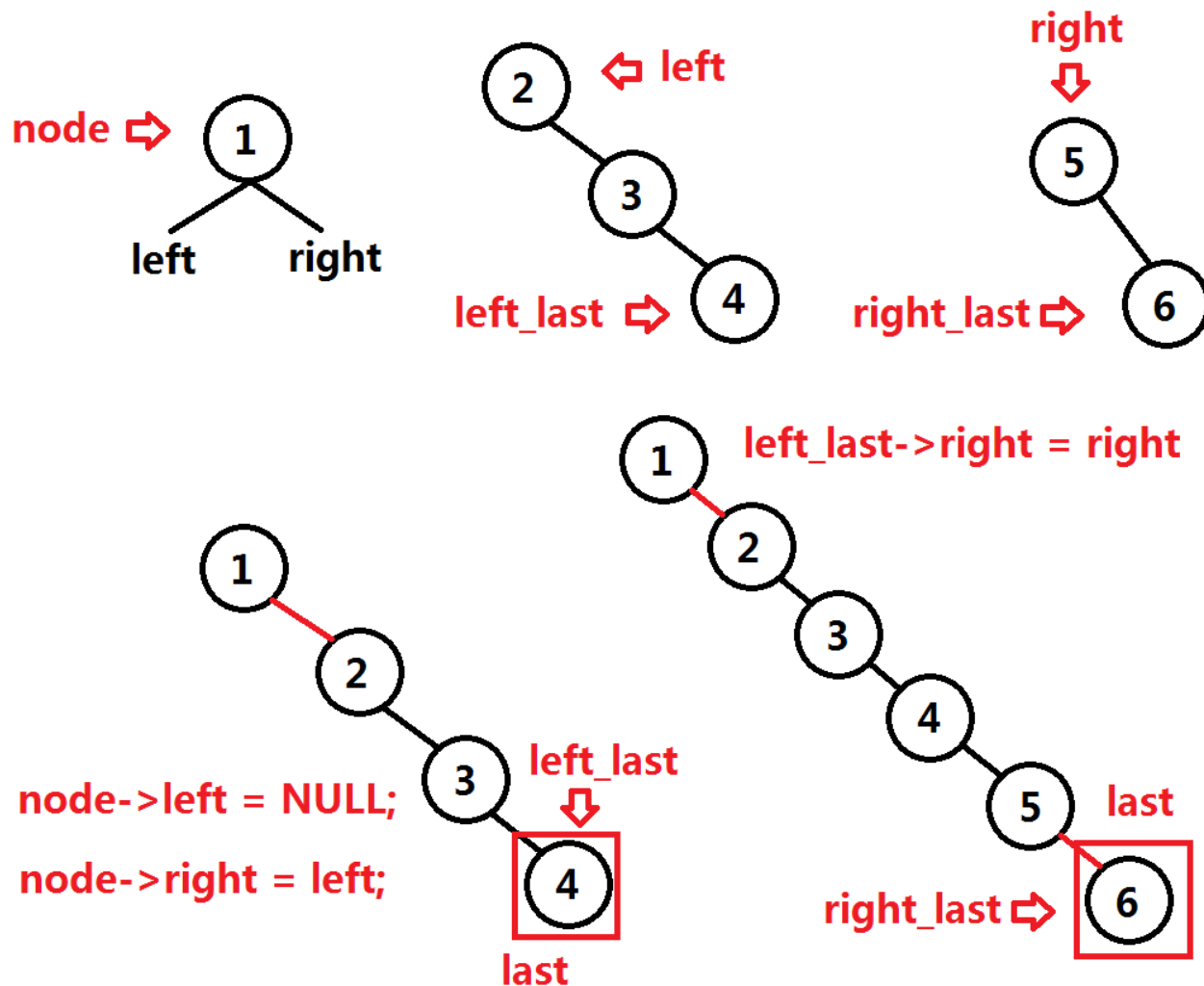


必要信息 即为链表尾部last指针  
●●●

# 例3:算法思路(拆解并解决子问题)



# 例3:算法思路(解决当前问题)



# 例3:方法2课堂练习

```
class Solution {
public:
    void flatten(TreeNode *root) {
        TreeNode *last = NULL;
        preorder(root, last);    //当前子树的先序遍历的
                                //最后一个节点，传引用会传出
    }
private:
    //当前的节点
```

```
void preorder(TreeNode *node, TreeNode *&last) {
```

```
    if (!node) {
        return;
    }
```

```
    if ( 1 ) {
        2
        return;
    }
```

```
    TreeNode *left = node->left;    //备份左右指针
```

```
    TreeNode *right = node->right;
```

```
    TreeNode *left_last = NULL;
```

```
    TreeNode *right_last = NULL;    //左右子树最后一个节点
```

```
    if (left) {
        preorder(left, left_last);    //若有左子树，递归将左子树转换单链表
        node->left = NULL;            //左指针赋空
```

```
        3
        last = left_last;    //将该节点的last保存为左子树的last
```

```
    }
    if (right) {    //若有右子树，递归将右子树转换单链表
```

```
        preorder(right, right_last);    //若node找到左子树最后一个节点
        if (left_last) {                (有左子树)
```

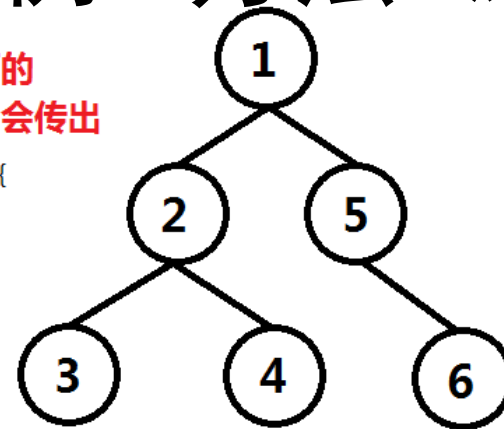
```
            4
```

```
        }
        5
    }
```

```
}
```

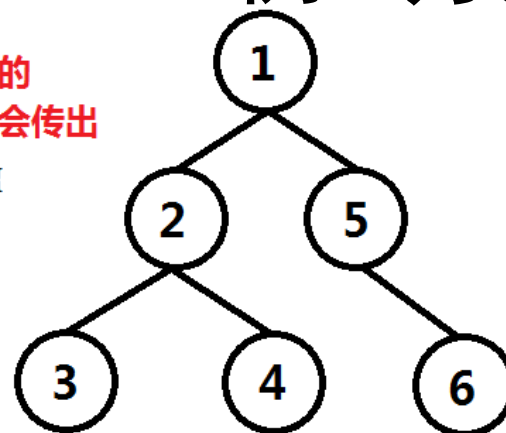
```
}
```

```
};
```



**5分钟**填写代码，  
**有问题随时提出！**

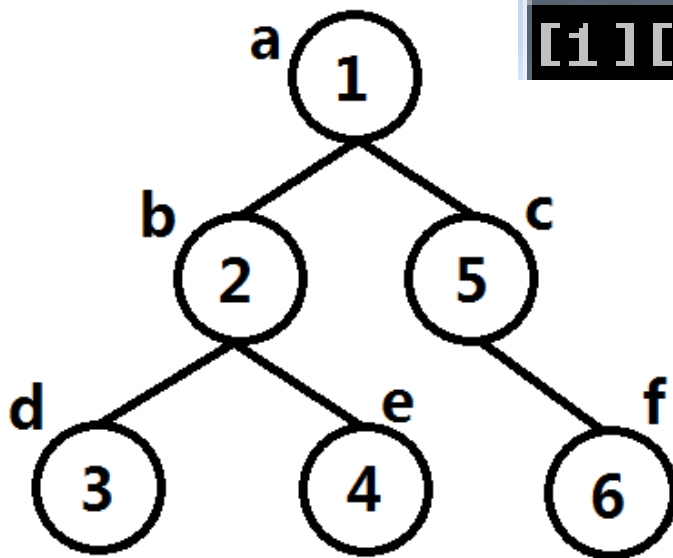
# 例3:方法2实现



```
class Solution {
public:
    void flatten(TreeNode *root) {
        TreeNode *last = NULL;
        preorder(root, last); //当前子树的先序遍历的
                               //最后一个节点，传引用会传出
    }
private:
    //当前的节点
    void preorder(TreeNode *node, TreeNode *&last) {
        if (!node) {
            return;
        }
        if (!node->left && !node->right) {
            last = node;
            return;
        }
        TreeNode *left = node->left; //备份左右指针
        TreeNode *right = node->right;
        TreeNode *left_last = NULL;
        TreeNode *right_last = NULL; //左右子树最后一个节点
        if (left) {
            preorder(left, left_last); //若有左子树，递归将左子树转换单链表
            node->left = NULL; //左指针赋空
            node->right = left;
            last = left_last; //将该节点的last保存为左子树的last
        }
        if (right) {
            preorder(right, right_last); //若有右子树，递归将右子树转换单链表
            //若node找到左子树最后一个节点
            if (left_last) {
                left_last->right = right;
            }
            last = right_last;
        }
    }
};
```

# 例3:测试与leetcode提交结果

```
int main() {
    TreeNode a(1);
    TreeNode b(2);
    TreeNode c(5);
    TreeNode d(3);
    TreeNode e(4);
    TreeNode f(6);
    a.left = &b;
    a.right = &c;
    b.left = &d;
    b.right = &e;
    c.right = &f;
    Solution solve;
    solve.flatten(&a);
    TreeNode *head = &a;
    while(head) {
        if (head->left) {
            printf("ERROR\n");
        }
        printf("[%d]", head->val);
        head = head->right;
    }
    printf("\n");
    return 0;
}
```



[1][2][3][4][5][6]

Flatten Binary Tree to Linked List

Submission Details

225 / 225 test cases passed.

Status: **Accepted**

Runtime: 6 ms

Submitted: 3 hours, 4 minutes ago

# 课间休息10分钟

---

## 有问题提出！

# 预备知识:二叉树层次遍历

二叉树**层次遍历**，又称为**宽度优先搜索**，按树的层次依次访问树的结点。层次遍历使用**队列**对遍历节点进行**存储**，先进入队列的结点，**优先遍历**拓展其左孩子与右孩子。

**设置队列 Q**

**将根节点 push 进Q**

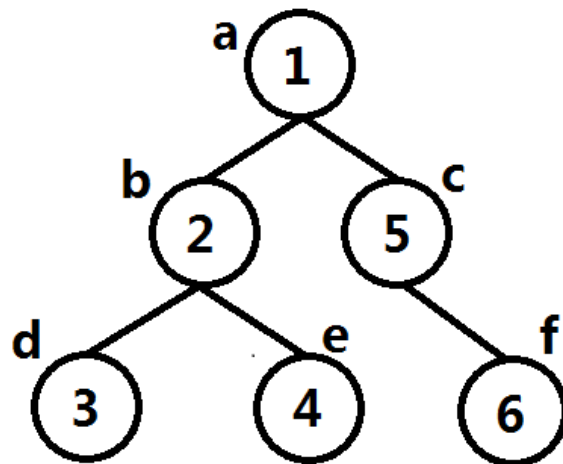
**while(Q不空){**

**取出队列头部节点node**

**对node访问**

**将node的左、右孩子push进队列**

**}**



层次遍历:

a(1), b(2), c(5), d(3), e(4), f(6)



# 预备知识:二叉树层次遍历

1. Q.push(a)

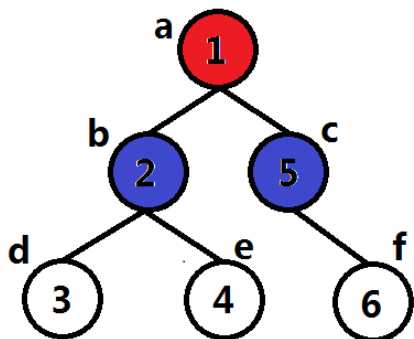
Q = [a]

search: a

Q.pop()

Q.push(b)

Q.push(c)



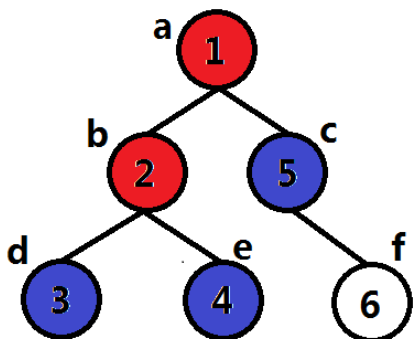
2. Q = [b, c]

search: b

Q.pop()

Q.push(d)

Q.push(e)

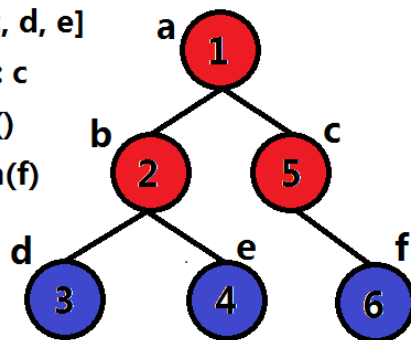


3. Q = [c, d, e]

search: c

Q.pop()

