MS-CS Course Note (Non-Credit Course 3)

Mark Zhou

July 2024

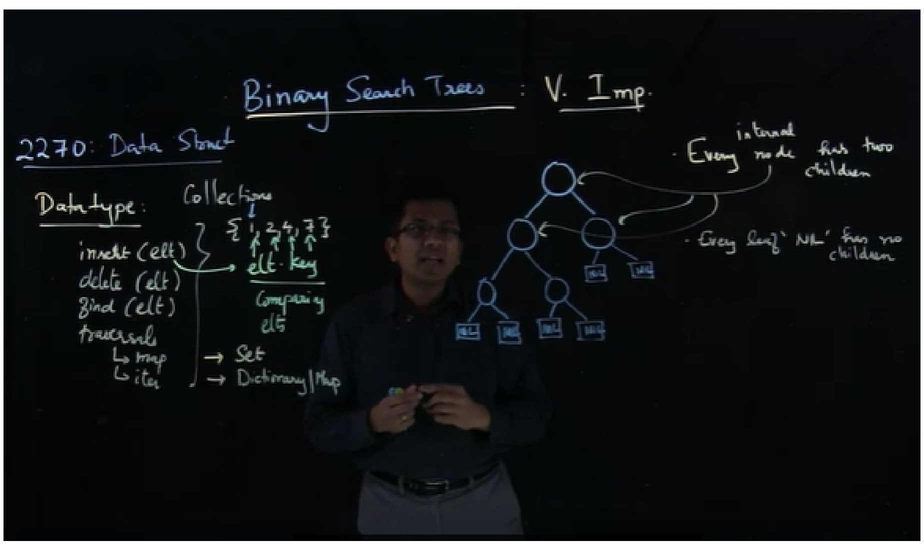
This is my course note on "Trees And Graphs: The Basics" provided by Colorado University of Boulder. This is a non-credit prep course for an MS-CS degree.

Contents

1 Binary Search Trees

1.1 Basic Concepts

Binary search tree is a binary tree is a kind of data type with set of data elements without repeatition.
We can insert, delete, search, and traverse the data elements in a binary search tree.
For each element in it, there will be a key of the element, which will always be a number.
With this setting in place, we can always comparing different elements by comparing their keys, even if the elements are not numbers.



In the figure, we have a binary search tree with some nodes and leaves. Every node has two children nodes and those leaves, which

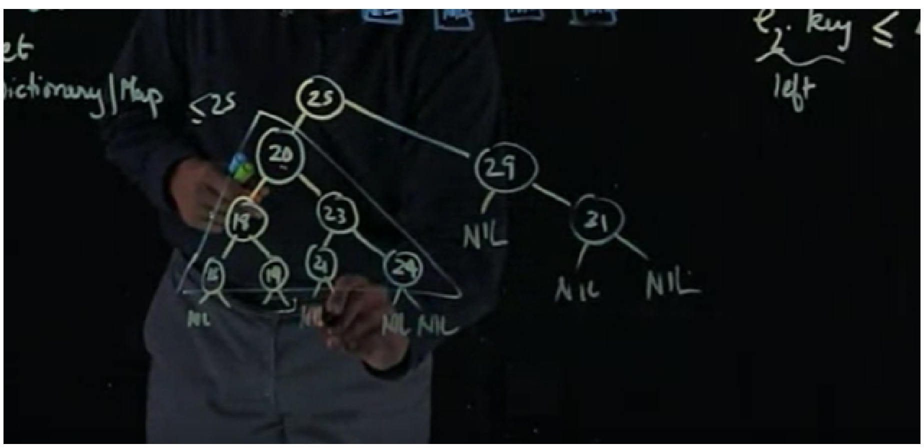
have no children nodes, are called nil nodes.

Every node has an element with a key, and the key of the left child node is always $<$ the key of the parent node, and the key of the right child node is always $>$ the key of the parent node.

The left and right child nodes are also binary search trees.

That is to say, the keys are always in a sorted order regardless of the structure of the tree. When we move the elements around, the keys will be different for each elements, in order to remain in the sorted order.

The leaves have no elements.



When there is a node with the key 25, every node in the left subtree will have a key $<$ 25, and every node in the right subtree will have a key $>$ 25.

