Trees

Last time: recursion

- In the last lecture, we learned about recursion & divide-and-conquer
  - Split the problem into smaller parts
  - Solve each of the smaller parts separately: easier to code & understand!
- Apply these techniques to storing data so that it is
  - Ordered
  - Easy and efficient to find
- List-type structures don't do *both*
  - Lists & arrays: ordered, but lookup is slow
- We want a structure that can do both!

Quickly finding a particular item…

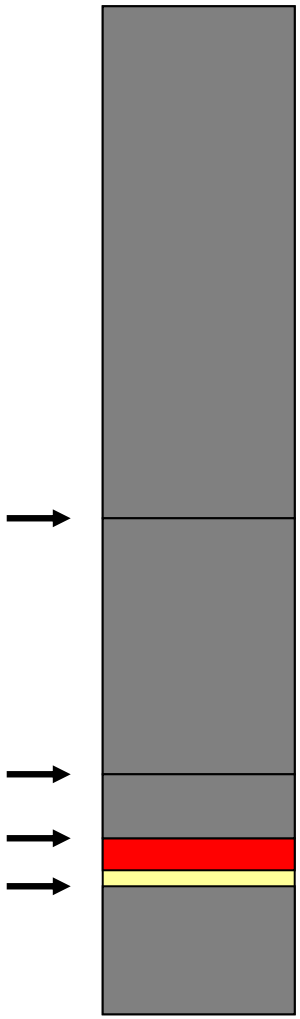
- Problem: in a class of *n* students, who has the *m*th best grade?
- Use a (sorted) linked list?
  - Easy to find: count *m* links from the start
  - Difficult to insert: must search along the list to find the correct insertion point
- Use an array?
  - Same kinds of advantages and disadvantages as linked list

What if we only have the value?

- Rather than find the *m*th best grade, find the student whose grade is 77
  - Can't just count *m* items any more!
  - Must scan the list / array until we find the correct student
- A better way: binary search

Binary Search

- Take a sorted array of values
- While (item not found)
  - "Guess" the item in the middle of the array
  - See if the desired item is above or below the guess
  - Narrow down the search area by half
- This works in $\log_2(N)$ tries on an array with N values
- Much faster than simply scanning



Binary search

- Similar to recursion
  - Problem split in half at each step
  - Main difference: ignore the half where the value ***isn't***
- Recursion doesn't usually save time
  - Easier to program, though
- Binary search saves time!
  - Rule out half of the remaining values at each step
  - Like recursion where we ignore half of the problem each time we recurse

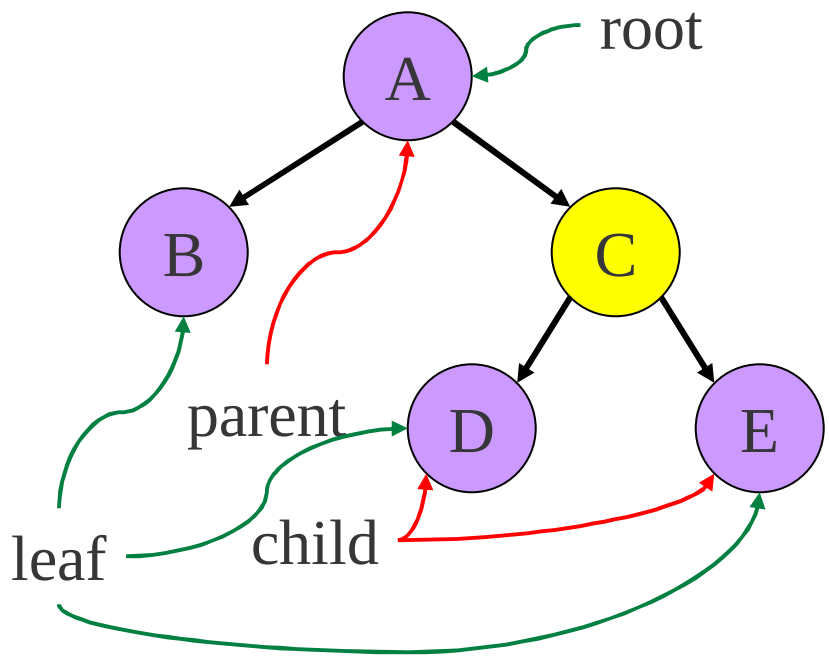
```
int bsearch (int values[],
             int findThis) {
 int range = values.length;
 int base = 0;
 int mid;
 while (range > 1) {
  range = (range+1)/2;
   mid = base+range;
  if (findThis > values[mid]) {
   base = mid;
  } else if (findThis==values[mid]){
   break;
  }
 }
 if (values[mid]==findThis) {
  return (mid);
 } else {
  return (-1);
 }
}
```

Binary search is great, but…

- Binary search works well with arrays
  - Easy to find element *n* in constant time
  - Difficult to insert things into the middle of the array
- Binary search doesn't work well with linked lists
  - Can't find element *n* in constant time: long lists => long time to find elements
  - Easy to insert and delete things in the middle
- Modify linked lists to make searching easier?
  - Keep references into the middle of the list (1/4, 1/2, 3/4, or similar)?
    - Good idea, but doesn't scale that well
    - Must recreate shortcuts when things are inserted or deleted
  - Create a new structure that uses links but is still easy to do binary search on?

