i206: Lecture 12:
Trees

Tapan Parikh

Spring 2013

Some slides courtesy Marti Hearst, John Chuang and others

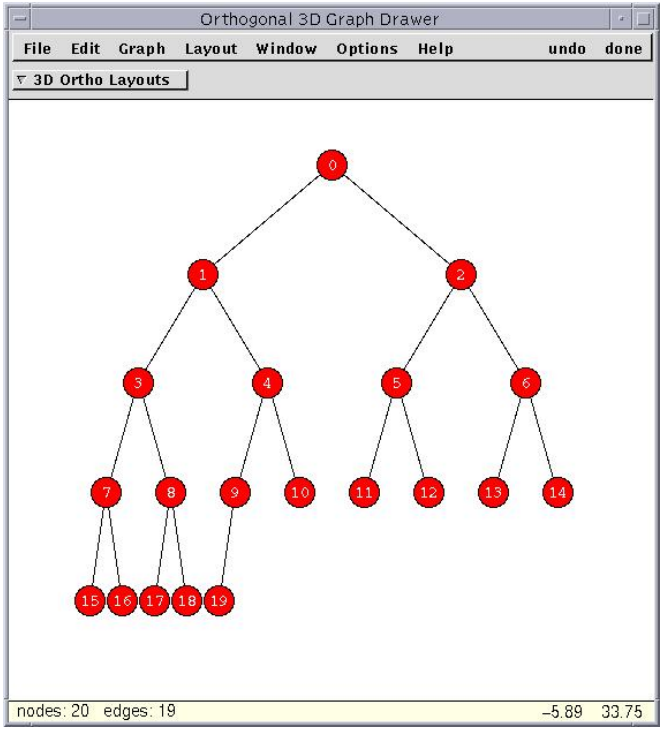
Data Structures Outline

- What is a data structure
- Basic building blocks: arrays and linked lists
- Data structures (uses, methods, performance):
  - List, stack, queue
  - Dictionary
  - Tree
  - Graph

Trees

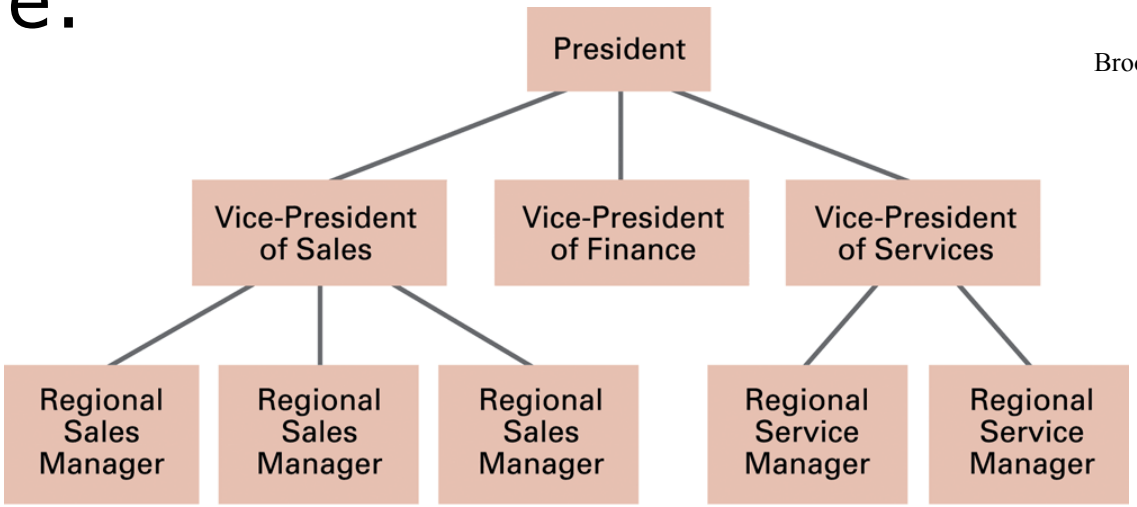
Trees

- Trees are very important and useful
- They are usually drawn upside-down
- The allow us to represent hierarchy
  - File system
  - Book structure
  - Employees in a bureacracy
- Every node has exactly one parent
  - Except the root



Tree Data Structure

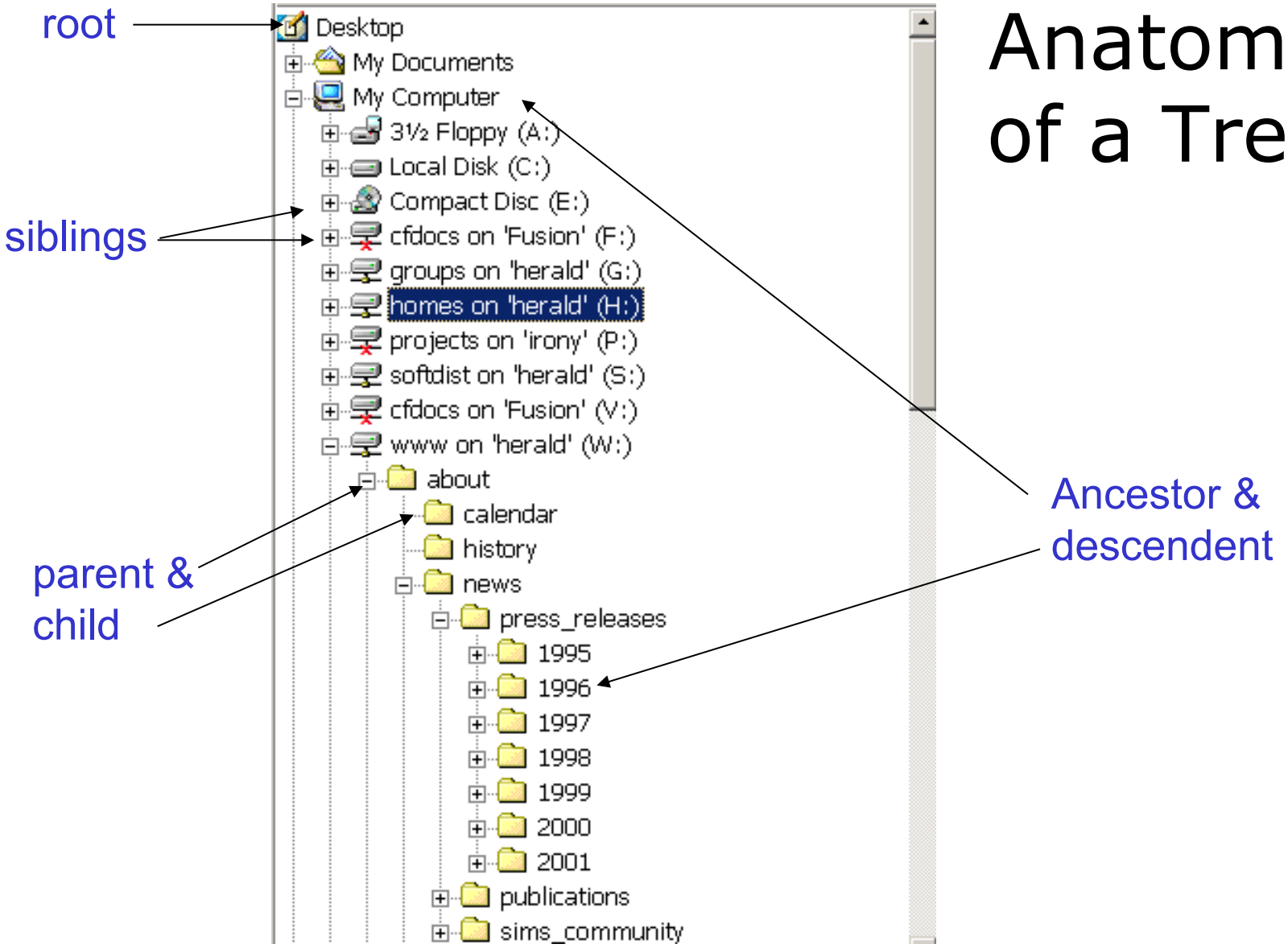
- **Tree** = a collection of data whose entries have a hierarchical organization
- Example:



