199. Binary Tree Right Side View

May 28, 2020 | 8K views

Average Rating: 4.71 (14 votes)

Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.

Example:

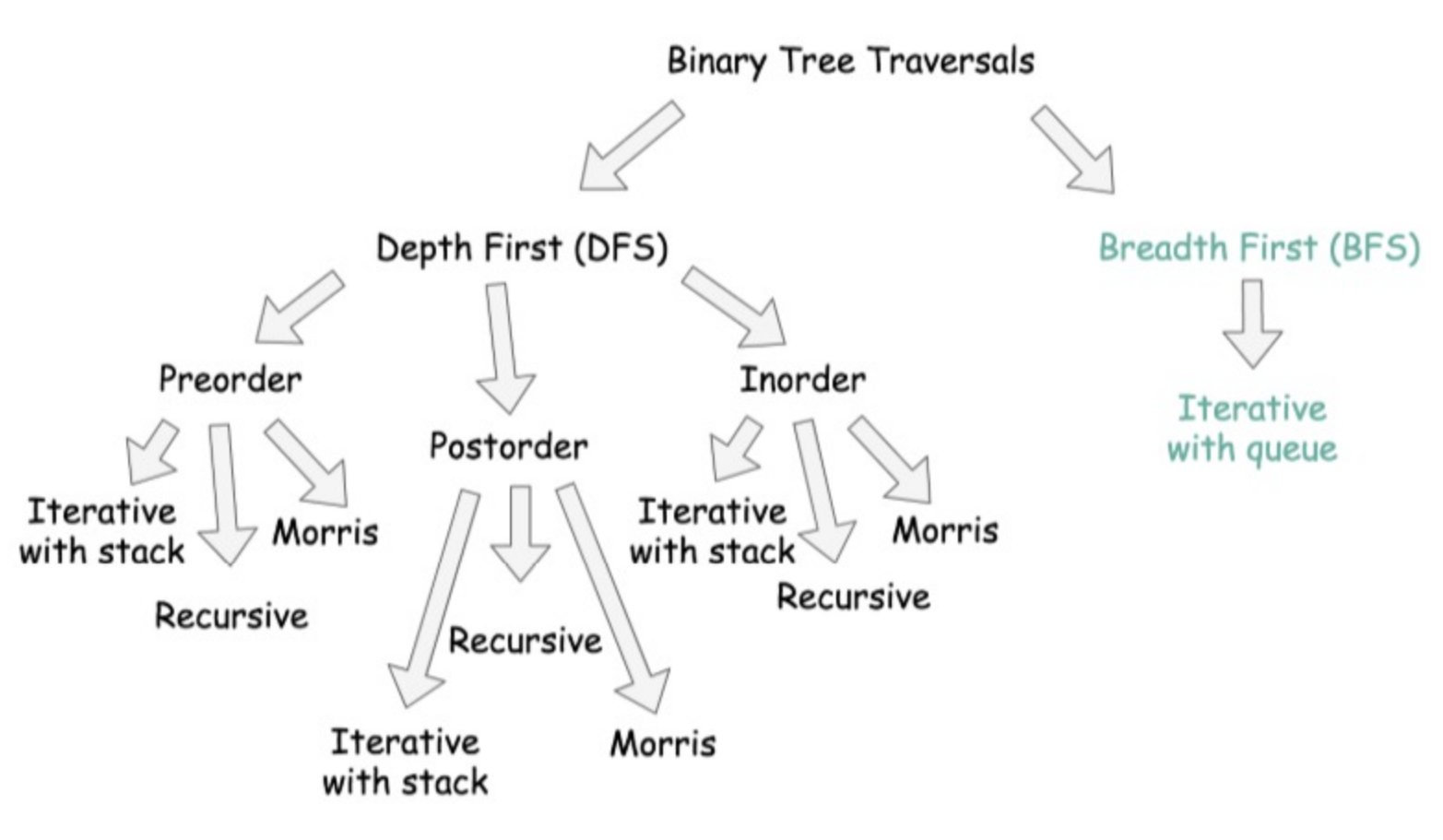
```
Input: [1,2,3,null,5,null,4]
Output: [1, 3, 4]
Explanation:

   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

Solution

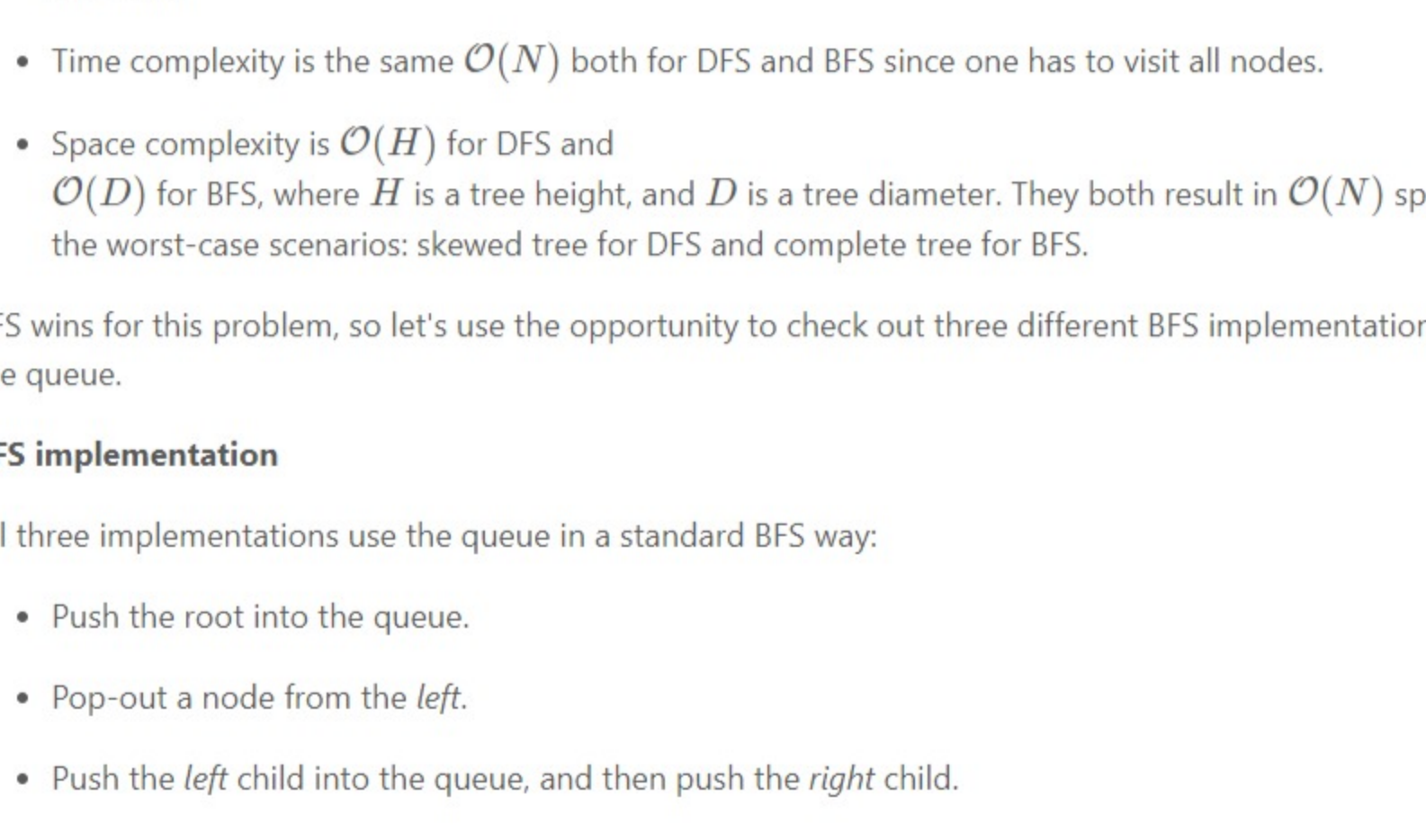
Overview

DFS vs. BFS

There are two ways to traverse the tree: DFS *depth first search* and BFS *breadth first search*. Here is a small summary



BFS traverses level by level, and DFS first goes to the leaves.



