TREE

Tree

- What is a tree?

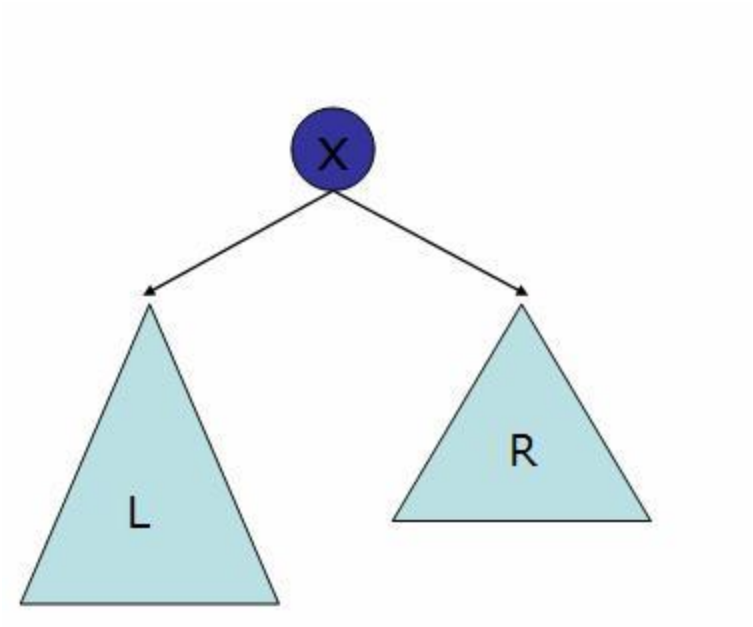
- Tree is a data structure similar to linked list
- Instead of pointing to one node each node can point to a number of point
- **Non linear data structure**
- Way of representing hierarchal nature of a structure in a graphical form

Operations

- Basic Operations
  - Inserting an element into tree
  - Deleting an element from tree
  - Searching for an element
  - Traversing the tree
- Auxiliary operations
  - Finding – height of tree, degree of node

Tree types

- General Tree
- Binary Tree
- Binary Search Tree

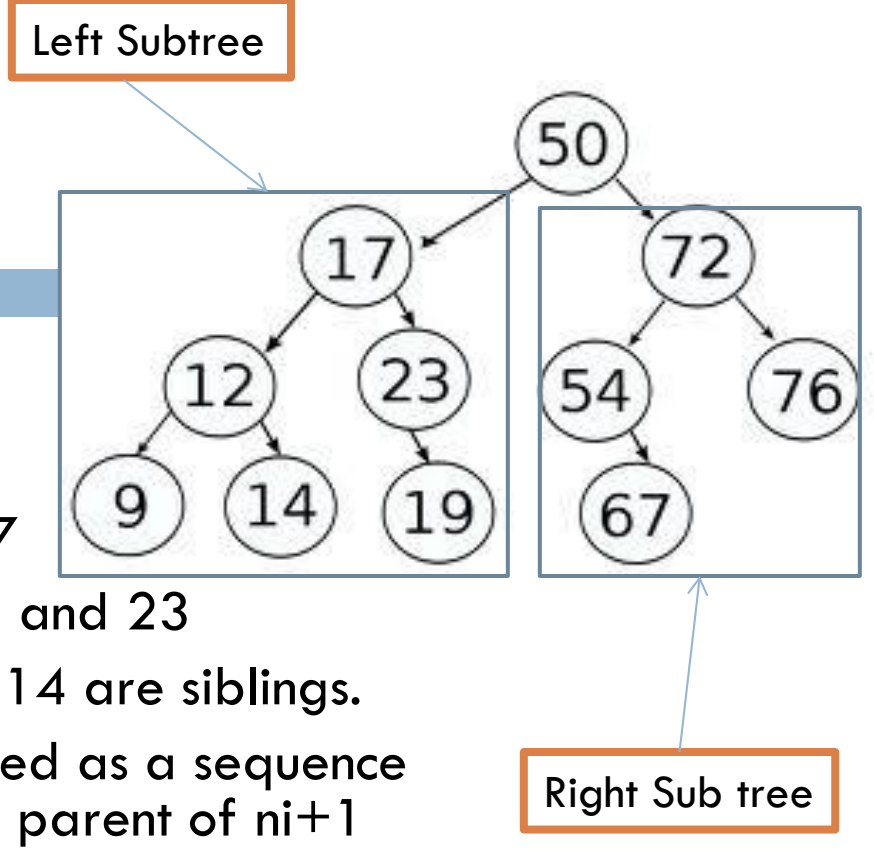


Tree

Definition of Tree

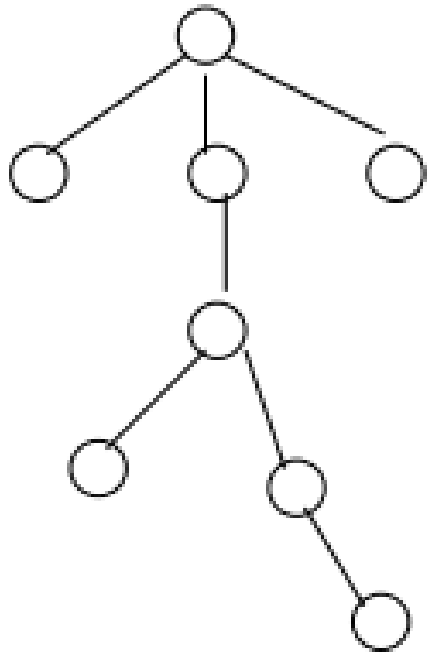
- A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into n>=0 disjoint sets T_1, ..., T_n, where each of these sets is a tree.
  - We call T_1, ..., T_n the subtrees of the root.

Tree terminology



- **Root :** 50
- **Leaves :** 9, 14, 19,67,76
- **Child(descendant)** : 12 is the child of 17
- **Parent(ancestor)** :17 is the parent of 12 and 23
- **Sibling:** Nodes with same parent. So, 9, 14 are siblings.
- **Path:** A path from node n1 to nk is defined as a sequence of nodes n1, n2,…., nk such that ni is the parent of ni+1 for i = 1 to k.
- **Path Length:** The length of the path is the number of edges on the path, namely k-1. So, 50 to 19 the length is 3.
- **Depth**= It is the length of the unique path from the root to ni. Thus, root is at depth 0 and 76 is at depth 2.
- **Height:** It is the length of the longest path from n_i to a leaf. Hence height of leaves are 0 and height of 23 is 1.

