Lab 7

Binary Search Trees

CS61B Fall 2023

Announcements

Homework 2 is due Wednesday, 10/4 at 11:59 pm

Project 2A has been released and is due Wednesday, 10/11 at 11:59 pm

Binary Search

Binary Search

We'll see in a minute what binary search trees look like and what they are - but let's first talk about what binary search is.

Binary Search

We'll see in a minute what binary search trees look like and what they are - but let's first talk about what binary search is.

Given a sorted array, we can find the position of a value through divide and conquer - that is, we can repeatedly halve the values we need to go through until we find the value that we want.

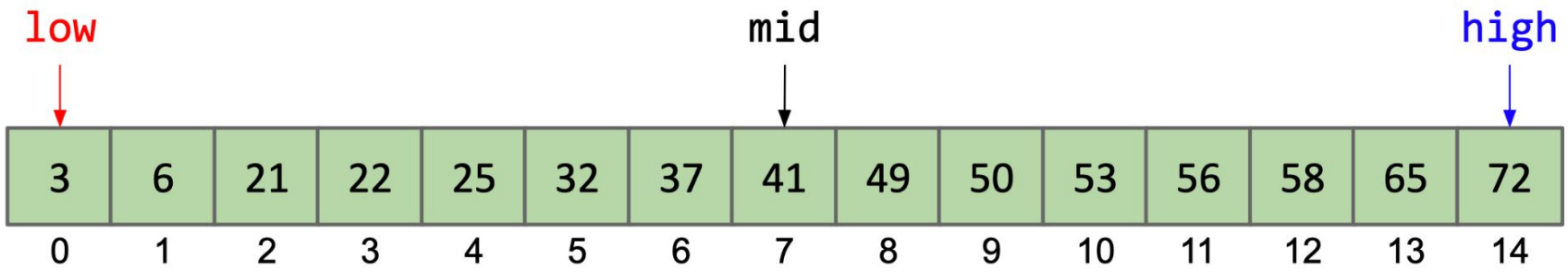
Binary Search

We'll see in a minute what binary search trees look like and what they are - but let's first talk about what binary search is.

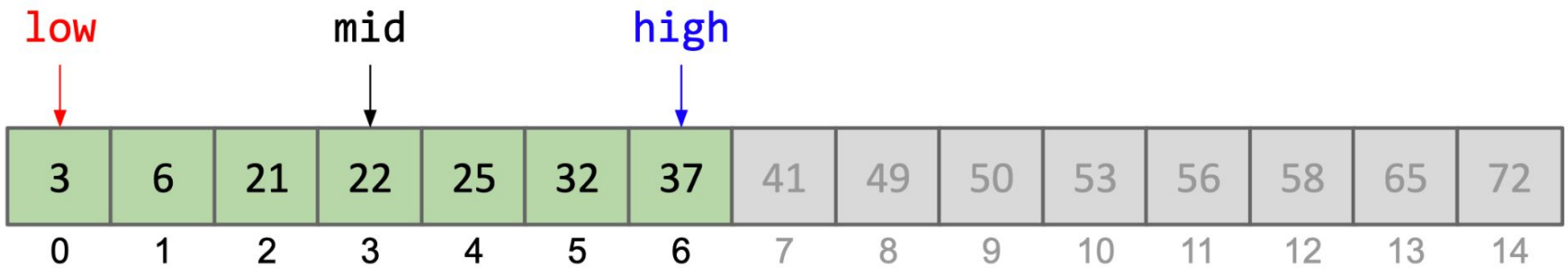
Given a sorted array, we can find the position of a value through divide and conquer - that is, we can repeatedly halve the values we need to go through until we find the value that we want.

