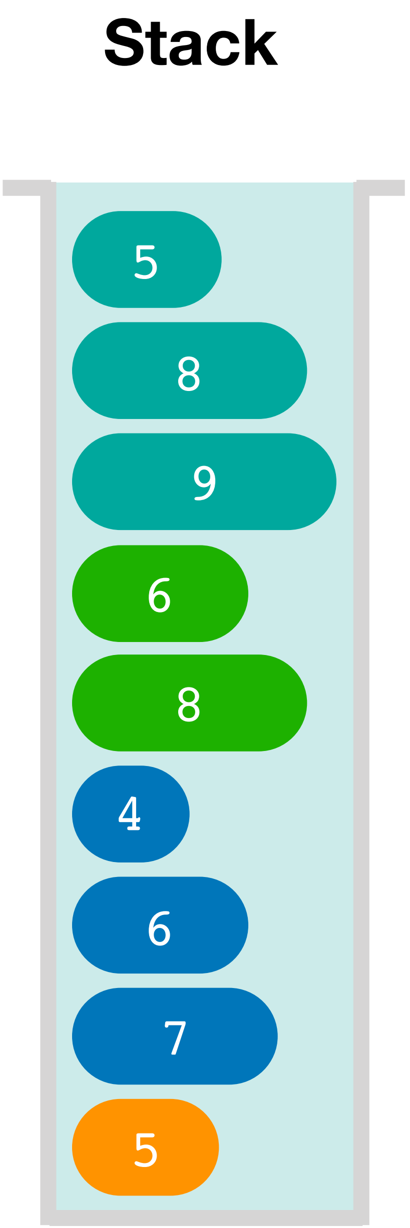
数据结构实验（4）

栈和队列（二）

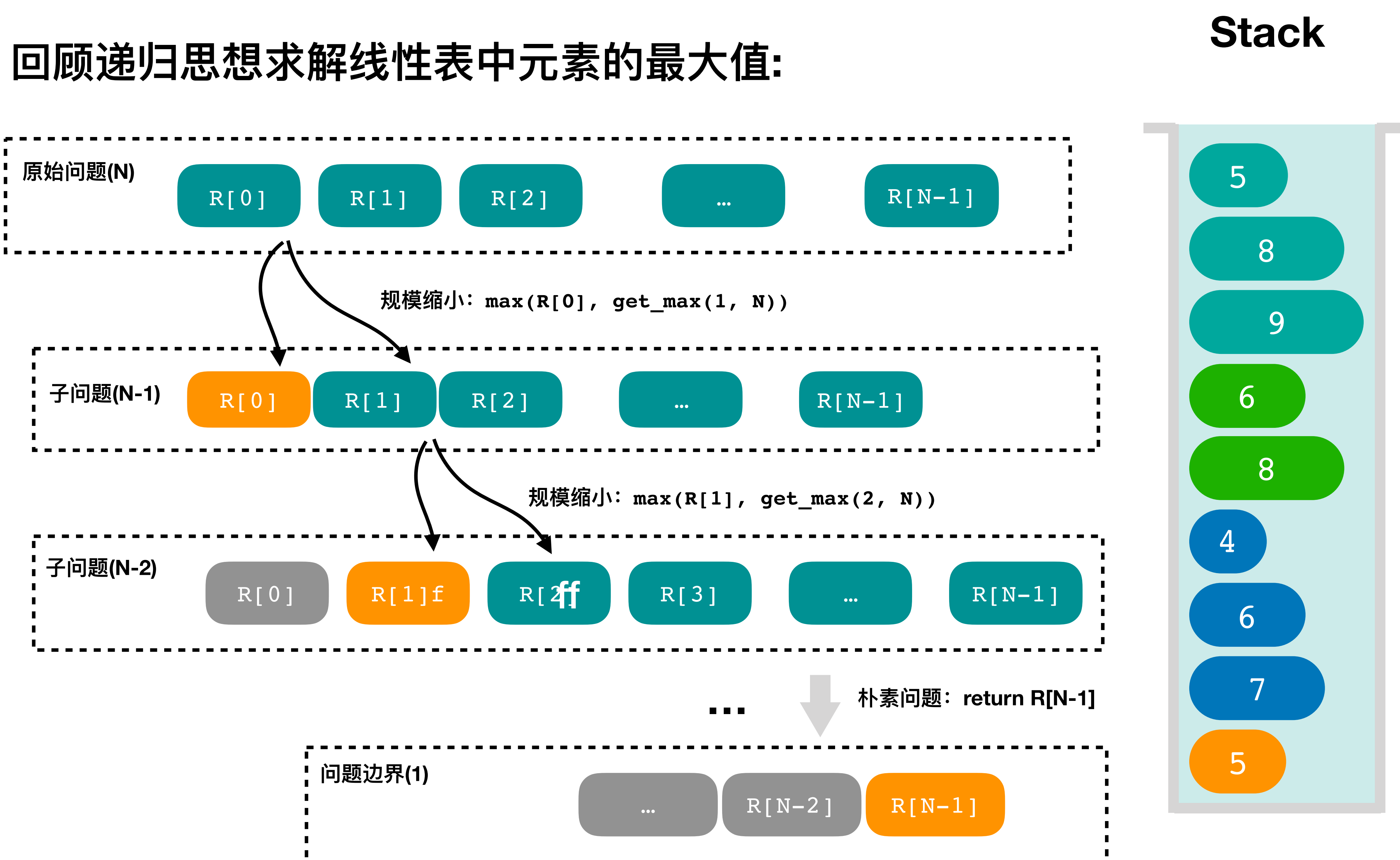
目录

- 栈与队列
  - 栈与队列的扩展
    - 栈 + get_max()
    - 使用栈模拟队列
  - 深度优先搜索

- 栈与队列的扩展
  - 栈 + get_max()



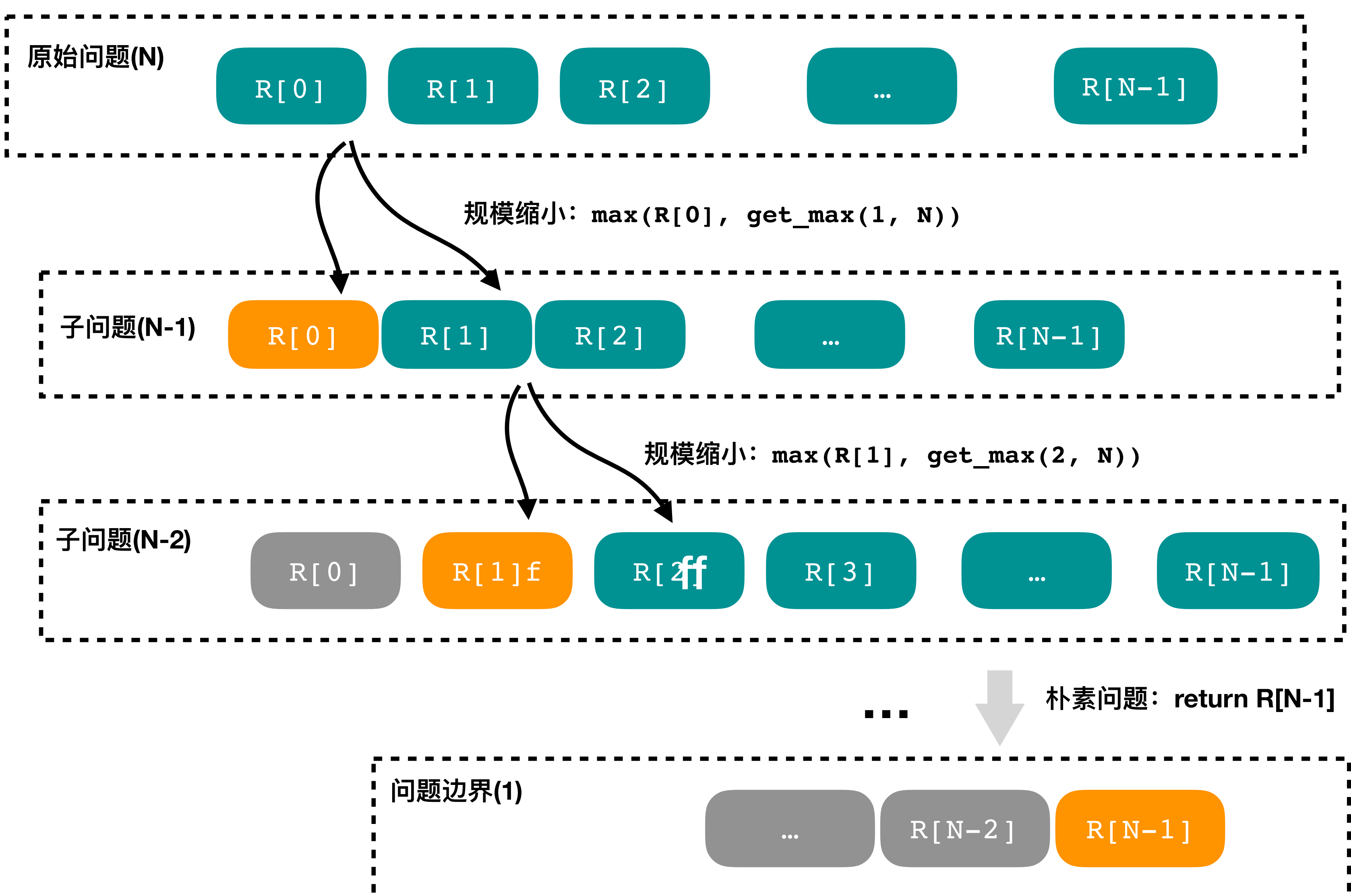
- 栈与队列的扩展
  - 栈 + get_max()



- 栈与队列的扩展
  - 栈 + get_max()

若使用一个线性表，其 第 i 个元素的值为栈中头i个元素的最大值/最小值

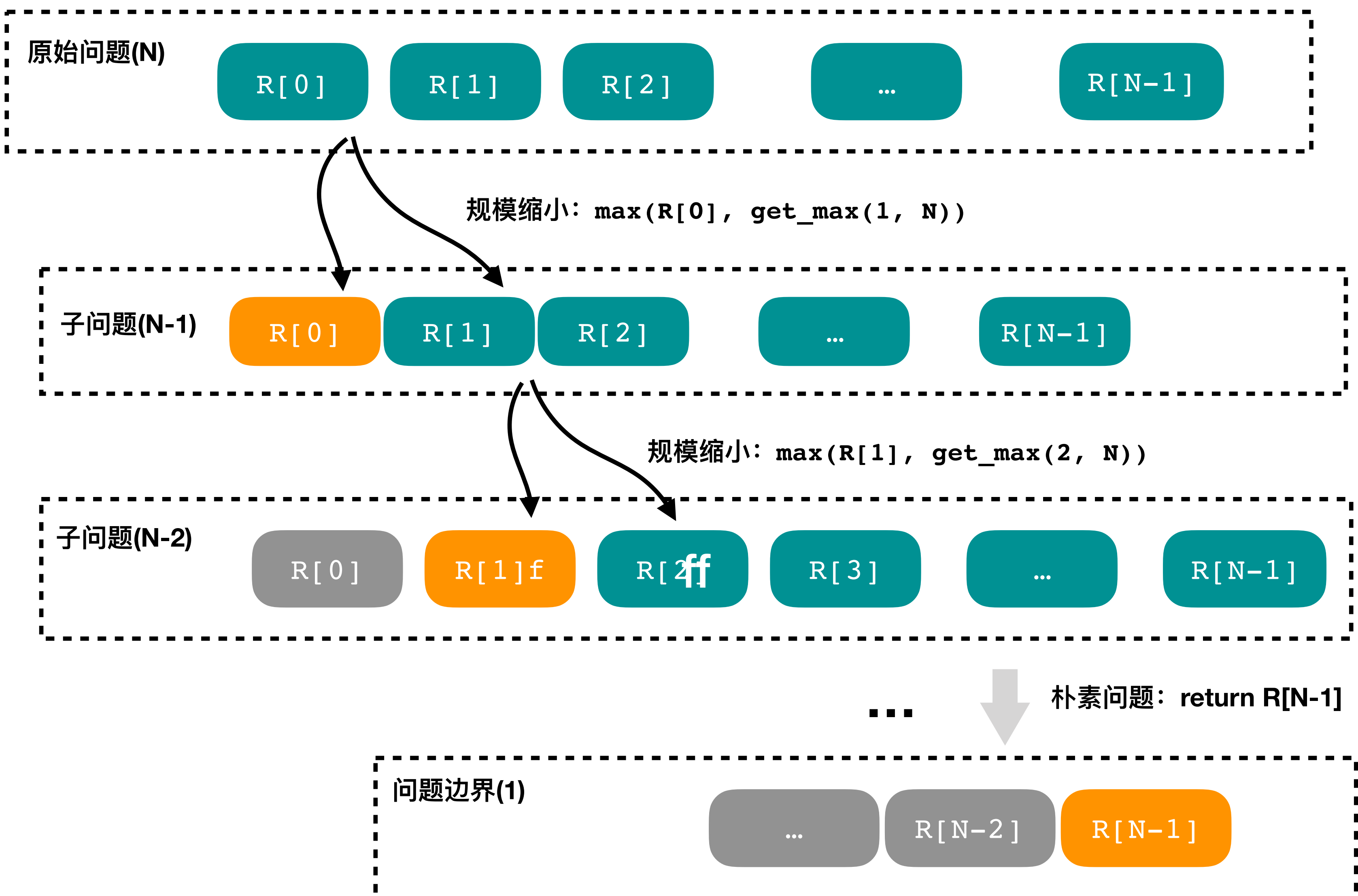
回顾递归思想求解线性表中元素的最大值:



- 栈与队列的扩展
    - 栈 + get_max()

