CHAPTER 7

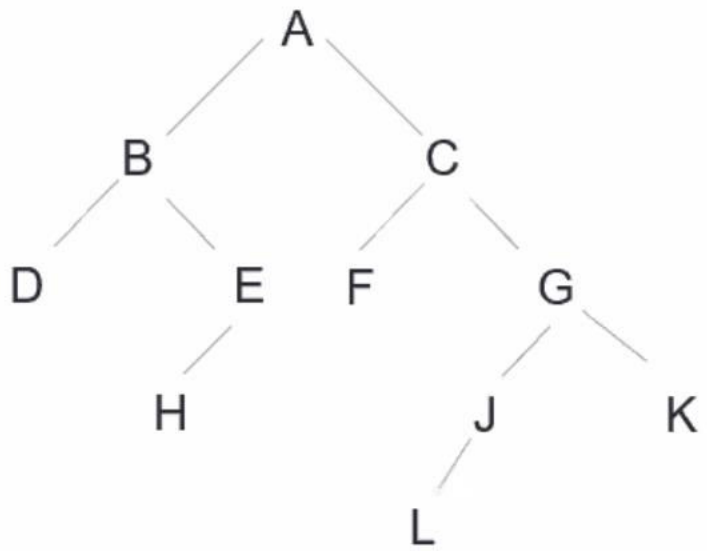
Tree

Tree and Binary Tree

- Tree is a nonlinear data structures used mainly to represent data containing a hierarchical relationship between elements.
- Binary Tree: A binary tree T is defined as a finite set of elements, called nodes, such that:
  a) T is empty (called the null tree or empty tree), or
  b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.
- If T does contain a root R, then the two trees T_1 and T_2 are called respectively the left and right subtrees of R.
- A left downward slanted line from a node N indicates a left successor of N, and a right downward slanted line from N indicates a right successor of N.

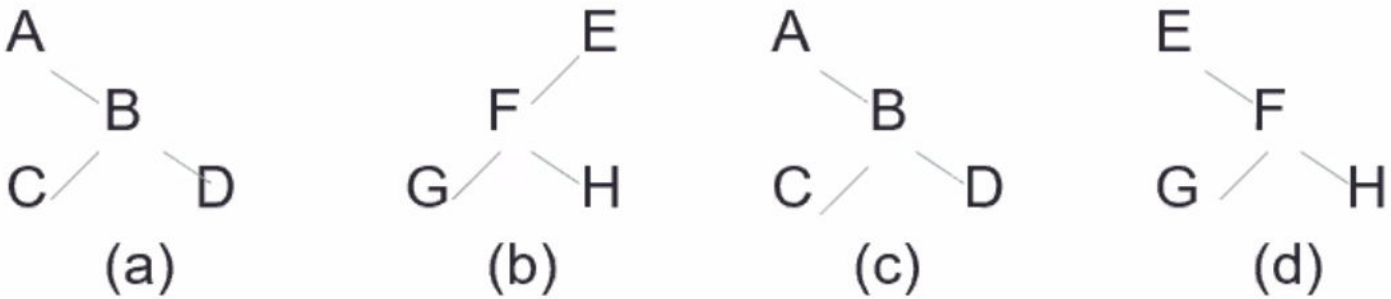
Binary Tree Continue…

- In the figure below A is the root.
- B and C are called the left and right successor of A.
- The left subtree of the root A consist of the nodes B, D, E and H, and the right subtree of the root A consists of the nodes C, F, G, J, K and L.



Binary Tree Continue…

- Binary trees T and P are called similar if they have the same structure.
- Binary trees T and Q are called copies if they are similar and if they have same contents at corresponding nodes.



- (a), (c) and (d) are similar.
- (a) and (c) are copies.
- (b) is neither similar nor a copy of those tree.

Terminology

- Suppose N is a node of T with left successor S_1 and right successor S_2, then N is called the parent of S_1 and S_2.
- Analogously, S_1 is called the left child or son of N and S_2 is called the right child or son of N.
- S_1 and S_2 are called siblings (or brothers).
- A node L is called a descendent of a node N (and N is called an ancestor of L) if there is a succession of children from N to L.
- Each node in a binary tree is assigned a level number start from 0; nodes with the same level number are said to belong to the same generation.
- The depth (or height) of a tree is the maximum no. of nodes in a branch of T. This turns out to be 1 more than the largest level no. of T.

Complete Binary Tree

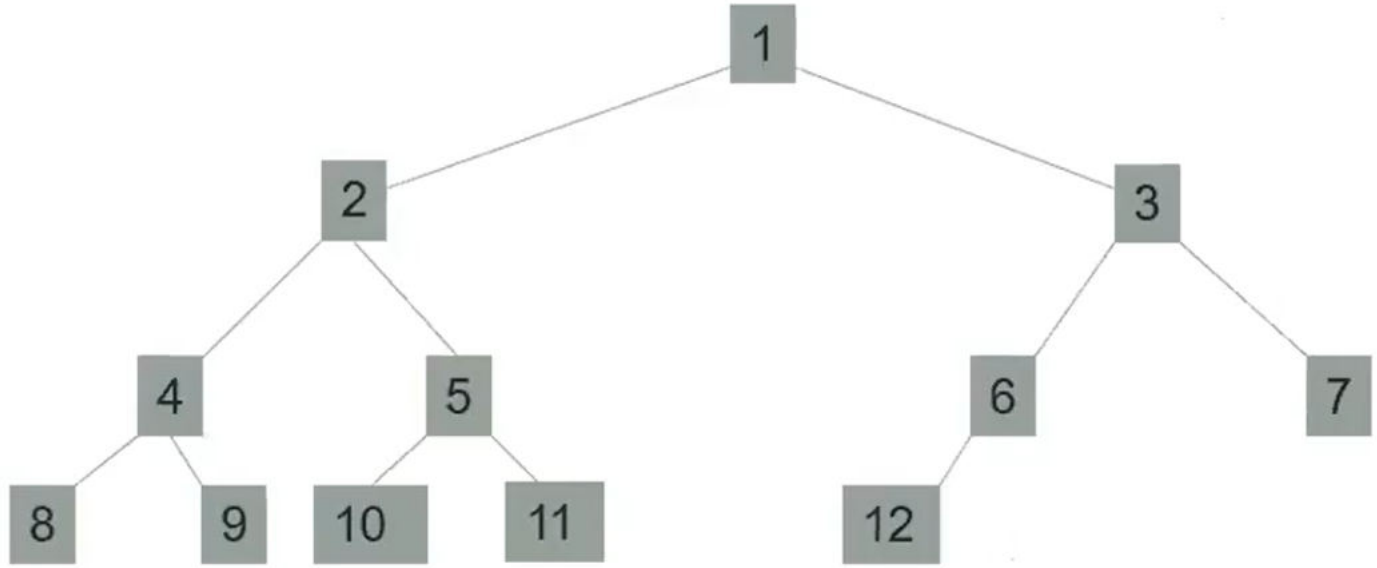
- The tree T is called Complete if all its levels, except possibly the last, have the maximum no. of possible nodes, and if all the nodes at the last level appear as far left as possible.
- The left and right child of node k is 2*k and 2*k+1 respectively. Ex. If k=10, left child=20 right child=21.
- The parent of node k is [k/2]. Ex. If k=9, parent is 4.
- The depth of the complete binary tree with n nodes is given by:

