Computer Science Department
College of Computer at Al-Lith

جامعة أم القرى
UMM AL-QURA UNIVERSITY

Data Structures

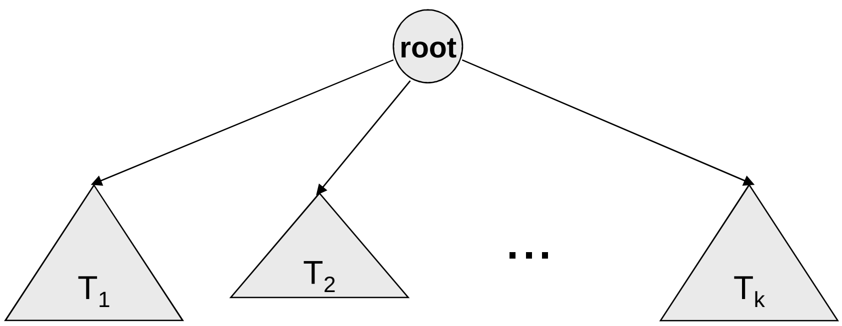
Chapter 4: Search trees

Instructor Maher Hadiji
hdiji.maher@gmail.com

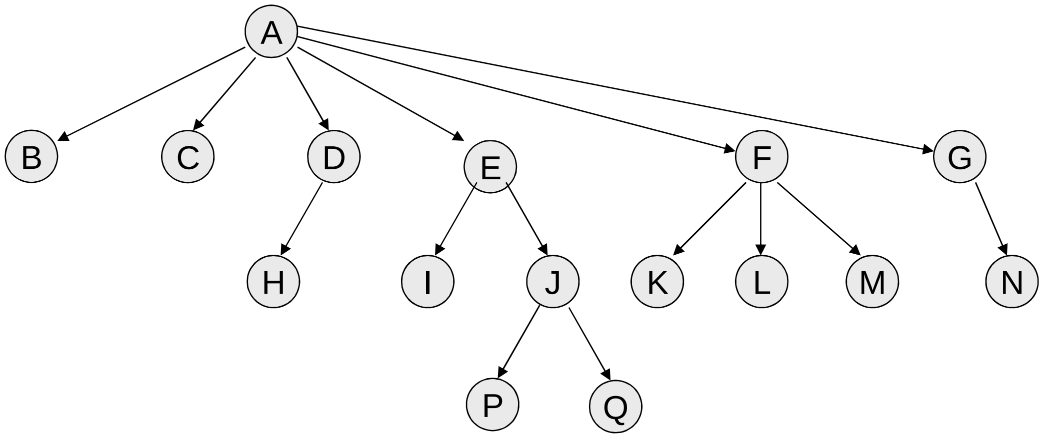
2015-2016

What is a Tree?

- A tree is a collection of nodes with the following properties:
  - The collection can be empty.
  - Otherwise, a tree consists of a distinguished node r, called *root*, and zero or more nonempty sub-trees $T_1, T_2, \dots, T_k$, each of whose roots are connected by a *directed edge* from r.
- The root of each sub-tree is said to be *child* of r, and r is the *parent* of each sub-tree root.
- If a tree is a collection of N nodes, then it has N-1 edges.



Preliminaries



- Node *A* has 6 *children*: B, C, D, E, F, G.
- B, C, H, I, P, Q, K, L, M, N are *leaves* in the tree above.
- K, L, M are *siblings* since F is parent of all of them.

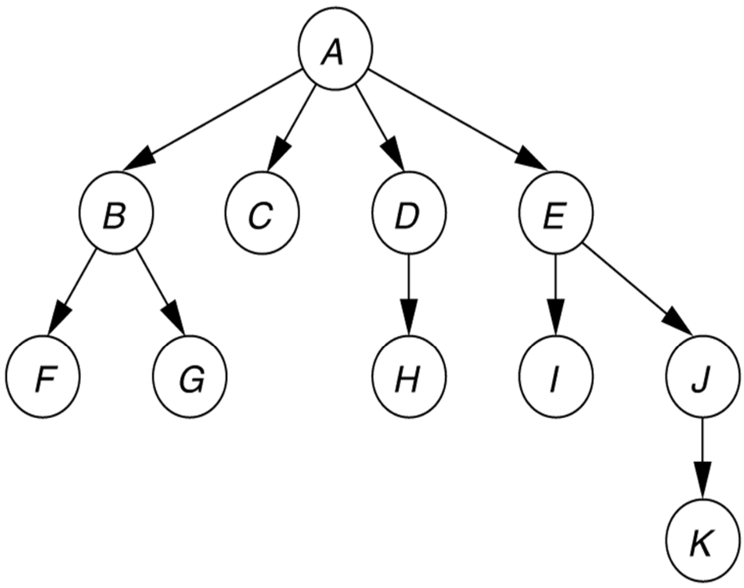
Preliminaries (continued)

- A ***path*** from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is parent of n_{i+1} $(1 \leq i < k)$
  - The ***length*** of a path is the number of edges on that path.
  - There is a path of length zero from every node to itself.
  - There is exactly one path from the root to each node.
- The ***depth*** of node n_i is the length of the path from *root* to node n_i
- The ***height*** of node n_i is the length of longest path from node n_i to a *leaf*.
- If there is a path from n_1 to n_2, then n_1 is ***ancestor*** of n_2, and n_2 is ***descendent*** of n_1.
  - If $n_1 \neq n_2$ then n_1 is *proper ancestor* of n_2, and n_2 is *proper descendent* of n_1.