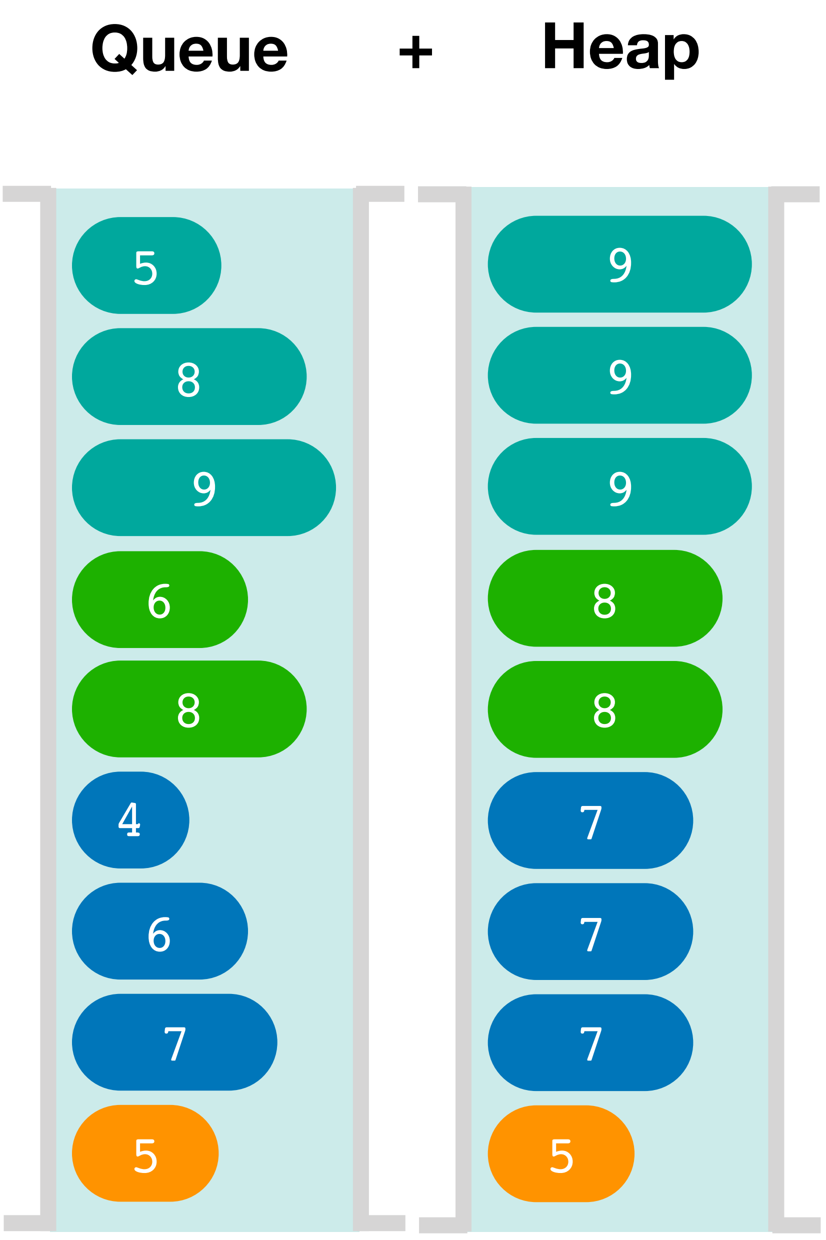
```
push(e) {
  stack.push(e);
  heap.push(max(heap.top(), e));
}
pop() {
  stack.pop();
  heap.pop();
}
```

回顾递归思想求解线性表中元素的最大值:



- 栈与队列的扩展
  - 队列 + get_max()

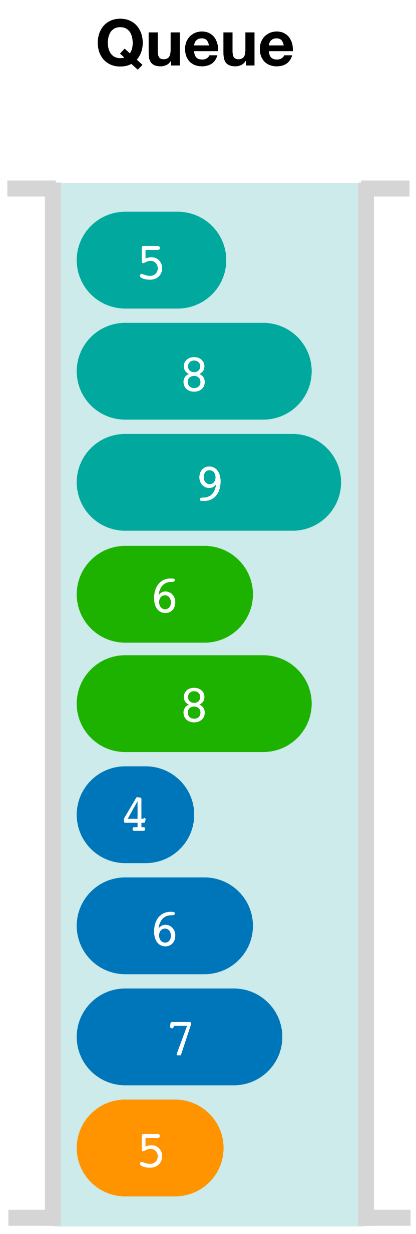
如何实现?



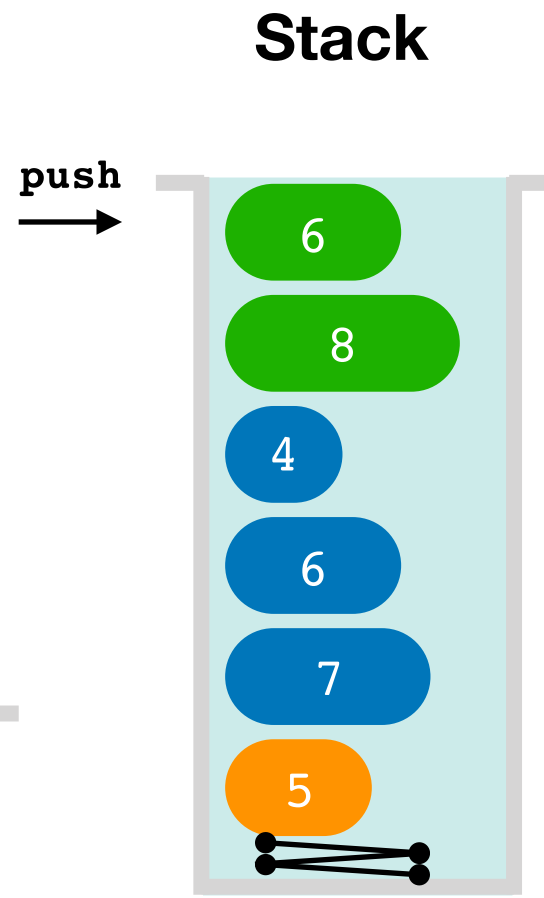
目录

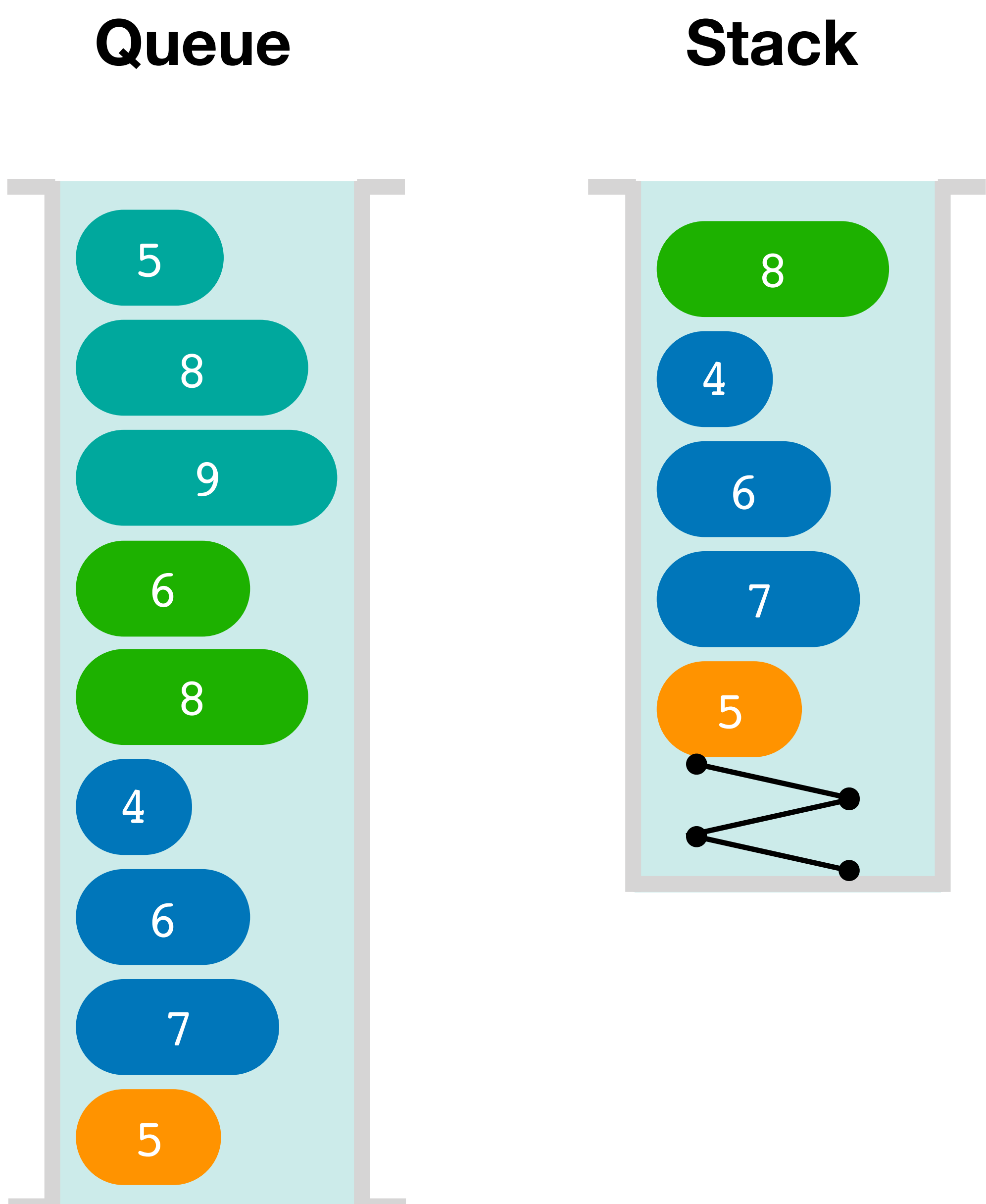
- 栈与队列
  - 深度优先搜索
  - 栈与队列的扩展
    - 栈 + get_max()
    - 使用栈模拟队列