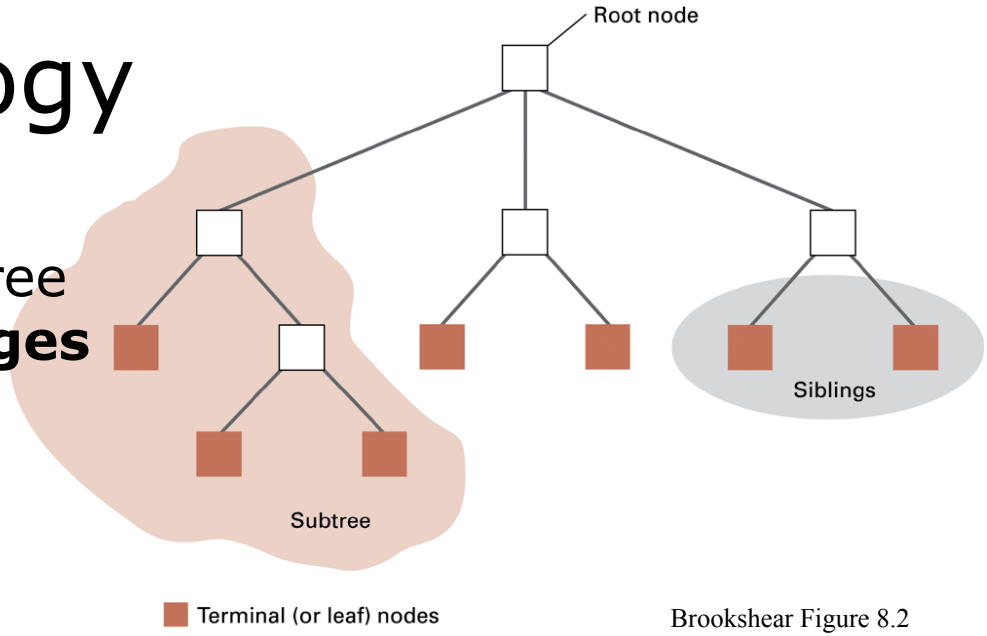
Brookshear Figure 8.1

- What are some examples of trees in computer systems?

Anatomy of a Tree



Tree Terminology

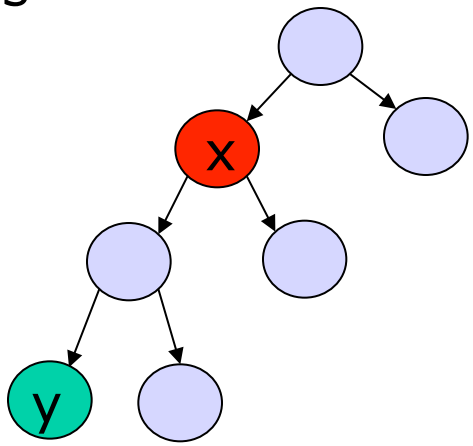


Brookshear Figure 8.2

- **Node** = an entry in a tree
- Nodes are linked by **edges**
- **Root** node = the node at the top
- **Terminal** or **leaf** node = a node at the bottom
- **Parent** of a node = the node immediately above the specified node
- **Child** of a node = a node immediately below the specified node
- **Ancestor** = parent, parent of parent, etc.
- **Descendent** = child, child of child, etc.
- **Siblings** = nodes sharing a common parent

Tree Terminology

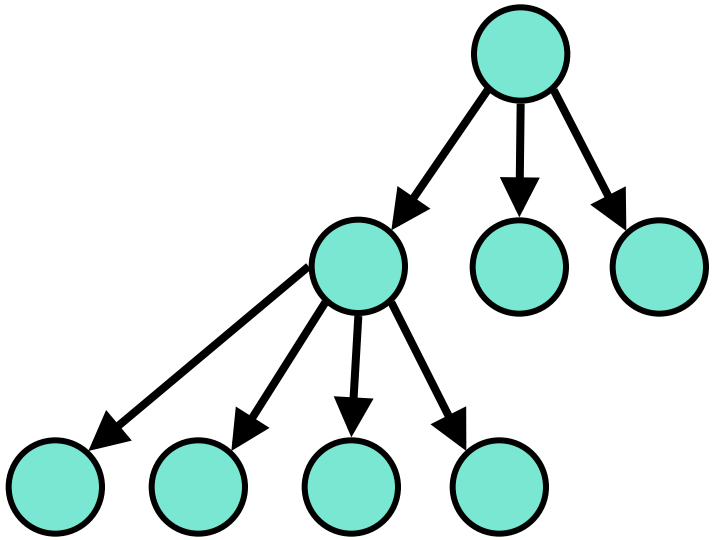
- Depth of a node
  - The depth of a node v in T is the number of ancestors of v, excluding v itself.
  - More formally:
    - If v is the root, the depth of v is 0
    - Else the depth of v is one plus the depth of the parent of v
- Height (or depth) of a tree
  - The depth of a tree is the maximum depth of any of its leaves



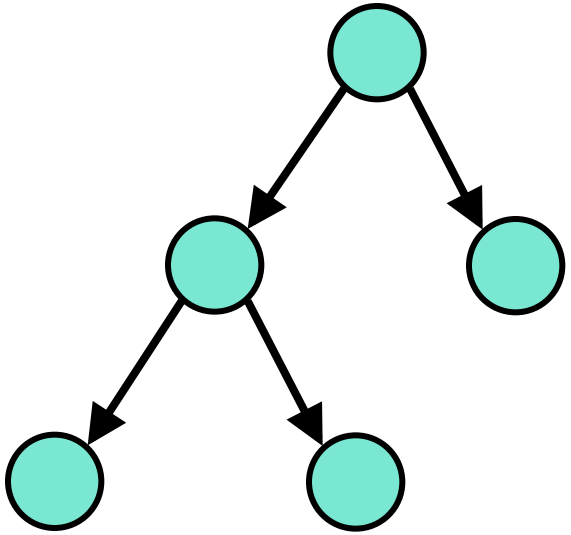
Depth(x) = 1
Depth(y) = 3
Depth(T) = 3

Types of Trees

- General tree – a node can have any number of children

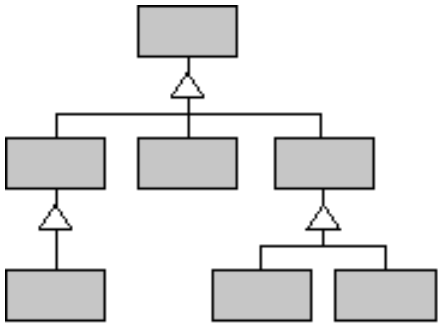


- Binary tree – a node can have at most two children

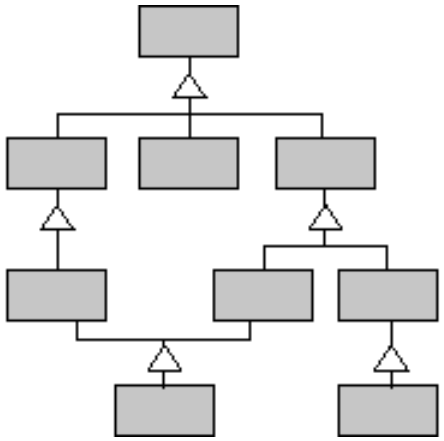


Trees vs. Graphs / Networks

- The Web is not a tree. Why not?
  - Nodes in a tree can have only one parent
- Graphs / Networks are the more general case



