Tree traversals and binary trees

Comp Sci 1575 Data Structures

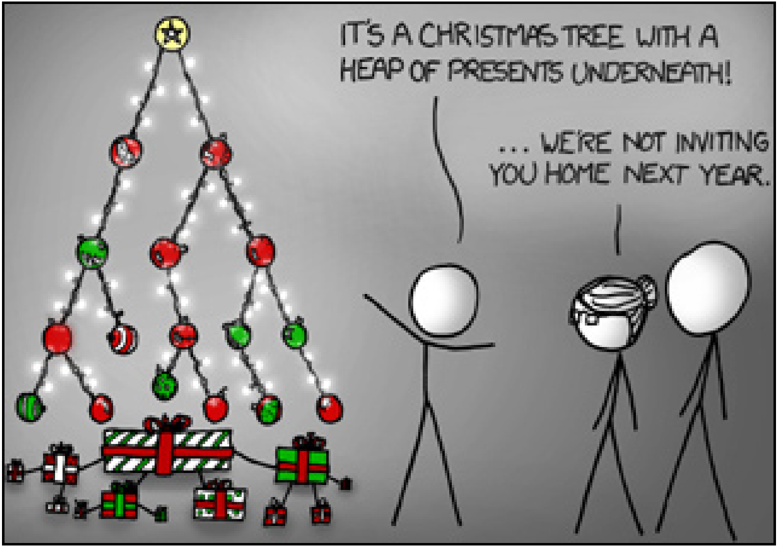
Missouri S&T | Computer Science

Outline

Definitions
Context
Definition
Parts of a tree
Root, leaves
Parents, children
Left, right
Internal nodes
Edges
KQ
Features of trees and nodes
Levels, depth, height
Paths
Degree
Examples
These are trees
There are NOT trees
Example trees
Binary trees
ADT
Variations
Example application
KQ

Outline

Trees!


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

Tree is a non-linear ordered ADT

```
                        COLLECTION
                       /          \
UNORDERED_COLLECTION    ORDERED_COLLECTION
          |                       |                |
         BAG                   LINEAR          NON_LINEAR
          |                      |                 |            |
         SET              LINKED_LIST      TREE       DIRECTED_GRAPH
          |                       |                 |                    |
   HASH_TABLE               DEQUE       BIN_TREE  UNDIRECTED_GRAPH
                                 /       \         /       \
                          STACK   QUEUE  BST    HEAP
```

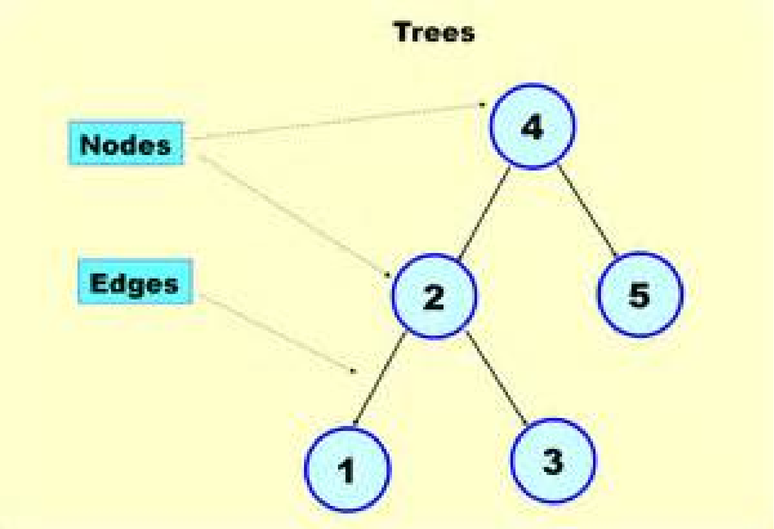
Outline

Trees are upside down in computer science

Real tree: leaves at the top, root(s) at the bottom

Computer science tree: root at the top, leave(s) at the bottom

Trees are composed of nodes and edges



- **Node** is an element in a tree
- **Edge** is the connection between one node and another

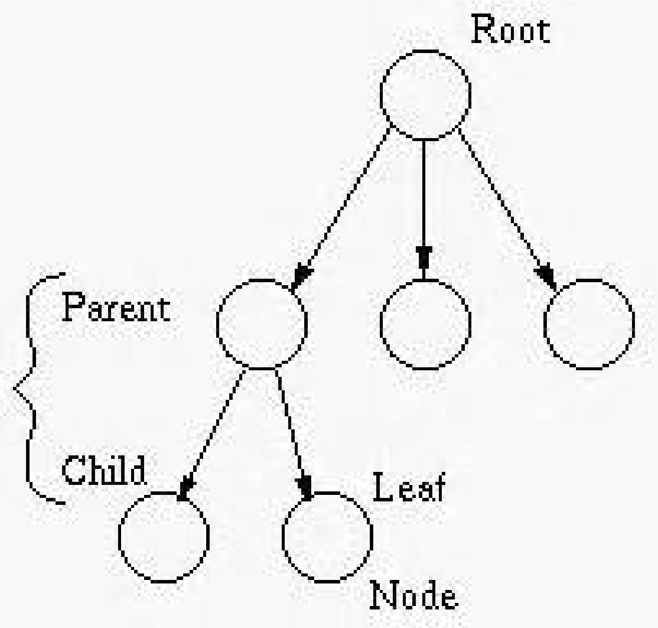
Tree

- Widely used abstract data type (ADT), or data structure implementing the ADT, that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.
- A (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle.