The rule will also apply to all those subtrees.

```
Example:
 25
/  \
15  50
/ \  / \
10 22 35 70
```

```
Question:
Binary Search Trees may look similar to Heaps, but it is important
to consider their differences.
In a Min-Heap, the smallest element must be the root node of the tree.
In a Binary Search Tree, on the other hand, how would we find the
smallest element?

A: We would traverse the left subtree of the root node until we
reach a leaf node, which means a node with a NIL as its left child.
```

1.2 The Height of a Binary Search Tree

The height of a binary search tree is the number of edges on the longest path from the root node to a leaf node.
We will define the height of a leaf node as 0. Then the height of number 25, a.k.a. the root node of the below binary search tree is 2.

```
    Example:
     25       -> height = 2
    /  \
   15  50     -> height = 1
  / \  / \
 10 22 35 70  -> height = 0, since they are leaf nodes
```

Let's assume we have a balanced binary search tree with n internal nodes.

One each layer from the root node, there'll be 2^0 **nodes,** 2^1 **nodes,** 2^2 **nodes,** $\cdots$, 2^h **nodes.**

The total number of nodes in the tree will be $2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$**.**

So, the height of the tree will be $h = \log_2(n+1) - 1$**.**

In the sense of big O notation, the height of a binary search tree is $O(\log_2(n))$**, in a balanced binary search tree scenario.**

For example, $\log_2^8 = 3$**, so the height of a binary search tree with 8 nodes is 3.**

$\log_2^{15} = 4$**, so the height of a binary search tree with 15 nodes is 4.**

In the worst case scenario, where the binary search tree is not balanced, the height of the tree will be $O(n)$**.**

That is to say, the tree will be a linked list looks like this;

```
Linked List Example:
 10
  \
   15
    \
     20
      \
       25
        \
         30
          \
           35
```

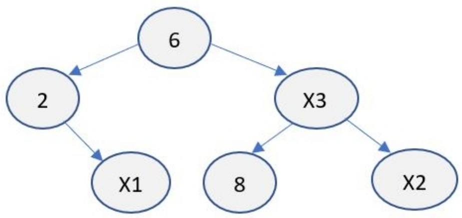
In this case, the height of the tree is 6, which is equal to the number of nodes in the tree.
The height of the tree is $O(n)$, which is the worst case scenario.
Normally, we will have something in between the best case scenario and the worst case scenario, $O(\log(n)) < height < O(n)$.

1.3 Basics of Binary Search Tree Quiz

1. Consider the following binary search tree below with missing values $X1$, $X2$ and $X3$.

1 / 1 point

Note that the leaves labeled NIL are not shown, but please assume that they exist.



Select all true statements about the tree.

- [ ] $X1$ can be any value less than or equal to 6.
- [x] $X1$ can be set to the number 5 while remaining a valid binary search tree.

Correct
$X1$ must also be ≥ 2 since it is the right child of 2, and $X1 \leq 6$ since it is in the left subtree of the root 6. Therefore, 5 is a possible value.

- [ ] $X3$ can be any number ≥ 6.
- [x] $X3$ can be any number ≥ 8 and $\leq X2$.

Correct
Correct

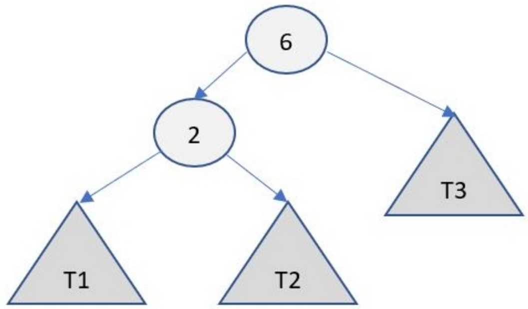
- [x] $X2$ must have a value ≥ 8 and $\geq X3$.

Correct
Correct

- [x] The height of the root node is 3.

Correct
Correct. Note that the root has a longest path of length 3 to a leaf.

2. Consider the following binary search tree with subtrees shown below. Select all true statements about it.



- [x] Every node in $T1$ must have value ≤ 2.

> **Correct**
> Correct since T1 is the left subtree of node 2.

- [x] Every node in $T2$ must have key ≥ 2 and ≤ 6.

