3/21/2017

Assignment 2 Report

Binary Trees

Jasindan Rasalingam: 100584595
Darron Singh: 100584624
Kaushal Patel: 100586212
Ashwin Kamalakannan: 100584423
UOIT SOFE2715

Problem:

For a binary tree, one of the traverse results is given as an input, our algorithm should create the corresponding two other traverse results. For example if the in-order traversal is given, the pre-order and post-order should be created.

Our solution:

We made a few assumptions in order to understand the problem better and come up with a well-ordered solution. First, we assumed that we were supposed to create a binary search tree rather than a binary tree. The reason for this is that the binary search tree is simply a subset of a binary tree with the exception being that each node in a binary search tree must have a left child that is less than it and a right child that is greater than it. This actually made the problem more approachable to us as we can analyze things like runtime complexity much easier since we already know that the best case scenario for a binary search tree is log(n) time. The second assumption we made is that for the purposes of this assignment, the traversal data given to us as input is an integer array and it represents the most well balanced version of that particular binary search tree. The reason for this is that you can have multiple binary search trees that represent the same traversal, which would cause a lot of confusion during testing.

The data structure should have an average log(n) complexity for searching and inserting data. We are assuming each input is already in its balanced state as stated above, and while this will affect our search and insert algorithms, it does not have any negative impacts on the load method which will remain at O(n) time. This is because our implementation of the load method only relies on the length of the input array and not the structure of the nodes in the tree itself. Our print methods for the pre/in/post order traversals are all standard methods that work for both binary and binary search trees. Each of these has linear O(n) complexity, relying on the total number of nodes in the tree.

Pseudo Code

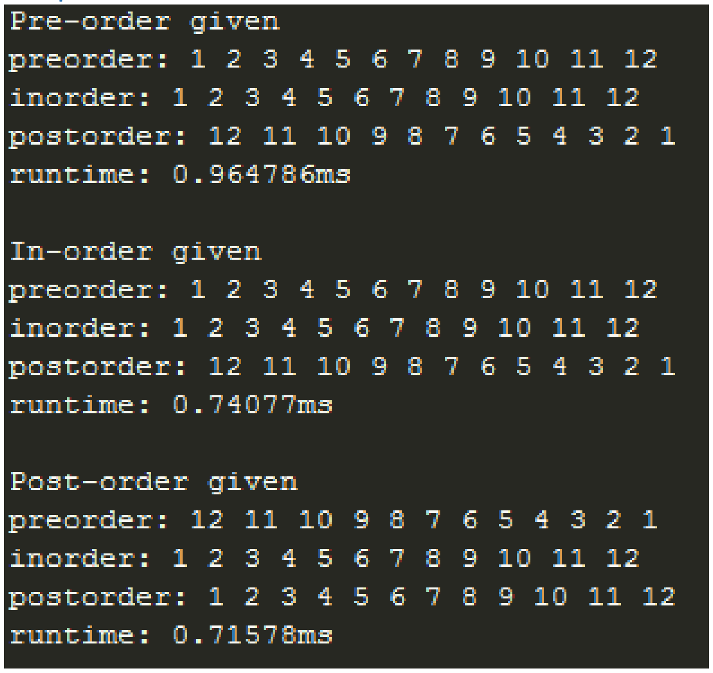
```
load(array, string) {
    if string = preorder
        Node <- first array element
        start <- 1, finish <- array_len, dir <- 1
    elif string = inorder
        Node <- first array element
        start <- 1, finish <- array_len, dir <- 1
    elif string = postorder
        Node <- last array element
        start <- array[array_len], finish <- -1, dir <- -1

    forloop  I <- start, i != finish, i <- dir + i
        if i less than Node
            if Node.left not exist
                create new node and insert i -> Node.left
            else
                insert i -> Node.left
        else if i greater than Node
            if Node.right not exist
                create new node and insert i -> Node.right
```

```
                    else
                              insert i -> Node.right
          else
                    insert i -> Node
     return Node
}

printPreOrder() {
     Print current
     If left not null
          Left -> printPreOrder()
     If right not null
          Right -> printPreOrder()
}
printInOrder() {
     If left not null
          Left -> printInOrder()
     Print current
     If right not null
          Right -> printInOrder()
}
printPostOrder() {
     If left not null
          Left -> printPostOrder()
     If right not null
          Right -> printPostOrder
     Print current
}
```

Output



```
Pre-order given
preorder: 1 2 3 4 5 6 7 8 9 10 11 12
inorder: 1 2 3 4 5 6 7 8 9 10 11 12
postorder: 12 11 10 9 8 7 6 5 4 3 2 1
runtime: 0.964786ms

In-order given
preorder: 1 2 3 4 5 6 7 8 9 10 11 12
inorder: 1 2 3 4 5 6 7 8 9 10 11 12
postorder: 12 11 10 9 8 7 6 5 4 3 2 1
runtime: 0.74077ms

Post-order given
preorder: 12 11 10 9 8 7 6 5 4 3 2 1
inorder: 1 2 3 4 5 6 7 8 9 10 11 12
postorder: 1 2 3 4 5 6 7 8 9 10 11 12
runtime: 0.71578ms
```

Results

Given traversal	Input	Average Runtime
pre-order	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}	0.8014 ms
in-order	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}	0.5984 ms
post-order	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}	0.7648 ms

Analysis

We noticed during testing that whichever test case we called first always had the slowest runtime. This was obviously strange and not a negative aspect of the algorithm itself. Instead, we believe that the initial slow runtime is due to the Java environment having to load resources into the RAM. In addition, the values for each of the three test cases are stored in the same variables, within the same main function. This means that they are using the same memory locations and were not deallocated after the previous run, allowing the system to quickly store and retrieve the data for the later test cases without the need to reallocate space in memory. To solve this issue and keep our runtimes consistent, we simply do an empty call of the load method at the beginning of the program without printing any data. This means that we do not have to add this process to our runtime calculation and the memory for all of the variables will be allocated already.

Our print methods and load method are both linear time, however the insert method is log (n) time for best-case scenario. For each iteration in the load method, we have a single insert method call, this means that the overall complexity of our algorithm is n*log (n) best case and n^2 in the worst case. If we wanted to guarantee an n*log (n) complexity, we would have to implement a balancing algorithm. However, adding a self-balancing feature would mean that we are essentially creating a splay tree, which is not a requirement of this assignment. Upon further analysis, in a real world scenario we would expect to be given an actual binary search tree as our input rather than just a string representing the traversal of the tree. This would remove the n*log (n) load time and only require the print methods, which would allow our program to have an overall linear complexity. In addition, we managed to optimize the number of lines of code we had in our load method by dynamically initializing the parameters of the for-loop. This results in more readable and efficient code that can easily be changed or implemented into a larger project. Despite some initial confusion regarding the nature of the given input, we managed to implement a clean and concise algorithm that produces both correct results and consistent runtimes.