Outline

Outline

Root at the top, leaves at the bottom

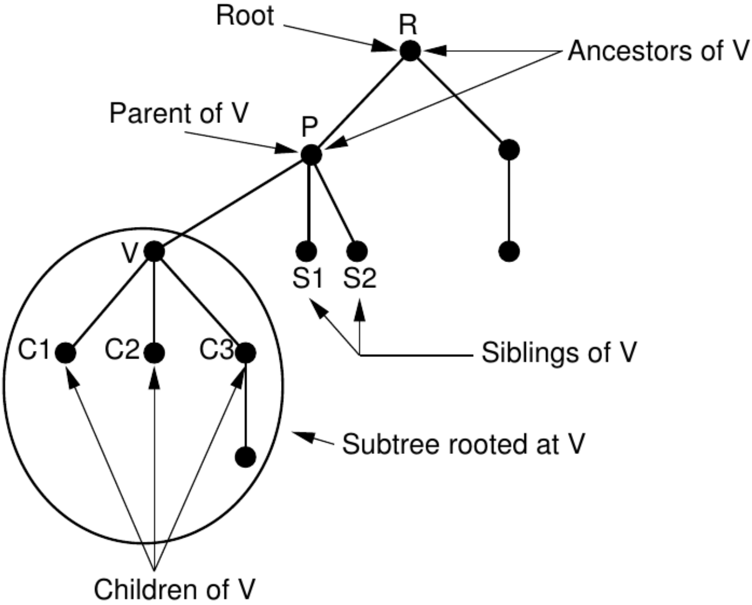


Family matters

- **Root** is the node at the top of the tree, and has no parent. There is only one root per tree and one path from the root node to any node.
- **Leaves** are bottom nodes without any sub-trees or children (less commonly called External node)

Outline

Parents are ancestors of children

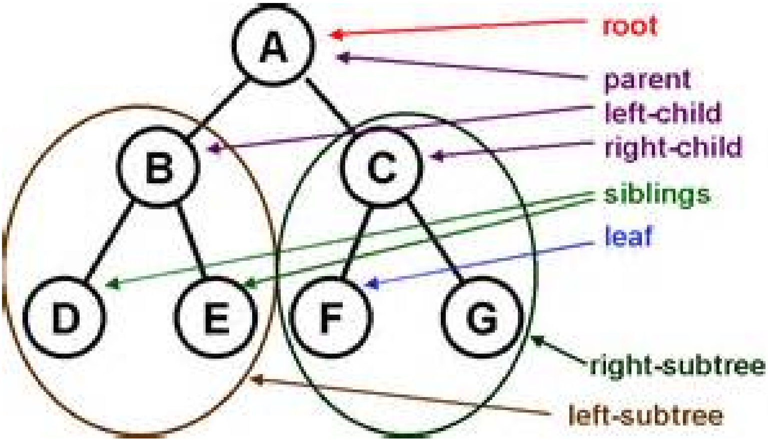


Family matters

- **Parent** of a node is the single node linked directly above it. Any node except the root node has one edge upward to a node called parent. Parent is the converse notion of a child.
- **Child** of a node is a node linked directly below it, directly connected by its edge downward, when moving away from the root
- **Siblings** are an n group of nodes with the same parent.
- **Ancestor** is any node from which a node descends directly or indirectly, which is any node reachable by repeated proceeding from child to parent.
- **Descendant** is any node that descends from a node directly or indirectly, which is any node reachable by repeated proceeding from parent to child.
- If there is a path from n_1 to n_2 , then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1 .

Outline

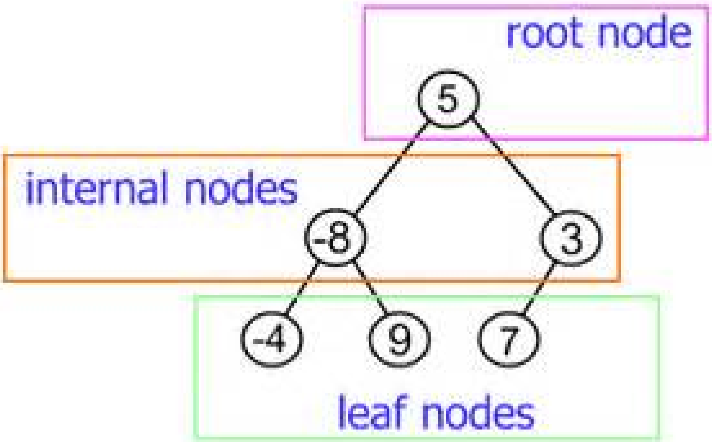
Left and right are defined for nodes and sub-trees



- **Sub-tree** is a smaller tree 'rooted' by some particular node in the tree, which are descendants of that node.

