CSE 250
Data Structures

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

Tree Traversal and Rotations
Announcements

- PA3 Tests due Sunday Monday
  - Remember, no grace days
  - Implementation still due next Sunday, START EARLY
  - Implementation Autograder will be up shortly
<table>
<thead>
<tr>
<th>Property</th>
<th>Seq</th>
<th>Set</th>
<th>Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Order</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enforced Uniqueness</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Iterable</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
**BST Operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

What is the runtime in terms of $n$?
# BST Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

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

Does it need to be that bad?
## BST Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

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

Does it need to be that bad? ...hold that thought
<table>
<thead>
<tr>
<th>Property</th>
<th>Seq</th>
<th>Set</th>
<th>Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Order</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enforced Uniqueness</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Iterable</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
**Goal:** Visit every element of a tree (in linear time?)

**Pre-Order (top-down)**
Visit the root, then the left subtree, then the right subtree.

**In-Order**
Visit the left subtree, then the root, then the right subtree.

**Post-Order (bottom-up)**
Visit the left subtree, then the right subtree, then the root.
Tree Traversal: In-Order

```scala
def inorderVisit[T](root: ImmutableTree[T], visit: ImmutableTree[T] => Unit) = {
  root match {
    case TreeNode(v, left, right) =>
      /* visit left */
      inorderVisit(left, visit)
      /* visit root */
      visit(v)
      /* visit right */
      inorderVisit(right, visit)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```
In-Order Traversal on a BST
In-Order Traversal on a BST

\[ \text{inorderVisit}(6) \]
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)
inorderVisit(empty)
inorderVisit(3)
inorderVisit(2)
inorderVisit(1)
inorderVisit(8)
inorderVisit(9)
inorderVisit(7)
inorderVisit(10)
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)
visit(1)

Output: 1
In-Order Traversal on a BST

\[
\text{inorderVisit}(6) \\
\text{inorderVisit}(4) \\
\text{inorderVisit}(1) \\
\text{inorderVisit}(3)
\]

Output: 1
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)
inorderVisit(3)
inorderVisit(2)

Output: 1
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)
inorderVisit(3)
visit(2)

Output: 1 2
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)
inorderVisit(3)

Output: 1 2
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)
visit(3)

Output: 1 2 3
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(1)

Output: 1 2 3
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)

Output: 1 2 3
In-Order Traversal on a BST

`inorderVisit(6)`
`inorderVisit(4)`
`visit(4)`

Output: 1 2 3 4
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)

Output: 1 2 3 4
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
inorderVisit(5)

Output: 1 2 3 4
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)
visit(5)

Output: 1 2 3 4 5
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(4)

Output: 1 2 3 4 5
In-Order Traversal on a BST

inorderVisit(6)

Output: 1 2 3 4 5
In-Order Traversal on a BST

\[ \text{inorderVisit}(6) \]

\[ \text{visit}(6) \]

Output: 1 2 3 4 5 6
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)

Output: 1 2 3 4 5 6
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(7)

Output: 1 2 3 4 5 6
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(7)
visit(7)

Output: 1 2 3 4 5 6 7
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(7)
inorderVisit(8)

Output: 1 2 3 4 5 6 7
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(7)
inorderVisit(8)
visit(8)

Output: 1 2 3 4 5 6 7 8
In-Order Traversal on a BST

\[\text{inorderVisit}(6)\]
\[\text{inorderVisit}(10)\]
\[\text{inorderVisit}(7)\]
\[\text{inorderVisit}(8)\]
