Data Structures

Binary Trees (with slide deck "BinaryTrees.ppt", version 1.2)

Slides 2-12

The theory behind the structure of the binary tree, the insert algorithm and an example are described here. One takeaway from reviewing this is the thinking behind a "binary tree" and a "binary search tree". The first few slides discuss a "binary tree" which focuses on the structure itself (a node having a left and right node pointers). Many of the examples we discuss in these slides will apply the binary tree structure by demonstrating a "binary search tree" which you can think of as a binary tree that is designed to simulate/apply the binary search algorithm. As you will see with other applications, the binary tree structure can be used in other ways.

Put simply: what's a binary tree? A dynamic data structure where each node has a left and a right pointer to another node. What is a binary search tree? It's a binary tree designed to simulate binary search. The goal is to have a structure where memory is only used when necessary and still have the same benefits of using the binary search algorithm.

Slide 13

The code is presented in these slides and also available in the code supplement on the web site. In this example, BinaryTree is a class that contains a root Node. Notice that this is the same structure definition as a linked list. In a standard linked list, a node is defined with a data property and a "next" pointer, and then the LinkedList itself contains a "head" node representing the start of the list.

In a binary tree, it's the same pattern. A node is defined with a data property and "left" and "right" pointers, and then the BinaryTree itself is contains a "root" node representing the start of the tree.

Slides 14-17

The code is presented on slides 14-15 and discussed on 16. We examine the effectiveness of this on slide 17. The main takeaway here is that when we look at a node and decide we need to "traverse left" or "traverse right", we are effectively ignoring the respective other side. This means if we are looking at a value of 100 in a node and decide to go left, any value on the right of it will be ignored. Given the rules of the binary search tree, this also means any value greater than 100 is being ignored.

This is the same concept with the binary search algorithm! Now if we were to have a binary search tree that is "balanced" (i.e. there are an equal number of nodes on each side of each node), traversing the tree is performing at O(log n) like binary search because with each node visit, you are effectively eliminating searching an entire half of a node.

This assumes, of course, that the tree is balanced. As with all data structures, there are pros and cons. At this point, you may have already noticed that a con here is that the tree with the insert algorithm depends on the nature of the data going in. This means that if you were to insert only positive numbers and the first number you insert is a 2, then the root of the tree will be 2. The only possibly value on its left will be a one, while the right can contain values greater than 2. Furthermore, if you next insert a 5, then from 5 the only nodes to its left could be a 3 and a 4, while on its right will be nodes greater than 5.

This leads to what we call an unbalanced binary tree. What's the worst case unbalanced tree? One where all the nodes are on one side (i.e. nodes inserted in ascending or descending order). This will degrade the search performance from O(log n) to an unfortunate O(n).

Slides 18-21

For a random distribution of inserted data, though, the tree will lean more towards balanced structure that allows us to simulate binary search. The theory and code is discussed on these slides.

(Bonus thought: slide 19 shows the search function – could this be done recursively??)

Slides 22-29

It is important from these slides to be familiar with using the terminology of tree structures. These may seem intuitive but are still good to internalize. Advanced tree structures will follow similar patterns so it will be good to know when someone talks about a "tree path" or the "height" of a tree, you can easily visualize it. Tree terminology can be fun and doesn't get overly fancy (you won't really need to know how to define a node's "cousin" or "uncle", but "parent", "child", "sibling", and sometimes "grandparent" are good to know).

Also, as you can probably see with binary trees, maintaining balance is an interesting concept. There are other advanced data structures like the "AVL" trees or the "Red-Black" trees that are interesting concepts to study which focus on the notion of self balancing (i.e. as nodes are inserted, the tree maintains balance). We will look at a self-balancing tree called a 2-3-4 tree later.

What makes these structures even more challenging though is when we look at deleting and updating nodes.

Slides 30-65

Normally we look at update first, but for a tree, updating a node's value and moving it to a new spot relative to where it is currently is not a simple feat. It's better to look at delete first and then do an update as a "search, delete, and insert". However, delete in a binary search tree is the pinnacle of tree maintenance complexity.

These slides go into detail of how the delete algorithm works. Go through these carefully and take your time with each slide. The logic makes sense early on but

once you get to the situation where the node to delete has 2 children, the complexity rises significantly. It's even more important at that point to study each slide slowly.

Slides 66-72

The code is no simple matter either, but it is as systematic as the theory described in the previous slides. Slide 66 starts with checking for an empty tree. If it's not empty, we start the search. Slide 67 shows how the search is done while also maintaining the a few variables:

1) current – which is the current node being examined to delete
2) parent – which is the current node's parent
3) currentIsLeft – this is a Boolean value that if true, means the current node being looked at is the left child of parent (this will be needed if current ends up being deleted and we need to set the parent node's appropriate child pointer)

If we don't find the node to delete, current will reach null and return false. Otherwise, we found it. Now we check for 1 of the 3 delete situations.