Outline

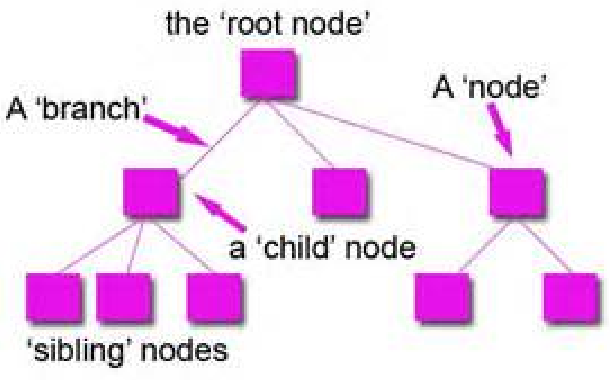
Internal nodes



- **Internal node** is a non-root node with at least one child.

Outline

Edges are sometimes called branches



- That there are $n-1$ **edges** follows from the fact that each edge connects some node to its parent, and every node except the root has one parent
- If a tree has n nodes, then it must have $n-1$ edges, as every node is connected to a parent, except for the root.

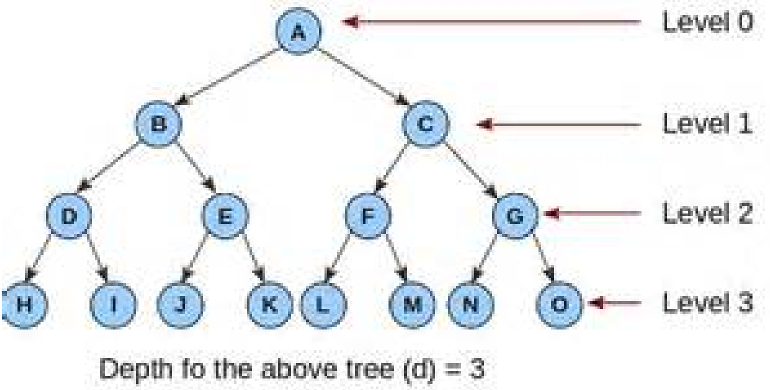
Outline

KQ and sketchpad

Outline

Outline

Trees have levels



Trees and nodes have depth and height

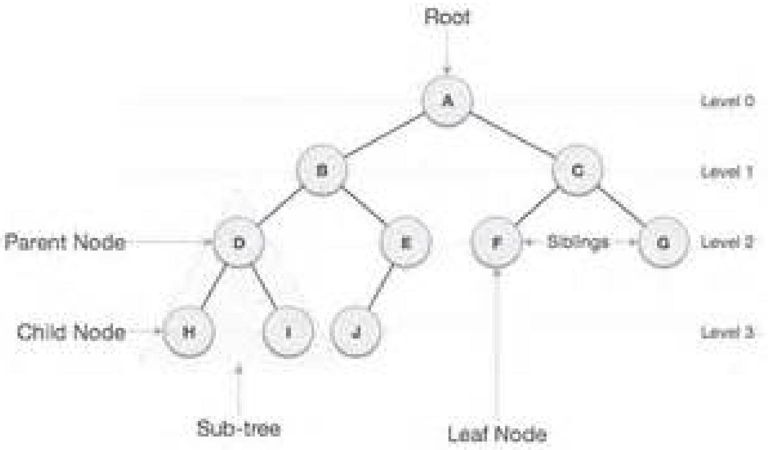
Depth

- **Level/depth of a node** represents the familial generation of a node, or the length of the path from the root to a node. The level of a node is defined by 1 + (the number of levels between the node and the root(0)). If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on. In other words, the number of edges from the tree's root node to the node.
- **Level/depth of tree** is length of the longest path from the root to the deepest leaf, or the maximum depth of any leaf node The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.

Height

- **height (opposite concept of level/depth) of a node,** n_i is the length of the longest path from n_i to a leaf. Thus all leaves are at height 0.
- **Height of a tree** is equal to the height of the root.

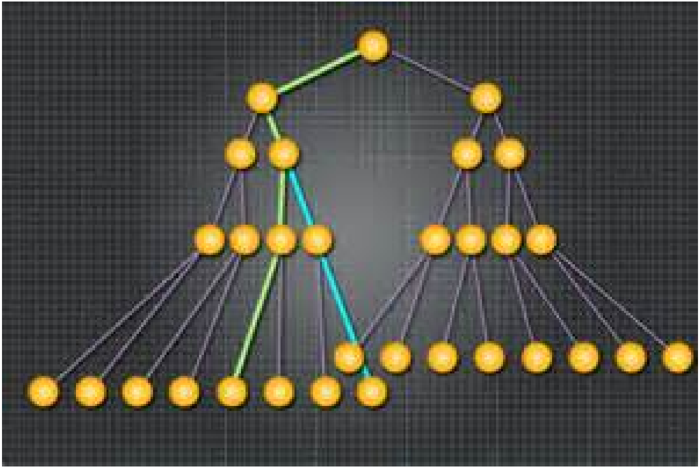
Trees have levels



- Depth: root is 0; its children are 1, etc.
- What is height of these nodes?

