SYLLABUS OF SEMESTER-I, MCA (Artificial Intelligence and Machine Learning)

Course Code: 24CS60TH1177 Course: Data Structures

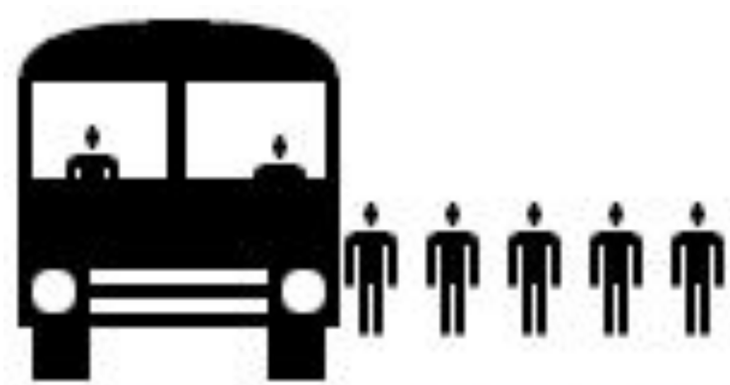
Unit - IV :

Queues: Definition and examples of queues, primitive operations, Types of Queues.

Trees: Definition and Basic Terminology of trees, Binary Tree, Binary Search Tree, Tree Traversal.

Queue

- A ***queue*** is a linear list of elements.
- **Deletions** can take place only at one end, called the ***front***.
- **Insertions** can take place only at one end, called the ***rear***.
- First inserted element deleted first, hence called **F**irst-**I**n-**F**irst-**O**ut (**FIFO**)



Queue of people at Bus stop

Array representation of Queues

Procedure: **QINSERT(QUEUE, N, FRONT, REAR, ITEM)**

1. If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:

   Write: OVERFLOW and Return.

2. If FRONT = NULL, then:

   Set FRONT := 1 and REAR := 1.

   Else If REAR = N, then:

   Set REAR := 1.

   Else:

   Set REAR := REAR + 1.

3. Set QUEUE[REAR] := ITEM

4. Return.

Array representation of Queues

QUEUE

FRONT : 0
REAR : 0

1	2	3	4	5	6	7	...	N

QUEUE

FRONT : 1
REAR : 3

AAA	BBB	CCC						
1	2	3	4	5	6	7	...	N

QUEUE

FRONT : 2
REAR : 5

	BBB	CCC	DDD	EEE				
1	2	3	4	5	6	7	...	N

QUEUE

FRONT : 4
REAR : 7

			DDD	EEE	FFF	GGG		
1	2	3	4	5	6	7	...	N

Array representation of Queues

Procedure: **QDELETE(QUEUE, N, FRONT, REAR, ITEM)**

1. If FRONT = NULL, then:

   Write: UNDERFLOW and Return.

2. Set ITEM := QUEUE[FRONT].
3. If FRONT = REAR, then:

   Set FRONT := NULL and REAR := NULL.

   Else If FRONT = N, then:

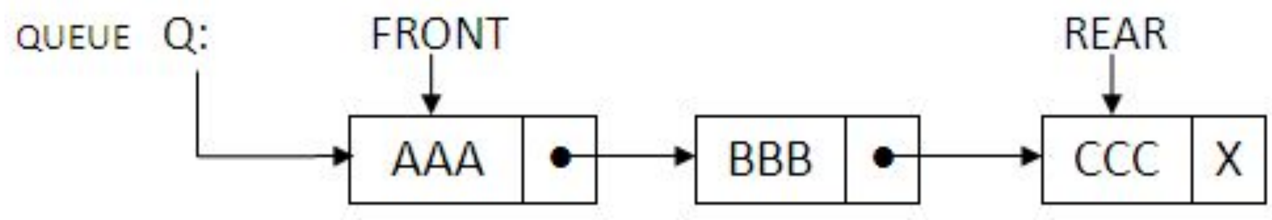
   Set FRONT := 1.

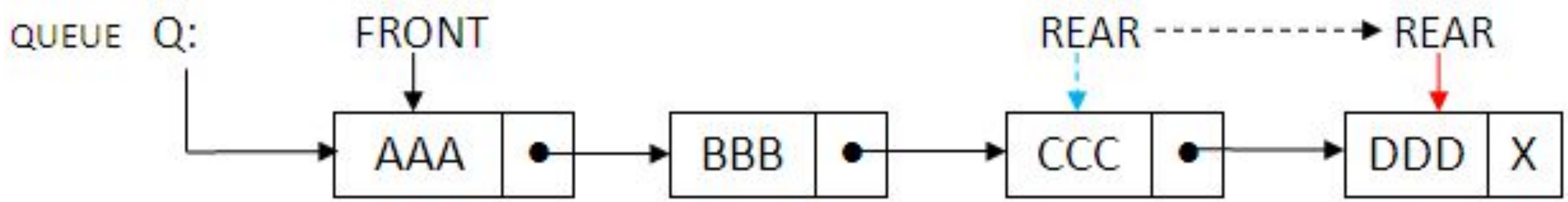
   Else:

   Set FRONT := FRONT + 1.

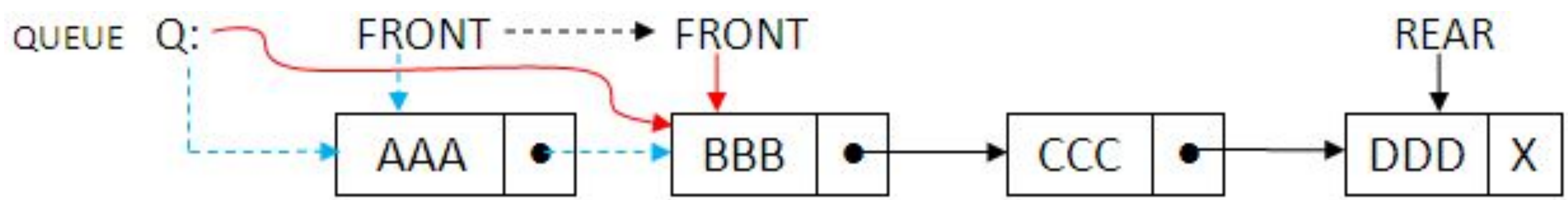
4. Return.