Slide 68 is situation 1. If current is a leaf, check to see if parent and current are the same. If they are, the current is still at root. Since current is a leaf node in this situation, this means the tree has just one node, so we simply delete the tree and set root to null.

Otherwise, we need to know if the current node being deleted is to the left or right of the parent. This is where we use the "currentIsLeft" Boolean value. If it's true, then to delete current, we simply set the parent's left child to null. Otherwise, set the right to null. Done.

Slide 69 is situation 2. It's a similar pattern of code as situation 1. First, see if current is root. If so, set the root to the appropriate child pointer of the current node being deleted. Otherwise, using the currentIsLeft Boolean value again, set the appropriate child pointer of current's parent to current's one and only child. It's a complex set of if…else statements, but the logic holds and fits nicely on one slide.

Now situation 3. Slide 70 starts in very functional manner, to first identify what the successor node is by calling "getSuccessor". Assuming that works, we use the same approach with the previous situation code and either set root to the successor (if current was root) or set the appropriate parent's child pointer to the successor.

We're almost done at this point. The deleted node's 2 children must be taken on by the successor node. getSuccessor will automatically take care of the right child as will be seen on the next slides so all that remains at this point of the code is to set the successor's left child to take on the deleted node's left child.

Now let's look at the getSuccessor function itself on slide 71. We follow a similar search pattern using a local version of a current node (not to be confused with current in the delete function) to determine what the successor and successor parent nodes are. We start on the right of current (i.e. go into the right subtree of the

deleted node), and then go left as far as possible using current. Once current hits null, successor and successorParent are identified.

Slide 72 is quite brilliantly handling the remaining steps! At this point, the successor will either be somewhere down the right subtree or be the first node of it (i.e. we didn't go down at all). If we don't go down any of the left nodes of the right subtree, then the successor is the right node of the node being deleted. In this case, when successor replaces the node to be deleted, its own right subtree (if any exists) is already correctly attached.

If we did go down at least one left node in the right subtree, then we have to make a couple of adjustments with the successor node. First, set its parent's left child to whatever is on the successor's right. This is the situation in slides 58-60 (node 12 is the successor to replace node 11 and it is somewhere down the left branch of 11's right subtree. The code "successorParent.left = successor.right" is setting node 15's left child (15 is the parent of successor 12) to 14 (which is the successor's right child).

This simple line of code will <u>always</u> apply correctly in this situation when the successor node is somewhere down the left branch of the right subtree of the node being deleted. The successor parent will always need to adjust its left child and it will always be set to the successor's right child.

The second line in that if statement back on slide 72 is now setting the successor's right child to the first node of the right subtree being deleted. In slide 70, we saw that when getSuccessor is done, the successor's right child is already correctly set to the right subtree of the node being deleted. This second line in slide 72 is doing just that. Going back to the example on slides 58-60, this line is setting node 12's right child to node 20.

That's it! The logic and the code is no simple matter, but it works, and works quite well. As before, study, practice, and draw pictures. You can probably guess what kind of questions would be asked on an exam when inserting, searching, and deleting nodes in a tree in different situations…

<u>Slides 73-77</u>

Another approach to delete is the "cop out" where we add a Boolean flag to denote if the node is considered to be deleted. The tree itself stays the same (no actual delete is performed) and the "isDeleted" flag can be used by other functions to maintain the data structure accordingly. The "cop out" has a negative connotation associated, but is actually quite useful in very complex situations. The pros and cons to this approach in general are listed on slide 73.

Slides 74 and 75 show how the update function can now use the insert and delete functions appropriately. With all this work, the question now is what value do we get with performance?

Slide 76 summarizes the performances of our 4 favorite functions. Ultimately, all perform at O(log n) because of the divide and conquer approach. Even if a delete has to find a successor node in a large subtree, the act of going down the left branch is

still eliminating a significant portion of the data with each traversal. Thus, we get to combine the effectiveness of binary search with an array at O(log n), as well as improve the performance of insert, update, and delete to O(log n)! Memory is also managed dynamically (versus static in arrays), so it seems like quite the no brainer to use.

However, the dependency is on the tree's balance. As data is maintained, the possibility of imbalance can lead to O(n) performance. "Cleanup" activities would need to be done to balance the tree which leads to the next question.

Is it possible to keep the tree balanced at all times? We will explore this next with 2-3-4 trees.

Slides 78-96

The remaining slides discuss another important aspect of binary trees and that is how to traverse them entirely. With sorted arrays, if we ever wanted to show all the elements in sorted order, it's a simple for loop. With a tree structure, it's not quite so easy and involves recursion. Walk through the slides here and practice with different tree examples to understand how all the tree traversals work (preorder, inorder, and postorder).

There is a healthy amount of material here so take your time with it. Practice and reach out to me if you have any questions. Enjoy!