Navigating Trees

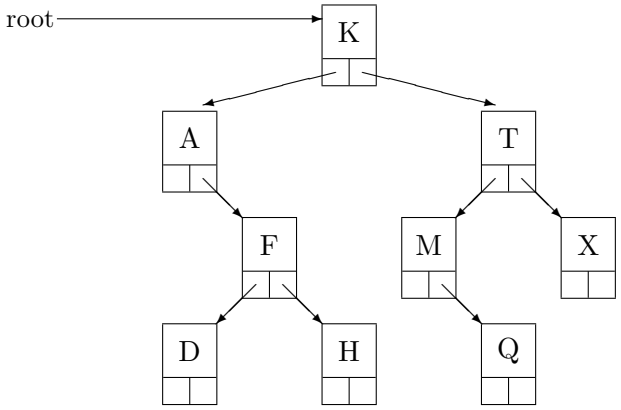
1 Introduction

This handout discusses variations and extensions to binary search trees to handle in-order traversal using one of the following techniques:

- Repeated search;
- An explicit stack;
- Parent pointers;
- Threading.

2 Iterating by Repeated Search

In this document, we consider a binary search tree of the following form:



A simple technique to implement iterators for in-order traversal is to simply search for the next element given the previous element. Of course we also need a place to start: the minimum element, but finding the minimum is very easy: start at the root and go left as far as possible. In our example case, this would lead to the "A" node.

Finding the next element uses a variation on binary search, in which we don't accept equality, but instead go right (toward bigger keys) when a match is found. Of course, since we don't accept equality, we won't stop going down the tree until we fall off the tree. The key is to remember the *smallest* key larger than what we are searching for that we have encountered during the search.

For example, if looking for the "next" node after "A," we start with the root, go left to "A," then go right to "F," left to "D" and go left, falling off the tree. The *smallest* key larger than the goal ("A") is the one in last node we went left from. In the case of the example, this is node "D."

For another example, consider finding the next key after "Q." We search for "Q" (rejecting equality) and visit (in turn) "K" (right), "T" (left), "M" (right), and "Q" (right) and then fall off the tree. The only (and last) time we went left is from the "T" node, and thus "T" is the next key.

Repeated search takes time $O(n \lg n)$ which is slower than $O(n)$, the time for an in order traversal. Thus in-order traversal is preferable to repeated search. The problem is that recursion does not lend itself to an iterator implementation because an iterator must leave control to the caller. In the next section, we show how we can use an explicit stack to replace recursive in-order traversal with non-recursive in-order traversal.

Self-Test Exercises

1. Describe how one would find the next key after "F" using an equality rejecting search.
2. Describe how one would find the next key after "K."
3. Describe how one would find the next key after "X."
4. Why can't `next()` use recursion? Give an example. (Hint: why doesn't it work for the "H" node case.)

3 Iterating Using a Stack

Recursive in-order traversal works as following:

> If not null:
>
> 1. Traverse the left subtree;
> 2. Visit this node;
> 3. Traverse the right subtree.

Now while we are traversing the *left* subtree, we still need to come back to this node. On the other hand, once we are done traversing the *right* subtree, we have nothing more to do.

When implementing an iterator, we can't use the control-stack of program points (recursion), we need an explicit stack of nodes. We will keep a collection of nodes that we have to come back to later in iteration. We use a stack because we want to handle a whole subtree before going on outside of that subtree. We want the most recently saved node.

So the idea is that we're going down the tree looking for the first element in the subtree. Since the earlier nodes are to the left, we go left. So we just travel left until we fall off the tree (hit the inevitable null pointer). But, whenever we leave a node (going left), we remember that we need to come back to the node later, and so push it on the "pending" stack.

After we've fallen off the tree, the pending stack remembers what we wantes to look at when someone one starts the iteration. when we pop off a node, we will then need to do something with its right subtree, but the node is now past, so we don't push it on the stack (when we go right).

In general, *processing a tree* will be going down the tree, always to the left, pushing every node as we go to its (left) child. When we start up the iterator, we process the root.