Baum-Struktur



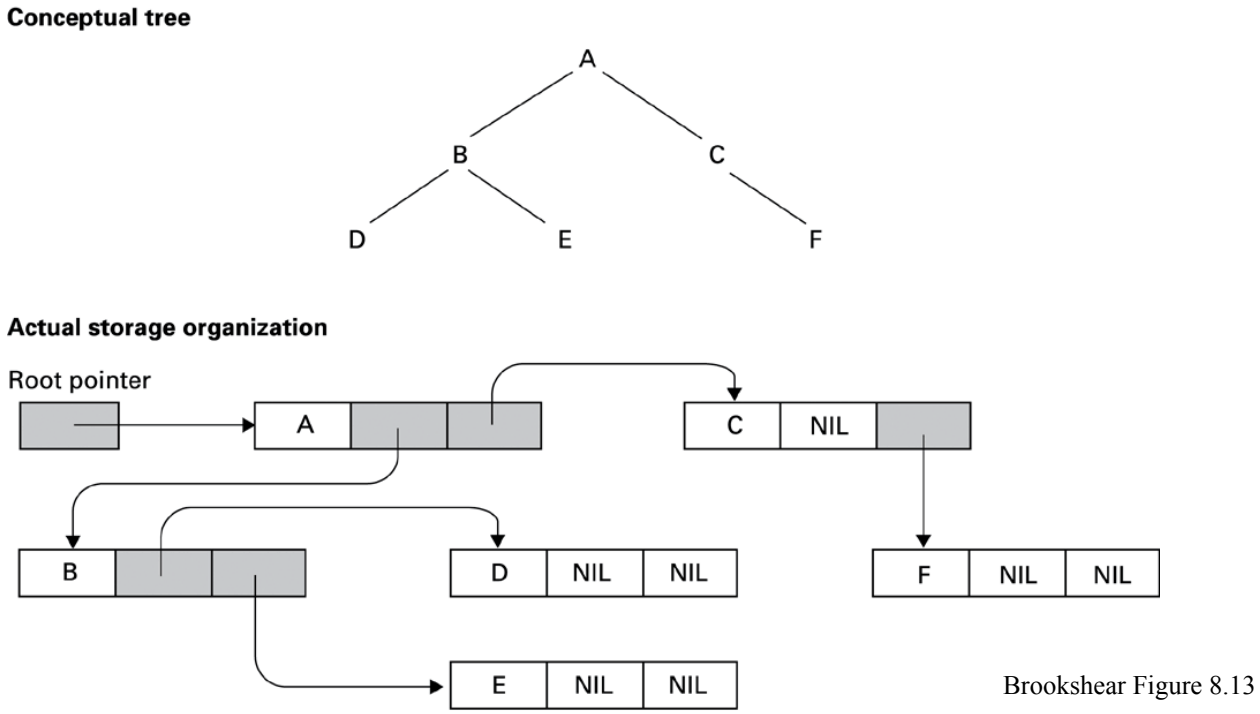
DAG

Binary Tree

- The simplest form of tree is a Binary Tree
- A Binary Tree consists of
  - (a) A node (called the root node) and
  - (b) Left and right subtrees
  - Both the subtrees are themselves binary trees
    - Note: this is a recursive definition

Implementing a Binary Tree

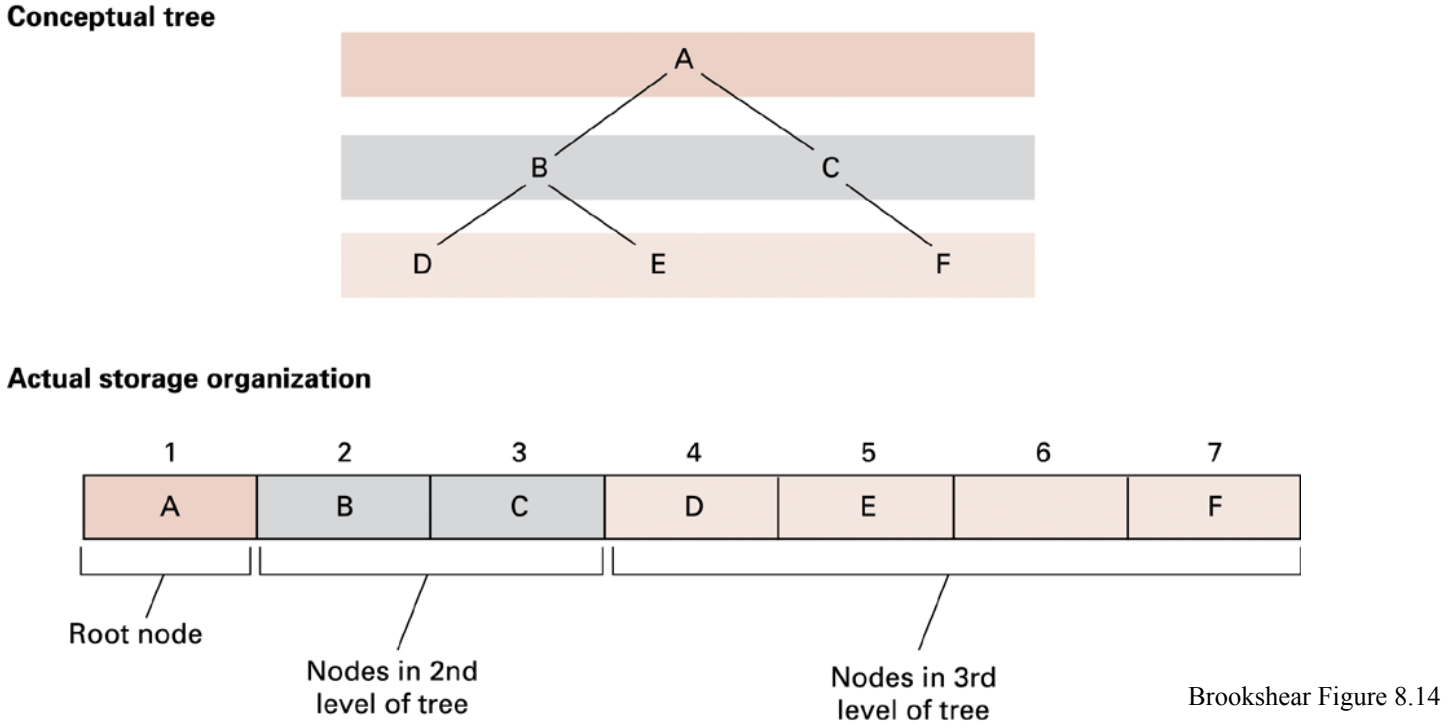
- Linked list
  - Each node = data cell + two child pointers
  - Accessed through a pointer to root node



