Topic

- Tree
- BST
Tree

- Each node in a tree has zero or more child nodes. A node that has a child is called the child's parent node. A node has at most one parent.

- An internal node is any node of a tree that has child nodes.

- An leaf node is any node that does not have child nodes.

- The topmost node in a tree is called the root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin.
Tree

- root
- parent of
- internal node
- child of
- leaf node
Height and Depth

- Height is calculated by traversing from leaf to the given node.
- Depth is calculated from traversal from root to the given node.
- Values of height and depth of a given tree are the same.
Binary Tree

- A binary tree is a tree in which no node has more than two children (right child and left child).
Binary Search Tree

- Binary search tree (BST) is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - Both the left and right subtrees must also be binary search trees.
Binary Search Tree
Tree traversal

- Preorder, perform the following operations recursively:
  - Visit the root.
  - Traverse the left subtree.
  - Traverse the right subtree.

- Inorder, perform the following operations recursively:
  - Traverse the left subtree.
  - Visit the root.
  - Traverse the right subtree.

- Postorder, perform the following operations recursively:
  - Traverse the left subtree.
  - Traverse the right subtree.
  - Visit the root.
A InOrder visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25
preorder(node)
    if node == null then return
    visit(node)
    preorder(node.left)
    preorder(node.right)

inorder(node)
    if node == null then return
    inorder(node.left)
    visit(node)
    inorder(node.right)

recursivePostorder(node)
    if node == null then return
    recursivePostorder(node.left)
    recursivePostorder(node.right)
    visit(node)
Implementation: BSTNode

- Different than LLNode, only one reference
- For binary tree, two references: left and right child
- For arbitrary tree, array of children
package support;

public class BSTNode<T extends Comparable<T>> {

    private T info;
    private BSTNode<T> left = null;
    private BSTNode<T> right = null;

    BSTNode(T info) { this.info = info; }

    public T getInfo() { return info; }

    public void setInfo(T info) { this.info = info; }

    public BSTNode<T> getLeft() { return left; }

    public void setLeft(BSTNode<T> link) { left = link; }

    public BSTNode<T> getRight() { return right; }

    public void setRight(BSTNode<T> link) { right = link; }

}
Implementation:
BinarySearchTree
package ch08.trees;

import ch03.stacks.*;  // used for iterative size()
import cho4.queues.*   // used for traversals
import support.BSTNode;

public class BinarySearchTree<T extends Comparable<T>> {  
    private BSTNode<T> root = null;
    private found;        // used by remove()
    private LinkUnbndQueue<T> BSTQueue;
    private LinkUnbndQueue<T> preOrderQueue;
    private LinkUnbndQueue<T> postOrderQueue;
    public boolean isEmpty() { return root == null; }
    public int size() { return recSize(root); }
    // rest later
}  // end of BinarySearchTree<T>
```java
private int recSize(BSTNode<T> tree) {
    if (tree == null)
        return 0;
    return recSize(tree.getLeft()) + recSize(tree.getRight()) + 1;
}
```

In this case, and often with trees, it is no contest: recursion is much more natural and consequentially much easier.
Find

Use the search key to direct a recursive binary search for a matching node

1. Start at the root node as current node

2. If the search key’s value matches the current node’s key then found a match

3. If search key’s value is greater than current node’s
   1. If the current node has a right child, search right
   2. Else, no matching node in the tree

4. If search key is less than the current node’s
   1. If the current node has a left child, search left
   2. Else, no matching node in the tree
1. start at the root, 45 is greater than 25, search in right subtree

2. 45 is less than 50, search in 50’s left subtree

3. 45 is greater than 35, search in 35’s right subtree

4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST
private boolean ecContains(T element, BSTNode<T> tree) {
    if (tree == null)
        return false;
    if (element.compareTo(tree.getInfo()) == 0)
        return true;
    if (element.compareTo(tree.getInfo()) < 0)
        return recContains(element, root.getLeft());
    return recContains(element, root.getRight());
}
Add()

1. Start at root node as current node

2. If new node’s key < current’s key
   1. If current node has a left child, search left
   2. Else add new node as current’s left child

3. If new node’s key > current’s key
   1. If current node has a right child, search right
   2. Else add new node as current’s right child
1. start at the root, 60 is greater than 25, search in right subtree

2. 60 is greater than 50, search in 50’s right subtree

3. 60 is less than 70, search in 70’s left subtree

4. 60 is less than 66, add 60 as 66’s left child
private BSTNode<T> recAdd(T element, BSTNode<T> tree) {
    if (tree == null) // insert new node here
        tree = new BSTNode<T>(element);
    else if (element.compareTo(tree.getInfo()) < 0)
        tree.setLeft(recAdd(element, tree.getLeft()));
    else
        tree.setRight(recAdd(element, tree.getRight()));
    return tree;
}
Remove()  

- Remove operation on binary search tree is more complicated than add and search. Basically, it can be divided into two stages:
  - search for a node to remove
  - if the node is found, run remove algorithm
Node to be removed has no children

- This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.
Node to be removed has one child.

- In this case, the node is cut from the tree and the algorithm links its single child (with its subtree) directly to the parent of the removed node.
Node to be removed has two children.

- This is the most complex case.
  - Find a minimum value in the right subtree
  - Replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
  - apply remove to the right subtree to remove a duplicate.
  - Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.
Example. Remove 12 from a BST.

Find minimum element in the right subtree of the node to be removed. In current example it is 19.
Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.

Remove 19 from the left subtree.
private T getPredcessorInfo(BSTNode<T> tree) {
    tree = getLeft(tree);
    while (tree.getRight() != null)
        tree = tree.getRight();
    return tree.getInfo();
}

private T getSuccessorInfo(BSTNode<T> tree) {
    tree = getRight(tree);
    while (tree.getLeft() != null)
        tree = tree.getLeft();
    return tree.getInfo();
}