Level	height	depth	Index
1	4	0	0
2	0, 3, 0	1	1,2,3
3	2	2	4
4	0, 1	3	5,6
5	0	4	7

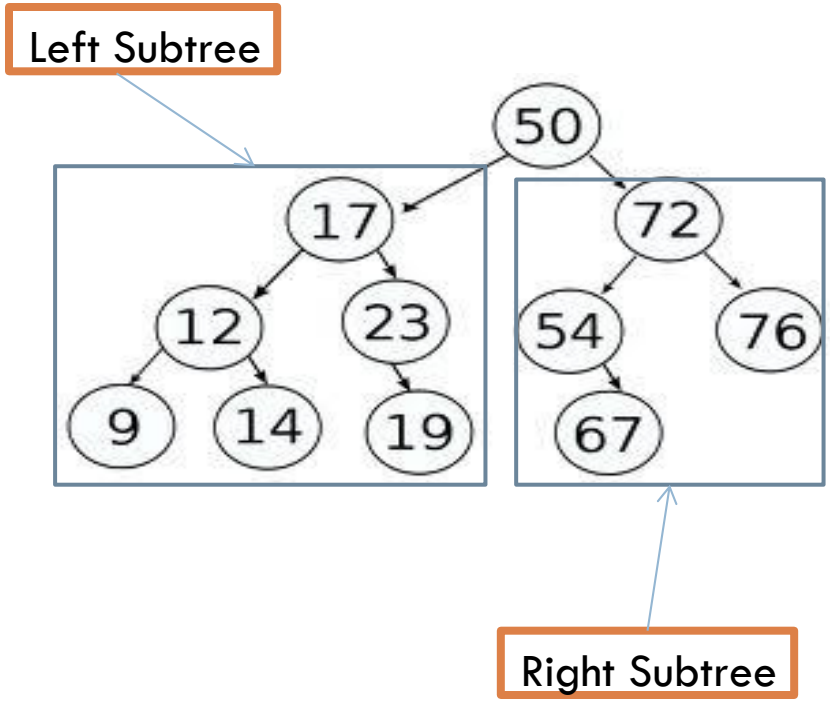


Tree terminology (cont.,)

- *Degree*: The number of subtrees of a node (No of nodes attached to a given node).
- *Degree of a tree*: The maximum of the degree among all the nodes in the tree.

Binary Tree Definition

- A **binary tree** is a tree data structure in which each node has at most two children (referred to as the *left* child and the *right* child).



Binary Tree Properties

- Properties of binary trees
  - **Lemma 5.1 [*Maximum number of nodes*]:**
  1. The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
  2. The maximum number of nodes in a binary tree of level k is 2^k-1, $k \geq 1$.
  - **Lemma 5.2 [*Relation between number of leaf nodes and degree-2 nodes*]:**

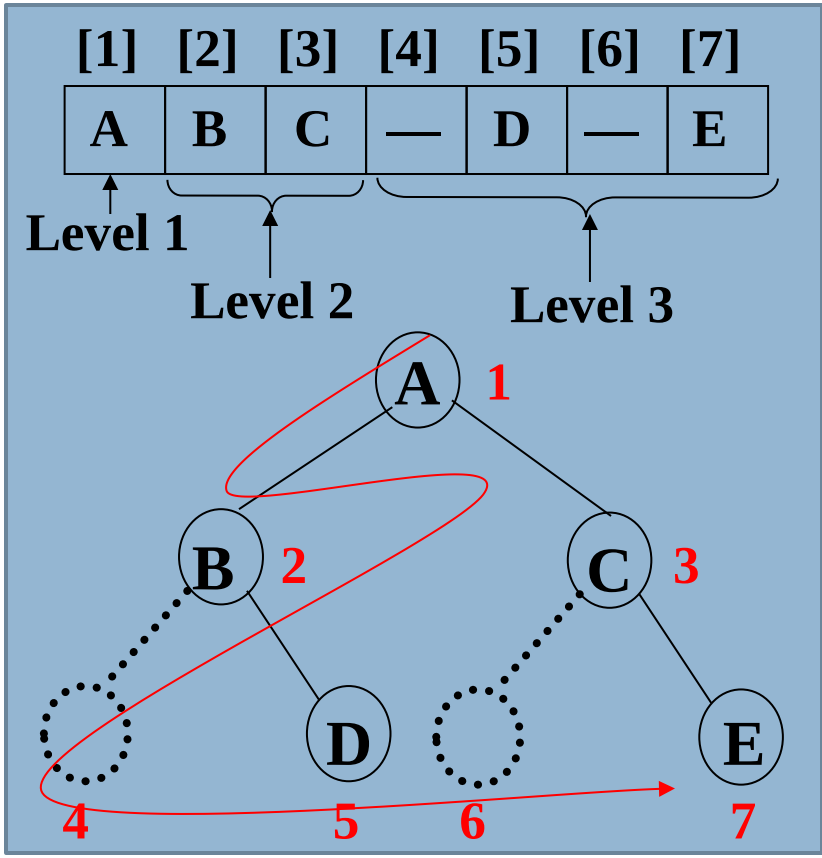
    For any nonempty binary tree, T, if n_l is the number of leaf nodes and n_d is the number of nodes of degree 2, then $n_l = n_d + 1$.
- These lemmas allow us to define ~~full~~ and complete binary trees

Binary Trees : Positioning and Labeling

Binary tree representations (using array)

- **Lemma 5.3:** If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

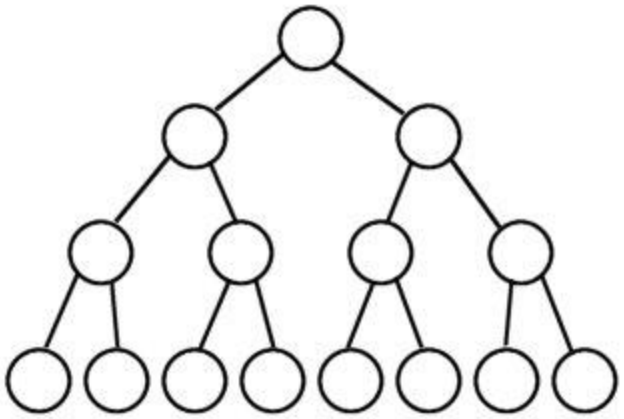
1. *parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
   If $i = 1$, i is at the root and has no parent.
2. *LeftChild(i)* is at $2i$ if $2i \le n$.
   If $2i > n$, then i has no left child.
3. *RightChild(i)* is at $2i+1$ if $2i+1 \le n$.
   If $2i+1 > n$, then i has no right child



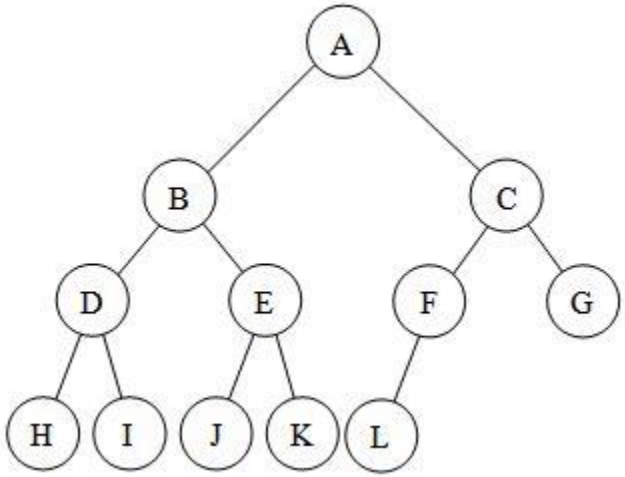
Full binary tree or Complete binary tree

- Full binary tree is a tree in which every node other than the leaves has two children.
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Full Binary Tree