Outline

Trees have paths



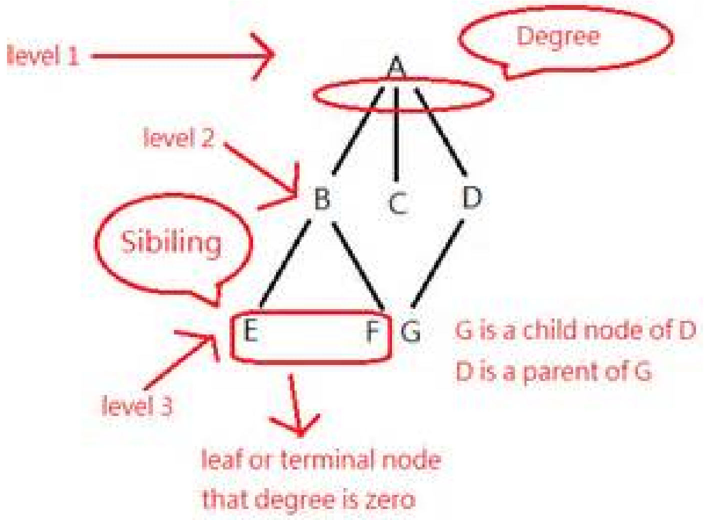
- **Path** is a sequence of nodes and edges connecting a node with a descendant (green or blue above for two different leaf nodes)
- Only one path from the root to each node.

Trees have paths

- A **path** from node n_1 to n_k is defined as a sequence of nodes $n_1,\ n_2,\ ...,\ n_k$ such that n_i is the parent of $n_i + 1$ for $1 \leq i < k$.
- The **length** of this path is the number of edges on the path, namely, $k - 1$.
- There is a path of length zero from every node to itself.
- With the root is at depth 0, for any node n_i, the depth of n_i is the length of the unique path from the root to n_i .

Outline

Degree of nodes and trees

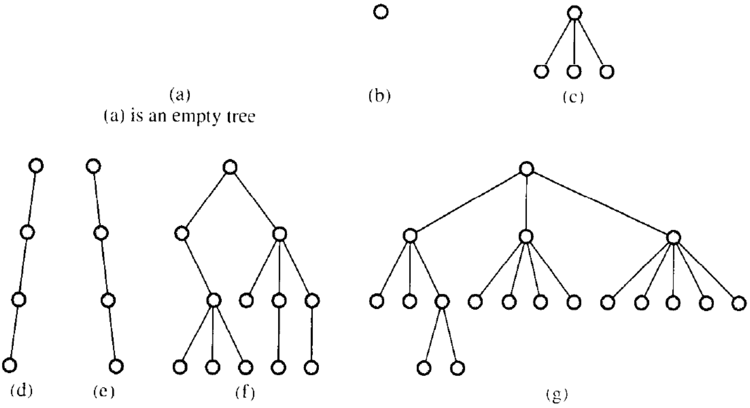


- **Degree of a node** is the number of subtrees of a node.
- **Degree of a tree** is the largest degree of any node in a tree.

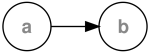
Outline

Outline

These are all trees

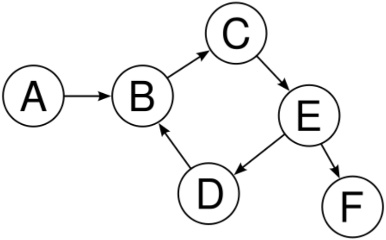


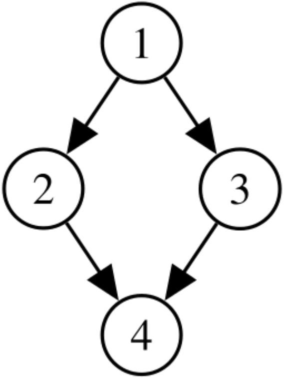
- Even single nodes can be considered trees
- Forest is a set of $n \geq 0$ disjoint trees.
- A list is trivially a tree:



