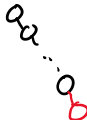


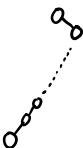


1 - Binary Search Trees

A - Properties of a Binary Search Tree

- Each node can have at most two children nodes
- The children of a nodes right subtree are all larger than the node – for any node
- The children of a nodes left subtree are all larger than the node – for any node

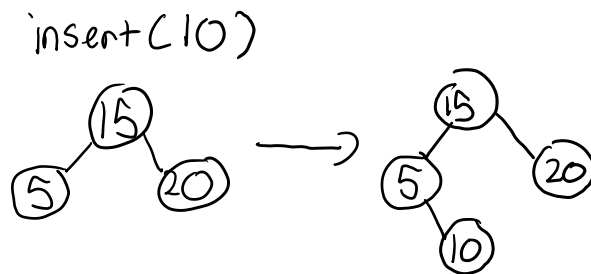
B -Time complexities:

Function	asymptotic worst-case n is nodes in tree	Analysis
Insert	$O(n)$	Finding the insertion point can take n iterations if the tree is like a linked list 
Delete	$O(n)$	Finding the node to delete can take up to n iterations id the tree is like a linked list 
Find-next	$O(n)$	 In its worst case you would have to traverse up n-1 nodes to discover that next node
Find-prev	$O(n)$	 In its worst case you would have to traverse up n-1 nodes to discover the prev node
Find-min	$O(n)$	Find min involves traversing the entire left subtree – if the BST just consists of left children, we must traverse all the nodes before reaching the minimum node.
Find-max	$O(n)$	Find max involves traversing the entire right subtree – if the BST just consists of right children, we must traverse all the nodes before reaching the maximum node.

C – Framework

Insert:

- i) This function is used to insert a new node (with a value) into the correct place in the BST. This function assumes that a BST is already initialized with a root node in the tree. I am also assuming that you can only insert values that are not already in the tree.
- ii) One edge case is if you are inserting a node that is already in the tree. Another edge case is if you do not have any nodes in the tree already.
- iii)



- iv) Alg: Start at the root node => inspectingNode

If the value you want to insert is larger than the inspectingNode

If there is no right node -> insert new node

If there is a right node -> right node => inspectingNode

Else

If there is no left node -> insert new node

If there is a right node-> left node => inspectingNode

Repeat this for every new value of inspectingNode until the node is inserted

- v) With this method we might have a big recursive stack if the tree is deep and we are doing it recursively. We are using constant space for this function which is good.
- vi) Code
- vii) If we are okay with using space we could keep a dictionary of all possible insertion points which searching through could be faster $O(\log(\text{insertion_points}))$ (in some instances).

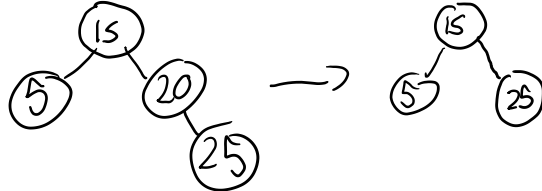
delete:

- i) This function takes a value and deletes it from the tree. I am assuming that you can only delete values that are already in the tree. And that the tree is initialized with a root node.
- ii) One edge case is removing a node not in the tree. There are also a few removal cases that are handled differently. For example if the node to remove has no children, one

child, or two children we must make sure that the tree is “reconnected” properly and it still maintains properties of a BST.

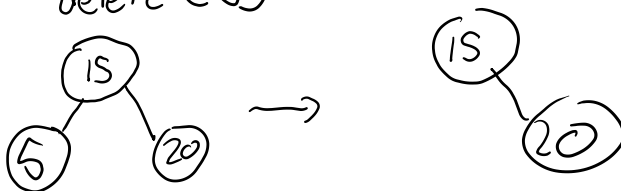
iii)

delete (20)



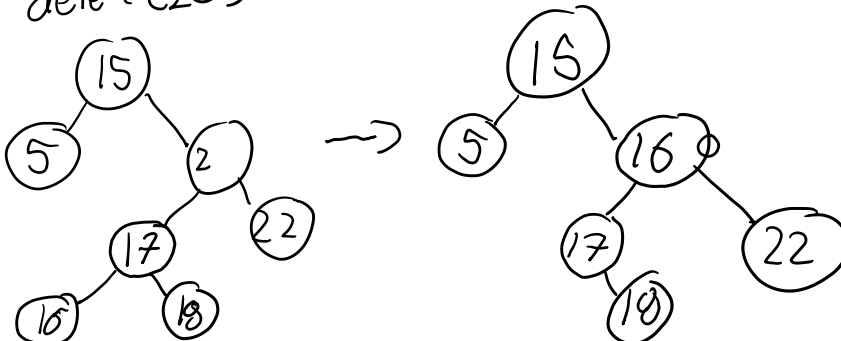
1 child

delete (5)



no
child

delete (20)



2 children

iv) Alg: traverse through the tree until you find the node to delete (recursively)

When you find the node you need to delete get its parent

Determine what case it is (no childer, 1 child, 2 children)

If its no child just set node to null

If there 1 child the parents next should point to node.next instead and make node null

If there are 2 children find the node to replace it and make sure that it assumes the correct left and right children that the previous node had.

v) One place to optimize is searching for the parent node. That takes a whole other traversal to do.