\[\text{inorderVisit}(9)\]

Output: 1 2 3 4 5 6 7 8
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(7)
inorderVisit(8)
visit(9)

Output: 1 2 3 4 5 6 7 8 9
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(7)
inorderVisit(8)

Output: 1 2 3 4 5 6 7 8 9
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(7)

Output: 1 2 3 4 5 6 7 8 9
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)

Output: 1 2 3 4 5 6 7 8 9
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
visit(10)

Output: 1 2 3 4 5 6 7 8 9 10
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(11)

Output: 1 2 3 4 5 6 7 8 9 10
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)
inorderVisit(11)
visit(11)

Output: 1 2 3 4 5 6 7 8 9 10 11
In-Order Traversal on a BST

inorderVisit(6)
inorderVisit(10)

Output: 1 2 3 4 5 6 7 8 9 10 11
In-Order Traversal on a BST

inorderVisit(6)

Output: 1 2 3 4 5 6 7 8 9 10 11
In-Order Traversal on a BST

Output: 1 2 3 4 5 6 7 8 9 10 11
Tree Traversal: In-Order Iterator

class ImmutableTreeIterator[T](root: ImmutableTree[T]) {
  /*** Initialize the Iterator ***/
  val toVisit = mutable.Stack[ImmutableTree[T]]
  pushLeft(root)

  def pushLeft(node: ImmutableTree[T]): Unit =
  node match {
    case EmptyTree => ()
    case t: ImmutableTree =>
      toVisit.push(t)
      pushLeft(t.left)
  }
...
Tree Traversal: In-Order Iterator

class ImmutableTreeIterator[T](root: ImmutableTree[T]) {
  /*** Initialize the Iterator ***/
  val toVisit = mutable.Stack[ImmutableTree[T]]
  pushLeft(root)

  def pushLeft(node: ImmutableTree[T]): Unit =
    node match {
      case EmptyTree => ()
      case t: ImmutableTree =>
        toVisit.push(t)
        pushLeft(t.left)
    }

  ...

  Initialize our iterator by recursively pushing the left trees (we know the FIRST element in an in-order traversal is the left-most
Tree Traversal: In-Order Iterator

class **ImmutableTreeIterator**[T](root: ImmutableTree[T]) {

    /*** Initialize the Iterator ***/
    val toVisit = mutable.Stack[ImmutableTree[T]]
    pushLeft(root)

    def pushLeft(node: ImmutableTree[T]): Unit =
    node match {
        case EmptyTree => ()
        case t: ImmutableTree =>
            toVisit.push(t)
            pushLeft(t.left)
    }

    ...

    Initialize our iterator by recursively pushing the left trees (we know the FIRST element in an in-order traversal is the left-most

    This pushes nodes onto our toVisit Stack, followed by their left trees (LIFO!)
class ImmutableTreeIterator[T](root: ImmutableTree[T]) {

... 

def isEmpty = toVisit.isEmpty

def next: T = {
    val nextNode = toVisit.pop
    pushLeft(nextNode.right)
    return nextNode.value
} 
}
Tree Traversal: In-Order Iterator

class ImmutableTreeIterator[T](root: ImmutableTree[T]) {

  ...

  def isEmpty = toVisit.isEmpty

  def next: T = {
    val nextNode = toVisit.pop
    pushLeft(nextNode.right)
    return nextNode.value
  }

  next pops the next node from our stack, and
  pushes it's right subtree, then returns it
}
In-Order Traversal with an Iterator
In-Order Traversal with an Iterator

When we create the iterator, the `toVisit` stack is initialized
In-Order Traversal with an Iterator

next pops the stack (1), and calls pushLeft on the right subtree of 1
In-Order Traversal with an Iterator

next pops the stack (2) and pushes the right subtree (nothing)
In-Order Traversal with an Iterator

next pops the stack (3) and pushes the right subtree (nothing)
In-Order Traversal with an Iterator

`next` pops the stack (4) and pushes the right subtree

Output: 1 2 3 4
In-Order Traversal with an Iterator

next pops the stack (5) and pushes the right subtree (nothing)
In-Order Traversal with an Iterator

next pops the stack (6) and pushes the right subtree (10 7)

Output: 1 2 3 4 5 6
In-Order Traversal with an Iterator

next pops the stack (7) and pushes the right subtree (8)

Output: 1 2 3 4 5 6 7
In-Order Traversal with an Iterator

`next` pops the stack (8) and pushes the right subtree (9)

Output: 1 2 3 4 5 6 7 8
In-Order Traversal with an Iterator

next pops the stack (9) and pushes the right subtree (nothing)
In-Order Traversal with an Iterator

next pops the stack (10) and pushes the right subtree (11)

