Data Structures so far

- Arrays
  - **Pros**: Fast random access, know the size, bounds checking, easy to find things, fast to sort
  - **Cons**: adding and removing, have to know the size in advance
- Linked Lists
  - **Pros**: easy to add and remove, can change the size
  - **Cons**: hard to find things, no random access, can't efficiently sort
- Binary Search Trees
  - **Pros**: fairly easy to add and remove, as easy to find things as an array, fast to sort
  - **Cons**: no random access.

What are trees?

- Parents-> children
- Connected nodes
- Directed, acyclic graph
- Some of our favorite trees:
  - File System
  - AST – abstract syntax tree
  - Priority queue
  - Databases
  - The heap

Printing a tree

Our old friend recursion is back! Super easy way to deal with trees.

Preorder

```
public void traverse () {
	traverse(root); System.out.println();
}
private static void traverse (TreeNode node) {
	if (node == null)
		return;
	System.out.print(node.datum + " ");
	traverse(node.lchild);
	traverse(node.rchild);
}
```

Inorder

```
public void traverse () {
       traverse(root); System.out.println();
}
private static void traverse (TreeNode node) {
       if (node == null)
              return;
       traverse(node.lchild);
       System.out.print(node.datum + " ");
       traverse(node.rchild);
}
```

Postorder

```
public void traverse () {
       traverse(root); System.out.println();
}
private static void traverse (TreeNode node) {
       if (node == null)
              return;
       traverse(node.lchild);
       traverse(node.rchild);
       System.out.print(node.datum + " ");
}
```

Adding to a tree

Adding is pretty straight forwards if we use recursion. There are a few steps:

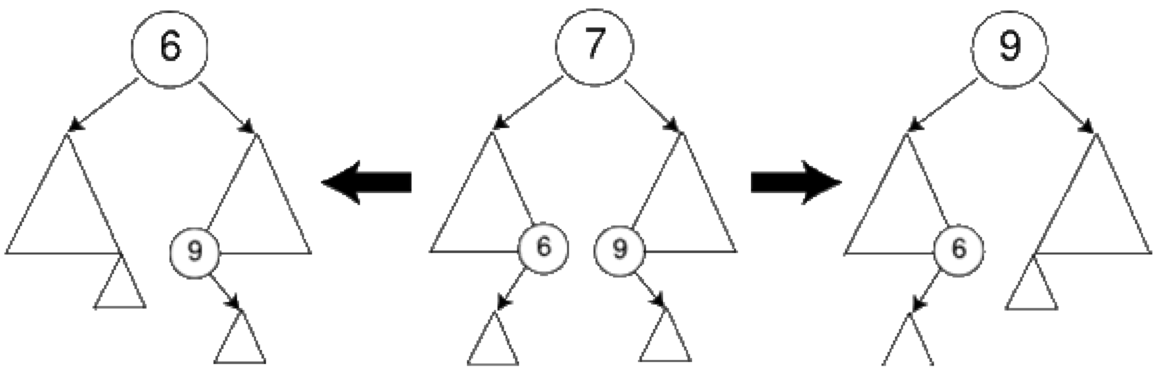
1. base case: tree is null, so this is the new root (this will happen when we get to a leaf node and need to make our addition one of its children)
2. If our insertion element is greater than the current node we are visiting, recursively call insert on the right tree.
3. If our insertion element is less than the current node we are visiting, recursively call insert on the left tree.

```
public void insert (int x) {
       root = insert(x, root);  // begin the recursion
}
private static TreeNode insert (int x, TreeNode node) {
       if (node == null)
              return new TreeNode(x);
       if (x < node.element)
              node.lchild = insert(x, node.lchild);
       else if (x > node.element)
              node.rchild = insert(x, node.rchild);
       return node;
}
```

Deletion

This is the hardest BST operation. There are several cases to be considered:

- **Deleting a leaf:** Deleting a node with no children is easy, as we can simply remove it from the tree.
- **Deleting a node with one child:** Delete it and replace it with its child.
- **Deleting a node with two children:** Suppose the node to be deleted is called *N*. We replace the value of N with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree).



Once we find either the in-order successor or predecessor, swap it with N, and then delete it. Since either of these nodes must have less than two children (otherwise it cannot be the in-order successor or predecessor), it can be deleted using the previous two cases. *(from wikipedia)*

```
public void delete(int x){
      delete(x, rootNode); // initiate the recursive delete
}
public treeNode delete(int x, treeNode t) {
  treeNode ptr;
  if (T == NULL) return null;
  else if (x < t.element) /* go left */
    T->left = delete(x, t.left);
  else if (x > t.element) /* go right */
    T->right = delete(x, t.right);
  else /* found element to be deleted */
  if (t.left && t.right) { /* two children */
    /* replace with smallest in right subtree */
    ptr = find_min(t.right);
    t.element = ptr.element;
    t.right = delete(ptr.element, ptr.right);
  }
  else { /* one or no children */
    ptr = t;
    if (t.left == null)  /* only a right child or no children */
      T = t.right;
    else
    if (t.right == null) /* only a left child */
      T = t.left;
  }
  return t;
}
```