$$D_n = [\log_2^n + 1]$$

Ex: for n=1,000,000 nodes the depth is D_n=21.

Complete Binary Tree Continue…

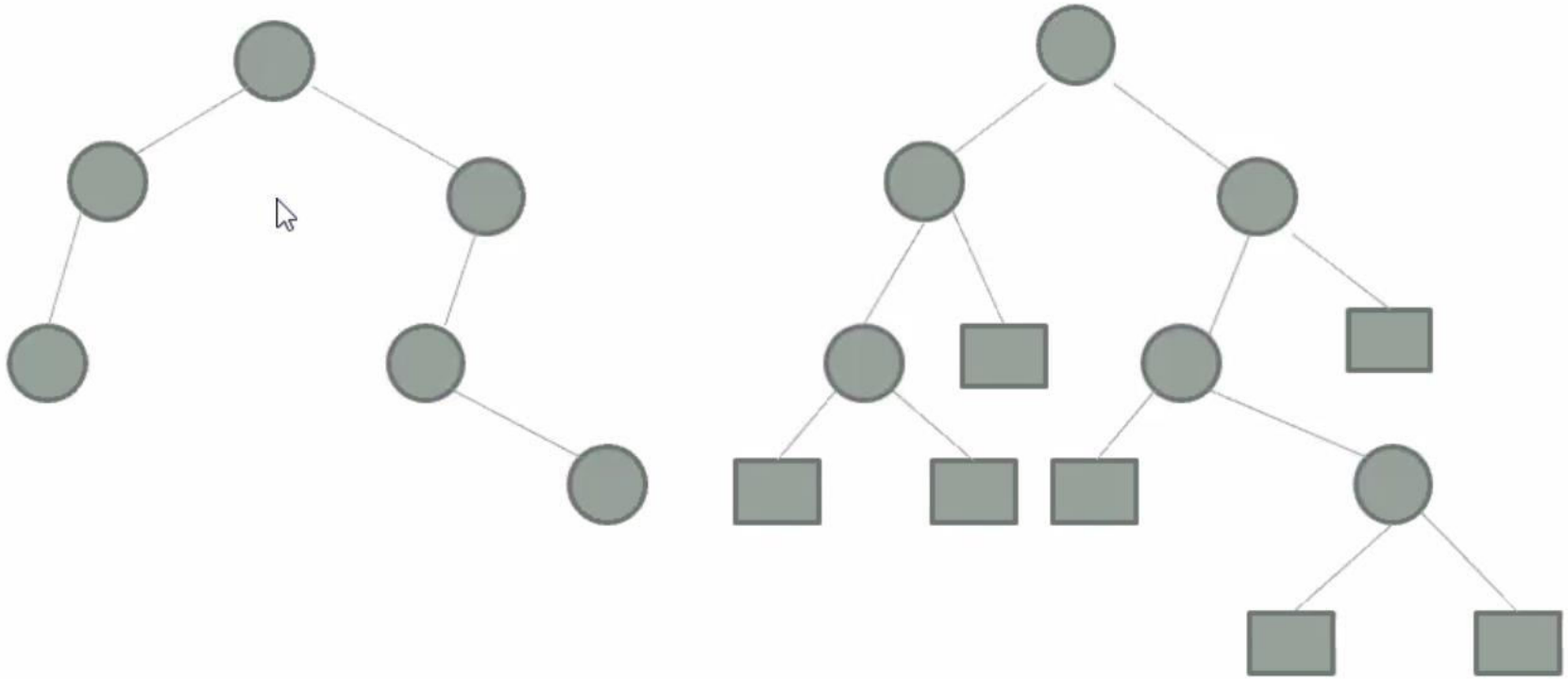
Example:



Extended Binary Tree

- A binary tree T is said to be a 2-tree or an extended binary if each node N has either 0 or 2 children.
- The nodes with two children are internal nodes.
- The nodes with zero children are external nodes.
- Internal nodes are represented by circle.
- External nodes are represented by square.

Extended Binary Tree Continue…



(a) Binary Tree T

(b) Extended 2-tree

Representing Binary Trees in Memory

- Two ways to represent T in memory.
- First one is Linked Representation (same as linked list).

- Second one is Sequential Representation (uses a single array).