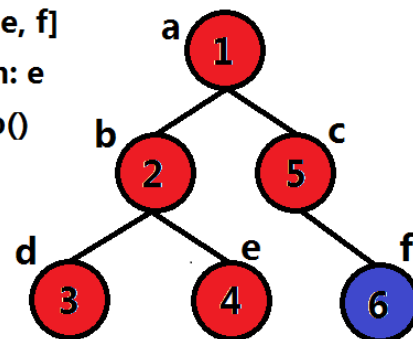
Q.push(f)



5. Q = [e, f]

search: e

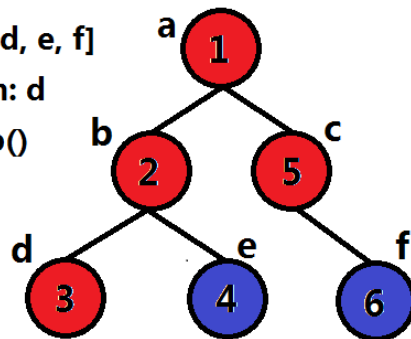
Q.pop()



4. Q = [d, e, f]

search: d

Q.pop()

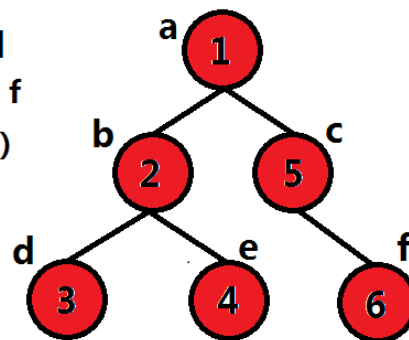


6. Q = [f]

search: f

Q.pop()

7. Q = []



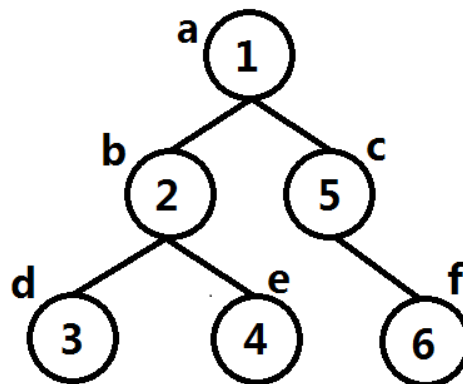
# 预备知识:二叉树层次遍历, 课堂练习

```
#include <stdio.h>
#include <vector>
#include <queue>

struct TreeNode { //二叉树数据结构
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

```
void BFS_print(TreeNode* root) { //宽度优先搜索二叉树
```

```
    std::queue<TreeNode> Q;
    [1]
    while ([2]) {
        TreeNode *node = Q.front();
        [3]
        printf("[%d]\n", node->val);
        if ([4]) {
            Q.push(node->left);
        }
        if (node->right) {
            [5]
        }
    }
}
```



层次遍历:

a(1), b(2), c(5), d(3), e(4), f(6)

```
int main() {
    TreeNode a(1);
    TreeNode b(2);
    TreeNode c(5);
    TreeNode d(3);
    TreeNode e(4);
    TreeNode f(6);
    a.left = &b;
    a.right = &c;
    b.left = &d;
    b.right = &e;
    c.right = &f;
    BFS_print(&a);
    return 0;
}
```

```
[1]
[2]
[5]
[3]
[4]
[6]
```

# 预备知识:二叉树层次遍历, 实现

```
#include <stdio.h>
#include <vector>
#include <queue>

struct TreeNode { //二叉树数据结构
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

```
void BFS_print(TreeNode* root) { //宽度优先搜索二叉树
```

```
    std::queue<TreeNode> Q;
```

```
    Q.push(root);
```

```
    while (!Q.empty()) {
```

```
        TreeNode *node = Q.front();
```

```
        Q.pop();
```

```
        printf("[%d]\n", node->val);
```

```
        if (node->left) {
```

```
            Q.push(node->left);
```

```
        }
```

```
        if (node->right) {
```

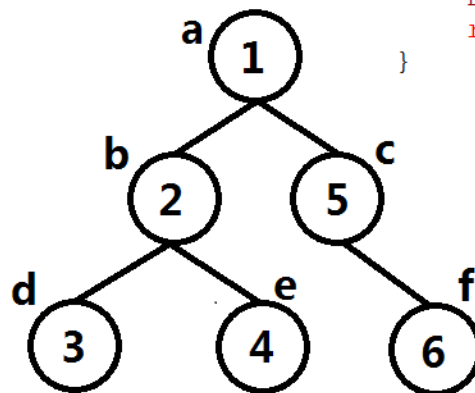
```
            Q.push(node->right);
```

```
        }
```

```
    }
```

```
}
```

```
int main() {
    TreeNode a(1);
    TreeNode b(2);
    TreeNode c(5);
    TreeNode d(3);
    TreeNode e(4);
    TreeNode f(6);
    a.left = &b;
    a.right = &c;
    b.left = &d;
    b.right = &e;
    c.right = &f;
    BFS_print(&a);
    return 0;
}
```



层次遍历:

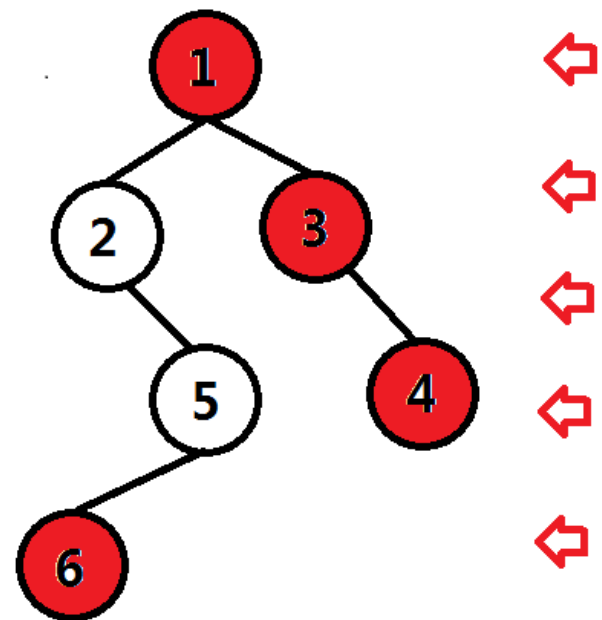
a(1), b(2), c(5), d(3), e(4), f(6)

```
[1]
[2]
[5]
[3]
[4]
[6]
```

# 例4:侧面观察二叉树

给定一个**二叉树**，假设从该二叉树的**右侧**观察它，将观察到的节点按照**从上到下**的顺序输出。

```
struct TreeNode {  
    int val;  
    TreeNode *left;  
    TreeNode *right;  
    TreeNode(int x) :  
        val(x), left(NULL), right(NULL) {}  
};  
class Solution {  
public:  
    std::vector<int> rightSideView(TreeNode* root) {  
    }  
};  
[1, 3, 4, 6]
```



选自 **LeetCode 199. Binary Tree Right Side View**

<https://leetcode.com/problems/binary-tree-right-side-view/description/>

难度:**Medium**

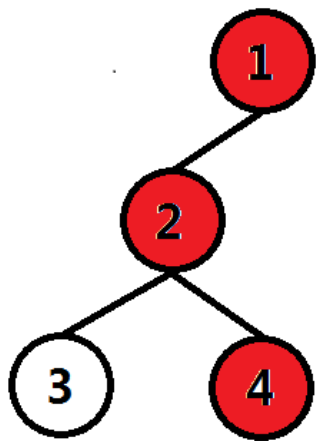
# 例4:思考与分析

从二叉树的**右侧**观察它，将观察到的节点按照**从上到下**的顺序输出，就是求**层次遍历二叉树**，每个层中的**最后一个节点**。

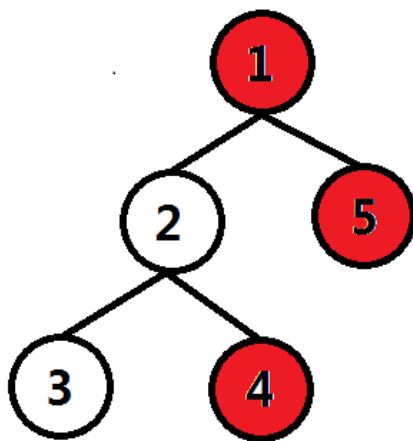
第0层

第1层

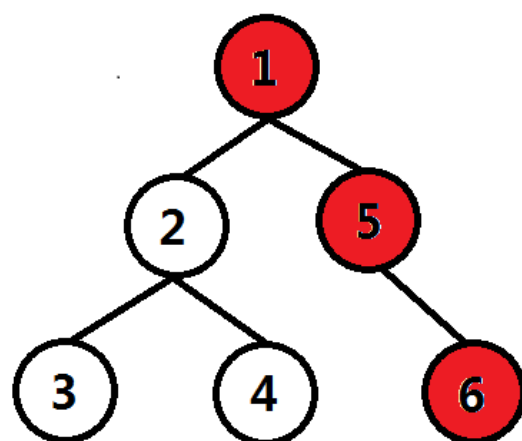
第2层



[1,2,4]



[1,5,4]



[1,5,6]

**思考:层次遍历**时，如何记录每一层中出现的**最后一个节点**？

# 例4:算法思路

**层次遍历**时，将**节点与层数**绑定为pair，压入队列时，将节点与层数**同时**压入队列，并记录每一层中出现的**最后一个节点**。

在**层次遍历**中，每一层中的**最后一个节点最后遍历到**，**随时更新**对每层的最后一个节点即可。

1.  $Q.push(<a, 0>)$

$Q = [<a, 0>]$

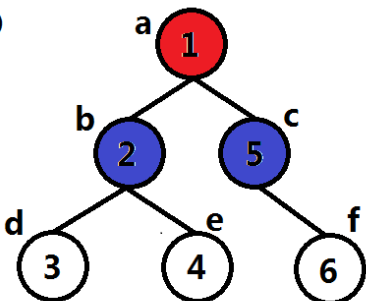
search: a

$Q.pop()$

$Q.push(<b, 1>)$

$Q.push(<c, 1>)$

$view[0] = 1(a)$



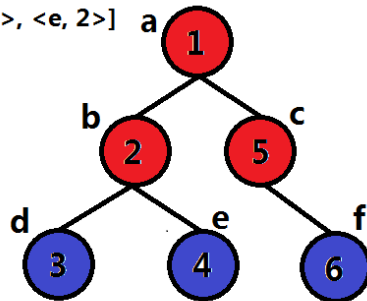
3.  $Q = [<c, 1>, <d, 2>, <e, 2>]$

search: c

$Q.pop()$

$Q.push(<f, 2>)$

$view[1] = 5(c)$

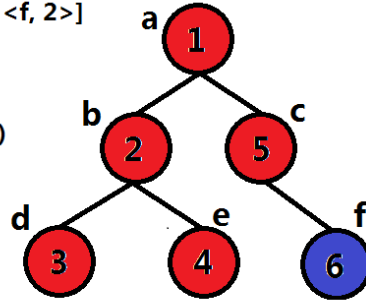


5.  $Q = [<e, 2>, <f, 2>]$

search: e

$Q.pop()$

$view[2] = 4(e)$



2.  $Q = [<b, 1>, <c, 1>]$

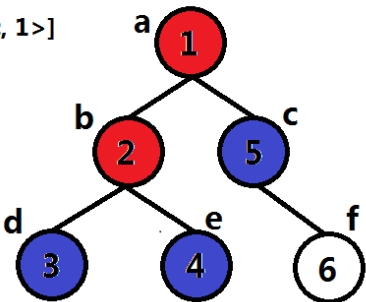
search: b

$Q.pop()$

$Q.push(<d, 2>)$

$Q.push(<e, 2>)$

$view[1] = 2(b)$

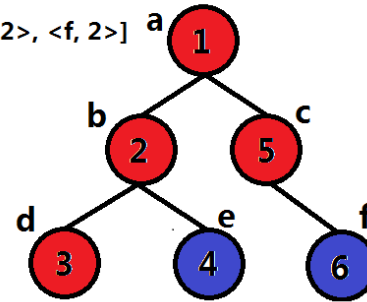


4.  $Q = [<d, 2>, <e, 2>, <f, 2>]$

search: d

$Q.pop()$

$view[2] = 3(d)$

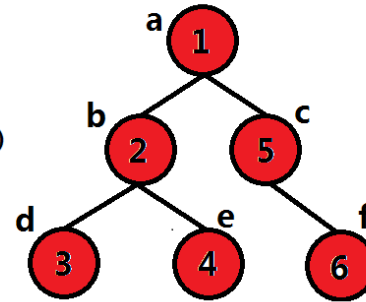


6.  $Q = [<f, 2>]$

search: f

$Q.pop()$

$view[2] = 6(f)$



7.  $Q = []$

# 例4:课堂练习

```
class Solution {
public:
    std::vector<int> rightSideView(TreeNode* root) {
        std::vector<int> view; //按层遍历的最后一个节点
        std::queue<std::pair<TreeNode *, int> > Q;
            //宽度优先搜索队列<节点, 层数>

        if (root) {
            Q.push(std::make_pair(root, 0));
        } //根节点非空时, 将<root, 0> push进入队列

        while (!Q.empty()) {
            TreeNode *node = Q.front().first; //搜索节点
            int depth = Q.front().second; //待搜索节点的层数
            Q.pop();
            if ( 1 ) {
                view.push_back(node->val);
            }
            else {
                2
            }
            if ( 3 ) {
                Q.push(std::make_pair(node->left, 4 ));
            }
            if (node->right) {
                Q.push(std::make_pair(node->right, 5 ));
            }
        }
        return view;
    }
};
```

5分钟填写代码,  
有问题随时提出!