- 栈与队列的扩展
  - 使用栈模拟队列

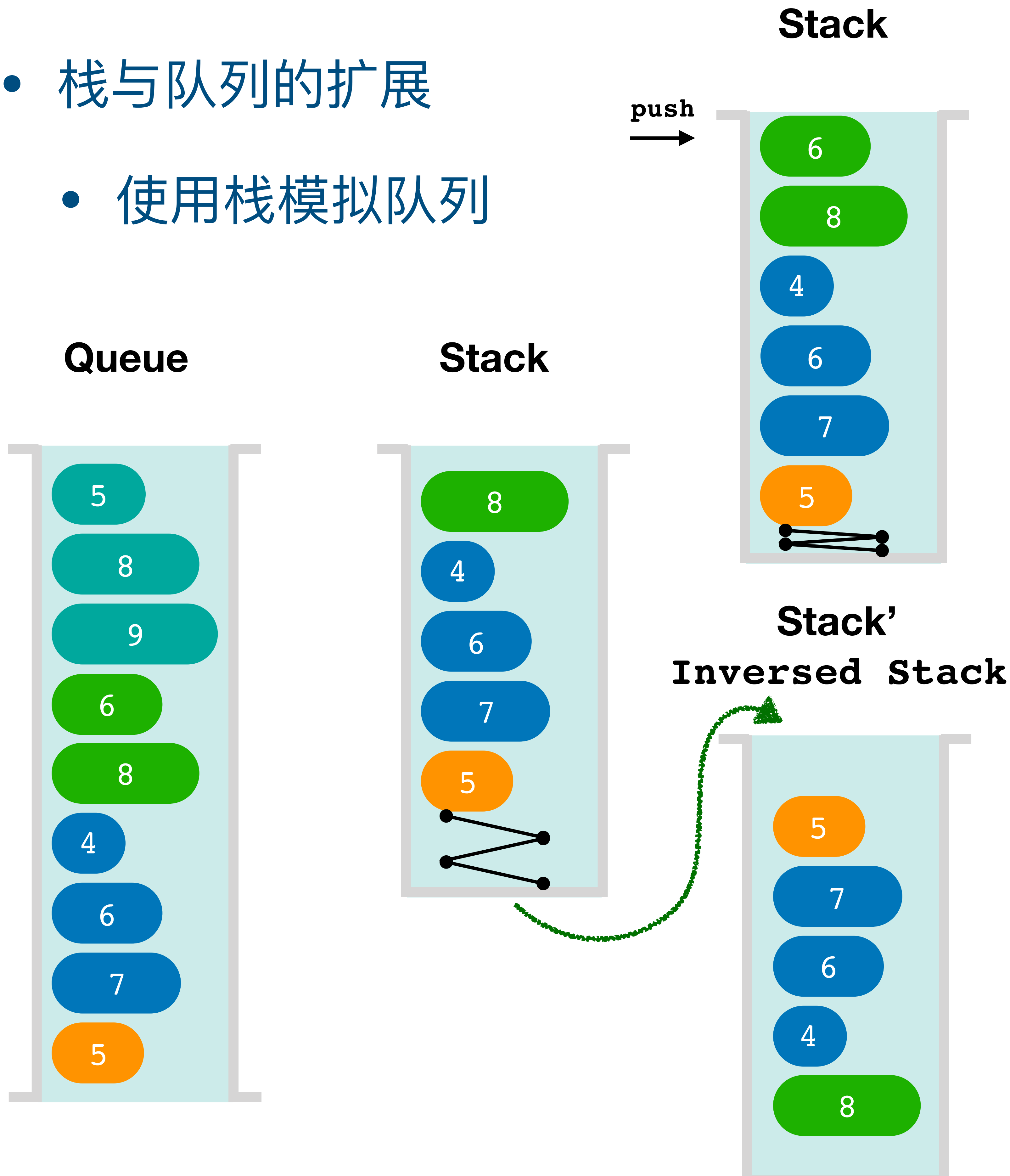


- 栈与队列的扩展
  - 使用栈模拟队列

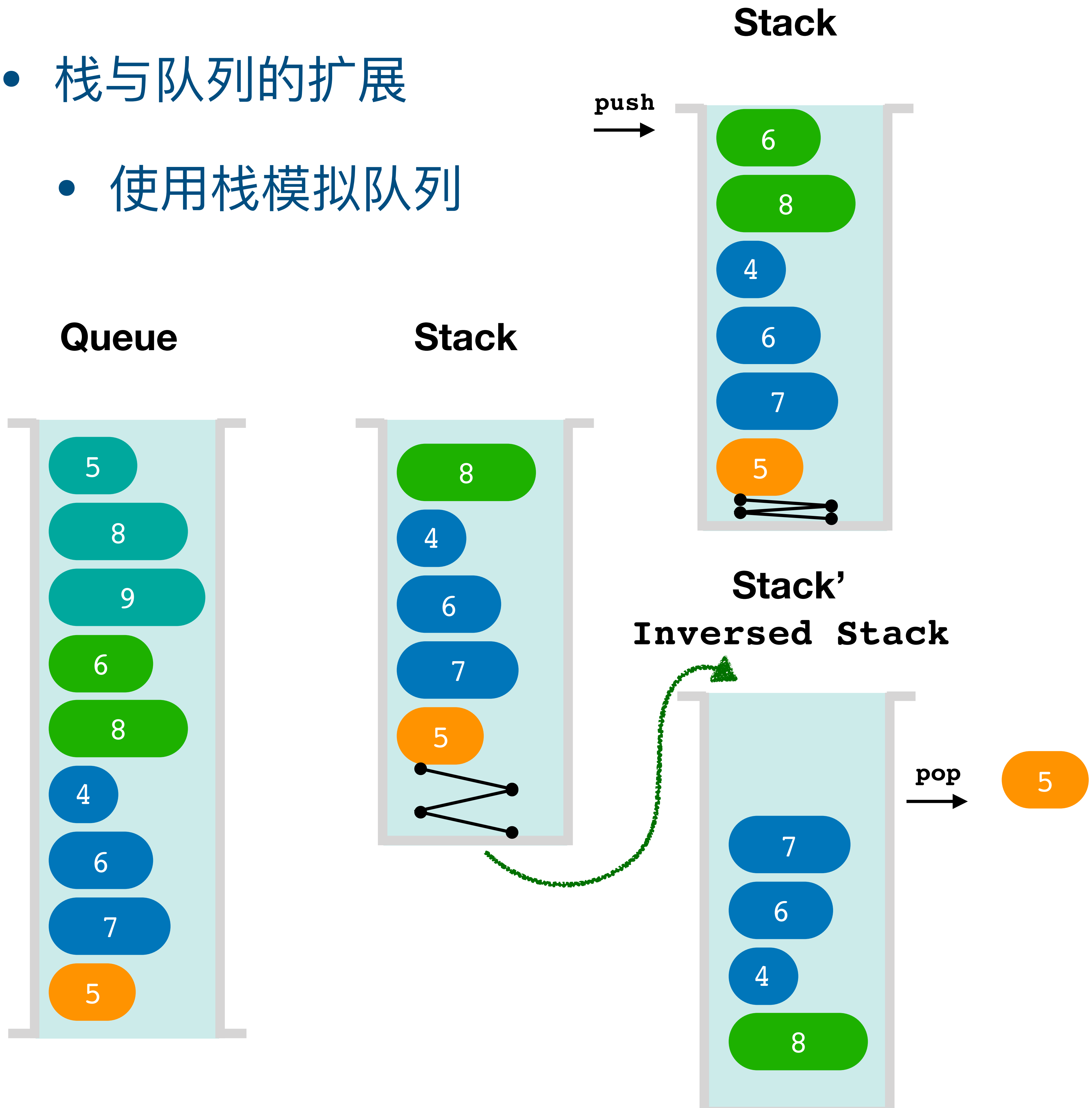




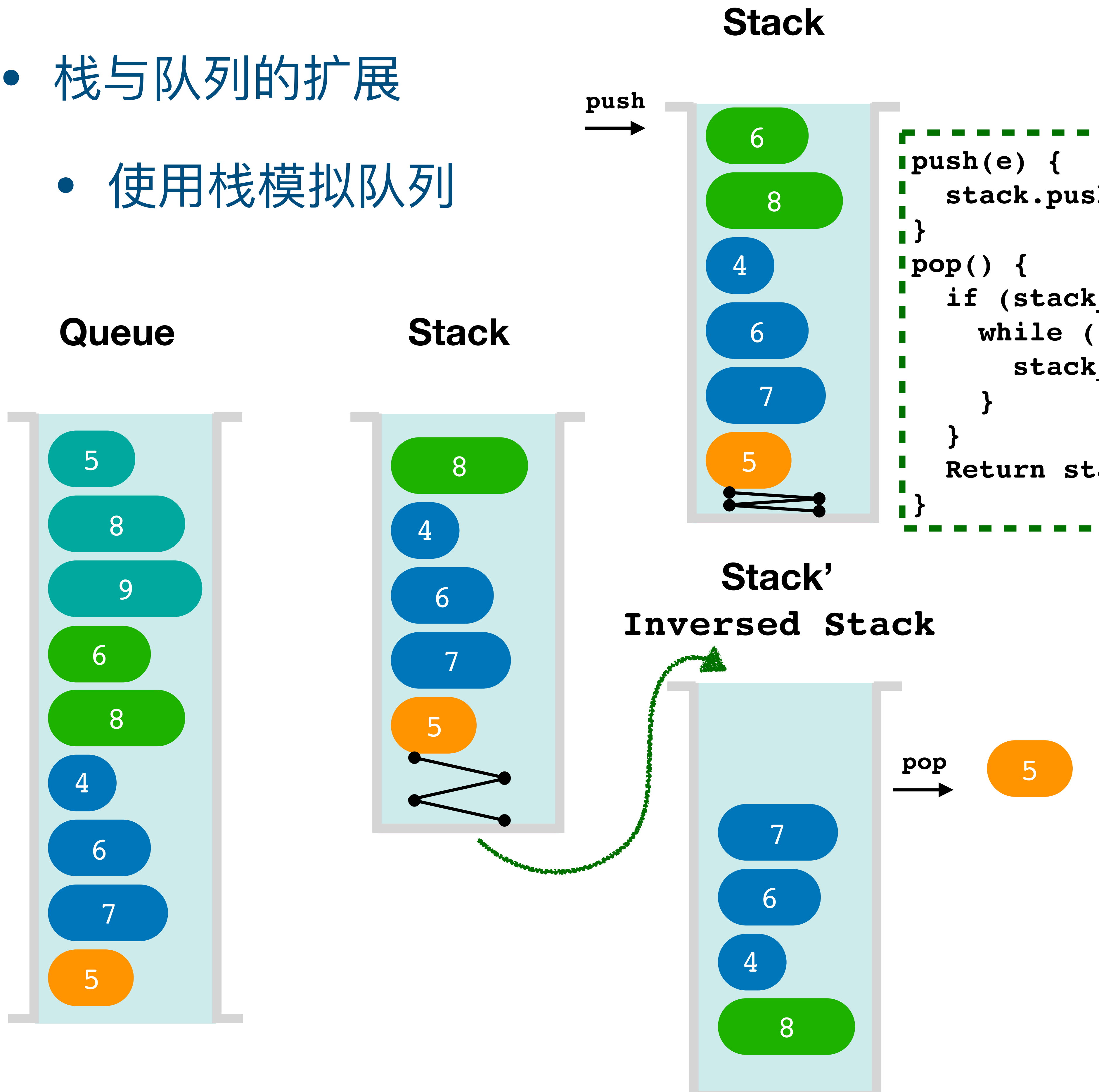
- 栈与队列的扩展
  - 使用栈模拟队列



- 栈与队列的扩展
  - 使用栈模拟队列



- 栈与队列的扩展
  - 使用栈模拟队列

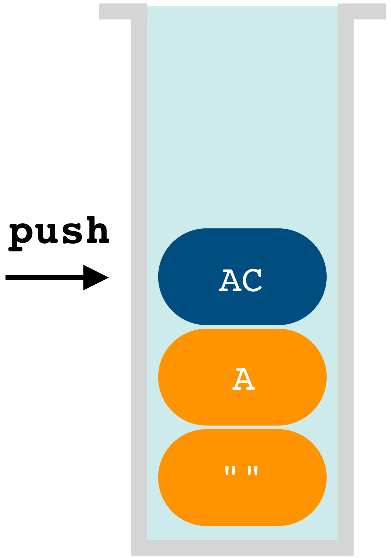


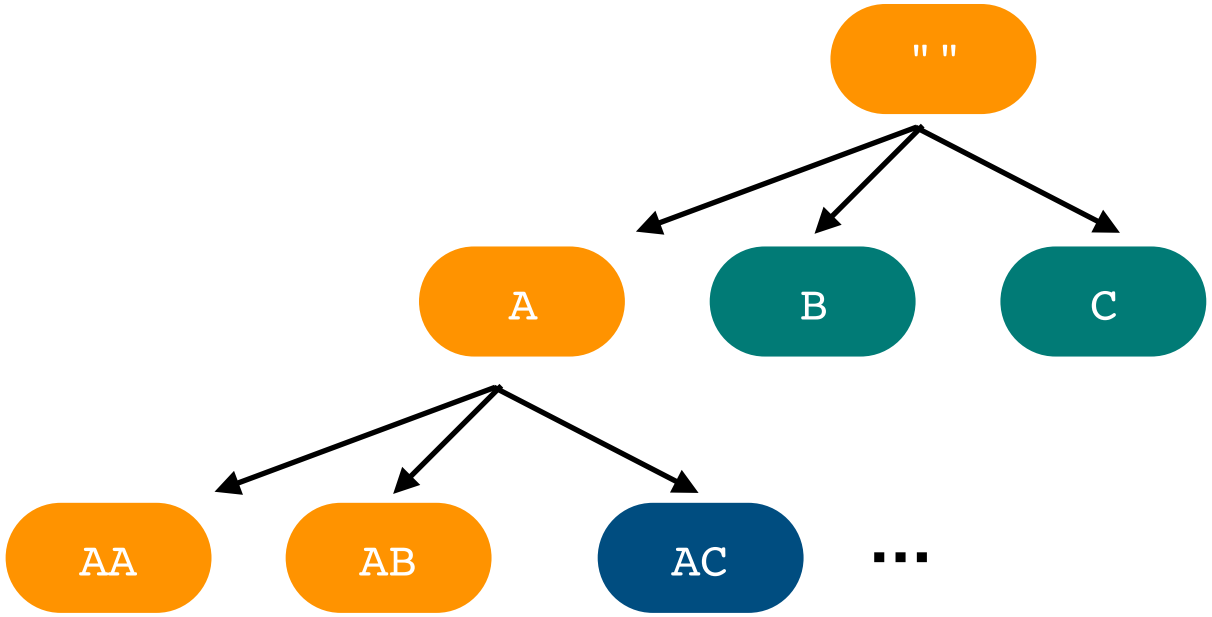

```
push(e) {
  stack.push(e);
}
pop() {
  if (stack_.isEmpty()) {
    while (!stack.isEmpty()) {
      stack_.push(stack.pop());
    }
  }
  Return stack_.pop();
}
```

目录

- 栈与队列
  - 栈与队列的扩展
    - 栈 + get_max()
    - 使用栈模拟队列
  - 深度优先搜索

- 深度优先搜索
    - 深度优先搜索回顾





Tag #Depth-first Search in LeetCode
https://leetcode.com/tag/depth-first-search/

- 深度优先搜索
  - 深度优先搜索（1）

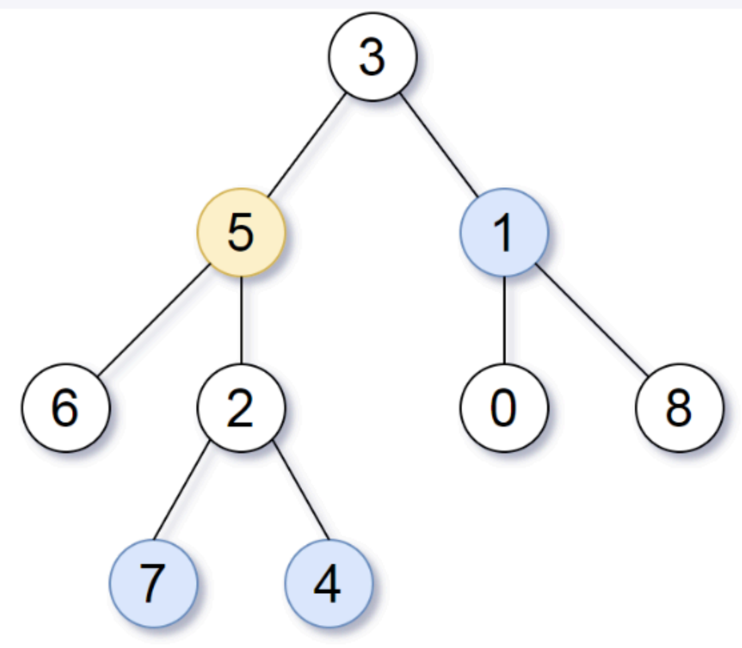
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

Example 1:

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

Output: [7,4,1]