- Typically speaking, we have several variables that we keep track of when we run binary search:
  - `low`, `mid`, `high`, and `k` (the value that we're looking for)

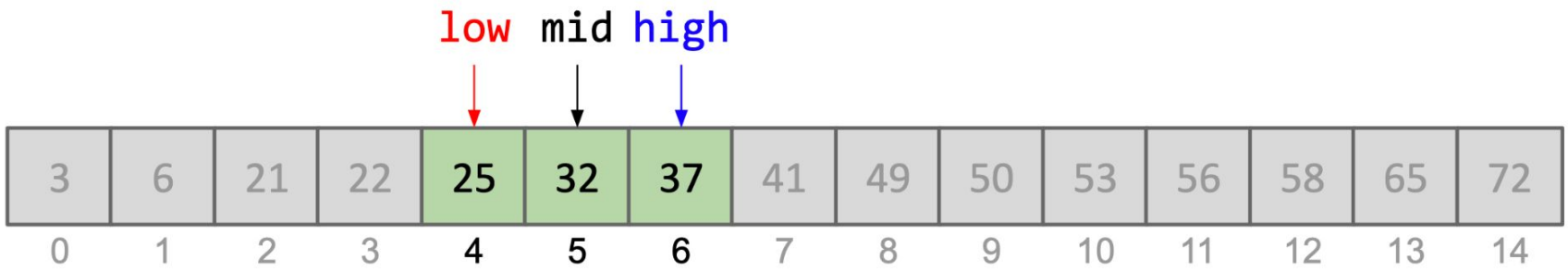
```
k = 25
low = 0
mid = 7
high = 14
```



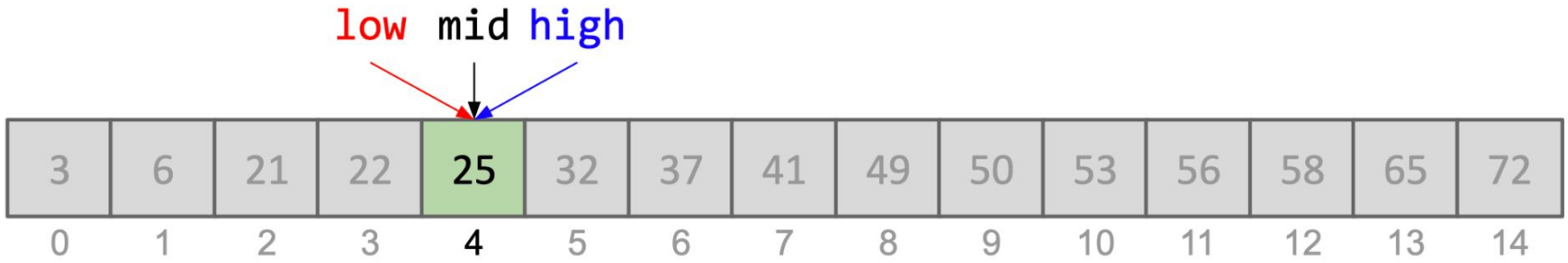
```
k = 25
low = 0
mid = 3
high = 6
```



```
k = 25
low = 4
mid = 5
high = 6
```



```
k = 25
low = 4
mid = 4
high = 4
```



Binary Search

At this point, we'll start to notice a pattern - whenever we look for our specific value of `k`, we always halve the number of values that we need to look for in each iteration.

Binary Search

At this point, we'll start to notice a pattern - whenever we look for our specific value of `k`, we always halve the number of values that we need to look for in each iteration.

So, why does that matter?

- At each iteration/step, if we keep halving roughly half the elements, the worst case for our runtime ends up being **log(N)** (keep in mind that it is in terms of log base 2 in computer science).

Binary Search Trees

Binary Search Trees

Given a sorted array, we're able to achieve a runtime of log(N) when searching for a specific element in the array.

- Note that we're able to achieve such a runtime under the assumption that we already have a sorted array. This is where **binary search trees** come in!

Binary Search Trees

Given a sorted array, we're able to achieve a runtime of log(N) when searching for a specific element in the array.

- Note that we're able to achieve such a runtime under the assumption that we already have a sorted array. This is where **binary search trees** come in!

With binary search trees, we can essentially take advantage of a tree structure while adding in some additional conditions to maintain the elements in a "sorted" order.

- By adding in additional constraints, we can determine how elements are placed into our tree structure, allowing us to effectively make it much faster to insert, lookup and remove items.

Binary Search Tree Def.

As the name implies, binary search trees are just binary trees, so the properties of a binary tree still applies (each node has at most 2 children).

Binary Search Tree Def.

As the name implies, binary search trees are just binary trees, so the properties of a binary tree still applies (each node has at most 2 children).

