# Trees Too

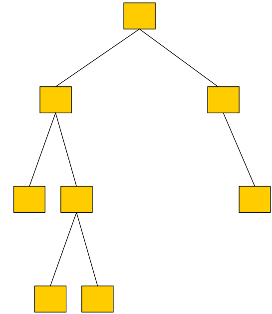
Exam 2: Due Monday, April 2

## The Binary Tree

The **height** of a node *n* is length of the longest path from *n* to a leaf below it. The **height of a tree** is the height of the root.

The **depth** of a node *n* is the length of the path from *n* to the root.

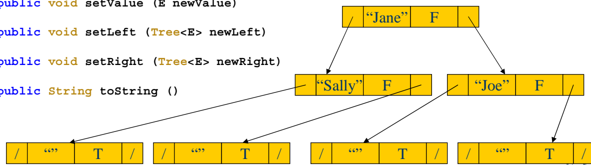
A binary tree is **height-balanced** iff at every node *n*, the heights of *n*'s left and right subtrees differ by no more than 1.



## Tree Implementation Contract

```
public Tree()

public Tree(E value, Tree<E> lt, Tree<E> rt)

public boolean isLeaf ()

public E getValue ()

public Tree<E> getLeft ()

public Tree<E> getRight ()

public void setValue (E newValue)

public void setLeft (Tree<E> newLeft)

public void setRight (Tree<E> newRight)

public String toString ()
```

```
// instance variables

private E value;
private Tree<E> lt, rt;
private boolean isLeaf;
```



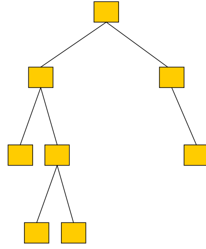
## Defining Class Methods for Syntatic Sugar

```
public static <E> Tree<E> leaf () {
  return new Tree<E>();
}

public static <E> boolean isLeaf (Tree<E> t) {
  return t.isLeaf;
}

public static <E> Tree<E> node (E val,
                        Tree<E> lt, Tree<E> rt) {
  return new Tree<E>(val, lt, rt);
}

public static <E> E getValue (Tree<E> t) {
  return t.getValue();
}

public static <E> Tree<E> getLeft (Tree<E> t) {
  return t.getLeft();
}

public static <E> Tree<E> getRight (Tree<E> t)
{
  return t.getRight();
}
```

```
public static <E> void setValue
          (Tree<E> t, E newValue) {
  t.setValue(newValue);
}

public static <E> void setLeft
       (Tree<E> t, Tree<E> newLeft) {
  t.setLeft(newLeft);
}

public static <E> void setRight
      (Tree<E> t, Tree<E> newRight) {
  t.setRight(newRight);
}
```

## Basic Operations: height() and countNodes()

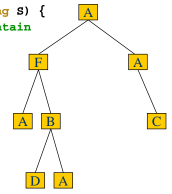
```
public static <E> int height (Tree<E> t) {
    // returns the height of the tree t
    if (isLeaf(t))
      return 0;
    else
      return 1 + Math.max(
        height(getLeft(t)),
        height(getRight(t)));
}

public static <E> int countNodes (Tree<E> t) {
    // returns the number of nodes in tree t
    if (isLeaf(t))
      return 0;
    else
      return 1 + countNodes(getLeft(t)) +
                 countNodes(getRight(t));
}
```



## Basic Operations: countOccurrences()

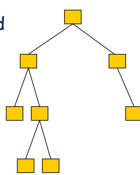
- **Count occurrences** of S in tree T
  - If root contains S, it is 1 +
  - The number of times it occurs in the left subtree +
  - The number of times it occurs in the right subtree

```
public static int countOccurrences (Tree<String> t, String S) {
    // returns the number of nodes in the tree t that contain
    // the input String S
    if (isLeaf(t))
      return 0;
    else {
      if (getValue(t).equals(S))
        return 1 + countOccurrences(getLeft(t), S) +
                   countOccurrences(getRight(t), S);
      else
        return countOccurrences(getLeft(t), S) +
                   countOccurrences(getRight(t), S);
    }
}
```