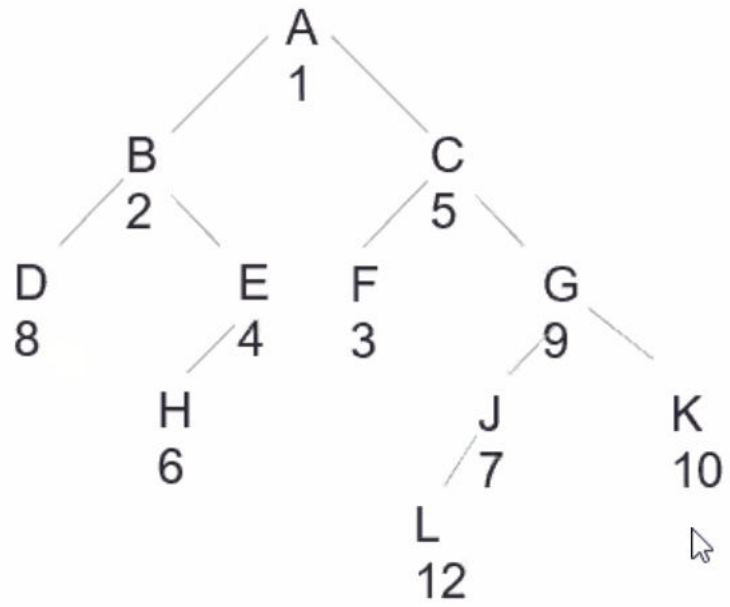
- However in any case, one should have direct access to the root R of T and, given any node N of T, one should have direct access to the children of N.

Linked Representation of Binary Trees

- T will be maintained in memory by means of a linked representation which uses three parallel arrays, INFO, LEFT, RIGHT and a pointer variable ROOT as follows.
  1. INFO[k] contains the data at the node N.
  2. LEFT[k] contains the location of the left child of node N.
  3. RIGHT[k] contains the location of the right child of node N.

- If any subtree is empty, then the corresponding pointer will contain the null value.
- If the tree itself is empty, then root will contain the null value.

Linked Rep. Continue…



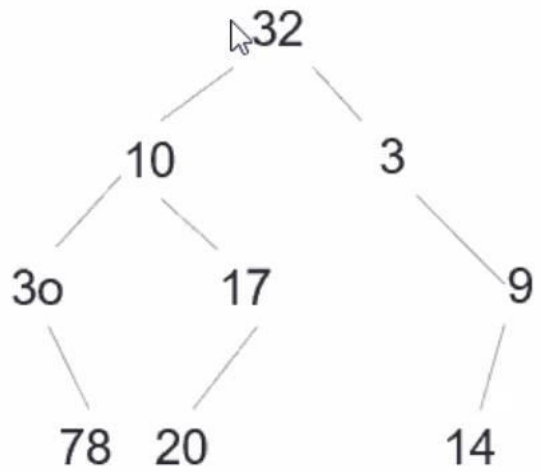
		INFO	LEFT	RIGHT
ROOT	1	A	2	5
	2	B	8	4
	3	F	0	0
	4	E	6	0
	5	C	3	9
	6	H	0	0
	7	J	12	0
	8	D	0	0
	9	G	7	10
	10	K	0	0
AVAIL	11		13	
	12	L	0	0
	13		0	

Sequential Representation of B. Tree

- Suppose T is a binary tree that is complete or nearly complete. The sequential representation uses only a single linear array TREE as follows.
  1. The root R of T is stored in TREE[1].
  2. If a node N occupies TREE[k], then its left child is stored in TREE[2*k] and its right child is stored in TREE[2*k+1].
- TREE[1]=NULL indicates that Tree T is empty.
- The sequential representation is shown on the next slide.

Sequential Rep. Continue…

- Note that we require 14 locations even though T has only 9 nodes.
- For depth d there requires approximately 2^{d+1} elements.
- So sequential representation is inefficient if the tree is not complete or nearly complete.



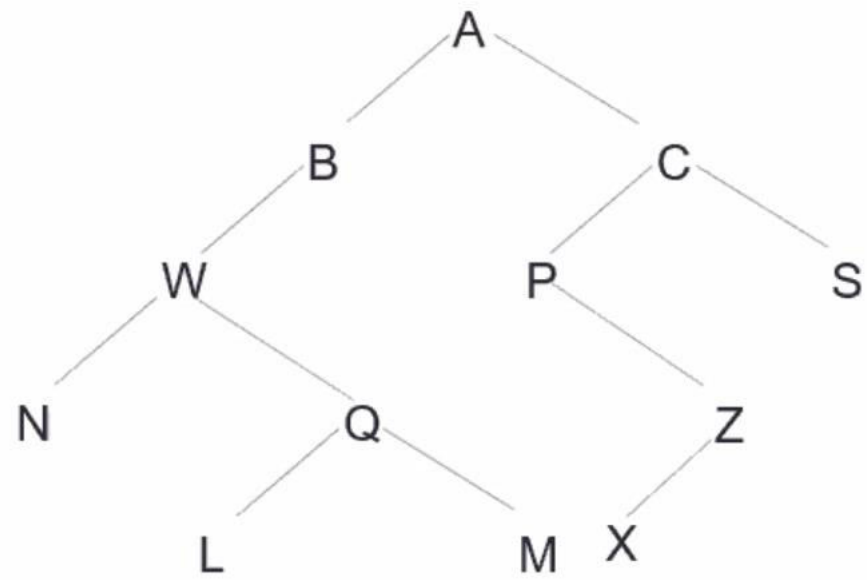
32	10	3	30	17		9		78	20				14				
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Traversing Binary Trees

- There are three standard ways of traversing a binary tree T with root R.
- PreOrder:
  1. Process the root.
  2. Traverse the left subtree of R in preorder.
  3. Traverse the right subtree of R in preorder.
- InOrder:
  1. Traverse the left subtree of R in inorder.
  2. Process the root.
  3. Traverse the right subtree of R in inorder.
- PostOrder:
  1. Traverse the left subtree of R in postorder.
  2. Traverse the right subtree of R in postorder.
  3. Process the root.

Traversing Binary Tree

- Consider the tree in the figure below.