The specific BST properties are as the following:

- Assume that we are at node X
  - Every value in X's left subtree is smaller than X
  - Every value in X's right subtree is greater than X

Insertion

To maintain the properties of the BST, let's consider how we insert an element T (assume no duplicates)

Insertion

To maintain the properties of the BST, let's consider how we insert an element T (assume no duplicates)

- We compare the value of T to the value in the current node (starting at the root)
  - If T is smaller than the current node's value, we go to the left subtree
  - If T is larger than the current node's value, we go the right subtree
  - We repeat this until we can insert T as a leaf node.
- Note that the left and right subtrees are both BSTs themselves!

Insertion Demo

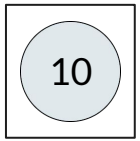
BST Insertion

Starting out, let's say we've inserted 14 into our binary search tree. So 14 is the root of our tree.

BST Insertion

Now let's insert a number into the binary search tree. We'll go with 10.

At this point, we need to figure how where 10 goes in the binary search tree. To do so, we want to make a comparison between 10 and 14.

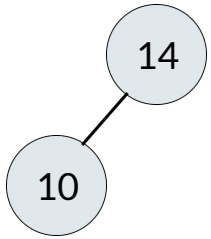


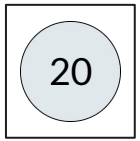
Next to insert.

BST Insertion

Since 10 is less than 14, we go to the left. Since 14 has no children so far, 10 can be inserted as leaf node to the left.

Now let's insert 20 into our tree and repeat the same process.



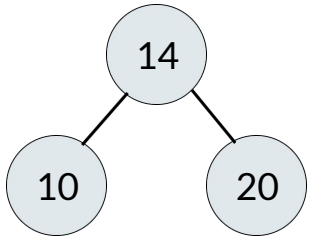


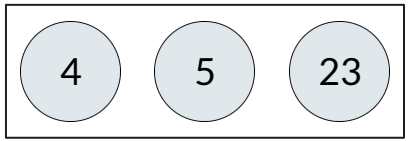
Next to insert.

BST Insertion

Since 20 is greater than 14, it will be inserted into the tree to the right of 14.

What will our tree look like if we inserted these numbers in this order: **4, 5, 23**?





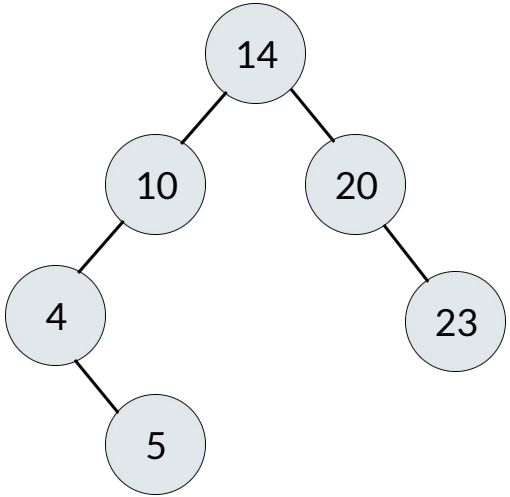
Next to insert.

BST Insertion

It'll look something like this!

When inserting an item, we make comparisons with the current node we're on to determine which way we traverse down the binary search tree.

Once we hit a leaf node, we place our item there.



Quick Tip

Recursion is your best friend!

- Remember, the left or right subtree of any node is also a BST!
    - When we're traversing through the tree (top-down), we know that the value is going to either be smaller or larger than the current value we are currently on. Based on this, we can traverse either the left subtree or right subtree (hint!).

Quick Tip

Recursion is your best friend!

- Remember, the left or right subtree of any node is also a BST!
    - When we're traversing through the tree (top-down), we know that the value is going to either be smaller or larger than the current value we are currently on. Based on this, we can traverse either the left subtree or right subtree (hint!).

Avoid "arms-length" recursion!

- Don't stop when the base case is one step away from the actual base case
- i.e., `T.left == null` instead of `T == null`

One Last Important Thing

How does the order of elements inserted into the BST affect the runtime of it (think about the structure that can be created)?

