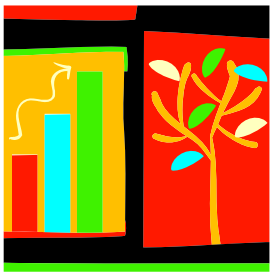


Trees

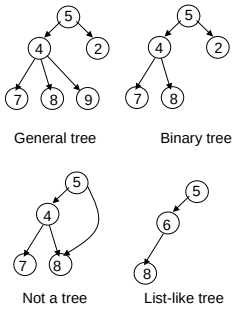
Lecture 7
CS211 – Summer 2007

Announcements

- Office Hours???
- Assignment 2 posted
  - Due Thursday

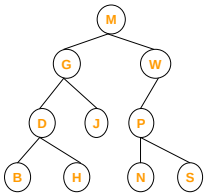
Tree Overview

- *Tree*: recursive data structure (similar to list)
  - Each cell may have two or more successors (children)
  - Each cell has at most one predecessor (parent)
    - Distinguished cell called *root* has no parent
  - All cells are reachable from *root*
- *Binary tree*: tree in which each cell can have at most two children: a left child and a right child



Tree Terminology

- M is the *root* of this tree
- G is the *root* of the *left subtree* of M
- B, H, J, N, and S are *leaves*
- N is the *left child* of P; S is the *right child*
- P is the *parent* of N
- M and G are *ancestors* of D
- P, N, and S are *descendents* of W
- Node J is at *depth* 2 (i.e., *depth* = length of path from root)
- Node W is at *height* 2 (i.e., *height* = length of longest path from leaf)
- A collection of several trees is called a ...?



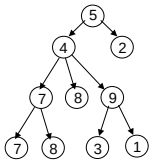
Class for Binary Tree Cells

```
class TreeCell {
    private Object datum;
    private TreeCell left;
    private TreeCell right;

    public TreeCell (Object x) {datum = x;}
    public TreeCell (Object x, TreeCell l, TreeCell r) {
       datum = x;
       left = l;
       right = r;
    }
    more methods: getDatum, setDatum,
    getLeft, setLeft, getRight, setRight
}
```

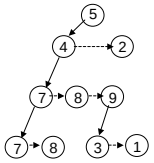

Class for General Trees

```
class GTreeCell {
  private Object datum;
  private GTreeCell left;
  private GTreeCell sibling;
  appropriate get and set
  methods□
}
```

- Parent node points directly only to its leftmost child
- Leftmost child has pointer to next sibling, which points to next sibling, etc.



General tree



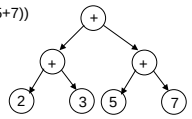
Tree represented using GTreeCell

Applications of Trees

- Most languages (natural and computer) have a recursive, hierarchical structure
- This structure is *implicit* in ordinary textual representation
- Recursive structure can be made *explicit* by representing sentences in the language as trees: Abstract Syntax Trees (ASTs)
- ASTs are easier to optimize, generate code from, etc. than textual representation
- A parser converts textual representations to AST

Example

- Expression grammar:
  - E → integer
  - E → (E + E)
- In textual representation
  - Parentheses show hierarchical structure
- In tree representation
  - Hierarchy is explicit in the structure of the tree

Text	AST Representation
-34	
(2 + 3)	
((2+3) + (5+7))	

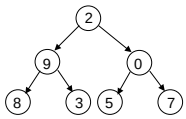
Recursion on trees

- Recursive methods can be written to operate on trees in an obvious way
- In most problems
  - Base case
    - empty tree
    - leaf node
  - Recursive case
    - solve problem on left and right subtrees
    - put solutions together to compute solution for full tree

Searching in a Binary Tree

```
public static boolean treeSearch(Object x, TreeNode node)
{



}
```

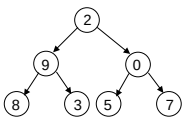
- Analog of linear search in lists: given tree and an object, find out if object is stored in tree
- Easy to write recursively, harder to write iteratively



Searching in a Binary Tree

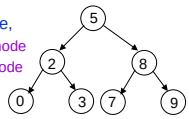
```
public static boolean treeSearch(Object x, TreeNode node)
{
    if (node == null) return false;
    if (node.datum.equals(x)) return true;
    return treeSearch(x, node.lchild) ||
           treeSearch(x, node.rchild);
}
```

- Analog of linear search in lists: given tree and an object, find out if object is stored in tree
- Easy to write recursively, harder to write iteratively



Binary Search Tree (BST)

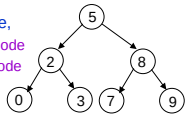
- If the tree data are *ordered* – in any subtree,
  - All *left* descendents of node come *before* node
  - All *right* descendents of node come *after* node
- This makes it *much* faster to search




```
public static boolean treeSearch (Object x, TreeNode node)
{



}
```