Linked representation of Queues





Insertion of 'DDD' into linked queue



Deletion of 'AAA' from linked queue

Insertion into Linked Queue

Procedure: **LINKQ_INSERT(INFO, LINK, FRONT, REAR, AVAIL, ITEM)**

1. If AVAIL = NULL, then Write: OVERFLOW and Exit.
2. Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM and LINK[NEW] := NULL.
4. If FRONT = NULL, then:

   Set FRONT := NEW and REAR := NEW

   Else:

   Set LINK[REAR] := NEW and REAR := NEW.
5. Exit.

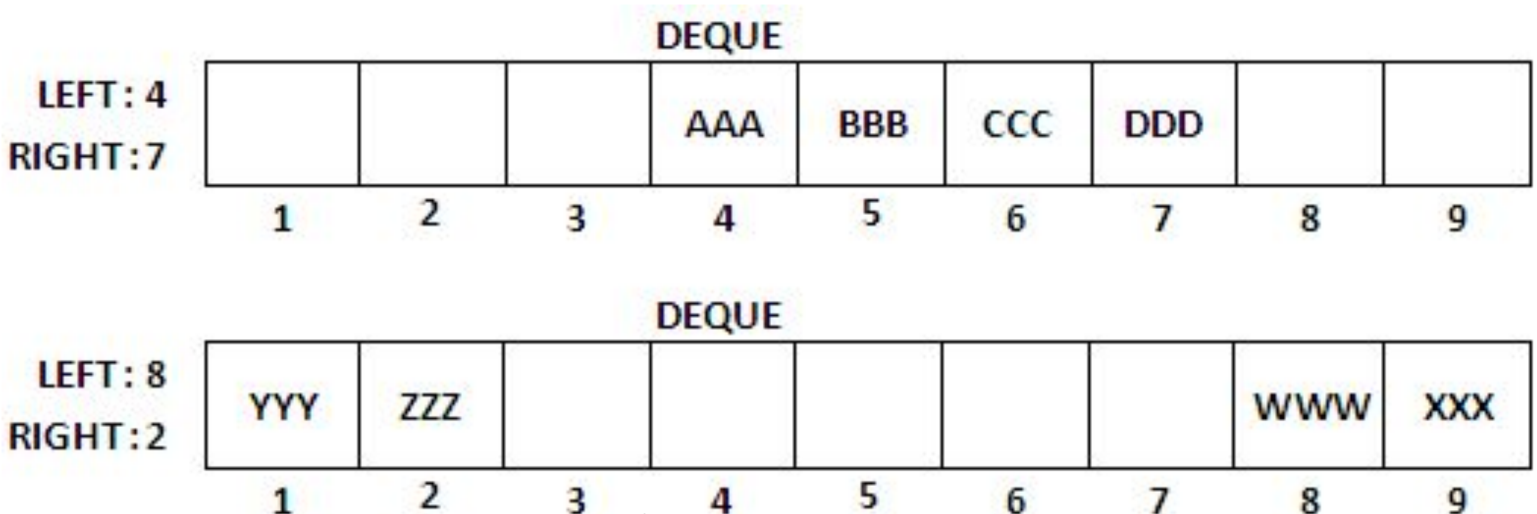
Deletion from Linked Queue

Procedure: **LINKQ_DELETE(INFO, LINK, FRONT, REAR, AVAIL, ITEM)**

1. If FRONT = NULL, then Write: UNDERFLOW and Exit.
2. Set ITEM := INFO[FRONT].
3. TEMP := FRONT. and FRONT := LINK[FRONT].
4. Set LINK[TEMP] := AVAIL and AVAIL := TEMP.
5. Exit.

Deques

- A ***deque*** is a linear list of elements.
- Elements can be added or removed at either end but not in middle.
- Deque is maintained by a circular array **DEQUE** with pointers **LEFT** and **RIGHT**.
- There are two variations of deque – **an input-restricted deque** and **an output-restricted deque**.
- An **input-restricted deque** allows insertions at only one end of the list but allows deletions at both ends of the list.
- An **output-restricted deque** allows deletions at only one end of the list but allows insertions at both ends of the list.



Priority Queues

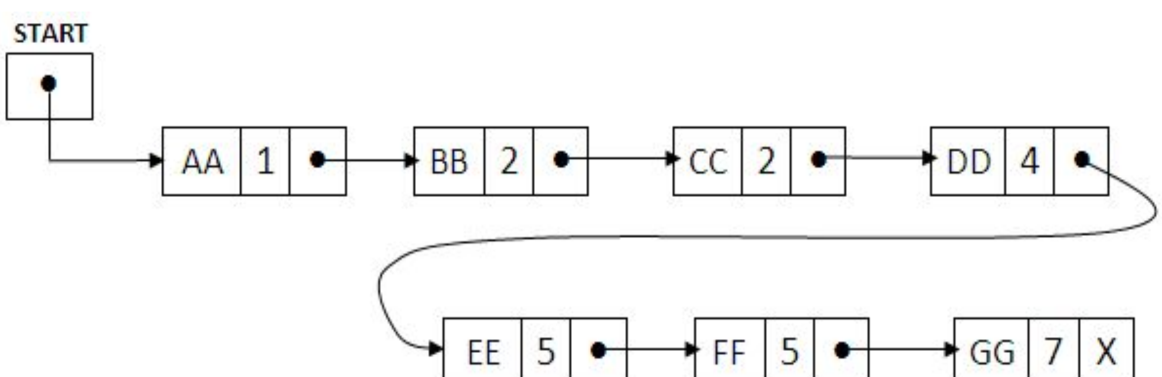
- A ***Priority queue*** is a linear list of elements such that each element has been assigned a priority.
- Following rules are apply for the deletion and processing of elements:
  1) An element of higher priority is processed before any elements of lower priority.
  2) Two elements with same priority are processed according to the order in which they were added to the queue.
- A prototype of a priority queue is timesharing system.

