# Data Structures and Algorithms

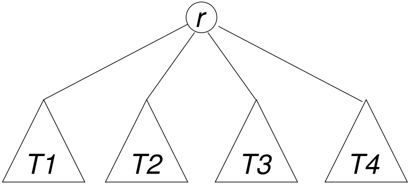
Spring 2008-2009

# Outline

# Outline

# Introduction

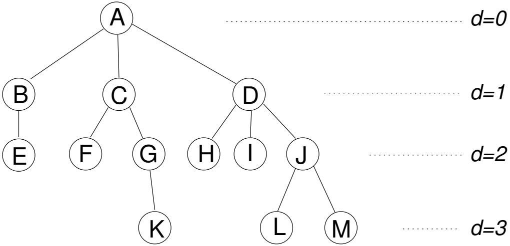
- A *tree* is defined recursively as follows:
  - A tree is a (possibly empty) collection of nodes
  - If non-empty, the tree has one special node, $r$, the *root*, and
  - Zero or more subtrees, $T_1, T_2, \ldots, T_k$
  - Subtrees $T_1, \ldots, T_k$ are connected to $r$ by a *directed* edge (from $r$ to $T_i$)
- The root node of a subtree is a *child* of $r$ and $r$ is the *parent* of each subtree root



# Introduction (contd.)

- A tree of $n$ nodes has $n-1$ edges since each node except $r$ has a parent node (denoted by an edge)
- Nodes may have an arbitrary number of children
- Nodes with no children are called *leaf* nodes
- Leaf nodes over are E, F, K, H, I, L and M
- Nodes with the same parent are called *siblings*
- A *path*, $p$, from node $n_1$ to node $n_k$ is the sequence of nodes $n_1, n_2, \ldots, n_k$ such that $n_i$ is the parent of $n_{i+1}$, where $1 \leq i < k$
- The length of $p$ is $k-1$
- The length of the path from a node to itself is 0

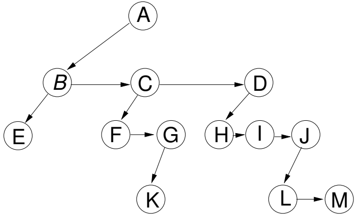
# Introduction (contd.)




- There is exactly one path from $r$ to every node
- The *depth* of a node, $n$, is the length of the path from $r$ to $n$
- The *height* of $n$ is the length of longest path from $n$ to a (descendant) leaf
- Node D (above) is at depth 1 and height 2
- If there is a path from $n_1$ to $n_2$ then $n_1$ is a (proper) *ancestor* of $n_2$ and $n_2$ is a (proper) *descendant* of $n_1$ (if $n_1 \neq n_2$)

# A Possibility

- Could think of earlier tree as follows




- This suggests the following `struct`

```
struct TreeNode
{
  Object el;
  TreeNode* firstChild;
  TreeNode* nextSibling;
}
```

- No need for anything more than two pointers

# A Better Implementation

- A `struct` can be thought of as an impotent C++ object
- This `struct` requires a linked list class but is more natural

```
struct TreeNode
{
  Object el;
  list<TreeNode> children;
}
```

# Systematically Processing a Tree

- Since a tree is defined recursively, to traverse a tree we can traverse each of its subtrees using the same algorithm
- We can "process" or "visit" the root node of each (sub)tree either before or after we traverse its children
- Processing the root node before (after, respectively) traversing the children is called a preorder (post-order)
- To "describe" or print out the names of the nodes of a tree using preorder traversal, the code on the following slide is a rough cut at it
- What's important is the position of the recursive call: colours match up with above

# Systematically Processing a Tree (contd.)

```
// assuming we're using the second
//   representation of a tree_node
void describe_tree(const tree_node& root,
                   const int depth = 0)
{
  // use depth as indentation!!
  print_name(root.el, depth);
  for (root.children.first();   // get first child
       !root.children;          // more children?
       ++root.children)         // get next child
    describe_tree(root.children(), depth+1);
    //  print_name(root.el, depth);
}
```

# Outline

1. Abstract Data Types (ADTs)
   - Trees
     - Implementing Trees
     - Tree Traversals
   - Binary Trees
   - Binary Search Trees

# When We Limit the Number of Children...

- In a binary tree a node can have at most two children
- These are referred to as the nodes *left* and *right* children
- On $i$th level of a binary tree can have at most $2^i$ nodes
- Over entire tree, number of nodes on $k$ levels, $N(k)$, is

$$N(k) = \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

- So, *at most* $2^k - 1$ nodes can be stored on $k$ levels

$$\boxed{n \leq 2^k - 1}$$

- What about the other way? How many levels do we need to store $n$ nodes?

