Types of Structures

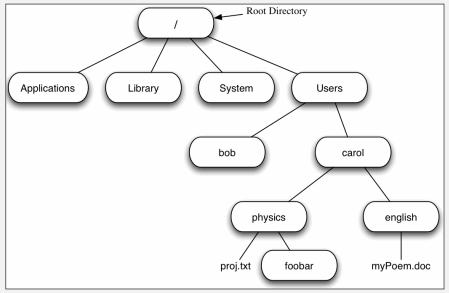
So far we've seen linear collections – there's a first, second, third thing.

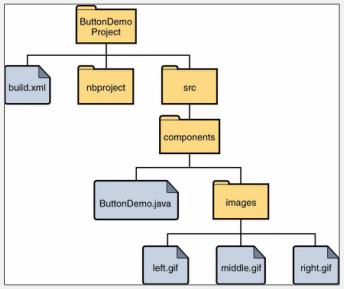
- even if we may not have direct access to elements by index

For hierarchical structures, we use *trees*.

Some Applications of Trees

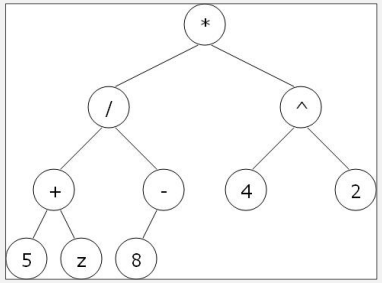
- directory hierarchy
  - print out listing of the whole structure (or a portion)
  - find a file or directory by name

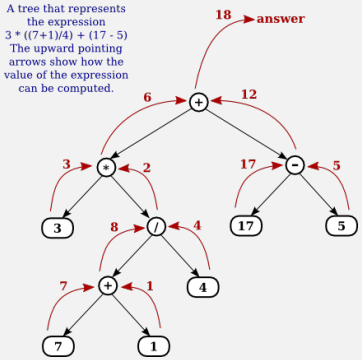




Some Applications of Trees

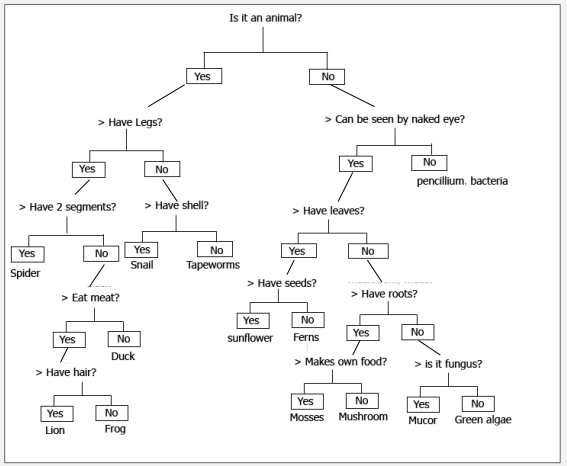
- expression tree
  - evaluate tree





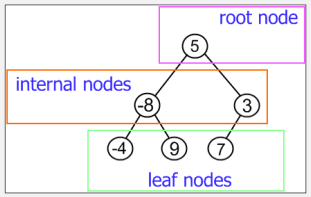
Some Applications of Trees

- decision tree
  - identify something by navigating through the tree to a leaf



Tree Terminology

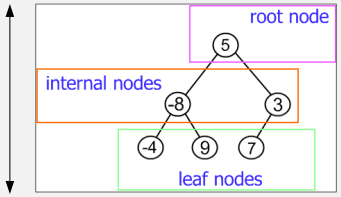
- *root*
- *child*/*parent*
- *ancestor*/*descendant*
  - parent, parent's parent, etc / children, children's children, etc
- *sibling*
  - two nodes with the same parent
- *leaf* (or *external node*)
  - node with no children
- *internal node*
  - node with at least one child
- *subtree*
  - tree whose root is the child of another node