Brookshear Figure 8.13

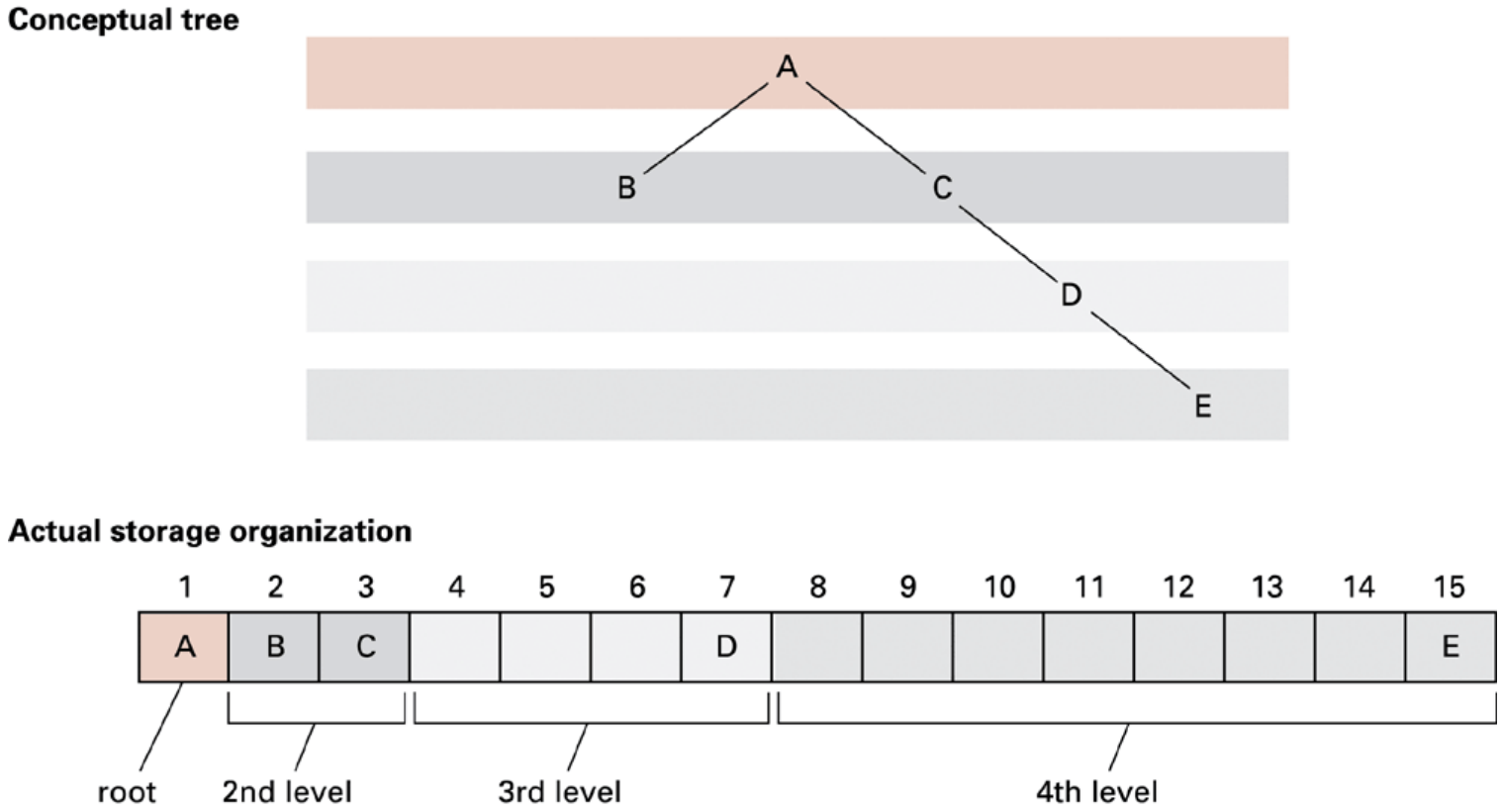
Implementing a Balanced Binary Tree

- Balanced: the depth of the two subtrees of every node never differ by more than 1
- Array
  - A[1] = root node
  - A[2],A[3] = children of A[1]
  - A[4],A[5],A[6],A[7] = children of A[2] and A[3]
  - …



Brookshear Figure 8.14

A sparse, unbalanced tree



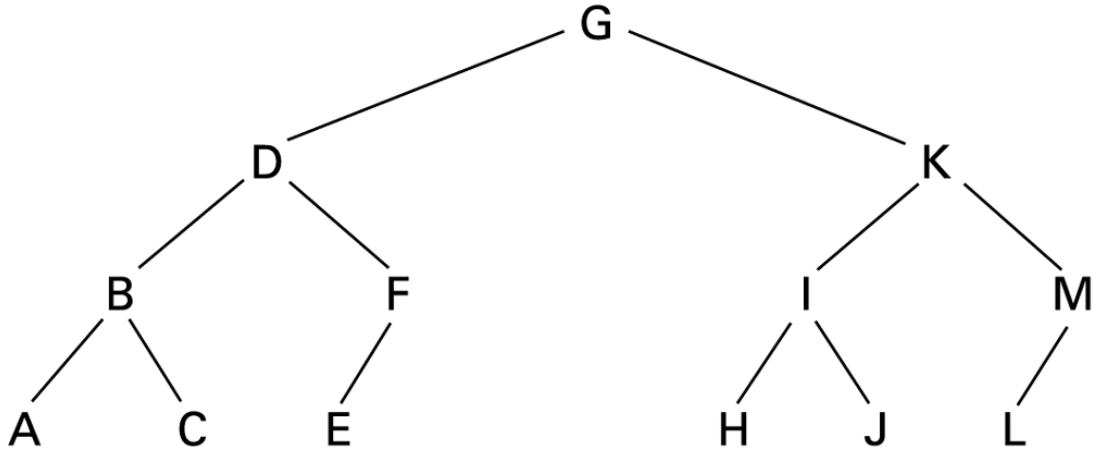
Brookshear Figure 8.15

Tree Methods

- search (or get) – Return a reference to a node in a tree
- insert (or put) – Put a node into the tree; if the tree is empty the node becomes the root of the tree
- delete (or remove) – Remove a node from the tree
- traversal – access all nodes in the tree

Application: Binary Search

- Recall that binary search algorithm is O (log n)
- One way of implementing the algorithm is to use a binary search tree (BST) data structure



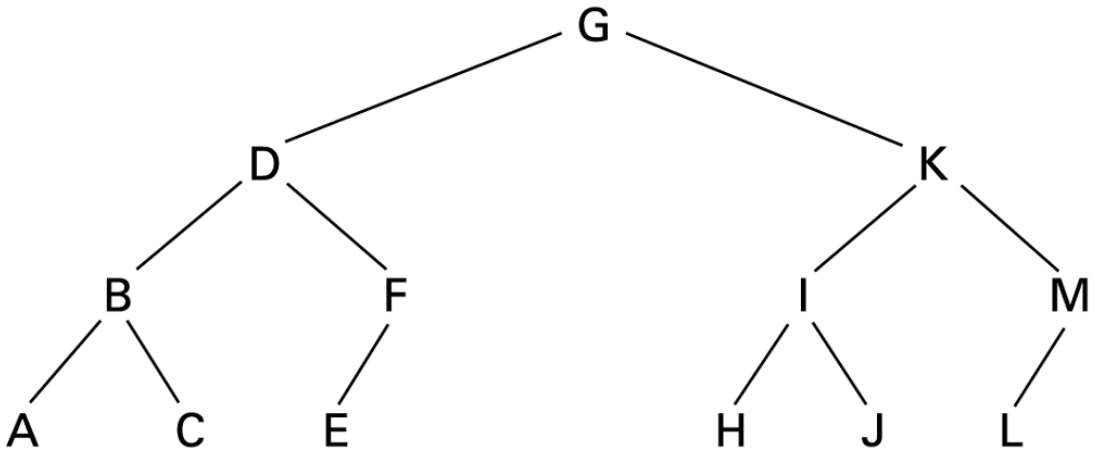
Brookshear Figure 8.17

Binary Search Tree

- Differs from Binary tree in that
  - Each internal node has a key
  - The key is used to give an order to the nodes
  - Nodes often store data
    - But sometimes stored only in leaf (B+Trees)
  - The keys are stored in order such that
    - Keys to the left of a node are less than keys to the right of that node
  - More formally,
    - For each internal node v with key k
      - Keys stored in nodes in the left subtree of v are <= k
      - Keys stored in nodes in the right subtree are >= k

Binary Search Tree

- Binary search tree (BST) data structure
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - Both the left and right subtrees must also be binary search trees.