> Which approach to choose, BFS or DFS?

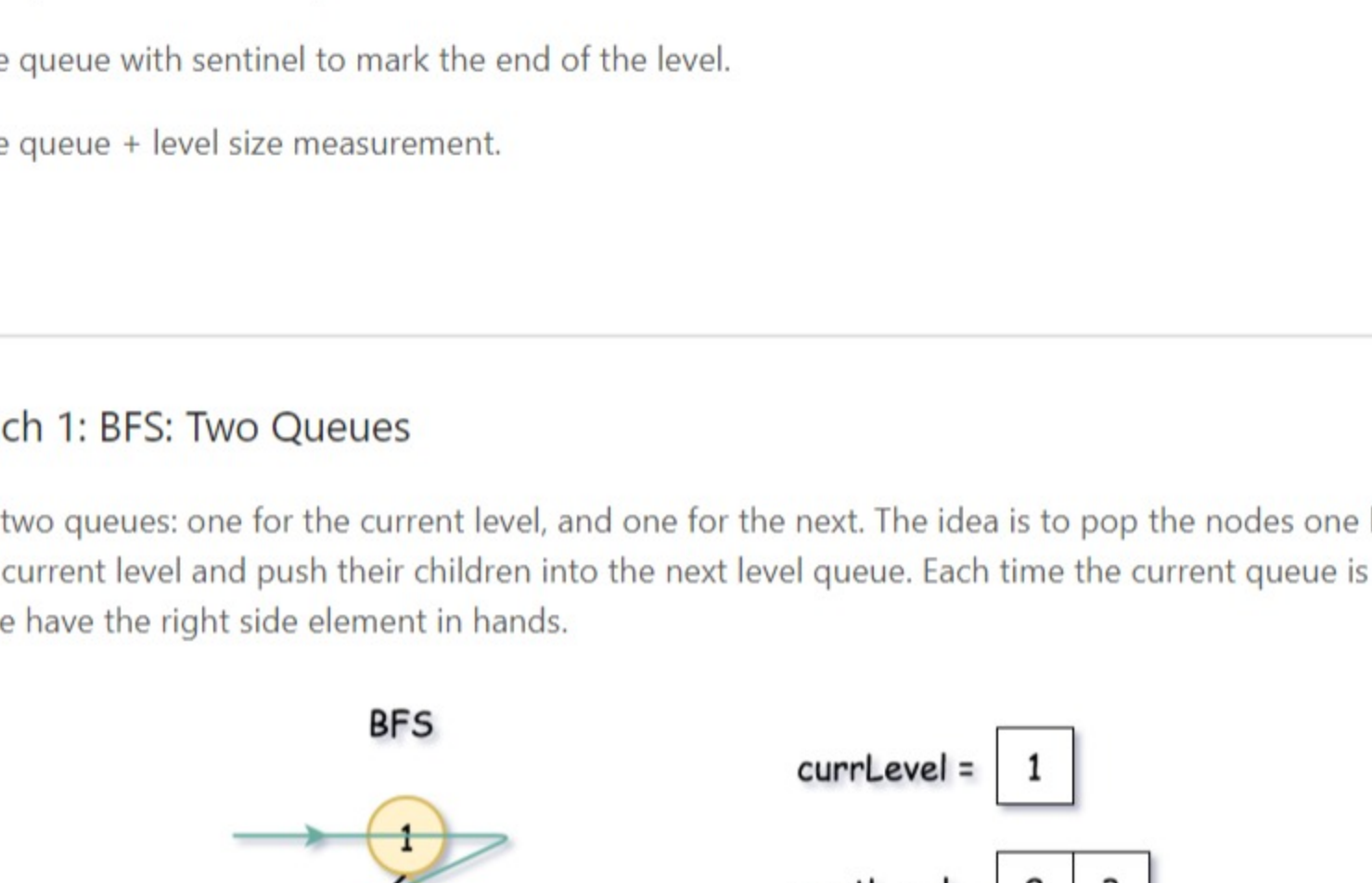
- The problem is to return a list of last elements from all levels, so it's the way more natural to implement BFS here.
- Time complexity is the same $\mathcal{O}(N)$ both for DFS and BFS since one has to visit all nodes.
- Space complexity is $\mathcal{O}(H)$ for DFS and $\mathcal{O}(D)$ for BFS, where H is a tree height, and D is a tree diameter. They both result in $\mathcal{O}(N)$ space in the worst-case scenarios: skewed tree for DFS and complete tree for BFS.