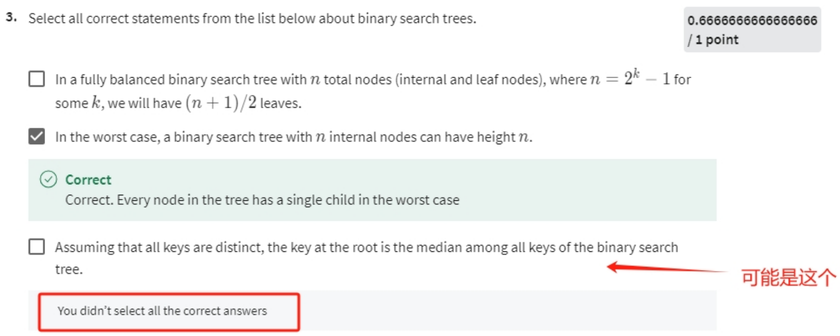
> **Correct**
> Correct since T1 is in the right subtree of node 2 and left subtree of the node 6.

- [ ] If the node with key 25 is found in the tree, we will find it in subtree $T2$.
- [ ] If the node with key -10 is to be found in the tree, it can be found in subtree $T2$.
- [x] If the node with key 7 is to be found in the tree, it will be found in $T3$.

> **Correct**
> Correct since 7 > 6 it will be found in the right subtree of the root node 6.

- [x] If the height of subtree $T1$ is 4 and that of subtree $T2$ is 2 then the height of node labeled 2 is 5.

> **Correct**
> Correct since max(4, 2) + 1 = 5



3. Select all correct statements from the list below about binary search trees.

0.6666666666666666 / 1 point

- [ ] In a fully balanced binary search tree with n total nodes (internal and leaf nodes), where $n = 2^k - 1$ for some k, we will have $(n+1)/2$ leaves.
- [x] In the worst case, a binary search tree with n internal nodes can have height n.

Correct
Correct. Every node in the tree has a single child in the worst case

- [ ] Assuming that all keys are distinct, the key at the root is the median among all keys of the binary search tree.

可能是这个

You didn't select all the correct answers

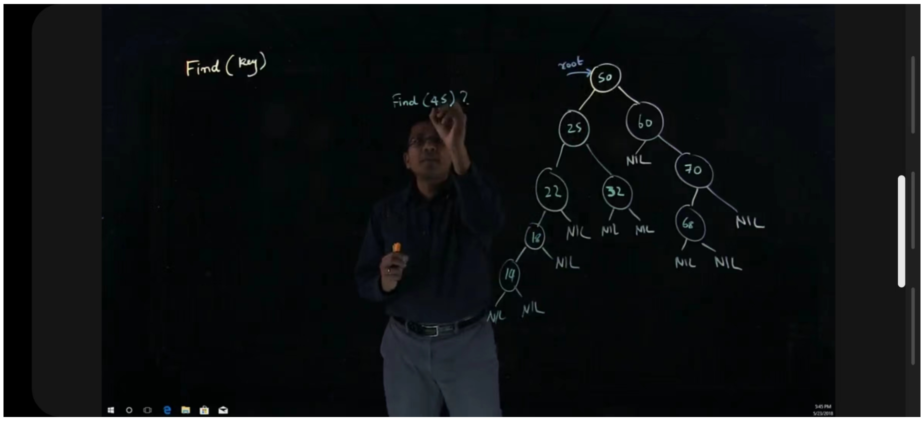
1.4 Insertion and Deletion in a Binary Search Tree

1.4.1 Insertion

We can insert a new element into a binary search tree by comparing the key of the new element with the key of the root node.
If the key of the new element is less than the key of the root node, we will insert the new element into the left subtree.
If the key of the new element is greater than the key of the root node, we will insert the new element into the right subtree.
We will repeat the process until we reach a leaf node.
For example, we have a binary search tree with the following nodes;

```
    Example:
     25
    /  \
   15  50
  / \  / \
 10 22 35 70
```

If we want to insert a new element with the key 40, we will compare 40 with 25, and then 40 with 50, and then 40 with 35.
Since 40 is greater than 35, we will insert 40 as the right child node of 35.



The above example actually consists of two steps.
First, we will search for the element to be inserted, which is called the $find()$ operation.
Then, after successfully locates the element, we will insert it into the binary search tree.
We will talk about find operation now.
Assuming we have an impefect binary search tree and we want to locate the key 45, how should we do that?

We will start from the root node, and then compare the key of the root node with the key of the element to be located.

