CS6140: Advanced Programming Lab
End Sem

Nov 13, 2014

1 Depth First Search on Trees and Applications

The goal of this problem is to use Depth First Search on a **binary** tree to answer several interesting questions. Given a binary tree, implement a depth first search on the tree and for each node u in the tree compute the pre-visit number (u), post-visit number (u), and height of the tree rooted at the node u.

The tree is given to you in a format which is slightly different from the usual graph representation. The input file consists of n bits (each bit is on a separate line) terminated by a special marker "end". Assume that the tree contains nodes which have ids ranging from 1 upto n with possibly nodes corresponding to some ids missing. The n bits in the input indicate precisely which node ids are present. That is, if the ith bit in the input is 1, the node with id i is present in the tree, otherwise if the ith bit is 0, the node with id i is absent. The root has node id 1 and for any node with id i, the left child (if present) has id $2i$ and right child (if present) has id $2i + 1$. Recall that this is similar to the heap representation in an array, except in a heap there cannot be missing nodes in between. You may assume that the largest index of any node is upper bounded by 1025.

You are given the following sources.

- main.cpp [do not edit]
- BinaryTree.cpp [do not edit]
- functions.cpp
- Input.txt (input file)

The file Input.txt is the input file whose format has been described earlier. The main program takes as an input 2 files, one the input file and another an output file. Given an input file, a readFile function (already implemented in BinaryTree.cpp) will read the input and populate a boolean array nodes where the i-th bit is 1 if and only if the ith node is present in our tree (correspondingly the i-th bit is one in our input file). For the ease of visualization of the tree a function gvout has been implemented in BinaryTree.cpp which produces the gv file containing the tree represented in "dot" format. You can view the output file by converting it into an eps file using the command:

dot -Tps *outputfilename* -o out.eps

You need to do the following:

- Implement a depth first search on the tree. The class BinaryTree already gives a wrapper function DFS() which is a public function and the signature for a function called *DFS_visit()*. Your goal is to implement this function DFS_visit. Recall the format in which the graph is specified and suitably tune the code of DFS_visit for that. While implementing DFS, traverse the left child of a node first and after that traverse the right child. In addition while doing DFS, ensure that for every node, we compute the following 3 values – (1) previsit number, (2) postvisit number, (3) the height of the subtree rooted at the node.

  The private variable visit_time provided in the class definition may be used to compute the previsit and postvisit numbers. Note that the height of NULL node is 0 and the height of any node is recursively defined as usual.

  Your DFS_visit function must populate for every node i present in the tree pre[i], post[i], and height[i]. Implementing this correctly will fetch you 5 marks.

The following functions are to be implemented using the pre-processing done by DFS on the tree. Further, note that these functions are not members of the class so they do not have access to private variables of the class. However, all the necessary interfaces are already provided via public functions of the class BinaryTree. To implement the functions below, make sure you do NOT modify the class definition.

1. **Ancestor relation between nodes**: Given a Binary Tree T and the ids of 2 nodes u and v of the tree (which are necessarily present in the tree), implement a function *isAncestor* which returns true if u is an ancestor of v in the tree. The function returns false otherwise. A correct implementation of this which runs in $O(1)$ time will fetch you **1** mark.

2. **Lowest Common Ancestor**: Given a Binary Tree T and the ids of 2 nodes u and v of the tree (which are necessarily present in the tree), implement a function LCA which returns the id of the node which is the Lowest Common Ancestor of u and v in the tree. Recall that the LCA of 2 nodes u and v is node x which is an ancestor of both u and v and no descendant of x is an ancestor of both u and v. A correct implementation of this will fetch you **3** marks.

3. **Size of the Subtree**: Given a Binary Tree T and the id of a node u (which is necessarily present in the tree), implement a function $SubtreeSize$ which returns the number of nodes in the subtree rooted at u including the node u. Ensure that your function takes $O(1)$ time. A correct implementation of this will fetch you **2** marks.

4. **Balanced nodes**: Given a Binary Tree T and the id of a node u which is necessarily present in the tree, implement a function *isBalanced* which returns true if and only if the subtree in T rooted at u is height balanced. A (sub)tree is height balanced if for every node in the subtree, the heights of its two children differ by at most 1. A correct implementation of this will fetch you **1** mark.

Only if you finish all of the above, consider the following question.

- **Diameter of tree**: Given a tree, compute the diameter of the tree in $O(n)$ time where n denotes the index of largest node present in the tree. A correct implementation will fetch you **3** marks.

2 Submission

You are required to submit your "functions.cpp" file, but it must be renamed "⟨rollnumber⟩.cpp" and tar zipped before you submit it. Please ensure that the file compiles without any error before you submit it.