Explanation:
The nodes that are a distance 2 from the target node (with value 5)
have values 7, 4, and 1.
```



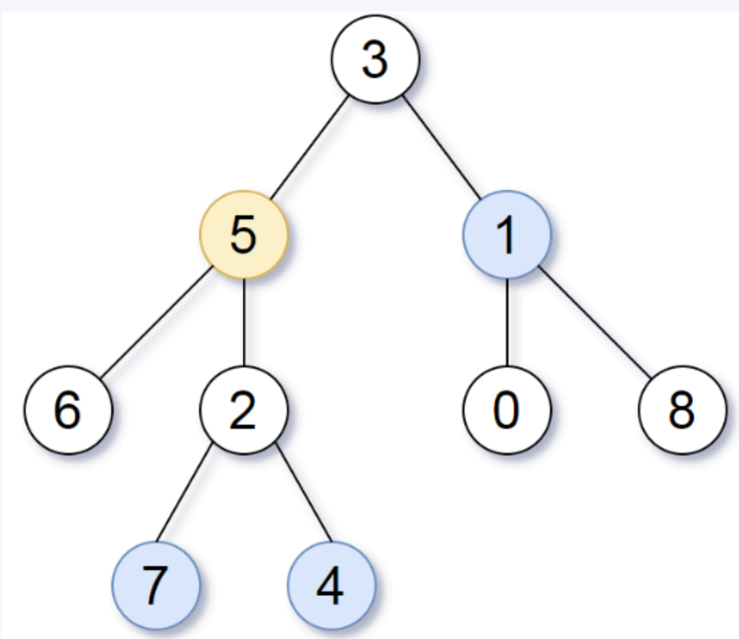
- 深度优先搜索
  - 深度优先搜索（1）

LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

Example 1:

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2
```

用线性表 t[] 存储完全二叉树:
1) t[0] 为 Root
2) 对于 t[i], t[i*2] 为左儿子，t[i*2]+1 为右儿子



- 深度优先搜索
  - 深度优先搜索（1）

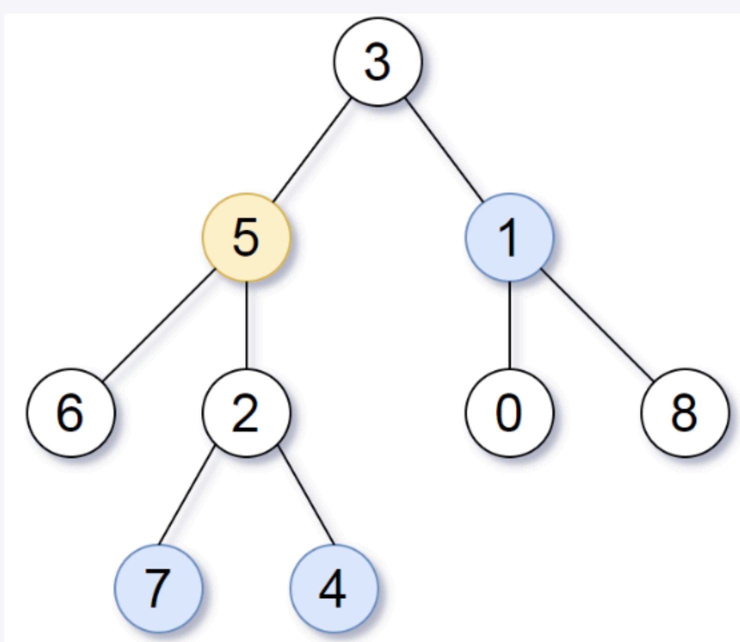
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

Example 1:

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

Output: [7,4,1]

Explanation:
The nodes that are a distance 2 from the target node (with value 5)
have values 7, 4, and 1.
```



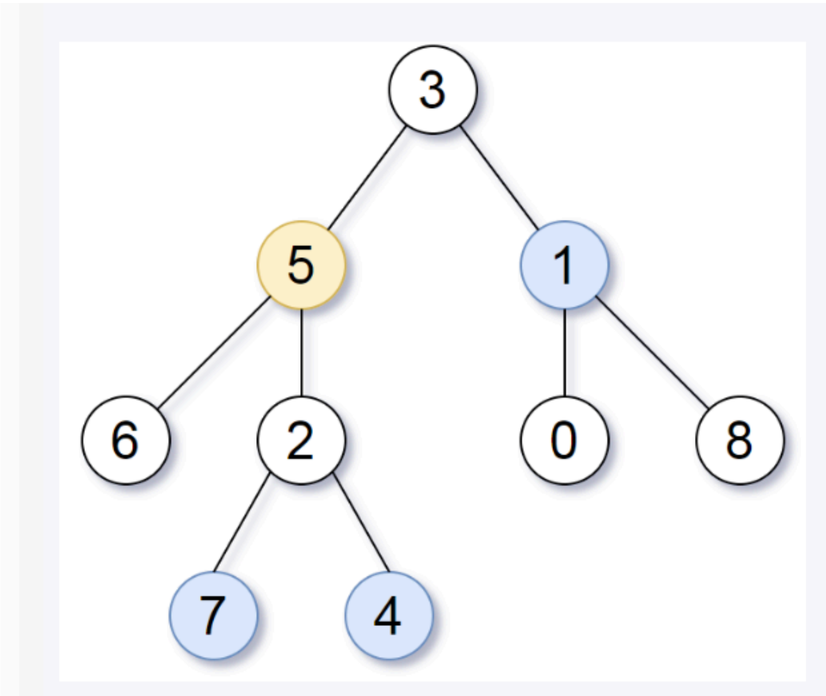
思路：
1) 从起点开始深度优先遍历二叉树；
2) 遍历到深度为 K 的节点时停止遍历；

- 深度优先搜索
  - 深度优先搜索（1）

    LeetCode 863. <All Nodes Distance K in Binary Tree>
    https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```cpp
std::vector<int> dfs(const std::vector<int>& tree, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            std::vector<int> part_parent = dfs(tree, target / 2, K - 1);
            copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            std::vector<int> part_left = dfs(tree, target * 2, K - 1);
            copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            std::vector<int> part_right = dfs(tree, target * 2 + 1, K - 1);
            copy(part_right.begin(), part_right.end(), std::back_inserter(result));
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, K);
}
```



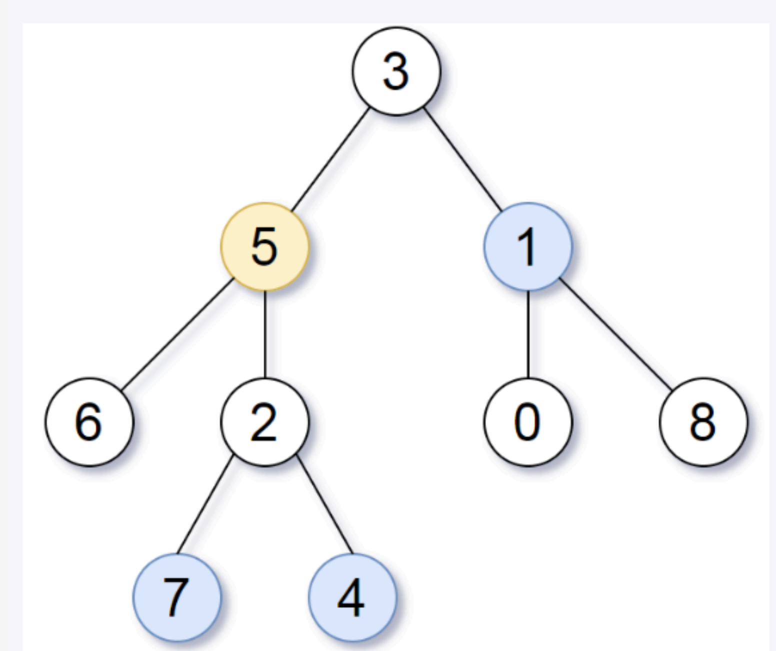
- 深度优先搜索
  - 深度优先搜索（1）

    LeetCode 863. <All Nodes Distance K in Binary Tree>
    https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            std::vector<int> part_parent = dfs(tree, target / 2, K - 1);
            copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            std::vector<int> part_left = dfs(tree, target * 2, K - 1);
            copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            std::vector<int> part_right = dfs(tree, target * 2 + 1, K - 1);
            copy(part_right.begin(), part_right.end(), std::back_inserter(result));
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, K);
}
```

注意：这里我们用 std 实现了一个线性表的合并



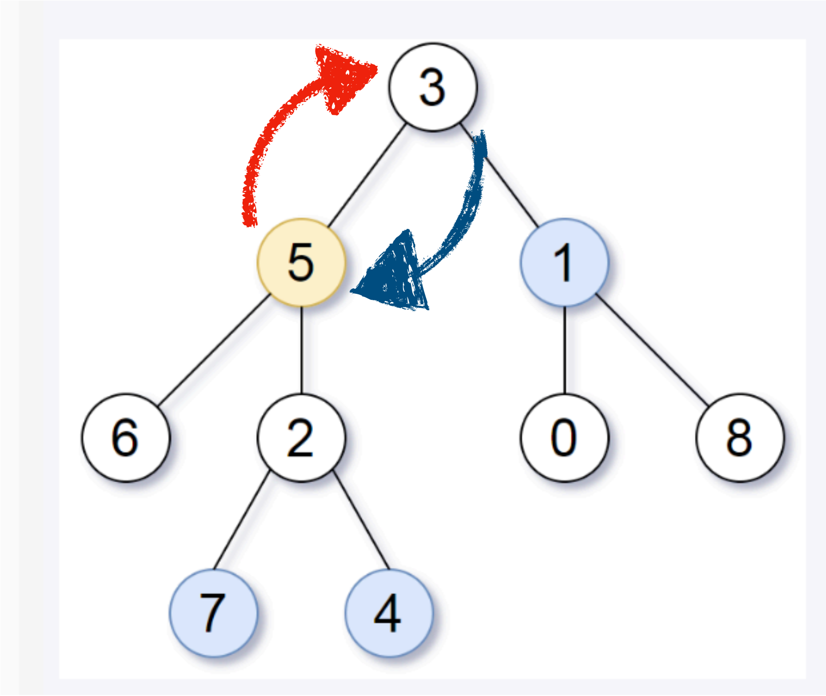
- 深度优先搜索
  - 深度优先搜索（1）

LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            std::vector<int> part_parent = dfs(tree, target / 2, K - 1);
            copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            std::vector<int> part_left = dfs(tree, target * 2, K - 1);
            copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            std::vector<int> part_right = dfs(tree, target * 2 + 1, K - 1);
            copy(part_right.begin(), part_right.end(), std::back_inserter(result));
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, K);
}
```

是否存在问题?
遍历存在环，要记录遍历过的节点

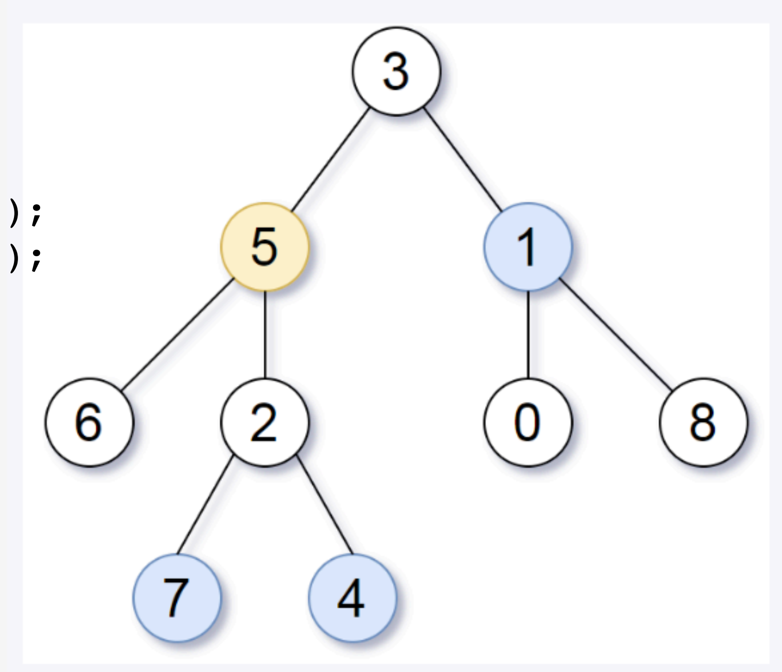


- 深度优先搜索
  - 深度优先搜索（1）

LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```



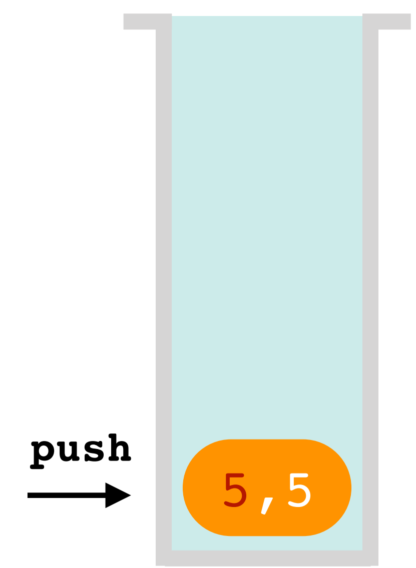
- 深度优先搜索
  - 深度优先搜索（1）

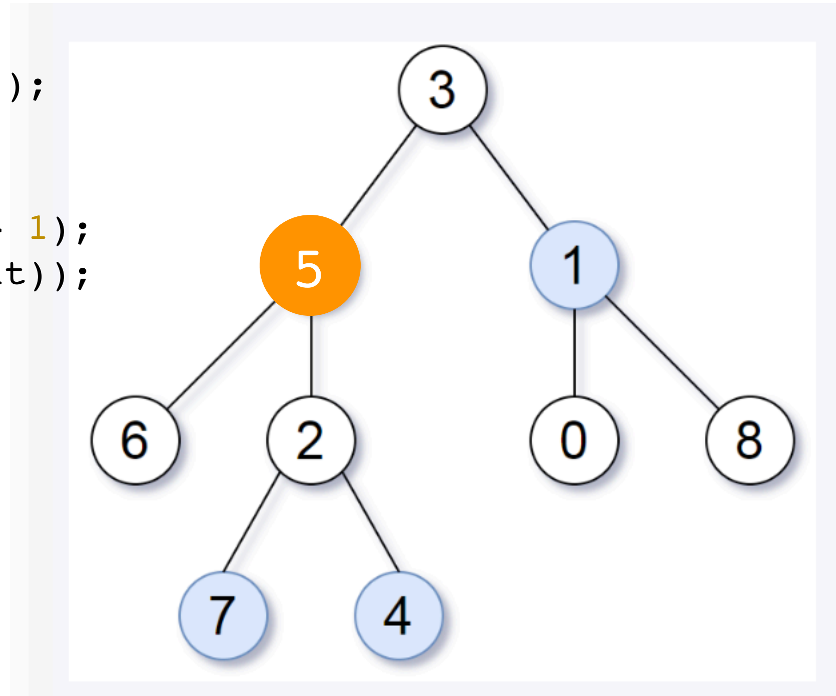
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```cpp
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

