Data Structures and Algorithms

Last Lecture

Topics : Review of Trees, Graphs. Linked Lists, DP

Note

This is just an overview of the contents covered in the class. If you haven't attended the lectures, I highly doubt it that you'd be able to follow this content as we'll just see the pseudocode of those algorithms. Every code covered in this lecture is a standard problem and you can find the problem statement with a single google search. Remember to try them out before looking at the solution.

Level Order Traversal

Algorithm Prints the elements in the level order fashion

Ensure: The Tree is not empty

```
1: function LEVEL_ORDER_TRAVERSAL(root)
   ▷ Queue stores the pointers of the nodes.

2:     queue.push(root)
3:     while Queue is not empty, do
4:         current_node ← queue.front
5:         queue.pop
6:         Print(current_node.data)
7:         if Left Child Exists then
8:             queue.push(current_node.left)
9:         if Right Child Exists then
10:            queue.push(current_node.right)

11:    Print(Level Order Traversal finished)
```

Print the Level Line By Line I

Followup : Can you print it line by line ? Each level should be printed in one single line
Answer : Yes, it can be done using 2 queues. We maintain 2 queues called *main_queue* and *aux_queue*. The second queue is used to hold all the elements of the next level temporarily while the current level is being processed. As soon as the current level is finished, we transfer the waiting elements back to this queue.

Algorithm Prints the elements in the level order fashion **Line By Line**

Require: A non empty tree
Ensure: Each level is printed in a different line

```
1:  function LEVEL_ORDER_TRAVERSAL(root)
    ▷ Queue stores the pointers of the nodes.

2:      main_queue.push(root)
3:      while Main Queue is not empty, do
4:          Print(Here comes a new level)
5:          while Main Queue is not empty, do
6:              current_node ← main_queue.front
7:              queue.pop
8:              Print(current_node.data)
9:              if Left Child Exists then
10:                 aux_queue.push(current_node.left)
11:             if Right Child Exists then
12:                 aux_queue.push(current_node.right)
13:         Print(This level has been printed)
14:         Swap(main_queue, aux_queue)

15:     Print(Level Order Traversal finished)
```

Followup : Is the time complextiy $O(n^2)$ due to nested *while* loops or is it still $O(n)$

Print the Level Line By Line II

Followup : Ok, looks good. But here's a challenge. Can you do it using a single queue?
Answer : Yes, it can be done with a single queue. The idea is to use the concept of a **marker**. It is something that you insert into the queue which creates a partition among the elements. Let us use *nullptr* as our marker. Whenever we are processing the first element of any level, we need to ensure that the marker is set at the back. This way, the marker would create a partition between these 2 levels. Now, whenever we reach a marker, it means the previous level has been finished and there are some elements of the next level waiting behind the marker. Hence, we extract the marker and place it at the back.

Algorithm Prints the elements in the level order fashion **Line By Line**

Require: A non empty tree
Ensure: Each level is printed in a different line and one queue is used

```
1:  function LEVEL_ORDER_TRAVERSAL(root)
    ▷ Queue stores the pointers of the nodes.
2:      marker ← nullptr
3:      queue.push(root)
4:      queue.push(marker)
5:      while Queue is not empty, do
6:          if queue.front == marker then
7:              Print(One Level has been finished)
8:              queue.pop
9:              if queue is not empty then
10:                 queue.push(marker)
11:         current_node ← queue.front
12:         queue.pop
13:         Print(current_node.data)
14:         if Left Child Exists then
15:             queue.push(current_node.left)
16:         if Right Child Exists then
17:             queue.push(current_node.right)
18:     Print(Level Order Traversal finished)
```

Print the Level Line By Line III

Followup : Ok, Let's see what else you can do ? Can you print the level line by line using a single queue and no marker?
Answer : Yes, it can also be done using a single queue. Notice that when we meet the first element of any level, all the other elements of that level would already be in the queue (and those will be the only elements). Suppose the size of the queue before taking out the first element of the level is *size*. It means that the next *size* elements belong to one level. Hence, we could run a loop with *size* iterations and print the incomming elements in a single line. In the meanwhile, we keep pushing elements at the back of the queue. As soon as the inner loop terminates, the next level would automatically start.