BFS wins for this problem, so let's use the opportunity to check out three different BFS implementations with the queue.

BFS implementation

All three implementations use the queue in a standard BFS way:

- Push the root into the queue.
- Pop-out a node from the *left*.
- Push the *left* child into the queue, and then push the *right* child.



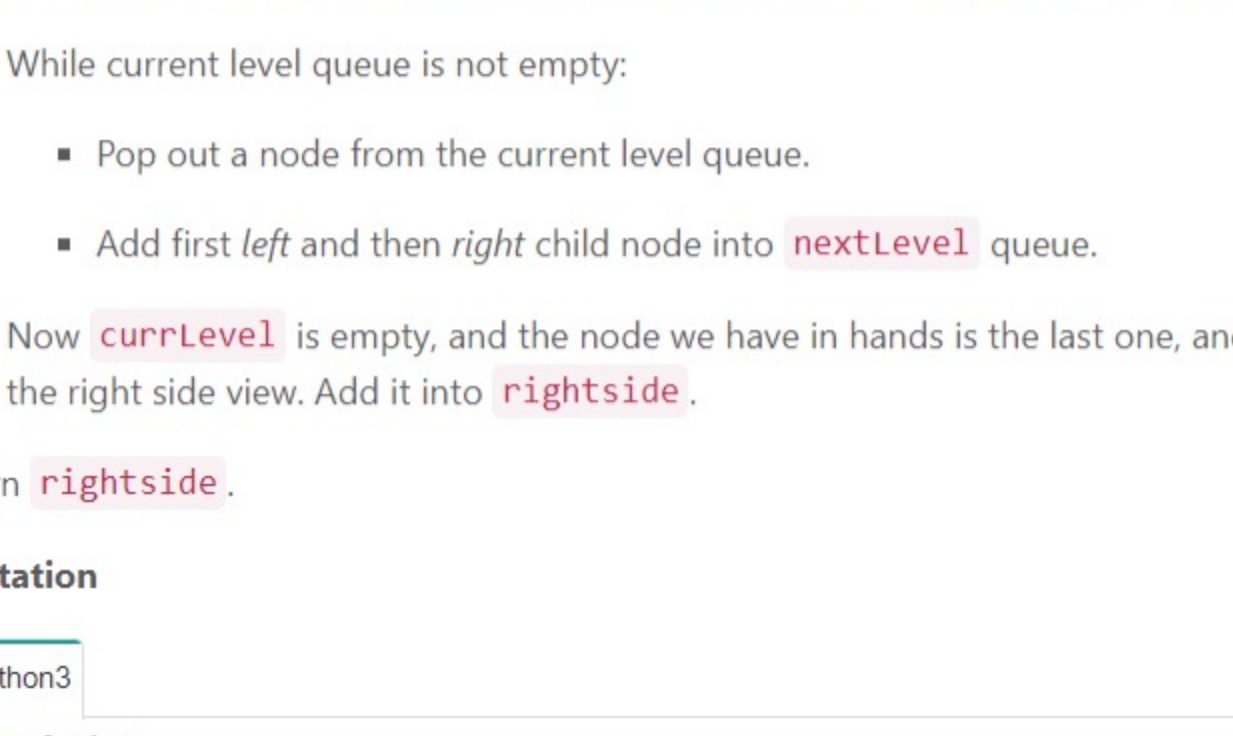
Three BFS approaches

The difference is how to find the end of the level, i.e. the rightmost element:

- Two queues, one for the previous level and one for the current.
- One queue with sentinel to mark the end of the level.