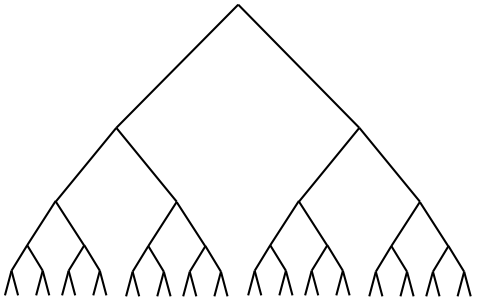
Binary Search Tree (BST)

- If the tree data are *ordered* – in any subtree,
  - All *left* descendents of node come *before* node
  - All *right* descendents of node come *after* node
- This makes it *much* faster to search

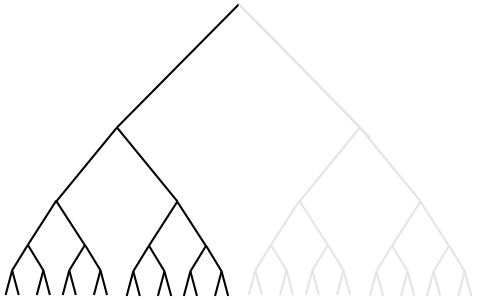



```
public static boolean treeSearch (Object x, TreeNode node)
{
    if (node == null) return false;
    if (node.datum.equals(x)) return true;
    if (node.datum.compareTo(x) > 0)
         return treeSearch(x, node.lchild);
    else return treeSearch(x, node.rchild);
}
```

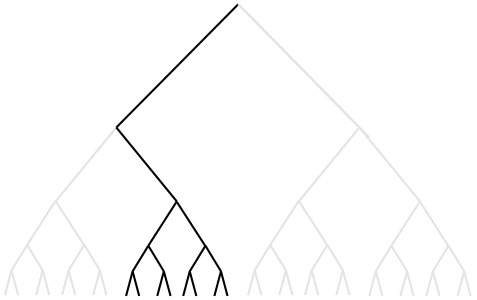

Binary Search Tree



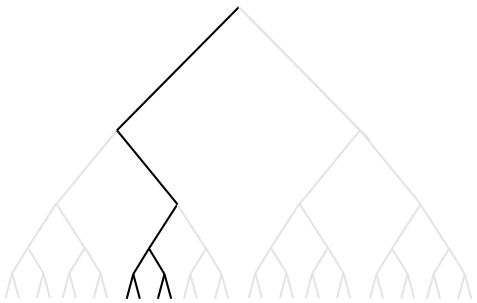
Binary Search Tree



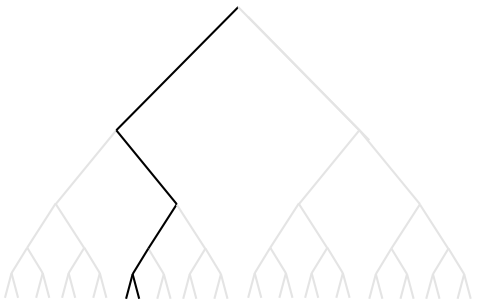
Binary Search Tree



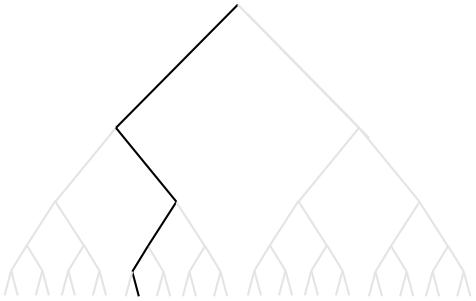
Binary Search Tree



Binary Search Tree

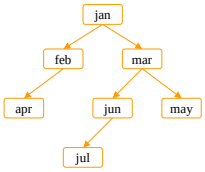


Binary Search Tree



Building a BST

- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree
- This can be done using either recursion or iteration
- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*



TreeNode

- This version is for a tree of Strings

```
class TreeNode {
   String datum;            //data stored at a node
   TreeNode lchild, rchild; //left and right children

   public TreeNode(String datum) { //constructor
      this.datum = datum;
      lchild = null;
      rchild = null;
   }
}
```


TreeNode

- ...but you can define a generic one

```
class TreeNode<T> {
   T datum;                  //data stored at a node
   TreeNode<T> lchild, rchild; //children

   public TreeNode(T datum) { //constructor
      this.datum = datum;
      lchild = null;
      rchild = null;
   }
}
```

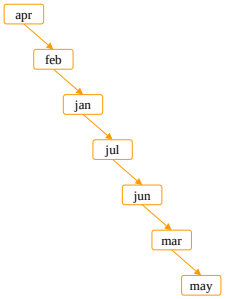
```
... new TreeNode<String>("hello") ...
```