Algorithm Prints the elements in the level order fashion **Line By Line**

Require: A non empty tree
Ensure: Each level is printed in a different line and one queue is used (with no *marker*)

```
1:  function LEVEL_ORDER_TRAVERSAL(root)
    ▷ Queue stores the pointers of the nodes.
2:      queue.push(root)
3:      while Queue is not empty, do
4:          Print(Here comes a new level)
5:          size ← queue.size
6:          for i = 1 : size do
7:              current_node ← queue.front
8:              queue.pop
9:              Print(current_node.data)
10:             if Left Child Exists then
11:                 queue.push(current_node.left)
12:             if Right Child Exists then
13:                 queue.push(current_node.right)
14:         Print(This level has been printed)
15:     Print(Level Order Traversal finished)
```

Vertical Order Traversal

Algorithm Print the Vertical Order Traversal of a Tree

Require: A non empty tree
Ensure: In case of equal x and y, the node which comes first in the level order traversal would be printed first

```
1: function VERTICAL_ORDER_TRAVERSAL(root)
   ▷ Horiz_Map is a map with key as the horizontal distance. (can be negative)
   ▷ The value is the vector of elements which are at the same horizontal level.
   ▷ The elements in the vector are ordered according to their appearance in Level Order Traversal

2:     horiz_dist ← 0
3:     queue.push(root, horiz_dist)
4:     while Queue is not empty, do
5:         (current_node, horiz_dist) ← queue.front
6:         queue.pop
7:         horiz_map[horiz_dist].push_back(current_node.data)
8:         if Left Child Exists then
9:             queue.push(current_node.left, horiz_dist − 1)
10:        if Right Child Exists then
11:            queue.push(current_node.right, horiz_dist + 1)

12:    for Sorted Keys in horiz_map do
13:        for all ordered elements in horiz_map[key] do
14:            Print(element)
```

Followup : As you can see, the time complexity is $O(n \log n)$. Can you do it in $O(n)$?
Answer : Yes. Use an *unordered_map* instead of an *ordered_map*. Now, how do we traverse the unordered map in a sorted order? While performing Level (and Vertical) Order Traversal, simply keep track of the minimum and maximum horizontal distance seen so far. Say, they are *horiz_min* and *horiz_max*. Start traversing the unordered map from *horiz_min* to *horiz_max* in increments of one. Can you see why every increment would be present in the Hash Map?

Followup : How do you print the top view of a binary tree?
Answer : For each vertical line, only the first element of that line is visible. Hence, just perform vertical order and instead of printing every element, just print the first element of that vertical line.

Followup : Can you print the top view without using Vectors?
Answer : Yes. While performing vertical order traversal, if any element is already hashed at a particular *horiz_dist*, then don't hash the incoming element. Of course, you should continue inserting its children into the queue if they exist. We don't lose out on any information in this way beacause the first element of each line is hashed first (due to level order traversal).

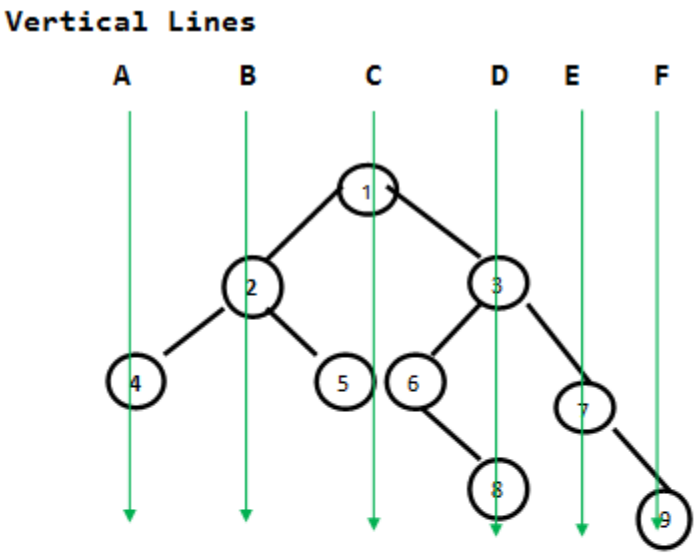

Image Credits : GeeksForGeeks

Zig Zag Tree Traversal

Here's an interesting crossover between various data structures. You need to print the first level from left to right, the next from right to left, and so on. Naturally, this gives us a hint to use stacks.