- The **subtree** of T rooted at a node v is the tree consisting of all the descendents of v in T (including v itself).
- An **edge** of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa.

Figure 1

A tree, with height and depth information

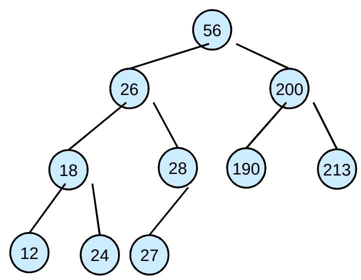


Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

binary tree

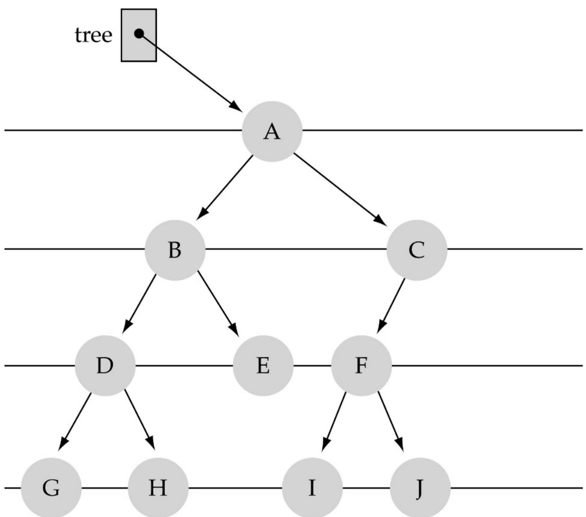
- Recursive definition
  1. An empty tree is a binary tree
  2. A node with two child subtrees is a binary tree
  3. Only what you get from 1 by a finite number of applications of 2 is a binary tree.

Is this a binary tree?



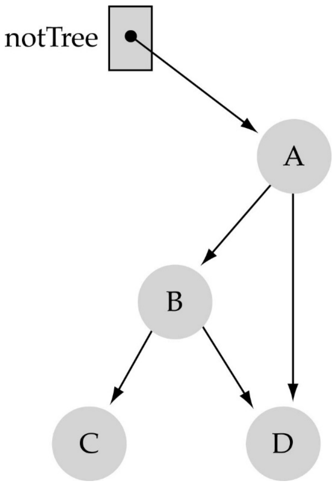
What is a binary tree?

- *Property 1:* each node can have up to two successor nodes.



What is a binary tree? (cont.)

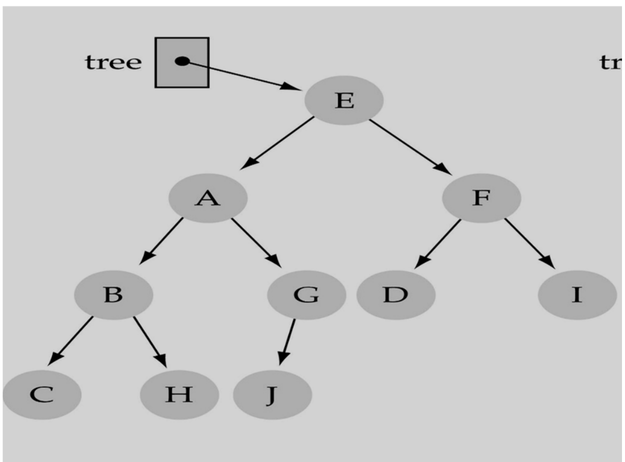
- *Property 2:* a unique path exists from the root to every other node



Not a valid binary tree!

Some terminology

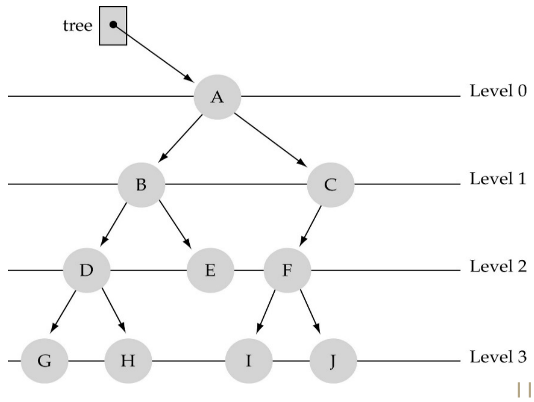
- The successor nodes of a node are called its *children*
- The predecessor node of a node is called its *parent*
- The "beginning" node is called the *root* (has no parent)
- A node without *children* is called a *leaf*



Some terminology (cont'd)

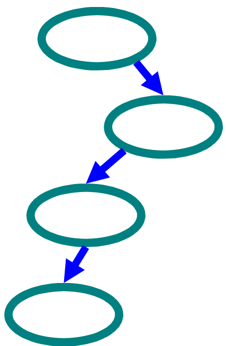
- Nodes are organize in levels (indexed from 0).
- **Level (or depth) of a node**: number of edges in the path from the root to that node.
- **Height of a tree h**: #levels = L
- **Full tree:** every node has exactly two children ***and*** all the leaves are on the same level.

not full!