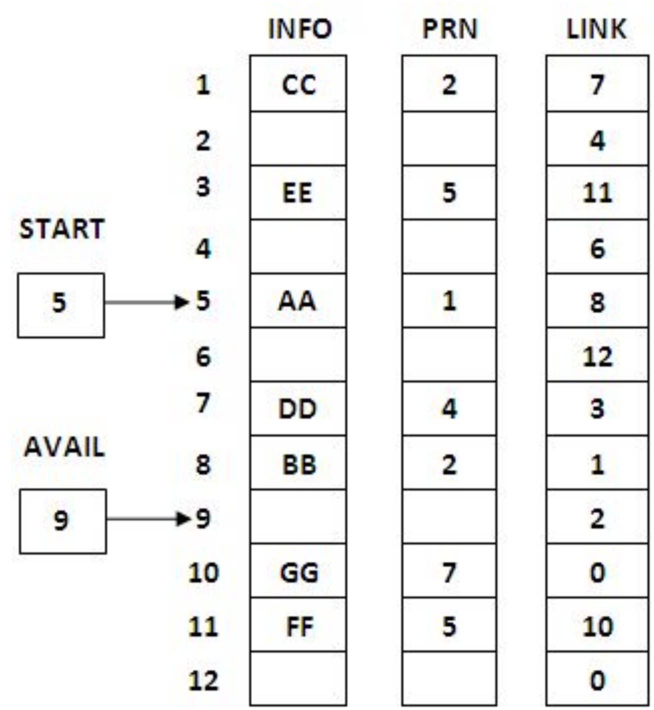
Priority Queues . . .

One-way List Representation of a Priority Queue

One way linked list use to maintain a priority queue in memory, as follows:

1) Node contains three items of information: INFO field, PRN priority number and LINK a link field.
2) A node X precedes a node Y in the list,
   a. when X has higher priority than Y or
   b. when both have the same priority but X was added to the list before Y.





Priority Queues . . .

Algorithm: *Deletes and process first element in a priority queue maintained by linked list.*

1. Set ITEM := INFO[START]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Algorithm: *Adds an ITEM with priority number N to a priority queue maintained by linked list.*

1. Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.
2. If no such node is found, insert ITEM as the last element of the list.

Priority Queues . . .

Array Representation of a Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number).

Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR.

	FRONT	REAR
1	2	2
2	1	2
3	0	0
4	4	4
5	5	6
6	0	0
7	1	1

	1	2	3	4	5	6
1		AA				
2	BB	CC				
3						
4				DD		
5					EE	FF
6						
7	GG					

Priority Queues . . .

Algorithm: *Deletes and process first element in a priority queue maintained by a two-dimensional array QUEUE.*

1. Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm: *Adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.*

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

Definition and Basic Terminology of trees

- ***Tree*** is a nonlinear data structure.
- Tree data structure is mainly used to represent data containing a hierarchical relationship between elements.

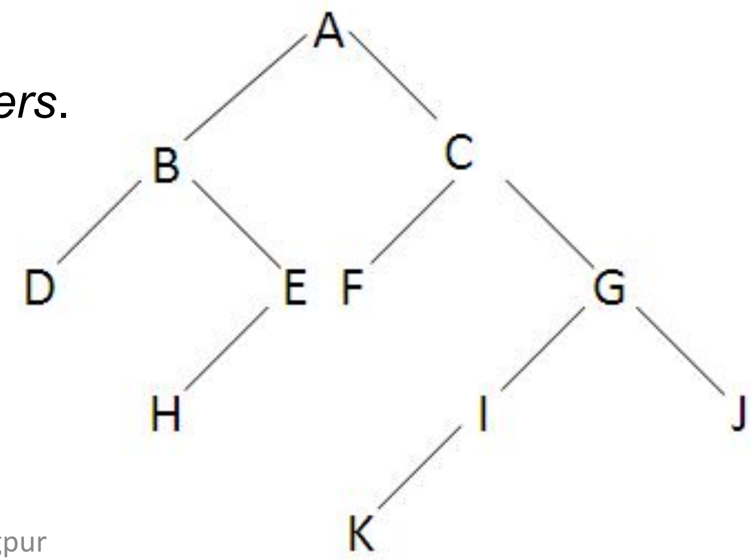
Terminology

- Suppose *N* is a node in tree *T* with left successor *S1* and right successor *S2*. Then *N* is called the *parent* (or *father*) of *S1* and *S2*.
- *S1* is called *left child* (or *son*) of *N* and *S2* is called *right child* (or *son*) of *N.*

  e.g. **A** is a *parent* of **B** and **C**, **B** is *parent* of **D** and **E**.
- *S1* and *S2* are said to be *siblings* (or *brothers*).

  e.g. **B** and **C**, **D** and **E** are *siblings or brothers*.
- Every node *N* in tree *T*, except the root, has a unique parent, called the *predecessor* of *N*.

  e.g. **B** is predecessor of **D**, **E** is predecessor of **H**.



Terminology . . .

- A node *L* is called a *descendent* of a node *N* if there is a succession of children from *N* to *L*. Also *N* is called an *ancestor* of *L*.
- *L* is called a *left* or *right descendent* of *N* according to whether *L* belongs to the left or right subtree of *N*.

  e.g. **E** is *right descendent* of **B**, **F** is *left descendent* of **C**. Also **B** is *ancestor* of **E**, **C** is *ancestor* of **F**.

- A terminal node is called *leaf* and a path ending in a leaf is called a *branch*.

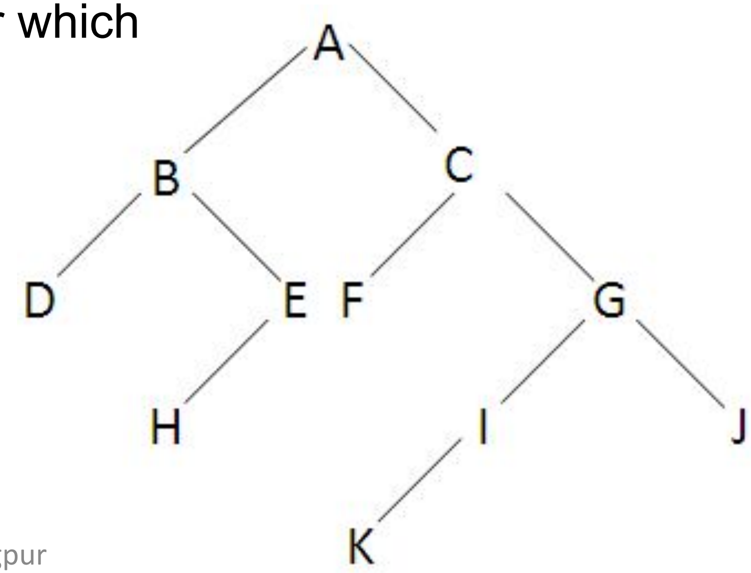
  e.g. **D**, **H**, **F**, **K** and **J** are *leaf* nodes.

