Trees III:

Binary Search Trees

A forest full of trees

- The generic toolkit of functions we have seen thus far can be applied to many types of data structures derived from binary trees
- Now we will look at a specific ADT based on the generic binary tree: binary search trees

BST characteristics

- Based on binary trees
- Defining quality has to do with the order in which data are stored
- Commonly used in database applications where rapid retrieval of data is desired

Binary Search Trees

- Entries in a BST must be objects to which total order semantics apply -- in other words, objects for which all the binary comparison operators are defined
- Storage rules -- for every node n:
  - every entry in n’s left subtree is less than or equal to the entry in n
  - every entry in n’s right subtree is greater than n’s entry

Binary Search Tree

There is no special requirement for the tree to maintain to maintain a particular shape -- but a balanced tree (in which there are approximately the same number of nodes in each subtree) facilitates data searching

The Dictionary Data Type

- A dictionary is a collection of items, similar to a bag.
- But unlike a bag, each item has a string attached to it, called the item's key.
The Dictionary Data Type

• A dictionary is a collection of items, similar to a bag.
• But unlike a bag, each item has a string attached to it, called the item’s key.

Example:
The items I am storing are records containing data about a state.

The Dictionary Data Type

• A dictionary is a collection of items, similar to a bag.
• But unlike a bag, each item has a string attached to it, called the item’s key.

Example:
The key for each record is the name of the state.

The Dictionary Data Type

void Dictionary::insert(The key for the new item, The new item);

• The insertion procedure for a dictionary has two parameters.

The Dictionary Data Type

• When you want to retrieve an item, you specify the key...

The Dictionary Data Type

Item Dictionary::retrieve("Washington");

• We'll look at how a binary tree can be used as the internal storage mechanism for the dictionary.
A Binary Search Tree of States

The data in the dictionary will be stored in a binary tree, with each node containing an item and a key.

A Binary Search Tree of States

Storage rules:

- Every key to the left of a node is alphabetically before the key of the node.
- Every key to the right of a node is alphabetically after the key of the node.

Example:
- 'Massachusetts' and 'New Hampshire' are alphabetically before 'Oklahoma'

A Binary Search Tree of States

Storage rules:

- Every key to the left of a node is alphabetically before
  the key of the node.
- Every key to the right of a node is alphabetically after
  the key of the node.

A Binary Search Tree of States

Storage rules:

- Every key to the left of a node is alphabetically before the key of the node.
- Every key to the right of a node is alphabetically after the key of the node.

Retrieving Data

Start at the root.

- If the current node has the key, then stop and retrieve the data.
- If the current node's key is too large, move left and repeat 1-3.
- If the current node's key is too small, move right and repeat 1-3.
Retrieve 'New Hampshire'

Start at the root.

- If the current node has the key, then stop and retrieve the data.
- If the current node's key is too large, move left and repeat 1-3.
- If the current node's key is too small, move right and repeat 1-3.

Adding a New Item with a Given Key

- Pretend that you are trying to find the key, but stop when there is no node to move to.
- Add the new node at the spot where you would have moved to if there had been a node.

Adding

- Pretend that you are trying to find the key, but stop when there is no node to move to.
- Add the new node at the spot where you would have moved to if there had been a node.
Adding

☆ Pretend that you are trying to find the key, but stop when there is no node to move to.
☆ Add the new node at the spot where you would have moved to if there had been a node.
Adding

Kazakhstan is the new right child of Iowa?

Removing an Item with a Given Key

↘ Find the item.
① If necessary, swap the item with one that is easier to remove.
② Remove the item.

Removing 'Florida'

↘ Find the item.

Removing 'Florida'

Florida cannot be removed at the moment...

Removing 'Florida'

① If necessary, do some rearranging.

The problem of breaking the tree happens because Florida has 2 children.
Removing 'Florida'  
① If necessary, do some rearranging.

For the rearranging, take the **smallest** item in the right subtree...

Removing 'Florida'  
① If necessary, do some rearranging.

...copy that smallest item onto the item that we're removing...

Removing 'Florida'  
① If necessary, do some rearranging.

... and then remove the extra copy of the item we copied...

Removing 'Florida'  
① If necessary, do some rearranging.

... and reconnect the tree

Removing 'Florida'  
Why did I choose the **smallest** item in the right subtree?

Because every key must be smaller than the keys in its right subtree
Removing an Item with a Given Key

☆ Find the item.
○ If the item has a right child, rearrange the tree:
   - Find smallest item in the right subtree
   - Copy that smallest item onto the one that you want to remove
   - Remove the extra copy of the smallest item (making sure that you keep the tree connected)
   - else just remove the item.

Summary

• Binary search trees are a good implementation of data types such as sets, bags, and dictionaries.
• Searching for an item is generally quick since you move from the root to the item, without looking at many other items.
• Adding and deleting items is also quick.
• But as you'll see later, it is possible for the quickness to fail in some cases -- can you see why?

Trees III:

Binary Search Trees
- ends -