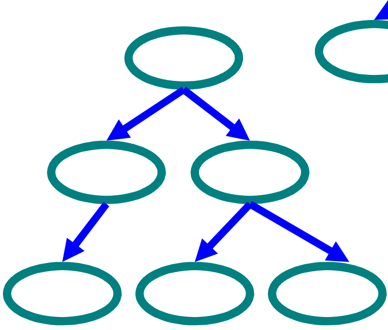


Types of Binary Trees

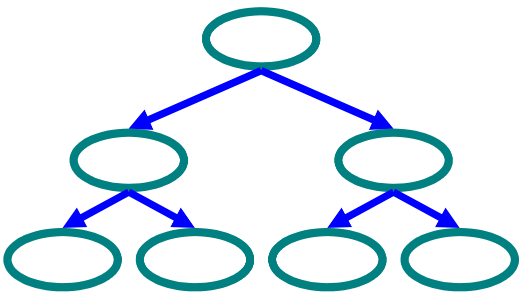
- Degenerate – only one child
- Complete – always two children
- Balanced – "mostly" two children



Degenerate binary tree



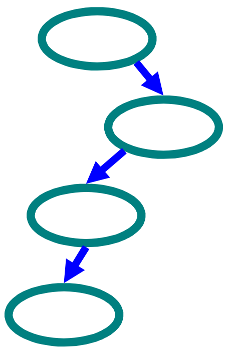
Balanced binary tree



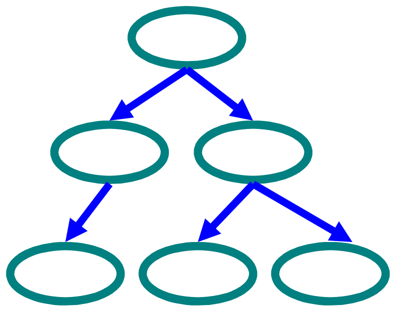
Complete binary tree

Binary Trees Properties

- Degenerate
  - Height = O(n) for n nodes
  - Similar to linked list
- Balanced
  - Height = O(log(n)) for n nodes
  - Useful for searches



Degenerate binary tree



Balanced binary tree

Traversals of Binary Trees

Preorder Traversal of a Binary Tree

- visit the root
- traverse in preorder the children (subtrees)

Algorithm preorder(T,v):

Perform the "visit" action for node v

for each child w of v in T do

Preorder(T,w)

- Preorder traversal can be applied to any binary tree as following:

Algorithm binaryPreorder(T,v):

if v has a left child u in T then

binaryPreorder(T,u) { recursively traverse left subtree}

if v has a right child w in T then

binaryPreorder(T,w)

Traversals of Binary Trees

Postorder Traversal of a Binary Tree

- traverse in postorder the children (subtrees)
- visit the root

Algorithm binaryPostorder(T,v):

```
if v has a left child u in T then
    binaryPostorder(T,u)
if v has a right child w in T then
    binaryPostorder(T,w)
perform the "visit" action for node v
```


Traversals of Binary Trees

Inorder Traversal of a Binary Tree

- An additional traversal method for a binary tree is the inorder traversal.
- We visit a node between the recursive traversals of its left and right subtrees.
- The inorder traversal of the subtree rooted at a node v in a binary tree T is given:

Algorithm inorder(T,v):

```
if v has a left child u in T then
    inorder(T,u)
perform the "visit" action for node v
if v has a right child w in T then
    inorder(T,w)
```

- The inorder traversal of a binary tree T is visiting the nodes of T "from left to right"
- Indeed, for every node v, the inorder traversal visits v after all the nodes in the left subtree of v and before all the nodes in the right subtree of v.

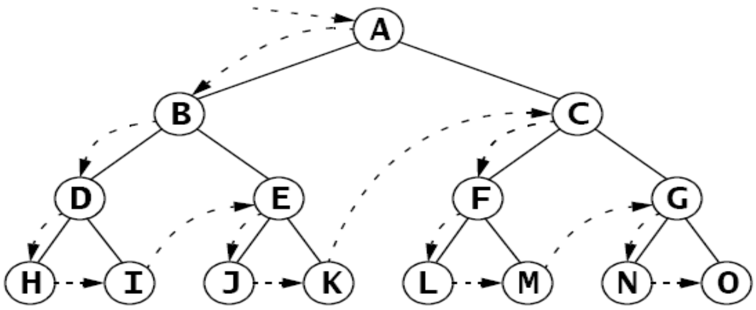


Figure The preorder traversal of a binary tree

The nodes are visited in the order **A, B, D, H, I, E, J, K, C, F, L, M, G, N, O.**

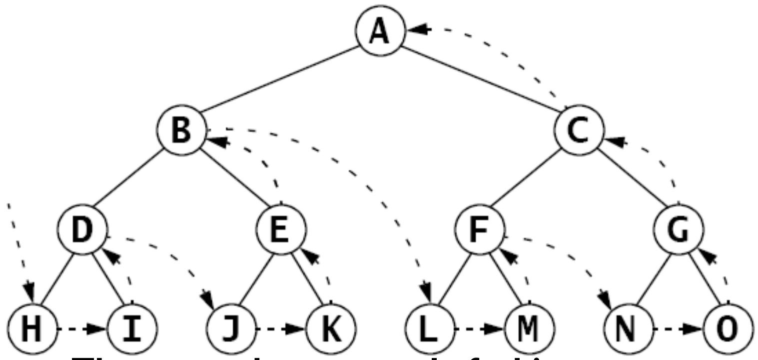


Figure The postorder traversal of a binary tree

The nodes are visited in the order **H, I, D, J, K, E, B, L, M, F, N, O, G, C, A.**

