CS 260
Binary Search Trees

Why Binary Search Trees

We have considered several ways of storing data: unsorted arrays, sorted arrays, and linked lists. They all have some advantages and disadvantages. The following table indicates the efficiencies of each of these data structures for adding new elements, deleting a given element and finding an element.

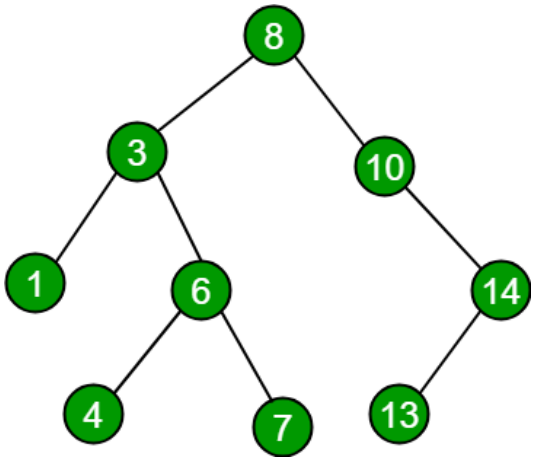
Data Structure	Add a new item	Find an item	Find and remove
Unsorted array	O(1)	O(N)	O(N)
Sorted Array	O(N)	O(log N)	O(N)
Linked List	O(1)	O(N)	O(N)

Because the height of a binary search tree is the log 2 of the number of elements, the time required for adding, finding, and removing are all O(log N) as we will see below.

Requirements

A binary search tree is a binary tree that is **strictly ordered.** This means that all elements in a subtree to the left of a given node have values less than the value of that node and all elements to the right of a given node have values greater than or equal to the value of the node. If the tree does not allow duplicates, then the right subtree has values less than the node.

An example of such a tree is shown in the following image.



In this tree the root has value 8 and its left child is 3 and its right child is 10. Note that 10 does not have a left child, but the right subtree of 10 has values greater than 10 as expected.

Finding an element

The algorithm is similar to the one that we use for a linked list. We start at the root and continue until we find the value or reach a nullptr (none). Instead of simply going to the next value, as in a linked list, we choose to go either right or left depending on a comparison of the value being searched for and the value in the node.

We can do this with an iterative search as shown in this algorithm:

```
bool find (value)
   ptr = root

   while ptr != nullptr

      if ptr.value == value
         return true

      if value < ptr.value
         ptr = ptr.left

      else
         ptr = ptr.right

   return false
```

The body of this algorithm is the while loop. It continues down the tree and terminates when we run out of nodes to consider. For each node encountered, first check for equality and return true since we have found the value being searched for. Then, if we do not have equality, we go to either the left or right subtree depending on the relationship between the value being searched for and the value in the node. Note that this algorithm will deal appropriately with tree that have duplicate values.

Alternately, we could write a recursive method that is passed in a value and a node (pointer) as below:

```
bool find (value)
   return recFind(value, root)

bool recFind(value, ptr)
   if ptr == nullptr
      return false

   if ptr.value == value
      return true

   if value < ptr.value
      return recFind(value, ptr.left)
   else
      return recFind(value, ptr.right)
```

As pointed out previously, this algorithm is O(log N) since there is at most one loop iteration or one recursive call per level in the tree and the tree height is log 2 of N.

Adding a new value

The rule for adding a new value is to always add at a leaf. So, we simply follow down the tree until we find a leaf and add either to its left or right, depending on the relation between the value being added and the value of the leaf.

Note that we have to special case adding a node to an empty tree. As in a linked list, we do not need to worry about dealing with a full data structure, since it grows as new items are added to it.

Here is the recursive version of adding a new value. We need to special case the empty tree, so we have a method that checks head before calling the recursive method. This allows the recursive method to assume that it is being passed a valid node instead of **nullptr/None/null**.

```
void addValue (value)

   if root == nullptr
      temp = new Node(value)
      root = temp
      return

   recAddValue(value, root)


void recAddValue (value, ptr)

   if value < ptr.value

      if ptr.left == nullptr
         temp = new Node(value)
         ptr.left = temp
      else
         recAddValue (value, ptr.left)

   else

      if ptr.right == nullptr
         temp = new Node(value)
         ptr.right = temp
      else
         recAddValue (value, ptr.right)
```

Once we have special cased adding to an empty tree (root == nullptr or none), we know that ptr has a value and can then call the recursive case. Creating a new node sets its left and right references to nullptr or none. We are not defining parent in this version, if we were, we could add a new line of code setting parent in the recursive function.

There is not an explicit base case for the recursive function, we know that each path down the tree ends with a leaf node, so when we find a nullptr (none), we are done and return. For each node encountered, we will either add the node as its appropriate child or we will continue down that subtree until we do reach a leaf

Removing a value

Removing a value from a binary search tree is more complex and is covered in a separate document.