CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 25: Trees (and Bags and Sets)

Announcements

- PA2 due on Sunday

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each item)

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each ~~item~~ key)

The Set ADT

`void add(T element)`

Store one copy of **`element`** if not already present

`boolean contains(T element)`

Return true if **`element`** is present in the set

`boolean remove(T element)`

Remove **`element`** if present, or return false if not

Bags

A **Bag** is an **unordered** collection of **non-unique** elements.

(order doesn't matter, and multiple copies with the same key is OK)

The Bag ADT

`void add(T element)`

Store one copy of **`element`**

`int contains(T element)`

Return the number of copies of **`element`** in the bag

`boolean remove(T element)`

Remove one copy of **`element`** if present, or return false if not

Note: Sometimes referred to as multiset. Java does not have a native Bag/Multiset class.

Collection ADTs

Property	Seq	Set	Bag
Explicit Order	✓		
Enforced Uniqueness		✓	
Iterable	✓	✓	✓

Implementing Sets/Bags

How could we implement Sets/Bags with what we already know?

- ArrayLists?
- LinkedLists?
- Our SortedList from PA1?
- Other options?

What would the complexity of the important operations look like?

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$		
LinkedList	$O(n)$		
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$

Adding can be $O(1)$ for Bags since we don't need to check if the item already exists

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	The reason add and remove are $O(n)$ for the sorted ArrayList is because even though we can find the element in $O(\log(n))$, we have to shift elements when we change the ArrayList		
LinkedList			
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$

(Rooted) Trees

(Even More) Tree Terminology

Rooted, Directed Tree - Has a single root node (node with no parents)

Parent of node X - A node with an out-edge to X (max 1 parent per node)

Child of node X - A node with an in-edge from X

Leaf - A node with no children

Depth of node X - The number of edges in the path from the root to X

Height of node X - The number of edges in the path from X to the deepest leaf

(Even More) Tree Terminology

Level of a node - Depth of the node + 1

Size of a tree (*n*) - The number of nodes in the tree

Height/Depth of a tree (*d*) - Height of the root/depth of the deepest leaf

(Even More) Tree Terminology

Binary Tree - Every vertex has at most 2 children

Complete Binary Tree - All leaves are in the deepest two levels

Full Binary Tree - All leaves are at the deepest level, therefore every vertex has exactly 0 or 2 children, and $d = \log(n)$

Tree Implementation with `Optional`

```
1  public class Tree<T> {
2    Optional<TreeNode<T>> root;
3  }
4
5  public class TreeNode<T> {
6    T value;
7    Optional<TreeNode<T>> leftChild;
8    Optional<TreeNode<T>> rightChild;
9  }
```

Trees are inherently recursive data structures…

So most of the methods we are about to talk about will be very naturally expressed with recursion

Computing Tree Height

The height of a tree is the height of the root

Computing Tree Height

The height of a tree is the height of the root

The children of the root are each roots of the left and right subtrees

Computing Tree Height

The height of a tree is the height of the root

The children of the root are each roots of the left and right subtrees

So we can compute height recursively:

$$h(root) = \begin{cases} 0 & \textbf{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \textbf{otherwise} \end{cases}$$

Computing Tree Height

```
1 public int height(Optional<TreeNode<T>> root) {
2   if (!root.isPresent()) return 0;
3   return Math.max(height(root.leftChild), height(root.rightChild)) + 1;
4 }
```

$$h(root) = \begin{cases} 0 & \textbf{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \textbf{otherwise} \end{cases}$$

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node X_L in the left subtree of node X: X_L**.key** < X**.key**

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node X_L in the left subtree of node X: X_L**.key** < X**.key**
- For every node X_R in the right subtree of node X: X_R**.key** > X**.key**