# 例4:实现

```
class Solution {
public:
    std::vector<int> rightSideView(TreeNode* root) {
        std::vector<int> view; //按层遍历的最后一个节点
        std::queue<std::pair<TreeNode *, int> > Q;
        //宽度优先搜索队列<节点, 层数>

        if (root) {
            Q.push(std::make_pair(root, 0));
        }
        //根节点非空时, 将<root, 0> push进入队列

        while (!Q.empty()) {
            TreeNode *node = Q.front().first; //搜索节点
            int depth = Q.front().second; //待搜索节点的层数
            Q.pop();
            if (view.size() == depth) {
                view.push_back(node->val);
            }
            else {
                view[depth] = node->val;
            }
            if (node->left) {
                Q.push(std::make_pair(node->left, depth + 1));
            }
            if (node->right) {
                Q.push(std::make_pair(node->right, depth + 1));
            }
        }
        return view;
    }
};
```

1. Q.push(<a, 0>)

Q = [<a, 0>]

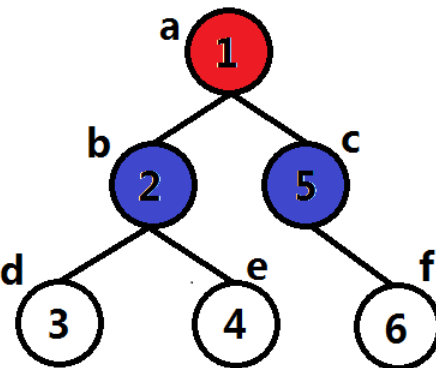
search: a

Q.pop()

Q.push(<b, 1>)

Q.push(<c, 1>)

view[0] = 1(a)



2. Q = [<b, 1>, <c, 1>]

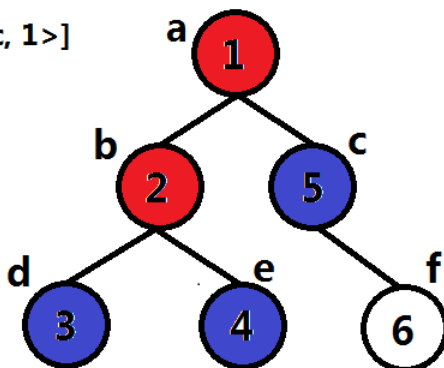
search: b

Q.pop()

Q.push(<d, 2>)

Q.push(<e, 2>)

view[1] = 2(b)



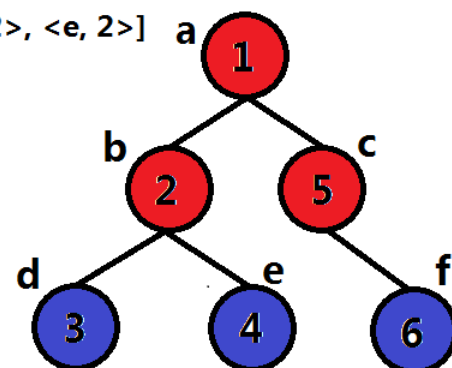
3. Q = [<c, 1>, <d, 2>, <e, 2>]

search: c

Q.pop()

Q.push(<f, 2>)

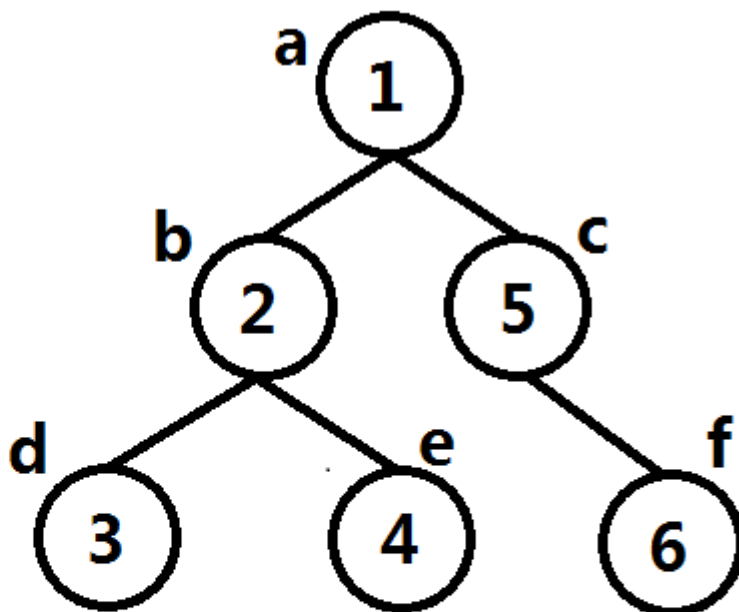
view[1] = 5(c)





# 例4:测试与leetcode提交结果

```
int main() {
    TreeNode a(1);
    TreeNode b(2);
    TreeNode c(5);
    TreeNode d(3);
    TreeNode e(4);
    TreeNode f(6);
    a.left = &b;
    a.right = &c;
    b.left = &d;
    b.right = &e;
    c.right = &f;
    Solution solve;
    std::vector<int> result = solve.rightSideView(&a);
    for (int i = 0; i < result.size(); i++) {
        printf("[%d]\n", result[i]);
    }
    return 0;
}
```



```
[1]
[5]
[6]
```

Binary Tree Right Side View

Submission Details

210 / 210 test cases passed.

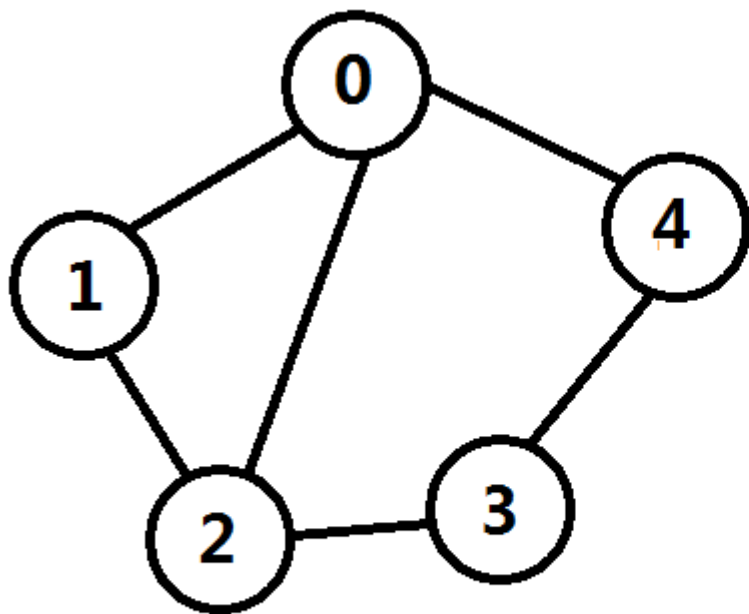
Status: Accepted

Runtime: 3 ms

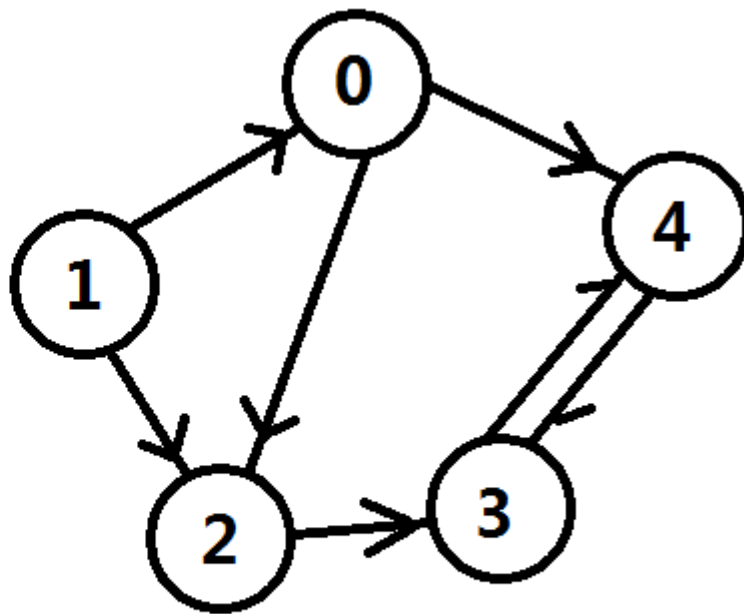
Submitted: 0 minutes ago

# 预备知识:图的定义

**图 (Graph)** 是由**顶点**的有穷非空集合和**顶点之间边**的集合组成，通常表示为： $G(V, E)$ ，其中， $G$ 表示一个图， $V$ 是图 $G$ 中**顶点**的集合， $E$ 是图 $G$ 中**边**的集合。图分**无向图**与**有向图**，根据图的边长，又分**带权图**与**不带权图**。



无向图



有向图

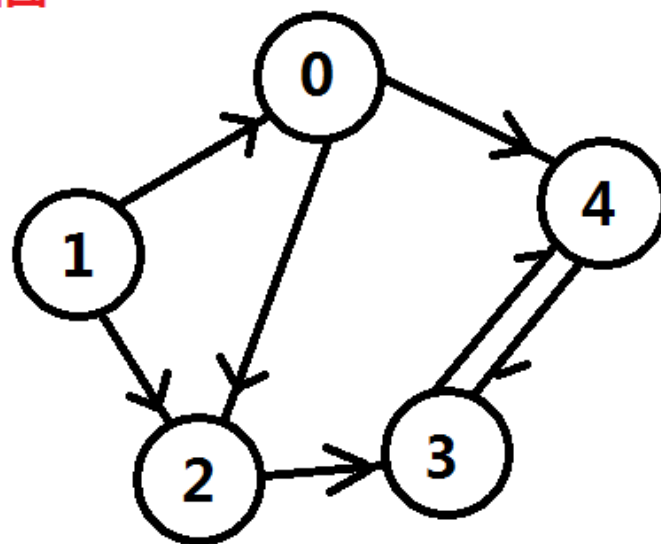
# 预备知识:图的构造与表示(邻接矩阵)

```
#include <stdio.h>
```

```
int main() {  
    const int MAX_N = 5; //一共5个顶点  
    int Graph[MAX_N][MAX_N] = {0}; //使用邻接矩阵表示  
    Graph[0][2] = 1;  
    Graph[0][4] = 1; //将图连通, 且不带权  
    Graph[1][0] = 1;  
    Graph[1][2] = 1; //一般用邻接矩阵表示稠密图  
    Graph[2][3] = 1;  
    Graph[3][4] = 1;  
    Graph[4][3] = 1;  
    printf("Graph:\n");  
    for (int i = 0; i < MAX_N; i++) {  
        for (int j = 0; j < MAX_N; j++) {  
            printf("%d ", Graph[i][j]);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

Graph:

0	0	1	0	1
1	0	1	0	0
0	0	0	1	0
0	0	0	0	1
0	0	0	1	0



```
#include <stdio.h>
#include <vector>
```

//图的邻接表数据结构

```
struct GraphNode{
    int label; //图的顶点的值 //相邻节点指针数组
    std::vector<GraphNode *> neighbors;
    GraphNode(int x) : label(x) {};
};
```

```
int main(){
    const int MAX_N = 5;
    GraphNode *Graph[MAX_N]; //5个顶点
```

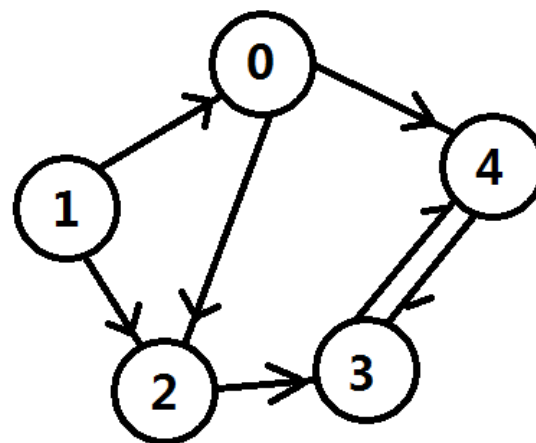
```
    for (int i = 0; i < MAX_N; i++){
        Graph[i] = new GraphNode(i);
    } //添加边
```