## Basic Operations: isMember()

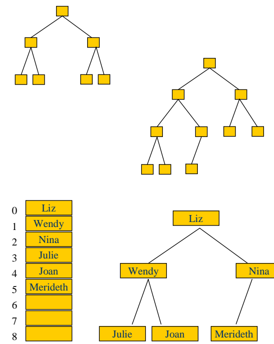
isMember() returns true iff the input word is contained somewhere in the tree



```
public static boolean isMember (String word,
                                Tree<String> t) {
   if (isLeaf(t))
      return false;
   else if (getValue(t).equals(word))
      return true;
   else
      return (isMember(word, getLeft(t)) ||
              isMember(word, getRight(t)));
}
```

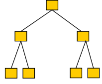
## Full and Complete

- A **full** binary tree is a binary tree of height *h* with no missing nodes
  - All leaves are at level *h*
  - All internal nodes each have two children
- A **complete** binary tree is a binary tree of height *h* that is full to level *h* – 1 and has level *h* filled in from left to right
- Full binary trees are complete
- There is an easy implementation for full and complete trees



## Recursive Definition of Full Binary Tree

- **Full** binary tree
  - If T is empty,
    T is a full binary tree of height 0
  - If T is not empty tree of height *h*,
    T is a full binary tree if its root's
    subtrees are both full binary trees of
    height *h* - 1



```
public static <E> boolean isFullBinaryTree (Tree<E> t) {
   if (isLeaf(t))
      return true;
   else
      return isFullBinaryTree(getLeft(t)) &&
             isFullBinaryTree(getRight(t)) &&
             (height(t)-1 == height(getLeft(t)) &&
             (height(t)-1 == height(getRight(t)));
}
```

## Array Representation of Binary Trees

***Array-based representation***

A binary tree is represented by using an array of tree nodes

Each tree node contains a data portion and two indexes (one for each of the node's children)

Requires the creation of a **free** list which keeps track of available nodes

The free list is being kept inside the tree array!

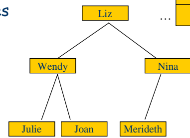
Allows for representation of any kind of tree, not just complete ones

tree

| | item | lt | rt |
|---|---|---|---|
| 0 | Liz | 1 | 2 |
| 1 | Wendy | 3 | 4 |
| 2 | Nina | 5 | -1 |
| 3 | Julie | -1 | -1 |
| 4 | Joan | -1 | -1 |
| 5 | Merideth | -1 | -1 |
| 6 | | -1 | 7 |
| 7 | | -1 | 8 |
| 8 | | -1 | 9 |
| … | … | … | … |

0 root

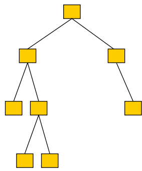
6 free



## Basic Operations: isBalanced()

A binary tree is **height-balanced** iff
the height of any node's right subtree
differs by no more than 1
from the height of the node's left subtree

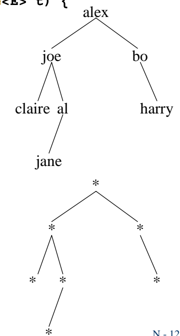
- Complete binary trees are balanced



```
public static <E> boolean isBalanced(Tree<E> t) {
    if (isLeaf(t))
      return true;
    else
      return ((Math.abs(height(getLeft(t)) -
                       height(getRight(t))) <= 1)
            && (isBalanced(getLeft(t)))
            && (isBalanced(getRight(t))));
}
```

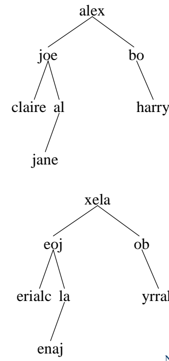
## Recording the Shape of a Tree

```
public static <E> Tree<String> makeStarTree (Tree<E> t) {
   // returns a new tree that has
   // the same size and shape as t, but with
   // all nodes in the tree containing
   // a String with a * character
     if (isLeaf(t))
       return leaf();
     else
       return node ("*",
                    makeStarTree(getLeft(t)),
                    makeStarTree(getRight(t)));
}
```



## Applying a Function on a Tree

```
public static Tree<String> makeReverseTree
                              (Tree<String> t) {
  // returns a new tree that is the same size
  // and shape as t, but with all of the
  // Strings labels in the tree reversed
    if (isLeaf(t))
      return leaf();
    else
      return node
        (reverseString(getValue(t)),
        makeReverseTree(getLeft(t)),
        makeReverseTree(getRight(t)));
}

public static String reverseString (String S) {
    // returns a String that is the
    // reverse of the input String S
    String newS = "";
    for (int i = 0; i < S.length(); i++)
      newS = S.charAt(i) + newS;
    return newS;
}
```