Tree Terminology

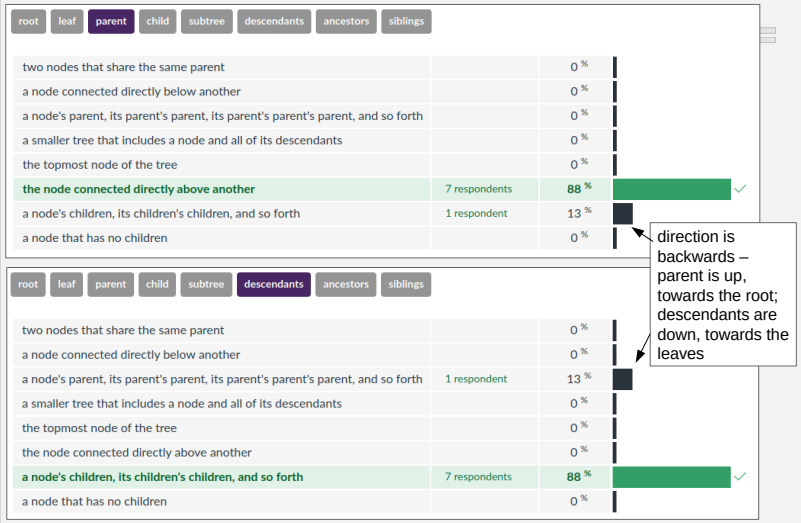
leaves

roots



note that data structure trees are upside down compared to real trees – root at the top, leaves at the bottom

https://commons.wikimedia.org/wiki/File:Tree_Transparent_Background.png



root | leaf | parent | child | subtree | descendants | ancestors | siblings

two nodes that share the same parent		0 %
a node connected directly below another		0 %
a node's parent, its parent's parent, its parent's parent's parent, and so forth		0 %
a smaller tree that includes a node and all of its descendants	7 respondents	**88 %** ✓
the topmost node of the tree		0 %
the node connected directly above another		0 %
a node's children, its children's children, and so forth	1 respondent	13 %
a node that has no children		0 %

a subtree also includes the root of the subtree (the node itself)

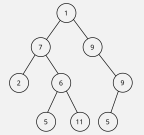
root | leaf | parent | child | subtree | descendants | **ancestors** | siblings

two nodes that share the same parent		0 %
a node connected directly below another		0 %
a node's parent, its parent's parent, its parent's parent's parent, and so forth	7 respondents	**88 %** ✓
a smaller tree that includes a node and all of its descendants		0 %
the topmost node of the tree		0 %
the node connected directly above another	1 respondent	13 %
a node's children, its children's children, and so forth		0 %
a node that has no children		0 %

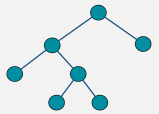
ancestors include the parent, but also more than that

Tree Terminology

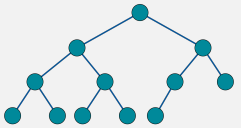
- *binary tree*
  - every node has at most two children



- *proper* binary tree
  - every internal (non-leaf) node has exactly two children



- *complete* binary tree
  - every level (except possibly the last) is completely full
  - the nodes in the last level are as far left as possible (no gaps)
  - sometimes refers to a binary tree where the last level is completely full



Proper Binary Trees

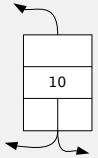
Why (proper) binary trees?