Solution: *trees*

- A *tree* is a linked data structure where nodes may have more than one "next"
- Terms
  - "next" of a node is its *child*
  - "prev" of a node is its *parent*
  - Base of the tree is the *root*
  - Nodes along path to root are *ancestors*
  - Nodes "below" this one are *descendants*
  - Nodes with no children are *leaf* nodes
- Binary tree: tree in which each node has at most two children



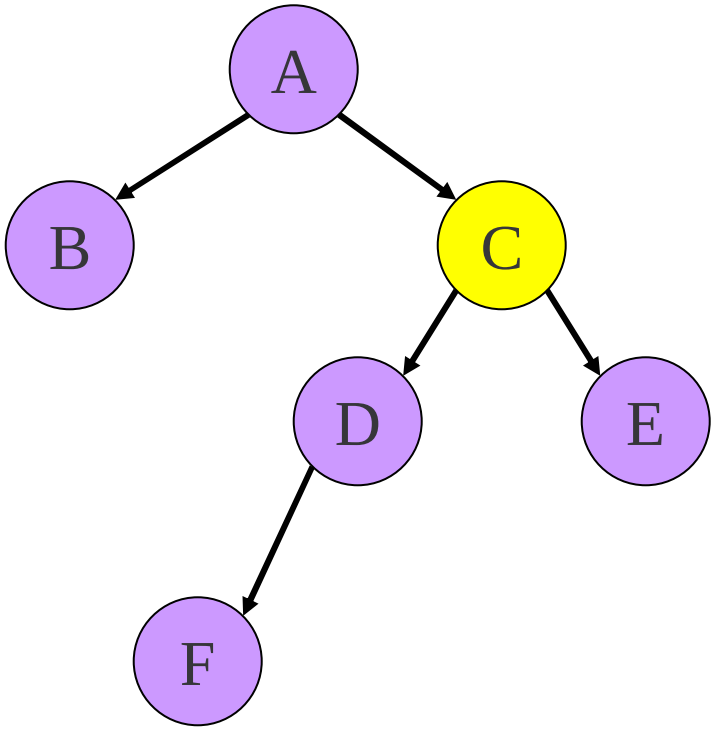

```
Class BTNode {
  Object item;
  BTNode left;
  BTNode right;
  BTNode parent;
}
```

Why use trees?

- Advantages of linked lists
  - Insert or delete anywhere with ease
  - Grow to any size
- Advantages of arrays
  - Easy to do binary search
  - Easy to keep sorted
- And, lookup can be done quickly if the tree is sorted
- Disadvantages?
  - Overhead: three references per node in the tree
  - It's easy to have trees grow the wrong way…

More tree terms

- Note: subtree can start at any node
  - There's a subtree rooted at C!
  - Subtrees follow same rules as trees
- A tree's *height* is the **largest** number of nodes from root to leaf
  - Height of the tree on the right is 4 (A->C->D->F)
- *Balanced* binary tree
  - For each node, the height of the left and right subtree differ by at most 1
  - This tree is not balanced!
- *Full* tree
  - No missing nodes
  - For all nodes, height of left and right subtree are equal



Classes used in building binary trees

- As with linked lists, two classes in binary trees
  - TreeNode: an individual node in the tree
  - BinaryTree: a subtree rooted at a particular TreeNode
- TreeNodes support the usual operations
  - TreeNode (Object newItem)
  - TreeNode (Object newItem, TreeNode lt, TreeNode rt)
  - Object getItem()
  - void setItem (Object newItem)
  - TreeNode getLeft/Right()
  - TreeNode setLeft/Right (TreeNode left)
  - Note: this implementation doesn't have "up" pointers in each node that point to the node's parent
- These operations are straightforward
  - Similar to operations in linked lists

Methods to build binary trees

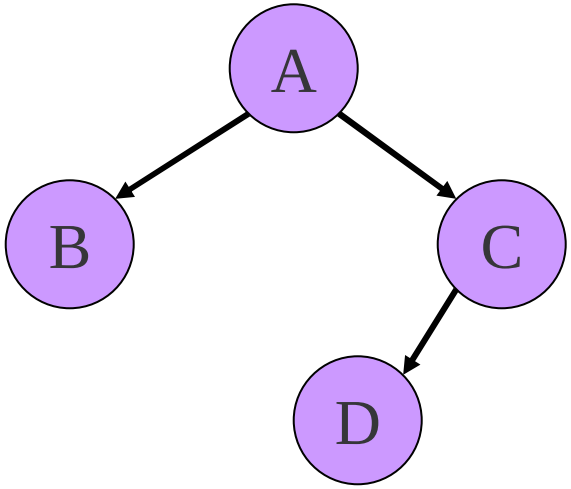
- Constructors
  - BinaryTree(): creates an empty tree
  - BinaryTree(Object rootItem): creates a tree with a root
  - BinaryTree(Object root, BinaryTree lt, BinaryTree rt): creates a tree with a root and left & right subtrees
- Attach things to the tree (root must already exist)
  - attachLeft/Right (Object newItem): attach an object to the left or right of the root
  - attachLeft/RightSubtree (BinaryTree tree): attach an entire tree to the root
  - attachLeft() could be done by creating a new subtree and attaching it with attachLeftSubtree()…
- Exceptions thrown for
  - Non-existent root
  - Trying to attach something on top of an existing subtree

Methods to take trees apart

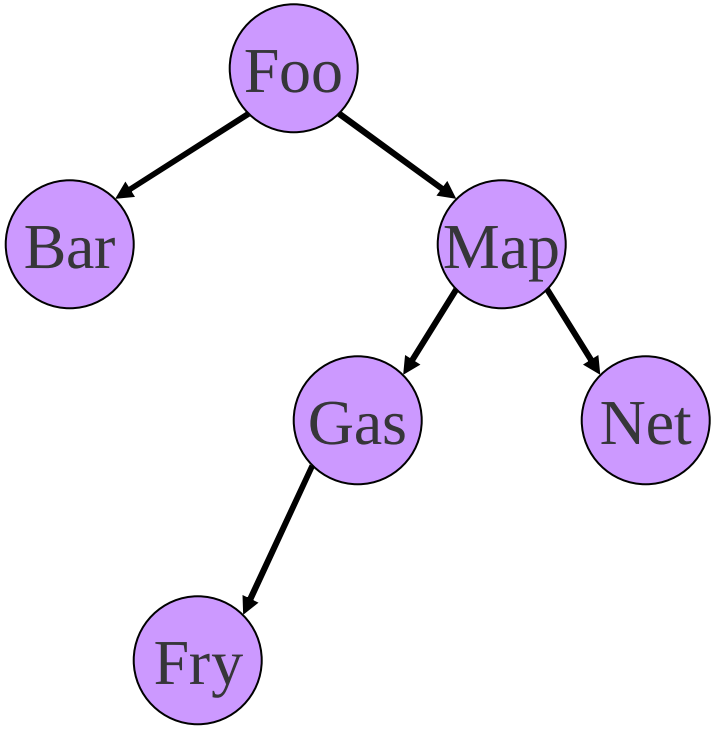
- Often, necessary to take a tree apart
  - Make it better (more balanced)
  - Delete an item
- This can be done with
  - BinaryTree detachLeft/RightSubtree (): detaches a subtree from the root, and returns it
  - Left or right node set to **null**
- Informational methods
  - Object getRootItem ()
  - void setRootItem (Object newItem)
  - boolean isEmpty()
- How can we use these methods to build up a tree?

