字典树（Trie）

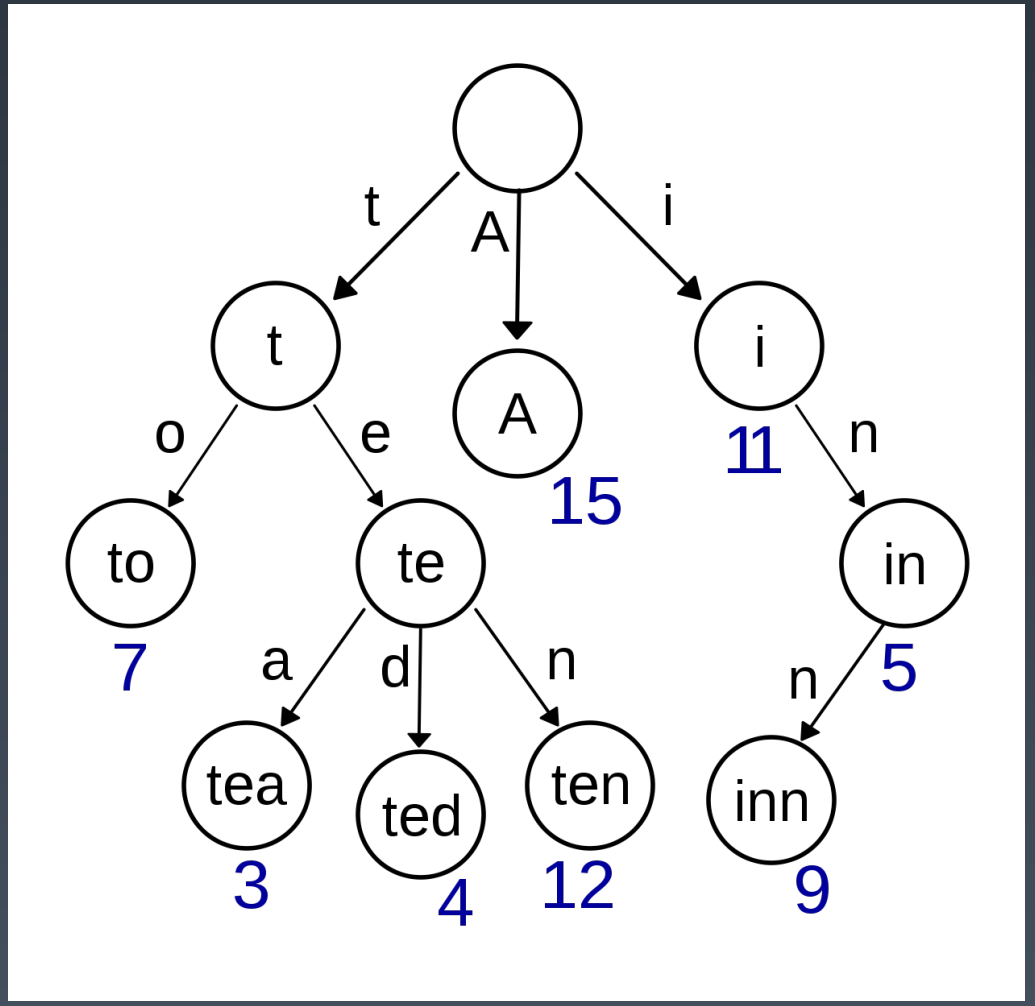
本节内容

1. Trie树的数据结构；
2. Trie树的核心思想；
3. Trie树的基本性质。

基本结构

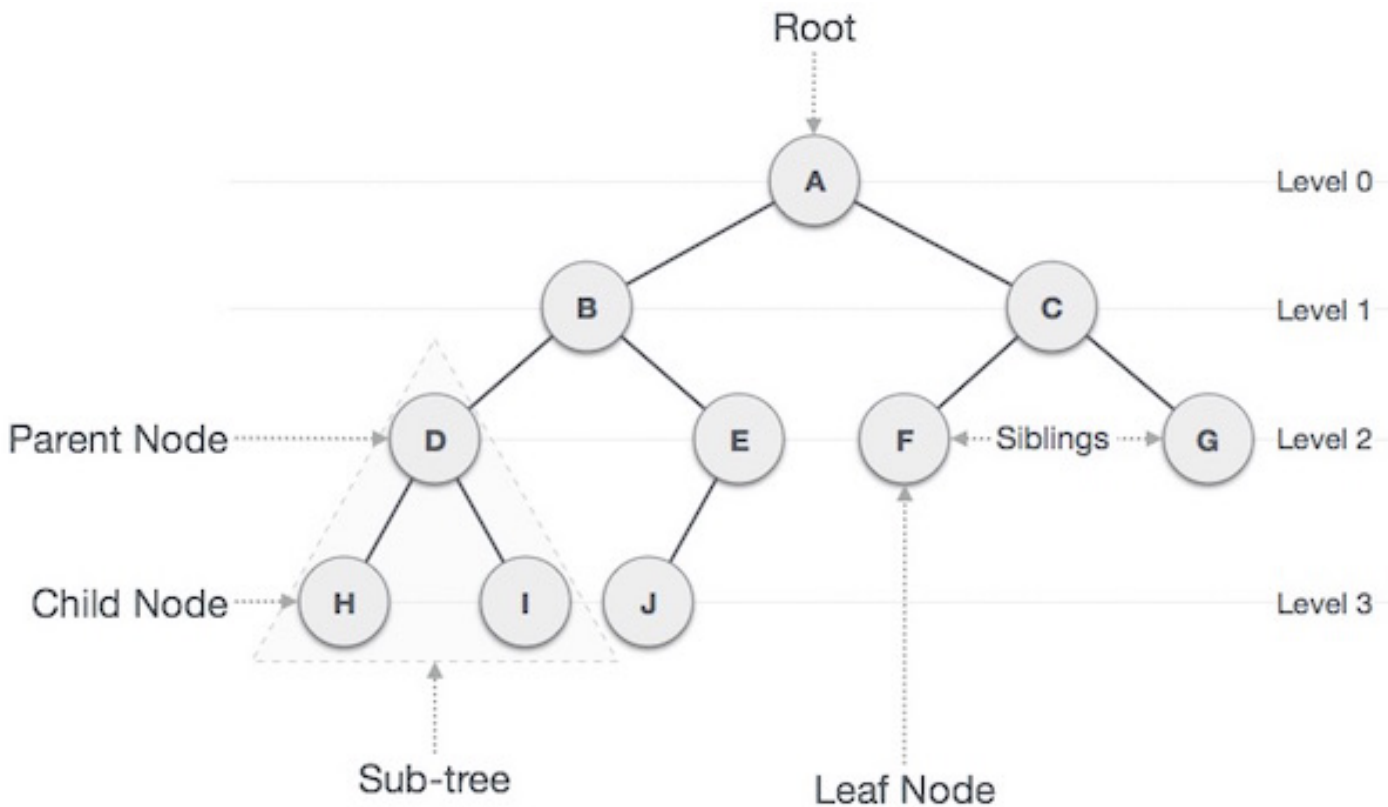
Trie树，即字典树，又称单词查找树或键树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。

