Binary Trees

- A binary tree is either empty or it contains a root node and left- and right-subtrees that are also binary trees.
  - A binary tree is a nonlinear data structure.
- The top node of a tree is called the root.
- Any node in a binary tree has at most 2 children.
- Any node in a binary tree has exactly one parent node (except the root).

Tree Terminology
Types of Binary Trees

Expression Trees

```
+ / 5 7
  6 2

(6 / 2 + 5) * (7 - 3)
```

Huffman Trees

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>45%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>30%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>5%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
A  1
B  00
C  010
D  011
```

1001010100 = ABACAB
Types of Binary Trees

Binary Search Trees

More Terminology

- A **full binary tree** is a binary tree such that
  - all leaves have the same level, and
  - every non-leaf node has 2 children.
- A **complete binary tree** is a binary tree such that
  - every level of the tree has the maximum number of nodes possible except possibly the deepest level, and
  - at the deepest level, the nodes are as far left as possible.
Examples

Binary Trees & Recursion

- Consider two nodes in a tree, X and Y.
- X is an ancestor of Y if
  - X is the parent of Y, or
  - X is the ancestor of the parent of Y.

  It’s RECURSIVE!

- X is a descendant of Y if
Binary Trees Levels

- The level of a node $Y$ is
  - BASE CASE
  - RECURSIVE CASE
Binary Tree - Height

- The **height of a binary tree** $T$ is the number of nodes in the longest path from the root node to a leaf node.
  - **BASE CASE**

- **RECURSIVE CASE**
Traversals are Recursive

- Preorder traversal
  1. Visit the root.
  2. Perform a preorder traversal of the left subtree.
  3. Perform a preorder traversal of the right subtree.
- Inorder traversal
  1. Perform an inorder traversal of the left subtree.
  2. Visit the root.
  3. Perform an inorder traversal of the right subtree.
- Postorder traversal
  1. Perform a postorder traversal of the left subtree.
  2. Perform a postorder traversal of the right subtree.
  3. Visit the root.
Traversals on Expression Trees

• What do you get when you perform a postorder traversal on an expression tree?

```
  *  
 / 
+ / 
5 7 3
 
6 2
```

Traversals on Binary Search Trees

• What do you get when you perform an INORDER traversal on a binary search tree?

```
  84
 / 
41 96
 / 
24 50 98
/  
13 37
```
Implementing a binary tree

- Use an array to store the nodes.
  - mainly useful for complete binary trees (more on this soon)
- Use a variant of a linked list where each data element is stored in a node with links to the left and right children of that node
- Instead of a head reference, we will use a root reference to the root node of the tree.

Binary Tree Node
(BTNode inner class)

```java
private static class BTNode<E> {
    private E data;
    private BTNode<E> left;
    private BTNode<E> right;

    public BTNode(E element) {
        data = element;
        left = null;
        right = null;
    }
}
```
public class BinaryTree<E> {

    private BTNode root;

    public BinaryTree() {
        root = null;
    }

    public BinaryTree(E element, BinaryTree<E> leftTree, BinaryTree<E> rightTree) {
        root = new BTNode<E>(element);
        if (leftTree != null)
            root.left = leftTree.root;
        if (rightTree != null)
            root.right = rightTree.root;
    }
}
BinaryTree class

public BinaryTree<E> getLeftSubtree() {
    if (root != null && root.left != null)
        return new BinaryTree<E>(root.left);
    else
        return null;
}

protected BinaryTree(BTNode<E> rootRef) {
    root = rootRef;
}

Preorder Traversal

public String preorder() {
    return preorder(root);
}

private String preorder(Node<E> node) {
    String result = "";
    if (node != null) {
        result += node.data + " ";
        result += preorder(node.left) + " ";
        result += preorder(node.right) + " ";
    }
    return result;
}
Binary Search Trees

- A binary tree $T$ is a binary search tree if
  - $T$ is empty, or
  - $T$ has two subtrees, $T_L$ and $T_R$, such that
    - All the values in $T_L$ are less than the value in the root of $T$,
    - All the values in $T_R$ are greater than the value in the root of $T$, and
    - $T_L$ and $T_R$ are binary search trees

Searching for 83 and 42

![Diagram showing searching for 83 and 42 in a binary search tree.](image)
Finding data in a BST

- If the root is null, return null.
- Compare the root value to the target.
- If they are equal, return a reference to the data in the root.
- Otherwise if the target is less than the root value, return the result of the search on the left subtree.
- Otherwise (since the target is greater than the root value), return the result of the search on the right subtree.

Inserting into a BST

```
62  96  11  39  21  83  45
```

![BST Diagram]
Inserting data into a BST

- If the root is null, replace empty tree with new tree containing the new element and return true.
- If the item is equal to the root data, return false.
- If the item is less than the root data, insert the new element into the left subtree and return the result of that insert operation.
- If the item is greater than the root data, insert the new element into the right subtree and return the result of that insert operation.

Efficiency of insert on a full BST

- A full binary search tree with height H has how many nodes?
  - $N = 1 + 2 + 4 + \ldots + 2^{H-1} = 2^H - 1$
  - Thus, $H = \log_2(N+1)$.
- An insert will take $O(\log N)$ time on a full BST since we have to examine one node at each level in the worst case before we find the insert point, and there are H levels.
Efficiency of insert on a BST

- Are all BSTs full? **NO!**
  - Insert the following numbers in order into a BST: 11  21  39  45  62  83  96
  - What do you get?

- An insert will take $O(N)$ time in the worst case on an arbitrary BST since there may be up to $N$ levels in a BST with $N$ nodes.
  - We will take a brief look at balanced search trees later this semester.

Removing from a BST

General idea:

- Start at the root and search for the item to remove by progressing one level at a time until either:
  - the item is found
  - we reach a leaf and the item is not found

- If the item is found, remove the node and repair the tree so it is still a BST.
Removing from a BST

- If the root is null, return null.
- Compare the item to the root data.
- If the item is less than the root data, return the result of removing the data from the left subtree.
- If the item is greater than the root data, return the result of removing the data from the right subtree.

(cont'd)

Removing from a BST

Data value is found at local root

- Store the local root data in a temp variable.
- If the local root has no children, set the parent reference to this root to null.
- Example:

  Remove 68

  ![Diagram of BST with nodes 45, 85, 68, and 91]
Removing from a BST
Data value is found at root (cont’d)

- If the local root has one child, set the parent reference to this root to the reference to this root's only child.

Example:
Remove 31.

Removing from a BST
Data value is found at root (cont’d)

- If the local root has two children:
  - If this root's left child has no right child
    - Copy the data from this root's left child into this root.
    - Set the left reference of this root to the reference to the left child of this root's left child.
  - Otherwise:
    - Find the rightmost node in the right subtree of this root's left child (a.k.a. the inorder predecessor).
    - Copy the data from the rightmost node from the previous step into this root's data field and then remove the original data by setting its parent to reference its left child.
Removing from a BST
Data value is found at root (cont’d)

- Example:
  Remove 29.

```
        45
       /  \
      29   36
     /    /  \
   17   17   null
  /   /  \
 6   27  null
```

(rightmost node of left subtree of 29)
Removing from a BST
Data value is found at root (cont’d)

- Example: Remove 29.

```
  45
  /  
45   36
 /  
 29 27
 /  
15  36
 /  /  
6 15 21
 / 
6 21
```

- If we remove the node referenced by our local root variable, how do we change the left or right field of its parent (if this node does not have a pointer back to the parent)?
  - Solution: Return a reference to the replacement reference (or null) as the result and let the previous invocation store the result in its local root node (which is the parent of the root node we want to remove).