Create a simple tree

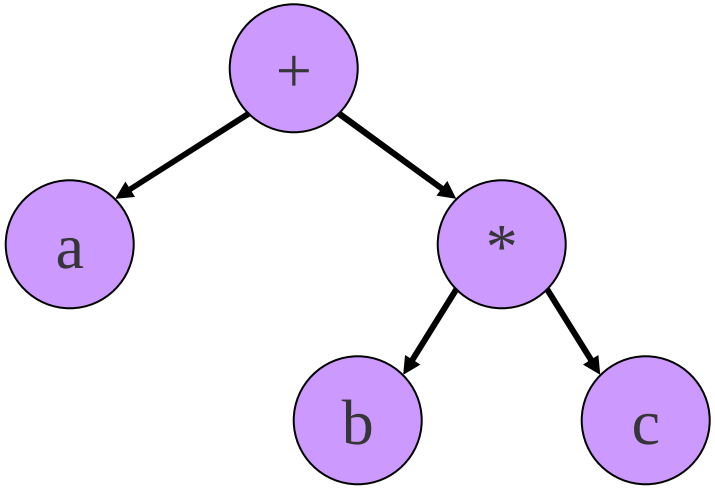
```
bt = BinaryTree ("A");
bt.attachLeft ("B");
ct = BinaryTree ("C");
ct.attachLeft ("D");
bt.attachRightSubtree (ct);
```



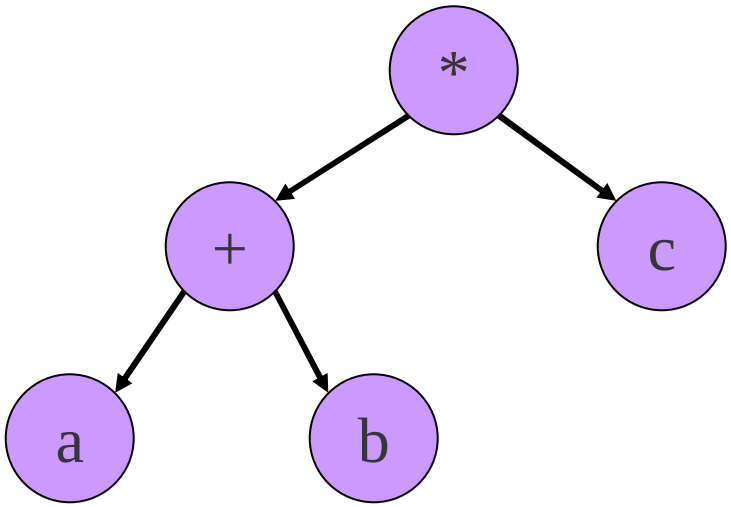
Example: Words Stored Lexicographically



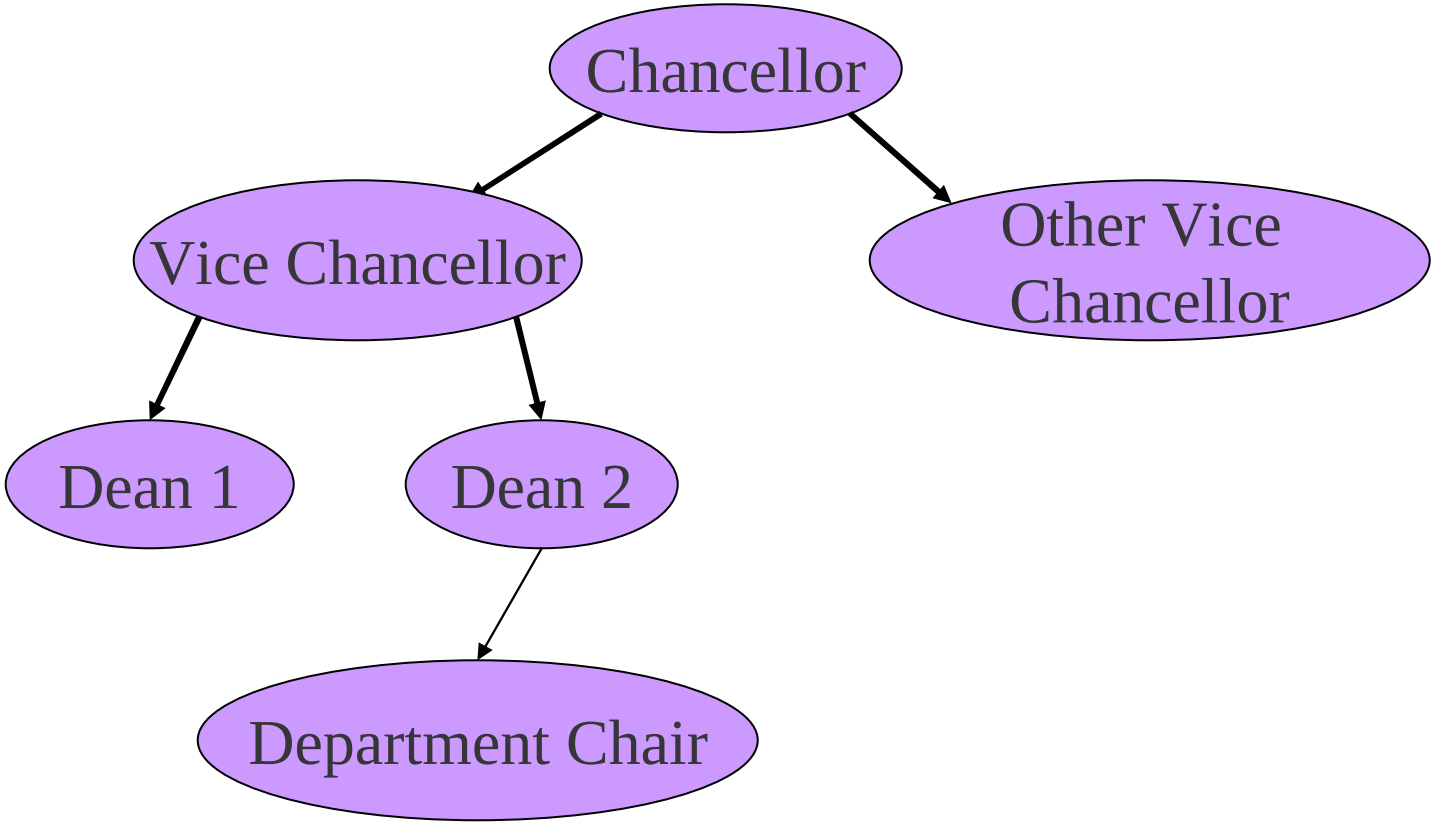
Example: Equation a + (b*c)