For example, suppose we have a small tree with root node N1 with value 10, with two leaves, N0 and N2, with values 1 and 100 respectively. When we start the iterator, we process the tree, by starting at the root and going left until we fall off the tree, pushing nodes as we go, so the stack starts off with N0 on top with N1 under it.

- N1, N0

  When `next()` is called, we pop off N0, and remember that we're going to return its value (1), once we process its right subtree, going left pushing nodes. The right subtree of N0 is null, so nothing happens. We just return 1.

- N1

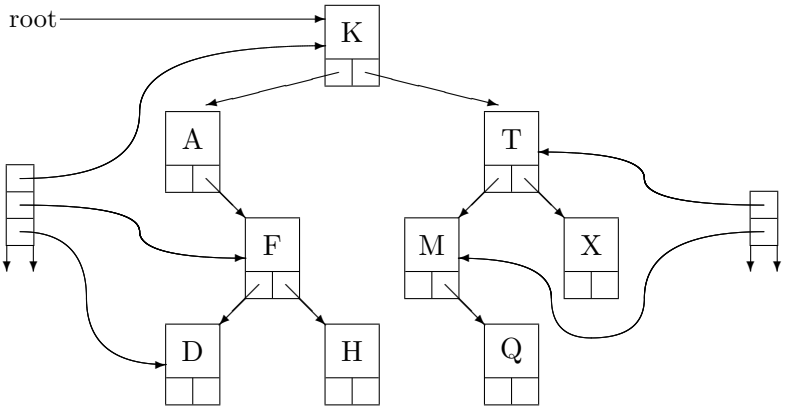
  When `next()` is called again, we pop off N1, and remember that we're going to return its value (10), once we process its right subtree. This time the right subtree is not empty, so we go all the way left on it, pushing nodes as we go. In this case, just one (N2). Then we return the saved value (10).

- N2

  When `next()` is called the third time, we pop off N2, and remember that we're going to return its value (100), once we process its right subtree. But since there's nothing there, so we just return 100.

- If someone calls `next()` again, we throw an exception since there is no next value.

Let's look at a bigger example now.



Thus when iterating in order over the tree, we start by pushing the "K" node and then immediately the "A" node, as we fall off the tree to the extreme left. This is the initial state of the stack when starting iteration. When we're asked to return the next value, we pop off the "A" node and remember that we are going to return "A." But before returning, we process the "A" node's right-subtree: the "F" node, which immediately pushes itself and goes onto "D" and we have the situation shown to the left of the tree: a stack with pointers in it to the "K," "F" and "D" nodes. This is the second value of the stack. A later value of the stack is given on the right-hand side of the picture.

In general, when asked to get the "next" value, we pop the top node off, remember its value, and process its right subtree (pushing the nodes down the left side) and finally return the saved value.

Self-Test Exercises

5. When does an iterator still have more to do? (In other words: when does `hasNext()` return true?)

6. If `hasNext()` is true, what will be the key returned by `next` ?

7. The stack to the left of the tree has all the ancestors starting from "D" except for "A." Why is "A" omitted?

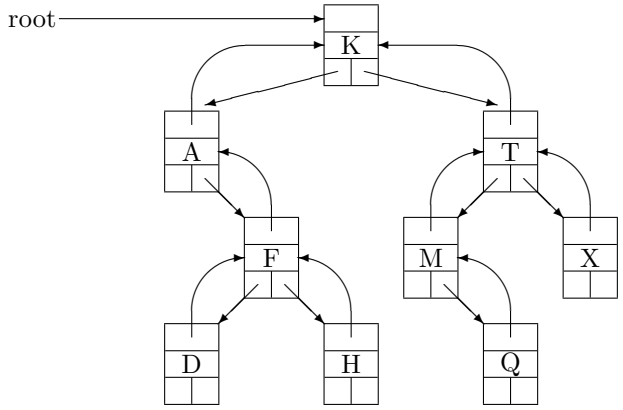
8. In general the stack will never include ancestors that . . . (fill in the blank).