```
    Graph[0]->neighbors.push_back(Graph[2]);
    Graph[0]->neighbors.push_back(Graph[4]);
    Graph[1]->neighbors.push_back(Graph[0]);
    Graph[1]->neighbors.push_back(Graph[2]);
    Graph[2]->neighbors.push_back(Graph[3]);
    Graph[3]->neighbors.push_back(Graph[4]);
    Graph[4]->neighbors.push_back(Graph[3]);
```

```
    printf("Graph:\n");
    for (int i = 0; i < MAX_N; i++){
        printf("Label(%d) : ", i);
        for (int j = 0; j < Graph[i]->neighbors.size(); j++){
            printf("%d ", Graph[i]->neighbors[j]->label);
        }
        printf("\n");
    }
    for (int i = 0; i < MAX_N; i++){
        delete Graph[i];
    }
```

```
    return 0;
}
```

# 预备知识:图的构造与表示(邻接表)



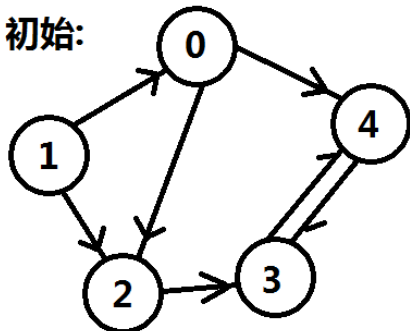
有向图

```
Graph:
Label(0) : 2 4
Label(1) : 0 2
Label(2) : 3
Label(3) : 4
Label(4) : 3
```

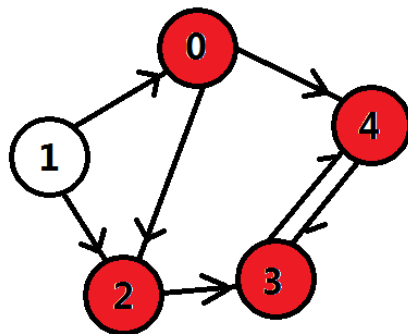
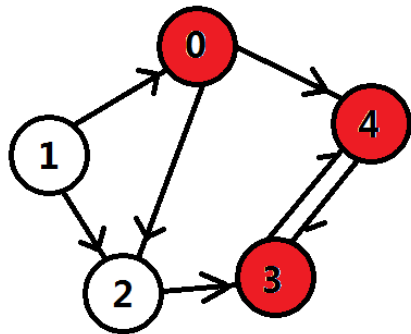
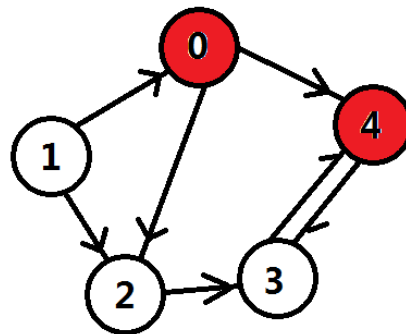
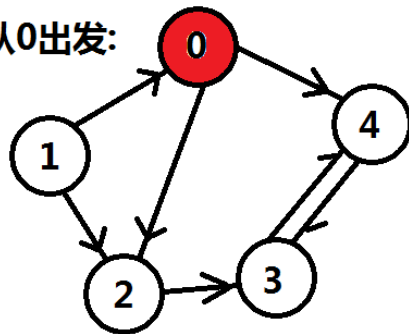
# 预备知识:图的深度优先遍历

从图中**某个顶点v**出发，首先访问该顶点，然后**依次**从它的各个未被访问的**邻接点**出发**深度优先搜索**遍历图，直至图中所有和v有**路径相通且未被访问**的顶点都被访问到。若此时尚有其他顶点**未被访问**到，则另选一个未被访问的顶点作**起始点**，**重复**上述过程，直至图中**所有顶点**都被访问到为止。

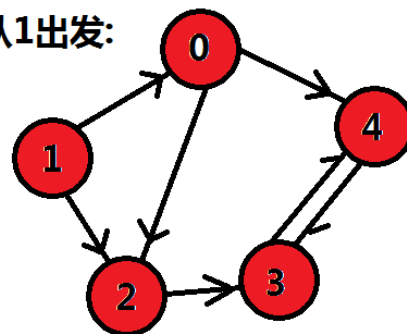
初始:



从0出发:



从1出发:



```

#include <stdio.h>
#include <vector> //图的邻接表数据结构

struct GraphNode{ //图的顶点的值 //相邻节点指针数组
    int label;
    std::vector<GraphNode*> neighbors;
    GraphNode(int x) : label(x) {}
};

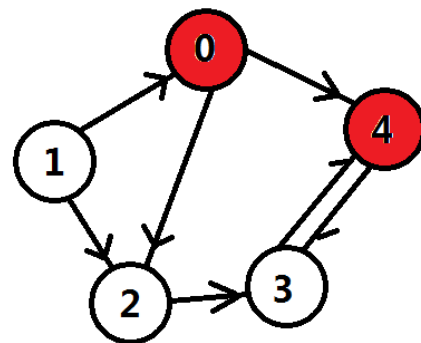
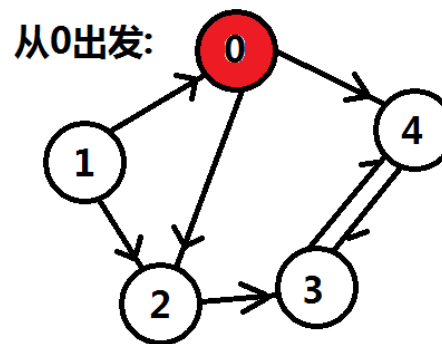
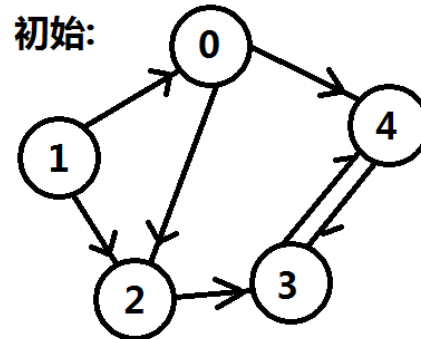
void DFS_graph(GraphNode *node, int visit[]){
    1
    printf("%d ", node->label);
    for (int i = 0; i < node->neighbors.size(); i++){
        if ( 2 ){
            DFS_graph(node->neighbors[i], visit);
        }
    }
}

int main(){
    const int MAX_N = 5; //创建图的顶点
    GraphNode *Graph[MAX_N];
    for (int i = 0; i < MAX_N; i++){
        Graph[i] = new GraphNode(i);
    }
    //添加图的边，注意添加边的顺序
    Graph[0]->neighbors.push_back(Graph[4]);
    Graph[0]->neighbors.push_back(Graph[2]);
    Graph[1]->neighbors.push_back(Graph[0]);
    Graph[1]->neighbors.push_back(Graph[2]);
    Graph[2]->neighbors.push_back(Graph[3]);
    Graph[3]->neighbors.push_back(Graph[4]);
    Graph[4]->neighbors.push_back(Graph[3]);

    int visit[MAX_N] = {0}; //标记已访问的顶点
    for (int i = 0; i < MAX_N; i++){
        if ( 3 ){
            printf("From label(%d) : ", Graph[i]->label);
            DFS_graph(Graph[i], visit);
            printf("\n");
        }
    }
    for (int i = 0; i < MAX_N; i++){
        delete Graph[i];
    }

    return 0;
}

```



预备知识:  
图的深度  
优先遍历  
，课堂练习

3分钟填写代码，  
有问题随时提出！

```

From label(0) : 0 4 3 2
From label(1) : 1

```

```

#include <stdio.h>
#include <vector> //图的邻接表数据结构

struct GraphNode{ //图的顶点的值 //相邻节点指针数组
    int label;
    std::vector<GraphNode*> neighbors;
    GraphNode(int x) : label(x) {}
};

void DFS_graph(GraphNode *node, int visit[]){
    visit[node->label] = 1; //标记已访问的顶点
    printf("%d ", node->label); //访问相邻的且没有被访问的顶点
    for (int i = 0; i < node->neighbors.size(); i++){
        if (visit[node->neighbors[i]->label] == 0){
            DFS_graph(node->neighbors[i], visit);
        }
    }
}

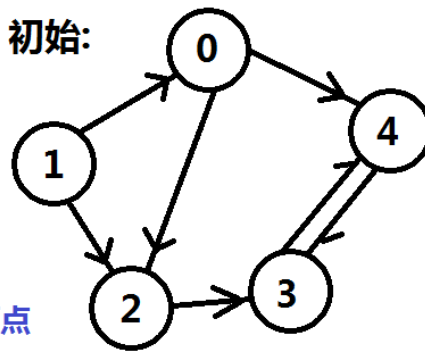
int main(){
    const int MAX_N = 5; //创建图的顶点
    GraphNode *Graph[MAX_N];
    for (int i = 0; i < MAX_N; i++){
        Graph[i] = new GraphNode(i);
    }
    //添加图的边，注意添加边的顺序
    Graph[0]->neighbors.push_back(Graph[4]);
    Graph[0]->neighbors.push_back(Graph[2]);
    Graph[1]->neighbors.push_back(Graph[0]);
    Graph[1]->neighbors.push_back(Graph[2]);
    Graph[2]->neighbors.push_back(Graph[3]);
    Graph[3]->neighbors.push_back(Graph[4]);
    Graph[4]->neighbors.push_back(Graph[3]);

    int visit[MAX_N] = {0}; //标记已访问的顶点
    for (int i = 0; i < MAX_N; i++){
        if (visit[i] == 0){ //顶点没有被标记才会访问
            printf("From label(%d) : ", Graph[i]->label);
            DFS_graph(Graph[i], visit);
            printf("\n");
        }
    }
    for (int i = 0; i < MAX_N; i++){
        delete Graph[i];
    }

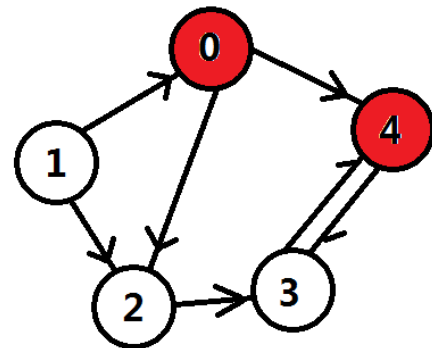
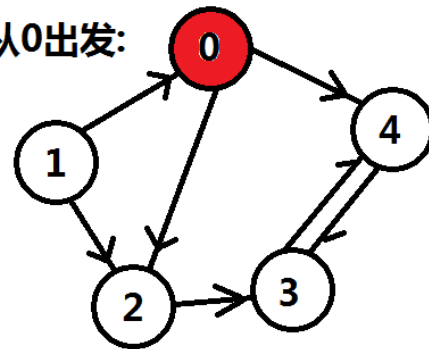
    return 0;
}

```

初始:



从0出发:



预备知识:  
图的深度  
优先遍历,  
实现

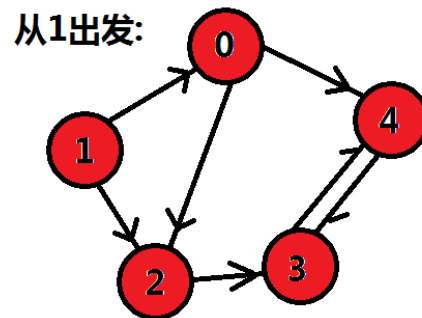
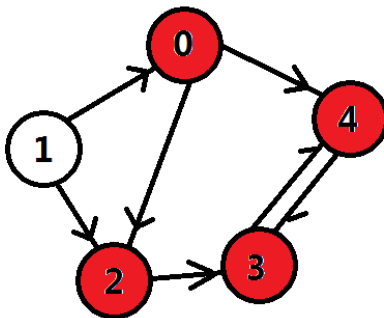
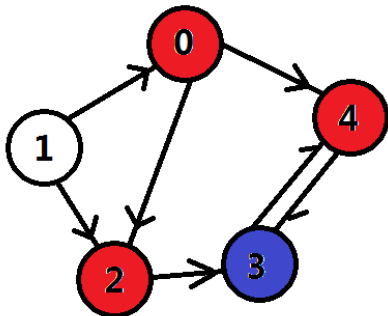
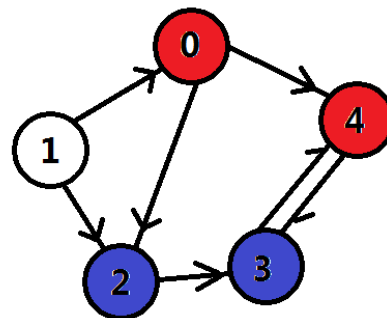
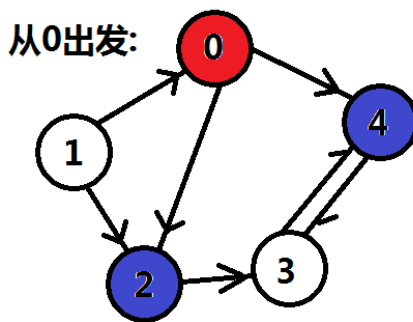
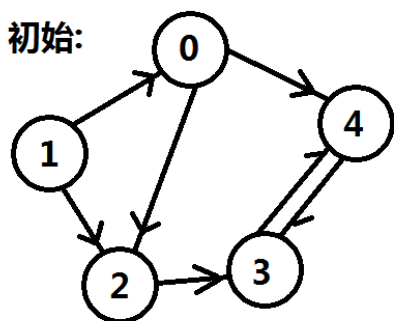
```