Output: 1 2 3 4 5 6 7 8 9 10
In-Order Traversal with an Iterator

next pops the stack (11) and pushes the right subtree (nothing)

Output: 1 2 3 4 5 6 7 8 9 10 11
In-Order Traversal with an Iterator

Our toVisit stack is empty, so isEmpty will now be true.

Output: 1 2 3 4 5 6 7 8 9 10 11
val toVisit = mutable.Stack[ImmutableTree[T]]
pushLeft(root)

What is our worst-case runtime to initialize the iterator?
val toVisit = mutable.Stack[ImmutableTree[T]]
pushLeft(root)

What is our worst-case runtime to initialize the iterator? \(O(d)\)
What is our worst-case runtime to initialize the iterator? $O(d)$

*(we may have to push as many as $d$ nodes onto the stack)*
What is our worst-case runtime to call `next`?
What is our worst-case runtime to call `next`? $O(d)$

(we may have to push as many as $d$ nodes onto the stack)
What is the worst-case complexity to visit ALL n nodes?
What is the worst-case complexity to visit ALL $n$ nodes?

Each node is at the top of the stack exactly once:
What is the worst-case complexity to visit ALL $n$ nodes?

Each node is at the top of the stack exactly once:

- One push $O(1)$
Complexity

What is the worst-case complexity to visit ALL $n$ nodes?

Each node is at the top of the stack exactly once:

- One push $O(1)$
- One pop $O(1)$
What is the worst-case complexity to visit ALL $n$ nodes?

Each node is at the top of the stack exactly once:

- One push $O(1)$
- One pop $O(1)$

Total: $O(n)$
Balancing Trees
### BST Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

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

$$\log(n) \leq d \leq n$$
If height(left) ≈ height(right)

\[ d = O(\log(n)) \]

If height(left) ≪ height(right)

\[ d = O(n) \]
Balanced Trees are good: Faster find, insert, remove
Balanced Trees are good: Faster find, insert, remove

What do we mean by balanced?
Balanced Trees

Balanced Trees are good: Faster find, insert, remove

What do we mean by balanced? \[|\text{height(left)} - \text{height(right)}| \leq 1\]
Balanced Trees are good: Faster find, insert, remove

What do we mean by balanced? $|\text{height(left)} - \text{height(right)}| \leq 1$

How do we keep a tree balanced?
Balanced Trees - Two Approaches

**Option 1**
Keep left/right subtrees within \(+/-1\) of each other in height

(Add a field to track amount of "imbalance")

**Option 2**
Keep leaves at some minimum depth \((d/2)\)

(Add a color to each node marking it as "red" or "black")
Ok...but how do we enforce this...?
Rebalancing Trees (rotations)
Rebalancing Trees (rotations)

Rotate(A, B)
Rebalancing Trees (rotations)

\[ \text{Rotate}(A, B) \]
Rebalancing Trees (rotations)

Make $A$ the left child of $B$

Rotate($A$, $B$)
Rebalancing Trees (rotations)

Make A the left child of B

What about Y?

Rotate(A, B)
Rebalancing Trees (rotations)

Make A the left child of B

What about Y?

Make it the right child of A

Rotate(A, B)
Rebalancing Trees (rotations)

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Rotate(A, B)
A became B's left child
B's left child became A's right child

Is ordering maintained?

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!
Rebalancing Trees (rotations)

A became B's left child
B's left child became A's right child

Is ordering maintained? Yes!

B used to be the right child of A
Therefore B is bigger than A
Therefore A is smaller than B ✓

Rotate(A, B)
A became B's left child
B's left child became A's right child

Is ordering maintained? Yes!

Y used to be in the left subtree of B
Therefore Y is smaller than B
It is still left of B ✓

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child
B's left child became A's right child

Is ordering maintained? Yes!

Y used to be in the right subtree of A
It is still in the right subtree of A ✓

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child
B's left child became A's right child

Is ordering maintained? Yes!

Complexity?

Rotate(A, B)
A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!

Complexity? \( O(1) \)
Rebalancing Trees
Rebalancing Trees

Rotate(1,2)
Rebalancing Trees

Rotate(2,3)
Rebalancing Trees

Rotate(3,4)
Rebalancing Trees

Rotate(3,2)
Rebalancing Trees

Rotate(5,6)