Preorder: ABWNQLMCPZXS

Inorder: NWLQMBAPXZCS

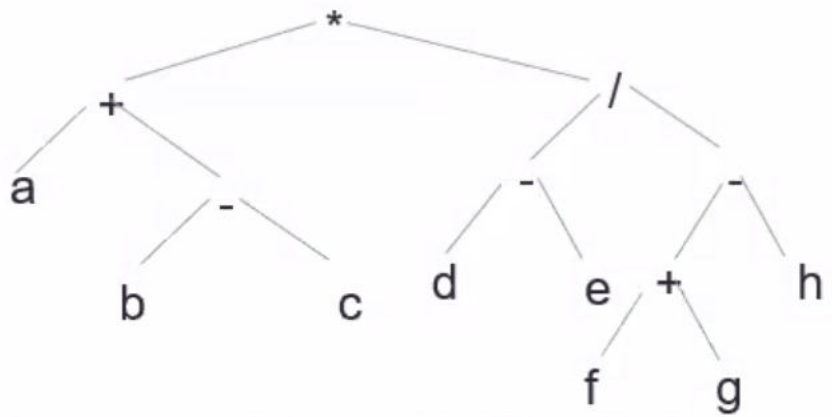
Postorder:

Traversing Binary Tree Example

- Consider the following algebraic expression:

  [a+(b-c)]*[(d-e)/(f+g-h)]

The resultant tree is shown below:



Preorder: *+a-bc/-de-+fgh (which is same as prefix)

Postorder: abc-+de-fg+h-/* (which is same as postfix)

Preorder Traversal Using Stack

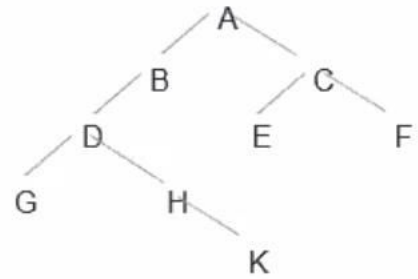
- Initially push null onto stack and then set ptr:=root. Then repeat the following steps while ptr!=null.

a. Proceed down the left-most path rooted at ptr, processing each node N on the path and pushing each right child R(N), if any, onto stack. The traversing ends after a node N with no left child L(N) is processed (Thus ptr is updated using the assignment ptr:=left[ptr], and traversing stops when left[ptr]=null).

b. [Backtracking] Pop and assign to ptr the top element on stack. If ptr!=null, then return to step (a); otherwise exit.

Preorder Traversal Using Stack Continue…

- Consider the following tree T. We simulate the preorder traversal algorithm with T, showing the contents of stack at each steps.



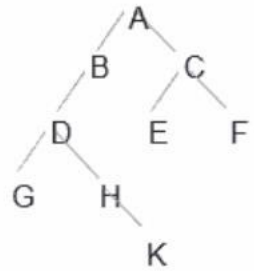
1. Initially push null onto stack: Stack: Φ
2. Proceed down the left most path rooted at ptr=A as follows:
   I. Process A and push its right child C onto stack. Stack: Φ, C
   II. Process B (There is no right child)
   III. Process D and push right child H onto stack. Stack: Φ, C, H
   IV. Process G (There is no right child)
   G has no left node, so no node is processed.

Preorder Traversal Using Stack Continue…

3. [Backtracking.] Pop the top element H from the stack and set ptr:=H.
   Here stack: Φ,C
   since ptr!=null, return to step (a) of the algorithm.

4. Proceed down the left most path rooted at ptr=H as follows:
   V. Process H and push its right child K onto stack. Stack: Φ, C, K
   H has no left node, so no node is processed.

5. [Backtracking.] Pop K from the stack and set ptr:=K.
   Here stack: Φ,C
   since ptr!=null, return to step (a) of the algorithm.

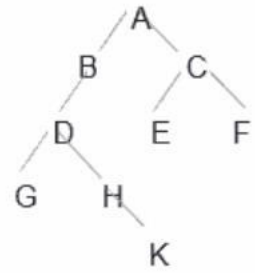


6. Proceed down the left most path rooted at ptr=K as follows:
   VI. Process K (There is no right child).
   K has no left node, so no node is processed.

7. [Backtracking.] Pop C from the stack and set ptr:=C.
   Here stack: Φ
   since ptr!=null, return to step (a) of the algorithm.

Preorder Traversal Using Stack Continue…

8. Proceed down the left most path rooted at ptr=C as follows:
   VII. Process C and push its right child F onto stack. Stack: Φ, F
   VIII. Process E (There is no right child).
   E has no left node, so no node is processed
9. [Backtracking.] Pop F from the stack and set ptr:=F.
   Here stack: Φ
   since ptr!=null, return to step (a) of the algorithm.



10. Proceed down the left most path rooted at ptr=F as follows:
    IX. Process F (There is no right child).
    F has no left node, so no node is processed.

11. [Backtracking.] Pop the top element null from the stack and set ptr:=null.
    since ptr=null, the algorithm is completed.

The nodes are processed in the order as follows: A, B, D, G, H, K, C, E, F.

Preorder Algorithm

- Preorder(INFO, LEFT, RIGHT, ROOT)

A binary tree T is in memory. The algorithm does a preorder traversal of T, applying an operation PROCESS to each of its nodes. An array stack is used to temporarily hold the addresses of nodes.

1. Set top:=1, stack[1]:=null and ptr:=root.
2. Repeat steps 3 to 5 while ptr!=null
3. Apply process to info[ptr]
4. If right[ptr]!=null then Set top:=top+1, and stack[top]:=right[ptr]
5. If left[ptr]!=null then Set ptr:=left[ptr]
   else ptr:=stack[top], and top:=top-1
6. Exit.