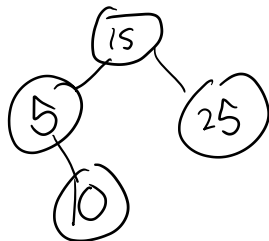
vi) Code

- vii) In this case if we wanted to trade off space we can really benefit a lot. If we kept an extra value that mapped what the parent node is we prevent ourselves from doing a traversal $O(\log n)$ to a constant time lookup $O(1)$.

Find-next:

- i) This function is used to find what the next node is given a value (what the next biggest node is). This function assumes that a BST is already initialized with a root node in the tree. It also assumes you are finding the next node of a node that already exists in the tree.
- ii) One edge case is if are looking for the next node of the largest node in the tree.
- iii)

Find-next(10) \Rightarrow 15



- iv) Alg: Starting at the node you want to find next of \Rightarrow inspectingNode

 If the node has a right child

 Next value is the min of the right subtree

 Else

 Get the parent node

 If the child is the parents right child

 Parent \Rightarrow inspectingNode

 Parent's parent \Rightarrow parent

 Else

 The parent's value is the next value to return

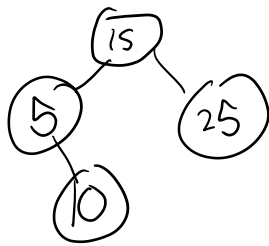
Repeat this for every new value of inspectingNode until the next node is found

- v) Finding the parent and the node from its value alone are expensive operations.
- vi) Code
- viii) In this case if we wanted to trade off space, we can really benefit a lot. If we kept an extra value that mapped what the parent node is we prevent ourselves from doing a traversal $O(\log n)$ to a constant time lookup $O(1)$.

Find-prev:

- i) This function is used to find what the prev node is given a value (what the next smallest node is). This function assumes that a BST is already initialized with a root node in the tree. It also assumes you are finding the next node of a node that already exists in the tree.
- ii) One edge case is if are looking for the prev node of the smallest node in the tree.
- iii)

Find-prev (10) \Rightarrow 5



- iv) Alg: Starting at the node you want to find next of \Rightarrow inspectingNode

If the node has a left child

prev value is the max of the left subtree

Else

Get the parent node

If the child is the parents left child

Parent \Rightarrow InspectingNode

Parent's parent \Rightarrow parent

Else

The parent's value is the next value to return

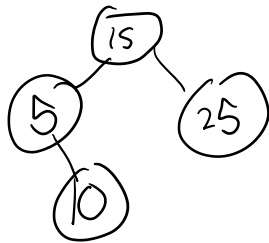
Repeat this for every new value of inspectingNode until the next node is found

- v) Finding the parent and the node memory location from their values alone are expensive operations.
- vi) Code
- ix) In this case if we wanted to trade off space, we can really benefit a lot. If we kept an extra value that mapped what the parent node is we prevent ourselves from doing a traversal $O(\log n)$ to a constant time lookup $O(1)$.

Find-max:

- i) This function is used to find what the maximum value of a sub-tree. This function assumes that a BST is already initialized with a root node in the tree.
- ii) There is not an extreme edge case for find max
- iii)

Find-max (Optional: 15) : 25



- iv) Alg: Starting at the root node of the sub-tree you

Continually

Check if node.right != null

Node = node.right

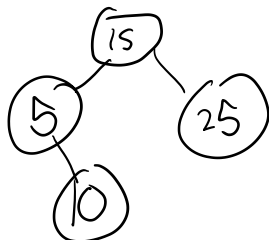
Return the value of the final node you are on

- v) This involves a traversal though the right most brach of the subtree so if this branch is long it will take more time
- vi) Code
- vii) There is not much of a trade off we can make unless we decided to store and update a maximum node value for the BST as we enter nodes into a tree.

Find-min:

- i) This function is used to find what the min value of a sub-tree. This function assumes that a BST is already initialized with a root node in the tree.
- ii) There is not an extreme edge case for find min
- iii)

Find-min (Optional: 15) \Rightarrow 5



- iv) Alg: Starting at the root node of the sub-tree

Continually

Check if node.left != null

Node = node.left

Return the value of the final node you are on

v) This involves a traversal though the left most branch of the subtree so if this branch is long it will take more time

vi) Code

vii) There is not much of a tradeoff we can make unless we decided to store and update a minimum node value for the BST as we enter nodes into a tree.

2) Sort It

a)

{ 5, 6, 7, 10, 11, 12, 16, 17, 18, 19, 20 }

b) starting with a list

enter all the values in the list into a BST

Do an in order recursive traversal of the tree

Insert the value of the left most nodes before inserting the nodes value followed by the right most nodes into a list

The resulting list is a sorted array