From label(0) : 0 4 3 2
From label(1) : 1

```

# 预备知识:图的宽度优先遍历

从图中**某个顶点v**出发，在访问了v之后**依次访问**v的各个**未曾访问过的**邻接点，然后**分别**从这些邻接点出发**依次访问**它们的邻接点，并使得“**先被访问**的顶点的邻接点**先于后被访问**的顶点的邻接点被访问”，直至图中所有**已被访问的顶点的邻接点**都被访问到。如果此时图中尚有顶点**未被访问**，则需要**另选**一个未曾被访问过的顶点作为新的起始点，**重复**上述过程，直至图中**所有顶点**都被访问到为止。

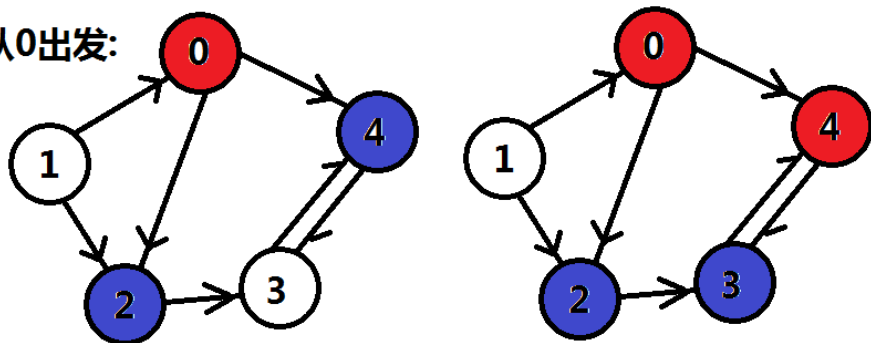




# 预备知识:图的宽度优先遍历, 课堂练习

```
void BFS_graph(GraphNode *node, int visit[]){
    std::queue<GraphNode *> Q;
    Q.push(node);
    visit[node->label] = 1;
    while( 1 ){
        GraphNode *node = Q.front();
        Q.pop();
        printf("%d ", node->label);
        for (int i = 0; i < node->neighbors.size(); i++){
            if ( 2 ){
                3
                visit[node->neighbors[i]->label] = 1;
            }
        }
    }
}
```

从0出发:



```
int main(){
    const int MAX_N = 5; //创建图的顶点
    GraphNode *Graph[MAX_N];
    for (int i = 0; i < MAX_N; i++){
        Graph[i] = new GraphNode(i);
    }
    //添加图的边, 注意添加边的顺序
    Graph[0]->neighbors.push_back(Graph[4]);
    Graph[0]->neighbors.push_back(Graph[2]);
    Graph[1]->neighbors.push_back(Graph[0]);
    Graph[1]->neighbors.push_back(Graph[2]);
    Graph[2]->neighbors.push_back(Graph[3]);
    Graph[3]->neighbors.push_back(Graph[4]);
    Graph[4]->neighbors.push_back(Graph[3]);

    int visit[MAX_N] = {0}; //标记已访问的顶点
    for (int i = 0; i < MAX_N; i++){
        if (visit[i] == 0){
            printf("From label(%d) : ", Graph[i]->label);
            BFS_graph(Graph[i], visit);
            printf("\n");
        }
    }

    for (int i = 0; i < MAX_N; i++){
        delete Graph[i];
    }

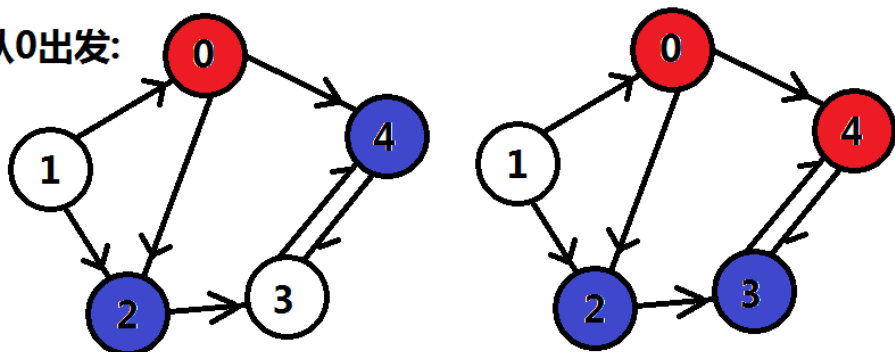
    return 0;
}
```

```
From label(0) : 0 4 2 3
From label(1) : 1
```

# 预备知识:图的宽度优先遍历, 实现

```
void BFS_graph(GraphNode *node, int visit[]){
    std::queue<GraphNode *> Q;
    Q.push(node);
    visit[node->label] = 1; //宽度优先搜索使用队列,
    while ( !Q.empty() ) { 队列不空即一直循环
        GraphNode *node = Q.front();
        Q.pop();
        printf("%d ", node->label);
        for (int i = 0; i < node->neighbors.size(); i++){
            if ( visit[node->neighbors[i]->label] == 0 ) {
                Q.push(node->neighbors[i]);
                visit[node->neighbors[i]->label] = 1;
            }
        }
    }
}
```

从0出发:



```
int main(){
    const int MAX_N = 5; //创建图的顶点
    GraphNode *Graph[MAX_N];
    for (int i = 0; i < MAX_N; i++){
        Graph[i] = new GraphNode(i);
    }
    //添加图的边, 注意添加边的顺序
    Graph[0]->neighbors.push_back(Graph[4]);
    Graph[0]->neighbors.push_back(Graph[2]);
    Graph[1]->neighbors.push_back(Graph[0]);
    Graph[1]->neighbors.push_back(Graph[2]);
    Graph[2]->neighbors.push_back(Graph[3]);
    Graph[3]->neighbors.push_back(Graph[4]);
    Graph[4]->neighbors.push_back(Graph[3]);

    int visit[MAX_N] = {0}; //标记已访问的顶点
    for (int i = 0; i < MAX_N; i++){
        if (visit[i] == 0){
            printf("From label(%d) : ", Graph[i]->label);
            BFS_graph(Graph[i], visit);
            printf("\n");
        }
    }

    for (int i = 0; i < MAX_N; i++){
        delete Graph[i];
    }

    return 0;
}
```

```
From label(0) : 0 4 2 3
From label(1) : 1
```

# 例5:课程安排

已知有n个课程，标记从0至n-1，**课程之间是有依赖**关系的，例如希望完成A课程，可能需要先完成B课程。已知n个课程的依赖关系，求**是否可以**将n个课程**全部完成**。

