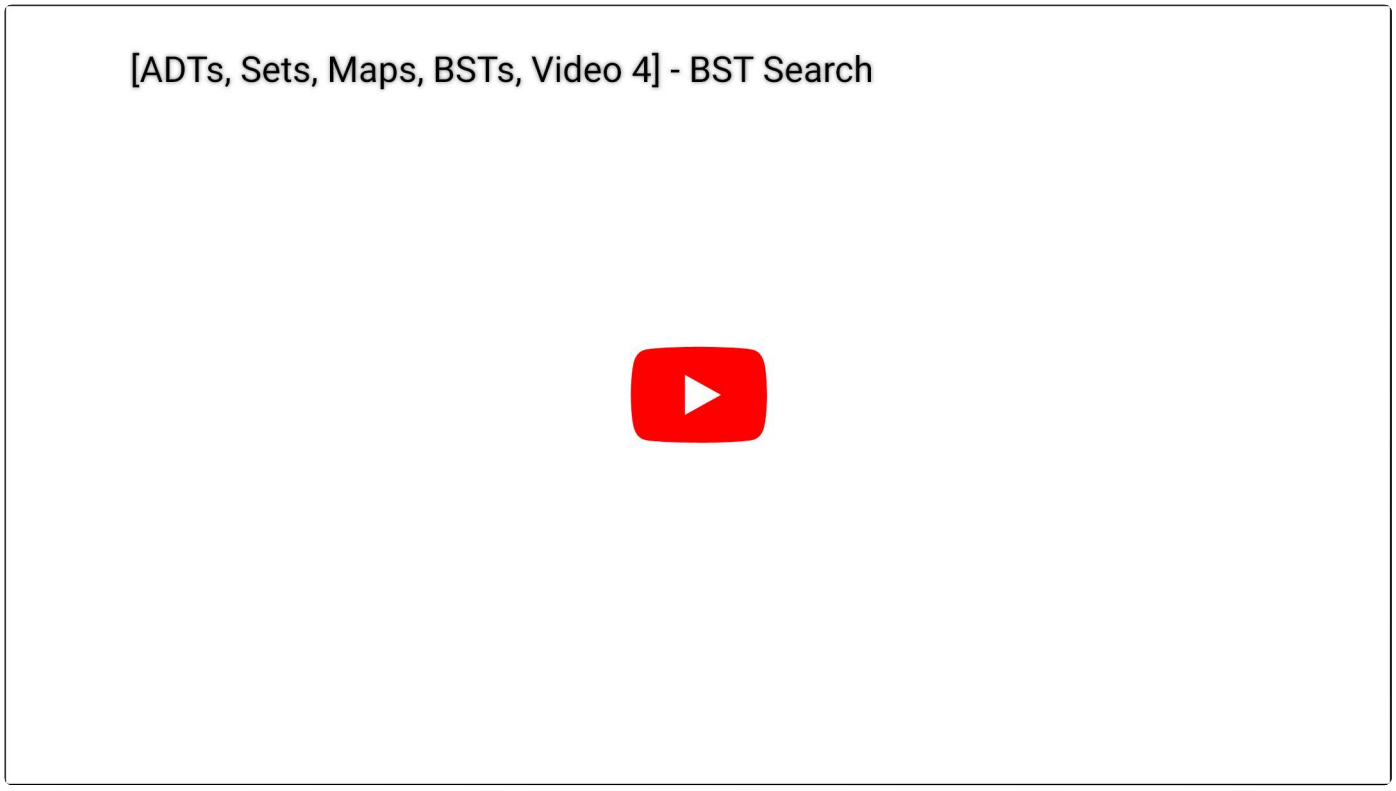
16.4 BST Operations



Search

To search for something, we employ binary search, which is made easy due to the BST property.

We know that the BST is structured such that all elements to the right of a node are greater and all elements to the left are smaller. Knowing this, we can start at the root node and compare it with the element, X, that we are looking for. If X is greater to the root, we move on to the root's right child. If its smaller, we move on to the root's left child. We repeat this process recursively until we either find the item or we get to a leaf, in which case the tree does not contain the item.