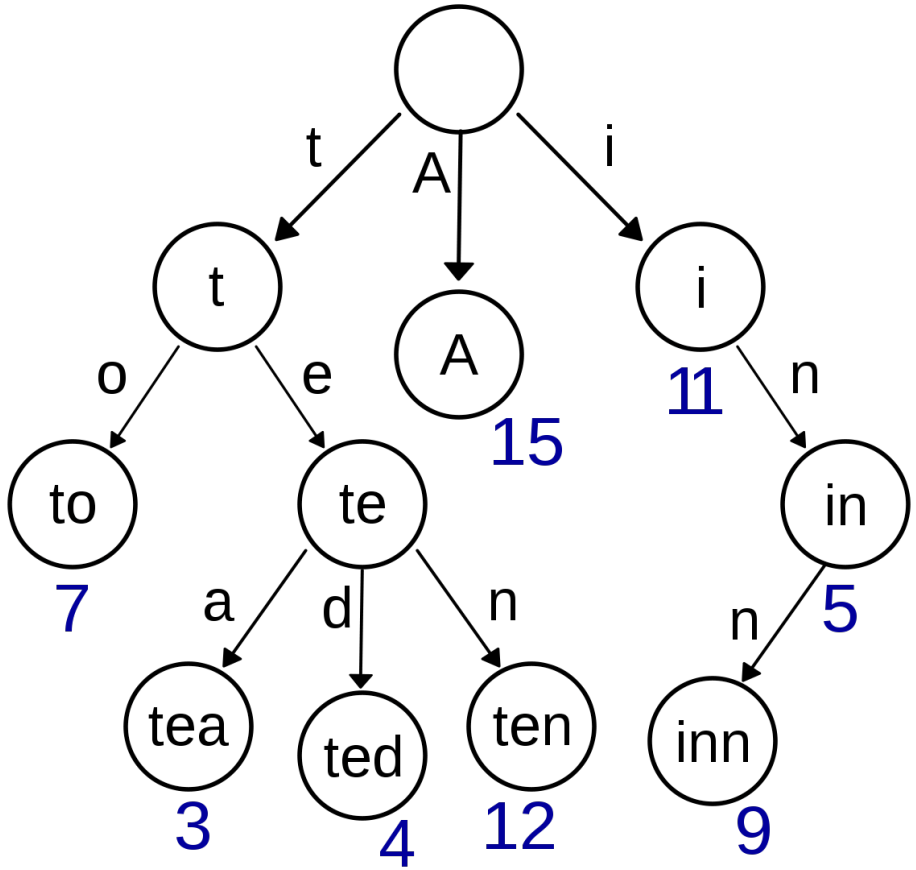
它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。

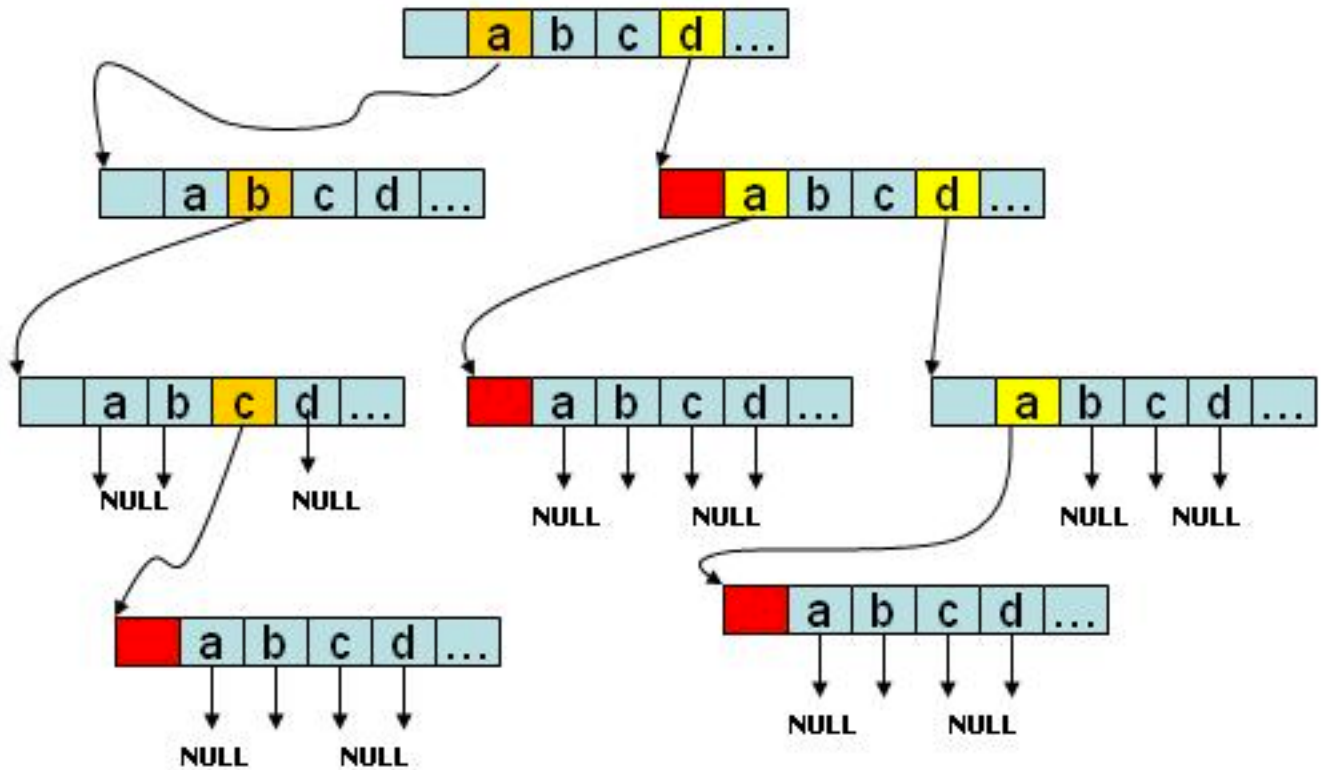


核心思想

Trie的核心思想是空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。


Root
A
B
C
D
E
F
G
H
I
J
Level 0
Level 1
Level 2
Level 3
Parent Node
Child Node
Siblings
Sub-tree
Leaf Node


t
A
i
t
A
i
o
e
n
11
15
to
te
in
7
a
d
n
n
5
tea
ted
ten
inn
3
4
12
9


a b c d ...
a b c d ...
a b c d ...
a b c d ...
a b c d ...
a b c d ...
a b c d ...
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL

基本性质

1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符都不相同。

预习题目（上课考察）

1.https://leetcode.com/problems/implement-trie-prefix-tree/#/description

2. https://leetcode.com/problems/word-search-ii/

3. Search suggestion - system design

```python
class Trie(object):

    def __init__(self):
        self.root = {}
        self.end_of_word = "#"

    def insert(self, word):
        node = self.root
        for char in word:
            node = node.setdefault(char, {})
        node[self.end_of_word] = self.end_of_word

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node:
                return False
            node = node[char]
        return self.end_of_word in node

    def startsWith(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node:
                return False
            node = node[char]
        return True
```

```java
class TrieNode {
    public char val;
    public boolean isWord;
    public TrieNode[] children = new TrieNode[26];
    public TrieNode() {}
    TrieNode(char c){
        TrieNode node = new TrieNode();
        node.val = c;
    }
}
```

```java
public class Trie {
    private TrieNode root;
    public Trie() {
        root = new TrieNode();
        root.val = ' ';
    }
    public void insert(String word) {
        TrieNode ws = root;
        for(int i = 0; i < word.length(); i++){
            char c = word.charAt(i);
            if(ws.children[c - 'a'] == null){
                ws.children[c - 'a'] = new TrieNode(c);
            }
            ws = ws.children[c - 'a'];
        }
        ws.isWord = true;
    }
    public boolean search(String word) {
        TrieNode ws = root;
        for(int i = 0; i < word.length(); i++){
            char c = word.charAt(i);
            if(ws.children[c - 'a'] == null) return false;
            ws = ws.children[c - 'a'];
        }
        return ws.isWord;
    }
    public boolean startsWith(String prefix) {
        TrieNode ws = root;
        for(int i = 0; i < prefix.length(); i++){
            char c = prefix.charAt(i);
            if(ws.children[c - 'a'] == null) return false;
            ws = ws.children[c - 'a'];
        }
        return true;
    }
}
```

```python
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

END_OF_WORD = "#"

class Solution(object):
    def findWords(self, board, words):
        if not board or not board[0]: return []
        if not words: return []

        self.result = set()

        root = collections.defaultdict()
        for word in words:
            node = root
            for char in word:
                node = node.setdefault(char, collections.defaultdict())
            node[END_OF_WORD] = END_OF_WORD

        self.m, self.n = len(board), len(board[0])

        for i in xrange(self.m):
            for j in xrange(self.n):
                if board[i][j] in root:
                    self._dfs(board, i, j, "", root)

        return list(self.result)
```

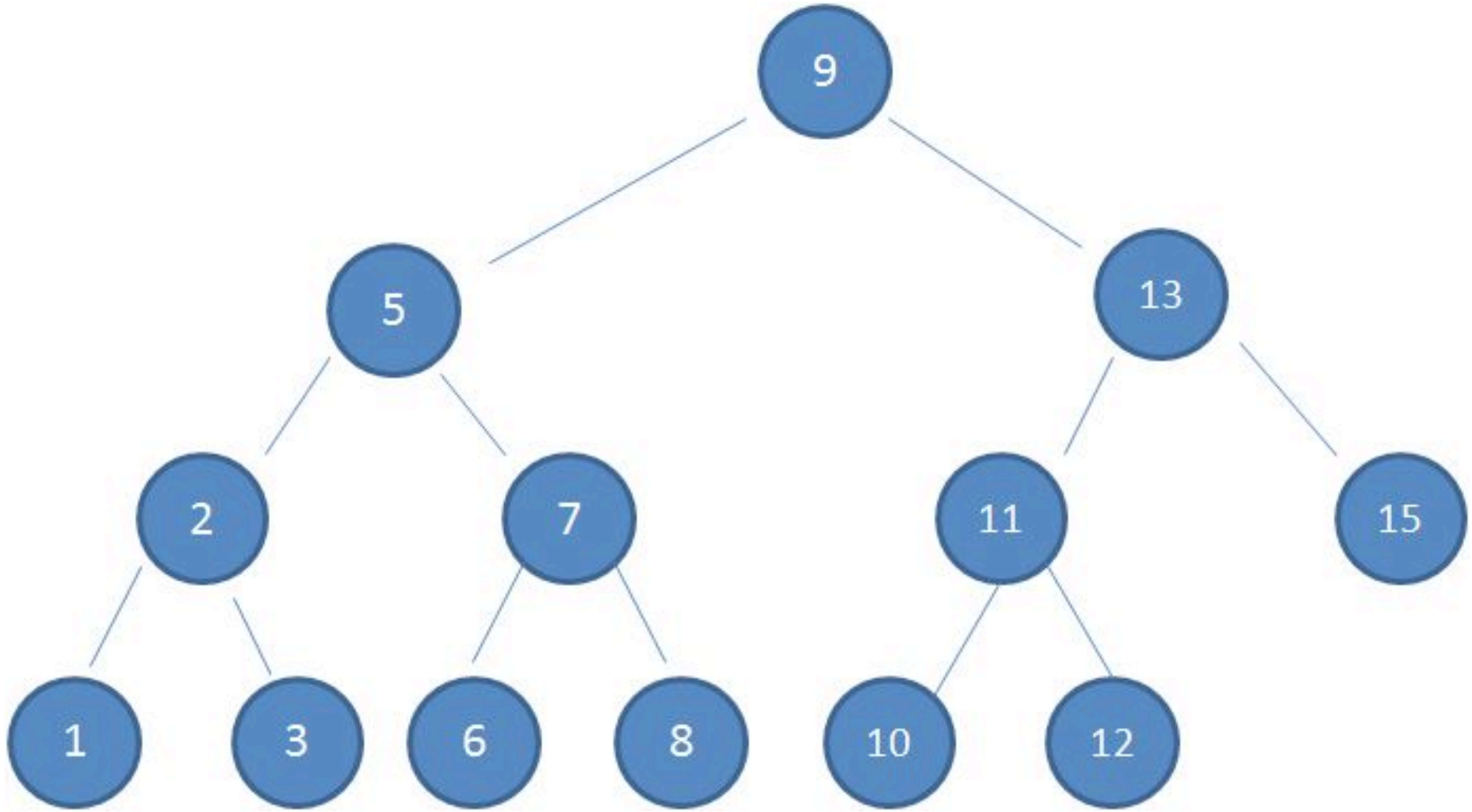
```
def _dfs(self, board, i, j, cur_word, cur_dict):

    cur_word += board[i][j]
    cur_dict = cur_dict[board[i][j]]

    if END_OF_WORD in cur_dict:
        self.result.add(cur_word)

    tmp, board[i][j] = board[i][j], '@'
    for k in xrange(4):
        x, y = i + dx[k], j + dy[k]
        if 0 <= x < self.m and 0 <= y < self.n \
            and board[x][y] != '@' and board[x][y] in cur_dict:
            self._dfs(board, x, y, cur_word, cur_dict)
    board[i][j] = tmp
```