```
class Solution {  
public:                                //课程数量  
    bool canFinish(int numCourses,  
                   std::vector<std::pair<int, int> >& prerequisites) {  
    }  
};                                     //课程依赖关系<课程1, 课程2> , 代表课程1依赖课程2
```

如: 2, [[1, 0]] , 函数返回true.

2, [[1, 0], [0, 1]] , 函数返回false.

选自 **LeetCode 207. Course Schedule**

<https://leetcode.com/problems/course-schedule/description/>

难度:**Medium**

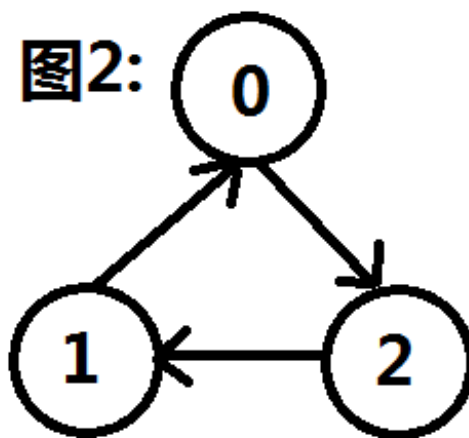
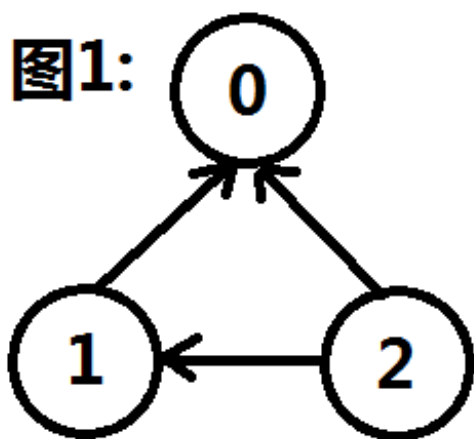
# 例5:分析

n个课程，它们之间有m个依赖关系，可以看成**顶点**个数为n，**边**个数为m的**有向图**。

图1:  $n = 3$ ,  $m = [[0, 1], [0, 2], [1, 2]]$ ; **可以**完成。

图2:  $n = 3$ ,  $m = [[0, 1], [1, 2], [2, 0]]$ ; **不可以**完成。

故，若**有向图无环**，则可以完成全部课程，否则不能。问题转换成，构建图，并**判断图是否有环**。



# 例5:方法1，深度优先搜索

在**深度优先搜索**时，如果正在搜索某一顶点(还未退出该顶点的递归深度搜索)，又**回到了**该顶点，即证明图有环。

如下图：

图1:

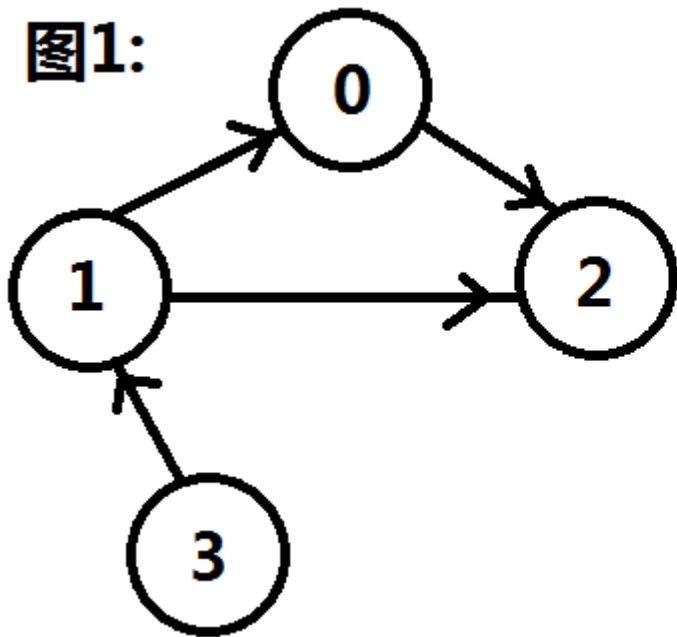
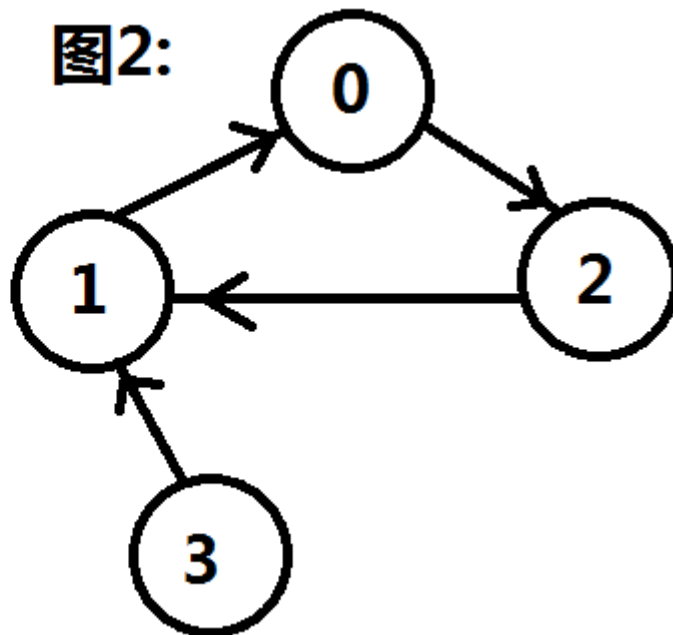


图2:



# 例5:算法思路(无环)

初始:

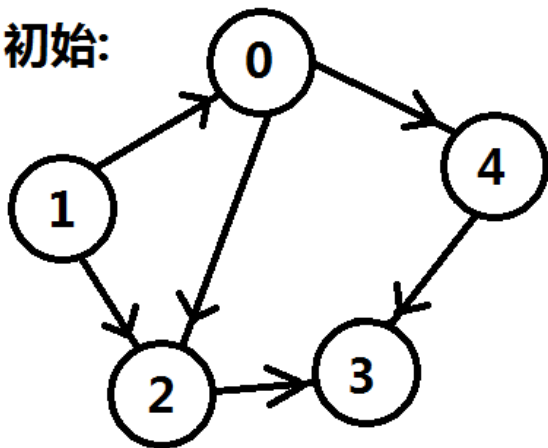


图1:

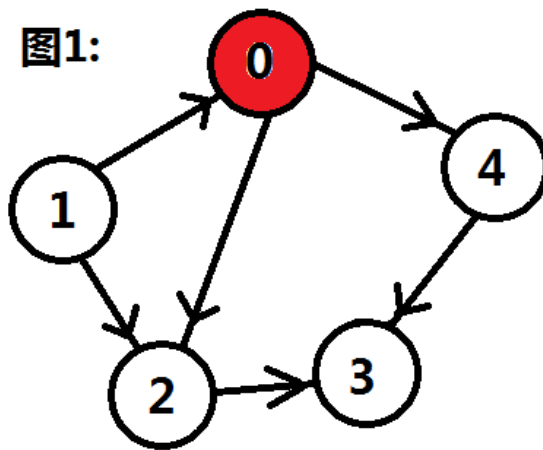


图2:

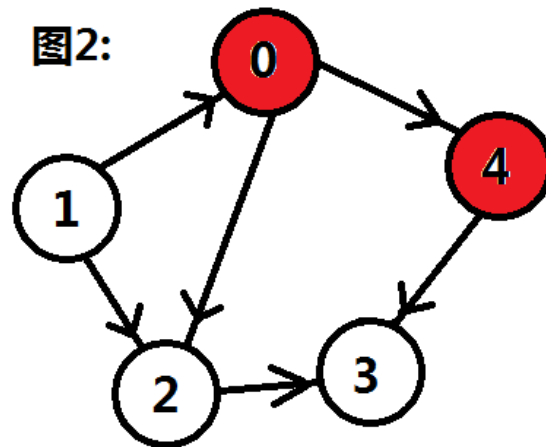


图3:

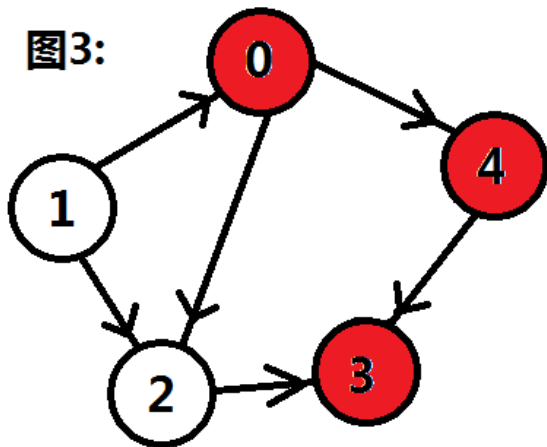


图4:

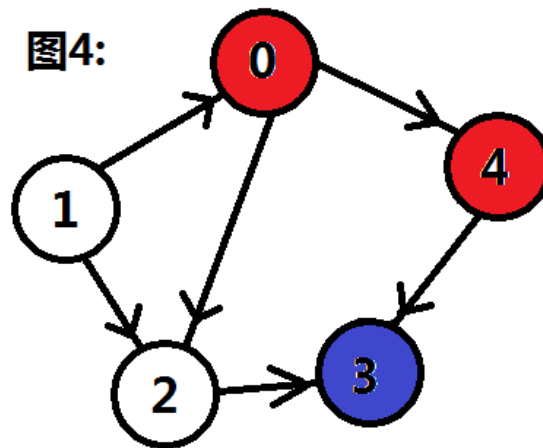
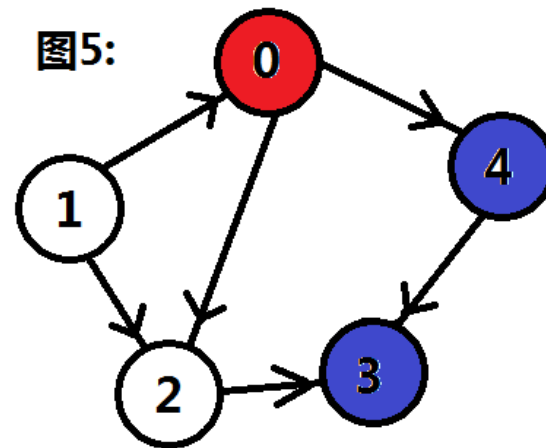
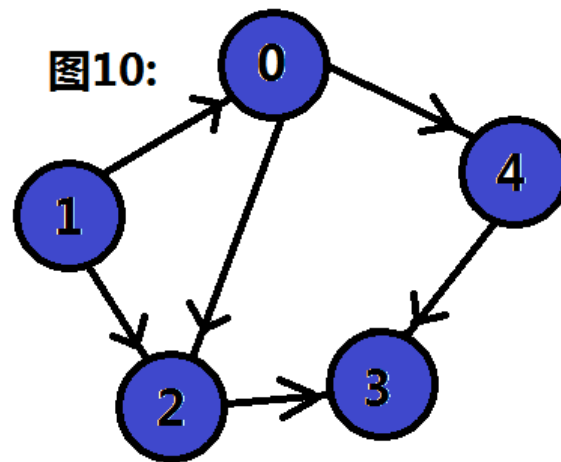
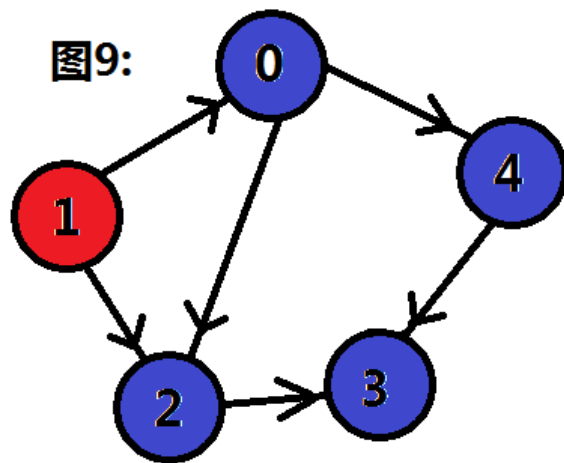
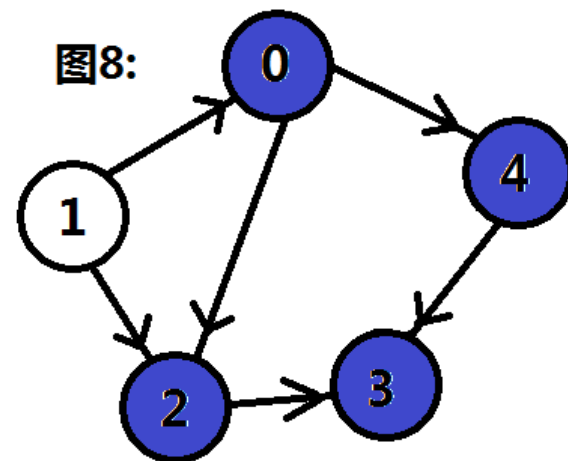
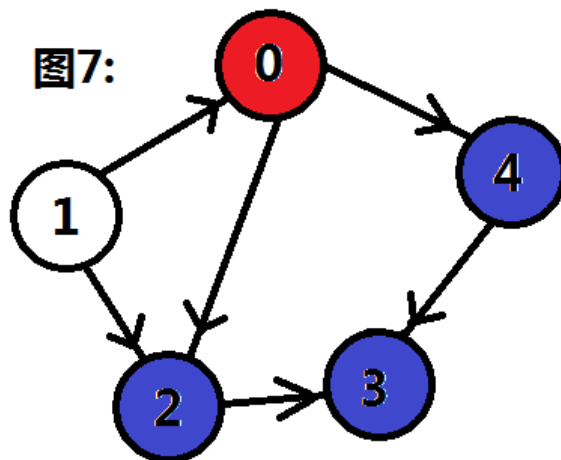
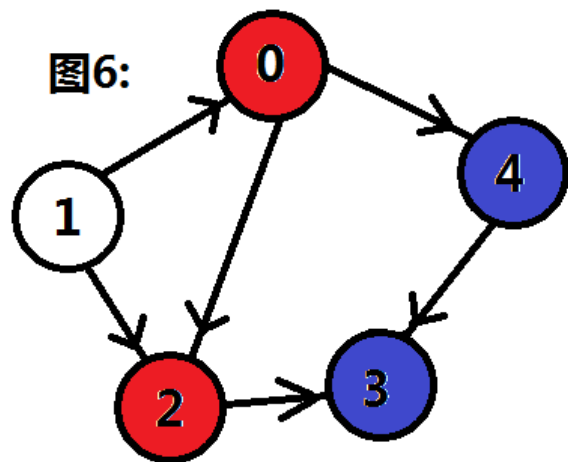


图5:



# 例5:算法思路(无环)



# 例5:算法思路(有环)

初始:

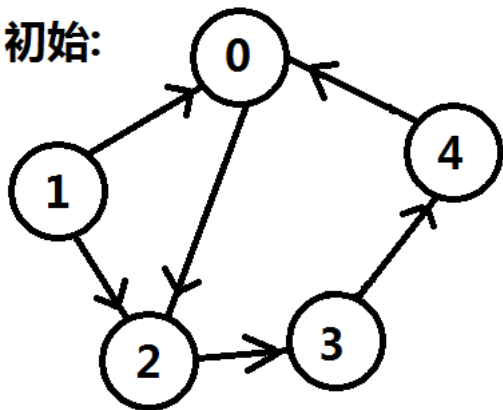


图1:

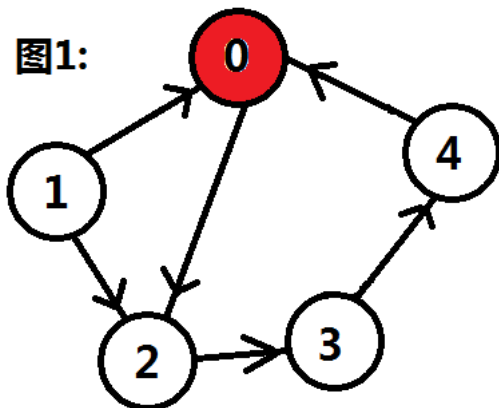


图2:

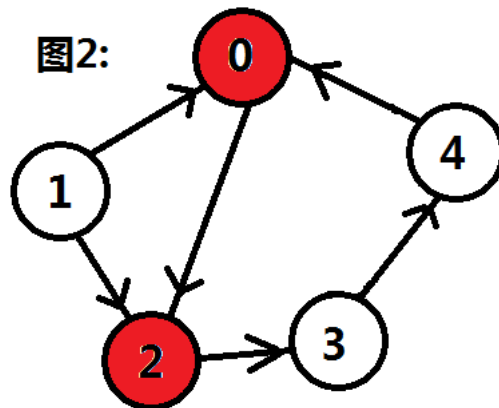


图3:

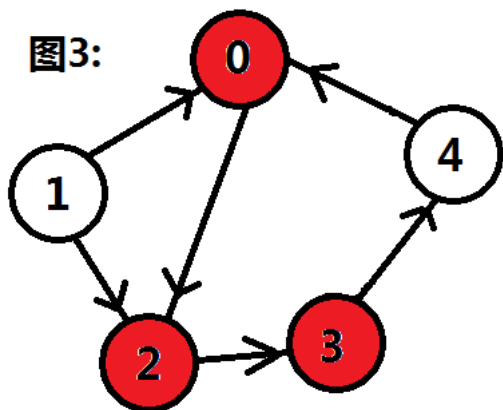


图4:

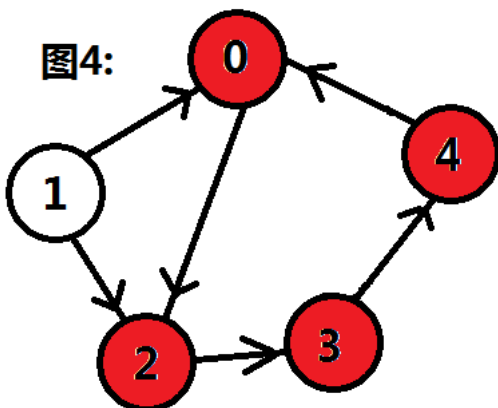
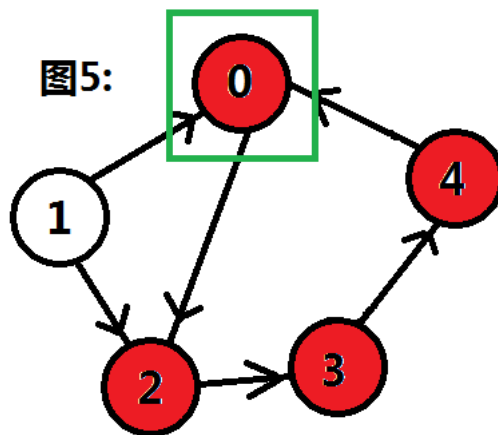


图5:





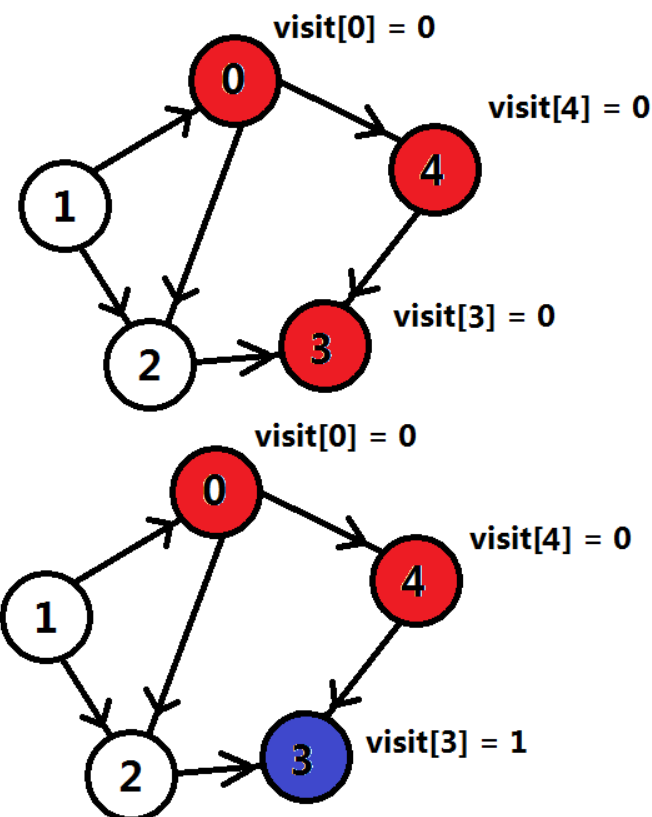
# 例5:方法1，调用代码

