Successor in Binary Search Trees

Nitin Verma
mathsanew.com

October 22, 2020

A *Binary Search Tree* (BST) is structurally same as a *Binary Tree*, but its nodes' keys follow certain property. When we define the *successor* of a node in BST, we can only depend upon its Binary Tree structure, not its keys. In a BST T, the *successor* of a node x can be defined to be the node which is processed just after x during *Inorder Tree Walk* of T (Chapter 12 in [1]). If x is the last processed node, then its successor is null. In this article, we will understand the method to locate successor of x starting with x.

Figure 1: Inorder Tree Walk

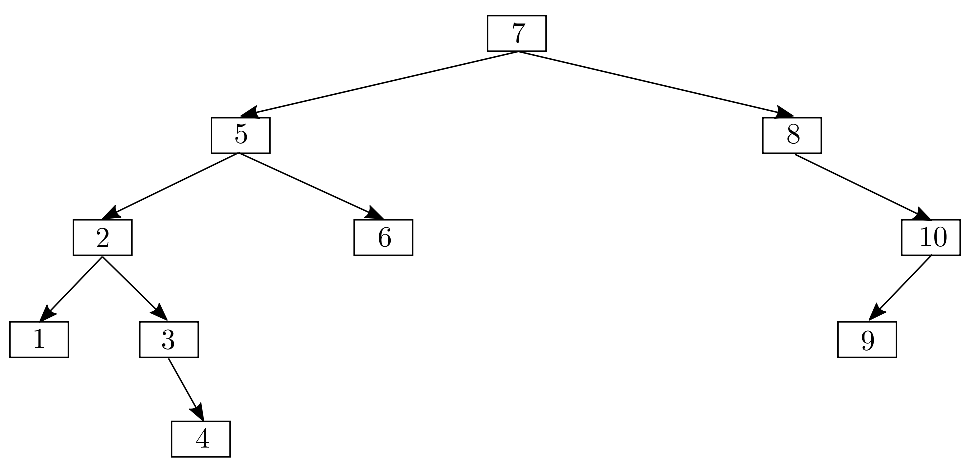

Figure 1 shows a Binary Tree with its nodes labeled according to their order of processing during Inorder Tree Walk. Observe where the first (1) and last processed (10) nodes are located in the tree. And how a node and its successor are located relative to each other, specially node 8's successor 9, and node 4's successor 5.

The recursive method for Inorder Tree Walk closely resembles the very definition of Inorder Tree Walk, and can be written in C language as:

```
void inorder(node *x)
{
  if(x->left)      inorder(x->left);   /* step (1) */

  process(x);                          /* step (2) */

  if(x->right)     inorder(x->right);  /* step (3) */
}
```

`node` represents a structure for tree nodes, its `left/right` members store the left/right child's pointers. The initial call has T's root as x.

We will try to relate the locations of a node and its successor just by understanding the above method's structure. First, we can conclude the following about the first and last processed nodes during $inorder(x)$, where x is any node in T:

1. Say f is the first node processed. Note that, at any point, all calls on the callstack except the topmost (latest) one, must be at their recursive step: step (1) or (3). Since $inorder(f)$ must be the first call which reaches step (2), its step (1) must not have recursed. Also, no invocation of this method has yet reached step (3) which can happen only after step (2). So, all calls starting at $inorder(x)$ till the caller of $inorder(f)$, have recursed only via step (1), by accessing the *left* pointer. That is, path from x to f consists of 0 or more *left* pointers, and $f \rightarrow left$ must be null.

2. Say l is the last node processed. Since $inorder(l)$ must be the last one which reaches step (2), no other call on the callstack must have recursed via step (1) as that is always followed by step (2) of processing the node. It means, they must have recursed via step (3), by accessing

the *right* pointer. Further, step (3) in $inorder(l)$ must not result in the recursive call. That is, path from x to l consists of 0 or more *right* pointers, and $l \rightarrow right$ must be null.