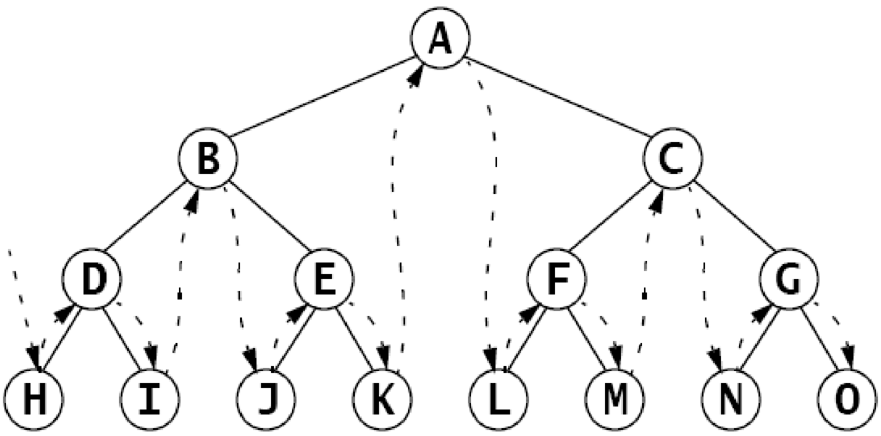


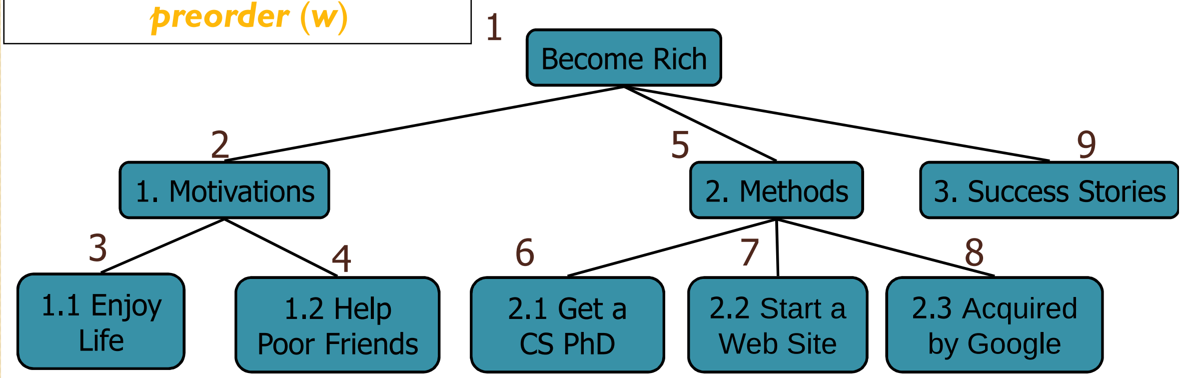
Figure 11.18 The inorder traversal of a binary tree

The nodes are visited in the order **H, D, I, B, J, E, K, A, L, F, M, C, N, G, O.**

Example Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

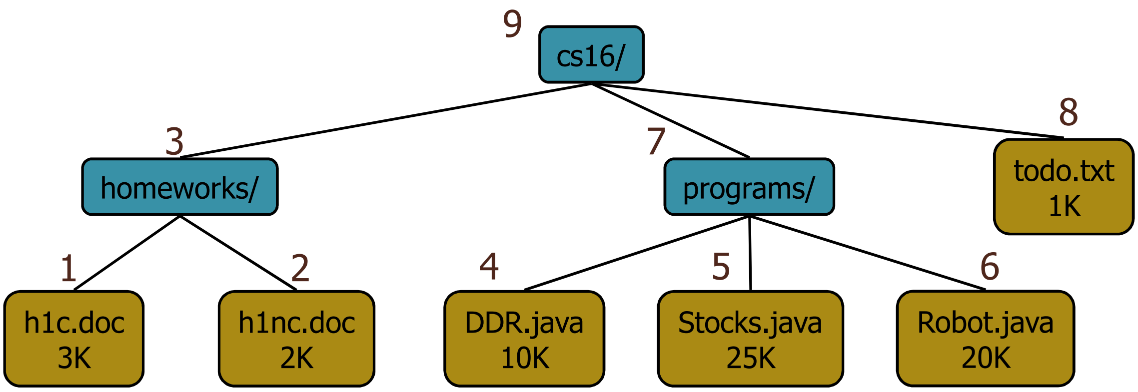
Algorithm ***preOrder***(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)



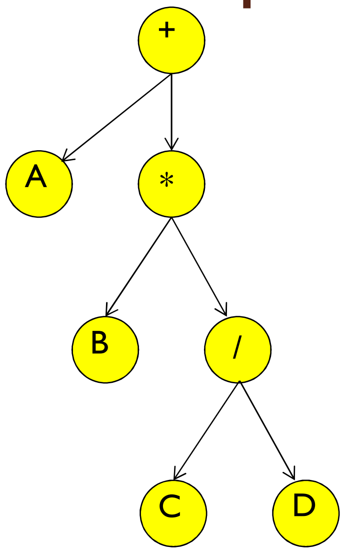
Example Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm ***postOrder***(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)