```
class Solution {
public:
    pair<课程1, 课程2> 课程1依赖课程2
    bool canFinish(int numCourses,
        std::vector<std::pair<int, int> >& prerequisites) {
        std::vector<GraphNode*> graph; //邻接表
        std::vector<int> visit; //节点访问状态,-1没有访问过,0代表正在访问,1代表已完成访问
        for (int i = 0; i < numCourses; i++){
            graph.push_back(new GraphNode(i)); //创建图的节点,并赋访问状态为空
            visit.push_back(-1);
        } //创建图,连接图的顶点
        for (int i = 0; i < prerequisites.size(); i++){
            GraphNode *begin = graph[prerequisites[i].second];
            GraphNode *end = graph[prerequisites[i].first];
            begin->neighbors.push_back(end); //课程2指向课程1
        }
        for (int i = 0; i < graph.size(); i++){
            if (visit[i] == -1 && !DFS_graph(graph[i], visit)){
                return false; //如果节点没访问过,进行DFS,如果DFS遇到环,
            }
        } 返回无法完成
        for (int i = 0; i < numCourses; i++){
            delete graph[i];
        }
        return true; //返回可以完成
    }
};
```

# 例5:方法1，课堂练习

```
#include <vector>
struct GraphNode{
    int label;
    std::vector<GraphNode*> neighbors;
    GraphNode(int x) : label(x) {}
};

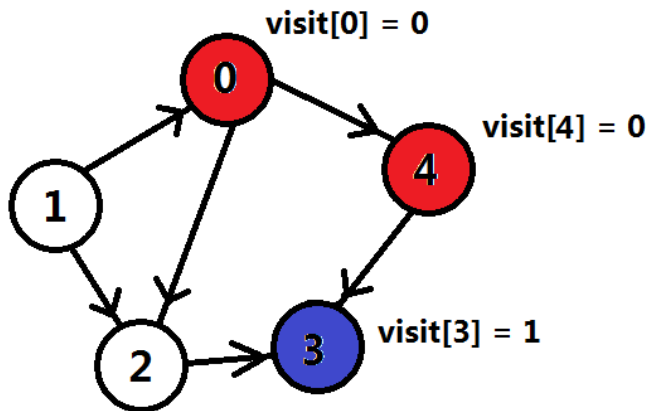
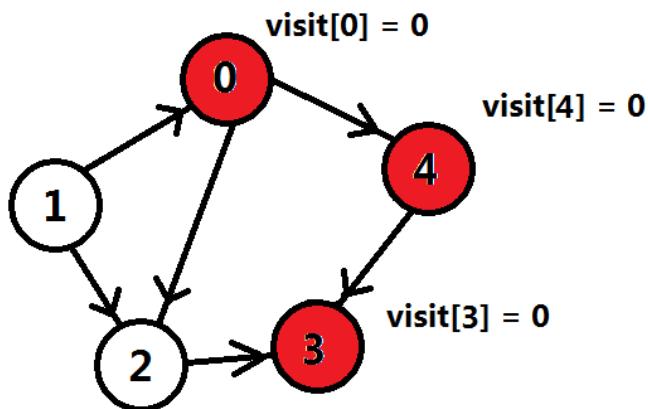
//节点访问状态,-1没有访问过,0代表正在访问,1代表已完成访问
bool DFS_graph(GraphNode *node, std::vector<int> &visit){
    1
    for (int i = 0; i < node->neighbors.size(); i++){
        if (2){
            if (DFS_graph(node->neighbors[i], visit) == 0){
3
            }
        }
        else if (visit[node->neighbors[i]->label] == 0){
4
        }
    }
5
    return true;
}
```



# 例5:方法1，实现

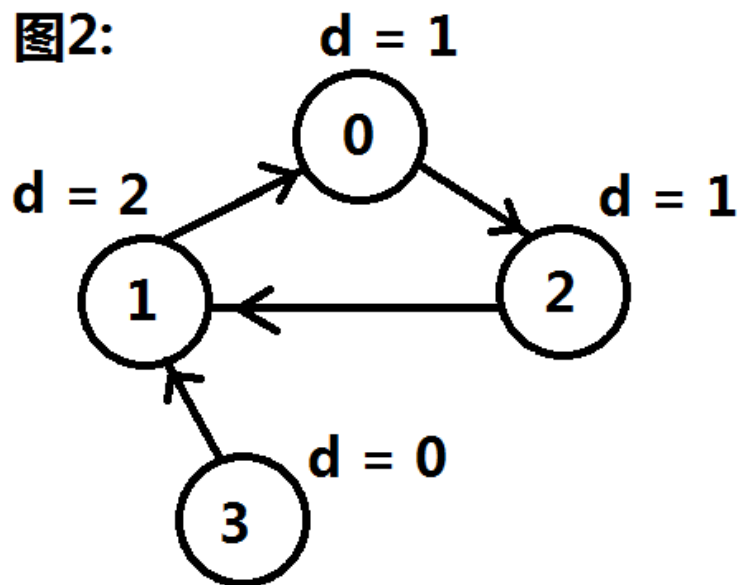
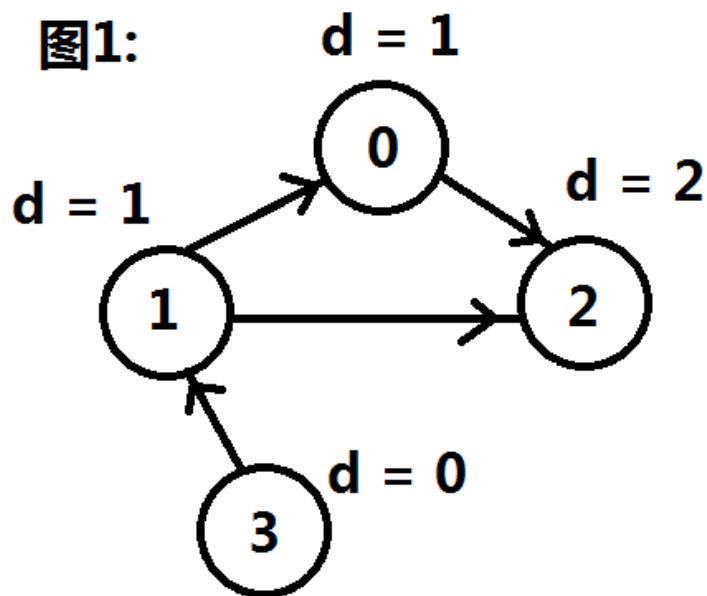
```
#include <vector>
struct GraphNode{
    int label;
    std::vector<GraphNode *> neighbors;
    GraphNode(int x) : label(x) {}
};