Example: Equation (a+b) * c

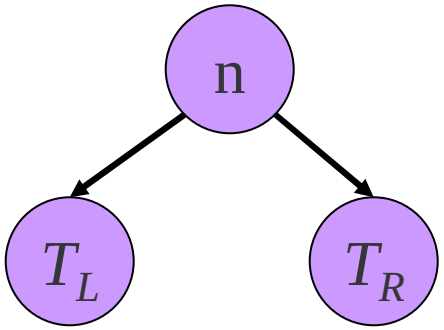


Example: Organization Chart



A Recursive Definition of Binary Trees

- A tree T is a binary tree if either
  - T has no nodes, or
  - T is of the form



  - Where n is a node, and T_L and T_R are both binary trees.

Height of a Tree

- Level of a node
  - If *n* is the root of *T*, it is at level 1
  - If *n* is not the root, its level is 1 higher than that of its parent
- Height of a tree
  - If *T* is empty, its height is 0
  - If *T* is not empty, its height is equal to the maximum level of its nodes
- Recursive definition of height
  - If T is empty, its height is 0
  - If T is nonempty, its height is equal to 1 + the height of its tallest subtree: $\text{height}(T) = 1+\max\{\text{height}(T_L),\text{height}(T_R)\}$

Types of Binary Trees

- **Full binary tree**
  - All nodes at level k<h have two children each
  - All leaves are at the same level
- **Complete binary tree**
  - All nodes at level k<h-1 have two children each, and
  - When a node has children, all nodes to its left have children, and
  - When a node has one child, it is a left child
- **Balanced binary tree**
  - The height of each node's subtrees differ by at most 1.

Binary Tree ADT

- Basic Operations
  - Create an empty binary tree
  - Create a one-node binary tree, given an item
  - Remove all nodes from a binary tree
  - Determine whether a binary tree is empty
  - Determine what data is at a binary tree's root
- General Operations
  - Create a binary tree given an item, and left/right subtrees
  - Set root item
  - Attach left or right item
  - Attach left/right subtree
  - Detach left/right subtree

Binary Tree Code

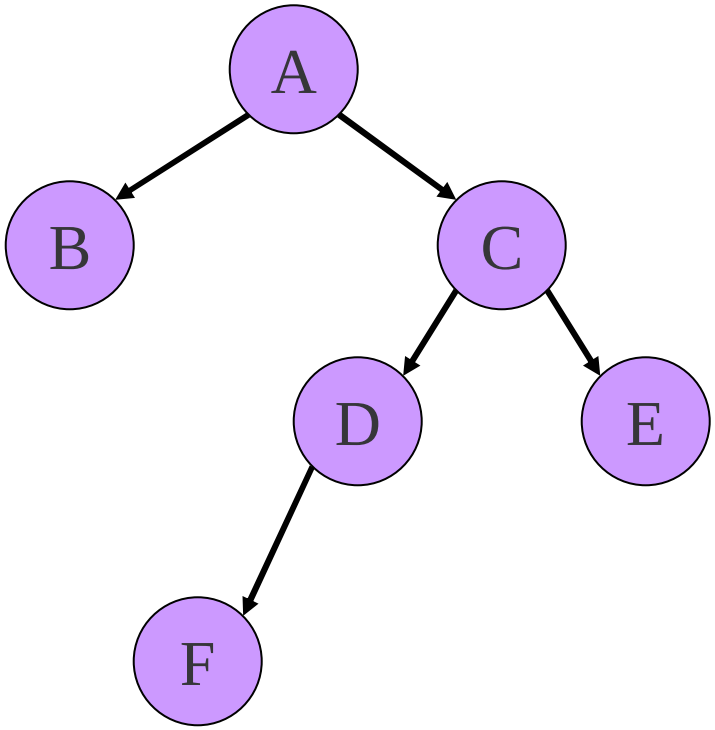
- See examples

Iterators

- Provide a general way of traversing a tree
  - Can't use internal types like TreeNode!
  - Instead, use an *iterator*
- An *iterator* is a class whose purpose is to allow other structures to be "read" in order
  - Example (sort of): Tokenizer
- An *iterator* supports a set of methods
  - Constructor: specifies the data structure to iterate over
  - hasNext(): **true** if there is another object to iterate to
  - next(): returns the next object in the traversal

