DSAD: Assignment 1 (PS1: Binary Tree)

Group Number: 323 and members are:

|  |  |  |
| --- | --- | --- |
| Name | ID No. | Email ID |
| RAGESH HAJELA | 2021SC04877 | 2021SC04877@wilp.bits-pilani.ac.in |
| THAKARE KEDAR SATISHRAO | 2021SC04878 | 2021SC04878@wilp.bits-pilani.ac.in |
| D V S RAMA KRISHNA | 2021SC04879 | 2021SC04879@wilp.bits-pilani.ac.in |

Introduction

This document covers:

1. Solution overview.
2. Solution Algorithm design.
3. Time complexity analysis in terms of input size (n nodes).
4. Worst case run time analysis of methods.

Solution Overview

In terms of algorithm, the problem statement is:

1. Given a sequence of BFS-based tree representation of nodes,
2. Read all the boundary nodes of the binary tree,
3. In anti-clockwise direction starting from the root.
4. The boundary traversal includes left boundary, leaves, and right boundary in order

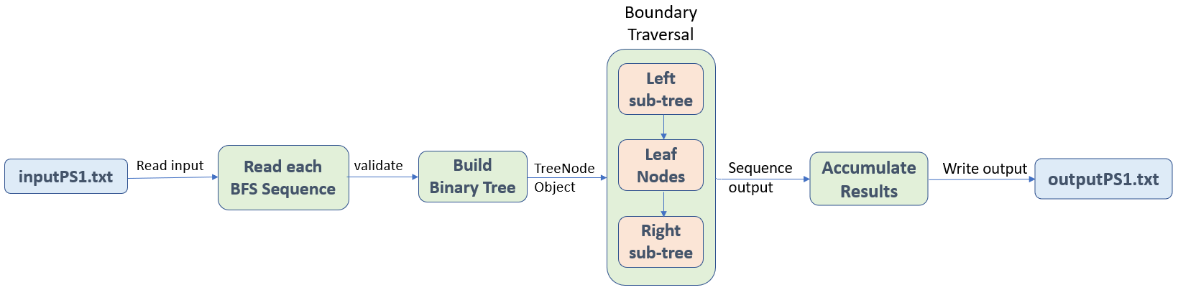
While building the solution, assumptions are:

1. Breadth-first search algorithm is the same where we explore each level, left to right
2. If a child for some node is not present, it is marked as None
3. But there will always be only one valid tree/solution.

The overall solution algorithm design has these stages:

1. Read text file and preprocess input sequence of nodes
2. Create binary tree using BFS algorithm from the sequence
3. Perform boundary traversal of the binary tree
4. Post process results and write in an output file.

**Workflow diagram is here:**

****

Solution Algorithm Design

**Tree Node Class Data Structure**

This is the class definition already provided to use as data structure for building the binary tree.

# Tree Node Data structure  
class TreeNode:  
 def \_\_init\_\_(self, data, left=None, right=None):  
 self.data = data  
 self.left = left  
 self.right = right

**Read Input and Preprocessing**

1. Read all lines from input file inputPS1.txt and create a list structure to hold the inputs
2. If there are no lines in the input file, print a message and exit
3. If data is valid, preprocess input and convert to either int or NoneType python structures.
4. Run execution for creating binary tree and perform boundary traversal.

**Create Binary Tree**

Use method “create\_binary\_tree” to traverse the BFS sequence of nodes and use TreeNode data structure to create binary tree

def create\_binary\_tree (tree\_nodes\_list: List) -> TreeNode:  
 *"""  
 A function to create a binary tree* ***:param*** *tree\_nodes\_list: input nodes as either int or NoneType* ***:return****: created binary tree node data structure  
 """*

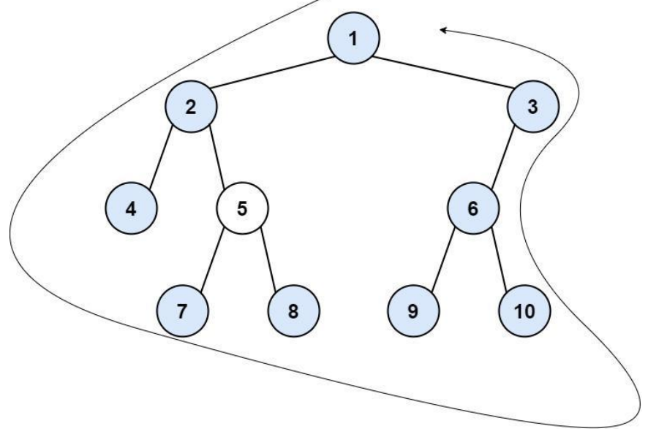
1. Breadth-first search will be used to construct a binary tree.
2. Read the first node as the root and build TreeNode class object.
3. Insert each element as TreeNode objects, starting from the root and add it to a queue.
4. At each iteration, once finish adding both left and right children of nodes, we pull the next node from the queue.
5. On running out of elements to add, traversal concludes and return the root TreeNode object which contains the entire binary tree.

