Discussion 6

Roadmap

- FAQ on hw3

FAQ on hw3

Note1: Pesudocode just provide you main idea of my solution. If you want to follow it, you need to do some adjustment and consider the edge cases on your own

Note2: Don't be limited by the pesudocode. We are glad to see various solutions.

Double Pointer

```
// You can think this is an array of pointers which point to TreeNode
TreeNode **children

// Demo
// Goal: Create an array whose length is 10 and each element is an int pointer
int length = 10;
int **array = new int*[length];
// Code below just to do some operations on one element in the array
int a = 1;
array[0] = &a;
std::cout << *array[0] << std::endl; // Result is 1
```

Tree Insertion

```
What is max_width in the tree?
It's the maximum number of children a TreeNode can hold
Example for max_width:
max_width: 3                            max_width: 4
           1                                  1
       2   3   4                        2   3   4   5
Example for Tree insertion:
max_width is 2
Insert 1 -> 2 -> 3 -> 4 -> 5
    1    |      1    |     1     |      1      |      1
         |     /     |    / \    |     / \     |     / \
         ->   2      ->  2   3   ->   2   3    ->   2   3
         |           |           |   /         |   / \
         |           |           |  4          |  4   5
```

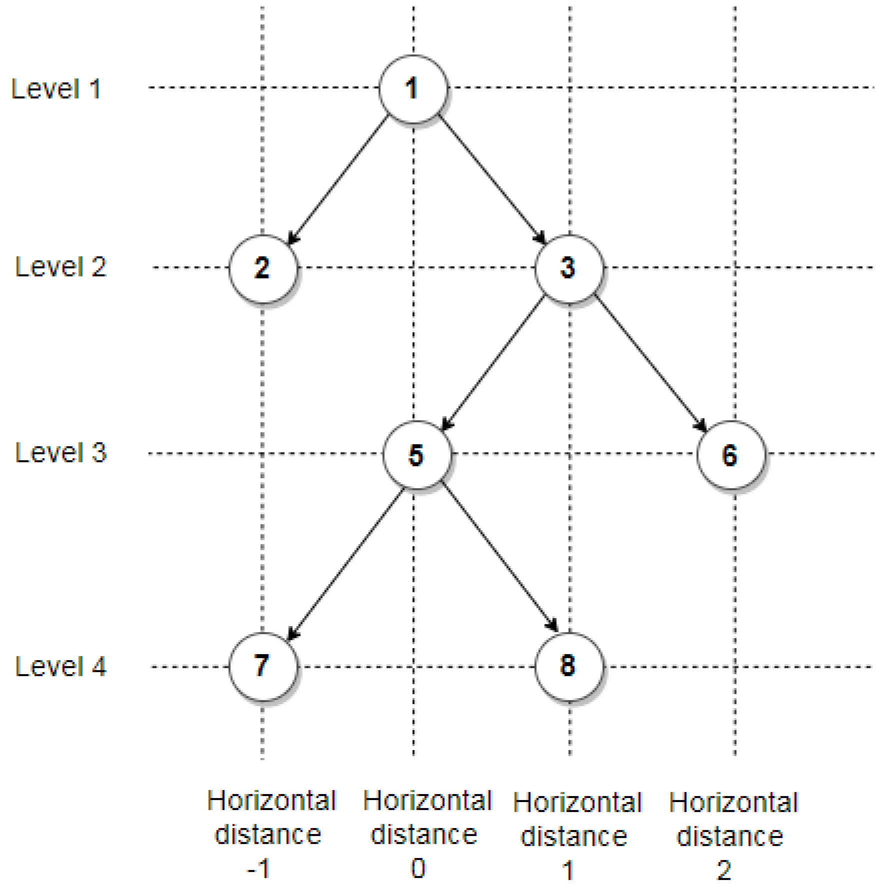
```
# One way for tree insertion:
# Use Levelorder traversal -> You can find more info from last discussion
def insert_node(root, queue, inode):
"""Pesudocode to insert a node in a tree

Args:
  root: root of the tree
  queue: a queue to store the node
  inode: the node you want to insert
"""
    queue.enqueue(root)
    while queue.head != None:
        node = queue.dequeue()
        # If the node is not full, insert the node
        # Full means you can not insert child on that node
        if not check_full(node):
          Insert the inode as a child of node
          return
        for child in node.children:
            queue.enqueue(child)

When you want to insert 6 in previous tree:
Queue: 1 -> 2 3 -> 3 4 5 -> Find 3 is not full -> Insert 6 (left child of 3)
```

Right & left view of the tree

```
Another variant of levelorder:
           1           ->       1
          / \
         2   3         ->       2  3
        / \ / \
       4  5 6  7       ->       4  5  6  7
      / \
     8   9             ->       8  9

The rigth view is 1 3 7 9, and the left view is 1 2 4 8

Step1: Get elements in each level (Using queue)
Step2: Get the leftmost/rightmost element in each level

Why left and right view of the tree are the same in input/1.txt?
The tree degenerates to a linked list:
                                1
                                2
     1 2 3 4 5       ->         3          <-    1 2 3 4 5
                                4
                                5
```

Bottom & Top view of the tree

- A binary tree as an example:



The bottom view above → 7, 5, 8, 6

The top view above → 2, 1, 3, 6

<u>Horizontal distance:</u>

- Horizontal distance of the root = 0
- Horizontal distance of a left child = horizontal distance of its parent - 1
- Horizontal distance of a right child = horizontal distance of its parent + 1

<u>Algorithms: (One method to solve this kind of problem)</u>

```python
def printBottom(root, dist, level, dict):
  """Pesudocode of bottom view

  Print what you can see from the bottom of the tree.
  BTW, you need to print all of the overlapped nodes.

  Args:
    root: root of the tree/subtree
    dist: horizontal distance of the root
    level: level of the root
    dict:
      key: horizontal distance
      value: (nodes, level)
  """
  # For the base cas
  if root is None:
    return

  # Main operations
  # 1. dist is not in dict
  if dist not in dict:
    Add (root, level) in dict[dist]
  # 2. dist is in dict and present node is at the higher level
  elif level > dict[dist][1]:
    Replace dict[dist] with (root, level)
  # 3. dist is in dict and present node is at the same level
  elif level == dict[dist][1]:
    Add node in dict[dist]

  # Recurrsion
  printBottom(root.left, dist - 1, level + 1, dict)
  printBottom(root.right, dist + 1, level + 1, dict)

# Actually it's a pre-order traversal for a tree
# For the top view, it's quite similar
```

```
Simple example:
                 1
                / \
               2   3
              / \ /
             4   5 6
Steps   |      node     |      dist    |      dict (Ignore level here for clarity)
  0     |       1       |       0      |     {0:1}
  1     |       2       |      -1      |     {0:1, -1:2}
  2     |       4       |      -2      |     {0:1, -1:2, -2:4}
  3     |       5       |       0      |     {0:5, -1:2, -2:4}
  4     |       3       |       1      |     {0:5, -1:2, -2:4, 1:3}
  5     |       6       |       0      |     {0:(5,6), -1:2, -2:4, 1:3}
The bottom view is 4 2 5 6 3
```

```
Time complexity: O(n)
Space complexity: O(n)
```