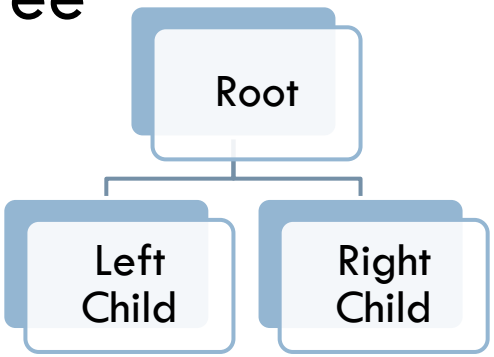


Complete Binary Tree

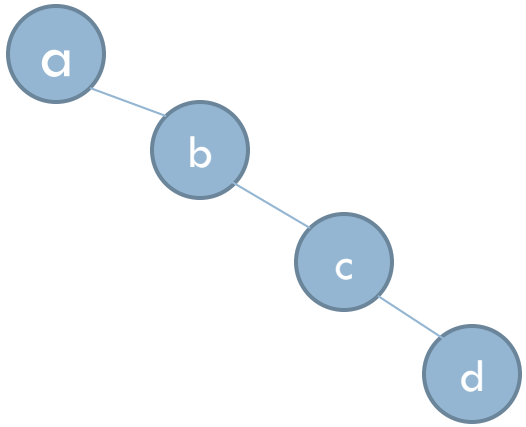


Types of binary tree

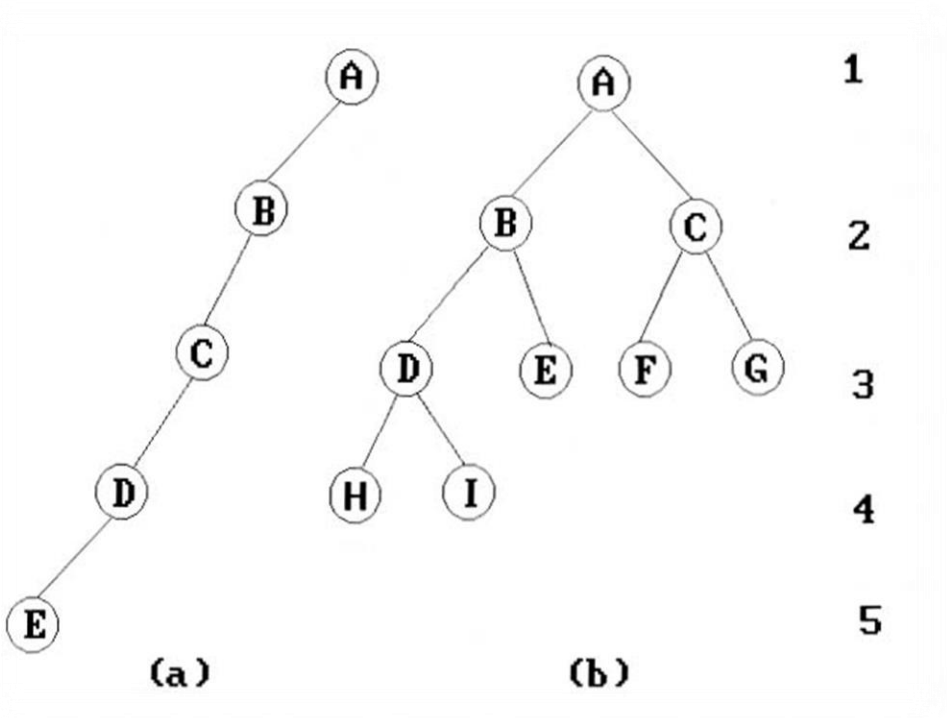
- Generic binary tree



- Skewed binary tree (Worst Case)



Binary tree representations (using array)

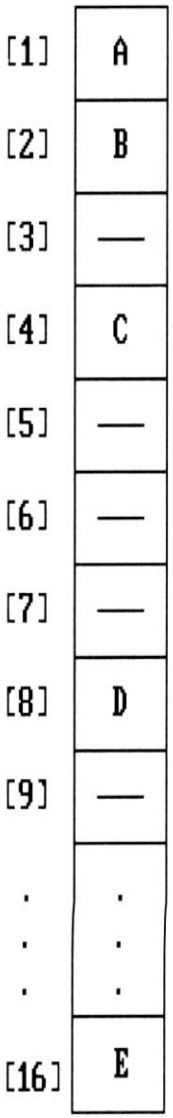


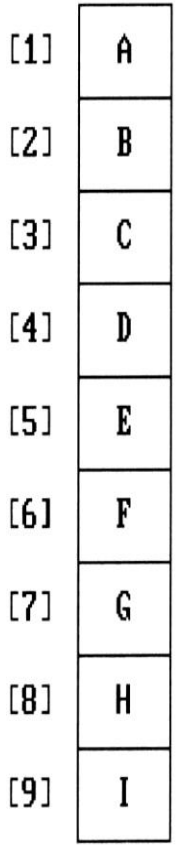
[1]	A
[2]	B
[3]	—
[4]	C
[5]	—
[6]	—
[7]	—
[8]	D
[9]	—
.	.
.	.
.	.
[16]	E

[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Binary Trees

- Binary tree representations (using array)
  - Waste spaces: in the worst case, a skewed tree of level *k* requires 2^k-1 spaces. Of these, only *k* spaces will be occupied
  - Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in the level of these nodes





Binary Trees representations (using link)

```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
        };
```

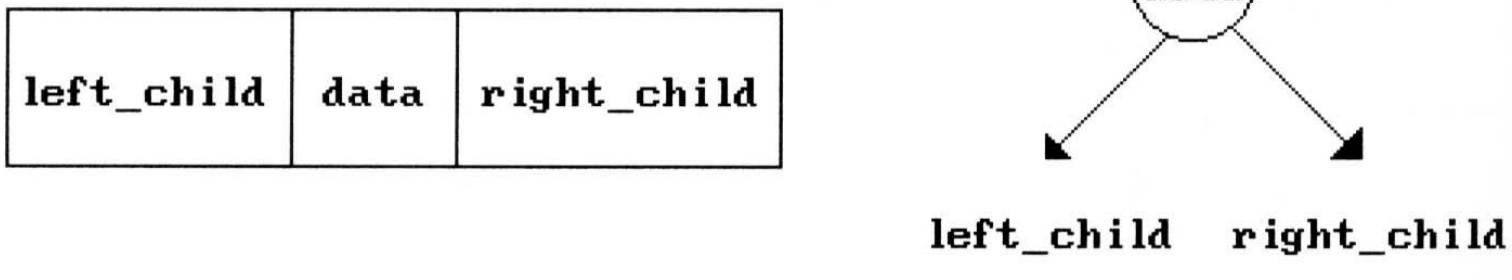


Figure 5.12: Node representation for binary trees

Binary Trees representations (using link)

