IDAA432C Assignment - 5

By:

- Aditya Vallabh
- Kiran Velumuri
- Neil Syiemlieh
- Tauhid Alam

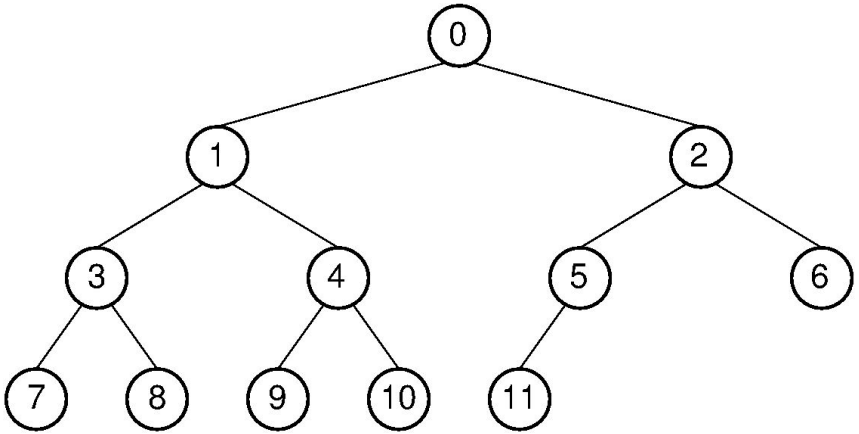
Question

Write an efficient algorithm which, given an inorder traversal, can construct a corresponding complete binary tree. Do the necessary experimentation and analysis with your algorithm.

Algorithm Design

Complete Binary Tree

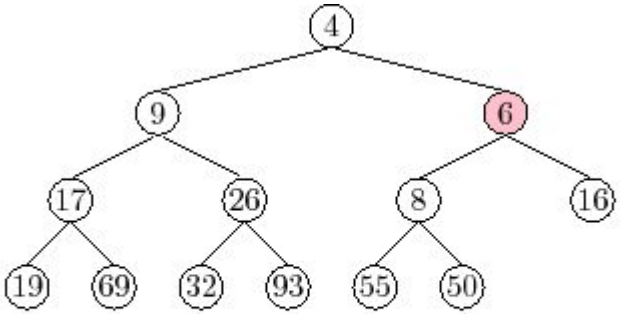
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- Sometimes it is also referred to as an almost complete binary tree.



Array Traversal

A Complete Binary Tree can be easily represented as array and array based representation is space efficient. If the parent node is stored at index i (0-indexed)

- The left child will be at 2 ∗ i + 1.
- The right child will be at 2 ∗ i + 2.



Main method

- In the main method, we first take the inorder traversal array size and the array as the input.
- Then we call the buildTree() function to convert the inorder traversal to a complete binary tree and store it in another array called tree.
- Then we verify the inorder traversal by calling the inorder() function which performs the inorder traversal on the complete binary tree.

Algorithm 1 Main Method

procedure MAIN

Input: arr, n

$buildTree(arr, tree, 0, n)$

$print(tree)$

$inorder(tree, 0, n)$

Build Tree

- An almost complete binary tree can be represented as an array.
- So this algorithm basically converts the given inorder traversal to an almost completely filled binary tree.

Algorithm 2 buildTree Algorithm

$k \leftarrow 0$
procedure BUILDTREE(arr, tree, i, n)
  if $i < n$ **then**
    $buildTree(arr, tree, 2 * i + 1, n)$
    $tree[i] \leftarrow arr[k]$
    $k \leftarrow k + 1$
    $buildTree(arr, tree, 2 * i + 2, n)$

In this algorithm we recursively insert the elements in the tree in the following fashion:

- Build the left sub-tree
- Insert the root element
- Build the right sub-tree

Here the insertion occurs in the same order as the given inorder traversal array.

This way, we'll construct an almost complete binary tree which has the same inorder traversal as the given array.

Inorder Traversal

In this kind of traversal we recursively print the elements of the tree in the following fashion:

1) Traverse left sub-tree

2) Print the root element

3) Traverse the right sub-tree

Algorithm 3 Inorder Traversal Algorithm

procedure INORDER(arr, i, n)

  if $i < n$ **then**

    $inorder(arr, 2 * i + 1, n)$

    $print(arr[i])$

    $inorder(arr, 2 * i + 2, n)$

This prints the inorder traversal of any given complete binary tree.

Comparison

- We have implemented an algorithm which converts the inorder traversal into an almost complete binary tree with a complexity of O(n).
- This is the best complexity one can achieve because given n elements, each one has to be inserted in the tree.
- Each insertion takes O(1) time.
- So n insertions would take: $n * O(1) = O(n)$

Therefore the minimum complexity one can achieve is O(n).