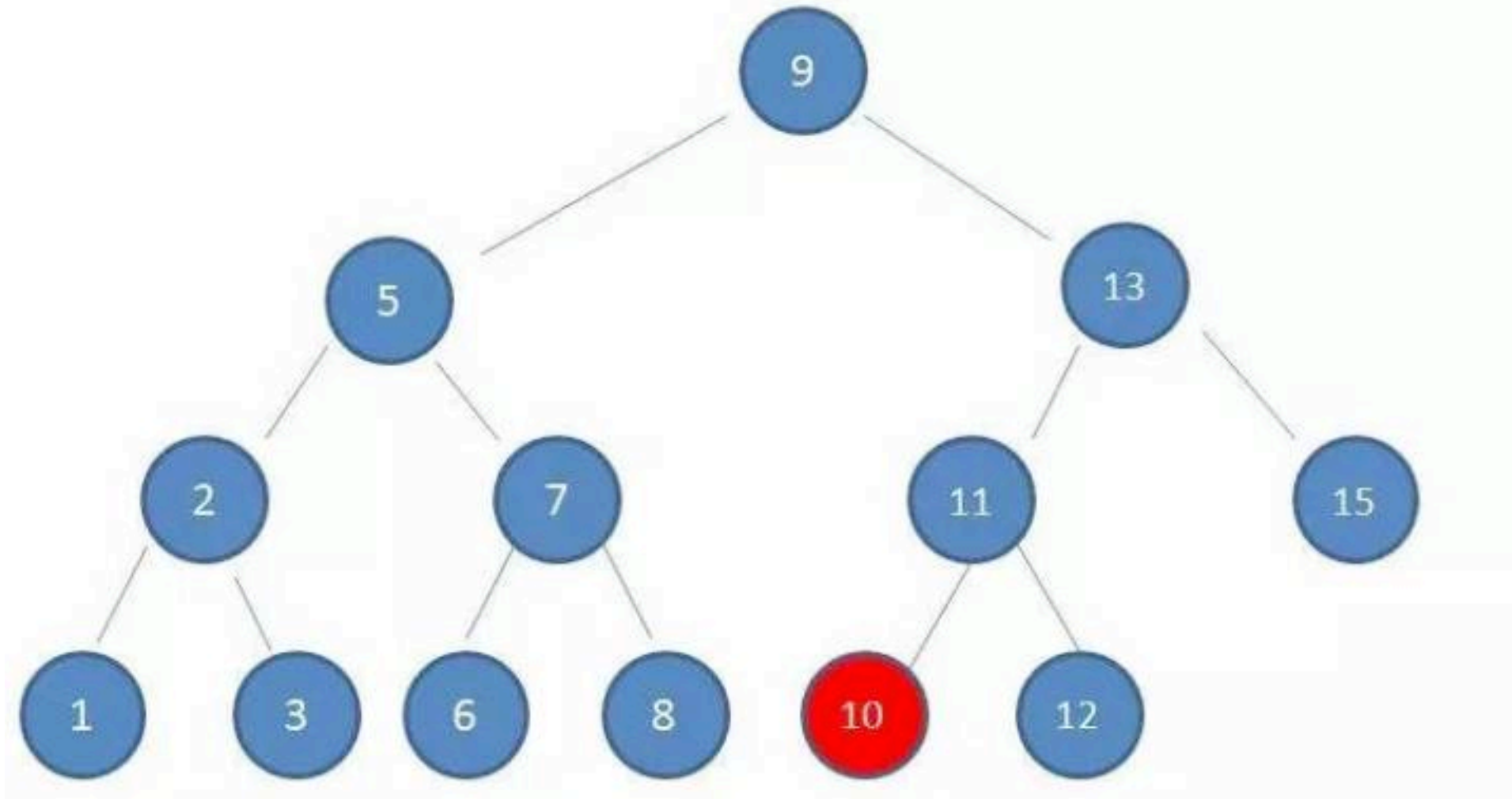
```java
public class Solution {
    Set<String> res = new HashSet<String>();

    public List<String> findWords(char[][] board, String[] words) {
        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }
        int m = board.length;
        int n = board[0].length;
        boolean[][] visited = new boolean[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                dfs(board, visited, "", i, j, trie);
            }
        }
        return new ArrayList<String>(res);
    }

    public void dfs(char[][] board, boolean[][] visited, String str, int x, int y, Trie trie) {
        if (x < 0 || x >= board.length || y < 0 || y >= board[0].length) return;
        if (visited[x][y]) return;

        str += board[x][y];
        if (!trie.startsWith(str)) return;

        if (trie.search(str)) {
            res.add(str);
        }
        visited[x][y] = true;
        dfs(board, visited, str, x - 1, y, trie);
        dfs(board, visited, str, x + 1, y, trie);
        dfs(board, visited, str, x, y - 1, trie);
        dfs(board, visited, str, x, y + 1, trie);
        visited[x][y] = false;
    }
}
```