- The root *R* of the tree *T* is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent.

  e.g. level of **H** is 3 and level of **K** is 4.

- The nodes with the same level number of nodes are belong to the same generation.
- The *depth* (or *height*) of a tree *T* is the maximum number of nodes in a branch of *T*.

  e.g. The *depth* or *height* of tree is **5**.



Binary Trees

❖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 root of *T*, and the remaining nodes of *T* form an ordered pair of disjoint binary trees *T1* and *T2*.

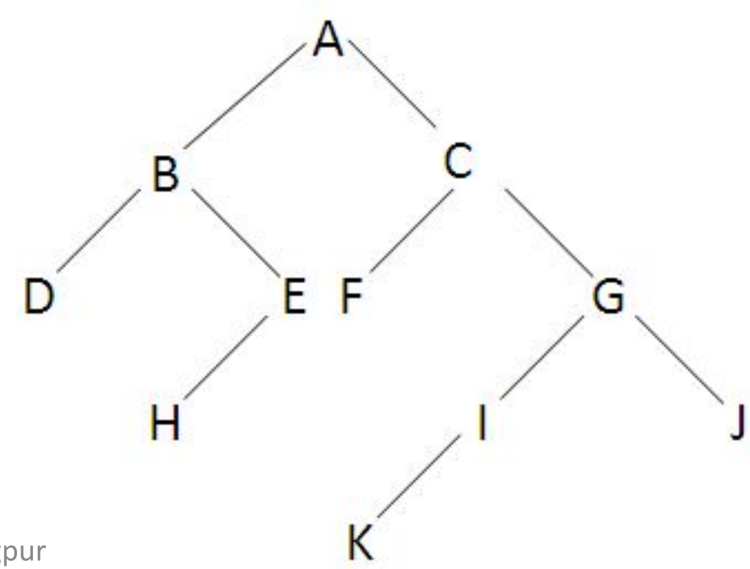
❖If *T* does contain a root *R*, then the two trees *T1* and *T2* are called, respectively, the *left* and *right subtrees* of *R*.

e.g. i) **B** is left successor and **C** is a right successor of the node **A**.

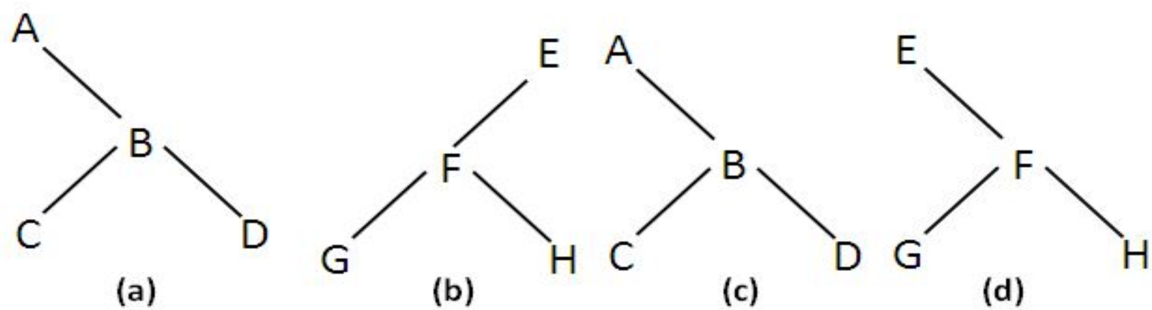
ii) Nodes **B, D, E**, & **H** are left subtree nodes of **A** and the nodes **C, F, G, I, J**& **K** are right subtree nodes of **A**.

❖ Any node *N* in binary *T* has either 0, 1 or 2 successors.

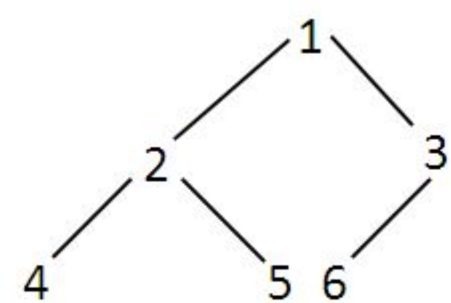
❖ If *N* is a terminal node, then both its left and right subtrees are empty.



- Binary trees *T* and *T'* are said to be *similar* if they have the same structure.
- The trees are said to be *copies* if they are similar and if they have the same contents at corresponding nodes.
    - The trees (a), (c) and (d) are similar
    - The trees (a) and (c) are copies due to same data at corresponding nodes.



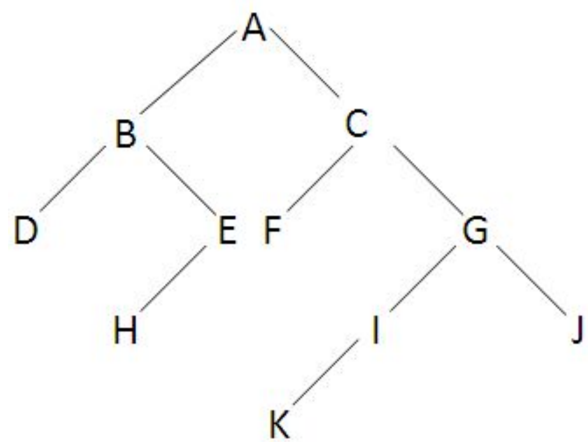
- The tree T is said to be ***complete*** if all its levels, except possibly the last, have maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.
- Specifically, the left and right children of the node **K** are, respectively, **2*K** and **2*K+1** and the parent of K is the node $\lfloor \mathbf{K/2} \rfloor$.