传入一个假的 parent





- 深度优先搜索

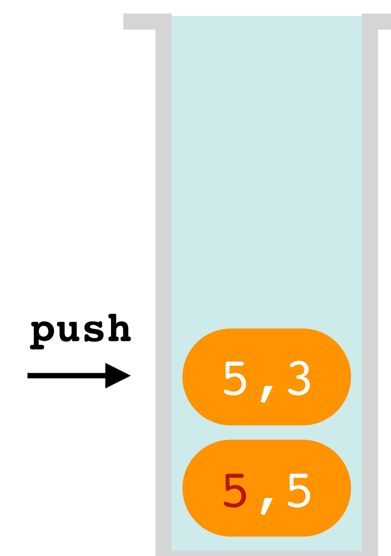
  - 深度优先搜索（1）

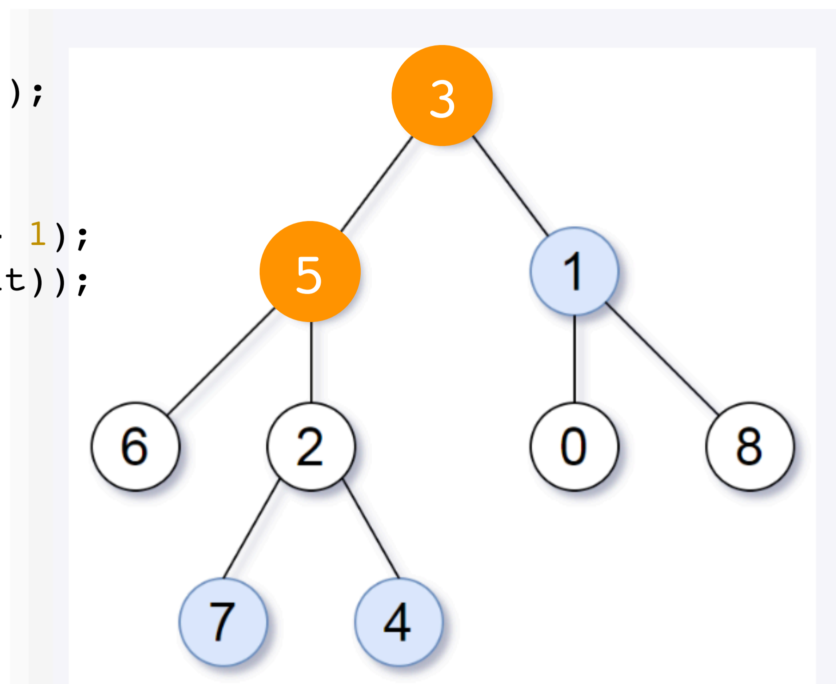
    LeetCode 863. <All Nodes Distance K in Binary Tree>
    https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

遍历其父节点

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```



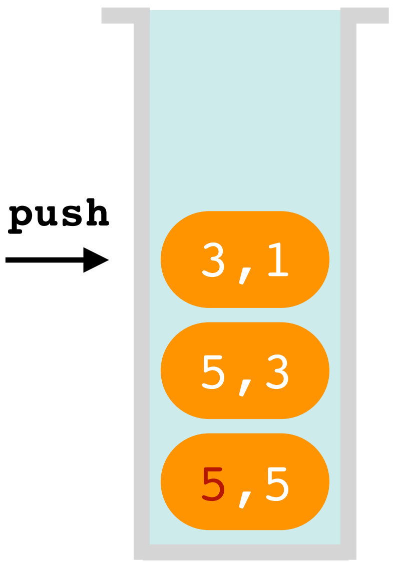


- 深度优先搜索
    - 深度优先搜索（1）

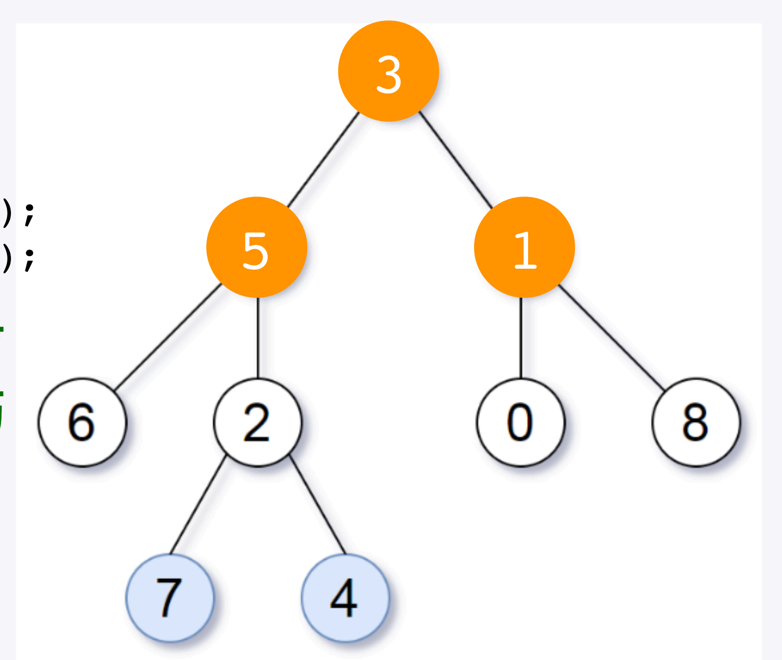
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```



其父节点为空，左节点已经遍历过，遍历其右儿子



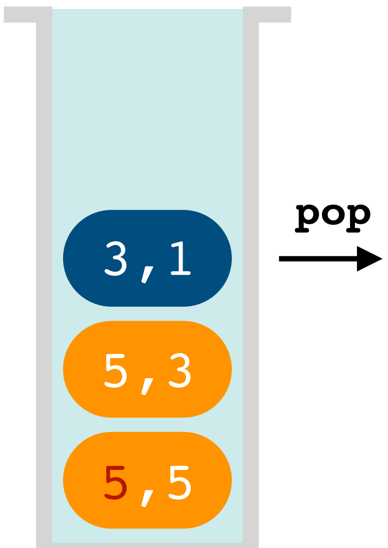
- 深度优先搜索
  - 深度优先搜索（1）

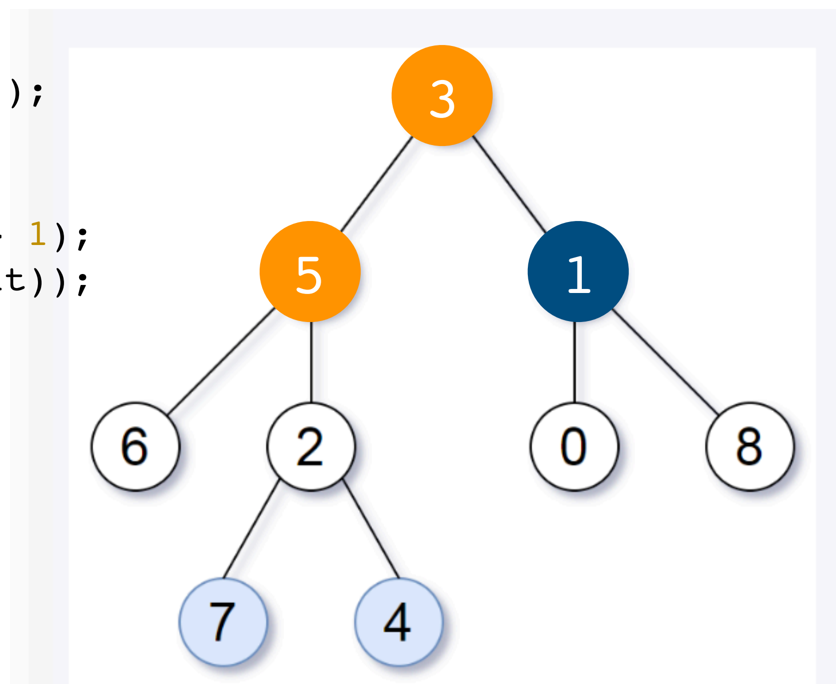
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```cpp
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

到达目标遍历深度





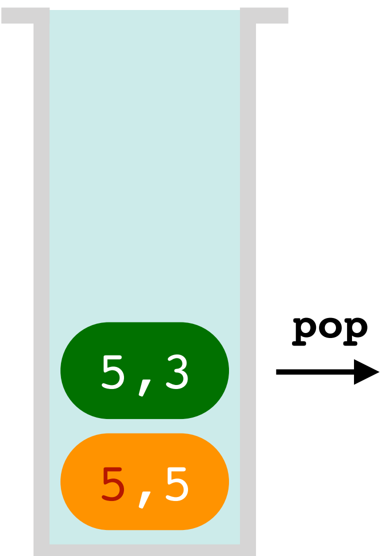
- 深度优先搜索
  - 深度优先搜索（1）

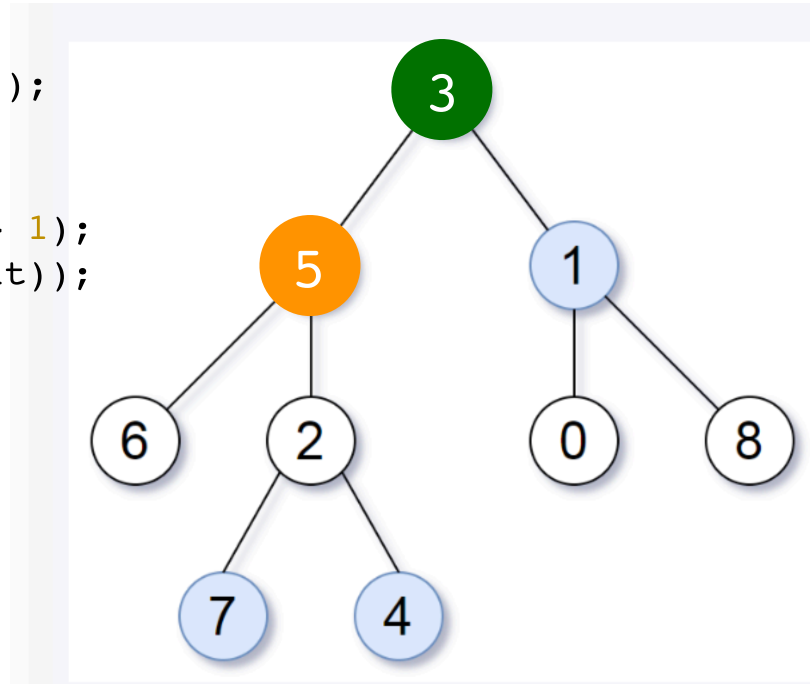
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

遍历结束





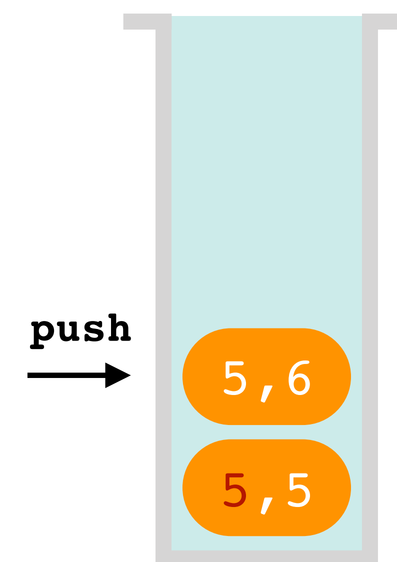
- 深度优先搜索
  - 深度优先搜索（1）

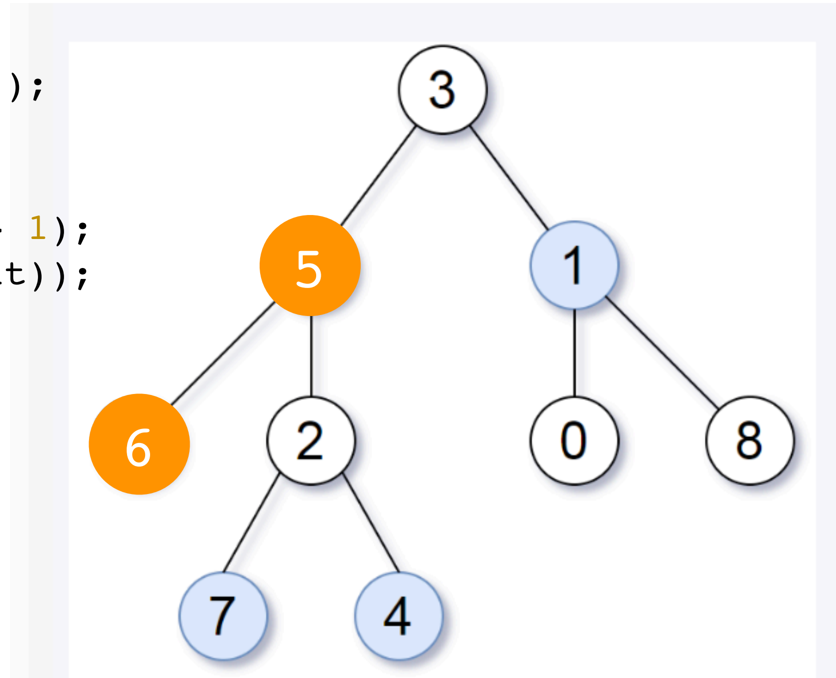
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