Example Traversing Trees



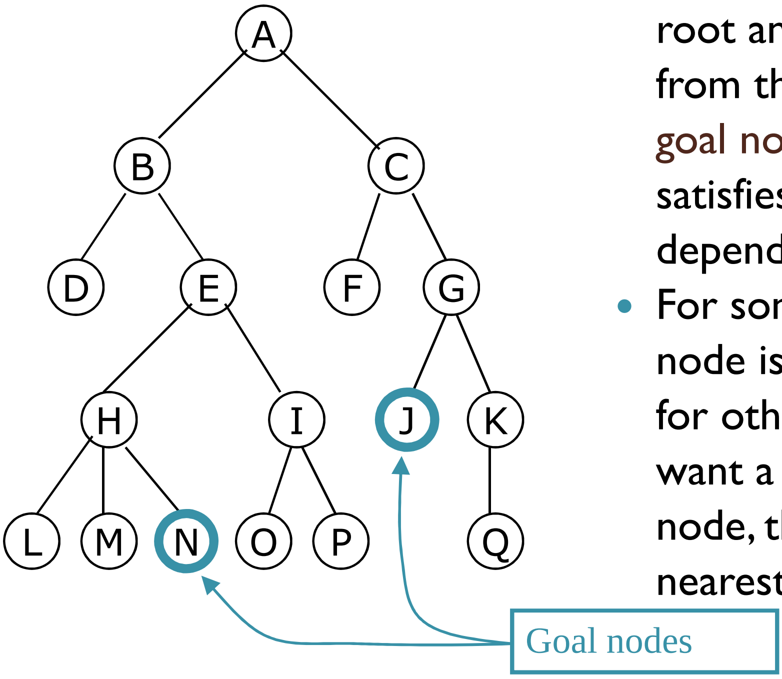
- Preorder: Root, then Children
  - + A * B / C D
- Postorder: Children, then Root
  - A B C D / * +
- Inorder: Left child, Root, Right child
  - A + B * C / D

Example Code for Recursive Preorder

```
void print_preorder ( TreeNode T)
{
  TreeNode P;
  if  ( T == NULL ) return;
  else  {  print_element(T.Element);
           P = T.FirstChild;
           while (P != NULL) {
               print_preorder ( P );
               P = P.NextSibling; }
         }
}
```

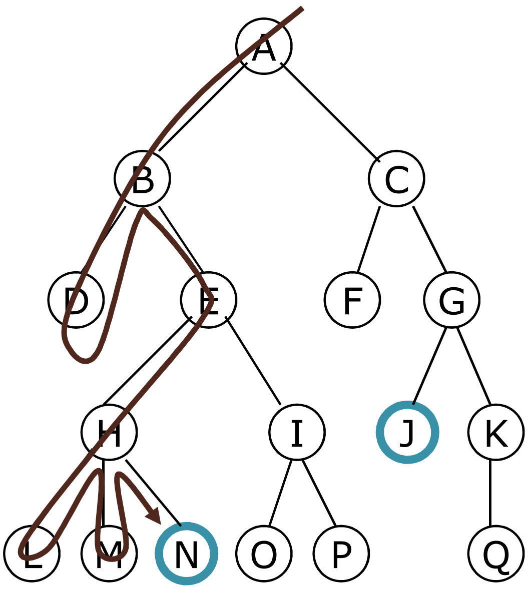
What is the running time for a tree with N nodes?

Tree searches



- A tree search starts at the root and explores nodes from there, looking for a goal node (a node that satisfies certain conditions, depending on the problem)
- For some problems, any goal node is acceptable (N or J); for other problems, you want a minimum-depth goal node, that is, a goal node nearest the root (only J)

Depth-first searching



- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

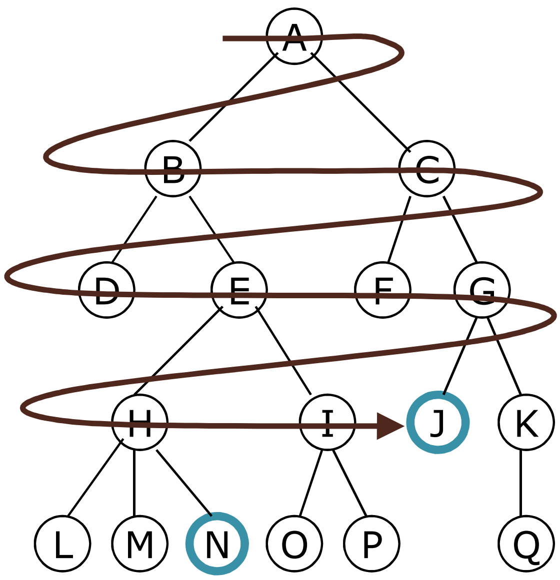
How to do depth-first searching

- Put the root node on a stack;
  while (stack is not empty) {
      remove a node from the stack;
      if (node is a goal node) return success;
      put all children of node onto the stack;
  }
  return failure;
- At each step, the stack contains some nodes from each of a number of levels
  - The size of stack that is required depends on the branching factor b
  - While searching level n, the stack contains approximately b*n nodes
- When this method succeeds, it doesn't give the path