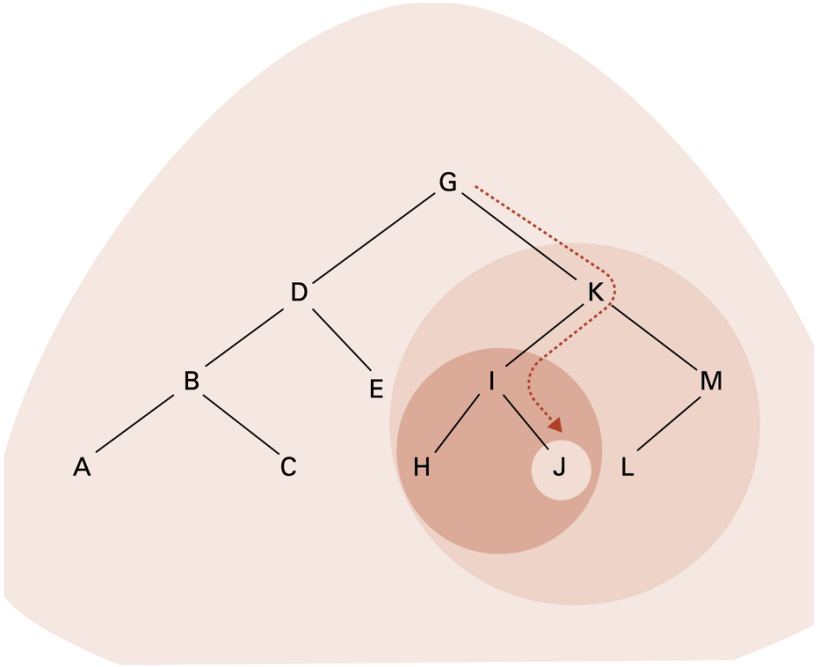


Brookshear Figure 8.17

Binary Tree Search

```
procedure Search(Tree, TargetValue)

if (root pointer of Tree = NIL)
   then
      (declare the search a failure)
  else
      (execute the block of instructions below that is
       associated with the appropriate case)
       case 1: TargetValue = value of root node
               (Report that the search succeeded)
       case 2: TargetValue < value of root node
               (Apply the procedure Search to see if
                  TargetValue is in the subtree identified
                   by the root's left child pointer and
                   report the result of that search)
       case 3: TargetValue > value of root node
               (Apply the procedure Search to see if
                   TargetValue is in the subtree identified
                   by the root's right child pointer and
                   report the result of that search)
      ) end if
```

Brookshear Figure 8.18



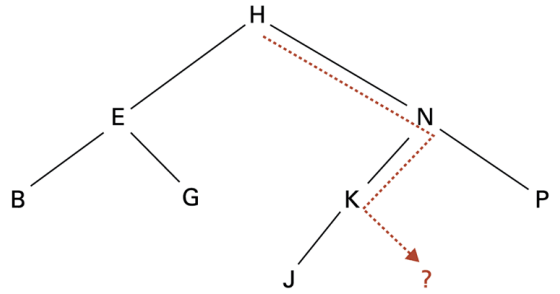
Example: Search (Tree, J)
Brookshear Figure 8.19

Node Insertion

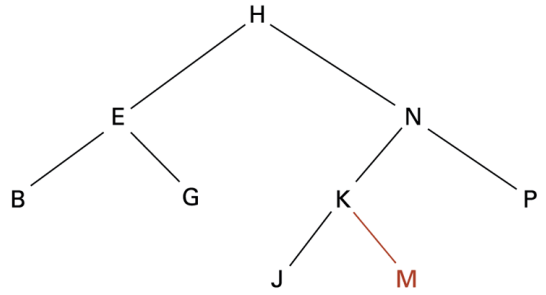
```
procedure Insert(Tree, NewValue)

if (root pointer of Tree = NIL)
  (set the root pointer to point to a new leaf
          containing NewValue)
  else (execute the block of instructions below that is
          associated with the appropriate case)
            case 1: NewValue = value of root node
                    (Do nothing)
            case 2: NewValue < value of root node
                    (if (left child pointer of root node = NIL)
                              then (set that pointer to point to a new
                                        leaf node containing NewValue)
                              else (apply the procedure Insert to insert
                                        NewValue into the subtree identified
                                        by the left child pointer)
             case 3: NewValue > value of root node
                    (if (right child pointer of root node = NIL)
                              then (set that pointer to point to a new
                                        leaf node containing NewValue)
                              else (apply the procedure Insert to insert
                                        NewValue into the subtree identified
                                        by the right child pointer)
        ) end if
```

Brookshear Figure 8.23

a. Search for the new entry until its absence is detected

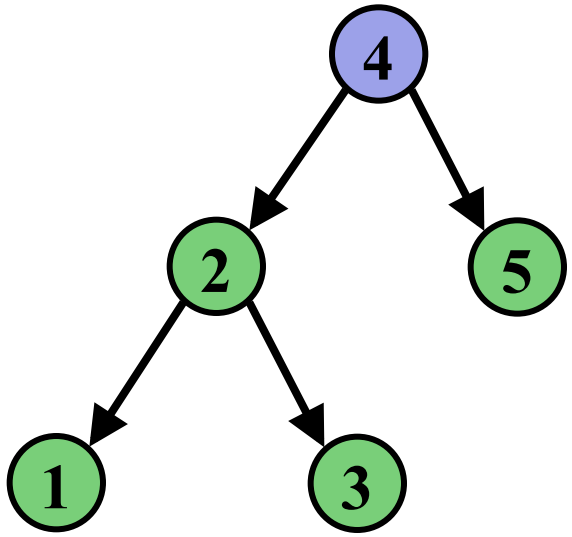


b. This is the position in which the new entry should be attached



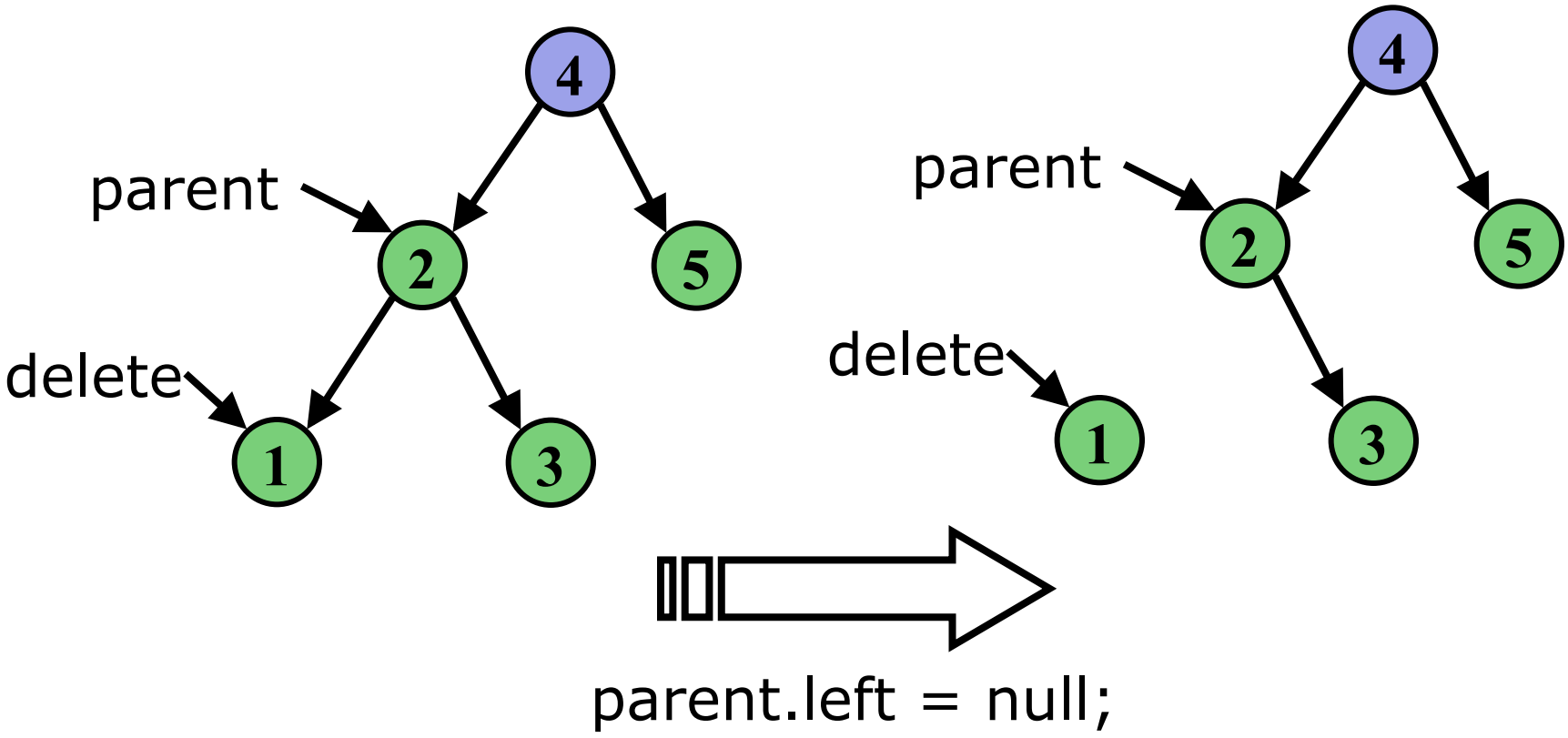
Example: Insert (Tree, M)
Brookshear Figure 8.22