遍历其左儿子节点





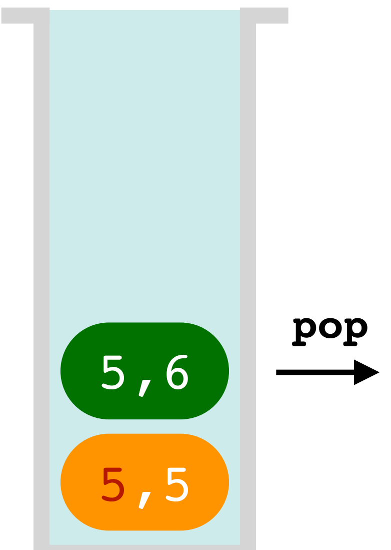
- 深度优先搜索
    - 深度优先搜索（1）

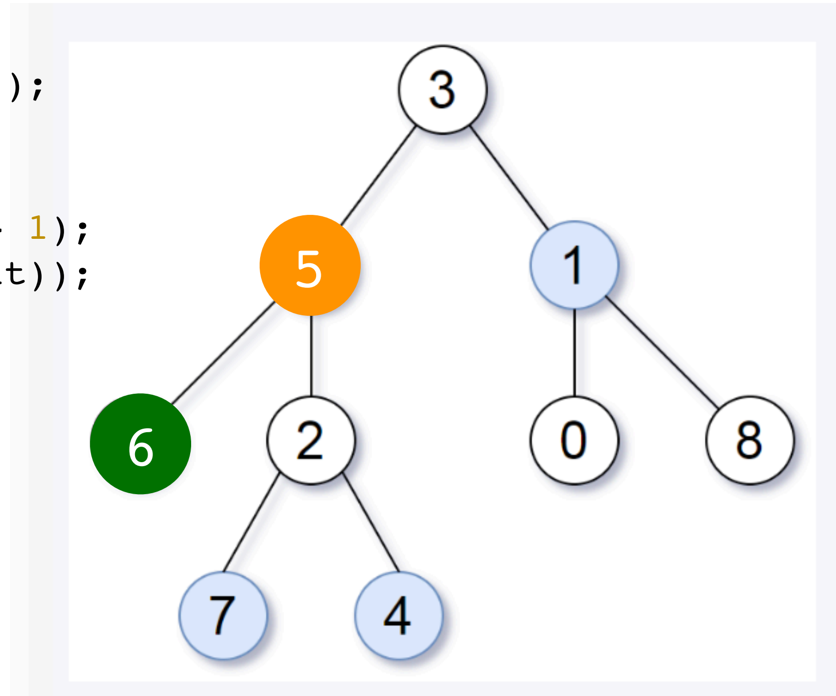
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

遍历结束





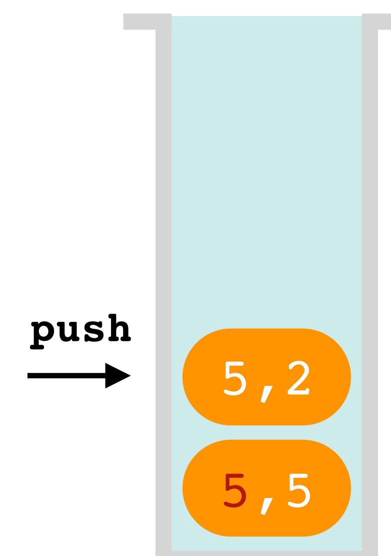
- 深度优先搜索
  - 深度优先搜索（1）

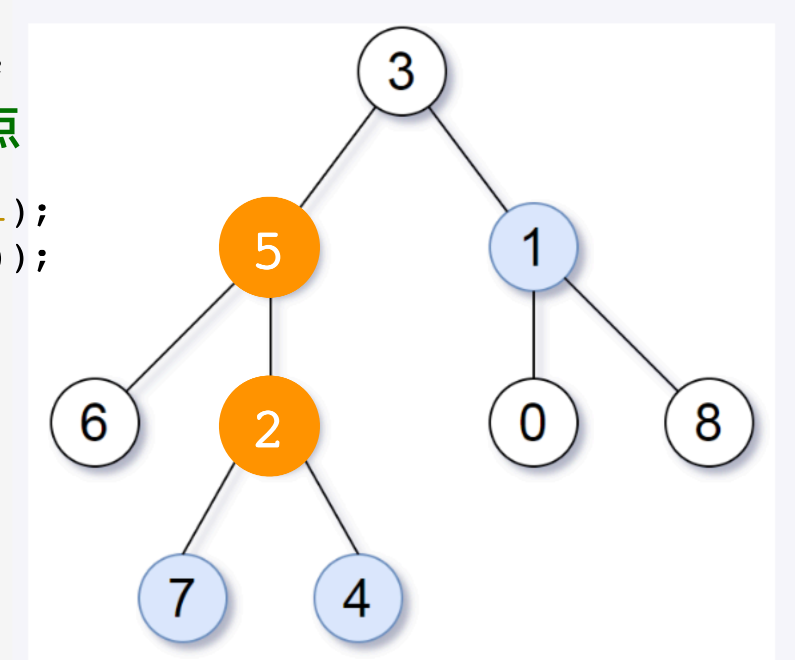
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```cpp
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

遍历其右儿子节点





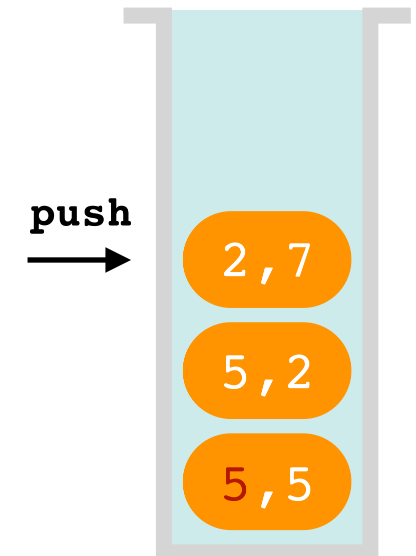
- 深度优先搜索
    - 深度优先搜索（1）

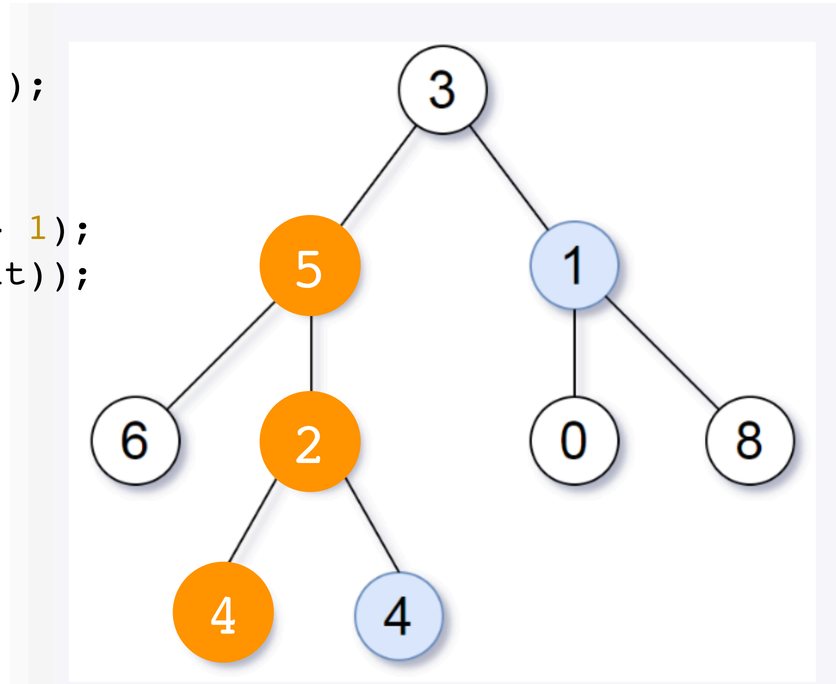
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```cpp
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

遍历其左儿子节点





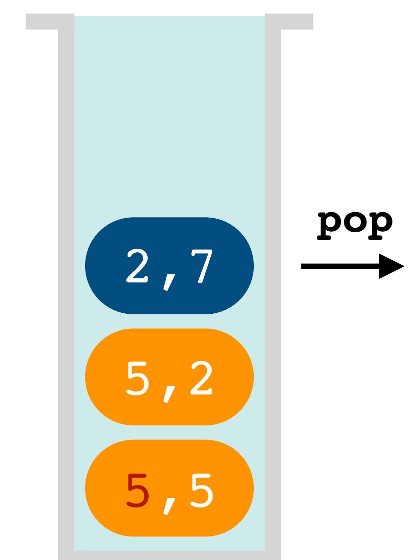
- 深度优先搜索
  - 深度优先搜索（1）

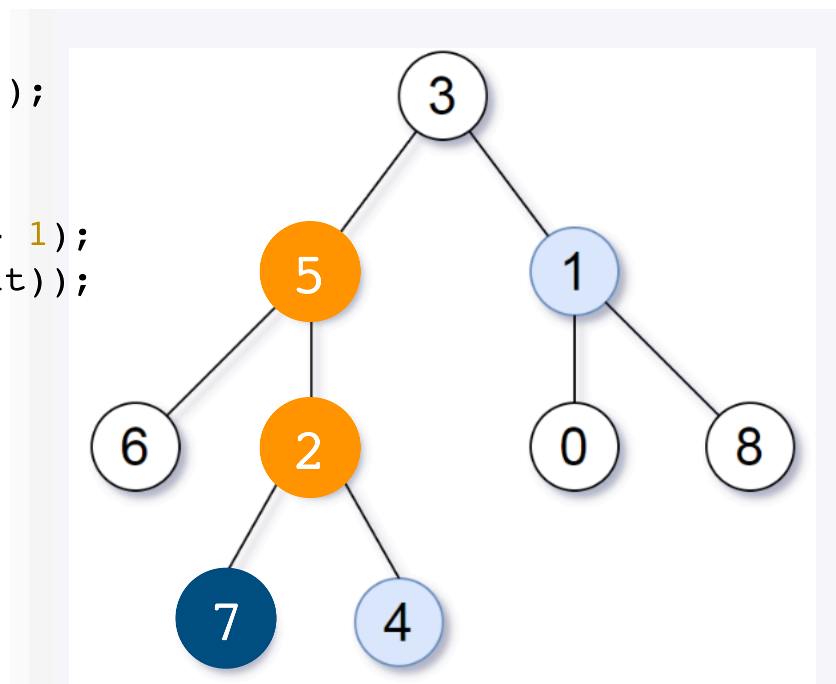
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

到达目标遍历深度





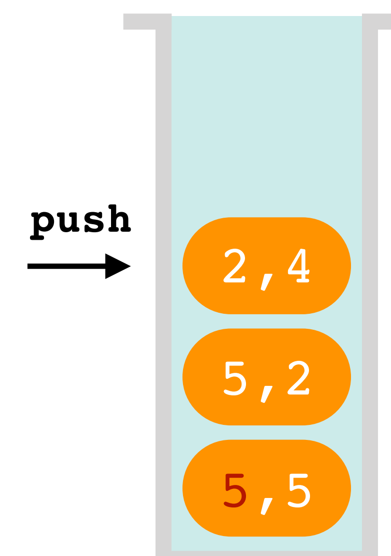
- 深度优先搜索
  - 深度优先搜索（1）

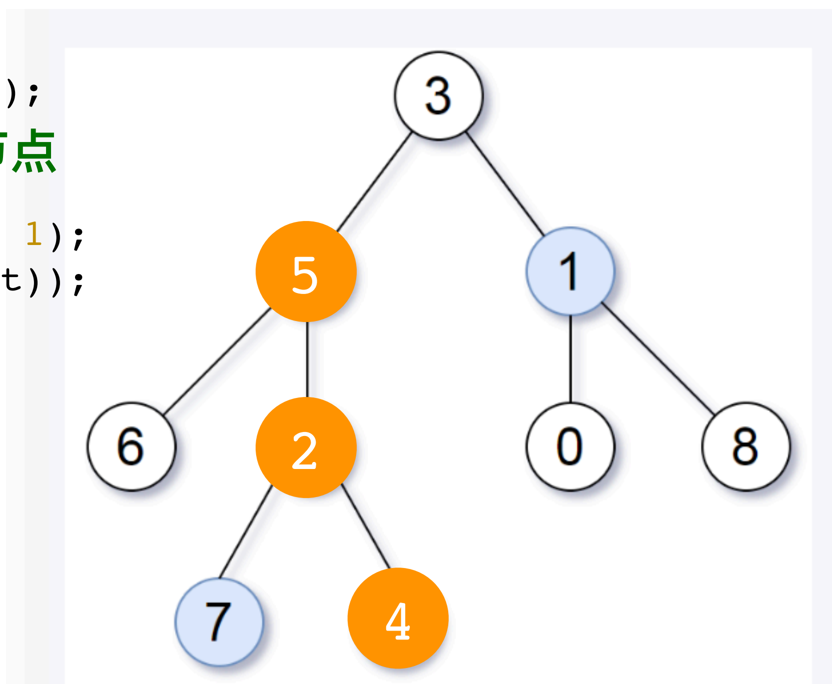
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