Binary Search Tree

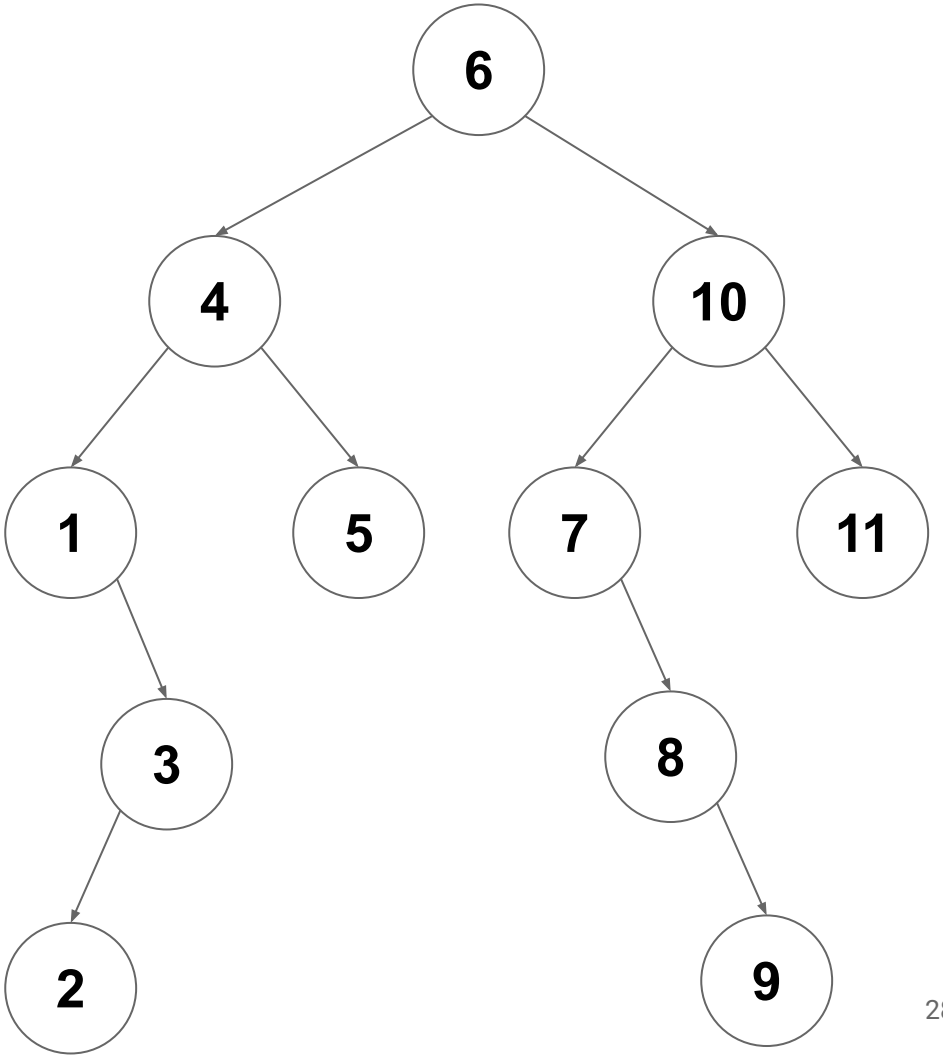
A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node X_L in the left subtree of node X: X_L**.key** < X**.key**
- For every node X_R in the right subtree of node X: X_R**.key** > X**.key**