Outline

These are NOT trees

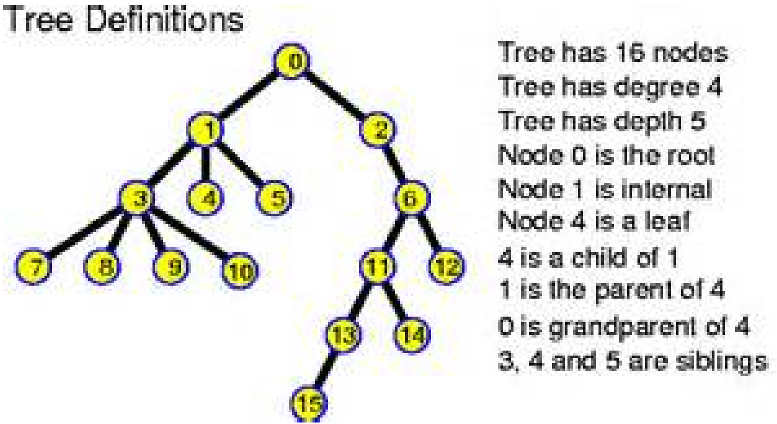




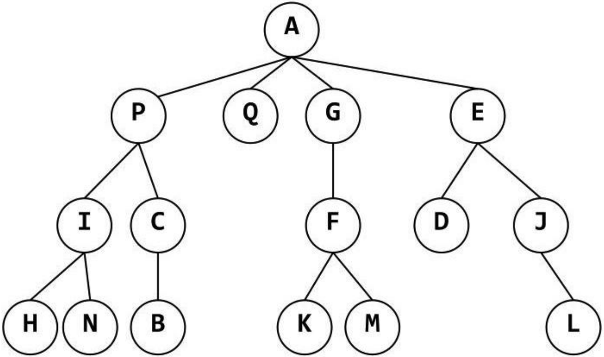
- No loops or multi-parent children

Outline

Example tree



Example tree of characters

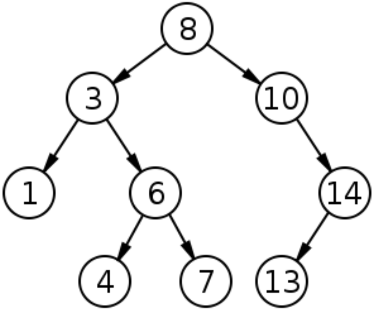


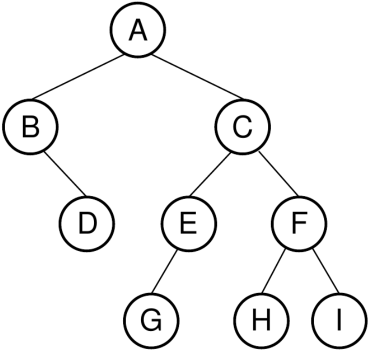
- Degree of tree
- Degree of each node
- Depth of tree
- Depth of each node
- Height?

Outline

Outline

Binary trees





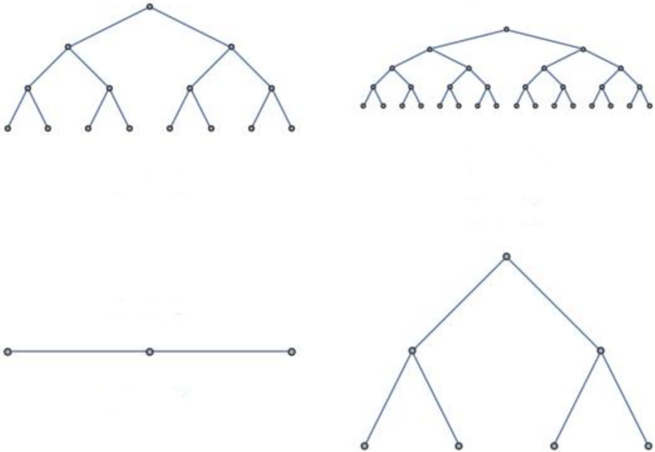
Binary tree

- Set of nodes is either empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root; disjoint means that they have no nodes in common.
- Each node has at most two children, which are referred to as the left child and the right child.
- No node can have more than two children.
- The depth of an average binary tree is considerably smaller than N, the average depth is O(sqrt(N)), and for a special type of binary tree, namely the binary search tree, the average value of the depth is O(log N).

Outline

Perfect binary tree

- Binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



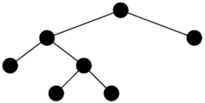
Full (left) and complete (right) binary trees

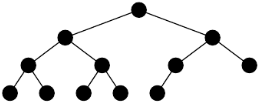
Full binary tree (left below)

- Tree in which every node in the tree has either 0 or 2 children.
- Each node in a full binary tree is either
  (1) an internal node with exactly two non-empty children
  (2) a leaf.

Complete binary tree (right below)

- Has a restricted shape obtained by starting at the root and filling the tree by levels from left to right.
- Every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- The bottom level has its nodes filled in from the left side.
- Can be efficiently represented using an array.