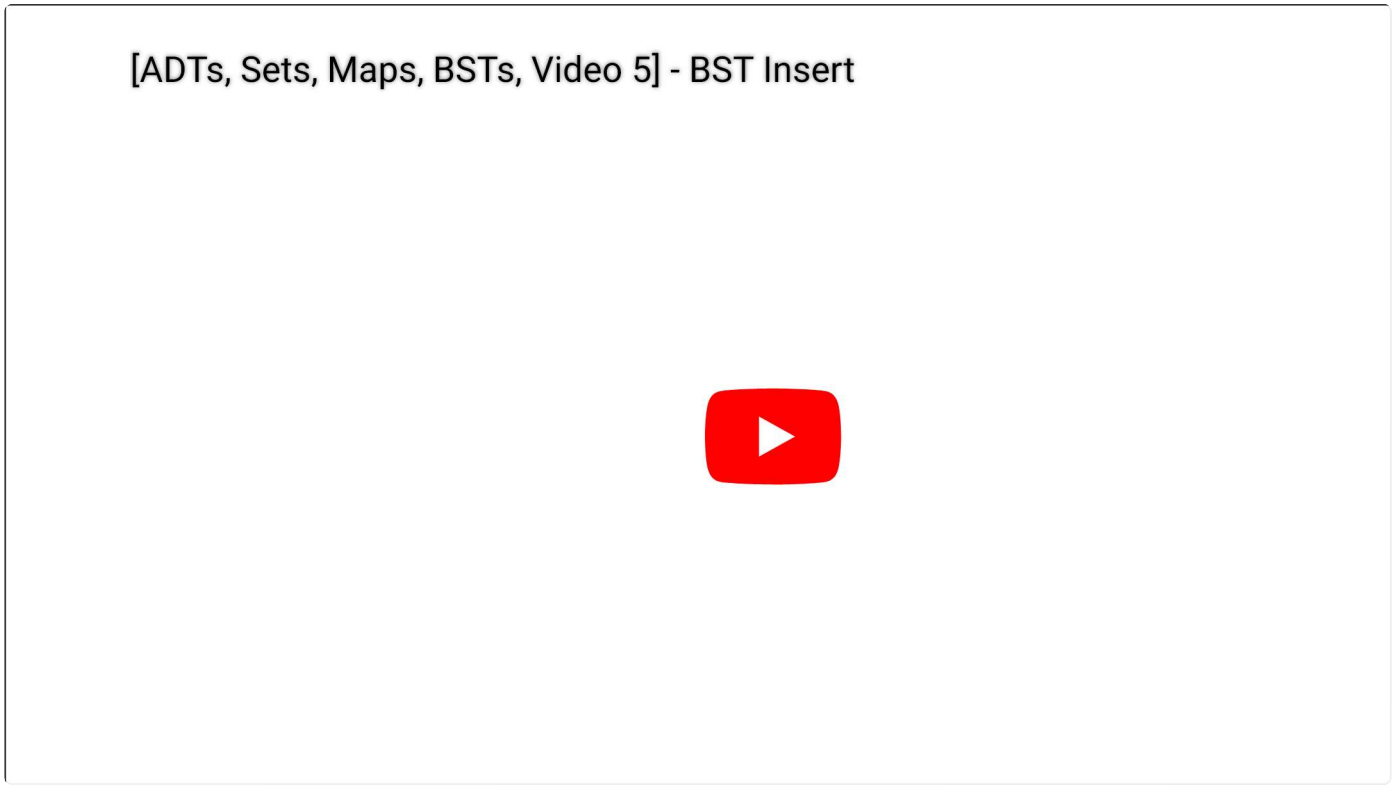
```
static BST find(BST T, Key sk) {
   if (T == null)
      return null;
   if (sk.equals(T.key))
      return T;
   else if (sk < T.key)
      return find(T.left, sk);
   else
      return find(T.right, sk);
}
```

If our tree is relatively "bushy", the find operation will run in $\log n$ time because the height of the tree is $\log n$.

Insert

We **always** insert at a leaf node!

First, we search in the tree for the node. If we find it, then we don't do anything. If we don't find it, we will be at a leaf node already. At this point, we can just add the new element to either the left or right of the leaf, preserving the BST property.



```
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik < T.key)
    T.left = insert(T.left, ik);
  else if (ik > T.key)
    T.right = insert(T.right, ik);
  return T;
}
```

Delete

Deleting from a binary tree is a little bit more complicated because whenever we delete, we need to make sure we reconstruct the tree and still maintain its BST property. Let's break this problem down into three categories:

- the node we are trying to delete has no children
- has 1 child
- has 2 children

Deletion: No Children

If the node has no children, it is a leaf, and we can just delete its parent pointer and the node will eventually be swept away by the garbage collector.