BST Code

```
public class BST {
   TreeNode root;     // The root of the BST

   public BST() {
      root = null;
   }

   public void insert(String string) {
      root = insert(string, root);
   }

   private static TreeNode insert(String string, TreeNode node) {
      if (node == null) return new TreeNode(string);
      int compare = string.compareTo(node.datum);
      if (compare < 0) node.lchild = insert(string, node.lchild);
      else if (compare > 0) node.rchild = insert(string, node.rchild);
      return node;
   }
}
```

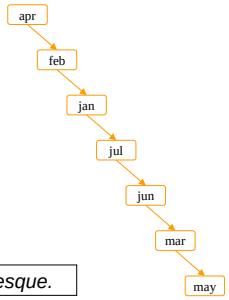

What Can Go Wrong?

- A BST makes searches very fast, *unless*…
  - Nodes are inserted in alphabetical order
  - In this case, we're basically building a linked list (with some extra wasted space for the `lchild` fields that aren't being used)
- BST works great if data arrives in random order



What Can Go Wrong?

- A BST makes searches very fast, *unless*…
  - Nodes are inserted in alphabetical order
  - In this case, we're basically building a linked list (with some extra wasted space for the **`lchild`** fields that aren't being used)
- BST works great if data arrives in random order



Beautiful BSTs are Rubenesque.

Printing Contents of BST

- Because of the ordering rules for a BST, it's easy to print the items in alphabetical order
  - Recursively print everything in the left subtree
  - Print the node
  - Recursively print everything in the right subtree

```
/**
* Show the contents of the BST in
* alphabetical order.
*/
public void show () {
   show(root);
   System.out.println();
}

private static void show(TreeNode node) {
   if (node == null) return;
   show(node.lchild);
   System.out.print(node.datum + " ");
   show(node.rchild);
}
```


Tree Traversals

- "Walking" over the whole tree is a *tree traversal*
  - This is done often enough that there are standard names
  - The previous example is an *inorder traversal*
    - Process left subtree
    - Process node
    - Process right subtree
- Note: we're using this for printing, but any kind of processing can be done
- There are other standard kinds of traversals
  - Preorder traversal
    - Process node
    - Process left subtree
    - Process right subtree
  - Postorder traversal
    - Process left subtree
    - Process right subtree
    - Process node

Some Useful Methods

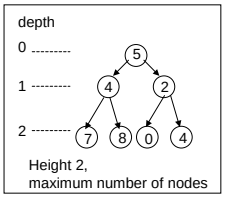
```
//determine if a node is a leaf
public static boolean isLeaf(TreeNode node) {


}

//compute height of tree using postorder traversal
public static int height(TreeNode node) {



}

//compute number of nodes in tree using postorder traversal
public static int nNodes(TreeNode node) {


}
```


Some Useful Methods

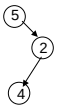
```
//determine if a node is a leaf
public static boolean isLeaf(TreeNode node) {
   return (node != null) && (node.lchild == null)
          && (node.rchild == null);
}

//compute height of tree using postorder traversal
public static int height(TreeNode node) {
   if (node == null) return -1; //empty tree -> height undefined
   if (isLeaf(node)) return 0;
   return 1 + Math.max(height(node.lchild),height(node.rchild));
}

//compute number of nodes in tree using postorder traversal
public static int nNodes(TreeNode node) {
   if (node == null) return 0;
   return 1 + nNodes(node.lchild) + nNodes(node.rchild);
}
```


Useful Facts about Binary Trees

- 2^d = maximum number of nodes at depth d
- If height of tree is h
  - Minimum number of nodes in tree = h + 1
  - Maximum number of nodes in tree = $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$
- Complete binary tree
  - All levels of tree down to a certain depth are completely filled



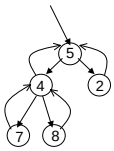
Height 2,
maximum number of nodes



Height 2,
minimum number of nodes

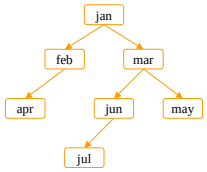
Tree with Parent Pointers

- In some applications, it is useful to have trees in which nodes can reference their parents
- Analog of doubly-linked lists



Things to Think About

- What if we want to *delete* data from a BST?
- A BST works great as long as it's *balanced*
  - How can we keep it balanced?



Tree Summary

- A *tree* is a recursive data structure
  - Each cell has 0 or more successors (*children*)
  - Each cell except the *root* has at exactly one predecessor (*parent*)
  - All cells are reachable from the *root*
  - A cell with no children is called a *leaf*
- Special case: *binary tree*
  - Binary tree cells have both a left and a right child
  - Either or both children can be null
- Trees are useful for exposing the recursive structure of natural language and computer programs