Trees 2: Linked Representation & Binary Search Trees
Linked representation of binary tree

- Again, as with linked list, entire tree can be represented with a single pointer -- in this case, a pointer to the root node.
- Nodes are instances of class BTnode, described in the next several slides.
- The class contains methods that operate on single nodes.
- Since each node is potentially the root node of a tree (or subtree), we will also include operations on entire trees.
public class BTNode<E> {
    private E data;
    private BTNode<E> left, right;

    Private member
    data holds the
    information in the
    node; members
    left and right are
    pointers to the
    node’s left and
    right children
Methods of BTnode class

- **Constructor**: creates node with specified data and left and right children
- **Observer methods**
  - `getData()` returns value of data member
  - `getLeft()` returns pointer to left child
  - `getRight()` returns pointer to right child
  - `getLeftmostData()` and `getRightmostData()` return the data members from the leaves
  - `isLeaf()` returns true if node is a leaf
Methods of BTnode class

• Mutators
  – setData(): changes data portion of node
  – setLeft(): assigns new left child to node
  – setRight(): assigns new right child to node
  – removeLeftmost() and removeRightmost(): recursive methods that prune off leaves, returning the parent node
Static methods that operate at the tree level

• Observers (entire tree)
  – treeSize() returns number of nodes in tree
  – treeCopy() returns a new tree copied from a source tree argument
Output methods

- Traversal methods (entire tree):
  - preorderPrint(): prints data from nodes in preorder traversal pattern
  - postorderPrint(): prints data from nodes in postorder traversal pattern
  - print(): prints data from nodes in inorder traversal pattern
public BTN ode(E initialData, BTN ode<E> initialLeft, BTN ode<E> initialRight)
{
    data = initialData;
    left = initialLeft;
    right = initialRight;
}
Simple accessor methods

```java
public E getData( ) {
    return data;
}
public BTNode<E> getLeft( ) {
    return left;
}
public BTNode<E> getRight( ) {
    return right;
}
public boolean isLeaf( ) {
    return (left == null) &&
           (right == null);
}
```
Recursive accessors

```java
public E getLeftmostData() {
    if (left == null)
        return data;
    else
        return left.getLeftmostData();
}

public E getRightmostData() {
    if (right == null)
        return data;
    else
        return right.getRightmostData();
}
```
public void setData(E newData)   {
    data = newData;
}

public void setLeft(BTNode<E> newLeft) { 
    left = newLeft;
}

public void setRight(BTNode<E> newRight) { 
    right = newRight;
}
public BTNode<E> removeLeftmost() {
    if (left == null)
        return right;
    else {
        left = left.removeLeftmost();
        return this;
    }
}

public BTNode<E> removeRightmost() {
    if (right == null)
        return left;
    else {
        right = right.removeRightmost();
        return this;
    }
}
public static <E> BTNodel<E> treeCopy(BTNodel<E> source) {
    BTNodel<E> leftCopy, rightCopy;
    if (source == null)
        return null;
    else {
        leftCopy = treeCopy(source.left);
        rightCopy = treeCopy(source.right);
        return new BTNodel<E>(source.data, leftCopy, rightCopy);
    }
}
public static <E> long treeSize(BTNode<E> root) {
    if (root == null)
        return 0;
    else
        return 1 + treeSize(root.left) + treeSize(root.right);
}
Any operation that performs some process on all the nodes in a tree must perform a tree traversal.

Traversal refers to visiting each node in turn.

Traversal is a recursive process: we visit each node, and we visit each node in the subtree of which the node is the root.
Now, more ways to climb!

- There are three basic tree traversal patterns, referring to the order in which nodes are visited and processed
  - Pre-order: visit root, then left subtree, then right
  - In-order: visit left subtree, then root, then right
  - Post-order: visit left subtree, then right, then root
Example: printing all nodes

// pre-order traversal
public void preorderPrint()
{
    System.out.println(data);
    if (left != null)
        left.preorderPrint();
    if (right != null)
        right.preorderPrint();
}
Pre-order traversal in action

Original tree:

Results:

K
I
K
W
R
O
O
D
Example: printing all nodes

// post-order traversal
public void postorderPrint( )
{
    if (left != null)
        left.postorderPrint( );
    if (right != null)
        right.postorderPrint( );
    System.out.println(data);
}
Post-order traversal in action

Original tree:

Results:
Example: printing all nodes

// in-order traversal, with indents
public void print(int depth) {
    int i;
    // Print the left subtree (or a dash
    // if there is a right child and
    // no left child)
    if (left != null)
        left.print(depth+1);
    else if (right != null) {
        for (i = 1; i <= depth+1; i++)
            System.out.print("    ");
        System.out.println("--");
    }
}
Example: printing all nodes

// Print the indentation and the data
// from the current node:

    for (i = 1; i <= depth; i++)
        System.out.print("    ");
    System.out.println(data);
Example: printing all nodes

// Print the right subtree (or a dash if
// there is a left child and no right
// child)
if (right != null)
    right.print(depth+1);
else if (left != null){
    for (i = 1; i <= depth+1; i++)
        System.out.print("   ");
    System.out.println("--");
    System.out.println("--");
}
In-order traversal in action

Original tree:

Results:

K
I
W
K
R
O
D
A forest full of trees

• The generic BTNode class we have seen thus far can be used to create 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 a particular Shape -- but a balanced tree (in which there are approximately the same number of nodes in each subtree) facilitates data search.
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.
- We have seen how a hash table might be used to implement the Dictionary ADT; now we'll implement it as binary search.
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.
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.
Where would you add this state?
Removing an Item with a Given Key

Find the item.
If necessary, swap the item with one that is easier to remove.
Remove the item.
Removing 'Florida'

1. **Find** the item.
Florida cannot be removed at the moment...
... because removing Florida would break the tree into two pieces.
The problem of breaking the tree happens because Florida has 2 children.

If necessary, do some rearranging.
Removing 'Florida'

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

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

... and then remove the extra copy of the item we copied...
Removing 'Florida'

... and reconnect the tree
Why did I choose the smallest item in the 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 … it is possible for the quickness to fail in some cases -- can you see why?