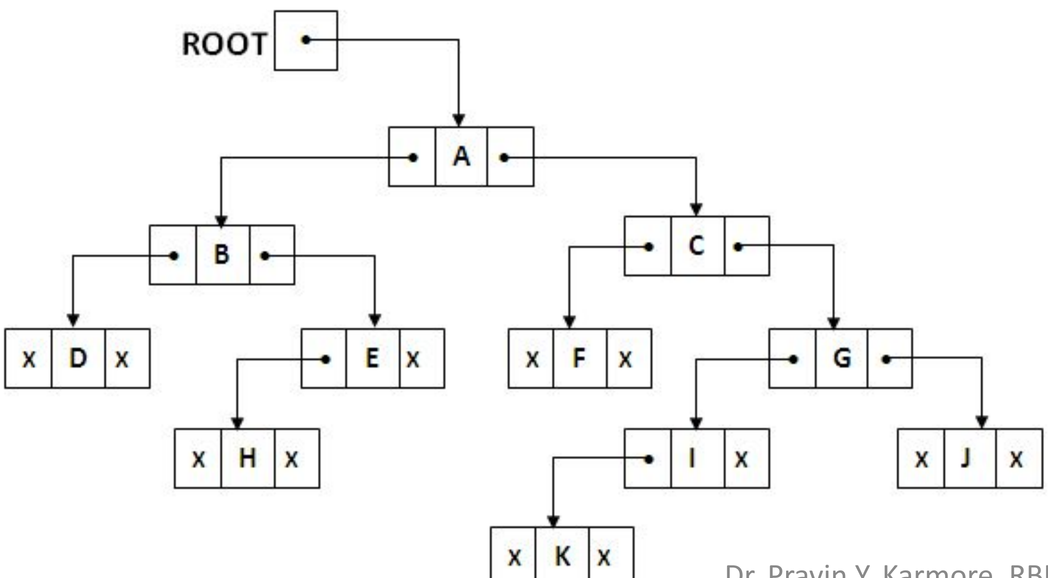


- The depth d_n of the complete tree with n nodes is
  $D_n = \lfloor \log_2 n + 1 \rfloor$

Linked Representation of Binary Trees in Memory

❖A ***Binary Tree*** *T* will be maintained in memory by means of a linked representation which uses three parallel arrays, INFO, LEFT and RIGHT and a pointer variable ROOT.





ROOT: 6

AVAIL: 11

	INFO	LEFT	RIGHT
1		7	
2	B	4	12
3	H	0	0
4	D	0	0
5	J	0	0
6	A	2	8
7		13	
8	C	15	14
9	K	0	0
10	I	9	0
11		1	
12	E	3	0
13		16	
14	G	10	5
15	F	0	0
16		0	

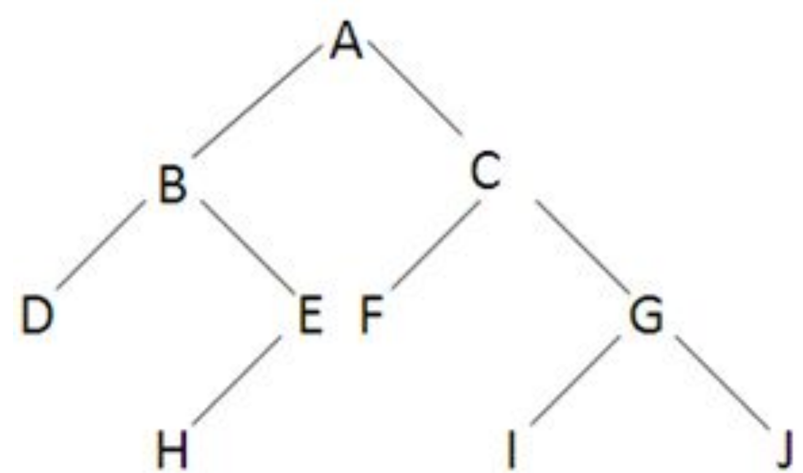
Sequential Representation of Binary Trees in Memory

- A ***Binary Tree*** *T* will be maintained in memory by means of a sequential representation which uses only a single linear array TREE.

  (a) The root R of T is stored in TREE[1].

  (b) 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].

- The sequential representation of a tree with depth *d* will require an array with approximately 2^{d+1} elements.



TREE	A	B	C	D	E	F	G			H				I	J		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Traversing Binary Trees

❖Three standard ways of traversing a binary tree T with root R.

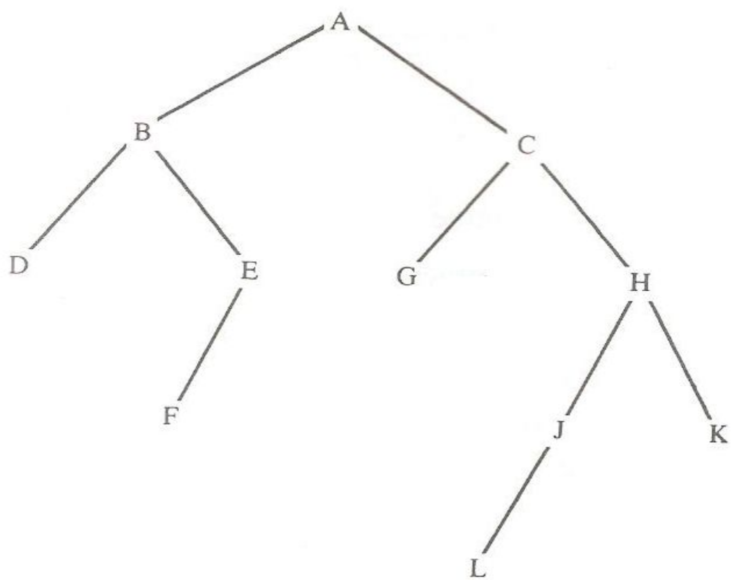
❖These algorithms called, Preorder, Inorder and Postorder, are as follows:

Preorder : (1) Process the root R.
(2) Traverse the left subtree of R in preorder.
(3) Traverse the right subtree of R in preorder.

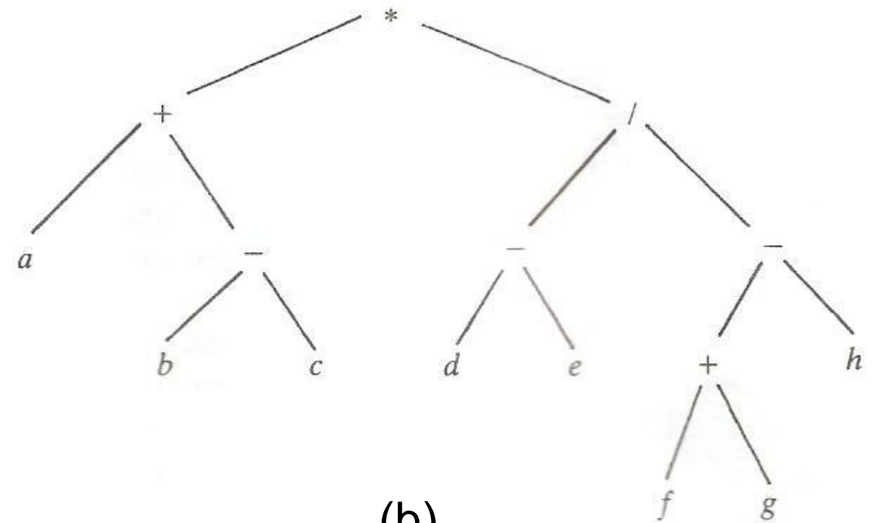
Inorder : (1) Traverse the left subtree of R in inorder.
(2) Process the root R.
(3) Traverse the right subtree of R in inorder.