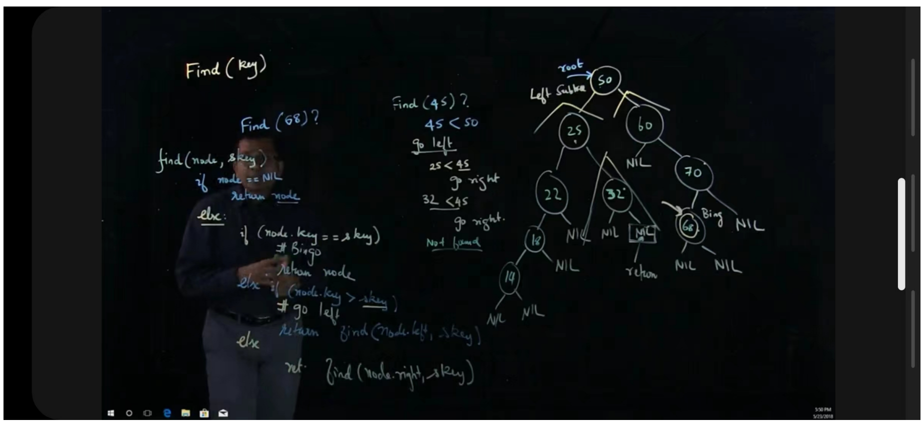
If the key of the root node is equal to the key of the element to be located, we will return the root node. If the key of the root node is greater than the key of the element to be located, we will search the left subtree. Otherwise, we will search the right subtree.
The overall process will be repeated until we reach a leaf node.
If we reach a leaf node and the key of the leaf node is not equal to the key of the element to be located, we will return NIL.

```
Here is the pseudo code for the find operation:
find(root, key)
    if root == NIL or root.key == key
        return root
    if root.key > key
        return find(root.left, key) # This is the recursive call.
    return find(root.right, key) # This is the recursive call.
```

As for the time complexity, the find operation will take $O(h)$ time, where h is the height of the binary search tree.
Now we will go ahead with the second step of the insertion operation. We will insert the new element into the binary search tree by comparing the key of the new element with the key of the root node.

```
Here is the pseudo code for the insertion operation:
insert(root, key)
    if root == NIL
        return new Node(key)
    if key < root.key
```

```
    root.left = insert(root.left, key) # This is the recursive call.
else
    root.right = insert(root.right, key) # This is the recursive call.
return root
```



1.4.2 Deletion

Now we will talk about the deletion operation.

There are three cases to consider when deleting a node from a binary search tree.

The node to be deleted can be a leaf node, a node with only one child,or one with two child nodes.

If both child nodes are NIL, we can simply delete the node.

If one of the child nodes is NIL, we can delete the node, then reconnect between the past-parrent node and past-child node, like this;

```
    Only One Child Node Example:

     25                      25

    /  \                    /  \

   15  50   delete 15     /   50

  / \       --------->   / \

 10 NIL                  10  NIL

 /\                     / \

NIL NIL                NIL NIL
```

What if we want to delete a node that has two non-NIL children?
We will find the smallest node in the right subtree of the node to be deleted, and then replace the node to be deleted with the smallest node.
To perform this operation, we will 'walk' right from the root node by one step and then 'walk' left until we reach a leaf node.
During each step, we will compare the key of the current node with the key of the node to be deleted.
The leaf node we reached will be the successor of the node we deleted.

```
    Two Child Nodes Example:

     25                      25

    /  \                    /  \

   15  50   delete 15     22   50

  / \       --------->   / \

 10 22                  10  NIL

 /\                     / \

NIL NIL                NIL NIL
```

```
Here is the pseudo code for the deletion operation:
delete(root, key)
    if root == NIL
        return root
    if key < root.key
        root.left = delete(root.left, key) # This is the recursive call.
    else if key > root.key
        root.right = delete(root.right, key) # This is the recursive call.
    else
        if root.left == NIL
            return root.right
        else if root.right == NIL
            return root.left
        root.key = minValue(root.right)
        root.right = delete(root.right, root.key) # This is the recursive call.
    return root
```

1.4.3 Tree Traversal

There are three ways to traverse a binary search tree.
In-order traversal, pre-order traversal, and post-order traversal.
In-order traversal will visit the left subtree, then the root node, and then the right subtree.
Pre-order traversal will visit the root node, then the left subtree, and then the right subtree.
Post-order traversal will visit the left subtree, then the right subtree, and then the root node.

TBC: Binary search tree: insertion and deletion video: preorder traversal