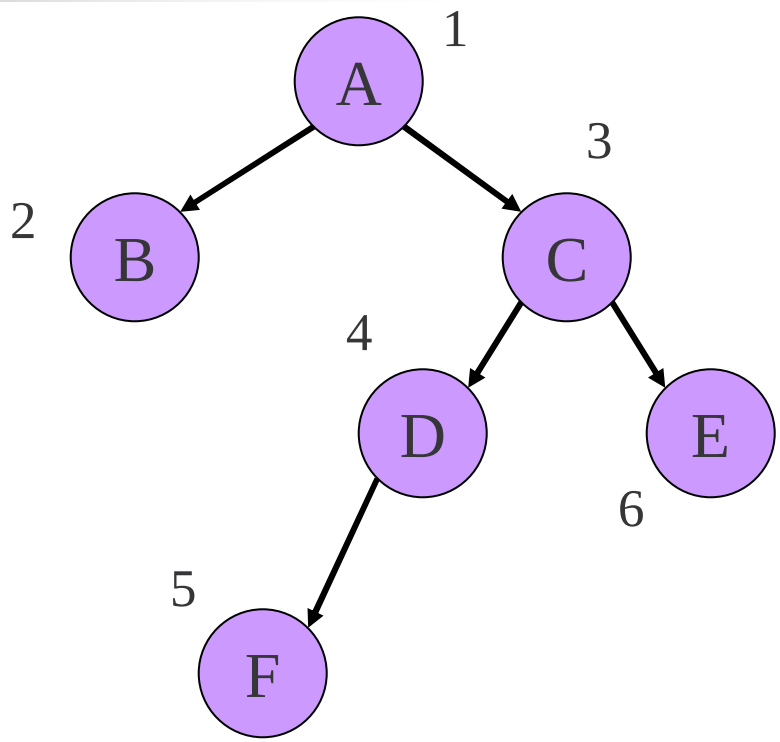
Traversing a binary tree

- Trees can be traversed in three orders
  - Pre-order: root, L, R
    - A, B, C, D, F, E
  - In-order: L, root, R
    - B, A, F, D, C, E
  - Post-order: L, R, root
    - B, F, D, E, C, A
- Order chosen depends on
  - What the tree is being used for
  - What the traversal is supposed to accomplish
- Traversal is done recursively!
  - Treat L, R as trees in their own right
  - Recursively visit them



Pre-order traversal

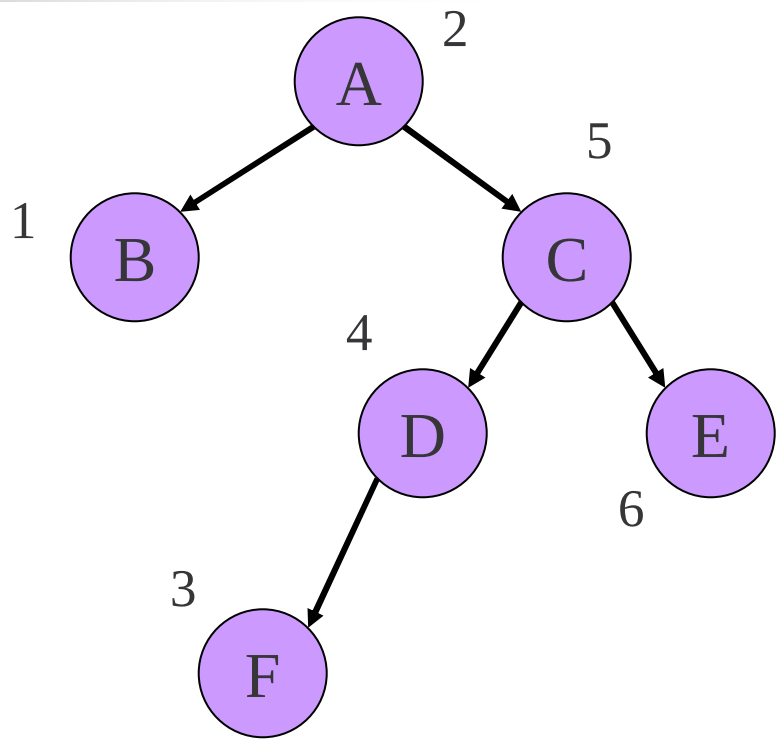
- Visit nodes in this order
  - Root node
  - Left subtree
  - Right subtree
- Recursive visit
- Perform operation at the leaf
  - Printing in this example




```
void preorder () {
  System.out.println (root.value);
  if (left != null) {
    left.preorder ();
  }
  if (right != null) {
    right.preorder ();
  }
}
```

In-order traversal

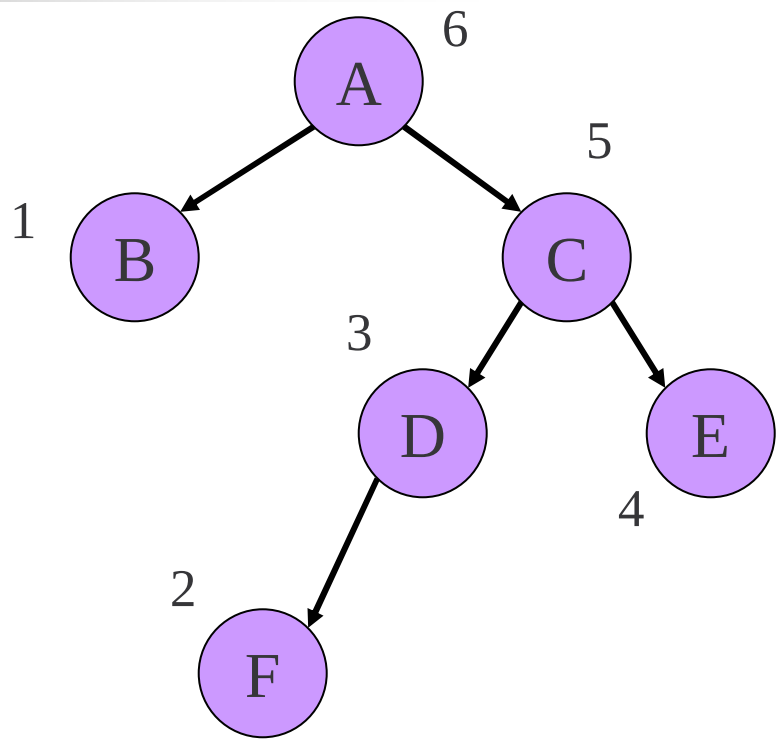
- Visit nodes in this order
  - Left subtree
  - Root node
  - Right subtree
- Recursive visit
- Perform operation at the leaf
  - Printing in this example




```
void inorder () {
  if (left != null) {
    left.inorder ();
  }
  System.out.println (root.value);
  if (right != null) {
    right.inorder ();
  }
}
```