Full and complete binary trees

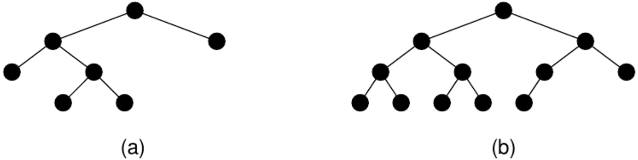
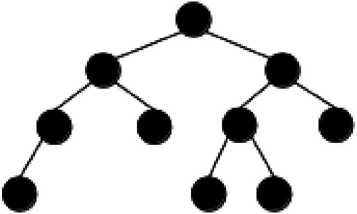


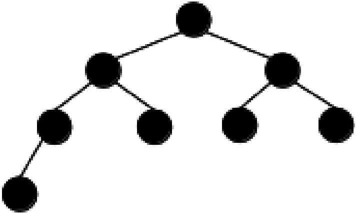
Figure 5.3 Examples of full and complete binary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

Full and complete binary trees

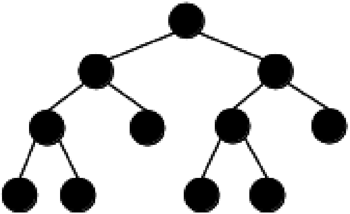
Neither complete nor full



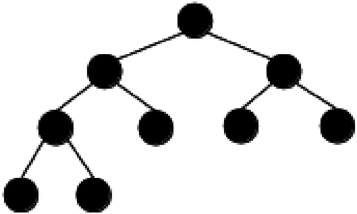
Complete but not full



Full but not complete



Complete and full



Balanced binary trees

- Has the minimum possible maximum height (a.k.a. depth) for the leaf nodes, because for any given number of leaf nodes, the leaf nodes are placed at the greatest height possible.
- One common balanced tree structure is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.
- One may also consider binary trees where no leaf is much farther away from the root than any other leaf. (Different balancing schemes allow different definitions of "much farther".)

Degenerate trees

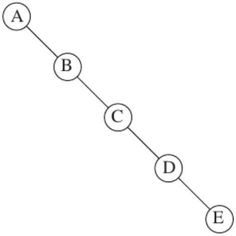


Figure 4.12 Worst-case binary tree

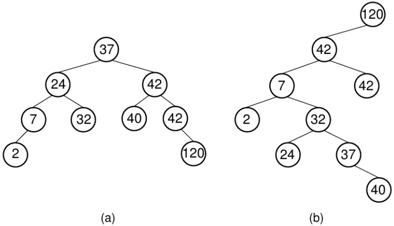


Figure 5.13 Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

Left A degenerate (or pathological) tree is where each parent node has only one associated child node; performance will behave like a linked list data structure.

Right Shape of the binary search tree depends entirely on the order of insertions and deletions, and can become degenerate.

Outline

Example application: expression trees

- Expression tree: The leaves are operands, such as constants or variable names, and the other nodes contain operators.

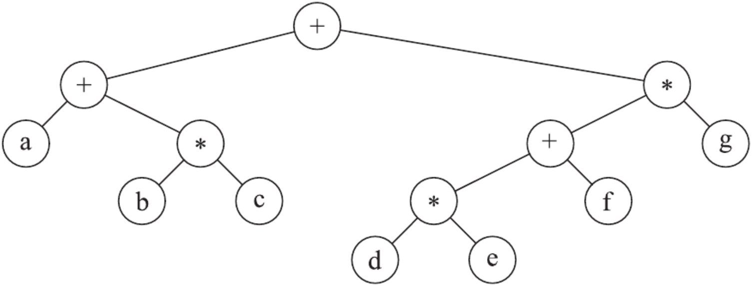


Figure 4.14 Expression tree for `(a + b * c) + ((d * e + f ) * g)`

Outline

KQ and sketchpad

Outline

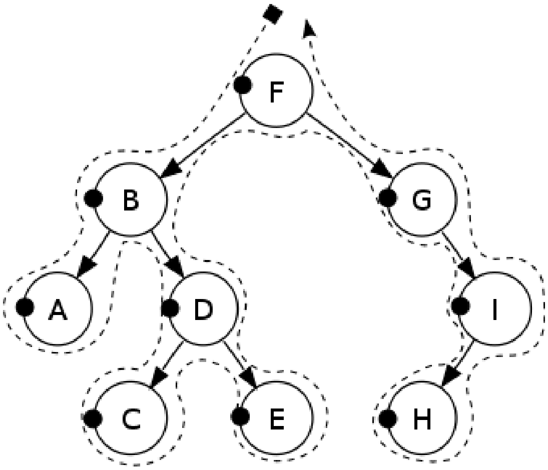
Traversals

- Order of list is obvious, tree not so much
- Any process for visiting all of the nodes in some order is called a traversal.
- Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes.

Outline

Pre-order

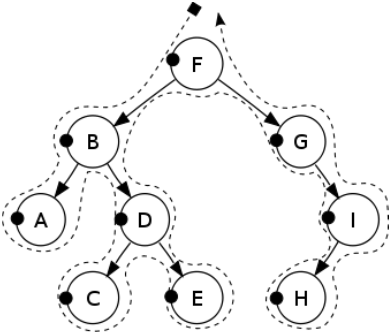
- Pre-order traversal: parents is visited before the children



Pre-order: F, B, A, D, C, E, G, I, H

Pre-order Recursive

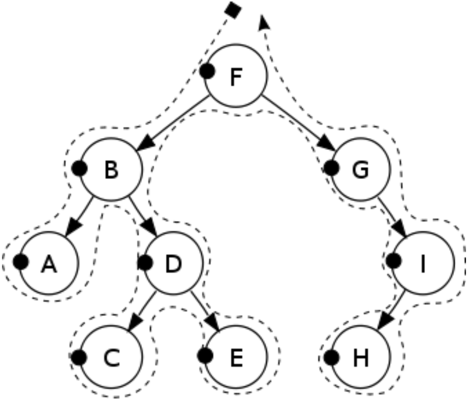
1. Check if the current node is empty / null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order function.
4. Traverse the right subtree by recursively calling the pre-order function.