Inorder Traversal Using Stack

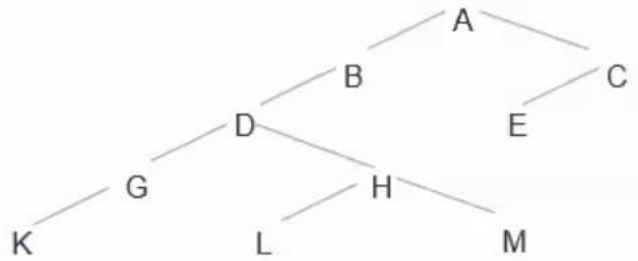
- Initially push null onto stack and then set ptr:=root. Then repeat the following steps until null is popped from the stack.

a. Proceed down the left-most path rooted at ptr, pushing each node N onto stack and stopping when a node N with no left child is pushed onto stack.

b. [Backtracking] Pop and process the nodes on stack. If null is popped, then exit. If a node N with a right child R(N) is processed, set ptr:=R(N) (by assigning ptr:=right[ptr] and return to step (a).

Inorder Traversal Using Stack Continue…

- Consider the following tree T. We simulate the inorder traversal algorithm with T, showing the contents of stack at each steps.



1. Initially push null onto stack: Stack: Φ
   Then Set ptr:=A the root of T.

2. Proceed down the left most path rooted at ptr=A, pushing the nodes A, B, D, G and K onto stack. Stack: Φ, A, B, D, G, K
   (No other node is pushed since K has no left child).

Inorder Traversal Using Stack Continue…

3. [Backtracking.] The node K, G and D are popped and processed.
   Stack: Φ, A, B

   (We stop the processing at D, since D has a right child) Then Set ptr:=H, the right child of D.

4. Proceed down the left most path rooted at ptr=H, pushing the nodes H and L onto stack. Stack: Φ, A, B, H, L.

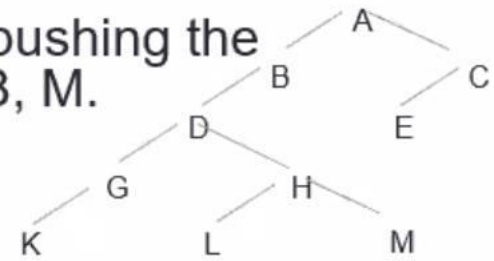
   (No other nodes are pushed since L has no left child.)

5. [Backtracking.] The nodes L and H are popped and processed.
   Stack: Φ, A, B.

   (We stop the processing at H, since H has a right child) Then Set ptr:=M, the right child of H.

6. Proceed down the left most path rooted at ptr=M, pushing the nodes M onto stack. Stack: Φ, A, B, M.

   (No other nodes are pushed since M has no left child.)



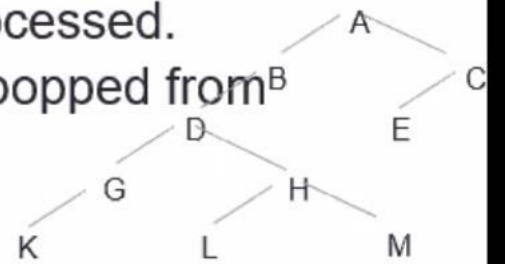
Inorder Traversal Using Stack Continue…

7. [Backtracking.] The nodes M, B and A are popped and processed. Stack: Φ.

   (We stop the processing at A, since A has a right child) Then Set ptr:=C, the right child of A.

8. Proceed down the left most path rooted at ptr=C, pushing the nodes C and E onto stack. Stack: Φ, C, E.

9. [Backtracking.] The nodes E is popped and processed.
   1. Since E has no right child node C is popped and processed.
   2. Since C has no right child, the next element, null is popped from stack.



Thus the nodes are processed as follows: K, G, D, L, H, M, B, A, E, C

Inorder Algorithm

Inorder(INFO, LEFT, RIGHT, ROOT)

A binary tree T is in memory. The algorithm does an inorder traversal of T, applying an operation PROCESS to each of its nodes. An array stack is used to temporarily hold the addresses of nodes.

1. Set top:=1, stack[1]:=null and ptr:=root.
2. Repeat while ptr!=null
   a) Set top:=top+1 and stack[top]:=ptr.
   b) Set ptr:=left[ptr].

3. Set ptr:=stack[top] and top:=top-1.
4. Repeat steps 5 to 7 while ptr!=null [Backtracking]
5. Apply process to info[ptr]
6. If right[ptr]!=null then Set (a) ptr:=right[ptr] and (b) goto step 2
7. Set ptr:=stack[top] and top:=top-1.
8. Exit.

Postorder Traversal Using Stack

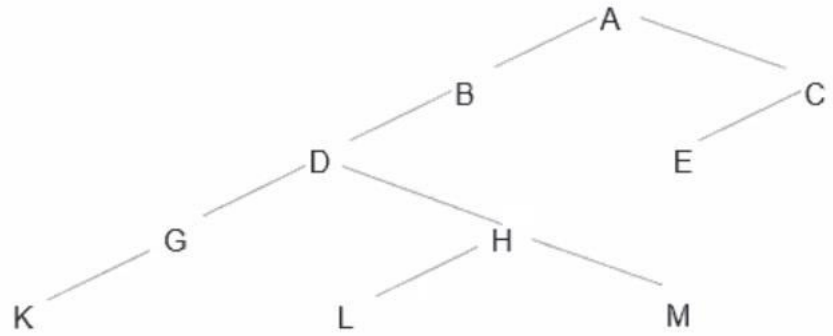
Initially push null onto stack and then set ptr:=root. Then repeat the following steps until null is popped from the stack.

a. Proceed down the left-most path rooted at ptr. At each node N of the path, push N onto stack and, if N has a right child R(N), push -R(N) onto stack.

b. [Backtracking] Pop and process positive nodes on stack. If null is popped, then exit. If a negative node is popped, that is, if ptr=-N for some node N, set ptr=N (by assigning ptr=-ptr) and return to step (a).

Postorder Traversal Using Stack Continue…

- Consider the following tree T. We simulate the postorder traversal algorithm with T, showing the contents of stack at each steps.