Post-order traversal

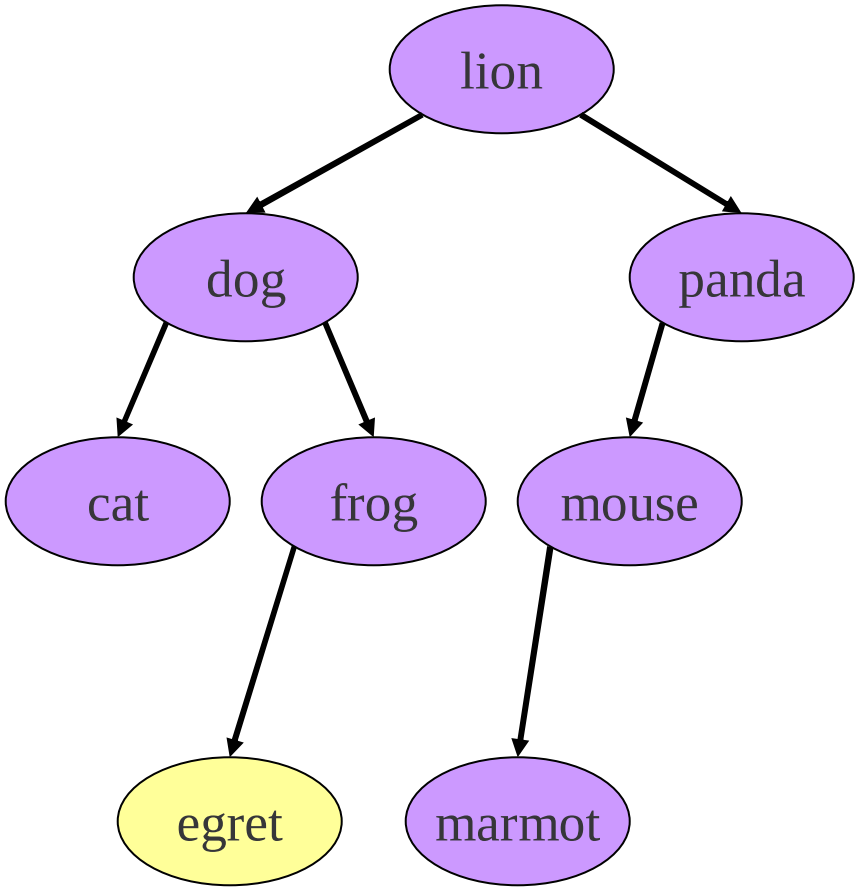
- Visit nodes in this order
  - Left subtree
  - Right subtree
  - Root node
- Recursive visit
- Perform operation at the leaf
  - Printing in this example




```
void postorder () {
  if (left != null) {
    left.postorder ();
  }
  if (right != null) {
    right.postorder ();
  }
  System.out.println (root.value);
}
```

Binary search tree

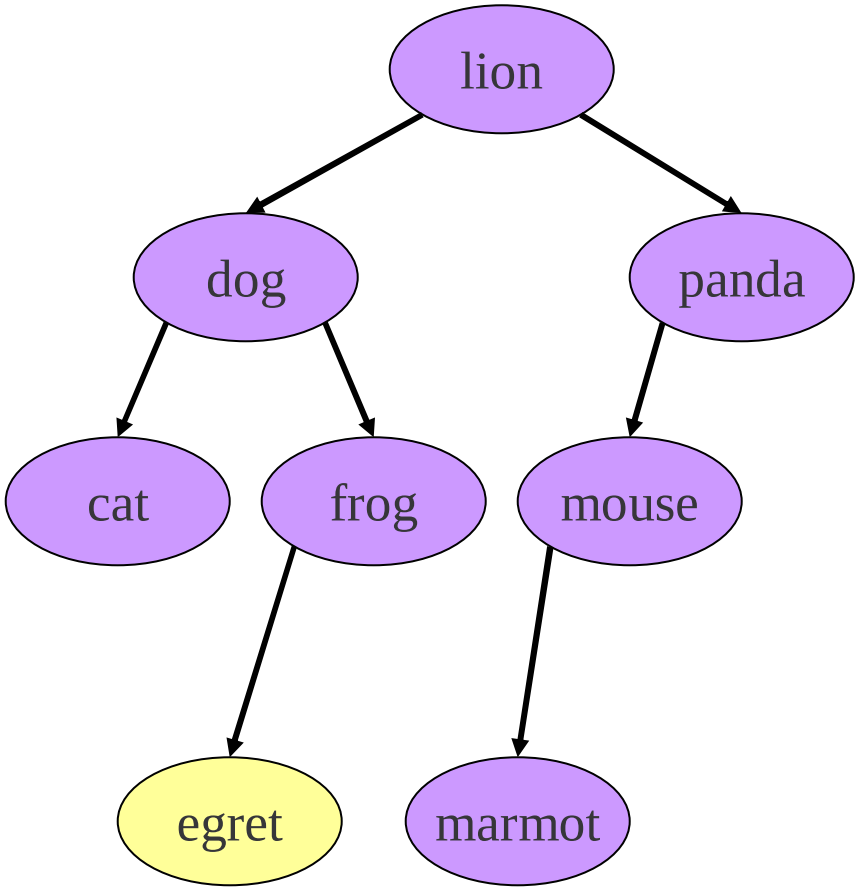
- Rule 1: left child is *less* than parent
- Rule 2: right child is *greater* than parent
- Insert a new node by
  - Following the links down
  - Attaching the new node where it "should" go
- Result:
  - All nodes in the left subtree are less than the root!
  - All nodes in the right subtree are greater than the root



