Binary Search Trees

Concepts
Goals

• Introduce the Binary Search Tree (BST)
• Conceptual implementation of Bag interface with the BST
• Performance of BST Bag operations
Binary Search Tree

• Binary search trees are binary trees where every node’s value is:
  – *Greater than* all its descendents in the *left subtree*
  – *Less than or equal* to all its descendents in the *right subtree*

• If tree is reasonably full (*well balanced*), searching for an element is *O*(log *n*). Why?
Intuition
Binary Search Tree: Example

- Alex
  - Abner
    - Abigail
    - Adam
  - Adela
    - Agnes
- Angela
  - Alice
    - Allen
  - Audrey
    - Arthur
BST Bag: Contains

- Start at root
- At each node, compare value to node value:
  - Return true if match
  - If value is less than node value, go to left child (and repeat)
  - If value is greater than node value, go to right child (and repeat)
  - If node is null, return false
- Dividing in half each step as you traverse path from root to leaf (assuming reasonably full!!)
BST Bag: Contains/Find Example

Object to find → Agnes

Abner
  Abigail
    Adam

Adela
  Agnes

Angela
  Alice
    Allen
  Audrey

Alex
• Do the same type of traversal from root to leaf
• When you find a null value, create a new node
BST Bag: Add Example

Before first call to add

Object to add: Aaron

“Aaron” should be added here
BST Bag: Add Example

After first call to add

Alex

Abner

Abigail

Aaron

Adam

Adela

Agnes

Audrey

Angela

Alice

Allen

Arthur

Next object to add → Ariel

“Ariel” should be added here
How would you remove Abigail? Audrey? Angela?
Who fills the hole?

• Answer: the leftmost child of the right subtree (smallest element in right subtree)

• Try this on a few values

• Alternatively: The rightmost child of the left subtree
BST Bag: **Remove Example**

### Before call to `remove`

- **Alex**
  - **Abner**
    - **Abigail**
    - **Adam**
  - **Adela**
    - **Agnes**
  - **Alice**
    - **Allen**
  - **Angela**
    - **Audrey**
    - **Arthur**

Replace with: `leftmost(right)`

Element to remove
BST Bag: Remove Example

After call to **remove**
Special Case

• What if you don’t have a right child?
• Try removing “Audrey”
  – Simply return left child
Complexity Analysis (contains)

• If reasonably full, you’re dividing in half at each step: $O(\log n)$

• Alternatively, we are running down a path from root to leaf
  
  – We can prove by induction that in a complete tree (which is reasonably full), the path from root to leaf is bounded by $\text{floor}(\log n)$, so $O(\log n)$
We’ve shown all Bag operations to be proportional to the length of a path, rather than the number of elements in the tree.

We’ve also said that in a reasonably full tree, this path is bounded by: \( \text{floor}(\log_2 n) \)

This Bag is faster than our previous implementations!
## Comparison

- **Average Case Execution Times**

<table>
<thead>
<tr>
<th>Operation</th>
<th>DynArrBag</th>
<th>LLBag</th>
<th>Ordered ArrBag</th>
<th>BST Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td>Contains</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
<td>O(logn)</td>
</tr>
<tr>
<td>Remove</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
</tbody>
</table>