Chapter 3

Binary Search Tree

Binary search trees (BSTs) are very simple to understand. We start with a root node with value $x$, where the left subtree of $x$ contains nodes with values $< x$ and the right subtree contains nodes whose values are $\geq x$. Each node follows the same rules with respect to nodes in their left and right subtrees.

BSTs are of interest because they have operations which are favourably fast: insertion, look up, and deletion can all be done in $O(\log n)$ time. It is important to note that the $O(\log n)$ times for these operations can only be attained if the BST is reasonably balanced; for a tree data structure with self balancing properties see AVL tree defined in §7).

In the following examples you can assume, unless used as a parameter alias that root is a reference to the root node of the tree.

![Figure 3.1: Simple unbalanced binary search tree](image-url)
3.1 Insertion

As mentioned previously insertion is an $O(\log n)$ operation provided that the tree is moderately balanced.

1) algorithm Insert(value)
   2) Pre: value has passed custom type checks for type T
   3) Post: value has been placed in the correct location in the tree
   4) if root = ∅
      5) root ← node(value)
   6) else
      7) InsertNode(root, value)
   8) end if
   9) end Insert

1) algorithm InsertNode(current, value)
   2) Pre: current is the node to start from
   3) Post: value has been placed in the correct location in the tree
   4) if value < current.Value
      5) if current.Left = ∅
         6) current.Left ← node(value)
      7) else
         8) InsertNode(current.Left, value)
      9) end if
   10) else
      11) if current.Right = ∅
         12) current.Right ← node(value)
      13) else
         14) InsertNode(current.Right, value)
      15) end if
   16) end if
   17) end InsertNode

The insertion algorithm is split for a good reason. The first algorithm (non-recursive) checks a very core base case - whether or not the tree is empty. If the tree is empty then we simply create our root node and finish. In all other cases we invoke the recursive InsertNode algorithm which simply guides us to the first appropriate place in the tree to put value. Note that at each stage we perform a binary chop: we either choose to recurse into the left subtree or the right by comparing the new value with that of the current node. For any totally ordered type, no value can simultaneously satisfy the conditions to place it in both subtrees.
3.2 Searching

Searching a BST is even simpler than insertion. The pseudocode is self-explanatory but we will look briefly at the premise of the algorithm nonetheless.

We have talked previously about insertion, we go either left or right with the right subtree containing values that are $\geq x$ where $x$ is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

1. the root = $\emptyset$ in which case value is not in the BST; or
2. root.Value = value in which case value is in the BST; or
3. value < root.Value, we must inspect the left subtree of root for value; or
4. value > root.Value, we must inspect the right subtree of root for value.

1) algorithm Contains(root, value)
2) Pre: root is the root node of the tree, value is what we would like to locate
3) Post: value is either located or not
4) if root = $\emptyset$
5) return false
6) end if
7) if root.Value = value
8) return true
9) else if value < root.Value
10) return Contains(root.Left, value)
11) else
12) return Contains(root.Right, value)
13) end if
14) end Contains
### 3.3 Deletion

Removing a node from a BST is fairly straightforward, with four cases to consider:

1. the value to remove is a leaf node; or
2. the value to remove has a right subtree, but no left subtree; or
3. the value to remove has a left subtree, but no right subtree; or
4. the value to remove has both a left and right subtree in which case we promote the largest value in the left subtree.

There is also an implicit fifth case whereby the node to be removed is the only node in the tree. This case is already covered by the first, but should be noted as a possibility nonetheless.

Of course in a BST a value may occur more than once. In such a case the first occurrence of that value in the BST will be removed.

![Binary Search Tree Deletion Cases](image)