- Are we always guaranteed to have a worst case runtime of log(N) if we're looking up an element in the BST?

One Last Important Thing

How does the order of elements inserted into the BST affect the runtime of it (think about the structure that can be created)?

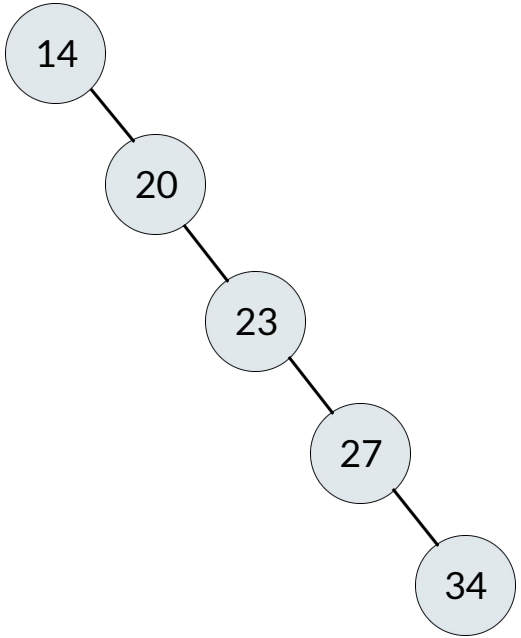
- Are we always guaranteed to have a worst case runtime of log(N) if we're looking up an element in the BST?

Nope! The insertion of elements can affect our worst case runtime for lookup (i.e. a bushy tree structure is O(log(N)) while a spindly tree is O(N)).

One Last Important Thing

Suppose we had the following spindly tree:

What would be the runtime of `containsKey(34)` be?

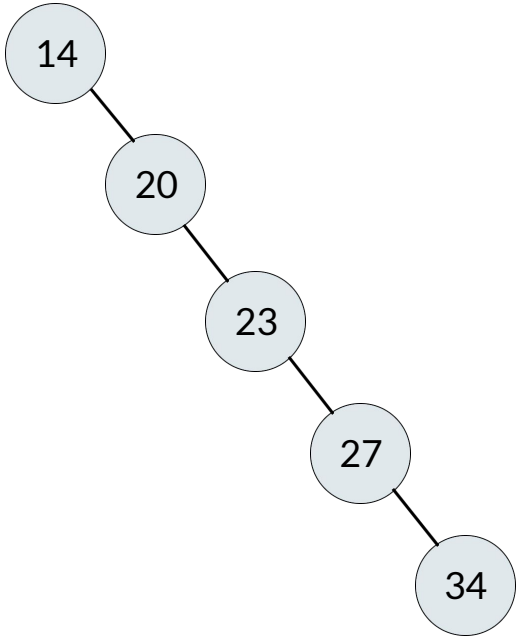


One Last Important Thing

Suppose we had the following spindly tree:

What would be the runtime of `containsKey(34)` be?

Worst case, O(N). Why?



Lab Overview

An Overview

Lab 07 is due Friday, 10/6 at 11:59 pm.

Deliverables:

- Complete your implementation of **BSTMap** and ensure that it implements the interface Map61B.
- Make sure to fill out **speedTestResults.txt** sufficiently, noting your observations down.
- Some tips:
  - We highly recommend you have some helper methods - they'll reduce the amount of code you need to write.
  - Think recursively!

For help, use the Lab queue: [INSERT YOUR LAB QUEUE HERE]

Lab Notes

Lab Notes

`printInorder()`: Here's a reminder of the pseudocode for in-order traversal of a binary tree

```
public void inOrder(Node node) {
    if (node == null) {
        return;
    }
    inOrder(node.left);
    processNode(node);
    inOrder(node.right);
}
```


Lab Notes

Another thing to note in this lab is that in your implementation, you should ensure that generic keys **K** in **BSTMap<K, V>** extends **Comparable**.

This is to make sure that you're able to use the compareTo method in your implementation, specifically with K (why might that be useful?).

```
public class BSTSet<K extends Comparable<K>>
implements Set61B<K> {...}
```

In this lab, your generic key K needs to implement Comparable. The above shows an example of what that may look like - the main difference being that your BSTMap takes <K, V> instead.