Pre-order: F, B, A, D, C, E, G, I, H

Pre-order Recursive

1. Process the root
2. Process the nodes in the left subtree with a recursive call
3. Process the nodes in the right subtree with recursive call

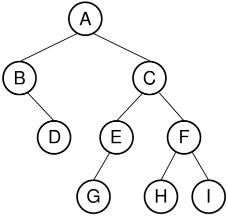


Pre-order: F, B, A, D, C, E, G, I, H.

Pre-order recursive pseudocode algorithm

Assuming each node is referenced to via a pointer called 'node', has a left and right pointer, and that leaves have two null pointers:

```
preorder(node)
    if(node == null)
        return
    visit(node)
    preorder(node.left)
    preorder(node.right)
```



What is trace on this tree?

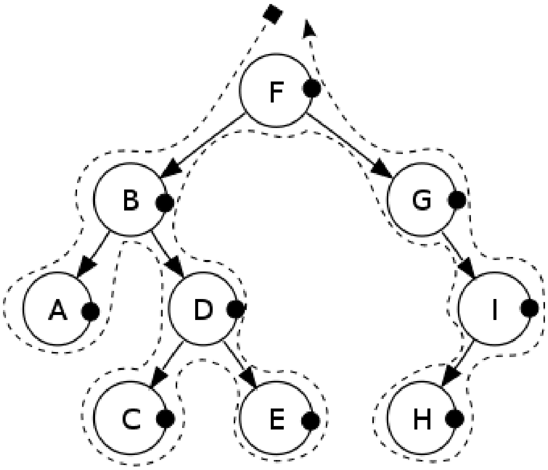
Pre-order iterative pseudocode algorithm


```
iterativePreorder(node)
  if(node == null)
    return
  s = empty stack
  s.push(node)
  while(not s.isEmpty())
    node = s.pop()
    visit(node)
    //right child is pushed first so that
    //left is processed first
    if (node.right != null)
      s.push(node.right)
    if (node.left != null)
      s.push(node.left)
```

Outline

Post-order

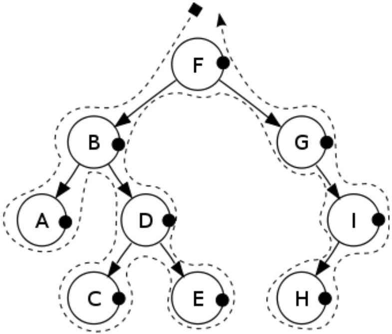
- Post-order traversal: children are visited before the parent



Post-order: A, C, E, D, B, H, I, G, F.

Post-order recursive

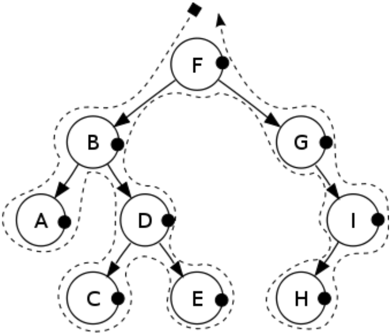
1. Check if the current node is empty / null.
2. Traverse the left subtree by recursively calling the post-order function.
3. Traverse the right subtree by recursively calling the post-order function.
4. Display the data part of the root (or current node).



Post-order: A, C, E, D, B, H, I, G, F.

Post-order recursive

1. Process the nodes in the left subtree with a recursive call
2. Process the nodes in the right subtree with recursive call
3. Process the root

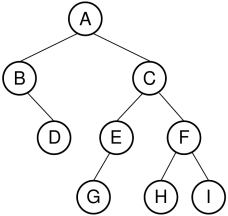


Post-order: A, C, E, D, B, H, I, G, F.

Post-order recursive pseudocode algorithm

Assuming each node is referenced to via a pointer called 'node', has a left and right pointer, and that leaves have two null pointers:

```
postorder(node)
    if(node == null)
        return
    postorder(node.left)
    postorder(node.right)
    visit(node)
```



What is trace on this tree?