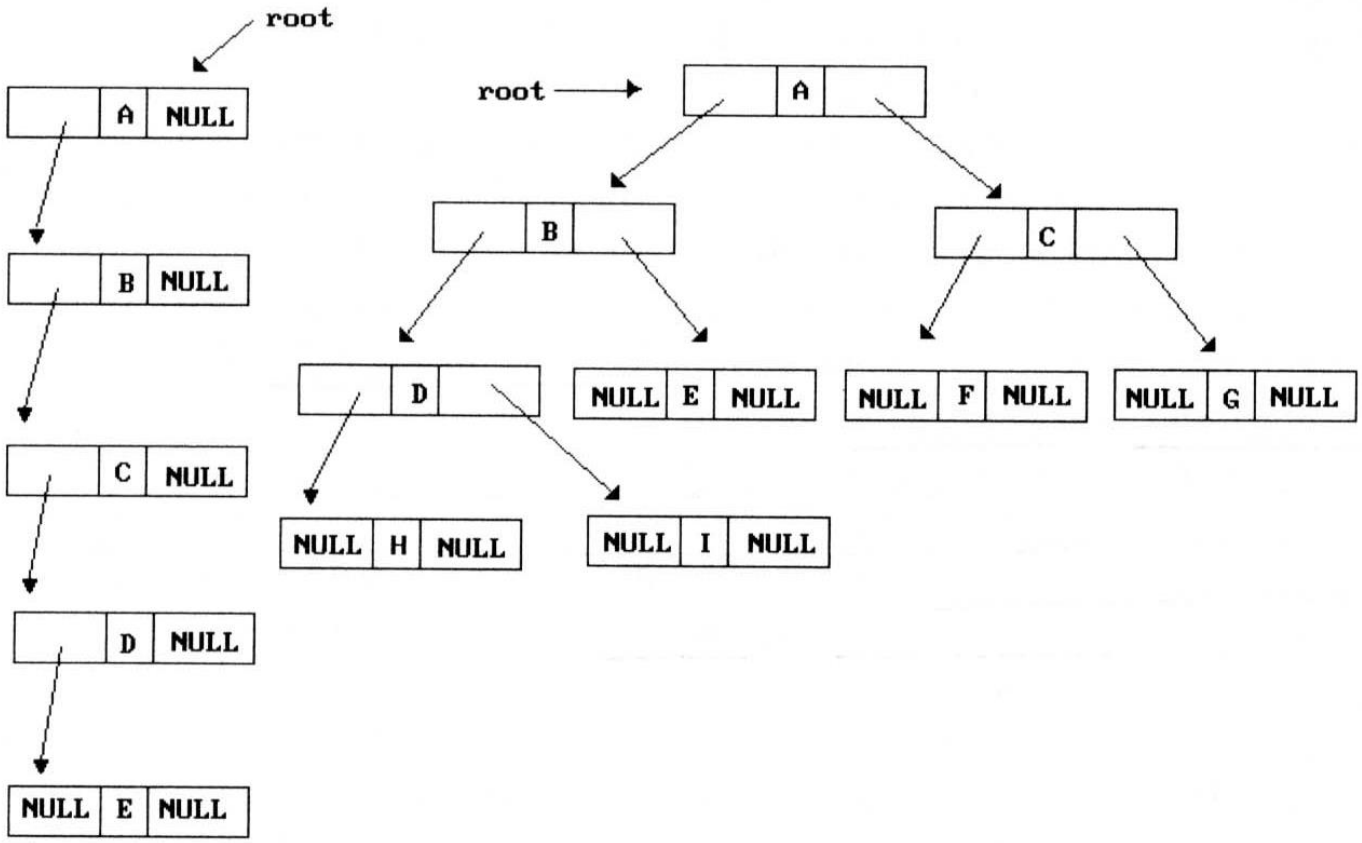
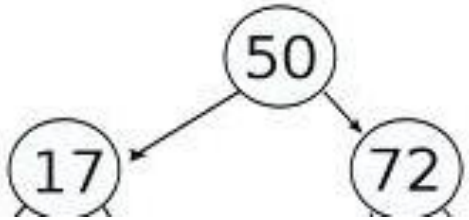


Figure 5.13: Linked representation for the binary trees of Figure 5.9

Binary Search Tree

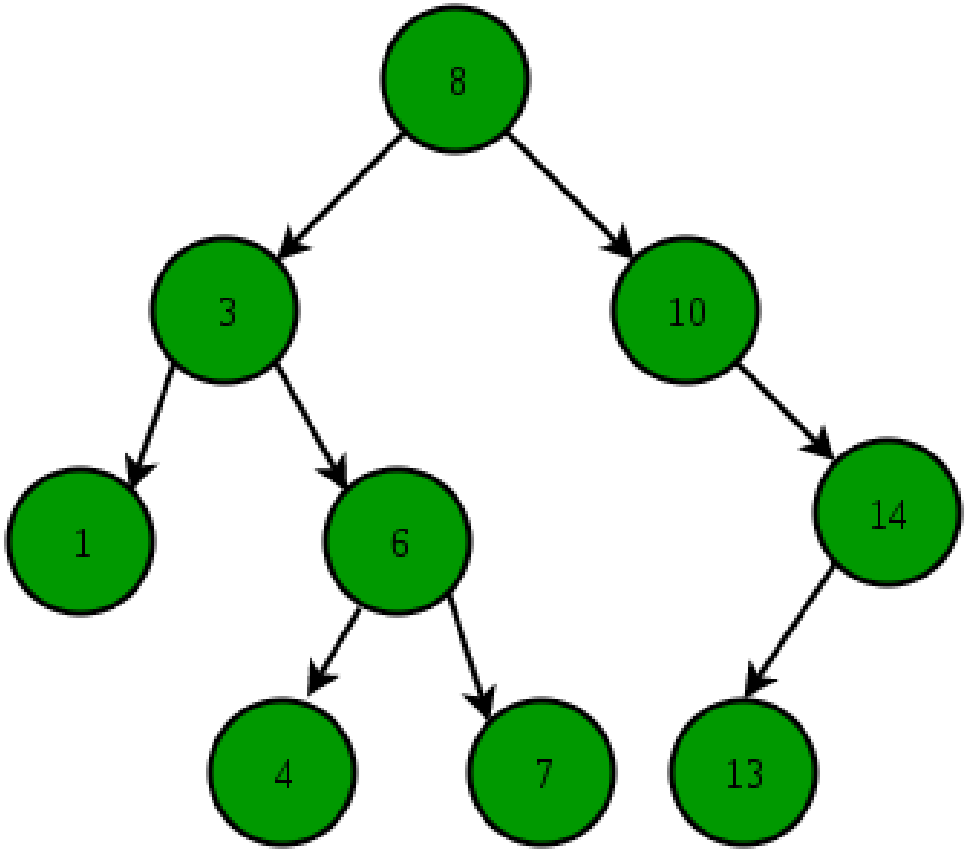
- In a binary tree, the left child having value less than the root node and the right child having value greater than root node is called Binary Search Tree.



In this example 17 is lesser than 50 and 72 is greater than 50.

So we can call this as a binary search tree.