Node Deletion

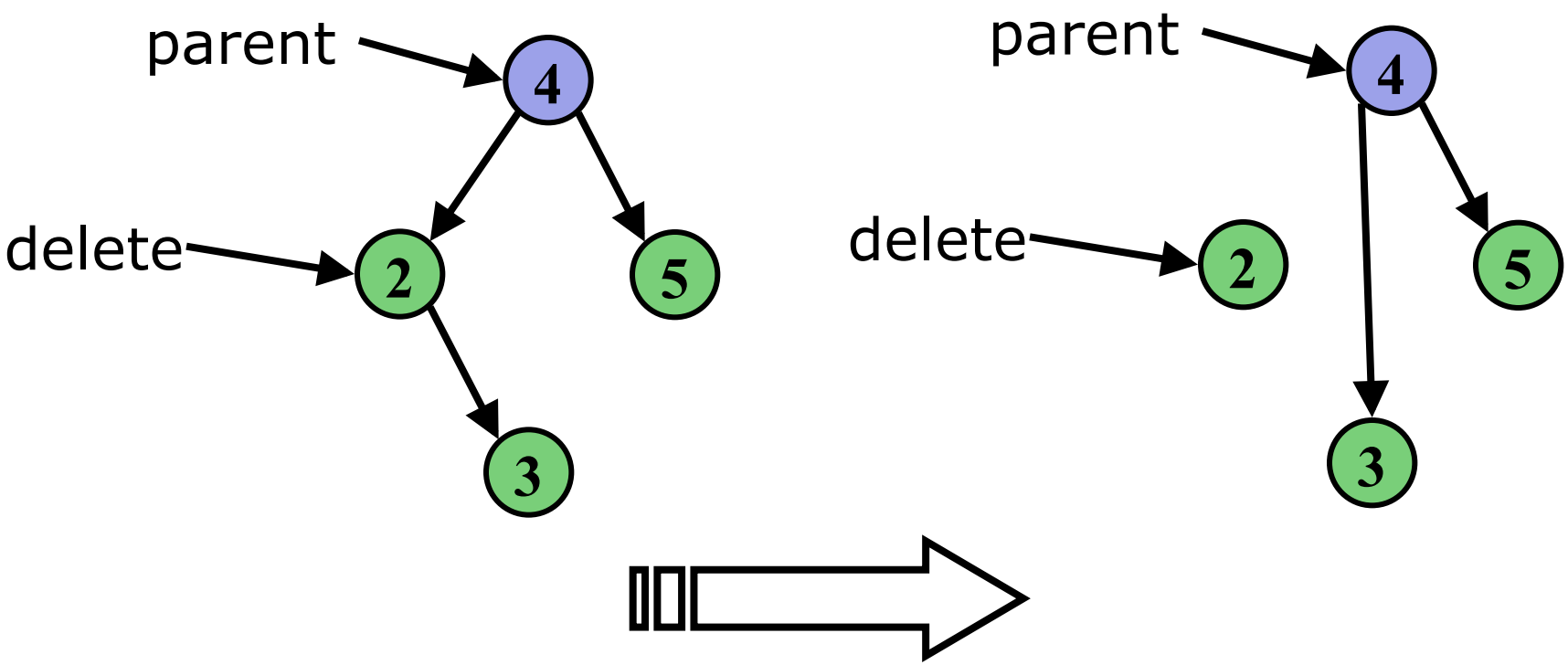


- When you delete a node that has children, you need to re-link the tree
- Three deletion cases:
  - Delete node with no children
  - Delete node with one child (which may be a subtree)
  - Delete node with more than one child
- Last case is a challenge with ordered trees
  - You want to preserve the ordering

Delete Nodes – No children



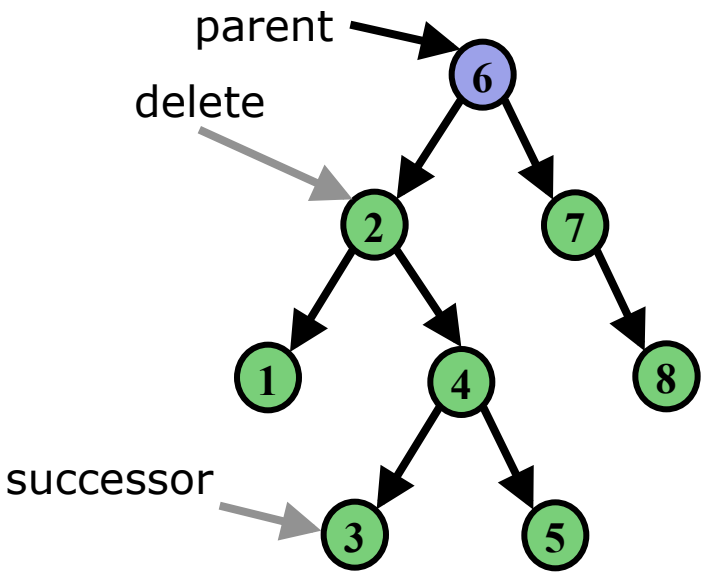
Delete Nodes – One child

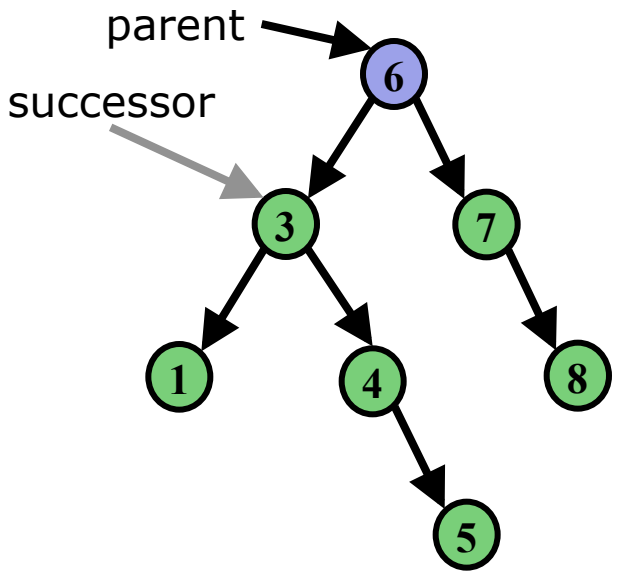


parent.left = delete.right;