1. Initially push null onto stack and set ptr:=A the root of T.

   Stack: Φ

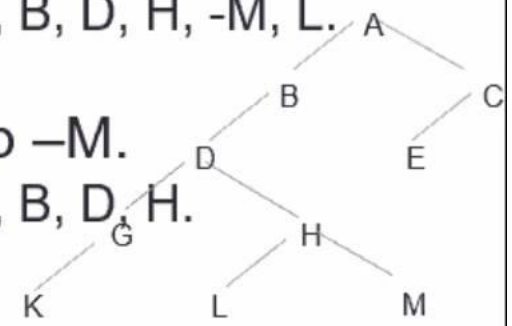
2. Proceed down the left most path rooted at ptr=A, pushing the nodes A, B, D, G and K onto stack. Furthermore, since A has a right child C, push –C onto stack after A but before B, and since D has a right child H, push -H onto stack after D but before G.

   Stack: Φ, A, -C, B, D, -H, G, K

Postorder Traversal Using Stack Continue…

3. [Backtracking.] Pop and process K, and pop and process G.
   Since –H is negative, only pop –H. Stack: Φ, A, -C, B, D.
   Now ptr=-H. Reset ptr:=H and return to step (a).

4. Proceed down the left most path rooted at ptr=H, and push the node H onto stack. Since H has a right child M, push –M onto stack after H and then push L onto stack.
   Stack: Φ, A, -C, B, D, H, -M, L.



5. [Backtracking.] Pop and process L but only pop –M.
   Stack: Φ, A, -C, B, D, H.
   Now ptr=-M. Reset ptr:=M and return to step (a).

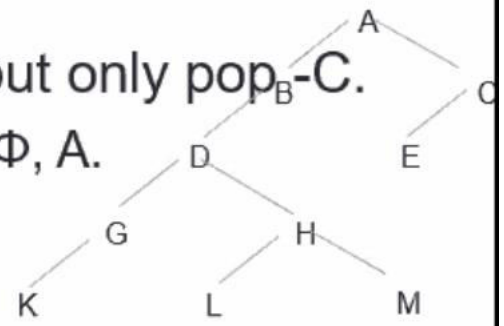
6. Proceed down the left most path rooted at ptr=M, and push the node M onto stack.
   Stack: Φ, A, -C, B, D, H, M.

Postorder Traversal Using Stack Continue…

7. [Backtracking.]Pop and Process M, H, D and B but only pop -C.

   Stack: Φ, A.

   Now ptr=-C. Reset ptr=C and return to step (a).



8. Proceed down the left most path rooted at ptr=C, and push the nodes C and then E.

   Stack: Φ, A, C, E.

9. [Backtracking.] Pop and process E, C and A. When null is popped, stack is empty and then algorithm is complete.

Thus the nodes are processed as follows: K, G, L, M, H, D, B, E, C, A

Postorder Algorithm

Postorder(INFO, LEFT, RIGHT, ROOT)

A binary tree T is in memory. The algorithm does an postorder traversal of T, applying an operation PROCESS to each of its nodes. An array stack is used to temporarily hold the addresses of nodes.

1. Set top:=1, stack[1]:=null and ptr:=root.
2. Repeat steps 3 to 5 while ptr!=null
3. Top:=top+1 and stack[top]:=ptr.
4. If right[ptr]!=null, then: Set top:=top+1 and stack[top]:=-right[top].
5. Set ptr:=left[ptr]
6. Set ptr:=stack[top] and top:=top-1.
7. Repeat while ptr>0
   a) Apply process to info[ptr].
   b) Set ptr:=stack[top] and top:=top-1.
8. If ptr<0 then:
   a) Set ptr:=-ptr. and
   b) goto step 2.
9. Exit.

Binary Search Tree

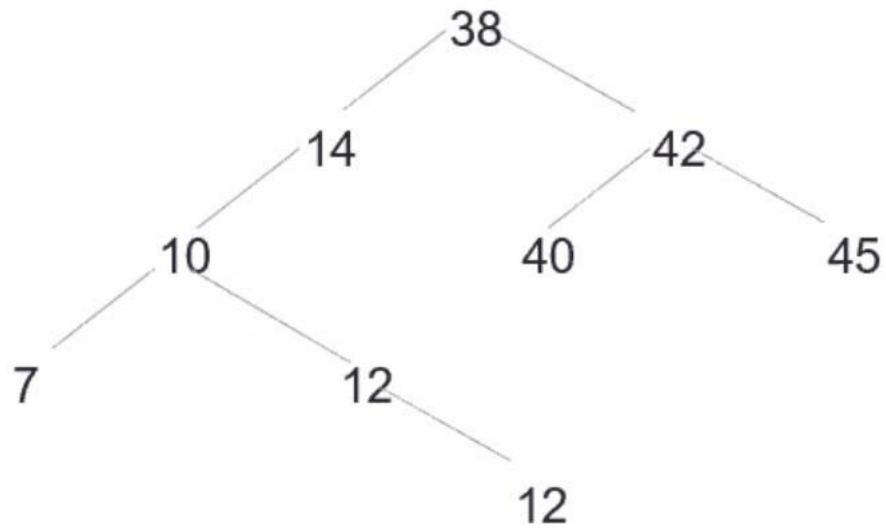
- Suppose T is a binary tree. Then T is called a Binary Search Tree (or Binary Sorted Tree) if each node N of T has the following property:
  1. The value at N is greater than every value in the left subtree of N
  2. And is less than or equal to every value in the right subtree of N.

- **These properties guarantees that the inorder traversal of T will yield a sorted listing of the elements of T.

- Why Binary Search Tree?
  1. One can search for an element with a running time $f(n)=O(\log_2{}^n)$, which is advantage of Sorted linear list but disadvantage of Linked list.
  2. One can easily insert and delete an elements, which is advantage of Linked list but disadvantage of Sorted linear list.

Binary Search Tree Continue…

- Consider the following Binary Search Tree in the figure below.



- Inorder Traversing: 7, 10, 12, 12, 14, 38, 40, 42, 45
- Next we show that, it is easy to insert and delete elements and searching is very fast.