遍历其右儿子节点





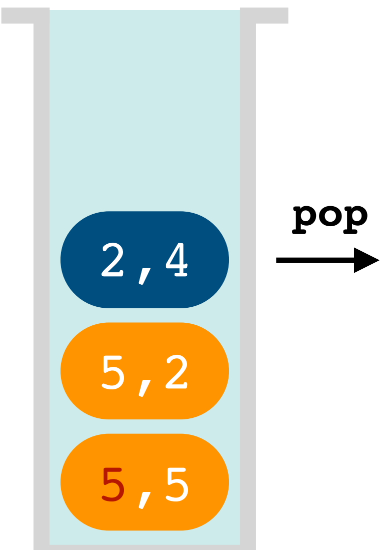
- 深度优先搜索
  - 深度优先搜索（1）

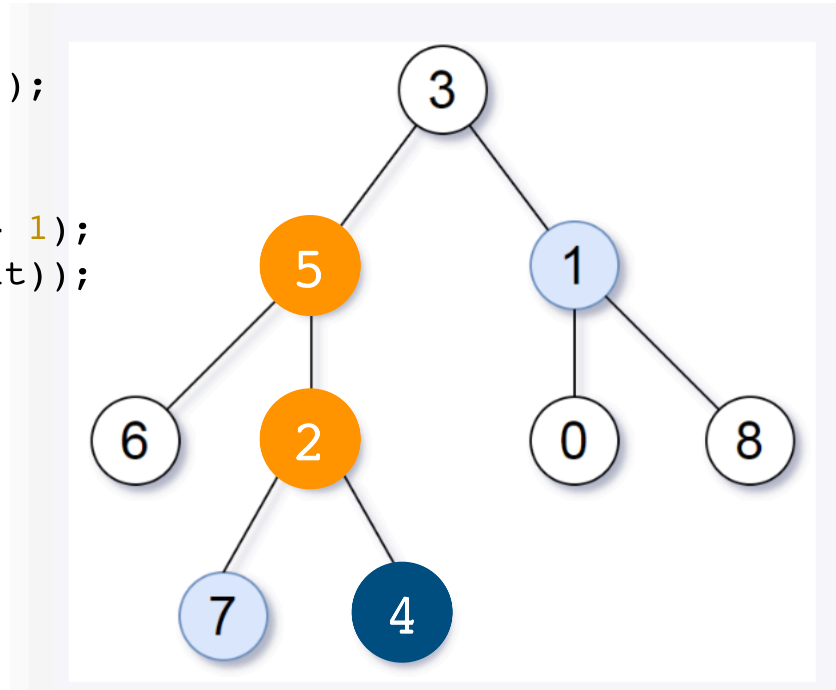
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

到达目标遍历深度





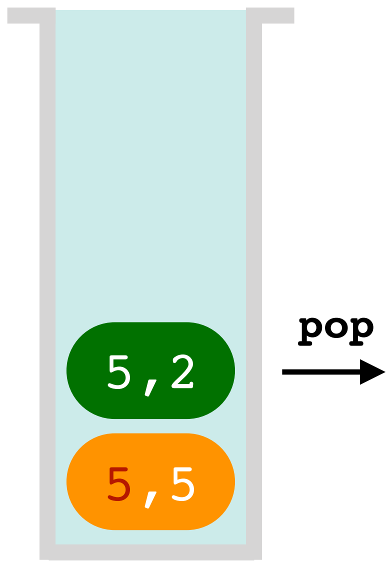
- 深度优先搜索
  - 深度优先搜索（1）

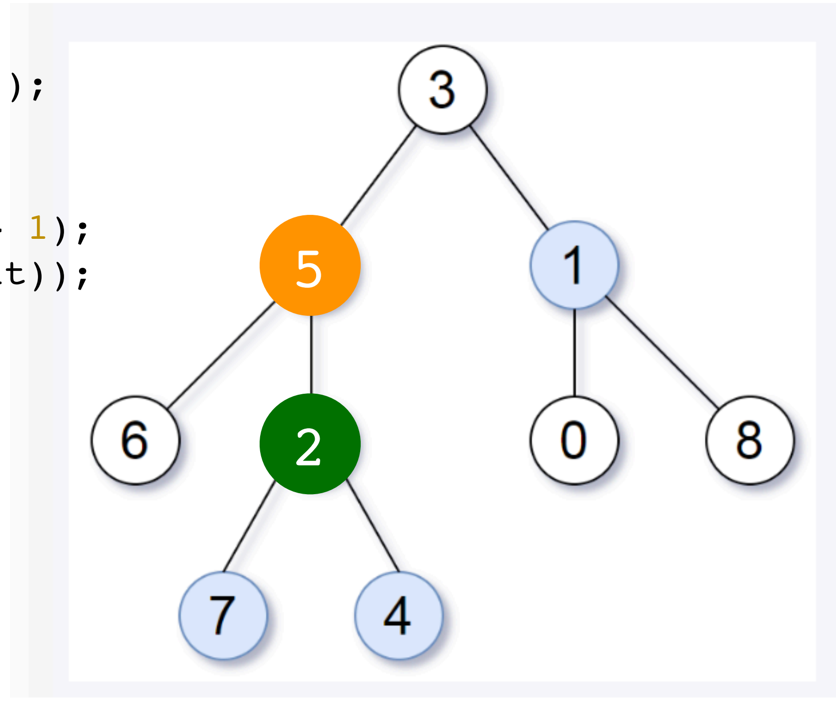
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```

遍历结束





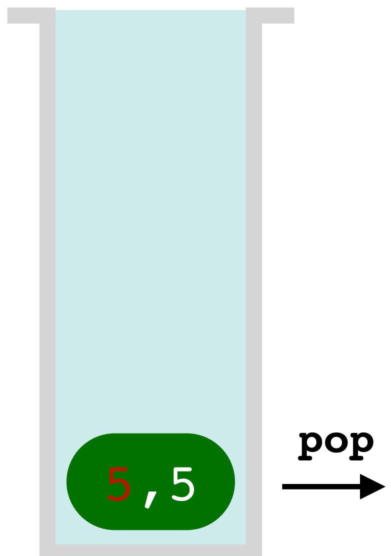
- 深度优先搜索
  - 深度优先搜索（1）

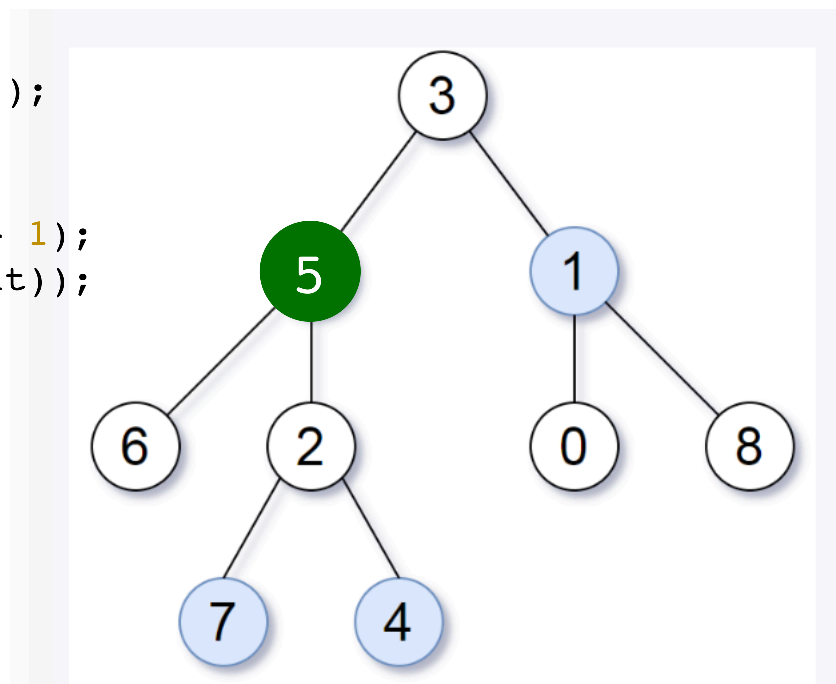
LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}
```

遍历结束

```
std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```



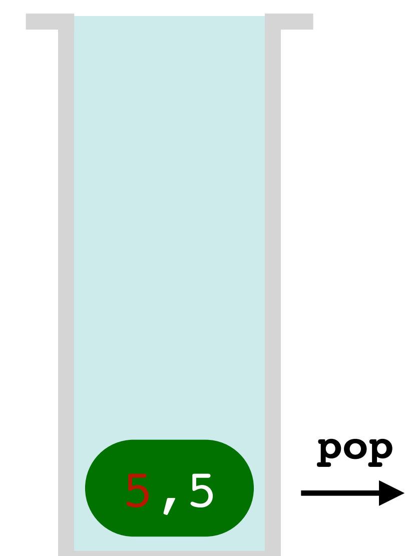


- 深度优先搜索
    - 深度优先搜索（1）

LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

```
std::vector<int> dfs(const std::vector<int>& tree, int parent, int target, int K) {
    std::vector<int> result;
    if (target >= 0 && target < tree.size() && tree[target] >= 0) {
        // 若该节点合法
        if (K == 0) {
            // 若到达遍历深度，则停止遍历，记录当前节点
            result.push_back(target);
        } else {
            // 否则，遍历其父节点和左、右儿子节点，并保存遍历的结果
            if (target / 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_parent = dfs(tree, target, target / 2, K - 1);
                copy(part_parent.begin(), part_parent.end(), std::back_inserter(result));
            }
            if (target * 2 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_left = dfs(tree, target, target * 2, K - 1);
                copy(part_left.begin(), part_left.end(), std::back_inserter(result));
            }
            if (target * 2 + 1 != parent) { // 判断目标节点是否已经被遍历过
                std::vector<int> part_right = dfs(tree, target, target * 2 + 1, K - 1);
                copy(part_right.begin(), part_right.end(), std::back_inserter(result));
            }
        }
    }
    return std::move(result);
}

std::vector<int> distanceK(const std::vector<int>& tree, int target, int K) {
    return dfs(tree, target, target, K);
}
```



总结：

1) 完全二叉树的线性表表示；

2) 深度优先遍历“树”结构；

3) 在函数栈中，记录深度信息（K）和父节点信息；

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

Example 1:

```
    2
   / \
  1   3

Input: [2,1,3]
Output: true
```

Example 2:

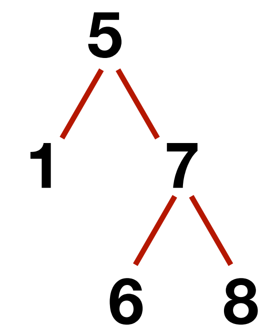
```
    5
   / \
  1   4
     / \
    3   6

Input: [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

思路一：
1) 递归访问二叉树；
2) 对于每个节点，判断其左儿子是否小于该节点；
3) 对于每个节点，判断其右儿子是否大于该节点；

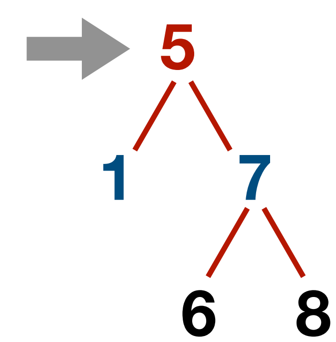


[5, 1, 7, N, N, 6, 8]

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

思路一：
1) 递归访问二叉树；
2) 对于每个节点，判断其左儿子是否小于该节点；
3) 对于每个节点，判断其右儿子是否大于该节点；

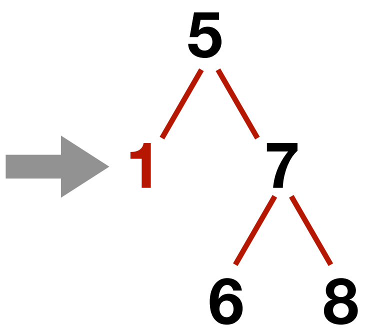


[5, 1, 7, N, N, 6, 8]

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

思路一：
1) 递归访问二叉树；
2) 对于每个节点，判断其左儿子是否小于该节点；
3) 对于每个节点，判断其右儿子是否大于该节点；

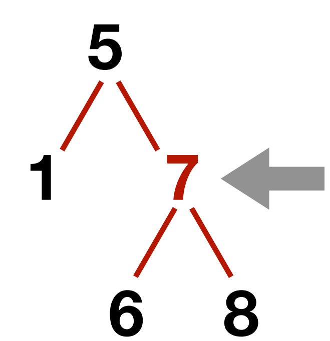


[5, 1, 7, N, N, 6, 8]

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

思路一：
1) 递归访问二叉树；
2) 对于每个节点，判断其左儿子是否小于该节点；
3) 对于每个节点，判断其右儿子是否大于该节点；

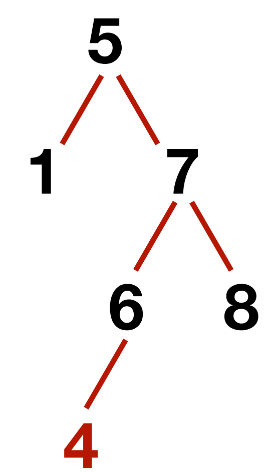


[5, 1, 7, N, N, 6, 8]

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

思路一：
1) 递归访问二叉树；
2) 对于每个节点，判断其左儿子是否小于该节点；
3) 对于每个节点，判断其右儿子是否大于该节点；

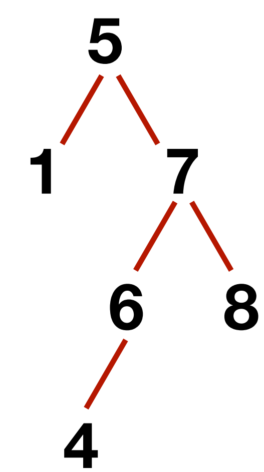


[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

思路二：
1) 递归访问二叉树；
2) 对于每个节点，若其左儿子存在，则先遍历左儿子；否则，遍历该节点；最后，若其右儿子存在，则先遍历右儿子；
3) 判断依次遍历的节点是否递增。

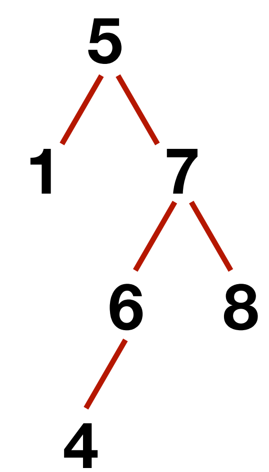


[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

- 深度优先搜索
  - 深度优先搜索（2）

    LeetCode 98. <Validate Binary Search Tree>
    https://leetcode.com/problems/validate-binary-search-tree/

```cpp
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```

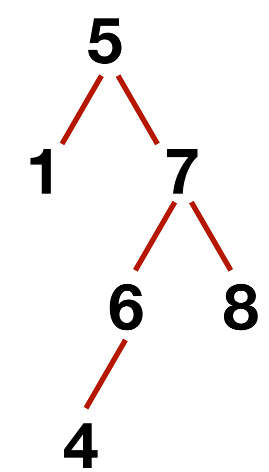


[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

- 深度优先搜索
    - 深度优先搜索（2）

      LeetCode 98. <Validate Binary Search Tree>
      https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