Algorithm Print the **Zig Zag** Traversal of a Binary Tree

Ensure: The tree is **not** empty

```
1:  function ZIG_ZAG_TRAVERSAL(root)
    ▷ current_stack contains all the elements of the current level.
    ▷ next_stack contains all the elements of the next level.
    ▷ left_to_right is true if the current level associates from left to right

2:      current_stack.push(root)
3:      left_to_right ← True
4:      while current_stack is not empty, do
5:          while current_stack is not empty, do
6:              node ← current_stack.top
7:              current_stack.pop
8:              Print(node.data)
9:              if left_to_right then
10:                 if Left Child Exists then
11:                     next_stack.push(node.left)
12:                 if Right Child Exists then
13:                     next_stack.push(node.right)
14:             else
15:                 if Right Child Exists then
16:                     next_stack.push(node.right)
17:                 if Left Child Exists then
18:                     next_stack.push(node.left)
19:         Print(One level has been printed)
20:         swap(current_stack, next_stack)                ▷ Go to the next level
21:         left_to_right ←!left_to_right                  ▷ Change the associativity of the level
```

Connect Nodes at the Same Level

Want to see the *marker* technique in practice? Read on!

Algorithm Populate the *next_right* pointers by connecting all the nodes at the same level

```
1: function CONNECT_NODES_AT_SAME_LEVEL(root)
   ▷ Queue stores the pointers of the nodes.

2:     if Tree is empty then
3:         return

4:     marker ← nullptr
5:     queue.push(root)
6:     queue.push(marker)
7:     while Queue is not empty do
8:         if queue.front == marker then
9:             queue.pop
10:            if queue is not empty then
11:                queue.push(marker)
12:        current_node ← queue.front
13:        queue.pop
14:        current_node.next_right ← queue.front      ▷ The queue can also contain the marker
15:        if Left Child Exists then
16:            queue.push(current_node.left)
17:        if Right Child Exists then
18:            queue.push(current_node.right)
```

Clone a Linked List with Random Pointers

Another question to demonstrate the power of data structures

Algorithm Clone a linked list with *next* and *random* pointers

```
1:  function CLONE_LINKED_LIST(old_head)
      ▷ node_just_below is a hash map where the keys are the address of the original list
      ▷ The values is the address of the corresponding node in the new list

2:      if List is empty then
3:          return

4:      backup ← old_head

5:      new_head ← Deep_Copy(old_head)
6:      node_just_below[old_head] ← new_head
7:      while old_head.next exist, do                          ▷ Make a normal copy
8:          new_head.next ← Deep_Copy(old_head.next)
9:          old_head ← old_head.next
10:         new_head ← new_head.next
11:         node_just_below[old_head] = new_head               ▷ Vertically link the nodes

12:     node_just_below[null] ← null
13:     for all key in HashMap do                              ▷ Correct all the random pointers
14:         if key is not null then
15:             node_just_below[key].random = node_just_below[key.random]
16:     return node_just_below[backup]
```

Trapping Rain Water

This is a good question to apply your **DP** skills.

Algorithm Find the maximum amount of water that can be trapped between the given bars

Ensure: One Based Indexing for the array

1: **function** $Trapping\_Rain\_Water(arr)$
▷ $left\_max[i]$ represents the largest bar to the left of i (including it)
▷ $right\_max[i]$ represents the largest bar to the right of i (including it)
▷ $contribution[i]$ denotes the maximum amount of water on top of the $i-th$ bar

2: $n \leftarrow arr.length$

3: $left\_max[1] \leftarrow a[1]$
4: **for** $i = 2 : n$ **do**
5: $left\_max[i] = max(left\_max[i-1], a[i])$

6: $right\_max[n] \leftarrow a[n]$
7: **for** $i = len - 1$ **downto** 1 **do**
8: $right\_max[i] = max(right\_max[i+1], a[i])$

9: **for** $i = 1 : n$ **do**
10: $max\_height \leftarrow min(left\_max[i], right\_max[i])$
11: $contribution[i] \leftarrow (max\_height - a[i])$

12: $total\_water \leftarrow 0$
13: **for** $i = 1 : n$ **do**
14: $total\_water \leftarrow total\_water + contribution[i]$