- One queue + level size measurement.

Approach 1: BFS: Two Queues

Let's use two queues: one for the current level, and one for the next. The idea is to pop the nodes one by one from the current level and push their children into the next level queue. Each time the current queue is empty, we have the right side element in hands.



Algorithm

- Initiate the list of the right side view `rightside`.
- Initiate two queues: one for the current level, and one for the next. Add root into `nextLevel` queue.
- While `nextLevel` queue is not empty:
  - Initiate the current level: `currLevel = nextLevel`, and empty the next level `nextLevel`.
  - While current level queue is not empty:
    - Pop out a node from the current level queue.
    - Add first *left* and then *right* child node into `nextLevel` queue.
  - Now `currLevel` is empty, and the node we have in hands is the last one, and makes a part of the right side view. Add it into `rightside`.
- Return `rightside`.

Implementation

Java | Python3 | Copy

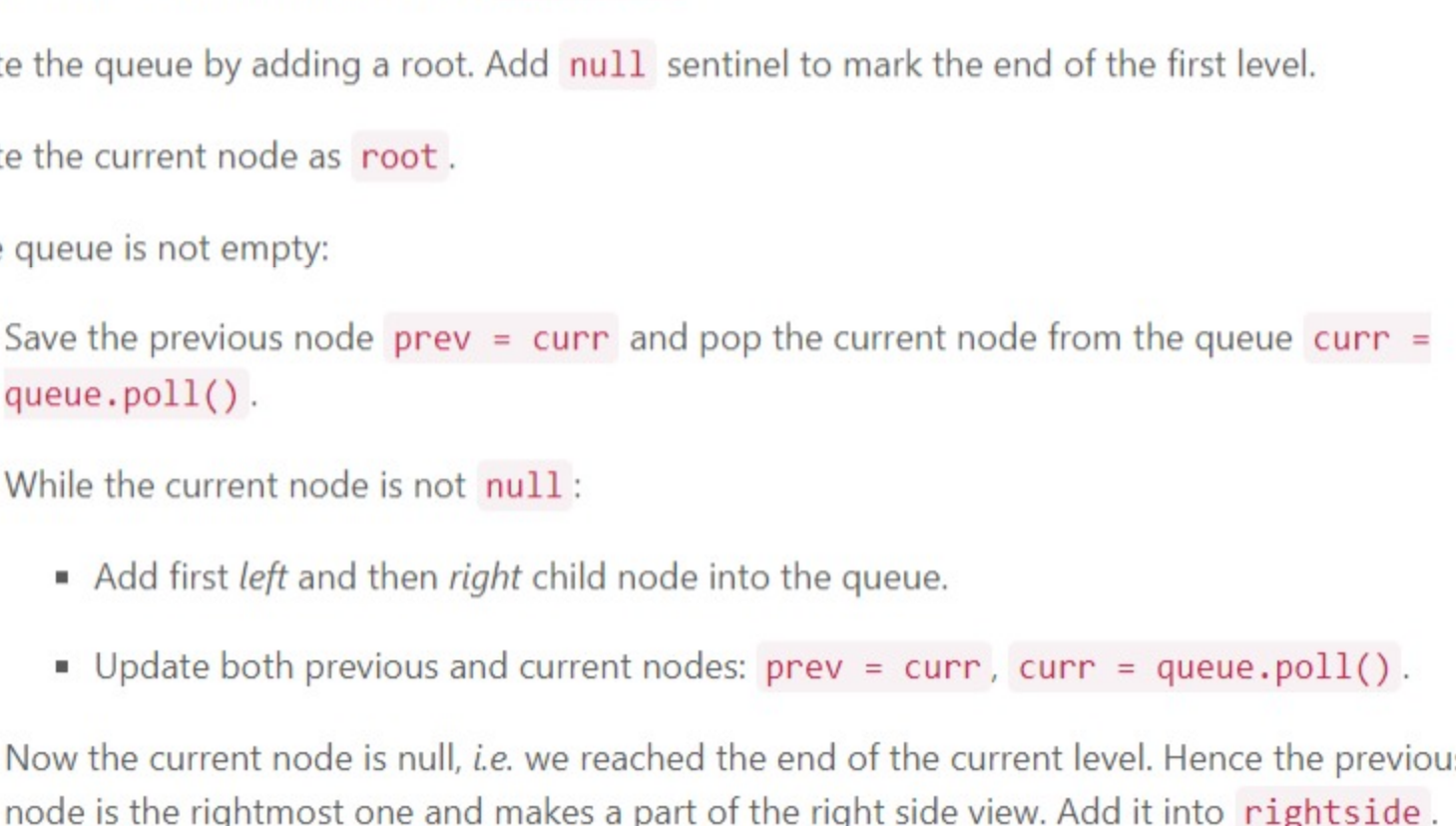
```python
class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if root is None:
            return []