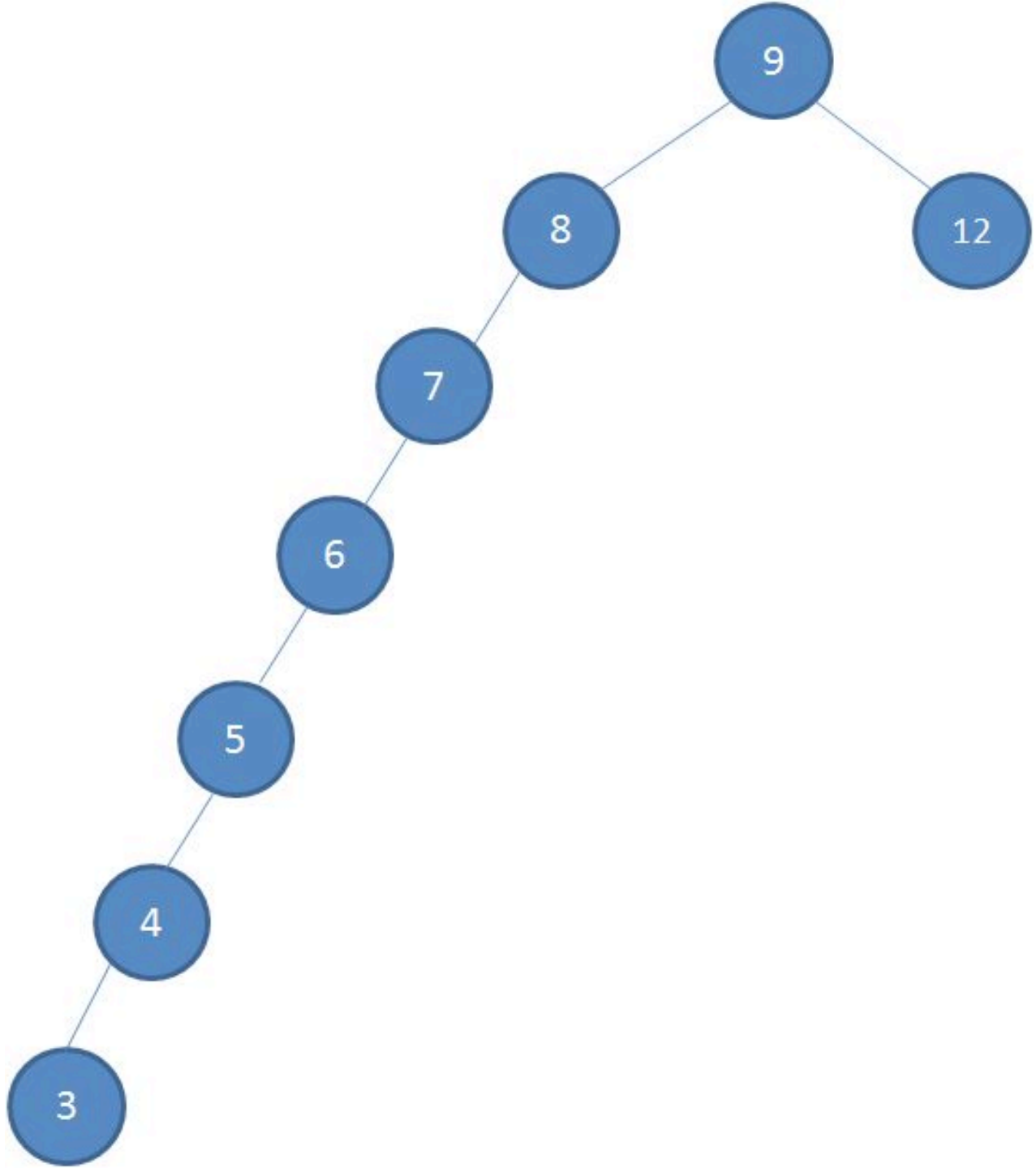
红黑树
(Red Black Tree)