Postorder : (1) Traverse the left subtree of R in postorder.
(2) Traverse the right subtree of R in postorder.
(3) Process the root R.

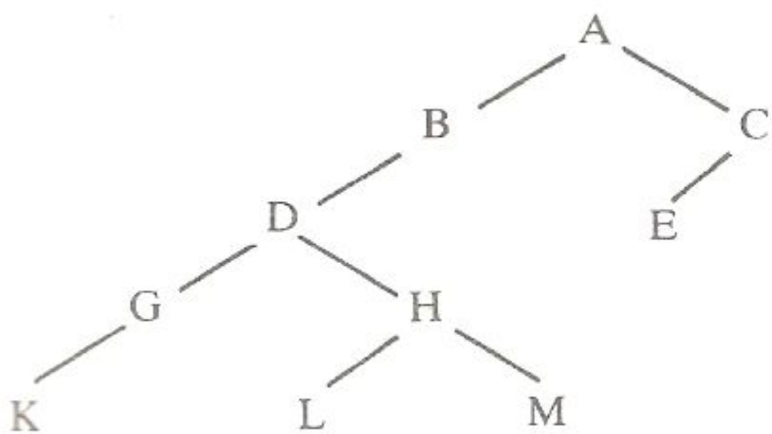
❖ Find Preorder, Inorder and Postorder traversal for following trees:



(a)



(b)



(c)

- (a)

  Preorder : **A, B, D, E, C, G, H, J, L, K**

  Inorder : **D, B, E, A, G, C, L, J, H, K**

  Postorder : **D, E, B, G, L, J, K, H, C, A**

- (b)

  Preorder : ***, +, a, -, b, c, /, -, d, e, -, +, f, g, h**

  Inorder : **a, +, b, -, c, *, d, -, e, /, f, +, g, -, h**

  Postorder : **a, b, c, -, +, d, e, -, f, g, +, h, -, /, ***

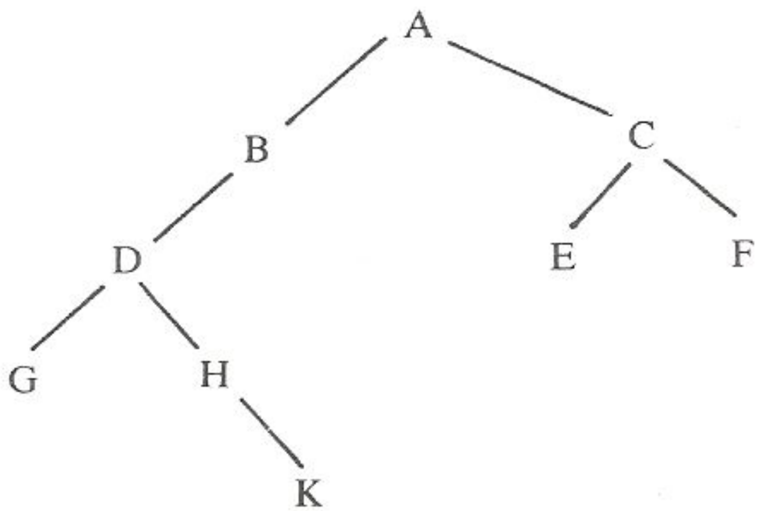
- (c)

  Preorder : **A, B, D, G, K, H, L, M, C, E**

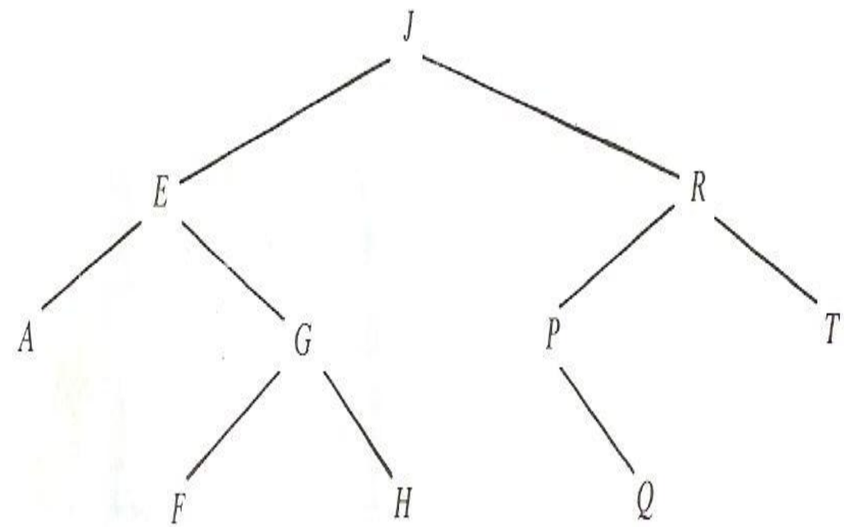
  Inorder : **K, G, D, L, H, M, B, A, E, C**

  Postorder : **K, G, L, M, H, D, B, E, C, A**

- Find Preorder, Inorder and Postorder traversal for following trees:



(a)



(b)

❖ **Construct the tree from the following traversals:**

a) Preorder : **A, B, D, E, C, G, H, J, L, K**

Inorder : **D, B, E, A, G, C, L, J, H, K**

b) Inorder : **a, +, b, -, c, *, d, -, e, /, f, +, g, -, h**

Postorder : **a, b, c, -, +, d, e, -, f, g, +, h, -, /, ***

c) Preorder: G, B, Q, A, C, K, F, P, D, E, R, H

Inorder: Q, B, K, C, F, A, G, P, E, D, H, R

d) Preorder : **A, B, C, E, D, F, G**

Inorder : **A, E, C, B, F, D, G**

e) Inorder : **a, -, b, *, c, /, d, +, e, *, f**

Postorder : **a, b, c, *, -, d, e, +, f, *, /**

Traversal Algorithms Using Stacks

Preorder Traversal

The preorder traversal algorithm uses a variable PTR (pointer) which will contain the location of the node N currently being scanned. This is pictured in Fig. 7-15, where L(N) denotes the left child of node N and R(N) denotes the right child. The algorithm also uses an array STACK, which will hold the addresses of nodes for future processing.