Binary Search Tree (BST)



Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

When elements are given in a sequence,

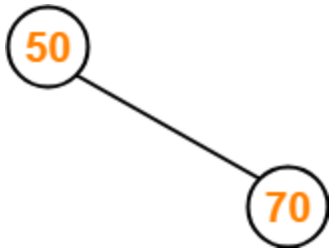
Always consider the first element as the root node.

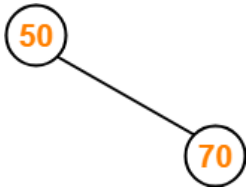
Consider the given elements and insert them in the BST one by one.

Insert 50-

Insert 70-

As 70 > 50, so insert 70 to the right of 50.

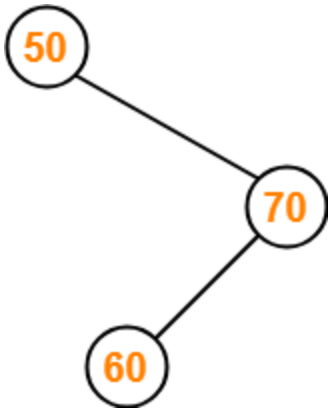


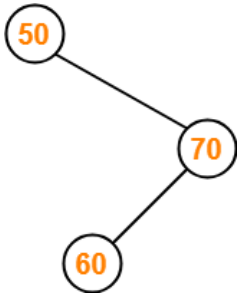


Insert 60-

As 60 > 50, so insert 60 to the right of 50.

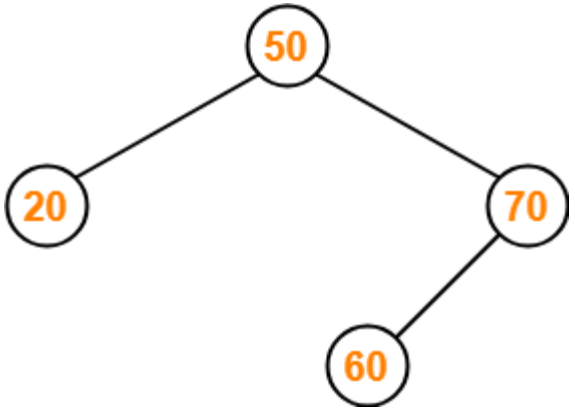
As 60 < 70, so insert 60 to the left of 70.



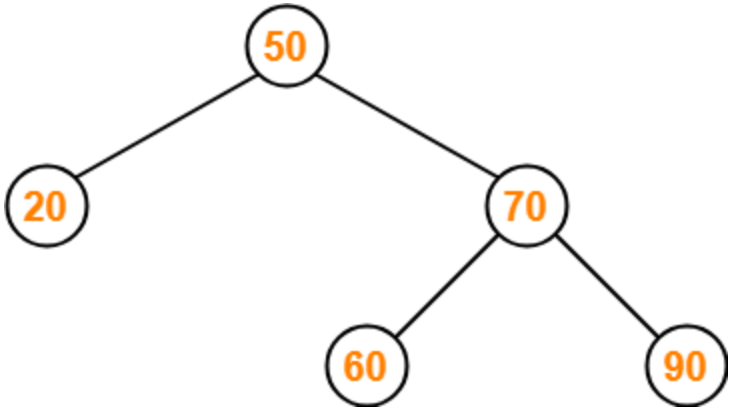


Insert 20-

As 20 < 50, so insert 20 to the left of 50.