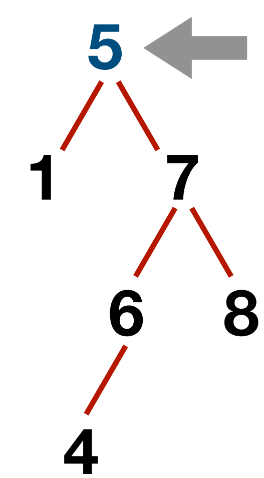
[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

prev_min = LONG_MIN

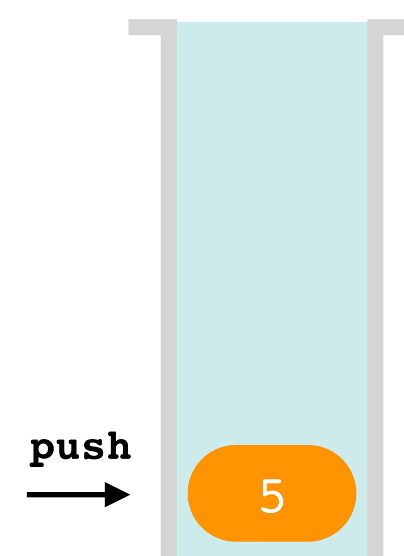
- 深度优先搜索
  - 深度优先搜索（2）

    LeetCode 98. <Validate Binary Search Tree>
    https://leetcode.com/problems/validate-binary-search-tree/

```cpp
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

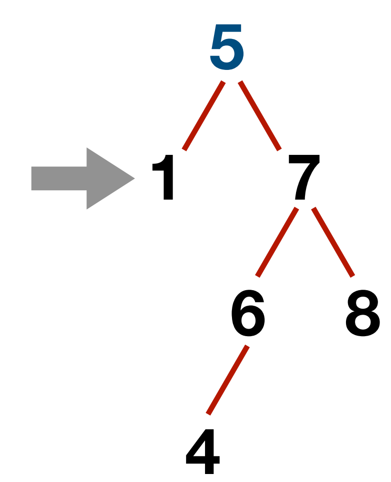


`prev_min = LONG_MIN`

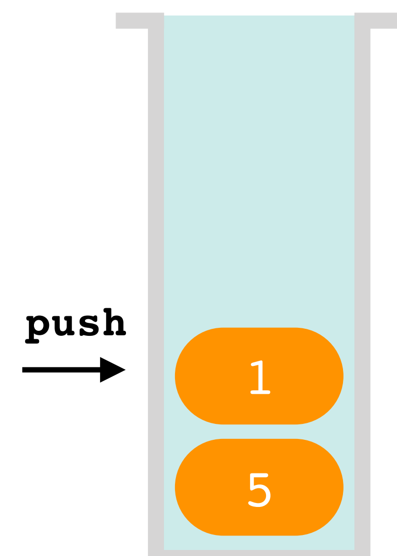
- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

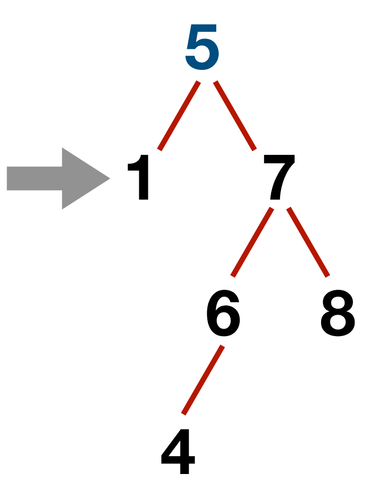


prev_min = LONG_MIN

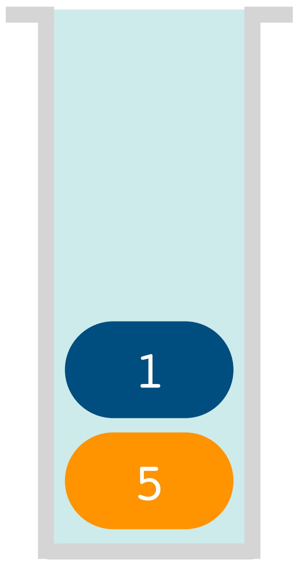
- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

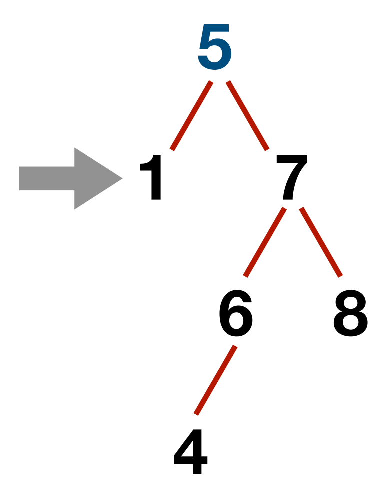


prev_min = 1

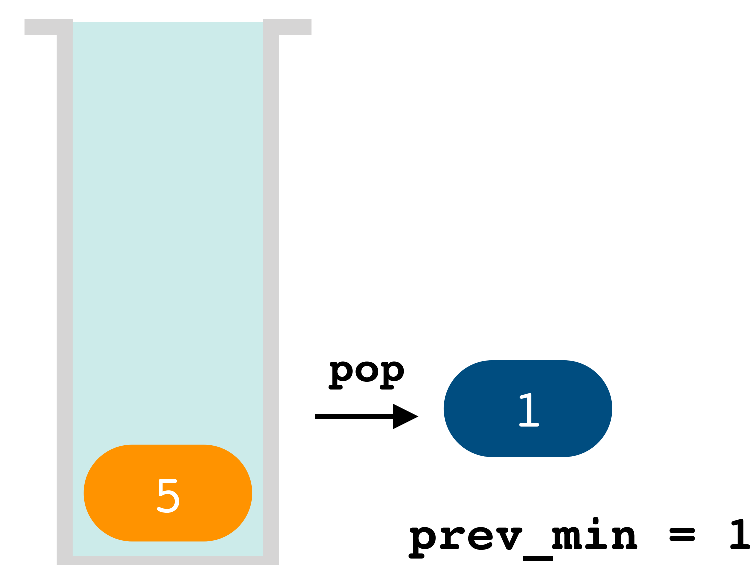
- 深度优先搜索
    - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



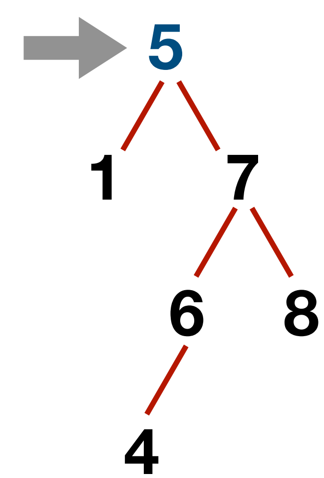
[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]



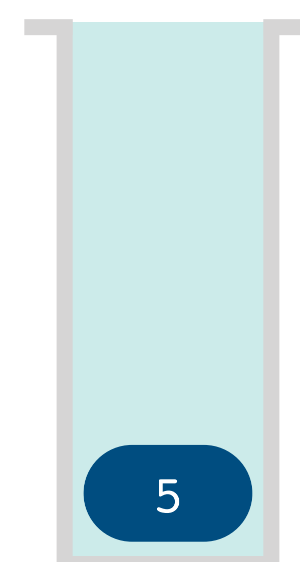
- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

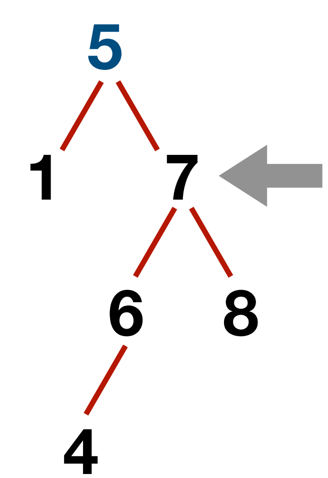


`prev_min = 5`

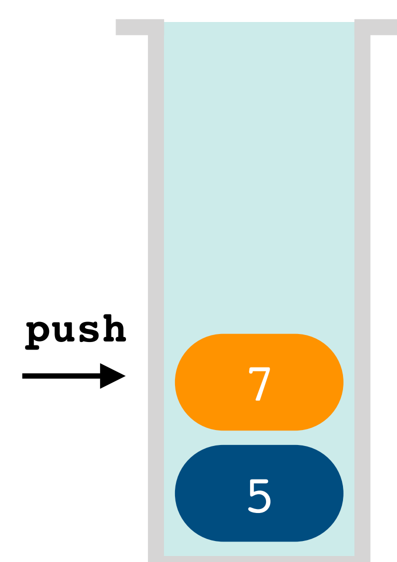
- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

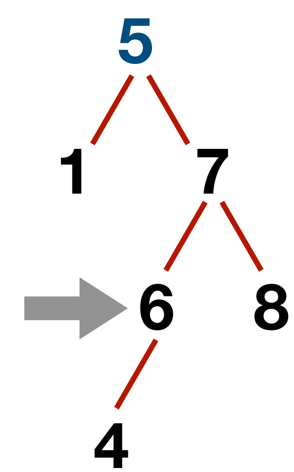


prev_min = 5

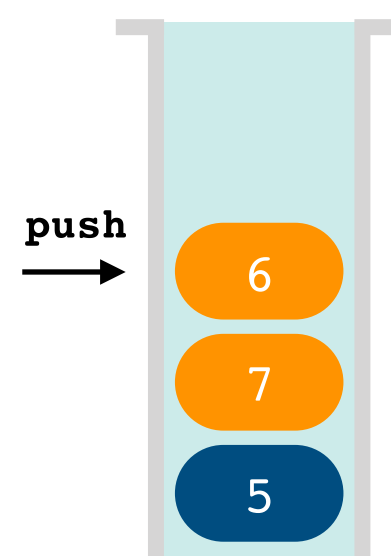
- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

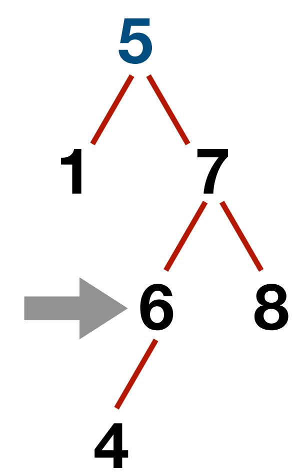


prev_min = 5

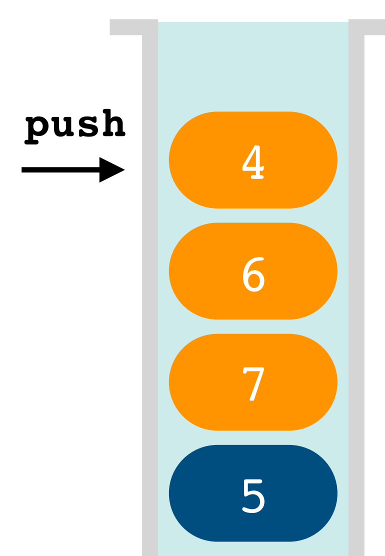
- 深度优先搜索
    - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```cpp
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

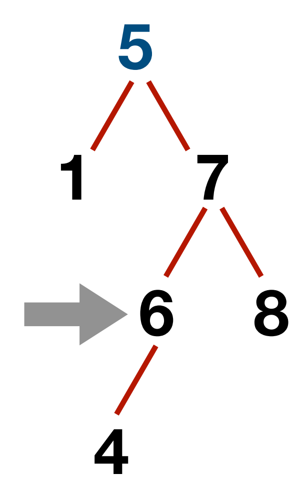


`prev_min = 5`

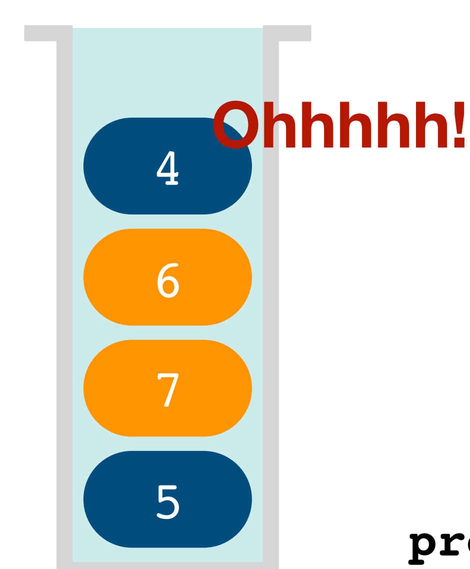
- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

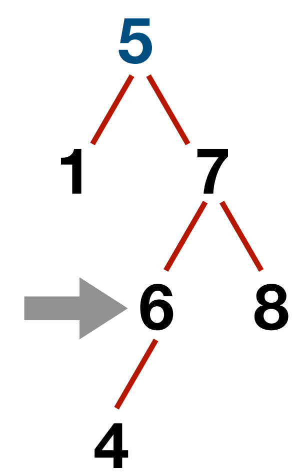


`prev_min = 5`

- 深度优先搜索
  - 深度优先搜索（2）

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

```
bool isValidBST(const std::vector<int>& tree, int root, int* prev_min) {
    if (root < tree.size() && tree[root] < 0) { // tree[root] is NULL
        return true;
    }
    // 遍历左儿子，若该节点左儿子存在，则判断其左测子树是否为 BST，并返回左测最大值
    if (!isValidBST(tree, root * 2, prev_min)) {
        return false;
    }
    // 判断该节点的值是否超过左测的最大值
    if (tree[root] > *prev_min) {
        *prev_min = tree[root];
    } else {
        return false;
    }
    // 遍历右儿子，若该节点右儿子存在，则判断其右测子树是否为 BST，并返回右测最大值
    if (!isValidBST(tree, root * 2 + 1, prev_min)) {
        return false;
    }
    return true;
}
```



[5, 1, 7, N, N, 6, 8, N, N, N, N, 4, N, N, N]

总结：

1) 理解 二叉排序树 的性质；

2) 理解二叉树的先序遍历；

扩展练习

LeetCode 863. <All Nodes Distance K in Binary Tree>
https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

LeetCode 98. <Validate Binary Search Tree>
https://leetcode.com/problems/validate-binary-search-tree/

LeetCode 127. <Word Ladder>
https://leetcode.com/problems/word-ladder/