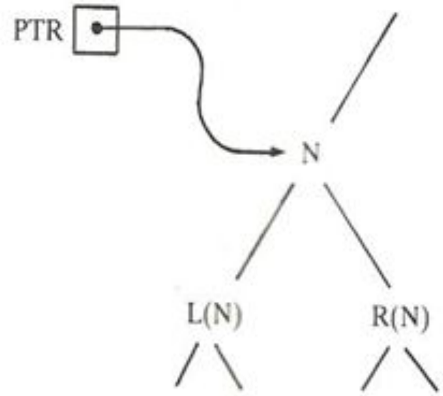


Fig. 7-15

Algorithm: Initially push NULL onto STACK and then set PTR := ROOT. Then repeat the following steps until PTR = NULL or, equivalently, 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 the 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.

(We note that the initial element NULL on STACK is used as a sentinel.)

Traversal Algorithms Using Stacks …

Algorithm: PREORD(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. [Initially push NULL onto STACK, and initialize PTR.]
   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. [Right child?]
   If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
   Set TOP := TOP + 1, and STACK[TOP] := RIGHT[PTR].
   [End of If structure.]
5. [Left child?]
   If LEFT[PTR] ≠ NULL, then:
   Set PTR := LEFT[PTR].
   Else: [Pop from STACK.]
   Set PTR := STACK[TOP] and TOP := TOP − 1.
   [End of If structure.]
   [End of Step 2 loop.]
6. Exit.

Traversal Algorithms Using Stacks …

Algorithm: INORD(INFO, LEFT, RIGHT, ROOT)

A binary tree is in memory. This 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. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat while PTR $\neq$ NULL: [Pushes left-most path onto STACK.]
   - (*a*) Set TOP := TOP + 1 and STACK[TOP] := PTR. [Saves node.]
   - (*b*) Set PTR := LEFT[PTR]. [Updates PTR.]

   [End of loop.]
3. Set PTR := STACK[TOP] and TOP := TOP − 1. [Pops node from STACK.]
4. Repeat Steps 5 to 7 while PTR $\neq$ NULL: [Backtracking.]
5. Apply PROCESS to INFO[PTR].
6. [Right child?] If RIGHT[PTR] $\neq$ NULL, then:
   - (*a*) Set PTR := RIGHT[PTR].
   - (*b*) Go to Step 2. 2.

   [End of If structure.]
7. Set PTR := STACK[TOP] and TOP := TOP − 1. [Pops node.]

   [End of Step 4 loop.]
8. Exit.

Traversal Algorithms Using Stacks …

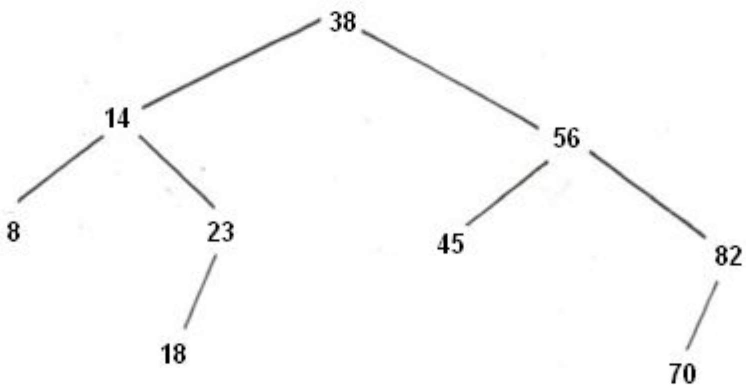
Algorithm : POSTORD(INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory. This algorithm does a postorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

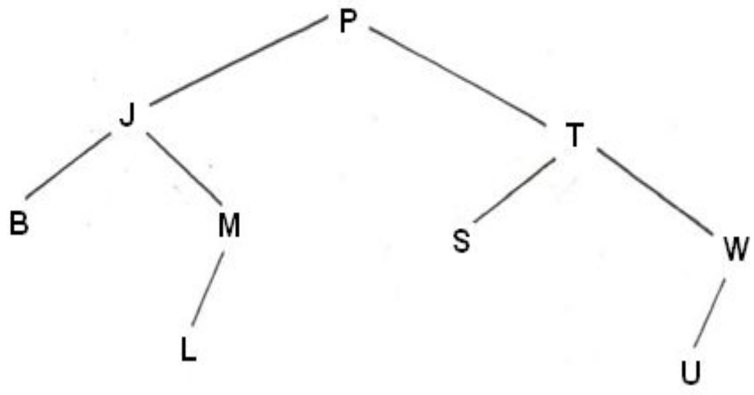
1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. [Push left-most path onto STACK.]
   Repeat Steps 3 to 5 while PTR $\neq$ NULL:
3. Set TOP := TOP + 1 and STACK[TOP] := PTR.
   [Pushes PTR on STACK.]
4. If RIGHT[PTR] $\neq$ NULL, then: [Push on STACK.]
   Set TOP := TOP + 1 and STACK[TOP] := −RIGHT[PTR].
   [End of If structure.]
5. Set PTR := LEFT[PTR]. [Updates pointer PTR.]
   [End of Step 2 loop.]
6. Set PTR := STACK[TOP] and TOP := TOP − 1.
   [Pops node from STACK.]
7. Repeat while PTR > 0:
   (*a*) Apply PROCESS to INFO[PTR].
   (*b*) Set PTR := STACK[TOP] and TOP := TOP − 1
   [Pops node from STACK.]
   [End of loop.]
8. If PTR < 0, then:
   (*a*) Set PTR := −PTR.
   (*b*) Go to Step 2.
   [End of If structure.]
9. Exit.

Binary Search Trees

❖A binary tree *T* is called a ***binary search tree*** (or ***binary sorted tree***) if each node N of T has the following property:

The value at N is greater than every value in the left subtree of N and is less than or equal to every value in the right subtree of N.