Searching and Inserting in Binary Search Tree

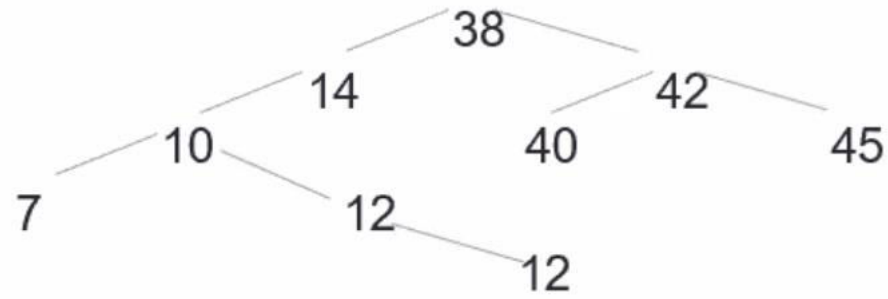
- Suppose an Item of information is given. The following algorithm finds the location of Item in Binary Search Tree T, or inserts Item as a new node in its appropriate place in the tree.
  a) Compare Item with the root node N of the tree.
     I. If Item<N, proceed to the left child of N.
     II. If Item>=N proceed to the right child of N.

  b) Repeat step (a) until one of the following occurs.
     I. We meet a node N such that Item=N, in this case the search is successful.
     II. We meet an empty subtree, which indicates that the search is unsuccessful, and we insert Item in place of the empty subtree.

- In other words, proceed from the root R down through the tree T until finding Item in T or inserting Item as a terminal node.

Binary Search Tree Simulation

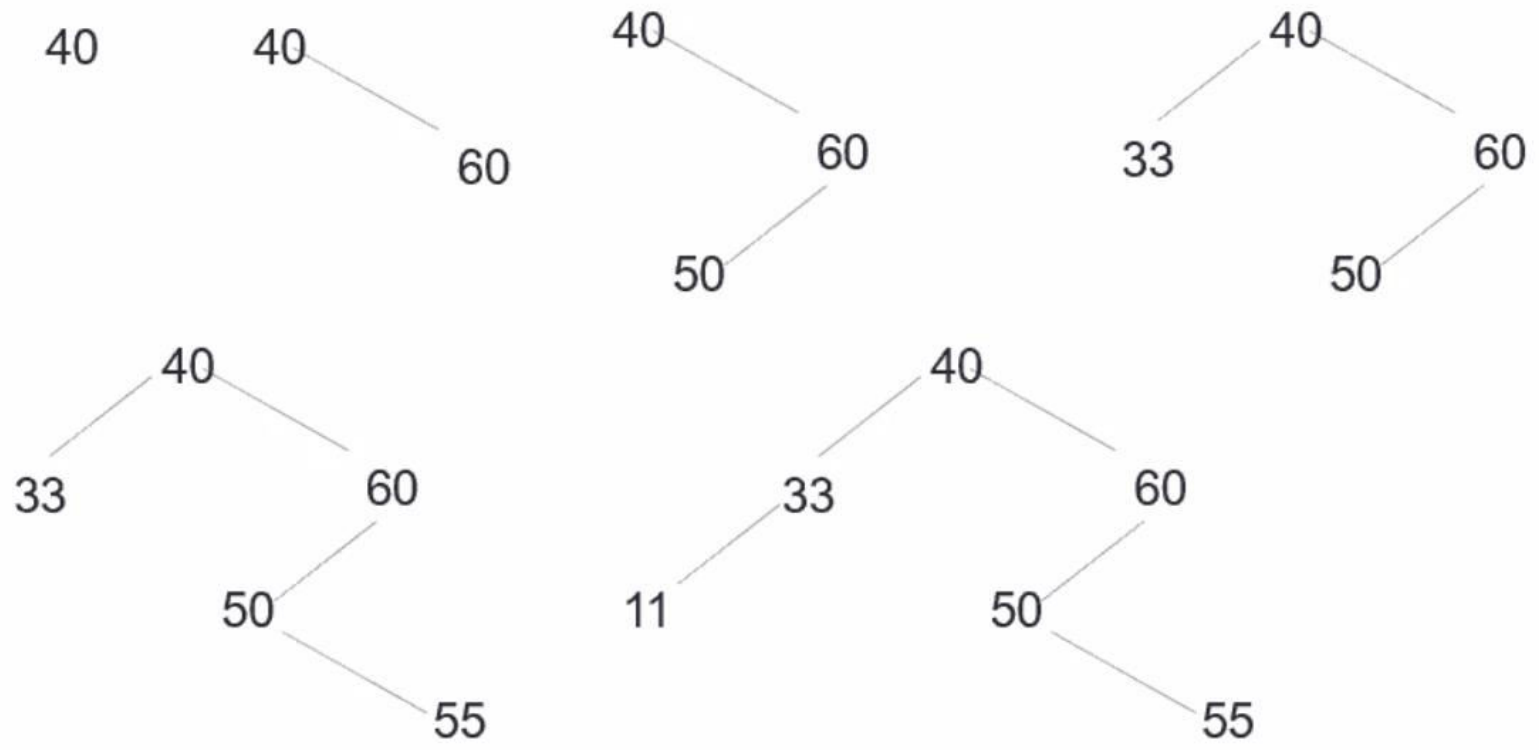
- Consider the tree below. Suppose Item=11. We simulate the algorithm. The steps are as follows.



1. Compare Item=11 with the root 38. Since 11<38, proceed to the left child of 38 which is 14.
2. Compare Item=11 with 14. Since 11<14, proceed to the left child of 14, which is 10.
3. Compare Item=11 with 10. Since 11>10, proceed to the right child of 10 which is 12.
4. Compare Item=11 with 12. since 11<12 and 12 does not have a left child, insert 11 as the left child of 12.

Adding to Binary Search Tree

- Suppose the following six numbers are inserted in order into an empty binary search tree. 40, 60, 50, 33, 55, 11.



Binary Search Tree Algorithm

- The following algorithm along with the procedure on the next slide finds the location loc of Item in T or adds item as a new node in T at location loc.