Deletion: One Child

If the node only has one child, we know that the child maintains the BST property with the parent of the node because the property is recursive to the right and left subtrees. Therefore, we can just reassign the parent's child pointer to the node's child and the node will eventually be garbage collected.

Deletion: Two Children

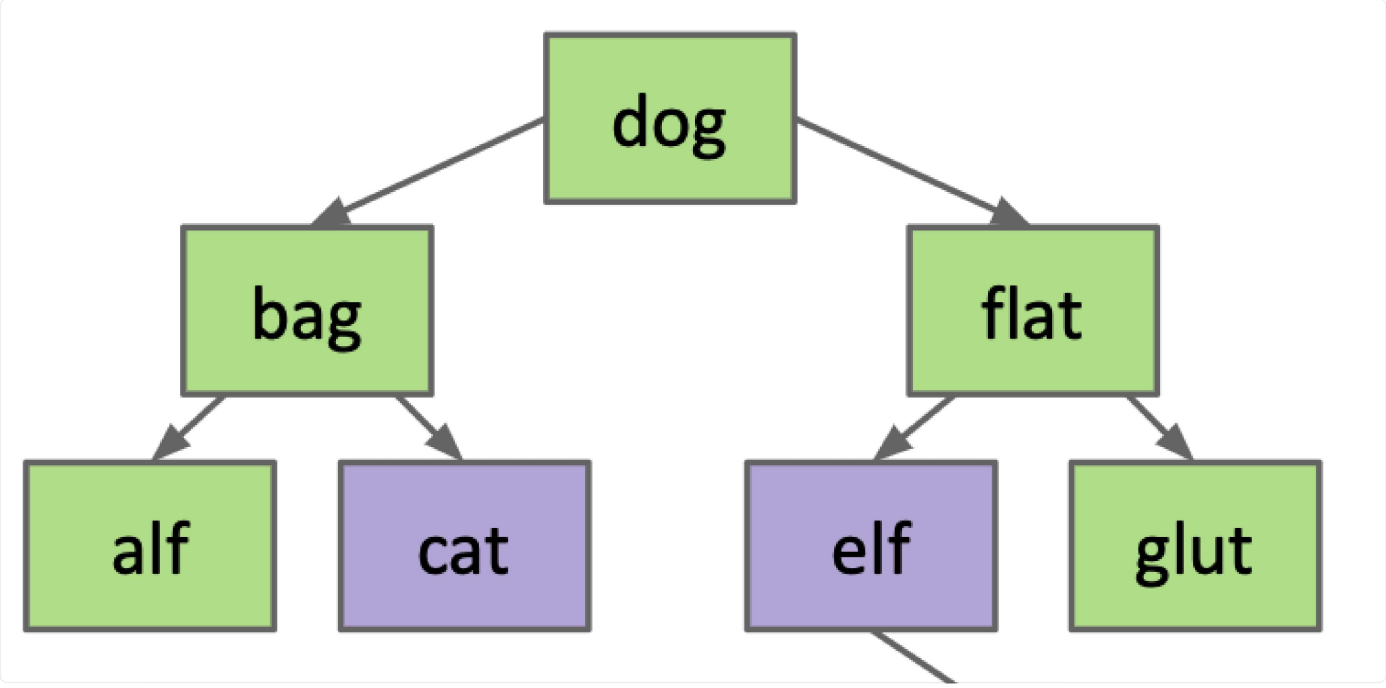
If the node has two children, the process becomes a little more complicated because we can't just assign one of the children to be the new root. This might break the BST property.

Instead, we choose a new node to replace the deleted one.

We know that the new node must:

- be > than everything in left subtree.
- be < than everything right subtree.

In the below tree, we show which nodes would satisfy these requirements given that we are trying to delete the `dog` node.



Possible candidates to replace `dog` after deletion

To find these nodes, you can just take the right-most node in the left subtree or the left-most node in the right subtree. Then, we replace the `dog` node with either `cat` or `elf` and then remove the old `cat` or `elf` node.

This is called **Hibbard deletion**, and it gloriously maintains the BST property amidst a deletion.

Previous
16.3 BST Definitions

Next
16.5 BSTs as Sets and Maps

Last updated 1 year ago