Recursive depth-first search

- search(node):
      if node is a goal, {print node and return success;}
      for each child c of node {
          if search(c) is successful, {print c and return success;}
      }
      return failure;
- The (implicit) stack contains only the nodes on a path from the root to a goal
  - The stack only needs to be large enough to hold the deepest search path
  - When a solution is found, the path is on the (implicit) stack, and can be extracted as the recursion "unwinds"

Breadth-first searching



- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

How to do breadth-first searching

- ```
  Put the root node on a queue;
  while (queue is not empty) {
      remove a node from the queue;
      if (node is a goal node) return success;
      put all children of node onto the queue;
  }
  return failure;
  ```
- Just before starting to explore level n, the queue holds *all* the nodes at level n-1
- In a typical tree, the number of nodes at each level increases *exponentially* with the depth
- Memory requirements may be infeasible
- When this method succeeds, it doesn't give the path
- There is *no* "recursive" breadth-first search equivalent to recursive depth-first search
  ```

Comparison of algorithms

- Depth-first searching:
  - ```
    Put the root node on a stack;
    while (stack is not empty) {
        remove a node from the stack;
        if (node is a goal node) return success;
        put all children of node onto the stack;
    }
    return failure;
    ```
- Breadth-first searching:
  - ```
    Put the root node on a queue;
    while (queue is not empty) {
        remove a node from the queue;
        if (node is a goal node) return success;
        put all children of node onto the queue;
    }
    return failure;
    ```


Depth- vs. breadth-first searching

- When a breadth-first search succeeds, it finds a minimum-depth (nearest the root) goal node
- When a depth-first search succeeds, the found goal node is not necessarily minimum depth
- For a large tree, breadth-first search memory requirements may be excessive
- For a large tree, a depth-first search may take an excessively long time to find even a very nearby goal node
- How can we combine the advantages (and avoid the disadvantages) of these two search techniques?

Depth-limited searching

- Depth-first searches may be performed with a depth limit:

```
boolean limitedDFS(Node node, int limit, int depth) {
    if (depth > limit) return failure;
    if (node is a goal node) return success;
    for each child of node {
        if (limitedDFS(child, limit, depth + 1))
            return success;
    }
    return failure;
}
```

- Since this method is basically DFS, if it succeeds then the path to a goal node is in the stack

Depth-first iterative deepening