- binary trees are a very common type of tree
- proper simplifies the implementation and is not limiting
  - other binary trees can be realized with *dummy leaves* (no element is stored there) – utilize only the internal nodes
- implementation ideas can easily be extended to general trees
- can implement general trees in terms of binary trees

Implementing Binary Trees

(links for parent pointers not drawn)

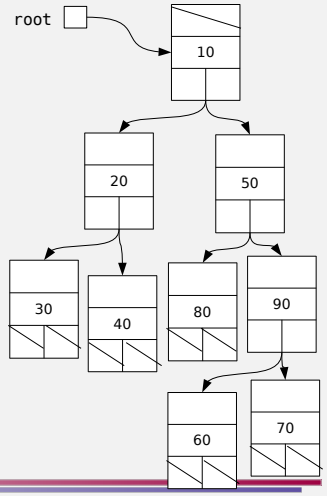
- as with linked lists, we need to first define a tree node type
  - element
  - left child, right child
  - parent may be omitted if there's no need to move up the tree



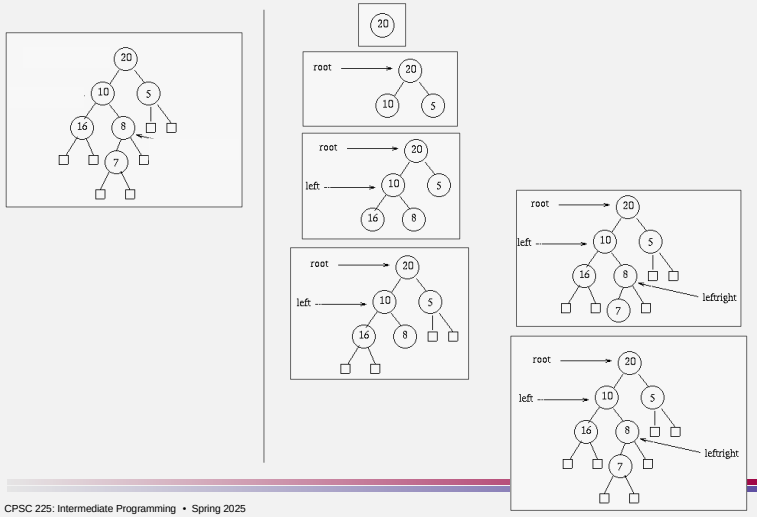

```
class TreeNode {
   int item;        // The data in this node.
   TreeNode left;   // Pointer to the left subtree.
   TreeNode right;  // Pointer to the right subtree.
}
```

private instance variables with public constructor(s), getters, setters is preferred unless the `TreeNode` class is purely a helper (inner) class

- the tree itself is a root pointer
  - similar to head for a linked list



Building Trees



Working With Trees – Patterns

Three main ways of moving through trees:

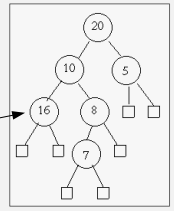
- moving up the tree
  - loop with current node being updated to parent until the root is reached
- moving down the tree, interested in only one child
  - loop with current node being updated to child until leaf is reached
- moving down the tree, interested in both children
  - recursion (left child and right child), with leaf as base case
  - if all nodes are visited, this is known as a *traversal*

(note – these are general patterns; modify specifics like starting or ending point as needed for a particular task)

Working With Trees – Patterns

```
/**
 * Compute the depth of the specified node. The depth corresponds to the
 * number of ancestors - the root has depth 0 (no ancestors), the children of
 * the root have depth 1 (each has one ancestor, the root of the tree), the
 * grandchildren of the root have depth 2 (each has 2 ancestors, the parent
 * and the parent's parent), and so forth.
 *
 * @param node
 *          the node
 * @return the depth of the node
 */
public static int getDepth ( TreeNode node ) {
```

node →

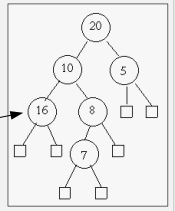


- moving up the tree
  - loop with current node being updated to parent until the root is reached

Working With Trees – Patterns

```
/**
 * Return the leftmost internal node in the tree.
 *
 * @param root
 *          the root of the tree
 * @return the leftmost internal node
 */
public static TreeNode findLeftmost ( TreeNode root ) {
```

leftmost →



- moving down the tree, interested in only one child
  - loop with current node being updated to child until leaf is reached