        next_level = deque([root,])
        rightside = []

        while next_level:
            # prepare for the next level
            curr_level = next_level
            next_level = deque()

            while curr_level:
                node = curr_level.popleft()

                # add child nodes of the current level
                # in the queue for the next level
                if node.left:
                    next_level.append(node.left)
                if node.right:
                    next_level.append(node.right)

            # The current level is finished.
            # Its last element is the rightmost one.
            rightside.append(node.val)

```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$ since one has to visit each node.
- Space complexity: $\mathcal{O}(D)$ to keep the queues, where D is a tree diameter. Let's use the last level to estimate the queue size. This level could contain up to $N/2$ tree nodes in the case of complete binary tree.

Approach 2: BFS: One Queue + Sentinel

Another approach is to push all the nodes in one queue and to use a sentinel node to separate the levels. Typically, one could use `null` as a sentinel.



The first step is to initiate the first level: `root` + `null` as a sentinel. Once it's done, continue to pop the nodes one by one from the left and push their children to the right. Stop each time the current node is `null` because it means we hit the end of the current level. Each stop is a time to update a right side view list and to push `null` in the queue to mark the end of the next level.

Algorithm

- Initiate the list of the right side view `rightside`.
- Initiate the queue by adding a root. Add `null` sentinel to mark the end of the first level.
- Initiate the current node as `root`.
- While queue is not empty:
  - Save the previous node `prev = curr` and pop the current node from the queue `curr = queue.poll()`.
  - While the current node is not `null`:
    - Add first *left* and then *right* child node into the queue.
    - Update both previous and current nodes: `prev = curr`, `curr = queue.poll()`.
  - Now the current node is null, *i.e.* we reached the end of the current level. Hence the previous node is the rightmost one and makes a part of the right side view. Add it into `rightside`.
  - If the queue is not empty, push the null node as a sentinel, to mark the end of the next level.
- Return `rightside`.

Implementation

Note, that `ArrayDeque` in Java doesn't support null elements, and hence the data structure to use here is `LinkedList`.

Java | Python3 | Copy

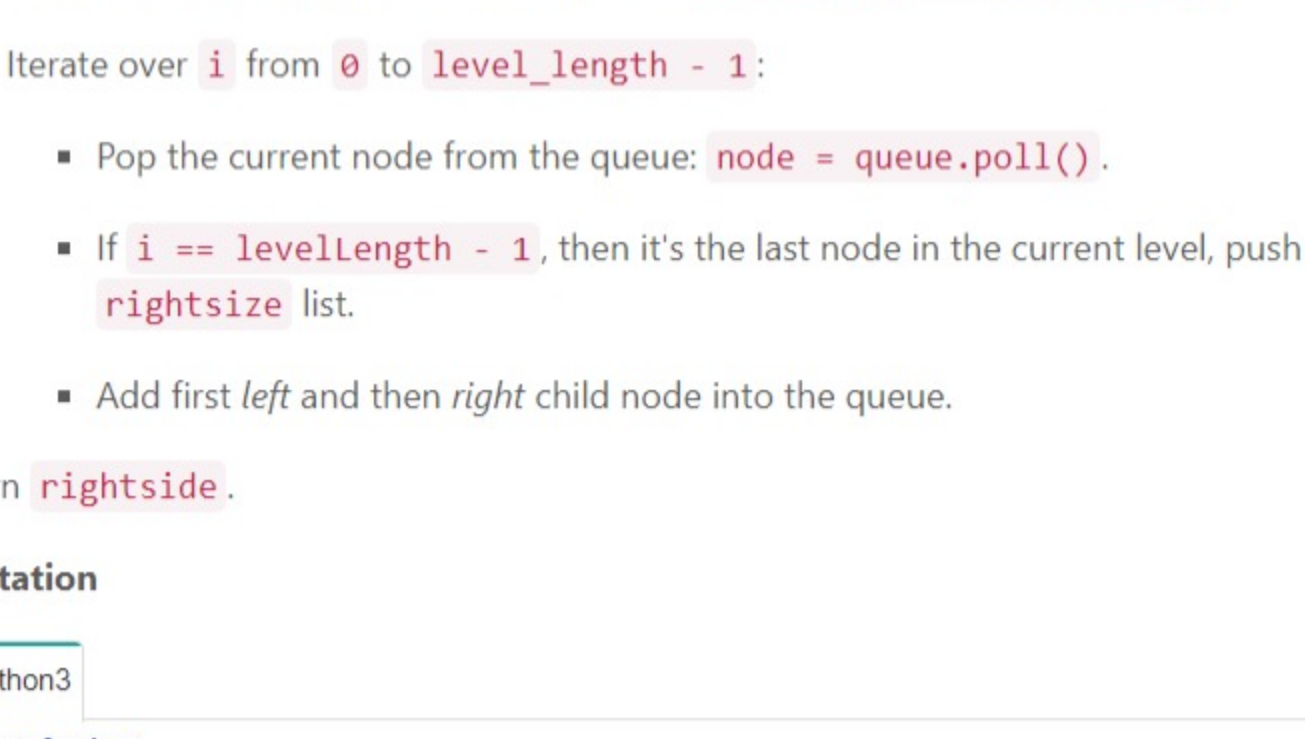
```python
class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if root is None:
            return []

        queue = deque([root, None,])
        rightside = []

        curr = root
        while queue:
            prev, curr = curr, queue.popleft()

            while curr:
                # add child nodes in the queue
                if curr.left:
                    queue.append(curr.left)
                if curr.right:
                    queue.append(curr.right)

                prev, curr = curr, queue.popleft()