存在基础

1. Binary Search Tree
2. Balanced


9
5
13
2
7
11
15
1
3
6
8
10
12

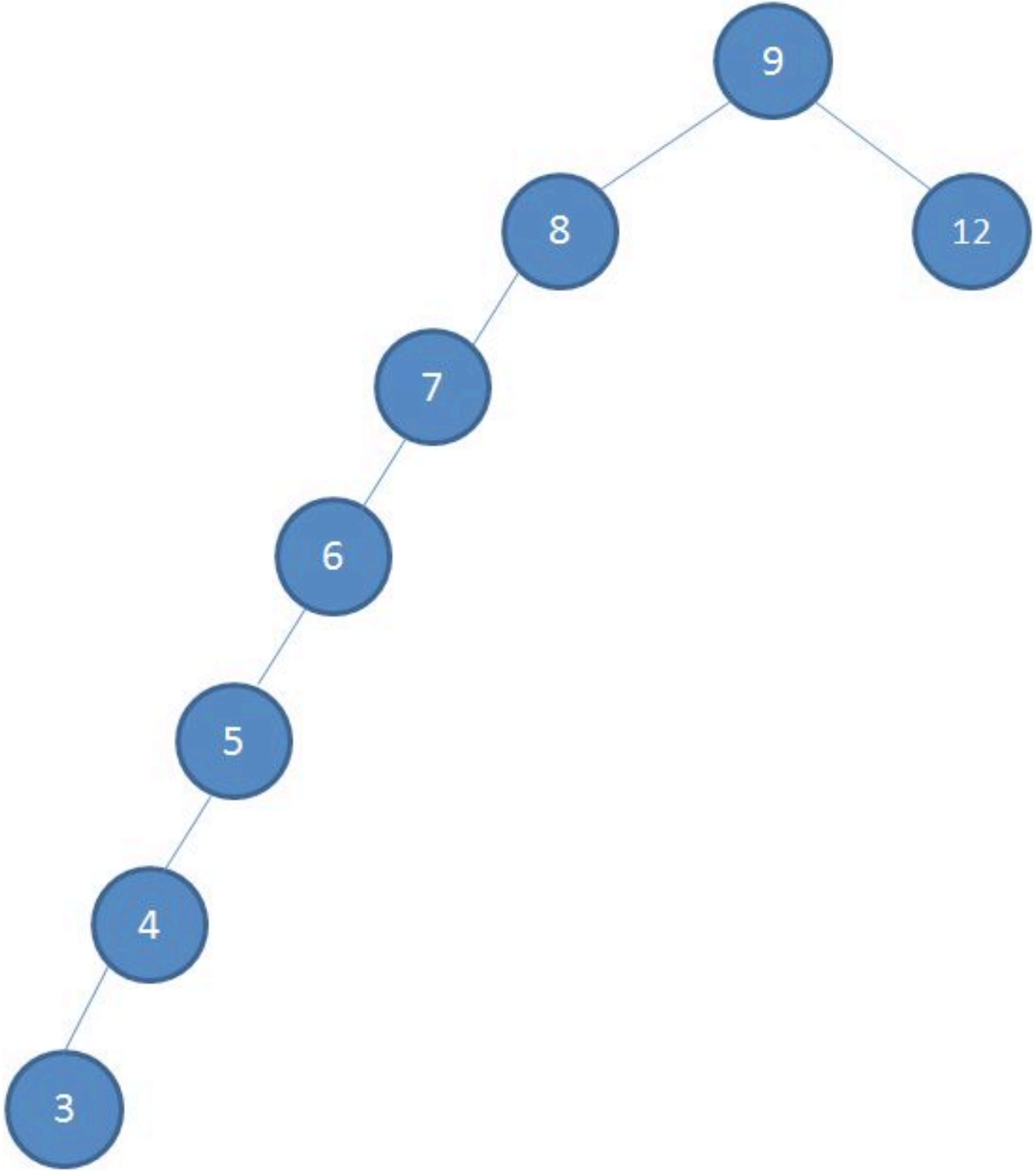

9
5
13
2
7
11
15
1
3
6
8
10
12


9
8
12
7
6
5
4
3

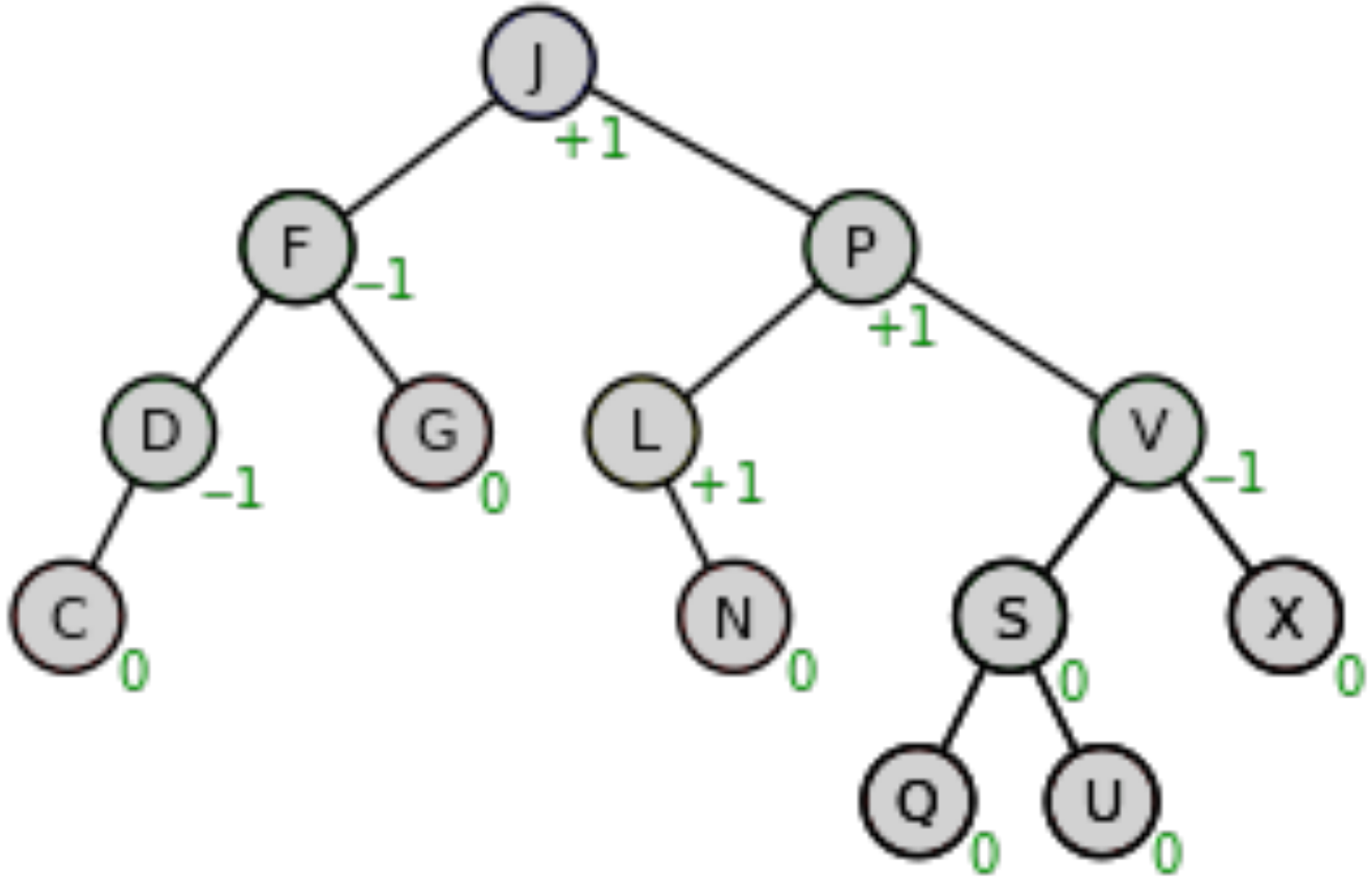
保证性能关键

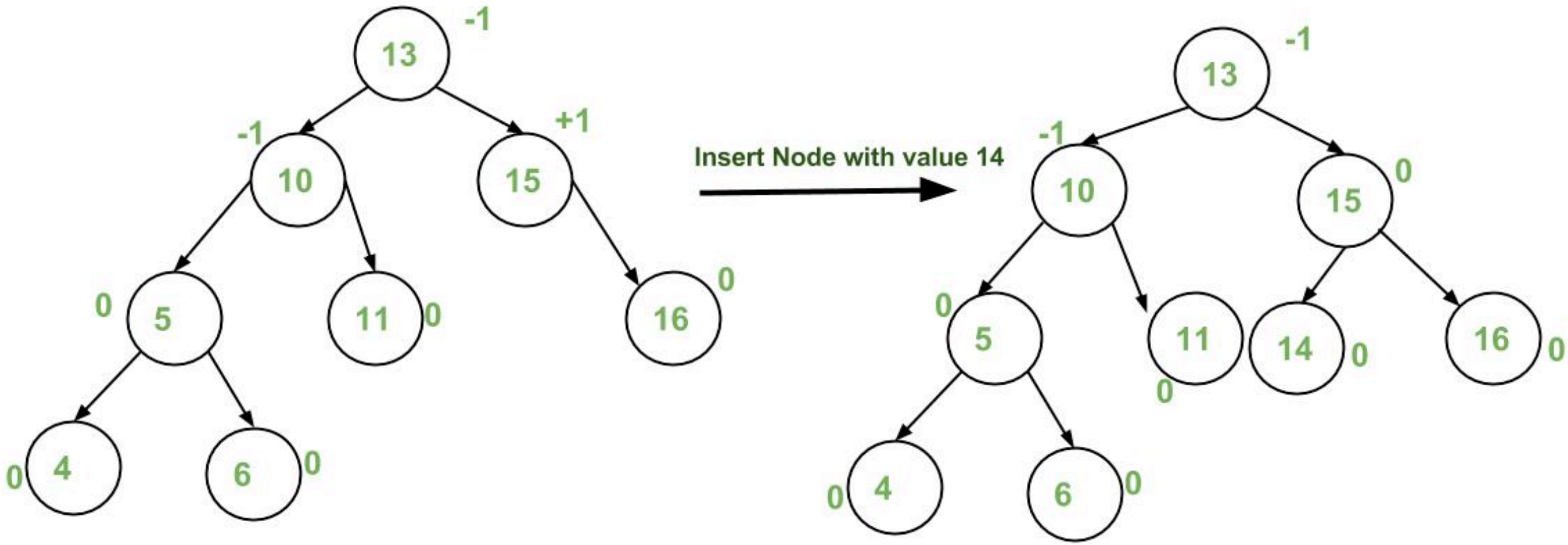
1. 保证二维维度！ —> 左右子树节点平衡（recursively）
2. Balanced
3. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