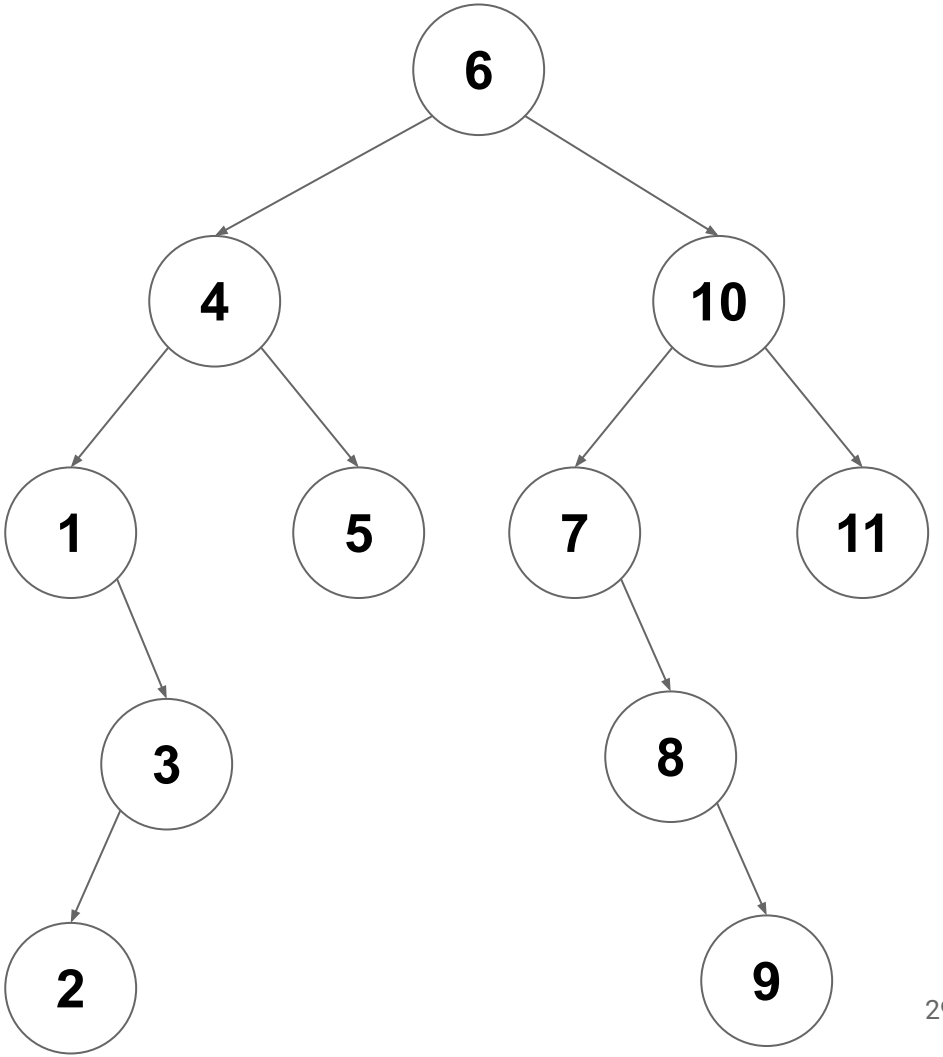
***X* partitions** its children

Is this a valid BST?



Is this a valid BST?

Yes!

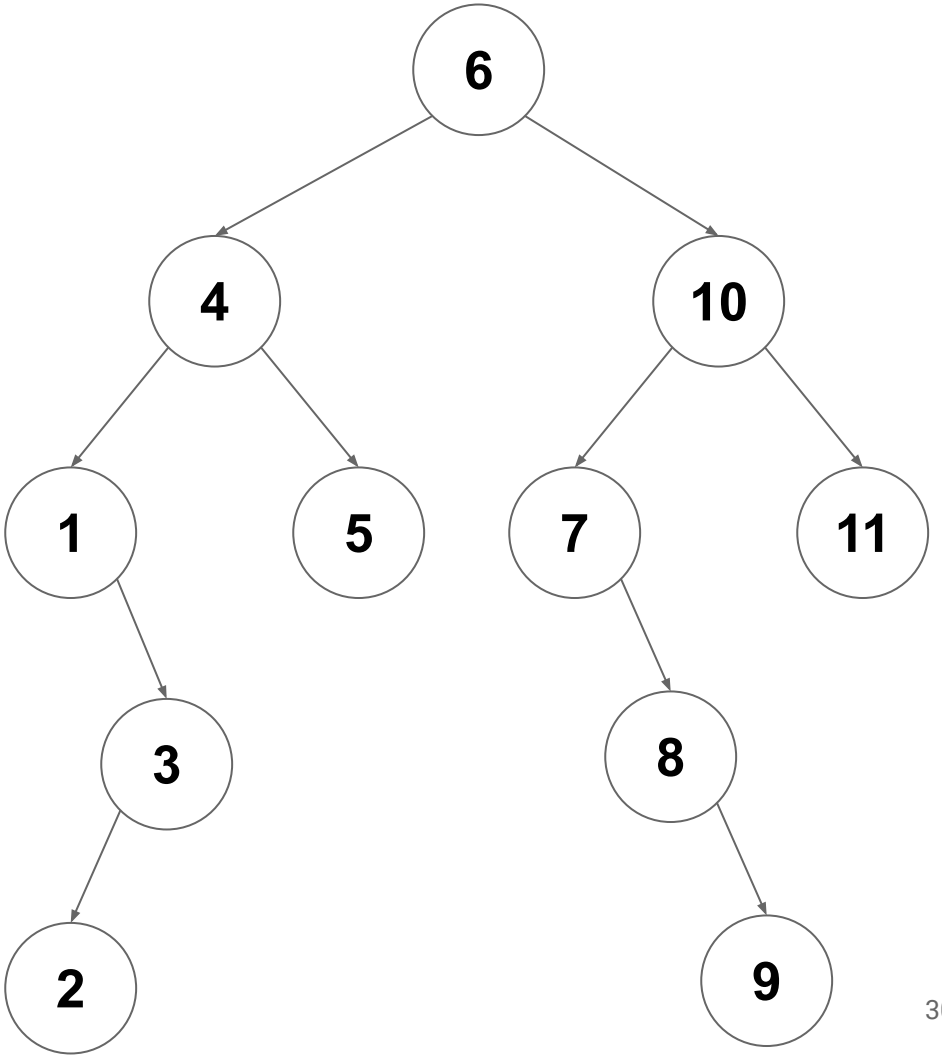


Is this a valid BST?