Deleting Nodes – Multiple children

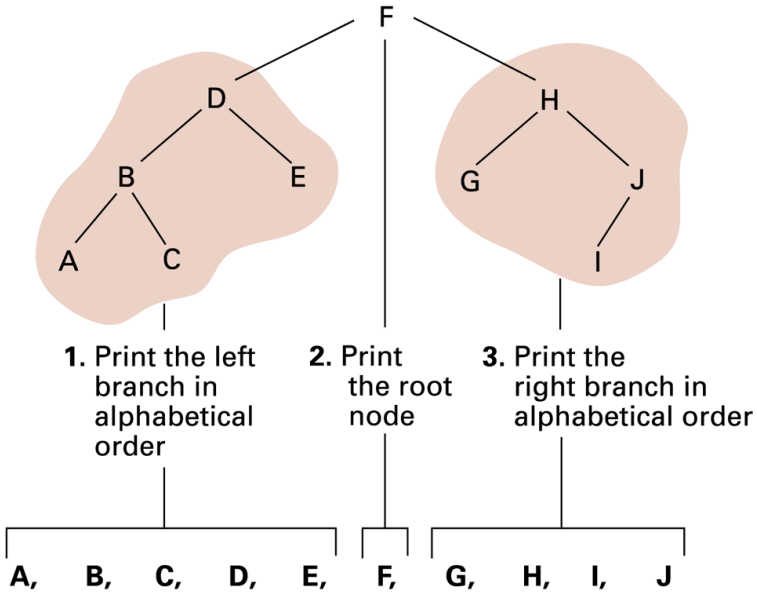
- Assuming you want to preserve the sort order, deletion is not straight forward
- Replace the deleted node with a node that preserves the sort order – the in-order successor node for in-order trees
- The successor node will be in the right subtree, left most node (or in the left subtree, right most node)





Tree Traversal

- A systematic way of accessing (or visiting) all the nodes of a tree
- Example: printing binary search tree in alphabetical order using “in-order” tree traversal

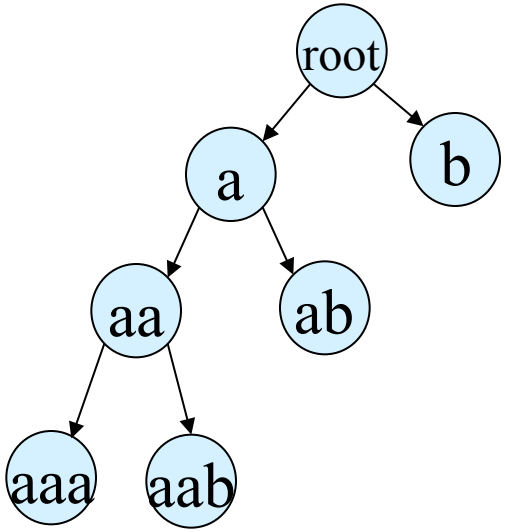


Brookshear Figure 8.20

```
procedure PrintTree (Tree)

if (Tree is not empty)
    then (Apply the procedure PrintTree to the tree that
              appears as the left branch in Tree;
          Print the root node of Tree;
          Apply the procedure PrintTree to the tree that
              appears as the right branch in Tree)
```

Brookshear Figure 8.21

Tree Traversal

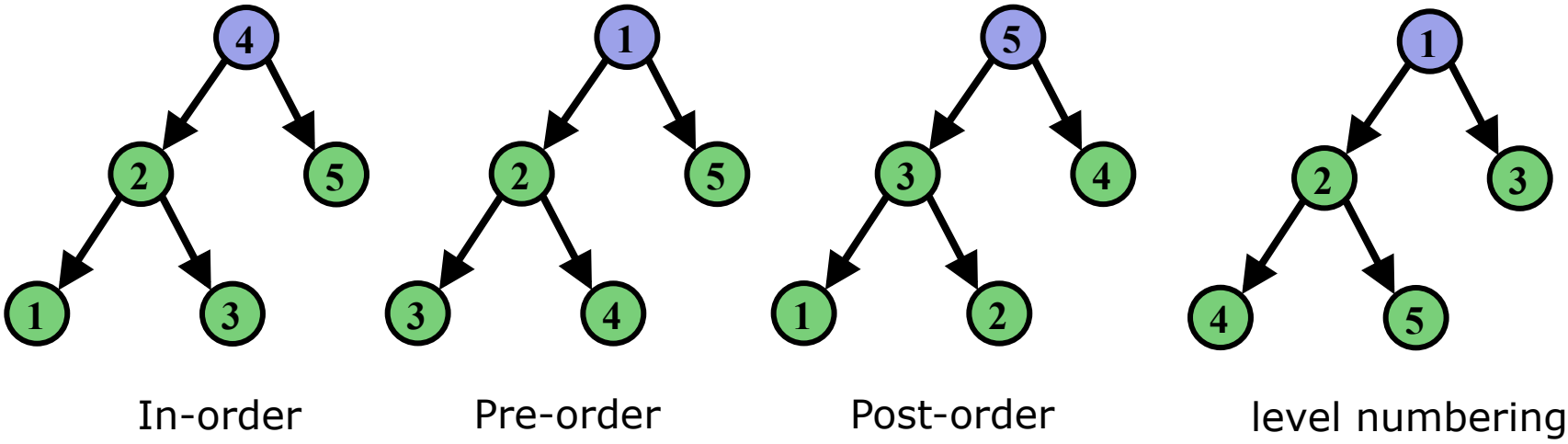


- Preorder
- Inorder
- Postorder

- All defined in terms of
  - When do you visit the node itself
  - When do you visit the lefthand side
  - When do you visit the righthand side

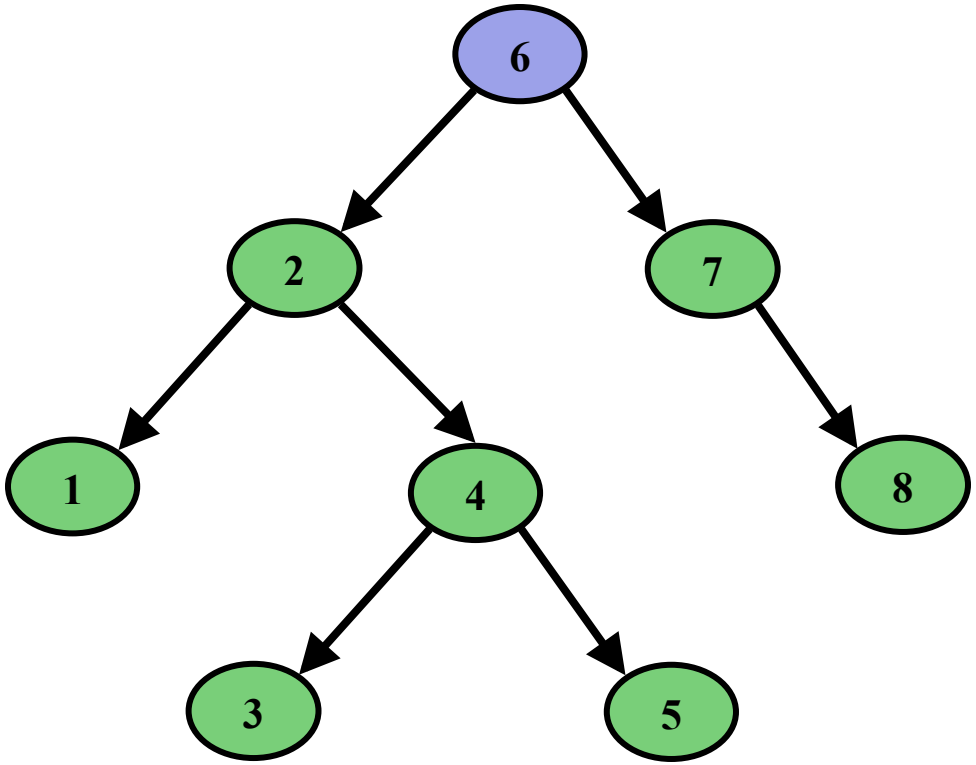
Tree Traversal

- Different types of traversal



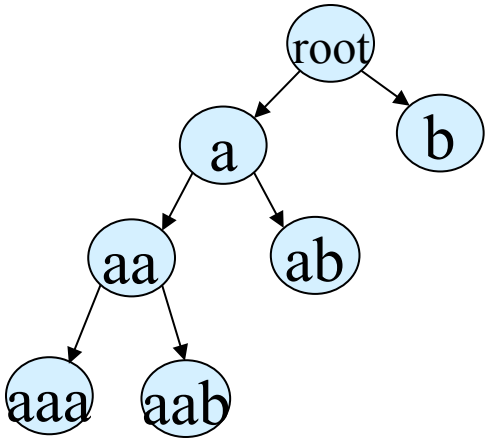
Tree Traversal

- In what order will these nodes be visited for pre-order, in-order, post-order?