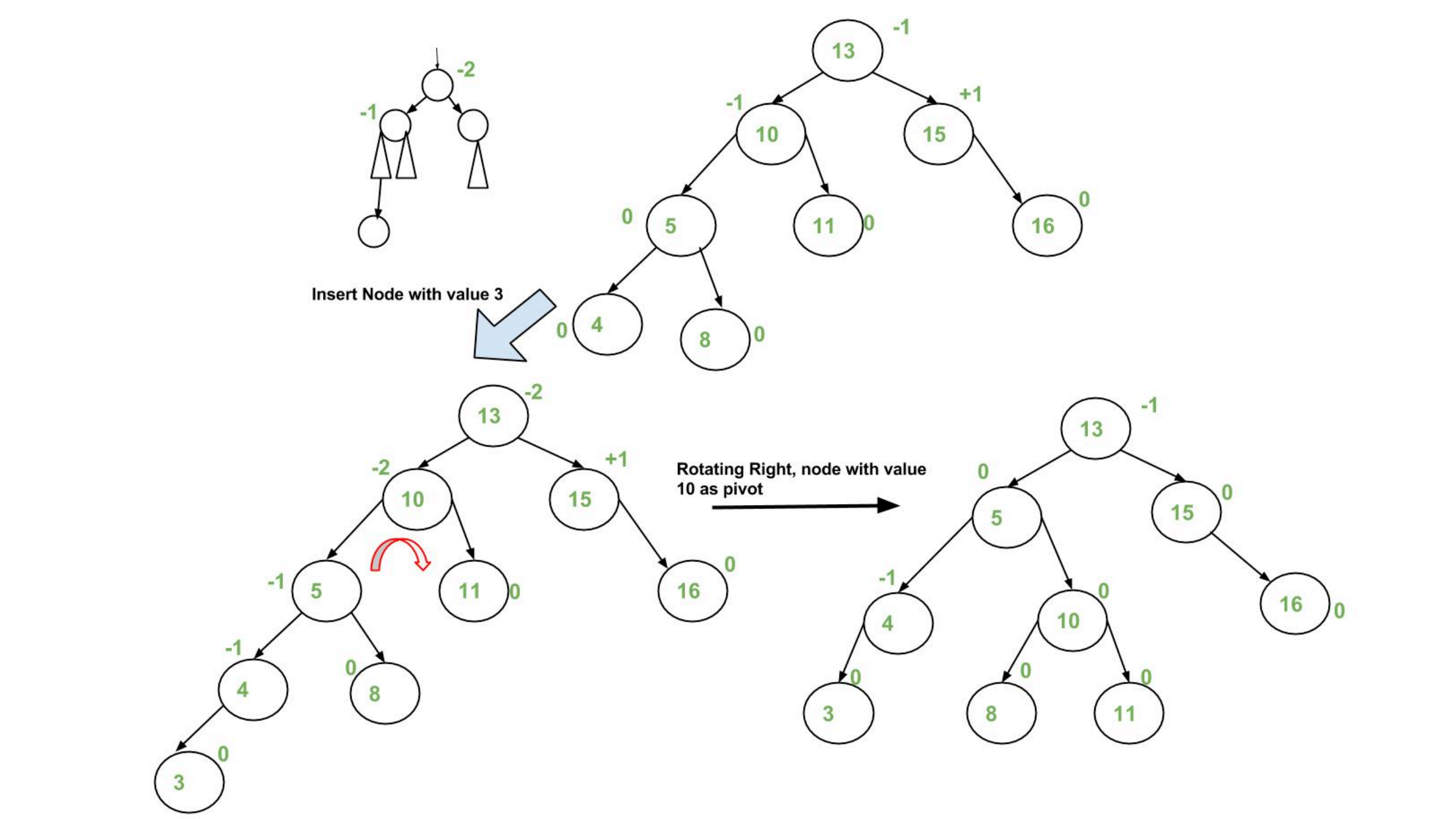
思考?
如何平衡?