**Figure 3.2: binary search tree deletion cases**

The *Remove* algorithm given below relies on two further helper algorithms named *FindParent*, and *FindNode* which are described in §3.4 and §3.5 respectively.
1) **algorithm** Remove(value)  
2) **Pre:** value is the value of the node to remove, root is the root node of the BST  
3) Count is the number of items in the BST  
4) nodeToRemove ← FindNode(value)  
5) if nodeToRemove = ∅  
6) return false // value not in BST  
7) end if  
8) parent ← FindParent(value)  
9) if Count = 1  
10) root ← ∅ // we are removing the only node in the BST  
11) else if nodeToRemove.Left = ∅ and nodeToRemove.Right = null  
12) // case #1  
13) if nodeToRemove.Value < parent.Value  
14) parent.Left ← ∅  
15) else  
16) parent.Right ← ∅  
17) end if  
18) else if nodeToRemove.Left = ∅ and nodeToRemove.Right ≠ ∅  
19) // case # 2  
20) if nodeToRemove.Value < parent.Value  
21) parent.Left ← nodeToRemove.Right  
22) else  
23) parent.Right ← nodeToRemove.Right  
24) end if  
25) else if nodeToRemove.Left ≠ ∅ and nodeToRemove.Right = ∅  
26) // case #3  
27) if nodeToRemove.Value < parent.Value  
28) parent.Left ← nodeToRemove.Left  
29) else  
30) parent.Right ← nodeToRemove.Left  
31) end if  
32) else  
33) // case #4  
34) largestValue ← nodeToRemove.Left  
35) while largestValue.Right ≠ ∅  
36) // find the largest value in the left subtree of nodeToRemove  
37) largestValue ← largestValue.Right  
38) end while  
39) // set the parents' Right pointer of largestValue to ∅  
40) FindParent(largestValue.Value).Right ← ∅  
41) nodeToRemove.Value ← largestValue.Value  
42) end if  
43) Count ← Count −1  
44) return true  
45) end Remove
CHAPTER 3. BINARY SEARCH TREE

3.4 Finding the parent of a given node

The purpose of this algorithm is simple - to return a reference (or pointer) to the parent node of the one with the given value. We have found that such an algorithm is very useful, especially when performing extensive tree transformations.

1) algorithm FindParent(value, root)
2) Pre: value is the value of the node we want to find the parent of
3) root is the root node of the BST and is ! = ∅
4) Post: a reference to the parent node of value if found; otherwise ∅
5) if value = root.Value
6) return ∅
7) end if
8) if value < root.Value
9) if root.Left = ∅
10) return ∅
11) else if root.Left.Value = value
12) return root
13) else
14) return FindParent(value, root.Left)
15) end if
16) else
17) if root.Right = ∅
18) return ∅
19) else if root.Right.Value = value
20) return root
21) else
22) return FindParent(value, root.Right)
23) end if
24) end if
25) end FindParent

A special case in the above algorithm is when the specified value does not exist in the BST, in which case we return ∅. Callers to this algorithm must take account of this possibility unless they are already certain that a node with the specified value exists.

3.5 Attaining a reference to a node

This algorithm is very similar to §3.4, but instead of returning a reference to the parent of the node with the specified value, it returns a reference to the node itself. Again, ∅ is returned if the value isn’t found.
CHAPTER 3. BINARY SEARCH TREE

1) algorithm FindNode(root, value)
2) Pre: value is the value of the node we want to find the parent of
3) root is the root node of the BST
4) Post: a reference to the node of value if found; otherwise ∅
5) if root = ∅
6) return ∅
7) end if
8) if root.Value = value
9) return root
10) else if value < root.Value
11) return FindNode(root.Left, value)
12) else
13) return FindNode(root.Right, value)
14) end if
15) end FindNode

Astute readers will have noticed that the FindNode algorithm is exactly the same as the Contains algorithm (defined in §3.2) with the modification that we are returning a reference to a node not true or false. Given FindNode, the easiest way of implementing Contains is to call FindNode and compare the return value with ∅.

3.6 Finding the smallest and largest values in the binary search tree

To find the smallest value in a BST you simply traverse the nodes in the left subtree of the BST always going left upon each encounter with a node, terminating when you find a node with no left subtree. The opposite is the case when finding the largest value in the BST. Both algorithms are incredibly simple, and are listed simply for completeness.

The base case in both FindMin, and FindMax algorithms is when the Left (FindMin), or Right (FindMax) node references are ∅ in which case we have reached the last node.

1) algorithm FindMin(root)
2) Pre: root is the root node of the BST
3) root ≠ ∅
4) Post: the smallest value in the BST is located
5) if root.Left = ∅
6) return root.Value
7) end if
8) FindMin(root.Left)
9) end FindMin
3.7 Tree Traversals

There are various strategies which can be employed to traverse the items in a
tree; the choice of strategy depends on which node visitation order you require.
In this section we will touch on the traversals that DSA provides on all data
structures that derive from BinarySearchTree.