//节点访问状态,-1没有访问过,0代表正在访问,1代表已完成访问
bool DFS_graph(GraphNode *node, std::vector<int> &visit){
    visit[node->label] = 0;
    for (int i = 0; i < node->neighbors.size(); i++){
        if (visit[node->neighbors[i]->label] == -1){
            if (DFS_graph(node->neighbors[i], visit) == 0){
                return false;
            }
        }
        else if (visit[node->neighbors[i]->label] == 0){
            return false;
        }
    }
    visit[node->label] = 1;
    return true;
}
```

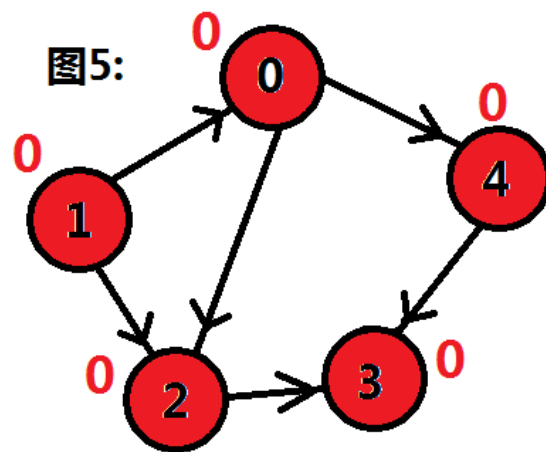
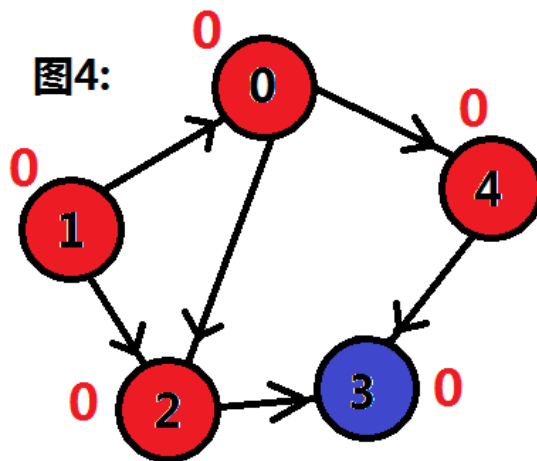
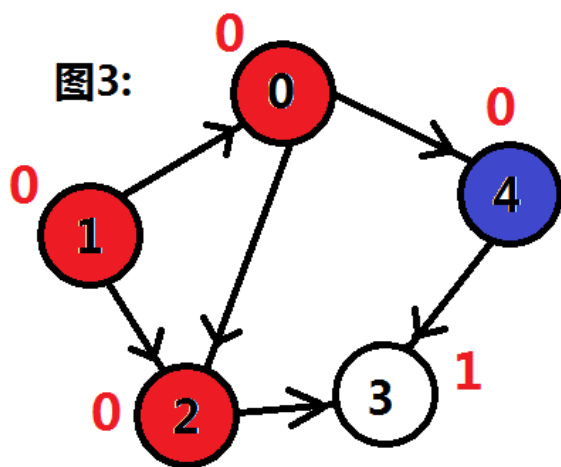
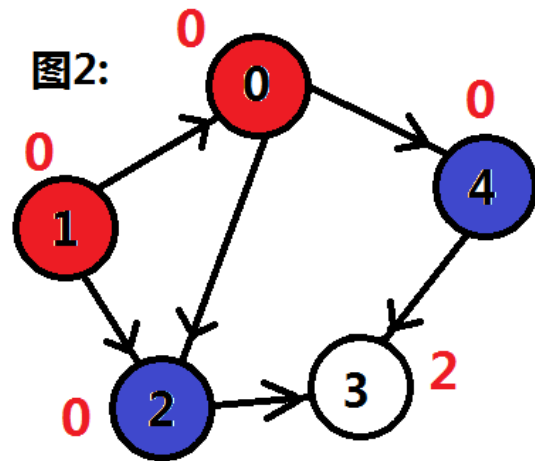
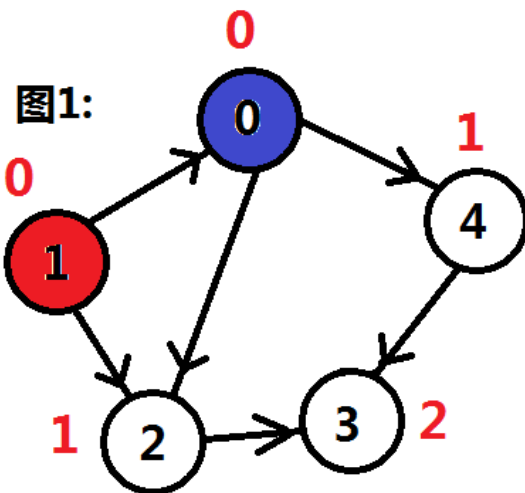
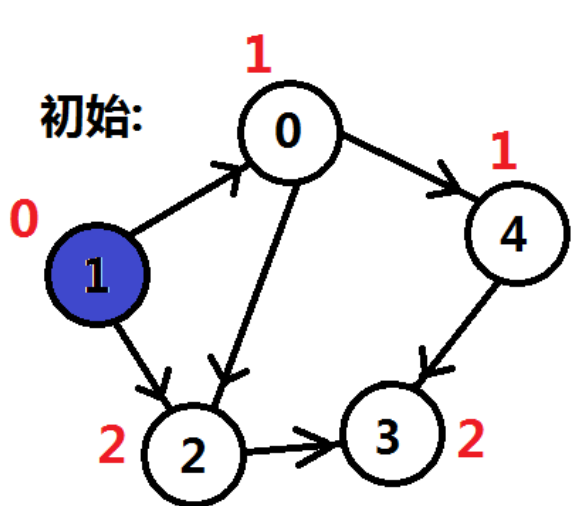


## 例5:方法2, 拓扑排序(宽度优先搜索)

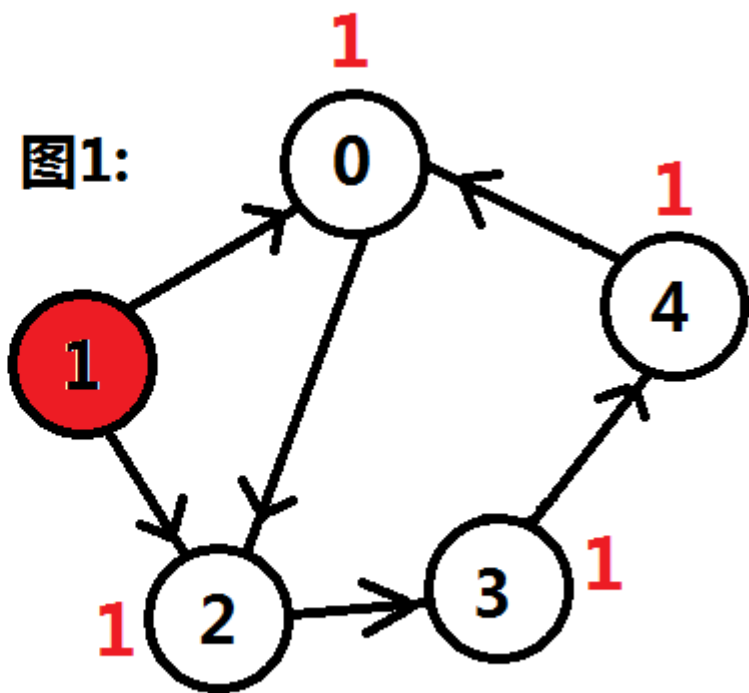
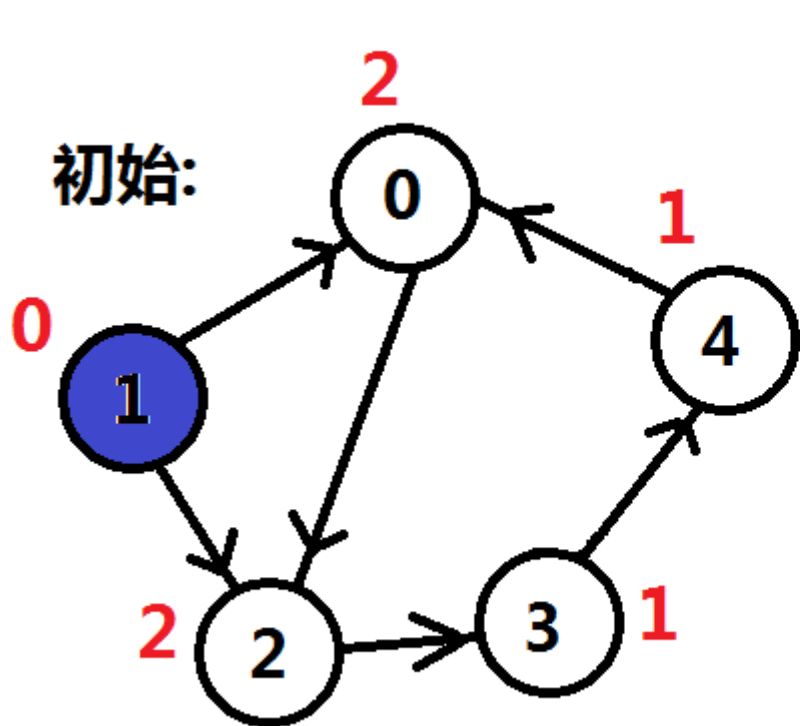
在**宽度优先搜索**时, 只将**入度**为0的点添加至队列。当完成一个顶点的搜索(从队列取出), 它指向的所有顶点**入度都减1**, 若此时某顶点入度为0则**添加**至队列, 若完成宽度搜索后, 所有的点入度都为0, 则**图无环**, 否则**有环**。



# 例5:算法思路(无环)



# 例5:算法思路(有环)

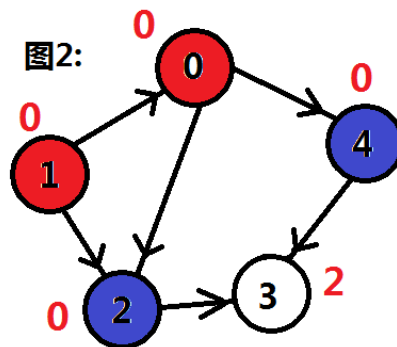
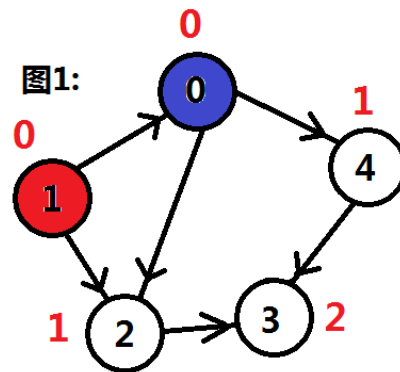
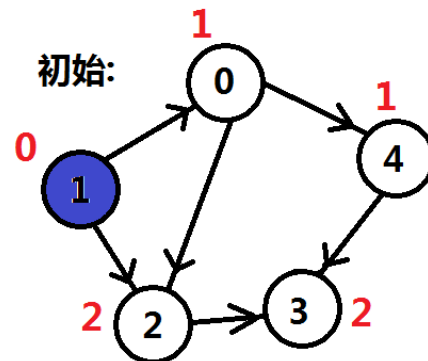


# 例5:方法2, 课堂练习

```
class Solution {
public:
    bool canFinish(int numCourses,
                  std::vector<std::pair<int, int> >& prerequisites) {
        std::vector<GraphNode*> graph;
        std::vector<int> degree; //入度数组

        for (int i = 0; i < numCourses; i++){
            degree.push_back(0);
            graph.push_back(new GraphNode(i));
        }
        for (int i = 0; i < prerequisites.size(); i++){
            GraphNode *begin = graph[prerequisites[i].second];
            GraphNode *end = graph[prerequisites[i].first];
            begin->neighbors.push_back(end); //入度++, 即pair<课程1, 课程2>
            1 课程1的入度++
        }

        std::queue<GraphNode*> Q;
        for (int i = 0; i < numCourses; i++){
            if (2) {
                Q.push(graph[i]);
            }
        }
        while(!Q.empty()){
            GraphNode *node = Q.front();
            Q.pop();
            for (int i = 0; i < node->neighbors.size(); i++){
                3
                if (4) {
                    Q.push(node->neighbors[i]);
                }
            }
        }
        for (int i = 0; i < graph.size(); i++){
            delete graph[i];
        }
        for (int i = 0; i < degree.size(); i++){
            if (5) {
            }
        }
        return true;
    }
};
```

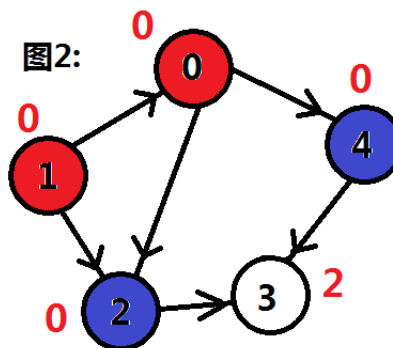
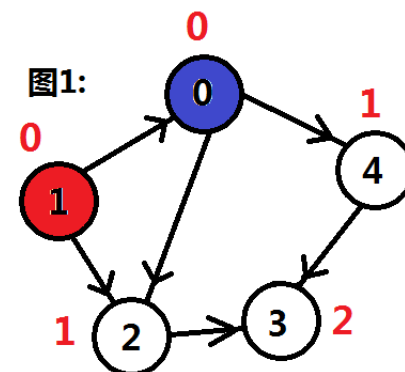
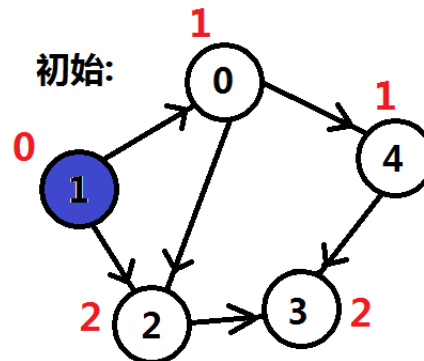


# 例5:方法2, 实现

```
class Solution {
public:
    bool canFinish(int numCourses,
        std::vector<std::pair<int, int> >& prerequisites) {
        std::vector<GraphNode*> graph;
        std::vector<int> degree; //入度数组

        for (int i = 0; i < numCourses; i++){
            degree.push_back(0);
            graph.push_back(new GraphNode(i));
        }
        for (int i = 0; i < prerequisites.size(); i++){
            GraphNode *begin = graph[prerequisites[i].second];
            GraphNode *end = graph[prerequisites[i].first];
            begin->neighbors.push_back(end);
            degree[prerequisites[i].first]++; //入度++, 即pair<课程1, 课程2>
            课程1的入度++
        }

        std::queue<GraphNode *> Q;
        for (int i = 0; i < numCourses; i++){
            if (degree[i] == 0) {
                Q.push(graph[i]);
            }
        }
        while(!Q.empty()){
            GraphNode *node = Q.front();
            Q.pop();
            for (int i = 0; i < node->neighbors.size(); i++){
                degree[node->neighbors[i]->label]--;
                if (degree[node->neighbors[i]->label] == 0) {
                    Q.push(node->neighbors[i]);
                }
            }
        }
        for (int i = 0; i < graph.size(); i++){
            delete graph[i];
        }
        for (int i = 0; i < degree.size(); i++){
            if (degree[i]){
                return false;
            }
        }
        return true;
    }
};
```





# 例5:测试与leetcode提交结果

```
int main() {  
    std::vector<std::pair<int, int> > prerequisites;  
    prerequisites.push_back(std::make_pair(1, 0));  
    prerequisites.push_back(std::make_pair(2, 0));  
    prerequisites.push_back(std::make_pair(3, 1));  
    prerequisites.push_back(std::make_pair(3, 2));  
    Solution solve;  
    printf("%d\n", solve.canFinish(4, prerequisites));  
    return 0;  
}
```

[Course Schedule](#)

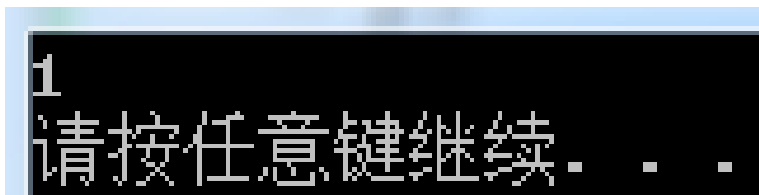
Submission Details

37 / 37 test cases passed.

Status: **Accepted**

Runtime: 13 ms

Submitted: 39 minutes ago



# 结束

---

非常感谢大家！

林沐

# 疑问

---

□ 问题答疑：<http://www.xxwenda.com/>

■ 可邀请老师或者其他回答问题

# 联系我们

---

## 小象学院：互联网新技术在线教育领航者

- 微信公众号：小象
- 新浪微博：ChinaHadoop