Binary search tree

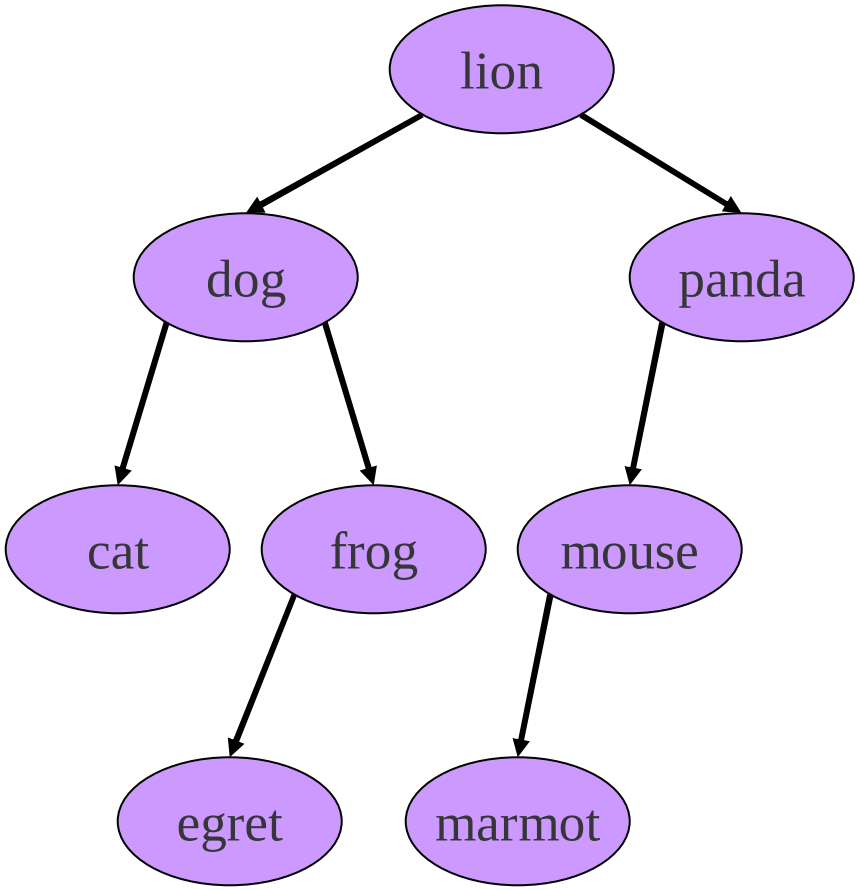
- The value stored in each node for comparison is a "key"
- Examples:
  - Directories
  - List of students
  - …

- Recursive definition of binary search tree
  - For each node *n*,
  - *n*'s key is greater than every key in T_L
  - *n*'s key is less than every key in T_R
  - T_L and T_R are binary search trees.



Searching in a binary search tree

- Searching is similar to insertion
  - Go left if less
  - Go right if more
- Can be done recursively
- Can be done non-recursively
  - Loop, setting appropriate subtree to root each time

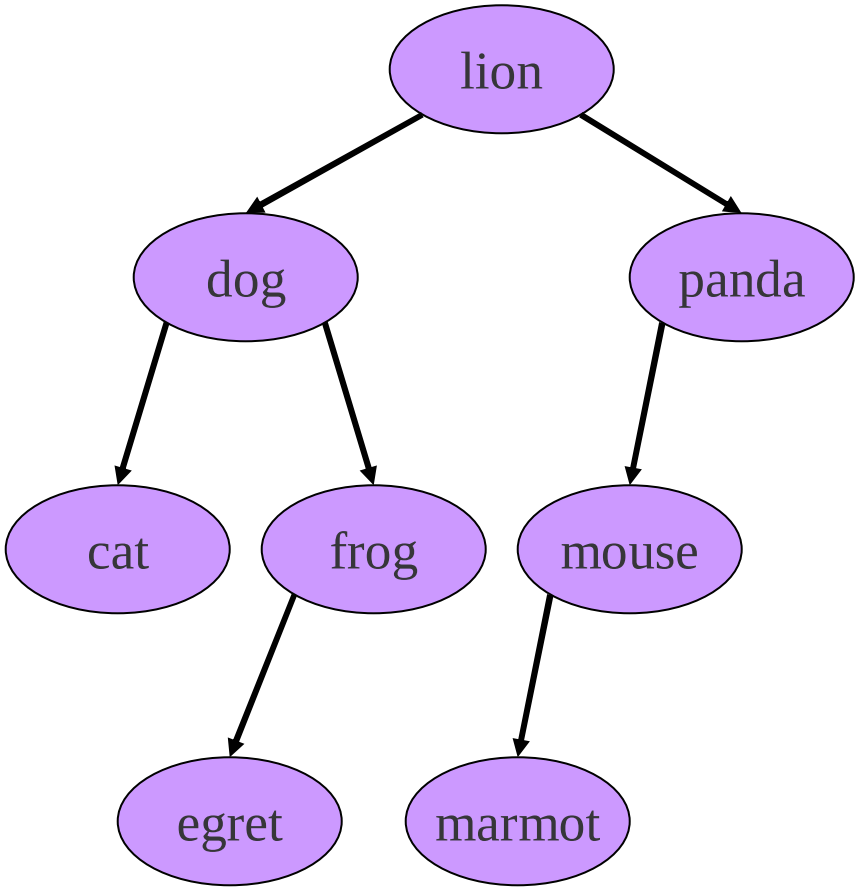


Recursively Searching a Binary Search Tree

```
search(binary search tree, searchKey) {
    if(empty)
        // not found
    else if(key = = searchKey)
        // found
    else if(key > searchKey)
        search(left subtree, searchKey);
    else
        search(right subtree, searchKey);
}
```

Accessing a BST in sorted order

- How can we print this tree in sorted order?
- Starting at root, we know
  - All nodes in left subtree are "less" than root
  - All nodes in right subtree are "greater" than root
  - Left & right subtrees can both be printed in sorted order
- Solution: in-order traversal!
  - Print all nodes less than root in sorted order
  - Print root
  - Print all nodes greater than root in sorted order