Algorithm 1

```
1: procedure MAIN()
2:     struct node* root ← NULL
3:     buildTree(tree, 0, n, input)
4:     root ← createBinaryTree(root, tree, 0, n)
5:     printinorder(root)
```

```
1: procedure void BUILDTREE(int tree[] ,int i ,int n ,int input[])
2:     if i<n then
3:         buildTree(tree, 2 * i + 1, n, input)
4:         tree[i] ← input[index + +]
5:         buildTree(tree, 2 * i + 2, n, input)
   =0
```

```
1: procedure void PRINTINORDER(struct node* root)
2:     if root− >left! =NULL then
3:         printInorder(root− > left)
4:     if root− >right! =NULL then
5:         printInorder(root− > right)
```

```
1: procedure structnode* CREATEBINARYTREE(struct node* root, int arr[], int i, int size)
2:     if i<size then
3:         struct node* newNode ← createNode(arr[i])
4:         root ← newNode
5:         root− > left ← createBinaryTree(root− > left, arr, (2 * i) + 1, size)
6:         root− > right ← createBinaryTree(root− > right, arr, (2 * i) + 2, size)
7:     return root;
```

```
1: procedure structnode* CREATENODE (int data)
2:     struct node* newNode ← (structnode*)malloc(sizeof(int))
3:     newNode− > n ← data
4:     newNode− > left ← NULL
5:     newNode− > right ← NULL
6:     return newNode
```

Complexity Analysis

Time Complexity

A binary tree in-order traversal can be described by the following recurrence relation, where T (n) is the time taken to traverse on a complete binary tree of size n.

$$T(n) = 2\ T(\frac{n}{2}) + O(1)$$
$$T(1) = O(1)$$

Here, the T (n/2) is for the traversal of both subtrees from the current node and the O(1) is for the current node

We can solve the recurrence relation as follows:

$$
\begin{aligned}
T(n) &= 2\ T(\frac{n}{2}) + 1 \\
&= 2\ [2\ T(\frac{n}{4}) + 1] + 1 \\
&= 4\ T(\frac{n}{4}) + 3 \\
&= 4\ [2\ T(\frac{n}{8}) + 1] + 3 \\
&\vdots \\
T(n) &= 2^k\ T(\frac{n}{2^k}) + (2^k - 1)
\end{aligned}
$$

For $T(1)$:

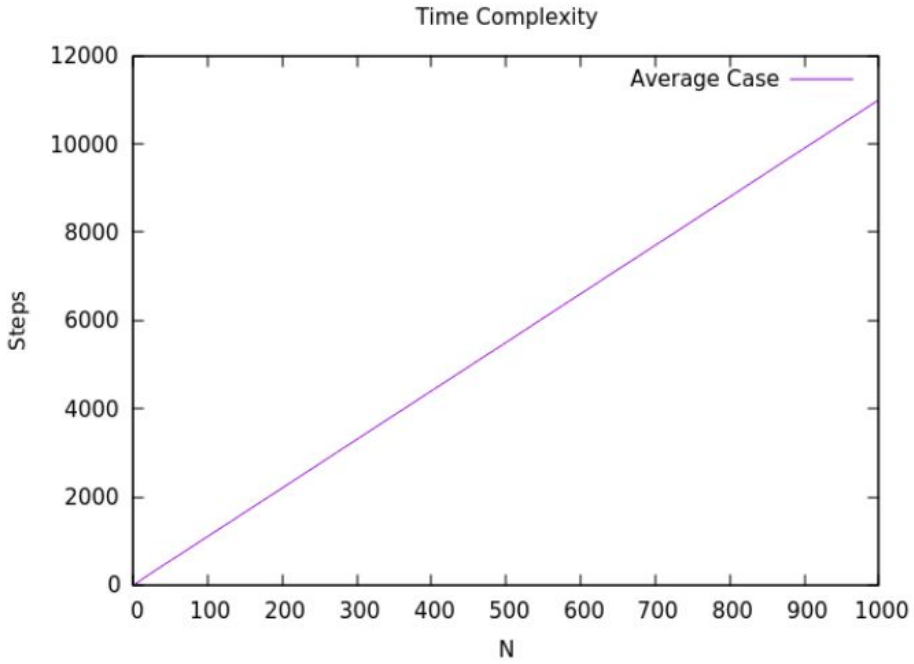
$$
\begin{aligned}
&\implies \frac{n}{2^k} = 1 \\
&\implies 2^k = n
\end{aligned}
$$

Entering the value of 2^k in the equation, we obtain the solution:

$$T(n) = 2n - 1$$

- Hence, the time complexity of the algorithm is O(n).
- There is no notion of best, average or worst case, because the computation time depends solely on the size of the input, not the values of the input elements.

Time Complexity



Space Complexity

- We are using the array representation of an almost complete binary tree hence for a tree containing n elements, we would require only n nodes.
- Hence the space complexity of our algorithm is O(n).

Conclusion

- We proposed the algorithm to construct an almost complete binary tree from the given inorder traversal.
- Moreover our algorithm has both time and space complexities of O(n).
- A tree is generally stored as a group of nodes where each node contains a key and the pointers to its children.
- This implementation takes at least thrice the space taken by an array.
- So we have used the array representation.


Thank you