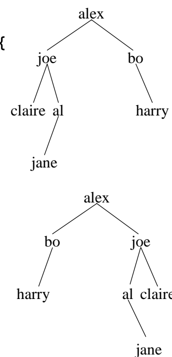


## Destructive Versions

```
public static void makeStarTree2 (Tree<String> t) {
    // alters the contents of the input tree so that all of the
    // nodes contain a String with a star character
    if (!(isLeaf(t))) {
      setValue(t, "*");
      makeStarTree2(getLeft(t));
      makeStarTree2(getRight(t));
    }
}

public static void makeReverseTree2 (Tree<String> t) {
    // alters the contents of the input tree so that all of the
    // Strings in the tree are reversed
    if (!(isLeaf(t))) {
      setValue(t, reverseString(getValue(t)));
      makeReverseTree2(getLeft(t));
      makeReverseTree2(getRight(t));
    }
}
```

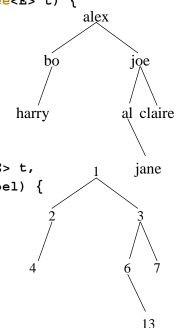
## daVinci Tree?

```
public static <E> void flipTree (Tree<E> t) {
    // alters the shape of tree by
    // changing it to a mirror reversal
    if (!isLeaf(t)) {
      Tree<E> tempTree = getLeft(t);
      setLeft(t, getRight(t));
      setRight(t, tempTree);
      flipTree(getLeft(t));
      flipTree(getRight(t));
    }
}
```



## The Index of an Array Representation

```
public static <E> Tree<String> makeLabelTree (Tree<E> t) {
  // returns a new tree with the same shape
  // and size as the input tree,
  // in which the value in each node
  // of the new tree is the index
  // of the node in an array representation
  return labelTree(t, 1);
}

public static <E> Tree<Integer> labelTree (Tree<E> t,
                                          int label) {
  // recursive helper method of makeLabelTree()
  if (isLeaf(t))
    return leaf();
  else
    return node(label,
      labelTree(getLeft(t), 2*label),
      labelTree(getRight(t), 2*label+1));
}
```



## makeHeightTree()

```
public static <E> Tree<Integer> makeHeightTree
                                    (Tree<E> t)
 {
  /* returns a new tree that has the same size
   and shape as t, and in which the node value
   is the height of the tree rooted at the
   corresponding node of the input tree */
  if (isLeaf(t))
    return leaf();
  else
    return node(height(t),
              makeHeightTree(getLeft(t)),
              makeHeightTree(getRight(t)));
}
```

alex
bo
joe
harry
al
claire
jane

4
2
3
1
2
1
1



## makeLevelTree()

```
public static <E> Tree<Integer> makeLevelTree
                                    (Tree<E> t) {
  /* returns a new tree that is the same size
   and shape as t, in which the value in each
   node of the new tree is the level of the
   corresponding node in the input tree */
  return levelTree(t, 0);
}

public static <E> Tree<Integer> levelTree
                              (Tree<E> t, int h) {
  // recursive helper method for makeLevelTree()
  if (isLeaf(t))
    return leaf();
  else
    return node(h + 1,
              levelTree(getLeft(t), h + 1),
              levelTree(getRight(t), h + 1));
}
```

alex
bo
joe
harry
al
claire
jane

1
2
2
3
3
3
4



## Binary Search Tree (BST)

- A binary tree that for each node *n*:
  - *n*'s value is greater than all values in its left subtree $T_L$
  - *n*'s value is less than all values in its right subtree $T_R$
  - Both $T_L$ and $T_R$ are binary search trees

5
2
6
1
4
7
3

A binary search tree of numbers

Liz
Gina
Nina
Elaine
Julie
Merideth
Wendy

A binary search tree of names



## Decision Trees for Expert Systems

Welcome to the game of Twenty Questions!
Think of an animal and I will try to guess it

- Does it have four legs?
- **yes**
- Does it have hooves?
- **no**
- Is it a dog?
- **yes**
- I won!
- Keep playing?
- **yes**
- Think of another animal and I will try to guess it
- Does it have four legs?
- **no**
- Does it swim?
- **no**
- Is it an ostrich?
- **no**
- I give up! What is your animal
- **me!**

Does it have 4 legs?
no
yes
Does it swim?
Does it have hooves?
no
yes
no
yes
Is it an Ostrich?
Is it a Goldfish?
Is it a Dog?
Is it a Cow?