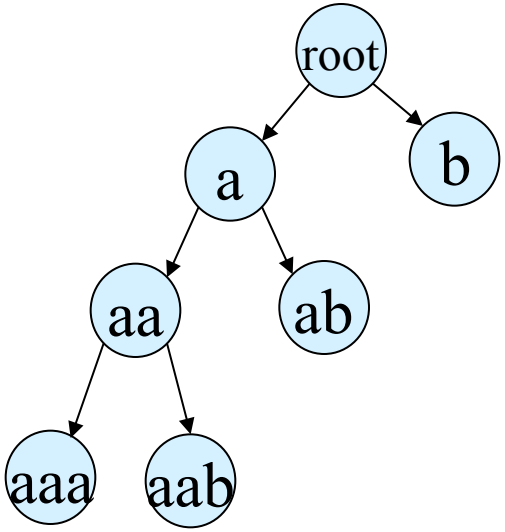


Binary Tree Implementation

```
class BinaryTree():
    def __init__(self, name="", value=-1, leftChild=None, rightChild=None):
        self.name = name
        self.value= value
        self.leftChild = leftChild
        self.rightChild = rightChild

    def setSubtrees(self, leftChild, rightChild):
        self.leftChild = leftChild
        self.rightChild = rightChild

def buildABinaryTree():
    root = BinaryTree("root", 0)
    childA = BinaryTree("a", 1)
    childB = BinaryTree("b", 1)
    root.setSubtrees(childA, childB)                                    # one way add child nodes
    childA.setSubtrees(BinaryTree("aa",2), BinaryTree("ab",2))  #another way
    childA.leftChild.setSubtrees(BinaryTree("aaa", 3), BinaryTree("aab", 3))

    print("Preorder: visit each node before its children")
    root.printPreOrder()
    print("Inorder: visit the left subtree, visit the node, visit the right subtree")
    root.printInOrder()

buildABinaryTree()
```



PreOrder



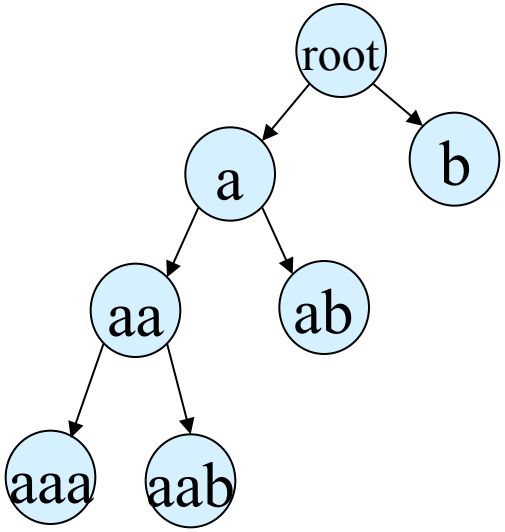

```
def printPreOrder(self):
    if not self: return
    print (self.name + " " + str(self.value))
    if self.leftChild:
        self.leftChild.printPreOrder()
    if self.rightChild:
        self.rightChild.printPreOrder()
```

```
Preorder: visit each node before its children
root 0
a 1
aa 2
aaa 3
aab 3
ab 2
b 1
```

Why? Textual representation of the tree (parents before children)

InOrder

```
def printInOrder(self):
    if not self: return
    if self.leftChild:
        self.leftChild.printInOrder()
    print (self.name + " " + str(self.value))
    if self.rightChild:
        self.rightChild.printInOrder()
```



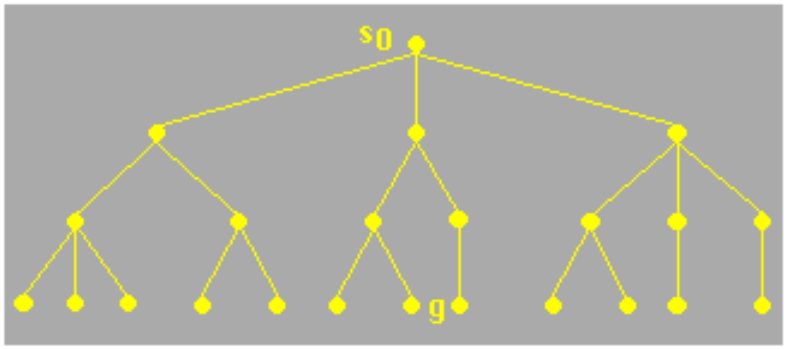

```
Inorder: visit the left subtree, visit the node, visit the right subtree
aaa 3
aa 2
aab 3
a 1
ab 2
root 0
b 1
```

Why? Useful for searching in ordered trees

Depth-first Search

A depth first search (DFS) through a tree starts at the root and goes straight down a single branch until a leaf is reached. Then, it continues at the nearest ancestor that hasn't been searched through yet.

Full tree



Note: DFS traversal is equivalent to PreOrder traversal

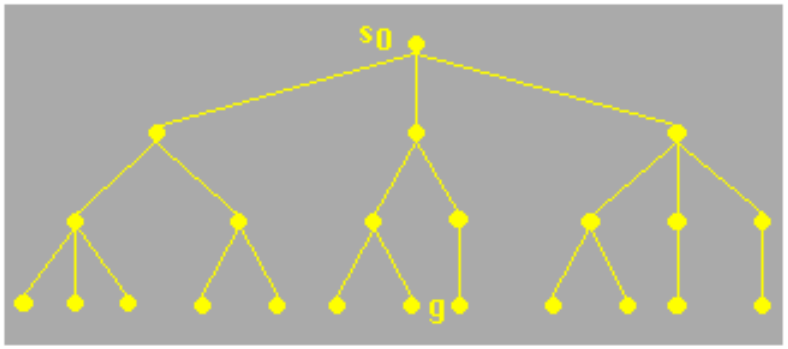
DFS (to see it, run in Presentation Mode)

From http://www.rci.rutgers.edu/~cfs/472_html/AI_SEARCH/ExhaustiveSearch.html

Breadth-first Search

A breadth first search (BFS) through a tree starts at the root and moves from nodes left to right at each level until all the nodes have been looked at or until the value is found.

Full tree



What do we need besides the runtime stack to make BFS work?

BFS (to see it, run in Presentation Mode)

From http://www.rci.rutgers.edu/~cfs/472_html/AI_SEARCH/ExhaustiveSearch.html

Breadth First Search

```
# use a helper function so initial call to BFS doesn't need a queue
def bfsHelper(self, name, queue):
    if len(queue) == 0: return False
    current = queue.popleft()
    if current.name == name: return current.value
    if current.leftChild: queue.append(current.leftChild)
    if current.rightChild: queue.append(current.rightChild)
    return self.bfsHelper(name, queue)


def breadthFirstSearch(self, name):
    if self is None: return False
    if self.name == name: return self.value
    queue = deque()
    if self.leftChild: queue.append(self.leftChild)
    if self.rightChild: queue.append(self.rightChild)
    return self.bfsHelper(name, queue)
```

Depth First Search

```
def depthFirstSearch(self, name):
    if not self: return False
    if (self.name == name): return self.value
    if self.leftChild:
        result = self.leftChild.depthFirstSearch(name)
        if result: return result      # if you found it, stop the recursion
    if self.rightChild:
        result = self.rightChild.depthFirstSearch(name)
        if result: return result
```

Efficiency of Binary Search Trees

- Search, insert, and delete operations: O(height of tree)

- Average Case
  - Search: O(log n)
  - Insertion: O(log n)
  - Deletion: O(log n)
  - Traversal: O(n)
- Worst Case
  - Search: O(n)
  - Insertion: O(n)
  - Deletion: O(n)
  - Traversal: O(n)

- AVL trees and Red-Black trees
  - Self-balancing trees with tree height O(log n)
  - Worst case O(logN) searches, insertions, and deletions
- Many other variations: Splay trees, B-trees, etc.