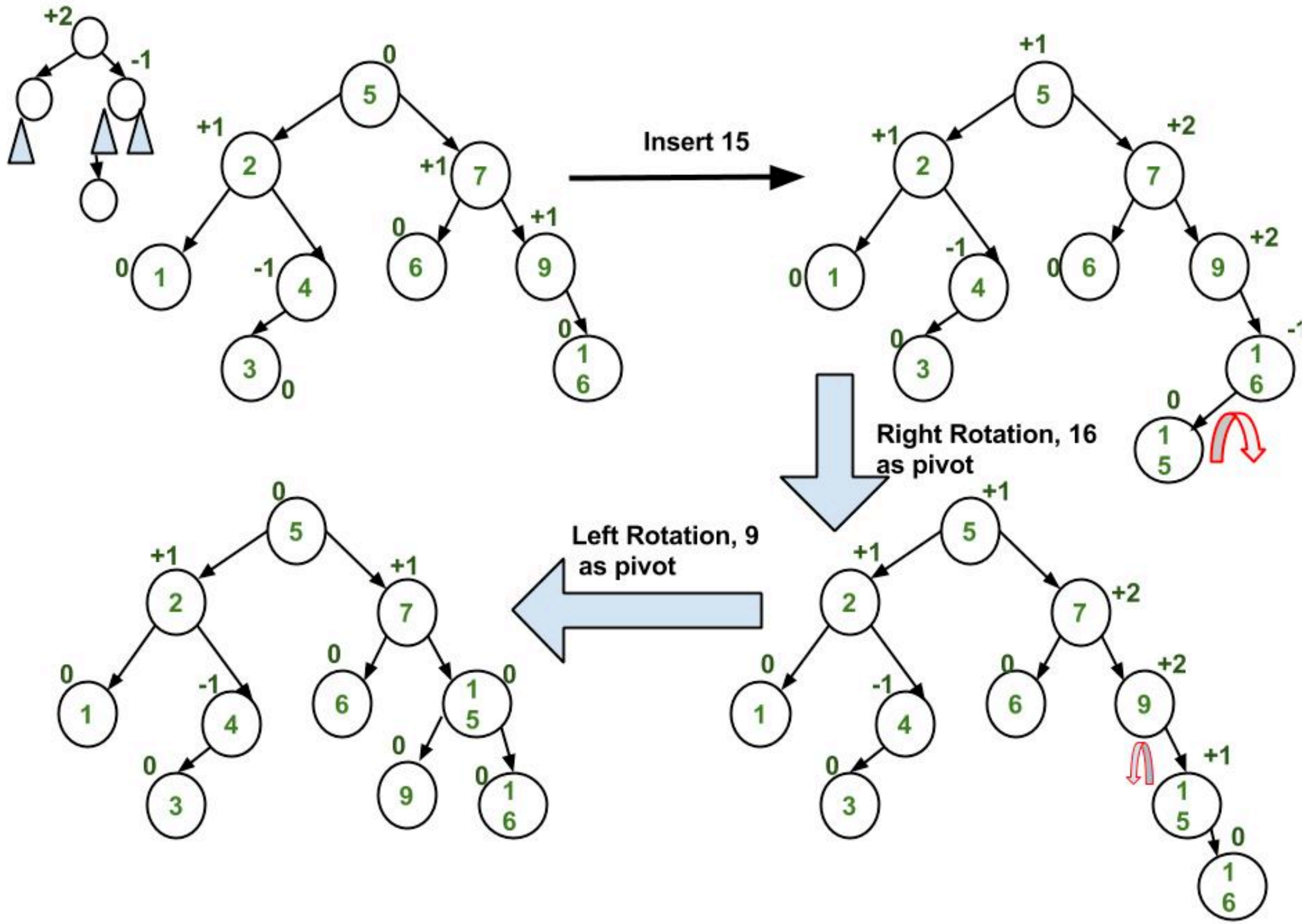


AVL Tree




13
-1
-1
10
+1
15
0
5
11
0
16
0
0
4
6
0
Insert Node with value 14
13
-1
-1
10
15
0
0
5
11
0
14
0
16
0
0
4
6
0

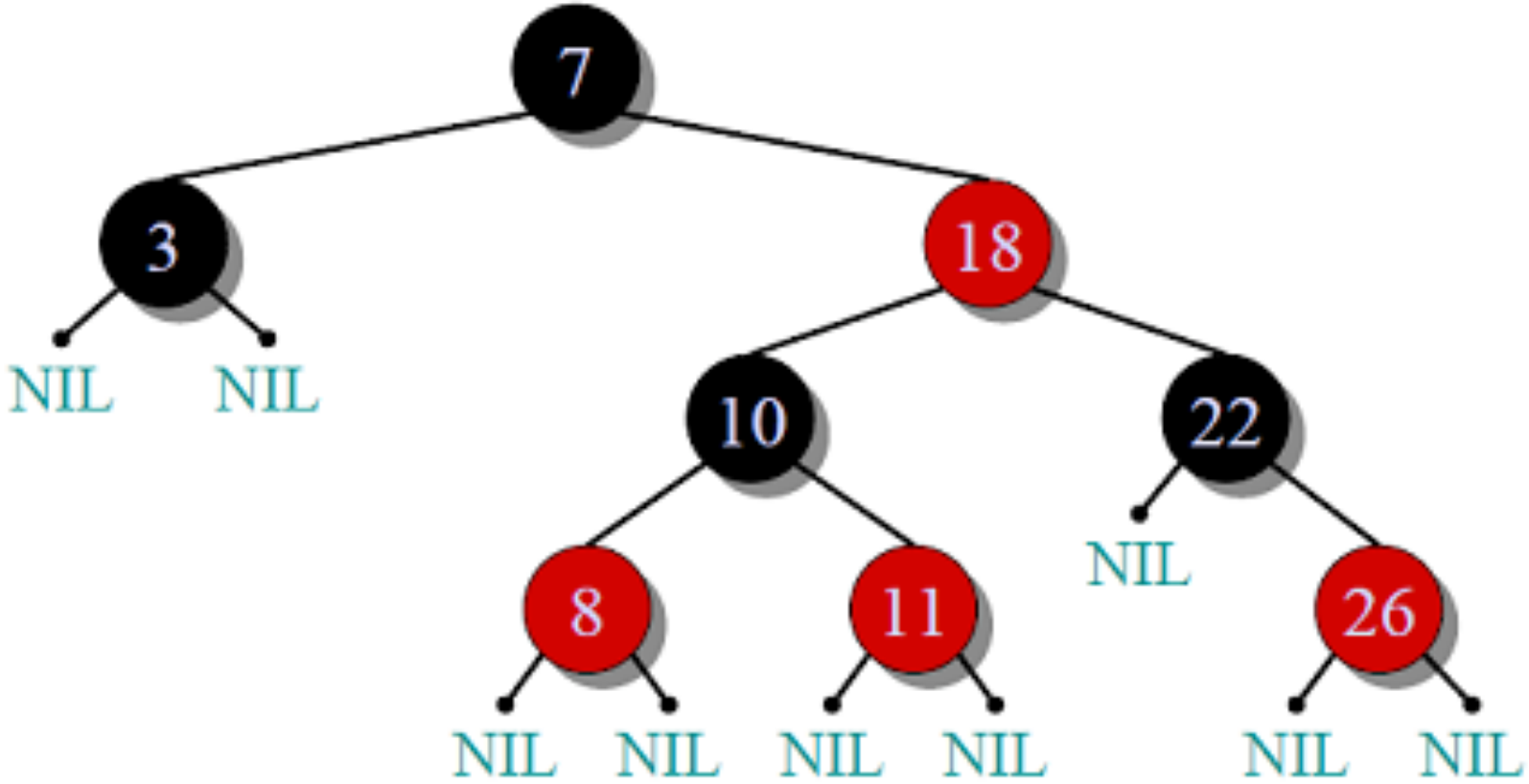

-2
-1
13
-1
-1
10
+1
15
0
5
11
0
16
0
Insert Node with value 3
0
4
8
0
13
-2
-2
10
+1
15
-1
5
11
0
16
0
-1
4
0
8
0
3
Rotating Right, node with value
10 as pivot
13
-1
0
5
15
0
-1
4
0
10
0
16
0
0
3
0
8
0
11


Insert 15
Right Rotation, 16
as pivot
Left Rotation, 9
as pivot

Red-black Tree

- 红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一陪。具体来说，红黑树是满足如下条件的二叉查找树（binary search tree）：
- 每个节点要么是红色，要么是黑色。
- 根节点必须是黑色
- 红色节点不能连续（也即是，红色节点的孩子和父亲都不能是红色）。
- 对于每个节点，从该点至null（树尾端）的任何路径，都含有相同个数的黑色节点。

Red-black Tree



对比

- AVL trees provide faster lookups than Red Black Trees because they are more strictly balanced.
- Red Black Trees provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.
- AVL trees store balance factors or heights with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.
- Red Black Trees are used in most of the language libraries like map, multimap, multisetin C++ whereas AVL trees are used in databases where faster retrievals are required.

THANKS!