### Binary Trees
- Linked lists: efficient insertion/deletion, inefficient search
  - ArrayList: search can be efficient, insertion/deletion not
- Binary trees: efficient insertion, deletion, and search
  - trees used in many contexts, not just for searching, e.g., expression trees
  - search in $O(\log n)$ like sorted array
  - insertion/deletion $O(1)$ like list, once location found!
  - binary trees are inherently recursive, difficult to process
  - not non-recursively, but possible
    - recursion never required, often makes coding simpler

### From doubly-linked lists to binary trees
- Instead of using prev and next to point to a linear arrangement, use them to divide the universe in half
  - Similar to binary search, everything less goes left, everything greater goes right
  - How do we search?
  - How do we insert?

### Basic tree definitions
- Binary tree is a structure:
  - empty
  - root node with left and right subtrees
- terminology: parent, children, leaf node, internal node, depth, height, path
  - link from node $N$ to $M$ then $N$ is parent of $M$
    - $M$ is child of $N$
  - leaf node has no children
    - internal node has 1 or 2 children
  - path is sequence of nodes, $N_1$, $N_2$, ..., $N_k$
    - $N_i$ is parent of $N_{i+1}$
    - sometimes edge instead of node
  - depth (level) of node: length of root-to-node path
    - level of root is 1 (measured in nodes)
  - height of node: length of longest node-to-leaf path
    - height of tree is height of root

### A TreeNode by any other name...
- What does this look like?
  - What does the picture look like?

```java
public class TreeNode {
    TreeNode left;
    TreeNode right;
    String info;

    TreeNode(String s, TreeNode llink, TreeNode rlink) {
        info = s;
        left = llink;
        right = rlink;
    }
}
```
Printing a search tree in order

- When is root printed?
  - After left subtree, before right subtree.

```java
void visit(TreeNode t)
{
    if (t != null) {
        visit(t.left);
        System.out.println(t.info);
        visit(t.right);
    }
}
```

- Inorder traversal

Insertion and Find? Complexity?

- How do we search for a value in a tree, starting at root?
  - Can do this both iteratively and recursively, contrast to printing which is very difficult to do iteratively
  - How is insertion similar to search?

- What is complexity of print? Of insertion?
  - Is there a worst case for trees?
  - Do we use best case? Worst case? Average case?

- How do we define worst and average cases
  - For trees? For vectors? For linked lists? For arrays of linked-lists?

See SetTiming code

- What about ISimpleSet interface
  - How does this compare to java.util?
  - Why are we looking at this, what about Java source?

- How would we implement most simply?
  - What are complexity repercussions: add, contains
  - What about iterating?

- What would linked list get us? Scenarios where better?
  - Consider N adds and M contains operations
  - Move to front heuristic?

What does contains look like?

```java
public boolean contains(E element) {
    return myList.indexOf(element) >= 0;
}
```

```java
public boolean contains(E element){
    return contains(myHead, element);
}
```

```java
private boolean contains(Node list, E element) {
    if (list == null) return false;
    if (list.info.equals(element)) return true;
    return contains(list.next,element);
}
```

- Why is there a private, helper method?
  - What will be different about Tree?
What does contains look like?

```java
public boolean contains(E element) {
    return contains(myRoot, element);
}
private boolean contains(TreeNode root, E element) {
    if (root == null) return false;
    if (list.info.equals(element)) return true;
    if (element.compareTo(root.info) <= 0) {
        return contains(root.left, element);
    } else {
        return contains(root.right, element);
    }
}
```

- What is recurrence? Complexity?
  - When good trees go bad, how can this happen?

What does insertion look like?

```java
Simple recursive insertion into tree (accessed by root)

```java
t = insert("foo", t);
```}
```java
public TreeNode insert(TreeNode t, String s) {
    if (t == null) t = new Tree(s, null, null);
    else if (s.compareTo(t.info) <= 0)
        t.left = insert(t.left, s);
    else t.right = insert(t.right, s);
    return t;
}
```

- Note: in each recursive call, the parameter t in the called clone is either the left or right pointer of some node in the original tree
  - Why is this important?
  - Why must the idiom `t = treeMethod(t, ...)` be used?

Implementing binary trees

- Trees can have many shapes: short/bushy, long/stringy
  - if height is h, number of nodes is between $h$ and $2^{h-1}$
  - single node tree: height = 1, if height = 3

- Java implementation, similar to doubly-linked list

```java
public class Tree {
    String info;
    TreeNode left;
    TreeNode right;
    TreeNode(String s, TreeNode llink, TreeNode rlink) {
        info = s; left = llink; right = rlink;
    }
}
```

Tree functions

- Compute height of a tree, what is complexity?

```java
int height(Tree root) {
    if (root == null) return 0;
    else {
        return 1 + Math.max(height(root.left), height(root.right));
    }
}
```

- Modify function to compute number of nodes in a tree, does complexity change?
  - What about computing number of leaf nodes?
**Tree traversals**

- Different traversals useful in different contexts
  - Inorder prints search tree in order
    - Visit left-subtree, process root, visit right-subtree
  - Preorder useful for reading/writing trees
    - Process root, visit left-subtree, visit right-subtree
  - Postorder useful for destroying trees
    - Visit left-subtree, visit right-subtree, process root

**Balanced Trees and Complexity**

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one

```java
boolean isBalanced(Tree root) {
    if (root == null) return true;
    return
    isBalanced(root.left) &&
    isBalanced(root.right) &&
    Math.abs(height(root.left) - height(root.right)) <= 1;
}
```

**What is complexity?**

- Assume trees are “balanced” in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis
- How to develop recurrence relation?
  - What is T(n)?
  - What other work is done?
- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify correctness

**Danny Hillis**

- The third culture consists of those scientists and other thinkers in the empirical world who, through their work and expository writing, are taking the place of the traditional intellectual in rendering visible the deeper meanings of our lives, redefining who and what we are.

(Wired 1998) And now we are beginning to depend on computers to help us evolve new computers that let us produce things of much greater complexity. Yet we don’t quite understand the process - it’s getting ahead of us. We’re now using programs to make much faster computers so the process can run much faster. That’s what’s so confusing – technologies are feeding back on themselves; we’re taking off. We’re at that point analogous to when single-celled organisms were turning into multicelled organisms. We are amoebas and we can’t figure out what the hell this thing is that we’re creating.