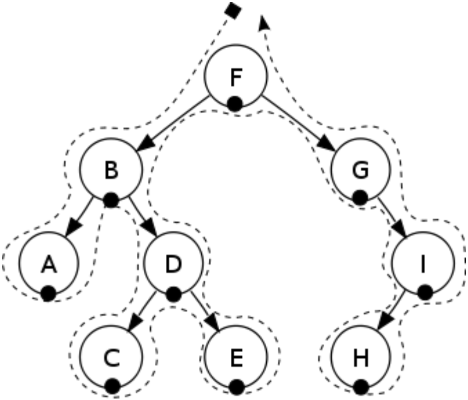
Post-order iterative pseudocode algorithm


```
iterativePostorder(node)
  s = empty stack
  lastNodeVisited = null
  while (not s.isEmpty() or node != null)
    if(node != null)
      s.push(node)
      node = node.left
    else
      peekNode = s.peek()
      // if right child exists and traversing 
      // from left child, then move right
      if(peekNode.right != null and
      lastNodeVisited != peekNode.right)
        node = peekNode.right
      else
        visit(peekNode)
        lastNodeVisited = s.pop()
```

Outline

In-order

- An inorder traversal first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree).
- Binary trees only

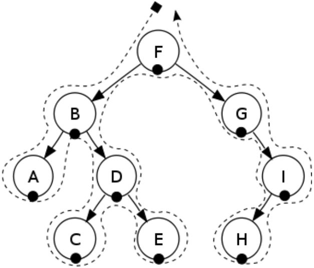


In-order: A, B, C, D, E, F, G, H, I.

In-order: recursive

1. Check if the current node is empty / null.
2. Traverse the left subtree by recursively calling the in-order function.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order function.

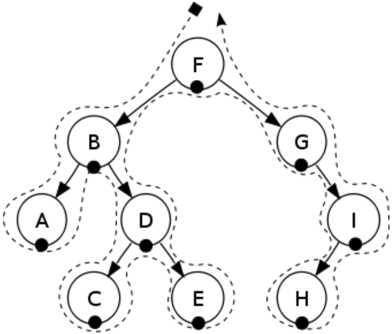
Why just for binary?



In-order: A, B, C, D, E, F, G, H, I.

In-order: recursive

1. Process the nodes in the left subtree with a recursive call
2. Process the root
3. Process the nodes in the right subtree with recursive call

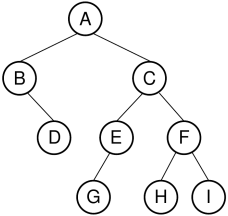


In-order: A, B, C, D, E, F, G, H, I.

In-order: recursive

Assuming each node is referenced to via a pointer called 'node', has a left and right pointer, and that leaves have two null pointers:

```
inorder(node)
    if(node == null)
        return
    inorder(node.left)
    visit(node)
    inorder(node.right)
```



What is trace on this tree?

In-order: iterative

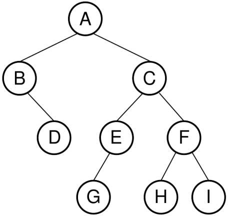
```
iterativeInorder(node)
  s = empty stack
  while(not s.isEmpty() or node != null)
    if(node != null)
      s.push(node)
      node = node.left
    else
      node = s.pop()
      visit(node)
      node = node.right
```

Outline

Backward in-order recursive

Useful for printing if indent each item to its depth in the tree

1. Process the nodes in the right subtree with a recursive call
2. Process the root
3. Process the nodes in the left subtree with a recursive call



What is trace on this tree?

Outline

Generalization of traversal to non-binary

1. Perform pre-order operation.
2. For each *i* from 1 to the number of children do:
   1. Visit i-th, if present.
   2. Perform in-order operation.
3. Perform post-order operation.

Outline

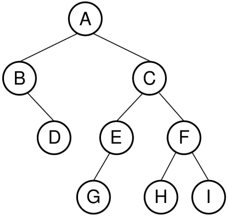
8 Traversals
- Pre-order
  - Recursive
  - Pre-order
- Post-order
  - Recursive
  - Iterative
- In-order
  - Recursive
  - Iterative
- Backward in-order
  - Recursive
- Generalization
- Examples

9 KQ

10 Apply

Counting example

```
template <typename E>
int count(BinNode<E> *root)
{
    // if Nothing to count
    if(root == NULL)
        return 0;
    return 1 + count(root->left())
        + count(root->right());
}
```



What is trace on this tree?

Outline

KQ and sketchpad

Outline

Apply a function to a whole tree?

- Functions as parameters to functions.
- Pass as parameter: *void func(int&)*

Whichever func used has to accept one int


```
void apply(void func(int&), int data[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        func(data[i]);
    }
}

void seven_up(int &i)
{
    i += 7;
}

apply(seven_up, data, 10);
```

func() now can have different parameter types

```
template <typename T>
void apply(void func(T&), int data[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        func(data[i]);
    }
}

// function that takes a character instead

apply(convert_to_upper, data, 10);
```

Only requirement is a single parametef for func


```
template <typename T>
void apply(T func, int data[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        func(data[i]);
    }
}

...

apply(convert_to_upper, data, 10);
apply(seven_up, data, 10);
...
```

Apply any operation to whole tree

```
template <typename T, typename E>
void preorder(T func, BTNode<E> *node_ptr)
{
  if(node_ptr != NULL)
  {
    func(node_ptr->data);
    preorder(func, node_ptr->left);
    preorder(func, node_ptr->right);
  }
}

void seven_up(int & i)
{
  i+=7;
}

preorder(seven_up, node_ptr);
```