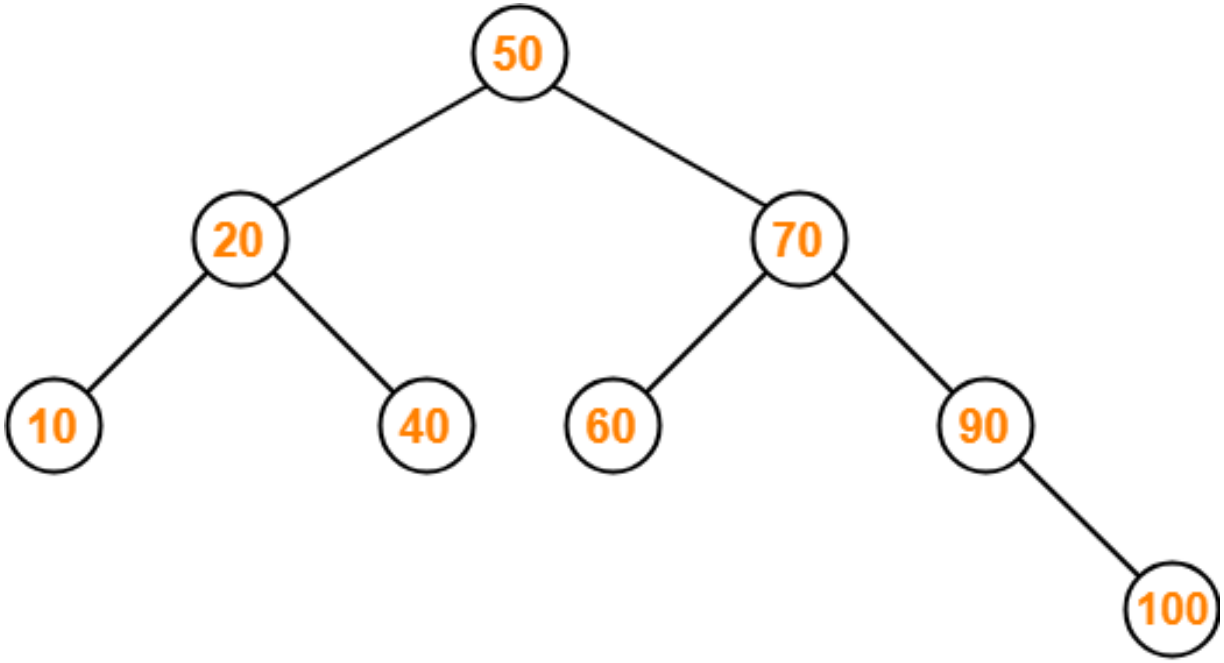


Insert 90-



After inserting 10, 40 100

- Final BST will look like this



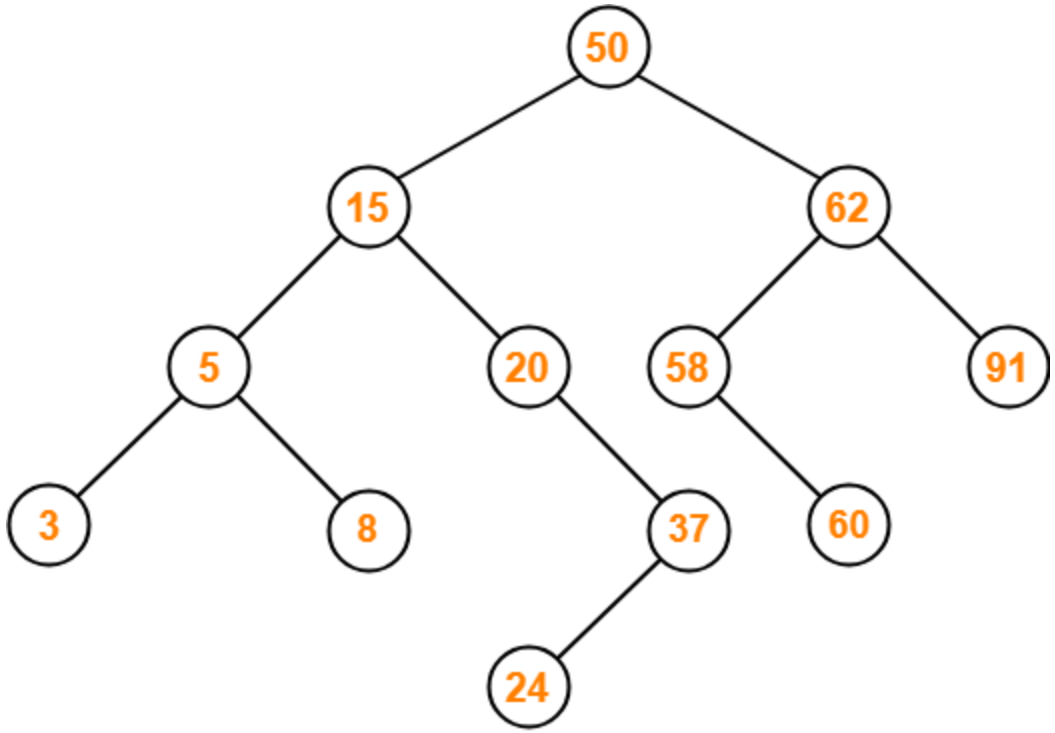
Binary Search Tree

Exercise

A binary search tree is generated by inserting in order of the following integers-

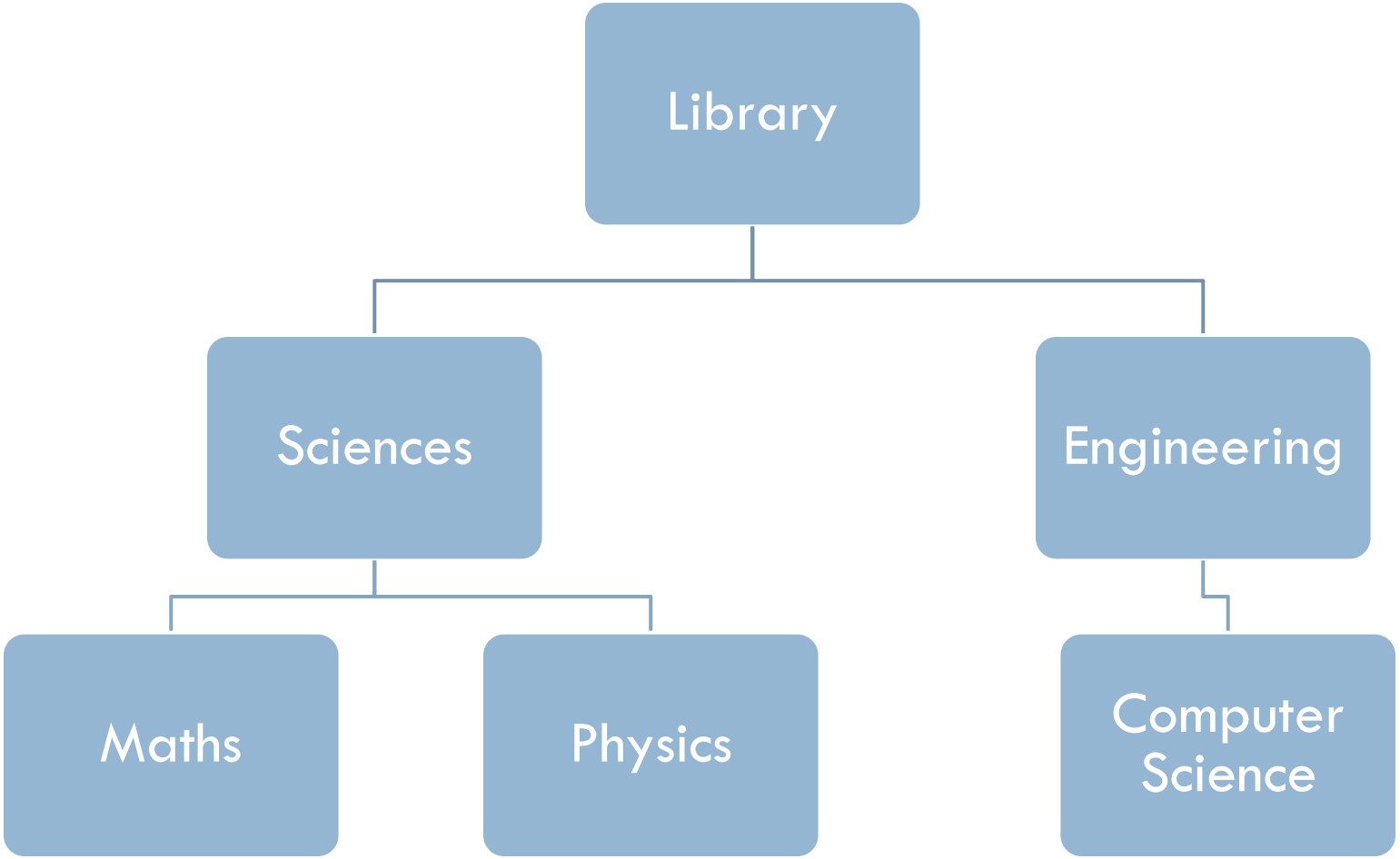
50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

Solution:

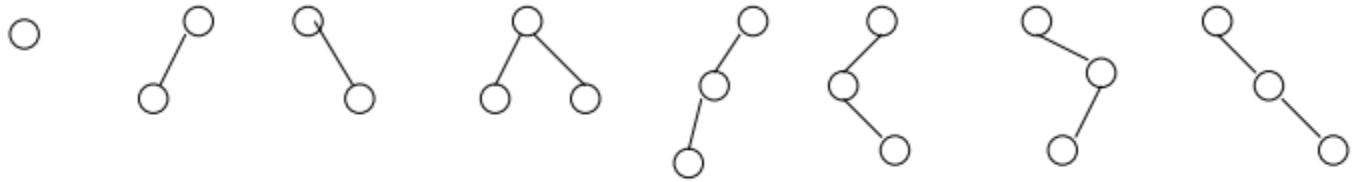


Binary Search Tree

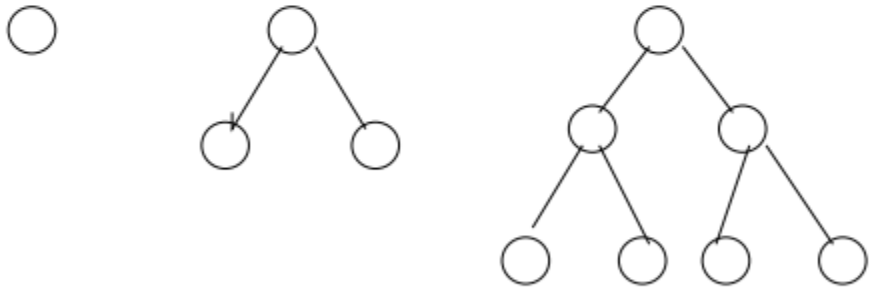
Real time example



Binary tree: a tree in which each node has at most two children.