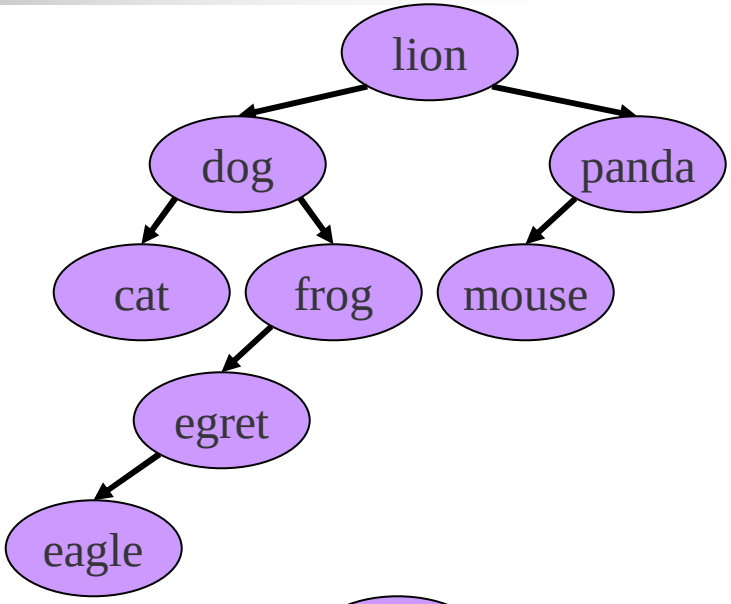


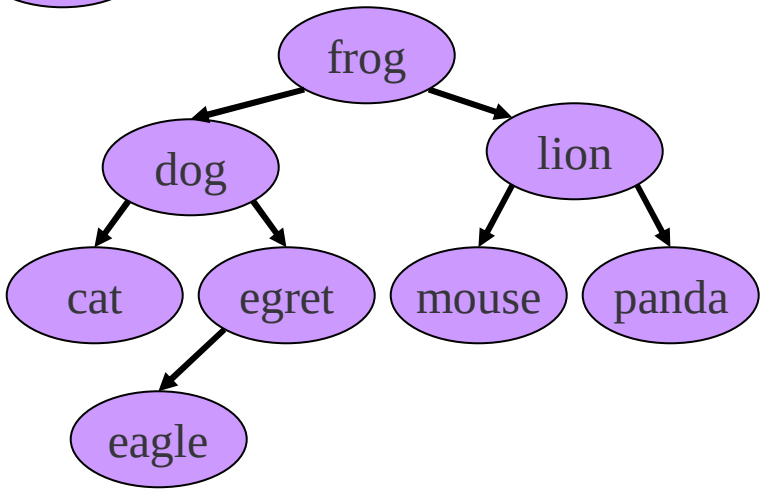
Binary Search Tree Operations

- Insert an item (O(logn))
  - Search until null is reached – place the item there
- Delete an item with a given search key (O(logn))
  - Three cases
  - N has no children – easy, set parent's reference to null
  - N has one child – like a linked list
  - N has two children – replace with inorder successor
- Retrieve an item with a given search key (O(logn))
  - Binary search (yay)
- Traverse the items in some order (O(n))

Balancing trees (overview)

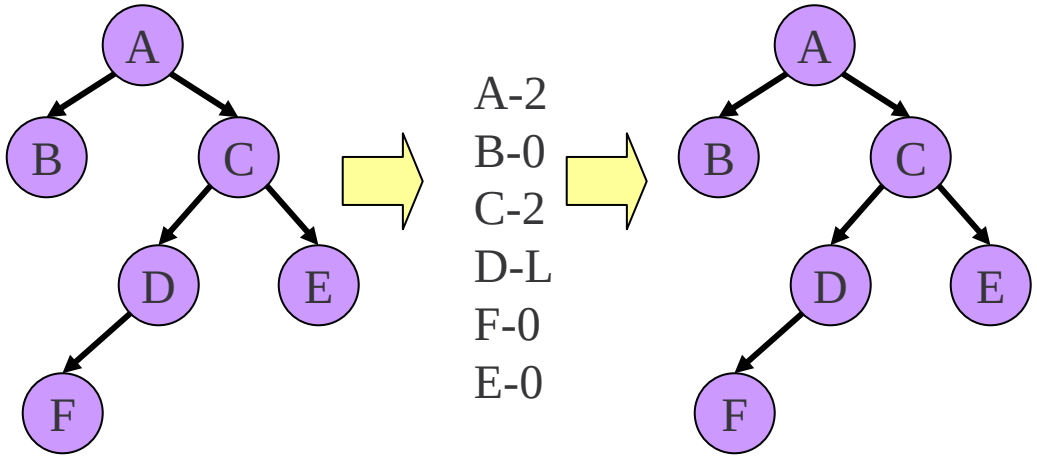
- Binary trees are most effective when they are *balanced*
  - Maximum depth of any node is no more than 1 greater than minimum depth of any node
- How can we balance a tree?
  - Preserve ordering rules!
  - Rearrange tree to fix heights
- Ensure tree is balanced after each insertion
  - Prevent it from getting too far out of balance
- Use a special kind of binary tree: 2-3 tree
  - General idea: keep the tree balanced at every step
  - Details beyond the scope of this class





Saving and restoring binary trees

- What if we want to save a tree?
  - Print it out in such a way that we can restore it later
  - Provide sufficient information to reconstruct an exact copy
- Use pre-order traversal
  - Print information about each node saying whether it has left, right, neither or both subtrees
  - Print the contents of the node
- Reconstruct recursively




```
Tree buildTree () {
  read value, status
  t = new Tree (value);
  if (status == 2  ||
      status == L) {
    t.attachL(buildTree());
  }
  if (status == 2 ||
      status == R) {
    t.attachR (buildTree());
  }
  return (t);
}
```

Why save and restore trees?

- Helpful when one program creates a tree that another wants to use
  - Placement within the tree could be important!
  - Might not simply be sorted…

General Trees

- What if you want to have any number of children in a tree?
  - What data structure would you use to store the children?