CSE 250
Data Structures

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

Trees (and Sets and Bags)

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 `mutable.Set[T]` ADT

`add(element: T): Unit`

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

`apply(element: T): Boolean`

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

`remove(element: T): Boolean`

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 `mutable.Bag[T]` ADT

`add(element: T): Unit`

Register the presence of a new (copy of) **`element`**

`apply(element: T): Integer`

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

`remove(element: T): Boolean`

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

Collection ADTs

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

(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 $d = \log(n)$

Quick Scala Tips

```scala
class TreeNode[T](
  var _value: T,
  var _left: Option[TreeNode[T]]
  var _right: Option[TreeNode[T]]
)

class Tree[T] {
  var root: Option[TreeNode[T]] = None // empty tree
}
```

We've seen how we can use options for objects that may not exist…

Quick Scala Tips

```scala
trait Tree[+T]

case class TreeNode[T](
  value: T,
  left: Tree[T],
  right: Tree[T]
) extends Tree[T]

case object EmptyTree extends Tree[Nothing]
```

But we can also use Traits and case classes…

Quick Scala Tips

```
trait Tree[+T]

case class TreeNode[T](
  value: T,
  left: Tree[T],
  right: Tree[T]
) extends Tree[T]

case object EmptyTree extends Tree[Nothing]
```

`TreeNode` and `EmptyTree` are two cases of `Tree`

But we can also use Traits and case classes…

Case Classes/Objects

Case Classes/Objects have two important features:

1. **Inline Constructors (no `new`):**

```
TreeNode(10,EmptyTree,EmptyTree)
```

2. **Match deconstructors:**

```
foo match { case TreeNode(v, l, r) => … }
```

Case Classes/Objects

```
def printTree[T](root: ImmutableTree[T], indent: Int) = {
  root match {
    case TreeNode(v, left, right) =>
      print((" " * indent) + v)
      printTree(left, indent + 2)
      printTree(right, indent + 2)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

Case Classes/Objects

```
def printTree[T](root: ImmutableTree[T], indent: Int) = {
  root match {
    case TreeNode(v, left, right) =>
      print((" " * indent) + v)
      printTree(left, indent + 2)
      printTree(right, indent + 2)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

If `root` is a `TreeNode` with value `v`, and subtrees `left` and `right`, print `v`, then call `printTree` on `left` and `right`

Case Classes/Objects

```
def printTree[T](root: ImmutableTree[T], indent: Int) = {
  root match {
    case TreeNode(v, left, right) =>
      print((" " * indent) + v)
      printTree(left, indent + 2)
      printTree(right, indent + 2)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

If `root` is an `EmptyTree` then don't do anything

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

```
def height[T](root: Tree[T]): Int = {
  root match {
    case EmptyTree =>
      0

    case TreeNode(v, left, right) =>
      1 + Math.max( height(left), height(right) )
}
```

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

Computing Tree Height

```
def height[T](root: Tree[T]): Int = {
  root match {
    case EmptyTree =>
      0

    case TreeNode(v, left, right) =>
      1 + Math.max( height(left), height(right) )
}
```

Case classes have a nice mapping onto functions with multiple cases

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

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 $\boldsymbol{X_L}$ in the left subtree of node $\boldsymbol{X}$: $\boldsymbol{X_L}$`.key` < $\boldsymbol{X}$`.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

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

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

What's the complexity?

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

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

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

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)

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

What is the complexity?
(how many calls to `insert`)?

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

What is the complexity?
(how many calls to `insert`)? ***O*(*d*)**

Remove

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

1. **`find`** the iterm
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)***

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 ($\boldsymbol{X_L} \leq \boldsymbol{X}$ 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 ($\boldsymbol{X_L \leq X}$ instead of **<**)

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

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**?* $O(n)$

Does it need to be that bad?