Complete binary tree: tree in which all leaves are on the same level and each non-leaf node has *exactly* two children.

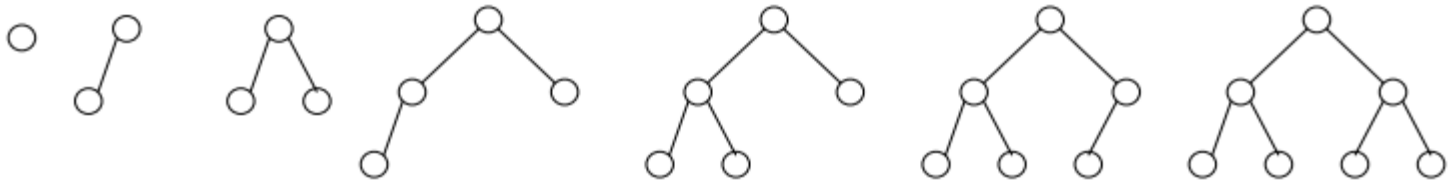


Important Facts:

- A complete binary tree with L levels contains $2^L - 1$ nodes.

Leftmost binary tree

Leftmost binary tree: like a complete binary tree, except that the bottom level might not be completely filled in; however, all leaves at bottom level are as far to the left as possible.



Important Facts:

- A leftmost binary tree with L levels contains between 2^{L-1} and $2^L - 1$ nodes.

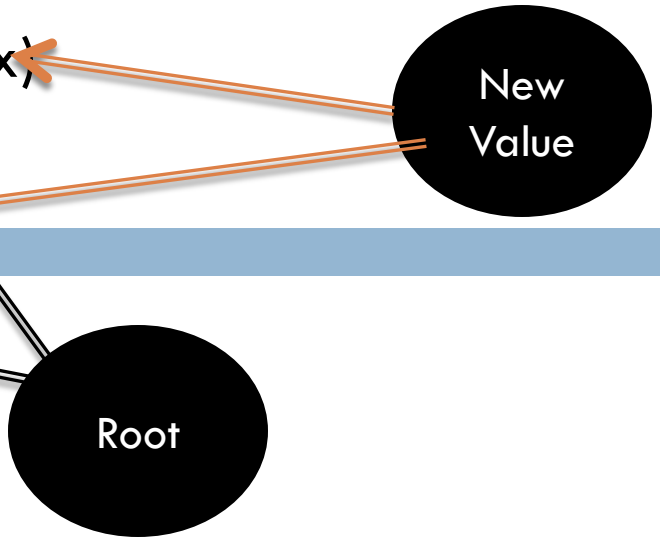
Implementation

```
 main()
{
 int op,n,r,s,f,level;
f = 0;
root=(struct node *)malloc(sizeof(struct node));
printf("Enter the root value:");
scanf("%d",&r);
root->data=r;
root->right=root->left=NULL;
```

```
do
{
    printf("\n 1.Insertion");
    printf("\n 2.Preorder");
    printf("\n 3.Inorder");
    printf("\n 4.Postorder");
    printf("\n 5.Searching");
    printf("\n 4.Deletion");
  printf("\n 7.Quit");
  printf("\n Enter your choice\n");
  scanf("%d",&op);
```

```
switch (op)
    {
     case 1: printf("\n Enter the element to insert\n");
                scanf("%d",&n);
                inser(root,n);
                break;
```

```
void inser(struct node *t,  int x)
{
if (t->data >x)
 if (t->left==NULL)
ins(t,x,1);
 else
inser(t->left,x);
 else if (t->data < x)
 if (t->right==NULL)
 ins(t,x,2);
 else
 inser(t->right,x);
 else
 printf("\n Element is already present in the list\n");
 }
```



```
void ins(struct node *n,int
   val,int opt)
{
   struct node *t;
   t=(struct node
   *)malloc(sizeof(struct
   node));
   t->data=val;
   t->right=t->left=NULL;
   if (opt==1)
     n->left=t;
   else
     n->right=t;
     printf("\n %d is
   inserted",val);
     if (opt==1)
     {
     printf("\tat the left\n");
     getch();
     }
     else
     {
     printf("\tat the right\n");
     getch();
     }
   }
```

```
case 2: printf("\n The preorder elements are\n");
          preorder(root);
          getch();
          break;
case 3: printf("\n The inorder elements are\n");
          inorder(root);
          getch();
          break;
case 4: printf("\n The postorder elements are\n");
         postorder(root);
         getch();
         break;
```

Preorder - NLR

```
void preorder(struct node *p)
 {
  if (p!=NULL)
  {
  printf("\n d",p->data);
  preorder(p->left);
  preorder (p->right);
  }
 }
```

Inorder - LNR

```
void inorder(struct node *p)
{
  if (p!=NULL)
  {
  inorder(p->left);
  printf("\n d",p->data);
  inorder (p->right);
  }
}
```

Postorder - LRN

```
void postorder(struct node *p)
 {
  if (p!=NULL)
  {
  postorder(p->left);
  postorder (p->right);
  printf("\n d",p->data);
  }
 }
```

Preoder, Inorder, Postorder

- In Preorder, the root is visited before (pre) the subtrees traversals
- In Inorder, the root is visited in-between left and right subtree traversal
- In Preorder, the root is visited after (pre) the subtrees traversals

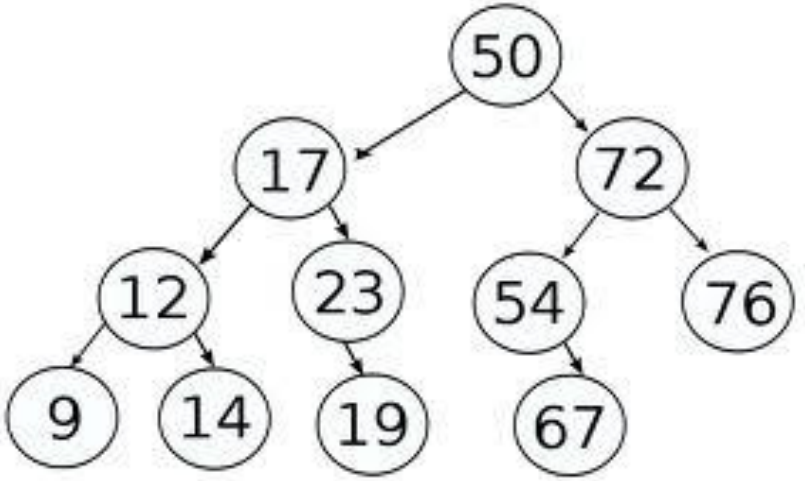
Preorder Traversal:
1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