            # the current level is finished
            # and prev is its rightmost element
            rightside.append(prev.val)
            # add a sentinel to mark the end
            # of the next level
            if queue:
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$ since one has to visit each node.
- Space complexity: $\mathcal{O}(D)$ to keep the queues, where D is a tree diameter. Let's use the last level to estimate the queue size. This level could contain up to $N/2$ tree nodes in the case of complete binary tree.

Approach 3: BFS: One Queue + Level Size Measurements

Instead of using the sentinel, we could write down the length of the current level.



Algorithm

- Initiate the list of the right side view `rightside`.
- Initiate the queue by adding a root.
- While the queue is not empty:
  - Write down the length of the current level: `levelLength = queue.size()`.
  - Iterate over `i` from `0` to `level_length - 1`:
    - Pop the current node from the queue: `node = queue.poll()`.
    - If `i == levelLength - 1`, then it's the last node in the current level, push it to `rightsize` list.
    - Add first *left* and then *right* child node into the queue.
- Return `rightside`.

Implementation

Java | Python3 | Copy

```python
class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if root is None:
            return []

        queue = deque([root,])
        rightside = []

        while queue:
            level_length = len(queue)

            for i in range(level_length):
                node = queue.popleft()
                # if it's the rightmost element
                if i == level_length - 1:
                    rightside.append(node.val)

                # add child nodes in the queue
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

        return rightside
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$ since one has to visit each node.
- Space complexity: $\mathcal{O}(D)$ to keep the queues, where D is a tree diameter. Let's use the last level to estimate the queue size. This level could contain up to $N/2$ tree nodes in the case of complete binary tree.

Approach 4: Recursive DFS

Everyone likes recursive DFS, so let's add it here as well. The idea is simple: to traverse the tree level by level, starting each time from the rightmost child.

Implementation

Java | Python3 | Copy

```python
class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if root is None:
            return []

        rightside = []

        def helper(node: TreeNode, level: int) -> None:
            if level == len(rightside):
                rightside.append(node.val)
            for child in [node.right, node.left]:
                if child:
                    helper(child, level + 1)

        helper(root, 0)
        return rightside
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$ since one has to visit each node.
- Space complexity: $\mathcal{O}(H)$ to keep the recursion stack, where H is a tree height. The worst-case situation is a skewed tree, when $H = N$.

Rate this article:

Previous Next

Comments: 7 Sort By

Type comment here... (Markdown is supported)

Preview Post

Parisaa ★1 June 8, 2020 11:32 AM

Definition for a binary tree node.

class TreeNode:

Read More

1 Share Reply

SHOW 1 REPLY

andvary STAFF ★767 May 31, 2020 3:16 PM

Hello @hughdbrown , to use Queue structure is an overkill here. It's designed for a safe exchange between multiple threads and hence requires locking which leads to a performance downgrade.

1 Share Reply

SHOW 1 REPLY

hughdbrown ★6 May 31, 2020 10:41 AM

I think you are making it way harder than you have to.

```
from queue import Queue
# Definition for a binary tree node.
# class TreeNode:
```

Read More

1 Share Reply

SHOW 2 REPLIES

shri99 ★11 June 29, 2020 1:25 AM

Simple level order traversal

```
class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> ans = new LinkedList<>();
```

Read More

0 Share Reply

varun_jain ★0 42 minutes ago

I got stack overflow with DFS. The only difference with the sol is the terminating condition.
So an extra call for each null child is too much I guess.

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
```

Read More

0 Share Reply

sm1090903 ★-1 June 29, 2020 10:06 AM

I did a pre-order traversal using a visited array.

```
class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        res = []
```

Read More

0 Share Reply

shuangpan ★21 June 21, 2020 4:36 AM

DFS left to right

```
class Solution {
    private List<Integer> list = new ArrayList<>();
    public List<Integer> rightSideView(TreeNode root) {
```

Read More

0 Share Reply