Searching and Inserting in Binary Search Trees

- Suppose *T* is a binary search tree and an ITEM of information is given.
- Following algorithm finds the location of ITEM in the 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. (Search is successful)

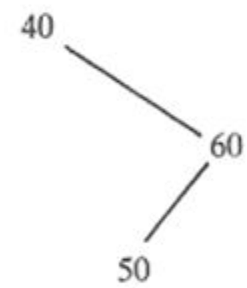
(ii) We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

Suppose the following six numbers are inserted in order into an empty binary search tree:

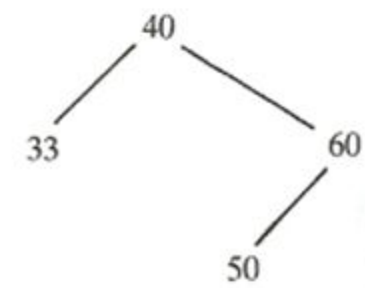
40, 60, 50, 33, 55, 11

(1) ITEM = 40.

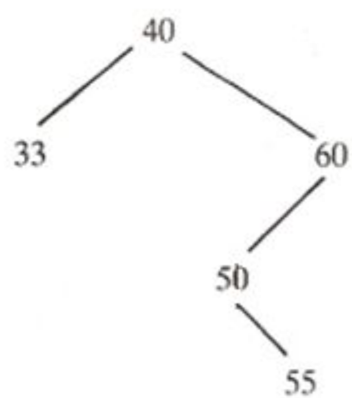
(2) ITEM = 60.



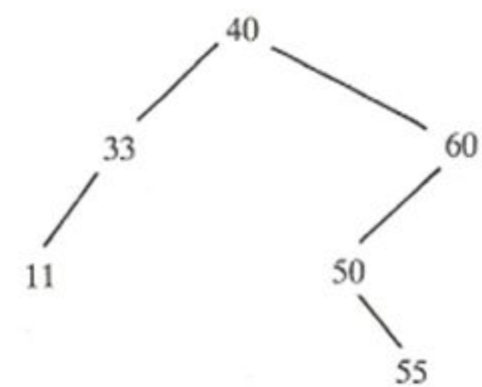
(3) ITEM = 50.



(4) ITEM = 33.



(5) ITEM = 55.



(6) ITEM = 11.

Algorithm : INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, 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 $\neq$ 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. [Add ITEM to tree.]
   If PAR = NULL, then:
       Set ROOT := NEW.
   Else if ITEM < INFO[PAR], then:
       Set LEFT[PAR] := NEW.
   Else:
       Set RIGHT[PAR] := NEW.
   [End of If structure.]
5. Exit.

Procedure: 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. [Tree empty?]
   If ROOT = NULL, then: Set LOC := NULL and PAR := NULL, and Return.
2. [ITEM at root?]
   If ITEM = INFO[ROOT], then: Set LOC := ROOT and PAR := NULL, and Return.
3. [Initialize pointers PTR and SAVE.]
   If ITEM < INFO[ROOT], then:
   Set PTR := LEFT[ROOT] and SAVE := ROOT.
   Else:
   Set PTR := RIGHT[ROOT] and SAVE := ROOT.
   [End of If structure.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5. [ITEM found?]
   If ITEM = INFO[PTR], then: Set LOC := PTR and PAR := SAVE, and Return.
6. If ITEM < INFO[PTR], then:
   Set SAVE := PTR and PTR := LEFT[PTR].
   Else:
   Set SAVE := PTR and PTR := RIGHT[PTR].
   [End of If structure.]
   [End of Step 4 loop.]
7. [Search unsuccessful.] Set LOC := NULL and PAR := SAVE.
8. Exit.

Deleting in a Binary Search Tree

- Suppose *T* is a binary search tree and an ITEM of information is given.
- For the deletion of node we have to consider following three cases:

  Case 1: *N has no children.* Then N deleted from T by simply replacing the location of N in the parent node P(N) the null pointer.

  Case 2: *N has exactly one child*. Then N is deleted from T by simply replacing the location in P(N) by the location of the only child of N.

  Case 3: *N has two children*. Let S(N) denote the inorder successor of N. Then N is deleted from T by first deleting S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

Algorithm : DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

A binary search tree T is in memory, and an ITEM of information is given. This algorithm deletes ITEM from the tree.

1. [Find the locations of ITEM and its parent, using Procedure]
   Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
2. [ITEM in tree?]
   If LOC = NULL, then: Write: ITEM not in tree, and Exit.
3. [Delete node containing ITEM.]
   If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL, then:
   Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   Else:
   Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LEFT[LOC] := AVAIL and AVAIL := LOC.
5. Exit.

Procedure: CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure deletes the node N at location LOC, where N does not have two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer CHILD gives the location of the only child of N, or else CHILD = NULL indicates N has no children.

1. [Initializes CHILD.]
   If LEFT[LOC] = NULL and RIGHT[LOC] = NULL, then:
       Set CHILD := NULL.
   Else if LEFT[LOC] $\neq$ NULL, then:
       Set CHILD := LEFT[LOC].
   Else
       Set CHILD := RIGHT[LOC].
   [End of If structure.]
2. If PAR $\neq$ NULL, then:
       If LOC = LEFT[PAR], then:
           Set LEFT[PAR] := CHILD.
       Else:
           Set RIGHT[PAR] := CHILD.
       [End of If structure.]
   Else:
       Set ROOT := CHILD.
   [End of If structure.]
3. Return.

Procedure: CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure will delete the node N at location LOC, where N has two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

1. [Find SUC and PARSUC.]
   (a) Set PTR := RIGHT[LOC] and SAVE := LOC.
   (b) Repeat while LEFT[PTR] ≠ NULL:
       Set SAVE := PTR and PTR := LEFT[PTR].
       [End of loop.]
   (c) Set SUC := PTR and PARSUC := SAVE.
2. [Delete inorder successor, using Procedure CASEA]
   Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).
3. [Replace node N by its inorder successor.]
   (a) If PAR ≠ NULL, then:
       If LOC = LEFT[PAR], then:
           Set LEFT[PAR] := SUC.
       Else:
           Set RIGHT[PAR] := SUC.
       [End of If structure.]
       Else:
           Set ROOT := SUC.
       [End of If structure.]
   (b) Set LEFT[SUC] := LEFT[LOC] and
       RIGHT[SUC] := RIGHT[LOC].
4. Return.