9. What should the stack to the left of the tree look like next?

10. What should the stack to the right of the tree look like next?

4 Parent Pointers

Instead of using a stack, we can traverse the tree using parent pointers:



Adding the parent pointers adds more redundancy to the data structure, and thus the need for a more complex invariant. Fortunately, the `_checkInRange` helper method can be enlisted to help check parent pointers too: the parent is passed in as a parameter and if the node parameter is not null, we simply check that the parent pointer is correctly set.

Adding the parent pointers also complicates addition or removal. In the case of addition, we can easily use the "lag" pointer to initialize the parent pointer, or, if using a recursive helper function, we can provide another parameter (as with `_checkInRange`) with what the parent pointer should be if a node needs to be added here. Removal of a node by re-parenting orphans will require the grandchild's parent pointer to be updated to the new parent. Since we have to change the pointer in the parent anyway, this is not a major difficulty.

Once we have (valid!) parent pointers, a cursor can just be a pointer to a node. When we want to find the next node, we see if we can go right. If there is a right child (for example, when finding the next node after the "K" node), we go right and then left as far as possible (for example, right to "T" and then to "M" where we stop).

On the other hand, if there is no right child, we have to go up. If we were a right child of our parent (for example when finding the next node after "H") then we have to keep going up, until we emerge as the left child of some node (in this case, we go up to "F", and then "A" and finally emerge as the left child of "K"). At this point (i.e., at the "K" node) we stop. If we fall off the root of the tree going up (e.g. "X" to "T" to "K" to *null*), then there is no next node.

Self-Test Exercises

11. Suppose a node to remove has two children (such as the "K" node), and thus we replace it with its closest successor. What parent pointers, if any, need to change? Explain!

12. If we need to remove the node that the cursor is on, we need to move the cursor to the *previous* node. Explain the algorithm for how to get there.

13. What is the next node after the "A" node? Describe how we get there.

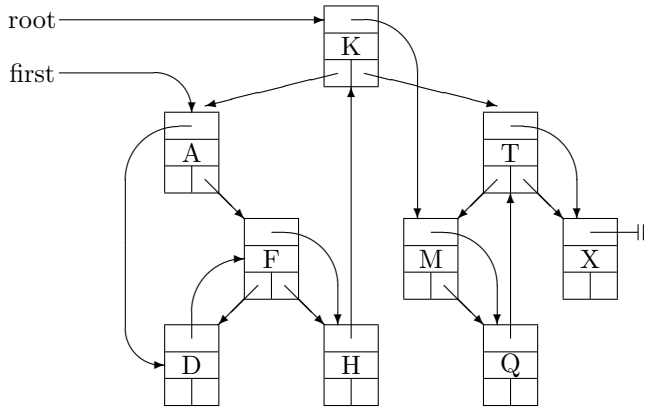
14. What is the next node after the "D" node? Describe how we get there.

15. What is the next node after the "F" node? Describe how we get there.

16. What is the next node after the "Q" node? Describe how we get there.

5 Threading

The final way described in these notes to implement in-order traversal uses threading in the node so that traversal can be implemented as easily as it is in a linked list: We link the nodes in order.



This variation of the binary search tree makes iteration the easiest but is the hardest to check and keep up-to-date. We need an additional recursive private helper method that checks all the links using a traditional (recursive) in-order traversal; in this case, in *reverse* order so we can check that each node points to the correct next node. The helper method will (as well as the ubiquitous subtree parameter) take the required next node and return the first node in the tree traversed.

When a new node is inserted in the tree, we need to link it into the sequence. This means our recursive helper function, or our iterative loop, will need, not only to keep track of a "lag" pointer, but also to keep track of the most recent node we went right from, which is the previous node in the linked list, should we need to add a new node. Removal is even messier.

Self-Test Exercises

17. Write the private recursive helper method to check the "next" links.

18. Explain why the predecessor of a newly added node is always above it in the tree.

19. Describe all that needs to change if node "M" needs to be removed.