```
limit = 0;
found = false;
while (not found) {
    found = limitedDFS(root, limit, 0);
    limit = limit + 1;
}
```

- This searches to depth 0 (root only), then if that fails it searches to depth 1, then depth 2, etc.
- If a goal node is found, it is a nearest node *and* the path to it is on the stack
  - Required stack size is limit of search depth (plus 1)

Time requirements for depth-first iterative deepening on binary tree

Nodes at each level	Nodes searched by DFS	Nodes searched by iterative DFS
1	1	1
2	+2 = 3	+3 = 4
4	+4 = 7	+7 = 11
8	+8 = 15	+15 = 26
16	+16 = 31	+31 = 57
32	+32 = 63	+63 = 120
64	+64 = 127	+127 = 247
128	+128 = 255	+255 = 502

Time requirements on tree with branching factor 4

Nodes at each level	Nodes searched by DFS	Nodes searched by iterative DFS
1	1	1
4	+4 = 5	+5 = 6
16	+16 = 21	+21 = 27
64	+64 = 85	+85 = 112
256	+256 = 341	+341 = 453
1024	+1024 = 1365	+1365 = 1818
4096	+4096 = 5461	+5461 = 7279
16384	+16384 = 21845	+21845 = 29124

Iterative deepening: summary

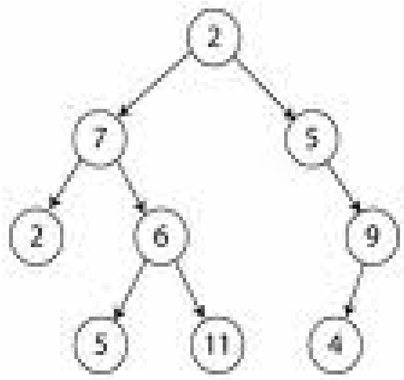
- When searching a binary tree to depth 7:
  - DFS requires searching 255 nodes
  - Iterative deepening requires searching 502 nodes
  - Iterative deepening takes only about twice as long
- When searching a tree with branching factor of 4 (each node may have four children):
  - DFS requires searching 21845 nodes
  - Iterative deepening requires searching 29124 nodes
  - Iterative deepening takes about 4/3 = 1.33 times as long
- The higher the branching factor, the lower the relative cost of iterative deepening depth first search

Other search techniques

- Breadth-first search (BFS) and depth-first search (DFS) are the foundation for all other search techniques
- We might have a weighted tree, in which the edges connecting a node to its children have differing "weights"
  - We might therefore look for a "least cost" goal
- The searches we have been doing are blind searches, in which we have no prior information to help guide the search
  - If we have some measure of "how close" we are to a goal node, we can employ much more sophisticated search techniques
  - We will *not* cover these more sophisticated techniques
- Searching a graph is very similar to searching a tree, except that we have to be careful not to get caught in a cycle
  - We *will* cover some graph searching techniques

How to search a binary tree?

(1) Start at the root
(2) Search the tree level by level, until you find the element you are searching for or you reach a leaf.

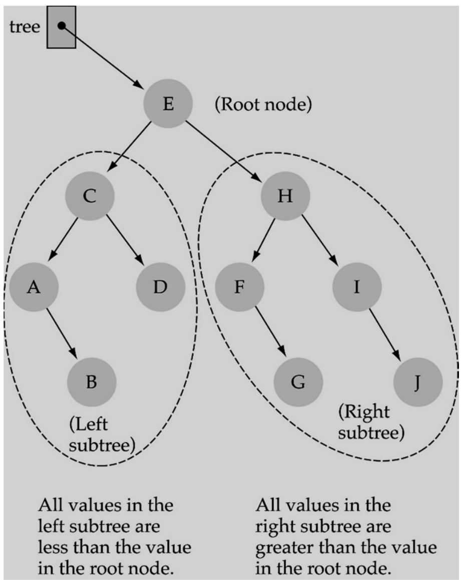


Is this better than searching a linked list?

No → O(N)

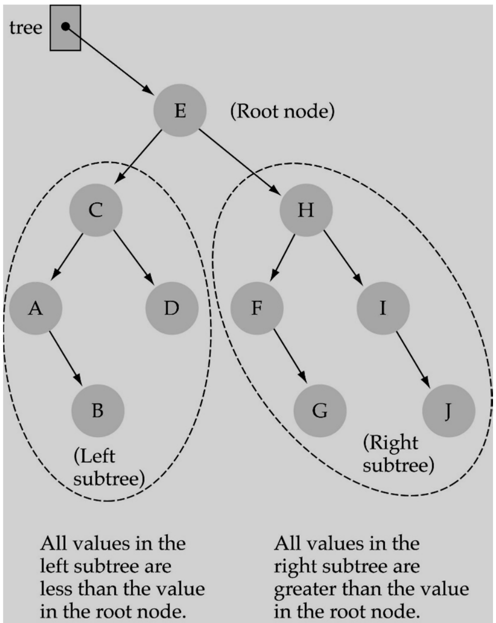
Binary Search Trees (BSTs)

- **Binary Search Tree Property**: The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



Binary Search Trees (BSTs)

In a BST, the value stored at the root of a subtree is *greater* than any value in its left subtree and *less* than any value in its right subtree!



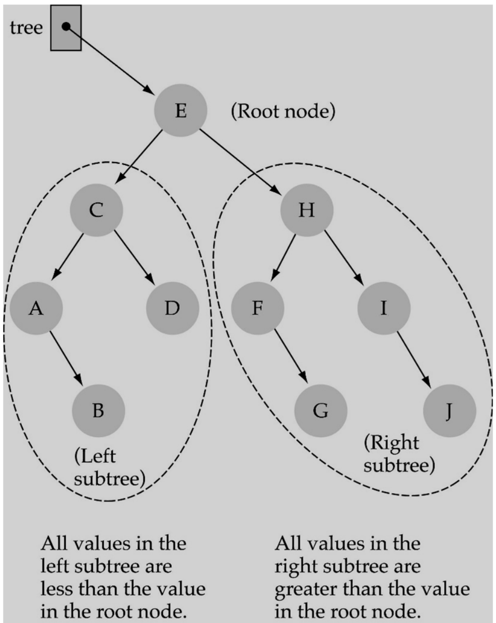
Binary Search Trees (BSTs)

Where is the smallest element?

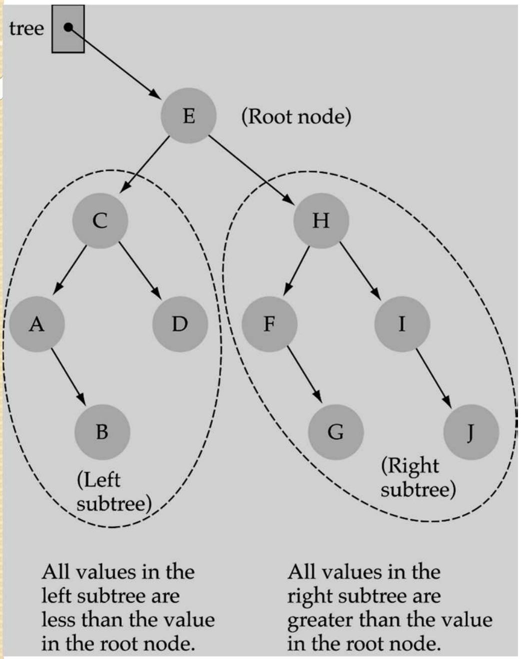
Ans: leftmost element

Where is the largest element?

Ans: rightmost



How to search a binary search tree?

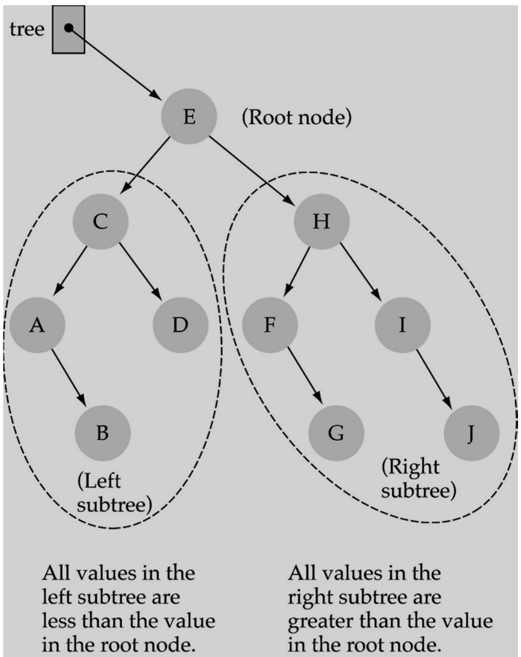


(1) Start at the root

(2) Compare the value of the item you are searching for with the value stored at the root

(3) If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*

How to search a binary search tree?



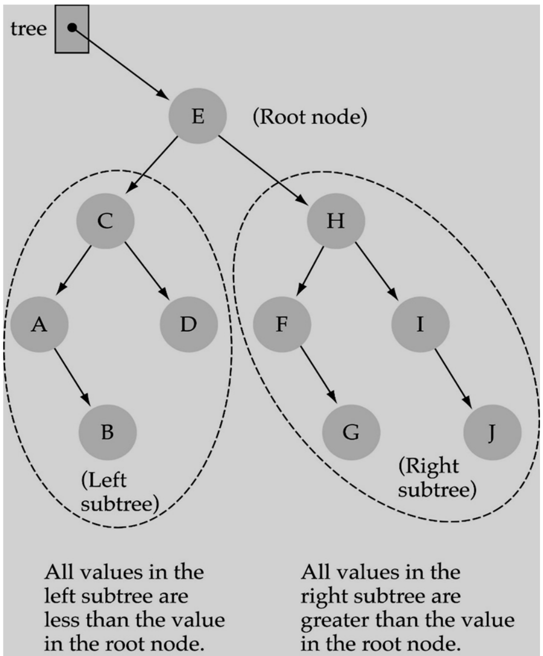
(4) If it is less than the value stored at the root, then search the left subtree

(5) If it is greater than the value stored at the root, then search the right subtree

(6) Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

How to search a binary search tree?

this better than searching a linked list?



Yes !! ---> O(logN)

Binary Tree Implementation