We now come to the problem of finding successor s of x. Recall that s is the node processed just after x during inorder walk of T. During this tree walk, consider the invocation of $inorder(x)$, where x is processed in step (2). Now, if x has the right-child x_R, then immediately after processing x, step (3) will do $inorder(x_R)$. Since any invocation of $inorder()$ processes at least one node, so will that of $inorder(x_R)$. The very first node processed during this will be the successor s of x. By applying conclusion (1) to $inorder(x_R)$, we can say that the path from x_R to s consists of 0 or more *left* pointers, and $s \rightarrow left$ must be null. That is, path from x to s consists of a *right* pointer, then 0 or more *left* pointers, such that $s \rightarrow left$ is null. In Figure 1, node 8's successor 9 is such an example.

But if x has no right-child, the call to $inorder(x)$ will return just after processing x, without processing any other node. Consider all other calls of $inorder()$ on the callstack, during $inorder(x)$. All these calls are for nodes ancestors to x; from root node till parent of x. Call the closest ancestor of x (its parent) as a_1, a_1's parent as a_2, a_2's parent as a_3 etc. So, $inorder(x)$ was called by $inorder(a_1)$. When $inorder(x)$ returns, $inorder(a_1)$ must have completed either step (1) or (3). If it is step (1), then next processed node is a_1, making it the successor s. If it is step (3), $inorder(a_1)$ would return to its caller, $inorder(a_2)$. Again, $inorder(a_2)$ must have completed either step (1) or (3), and similar argument applies. Thus, the successor s of x is its closest ancestor such that $inorder(s)$ is at step (1) during $inorder(x)$.

All ancestors closer than s are at step (3) and will simply return once x is processed. And then, $inorder(s)$ will process node s via its step (2). Say, left-child of s is s_L. x is the last node processed during step (1) of $inorder(s)$, i.e. $inorder(s_L)$. Hence, if x has no right-child, the path from s to x consists of a *left* pointer (accessed in step (1)), followed by 0 or more *right* pointers (accessed in step (3)). In Figure 1, node 4's successor 5 is such an example.

It is also possible that no ancestor of x is at step (1) during $inorder(x)$. Then, the path from root to x consists of 0 or more *right* pointers. By applying conclusion (2) to $inorder(root)$, x is the last processed node of T with no successor.

To summarize:

1. If x has the right-child x_R, then s is the first node processed during *inorder*(x_R).

2. Otherwise, s is its closest ancestor whose left-child s_L appears in path from s to x. x is the last node processed during *inorder*(s_L).

Alternate Derivation

Here is an alternate way to derive the relation between locations of x and s. For below discussion, we allow ancestors of a node to include that node itself. We know that any two nodes in a binary tree must have at least one common ancestor. Say, c is the common ancestor of x and s closest to them. During *inorder*(c), both x and s must get processed. They both cannot get processed during step (1) of *inorder*(c), because then c's left-child would be their closest common ancestor, a contradiction. Similarly, they both cannot get processed during the step (3). Also, x and s should not get processed during step (1) and (3) respectively, because step (2) in the middle processes c, but s is the successor of x.

This leaves only below possibilities:

1. x is processed during step (1) and so its successor s must be processed at step (2), meaning $c = s$. x has to be the last node processed during step (1) of *inorder*$(c = s)$. In this case, s is an ancestor of x.

2. x is processed at step (2) and so its successor s must be processed during step (3), meaning $c = x$. s has to be the first node processed during step (3) of *inorder*$(c = x)$. In this case, x is an ancestor of s.

(The second possibility corresponds to the case of last section with x having the right-child, and first possibility corresponds to x having no right-child.)

By earlier conclusions on page 2, we already know how to locate the first and last node processed during *inorder*() call of any node.

We can use similar reasoning to locate the next processed node after a given node during *Preorder* and *Postorder Tree Walks*.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*, Third Edition. The MIT Press, 2009.