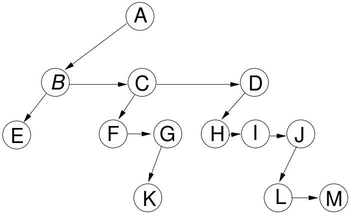
# When We Limit the Number of Children... (contd.)

- From above, $n \leq 2^k - 1 \Rightarrow 2^k \geq n + 1 \Rightarrow k \geq \log(n+1)$
- So putting 7 nodes in a binary tree cannot be done with fewer than $\log_2 8 = 3$ levels; putting 8 nodes in a tree requires minimum of $\log_2 9 = 3.1699$ levels!
- ...must round up if necessary to next highest integer
- Need at least $\boxed{k \geq \lceil \log(n+1) \rceil}$ levels to store $n$ nodes
- Therefore the path length to the furthest node in a binary tree *can* be $O(\log n)$
- This makes a binary tree a useful candidate for storing information

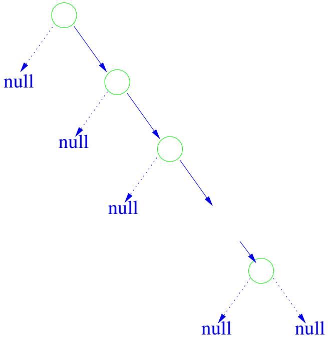
✗ Common mistake: an $n$-node binary tree does *not* have depth $O(\log n)$ in general

✓ We want it to have depth $O(\log n)$ but we will need to do some work to achieve it

# Binary Tree Examples




Tree seen earlier




A binary tree with height, $h = n$

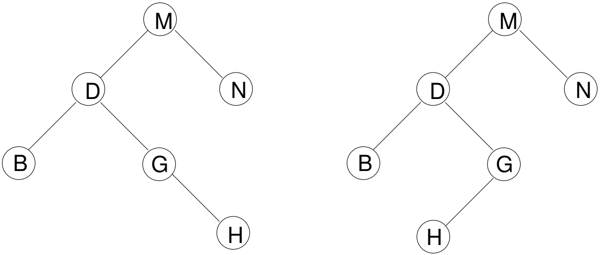
# Outline

1. Abstract Data Types (ADTs)
   - Trees
     - Implementing Trees
     - Tree Traversals
   - Binary Trees
   - Binary Search Trees

# A Third Type of Tree Traversal

- With a binary tree we have a third type of traversal: *inorder*
- An inorder traversal of a tree recursively traverses the left subtree, “visits” or “processes” the tree root and traverses the right subtree
- A binary tree is a *binary search tree* if an inorder traversal of the tree visits the nodes in sorted order

# A Third Type of Tree Traversal (contd.)



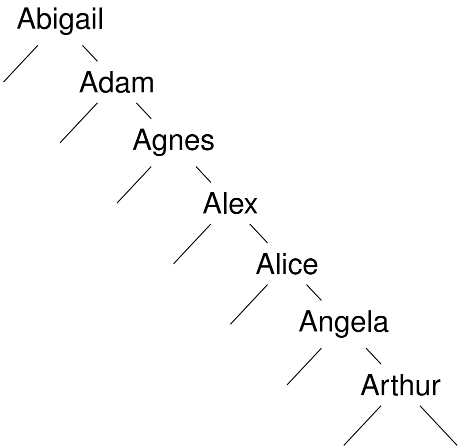

Binary Search Tree

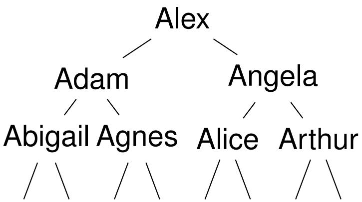
Binary Tree

- Traversals of left and right trees above (assuming that we process siblings left to right):

| | left | right |
|---|---|---|
| Preorder | MDBGHN | MDBGHN |
| Inorder | BDGHMN | BDHGMN |
| Postorder | BHGDNM | BHGDNM |

# A bad and good BST




Tree has $k = \lceil \log(n+1) \rceil = 3$ levels, the smallest possible; this shows us that with luck(??) we can achieve a very dense tree.