CS 310
Assignment 223

Hieu Le

1. Explain the changes you made to the program and the algorithm you used to implement the traversal.

Answer:

The function `levelorder()` implements a level order traversal of a tree with unrestricted number of children per node. The function takes in a pointer to the root of the tree as the sole argument and prints out the content at each of its node, level by level. It does so by iteratively visiting all nodes at every depth of the tree. This approach has the flavor of a bread-first search algorithm, and certainly so with the exception being that the search does not terminate until all the nodes have been visited.

The test `root == nullptr` checks whether the tree is empty. If the test is true, `levelorder()` returns since there is nothing to print from the tree. The test is important since it allows the function to execute without any error when an empty tree is input. Since the function is iterative in nature, this test differs from the base case test for `nullptr` leaf nodes found in many recursive tree procedures.

The tree nodes yet to visit is stored in a queue object called `frontier`. The use of a sequential data structure that supports first-in first-out element access is crucial since we want to ensure the 'lefter' node at a smaller depth is visited first. The use of a single queue throughout the function means that there is no distinguishing between nodes located at different depths. This is acceptable since the content of the nodes is printed on the same line.

```
tree_node * child = current->get_first_child();
while( child != nullptr )
{
    frontier.push( child );
    child = child->get_right_sibling();
}
```

The nested while loop shown above iterates through all the children of `current` which is the node being visited and push them onto `frontier`. The outer while loop terminates when `frontier.empty() == true`, i.e all the nodes in the tree have been visited.

As is the case with most tree traversal algorithms, the running time complexity of this algorithm is $\Theta(n)$ where n represents the number of nodes in the tree. The algorithm is linear since each node is pushed onto the queue exactly once.

2. For an arbitrary non-empty binary search tree, find a precise expression for the number of `nullptr`. State and explain your answer.

Answer:

Method 1. By observation.

Let T be a non-empty binary tree with n nodes. Since each node contains 2 pointers to its left and right children, there are a total of $2n$ pointers in T. Each pointer either points to an interior node excluding the root of the tree or a `nullptr`. As there are $n-1$ such interior nodes, there must be $2n-(n-1)=n+1$ `nullptr` in T.

Method 2. By induction.

Proposition: a non-empty binary tree with n nodes has $n+1$ `nullptr`.

When $n=1$, the proposition holds since the tree consists of a single node and 2 `nullptr`.

Assume that for an arbitrary positive integer k, the number of `nullptr` in every non-empty binary tree of size $n \leq k$ is $n+1$. Let T be an arbitrary binary tree of size $n=k+1$. Let T_L and T_R be the left and right subtrees of T respectively. Subsequently, T_L and T_R cannot both be empty since there are at least 2 nodes in T and therefore at least 1 node in one of these subtrees.

Case 1: Both subtrees are non-empty. Therefore, the number of `nullptr` of T is the sum of the numbers of `nullptr` from each of these subtrees. Let $size(X)$ denotes the size of a tree X. By the inductive hypothesis, the number of `nullptr` of T is equal to:

$$(size(T_L)+1)+(size(T_R)+1)=(size(T_L)+size(T_R)+1)+1=size(T)+1$$

Case 2: One of the subtrees is empty. Without loss of generality, assume that T_L is empty. Subsequently, the number of `nullptr` of T is equal to:

$$1+(size(T_R)+1)=size(T)+1$$

By mathematical induction, it is therefore proven that the number of `nullptr` in a non-empty binary tree is 1 more than the number of nodes in the tree.

Since a binary search tree is itself a binary tree, the proven result can be applied to a binary search tree.