```
Class Node {
    int data; // Could be int, a class, etc
    Node *left, *right;    // null if empty

    void insert ( int data ) { … }
    void delete ( int data ) { … }
    Node *find ( int data ) { … }
        …
}
```

Iterative Search of Binary Tree

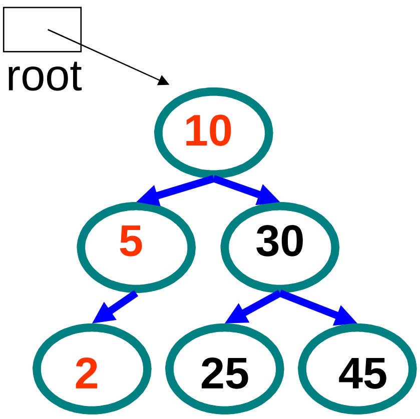
```
Node *Find( Node *n, int key) {
    while (n != NULL) {
        if (n->data == key)  // Found it
            return n;
        if (n->data > key)         // In left subtree
            n = n->left;
        else                       // In right subtree
            n = n->right;
    }
    return null;
}
Node * n = Find( root, 5);
```


Recursive Search of Binary Tree

```
Node *Find( Node *n, int key) {
    if (n == NULL)            // Not found
        return( n );
    else if (n->data == key)       // Found it
        return( n );
    else if (n->data > key) // In left subtree
        return Find( n->left, key );
    else                           // In right subtree
        return Find( n->right, key );
}
Node * n = Find( root, 5);
```

Example Binary Searches

Find (root, 2)



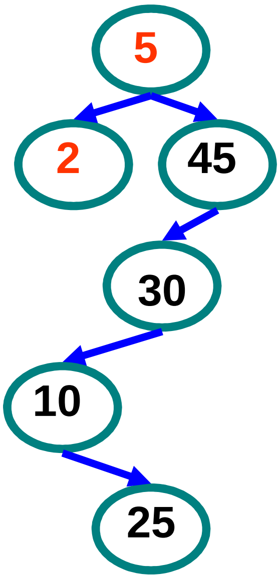
10 > 2, left

5 > 2, left

2 = 2, found

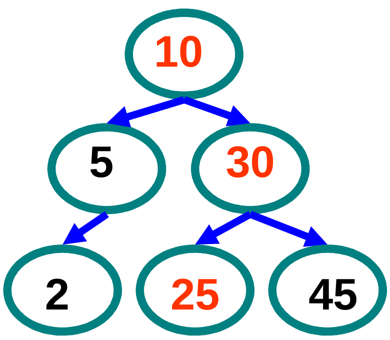
5 > 2, left

2 = 2, found



Example Binary Searches

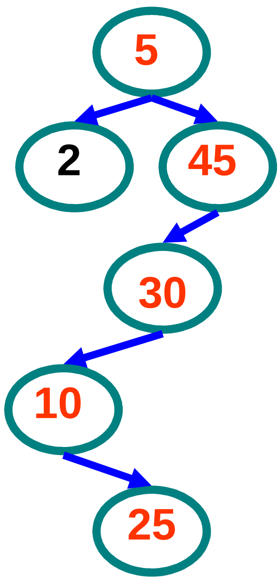
Find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

25 = 25, found