- INSBST(info, left, right, root, item, loc)

  A binary Search Tree T is in memory and an Item of information is given. This algorithm finds the location loc of Item in T or adds Item as a new node in T at location loc

  1. Call FIND(info, left, right, root, item, loc, par)
  2. If loc!=null then exit.
  3. [Copy item into new node in avail list]
     a) If avail=null then write: Overflow and exit.
     b) Set new:=avail, avail:=left[avail] and info[new]:=item.
     c) Set loc:=new, left[new]:=null and right[new]:=null
  4. If par=null then Set: root:=new
  5. Else if item<info[par] then Set: left[par]:=new.
  6. Else Set right[par]:=new.
  7. Exit.

Binary Search Tree Algorithm Continue…

- FIND(info, left, right, root, item, loc, par)

  A binary search tree T is in memory and an item of information is given. This procedure finds the location loc of item in T and also the location par of the parent of item. There are three special cases:

  I. Loc=null and par=null will indicate that the tree is empty.
  II. Loc!=null and par=null will indicate that item is the root of T.
  III. Loc=null and par!=null will indicate that item is not in T and can be added to T as a child of the node N with location par.

1. If root=null then Set: loc:=null and par:=null and return.
2. If item=info[root] then Set: loc:=root and par:=null and return.
3. If item<info[root] then Set: ptr:=left[root] and save:=root.
4. Else Set: ptr:=right[root] and save:=root.
5. Repeat steps 6 and 7 while ptr!=null
6. If item=info[ptr] then Set: loc:=ptr and par:=save and Return.
7. If item<info[ptr] then Set: save:=ptr and ptr:=left[ptr]
   Else Set: save:=ptr and ptr:=right[ptr].
8. [Search Unsuccessful] Set: loc:=null and par:=save.
9. Exit.

Heap

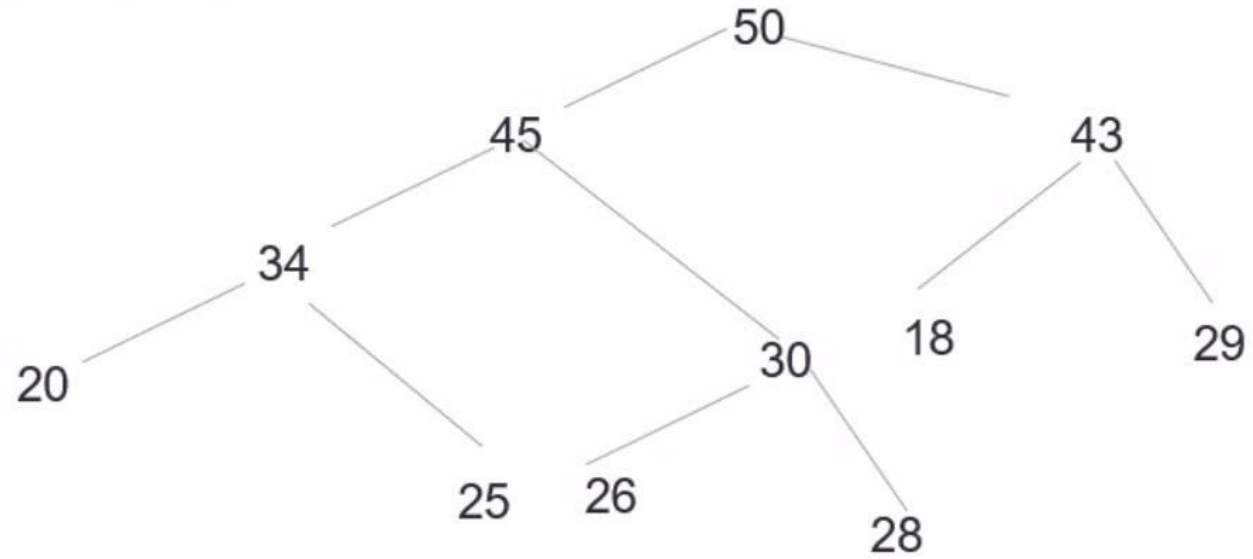
- Suppose H is a complete binary tree with n elements. Then H is called a heap or a maxheap if each node K of H has the following property:
  - The value at K is greater than or equal to the value at each of the children of K. Accordingly the value at K is greater than or equal to the value at any of the descendants of K.

- A minheap is defined analogously: The value at K is less than or equal to the value at any of the children of N.
- Unless otherwise stated, we assume that H is maintained in memory by a linear array TREE using the sequential representation of H, not a linked representation.

MaxHeap Example

- The elements of the tree are stored in the sequential memory location as follows:

Index:	1	2	3	4	5	6	7	8	9	10	11
Data:	50	45	43	34	30	18	29	20	25	26	28



Inserting into a MaxHeap

- Suppose H is a maxheap, and suppose an item of information is given. We insert item into the heap H as follows:
  - First adjoin item at the end of H so that H is still a complete tree, but not necessarily a heap.
  - Then let item rise to its appropriate place in H so that H is finally a maxheap.

- Let us consider the tree on the previous slide and suppose we wish to insert 46.
  1. First we adjoin 46 at position 12. i.e. tree[12]=46.
  2. Compare 46 with its parents 18. since 46 is greater than 18, interchange 46 and 18.
  3. Compare 46 with its new parent 43. since 46 is greater than 43, interchange 46 and 43.
  4. Compare 46 with its new parent 50. since 46 does not exceed 50, item 46 is no in its appropriate position.

Inserting into a MaxHeap Algorithm

- INSHEAP(TREE, N, ITEM)

A heap H with N elements is stored in the array TREE, and an ITEM of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

1. Set n:=n+1 and ptr:=n.
2. Repeat steps 3 to 6 while ptr>1.
3. Set par:=ptr/2.
4. If item<=tree[par] then: Set tree[ptr]:=item and Return.
5. Set tree[ptr]:=tree[par].
6. Set ptr:=par
7. Tree[1]:=item.
8. Return.