Yes!

Everything in the left subtree is < 6

Everything in the right subtree is > 6

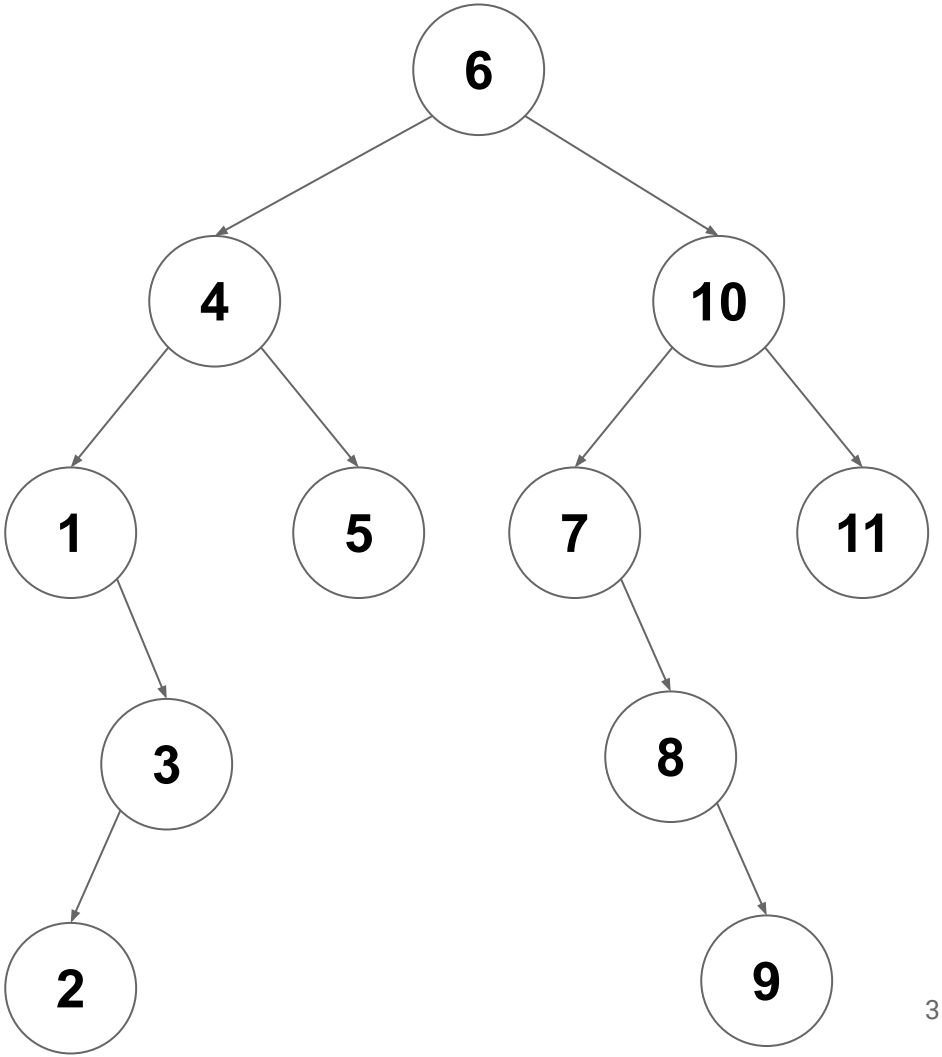


Is this a valid BST?

Yes!

AND:

The left subtree is a BST
The right subtree is a BST



Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (If yes, search the right subtree)

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

If the current node we are looking at is empty, then the value we are looking for is not in the tree

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

Otherwise, search the BST whose root is the left child if the target is less than our current value

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

… or search the BST whose root is the right child if the target is greater than our current value…

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

… or the target is equal to our current value so return our current node

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

What's the complexity?

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

*What's the complexity? (how many times do we call **find**)?*

find

```
1  public Optional<TreeNode<T>> find(T target, Optional<TreeNode<T>> root) {
2    if (!root.isPresent()) return Optional.empty();
3    if (target < root.value) {
4      return find(target, root.leftChild);
5    } else if (target > root.value()) {
6      return find(target, root.rightChild);
7    } else {
8      return root;
9    }
10 }
```