3.7.1 Preorder

When using the preorder algorithm, you visit the root first, then traverse the left
subtree and finally traverse the right subtree. An example of preorder traversal
is shown in Figure 3.3.

1) algorithm Preorder(root)
2) Pre: root is the root node of the BST
3) Post: the nodes in the BST have been visited in preorder
4) if root ≠ ∅
5) yield root.Value
6) Preorder(root.Left)
7) Preorder(root.Right)
8) end if
9) end Preorder

3.7.2 Postorder

This algorithm is very similar to that described in §3.7.1, however the value
of the node is yielded after traversing both subtrees. An example of postorder
traversal is shown in Figure 3.4.

1) algorithm Postorder(root)
2) Pre: root is the root node of the BST
3) Post: the nodes in the BST have been visited in postorder
4) if root ≠ ∅
5) Postorder(root.Left)
6) Postorder(root.Right)
7) yield root.Value
8) end if
9) end Postorder
Figure 3.3: Preorder visit binary search tree example
Figure 3.4: Postorder visit binary search tree example
3.7.3 Inorder

Another variation of the algorithms defined in §3.7.1 and §3.7.2 is that of inorder traversal where the value of the current node is yielded in between traversing the left subtree and the right subtree. An example of inorder traversal is shown in Figure 3.5.

![Inorder visit binary search tree example](image)

1) **algorithm** Inorder(root)
2) **Pre:** root is the root node of the BST
3) **Post:** the nodes in the BST have been visited in inorder
4) if root \(\neq \emptyset\)
5) Inorder(root.Left)
6) **yield** root.Value
7) Inorder(root.Right)
8) end if
9) **end** Inorder

One of the beauties of inorder traversal is that values are yielded in their comparison order. In other words, when traversing a populated BST with the inorder strategy, the yielded sequence would have property \(x_i \leq x_{i+1}\ \forall i\).
3.7.4 Breadth First

Traversing a tree in breadth first order yields the values of all nodes of a particular depth in the tree before any deeper ones. In other words, given a depth \( d \) we would visit the values of all nodes at \( d \) in a left to right fashion, then we would proceed to \( d + 1 \) and so on until we had no more nodes to visit. An example of breadth first traversal is shown in Figure 3.6.

Traditionally breadth first traversal is implemented using a list (vector, resizeable array, etc) to store the values of the nodes visited in breadth first order and then a queue to store those nodes that have yet to be visited.

![Figure 3.6: Breadth First visit binary search tree example](image-url)
1) **algorithm** BreadthFirst(root)
2) **Pre:** root is the root node of the BST
3) **Post:** the nodes in the BST have been visited in breadth first order
4) \( q \leftarrow \text{queue} \)
5) **while** root \( \neq \emptyset \)
6) \( \text{yield root.Value} \)
7) \( \text{if root.Left} \neq \emptyset \)
8) \( q.\text{Enqueue(root.Left)} \)
9) **end if**
10) \( \text{if root.Right} \neq \emptyset \)
11) \( q.\text{Enqueue(root.Right)} \)
12) **end if**
13) **if** \( \neg q.\text{IsEmpty()} \)
14) \( root \leftarrow q.\text{Dequeue()} \)
15) **else**
16) \( root \leftarrow \emptyset \)
17) **end if**
18) **end while**
19) **end** BreadthFirst

### 3.8 Summary

A binary search tree is a good solution when you need to represent types that are ordered according to some custom rules inherent to that type. With logarithmic insertion, lookup, and deletion it is very efficient. Traversal remains linear, but there are many ways in which you can visit the nodes of a tree. Trees are recursive data structures, so typically you will find that many algorithms that operate on a tree are recursive.

The run times presented in this chapter are based on a pretty big assumption - that the binary search tree’s left and right subtrees are reasonably balanced. We can only attain logarithmic run times for the algorithms presented earlier when this is true. A binary search tree does not enforce such a property, and the run times for these operations on a pathologically unbalanced tree become linear: such a tree is effectively just a linked list. Later in §7 we will examine an AVL tree that enforces self-balancing properties to help attain logarithmic run times.