15: **return** $total\_water$

Remove Duplicates From List

Algorithm Remove all duplicates from a sorted list

Ensure: You should retain one copy of the duplicate elements

```
1: function REMOVE_DUPLICATES(head)
2:     if List is of length 0 or 1 then
3:         return head

   ▷ Recursively remove the duplicates to the right and connect the two lists

4:     clean_sub_list ← Remove_Duplicates(head.next)
5:     head.next ← clean_sub_list

6:     if head.data == head.next.data then
7:         return clean_sub_list
8:     else
9:         return head
```

Remove Duplicates From List II

Followup : What if you don't need to retain any copy of the duplicate elements? Will Recursion still work?
Answer : Yes, it would work. But, you need to be very careful with the function definition. Even if you remove duplicate in the sub list of $n-1$ elements, you wouldn't be able to fulfill the responsibility part, that is, you wouldn't be able to remove copies of the first element effectively. For example, if the list is $(1, 1, 1, 2)$ and you use the *sub_list* technique on $(1, 1, 2)$, you'll get the result as (2). But notice that you wouldn't be able to judge whether you should keep the first 1 or leave it (as its duplicate copy has vanished). Hence, we need to smartly decide which task to outsource to recursion. Let us do a bit more work from our side. We'll handle all the copies of the first element by ourselves and defer the remaining work

Algorithm Remove all duplicates from a sorted list

Ensure: You should delete **All** copies of the duplicate elements

```
1:  function REMOVE_DUPLICATES(head)
2:      if List is of length 0 or 1 then
3:          return head

    ▷ If the first element is repeated, go to its last copy
4:      if head.data == head.next.data then
5:          while head.next exists and head.data == head.next.data do
6:              head ← head.next
7:          return Remove_Duplicates(head.next)          ▷ Head points to the last copy
8:      else
9:          clean_sub_list ← Remove_Duplicates(head.next)
10:         head.next ← clean_sub_list          ▷ First element is not repeated, hence no effect
11:         return head
```

Word Ladder

Here's a classic application of BFS. Please attempt this question before proceeding to the solution. You can submit it on **Leetcode**.

Algorithm Find the number of nodes in the shortest path to convert *begin_word* to *end_word* using only valid conversions

---

Require: The length of both the words should be same
Ensure: You can only change one character at a time

```
1: function WORD_LADDER(begin_word, end_word, word_list)
   ▷ dictionary is a hashmap containing all the valid words in the dictionary
   ▷ visited is a hashmap containing all the strings which are already processed in the queue

2:     for each word in word_list do
3:         dictionary.insert(word)
4:     queue.push(begin_word, 1)
5:     visited.insert(begin_word)
6:     while queue is not empty do                                  ▷ Perform BFS
7:         (current_string, level) ← queue.front
8:         queue.pop
9:         if current_string == end_word then
10:            return level
11:        for Each element in current_string do
12:            for Each char in the alphabet do
13:                backup ← element
14:                Replace element by char                          ▷ Create the new string
15:                if current_string is in dictionary then
16:                    if current_string is not in visited then
17:                        queue.push(current_string, level + 1)
18:                        visited.push(current_string)
19:                Replace char by backup                           ▷ Get the original string back
20:    return Not Possible
```

Epilogue

So, we are finally at the end of this course. Time sure passes by quickly. However, as you already know, no matter what time it is, it's never too late to start over. A big round of applause for the people who stayed till the end and made an effort to go through the problem sets and programming assignments.

The link for the programming assignments would remain active for a couple of more days. Please go through them if you haven't already (especially the last programming assignment). The lecture notes would remain available on *Github* for the time being. You can star this repository to revisit the notes later. I'm sorry I couldn't prepare proper lecture notes for the topics covered in the class.

The goal of this course was not to make you memorize things and spit it out during tests/interviews. The real motive behind it was to re-ignite the spark of programming inside you, to show you what you can do with the tools at your disposal. Programming is an acquired taste for a lot of people and it might as well be for you. All you gotta do is take that first step. I would consider my efforts to be successful if even one of you takes up programming passionately. Mind you, I don't want to keep a flawed metric such as getting placed in a good company. Although I do wish you all the best for that!