**Boundary Traversal**

Once the complete BFS-based nodes sequence is transformed into the binary tree object, we begin the tree boundary traversal using method “traverse\_boundary”

def traverse\_boundary(root: TreeNode) -> None:  
 *"""  
 A function to perform boundary traversal of a given binary tree* ***:param*** *root: TreeNode structure representing all tree nodes* ***:return****: only appends output, nothing to return  
 """*

Here is an example of boundary traversal of a sample tree. Input Sequence is = 1 2 3 4 5 6 None None None 7 8 9 10



**Solution approach** is the following:

1. The left boundary is defined as the path from the root to the left-most node.
2. The right boundary is defined as the path from the root to the right-most node.
3. If the root doesn’t have left subtree or right subtree, then the root itself is left boundary or right boundary.
4. The left-most node is defined as a leaf node we could reach when always firstly travel to the left subtree if it exists.
5. If not, travel to the right subtree. Repeat until we reach a leaf node.
6. The right-most node is also defined in the same way with left and right exchanged.

For example, **boundary traversal** for above example tree is: 1 2 4 7 8 9 10 6 3.

Based on this, our boundary traversal **algorithm** is split into these three steps:

1. Read the left boundary in top-down manner.
2. Read all leaf nodes from left to right, which can again be sub-divided into two sub-parts:
   1. Read all leaf nodes of left sub-tree from left to right.
   2. Read all leaf nodes of right subtree from left to right.
3. Read the right boundary in bottom-up manner.

Run Time Analysis

**Algorithm Pseudo Code**

Boundary\_Traversal(head):

root —> next\_node(head)

q —> empty list

q.append(root)

while (next\_node(head))

node —> q.pop()

node.left = next\_node(head)

q.append(node.left)

node.right = next\_node(head)

q.append(node.right)

**How Algorithm Works**

For the same example above, here is how we traverse boundary nodes.

**Step 1**: Read the root node

|  |
| --- |
| 1 |

**Step 2**: Read the left sub-tree boundary nodes.

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 4 |

**Step 3**: Read the left sub-tree leaf nodes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 7 | 8 |

**Step 4**: Read the right sub-tree leaf nodes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 7 | 8 | 9 | 10 |

**Step 5**: Read the right sub-tree boundary nodes

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 7 | 8 | 9 | 10 | 6 | 3 |

* Since we want the leaves to come in a specific order which is bottom-left to top-right,
* and a level order traversal will not work because it will print the upper-level leaves first.
* Therefore, we **use a preorder traversal**

**Finally**, the boundary traversal output is the following

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 7 | 8 | 9 | 10 | 6 | 3 |

**Time Complexity Analysis**

In terms of processing each input sequence, these are the two functions primarily driving the time complexity:

|  |  |  |
| --- | --- | --- |
| Method | Task | Time Complexity |
| main() | Loop once for each input | O(n) |
| analyze\_tree() | Loop once for each input sequence, build tree and perform boundary traversal analysis | O(n) |
| create\_binary\_tree() | Loop once for each node element of the input sequence | O(n) |
| traverse\_boundary() | Loop once for each node element of the input sequence to traverse the tree boundary | O(n) |

Let’s say, Binary tree has n-number of nodes, and here are the major operations:

1. For left or right sub-tree traversal, in every iteration, we go down a level in the tree, and there are log2n number of levels in a binary tree
2. For leaf nodes, we recursively visit every node in the binary tree, and there are N numbers of nodes.
3. So, time complexity should be O(log2n) + O(n) = **O(n)**

**Conclusions on time complexity:**

1. The time complexity of the above solution is O(n), where n is the total number of nodes per input sequence.
2. Time complexity is linear because we have traversed through all the nodes of the tree on left and right subtree along with leaf nodes.

**Space Complexity Analysis**

Let’s say, Binary tree has n-number of nodes, and here are the major operations:

1. We traverse the binary tree and list of nodes found for all three different operations i.e., left, and right subtree along with leaf nodes.
2. And there can only exist log2N numbers of lefts and rights, but there can exist [1, n/2] number of leaf nodes
3. So, total space complexity results to **O(n)**

**Conclusions on space complexity:**

1. The space complexity of the above solution is O(n), where n is the total number of nodes per input sequence.
2. Space is needed for the two recursion stacks while traversing boundary nodes. In the worst case (skewed tree), space complexity will be linear.
3. This is because the compiler stack uses space. And the functions used to print the boundary elements use recursion which counts for compiler stack.

Alternative Solutions

1. As other alternatives to solve the problem, we can use either iterative approach or use combinations of pre, post and in order traversal algorithms to traverse through the boundary nodes of a binary tree with the **time complexity of O(n)**
2. We will have to take special care of the corner case that same nodes are not traversed again. If any node is a part of the left boundary as well as leaf nodes, or any node is a part of the right boundary as well as leaf nodes. So, we will have to traverse only till the second last node of both the boundaries in that case. Also, we should not traverse the root again.