What's the complexity? (how many times do we call **`find`***)?* ***O(d)***

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (call insert on right subtree)

```java
public TreeNode<T> insert(T target, TreeNode<T> root) {
  if (target < root.value) {
    if (root.leftChild.isPresent()) {
      return insert(target, root.leftChild.get());
    } else {
      root.leftChild = Optional.of(new TreeNode<>(target));
      return root.leftChild.get();
    }
  } else if (target > root.value) {
    if (root.rightChild.isPresent()) {
      return insert(target, root.rightChild.get());
    } else {
      root.rightChild = Optional.of(new TreeNode<>(target));
      return root.rightChild.get();
    }
  } else {
    return root; // The value is already in the tree
}}
```

```
1  public TreeNode<T> insert(T target, TreeNode<T> root) {
2    if (target < root.value) {
3      if (root.leftChild.isPresent()) {
4        return insert(target, root.leftChild.get());
5      } else {
6        root.leftChild = Optional.of(new TreeNode<>(target));
7        return root.leftChild.get();
8      }
9    } else if (target > root.value) {
10     if (root.rightChild.isPresent()) {
11       return insert(target, root.rightChild.get());
12     } else {
13       root.rightChild = Optional.of(new TreeNode<>(target));
14       return root.rightChild.get();
15     }
16   } else {
17     return root; // The value is already in the tree
18 }}
```

If the target is smaller than our current root and the left subtree exists, insert into the left subtree

```
1  public TreeNode<T> insert(T target, TreeNode<T> root) {
2    if (target < root.value) {
3      if (root.leftChild.isPresent()) {
4        return insert(target, root.leftChild.get());
5      } else {
6        root.leftChild = Optional.of(new TreeNode<>(target));
7        return root.leftChild.get();
8      }
9    } else if (target > root.value) {
10     if (root.rightChild.isPresent()) {
11       return insert(target, root.rightChild.get());
12     } else {
13       root.rightChild = Optional.of(new TreeNode<>(target));
14       return root.rightChild.get();
15     }
16   } else {
17     return root; // The value is already in the tree
18 }}
```

If the target is smaller than our current root and the left subtree does not exist, insert the new node as the left subtree

```
1  public TreeNode<T> insert(T target, TreeNode<T> root) {
2    if (target < root.value) {
3      if (root.leftChild.isPresent()) {
4        return insert(target, root.leftChild.get());
5      } else {
6        root.leftChild = Optional.of(new TreeNode<>(target));
7        return root.leftChild.get();
8      }
9    } else if (target > root.value) {
10     if (root.rightChild.isPresent()) {
11       return insert(target, root.rightChild.get());
12     } else {
13       root.rightChild = Optional.of(new TreeNode<>(target));
14       return root.rightChild.get();
15     }
16   } else {
17     return root; // The value is already in the tree
18 }}
```

Repeat the same logic for the right tree

```
1  public TreeNode<T> insert(T target, TreeNode<T> root) {
2    if (target < root.value) {
3      if (root.leftChild.isPresent()) {
4        return insert(target, root.leftChild.get());
5      } else {
6        root.leftChild = Optional.of(new TreeNode<>(target));
7        return root.leftChild.get();
8      }
9    } else if (target > root.value) {
10     if (root.rightChild.isPresent()) {
11       return insert(target, root.rightChild.get());
12     } else {
13       root.rightChild = Optional.of(new TreeNode<>(target));
14       return root.rightChild.get();
15     }
16   } else {
17     return root; // The value is already in the tree
18 }}
```

Otherwise the value already exists so don't insert

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (call insert on right subtree)

What is the complexity?

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (call insert on right subtree)

What is the complexity? ***O(d)***

Remove

Goal: Remove the item with key ***k*** from a BST rooted at **`root`**

1. **`find`** the item
2. Replace the found node with the right subtree
3. Insert the left subtree under the right

We'll look at this in more detail later, but for now…

What's the complexity?

Remove

Goal: Remove the item with key ***k*** from a BST rooted at **`root`**

1. **`find`** the item
2. Replace the found node with the right subtree
3. Insert the left subtree under the right

We'll look at this in more detail later, but for now…

What's the complexity? ***O*(*d*)**

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

*What is the runtime in terms of **n**?*

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of n*?* $O(n)$

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of n*?* $O(n)$

What about the lower bound?

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of n*?* $O(n)$

What about the lower bound? $\Omega(\log(n))$

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of n*?* $O(n)$

What about the lower bound? $\Omega(\log(n))$

Can we do better?

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of n*?* $O(n)$

What about the lower bound? $\Omega(\log(n))$

Can we do better? **TBD…**

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Idea 1: Allow multiple copies ($X_L \leq X$ instead of $<$)

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Idea 1: Allow multiple copies ($X_L \leq X$ instead of $<$)

Idea 2: Only store one copy of each element, but also store a count