Inorder Traversal:
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

Postorder Traversal:
1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

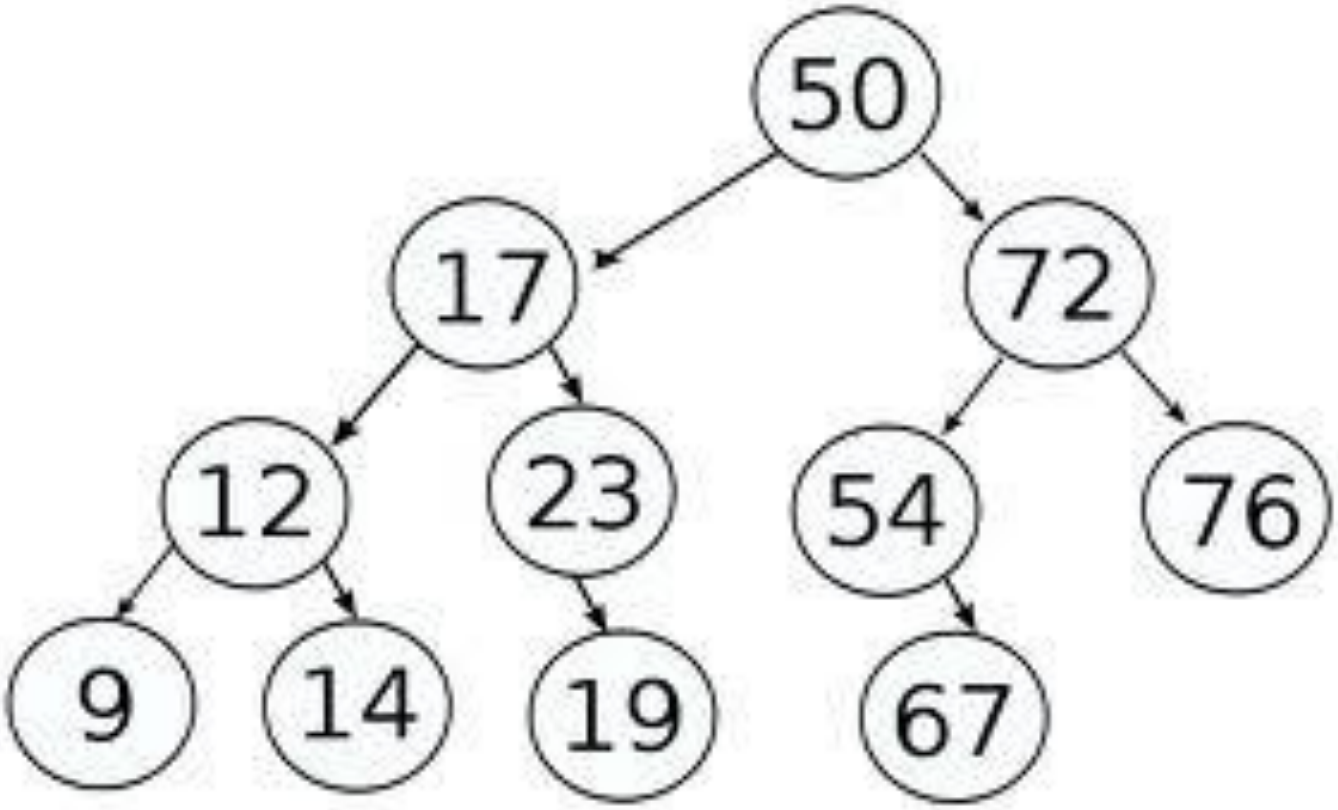
Tree traversal

- Preorder : NLR
- Inorder : LNR (Sorted)
- Postorder : LRN

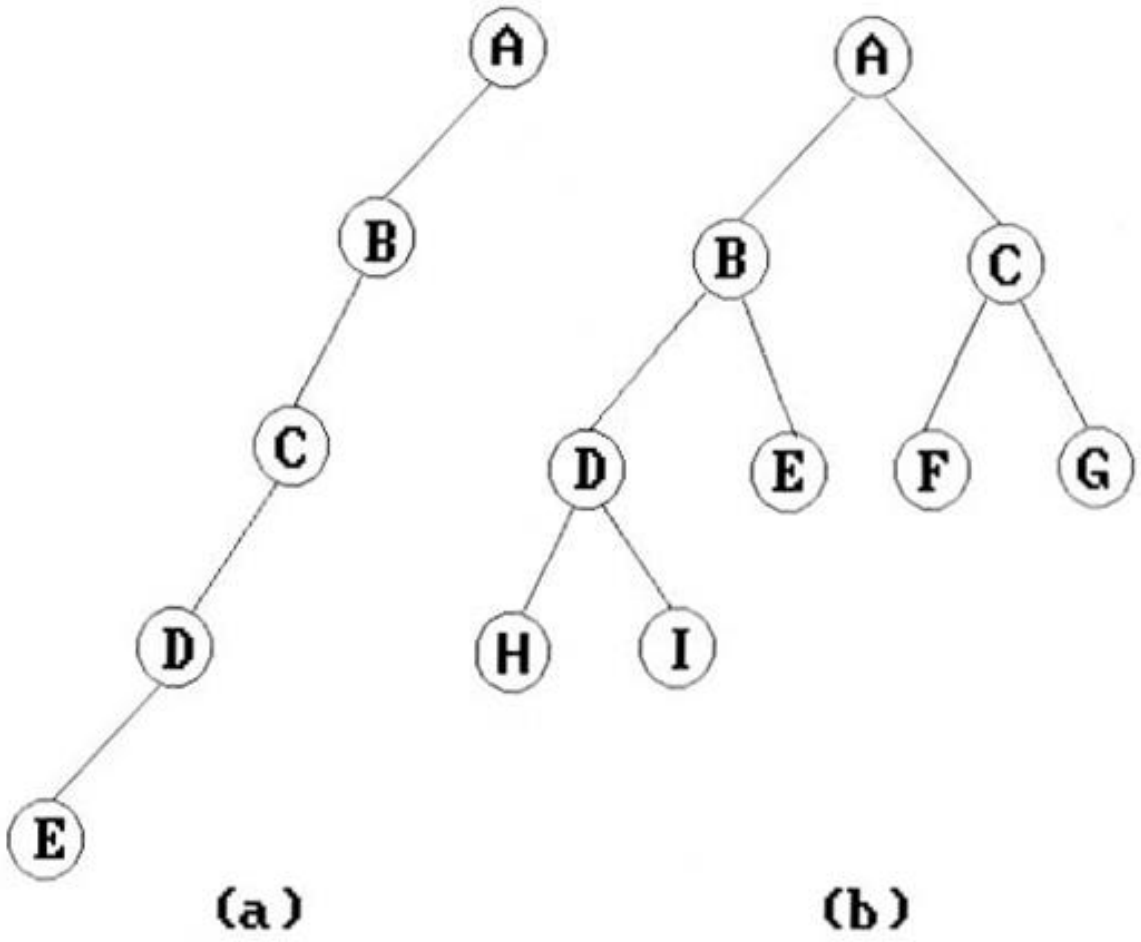


- Inorder : 9, 12, 14, 17, 23, 19, 50, 54, 67, 72, 76
- Preorder : 50, 17, 12, 9, 14, 23, 19, 72, 54, 67, 76
- Postorder : 9, 14, 12, 19, 23, 17, 67, 54, 76, 72, 50

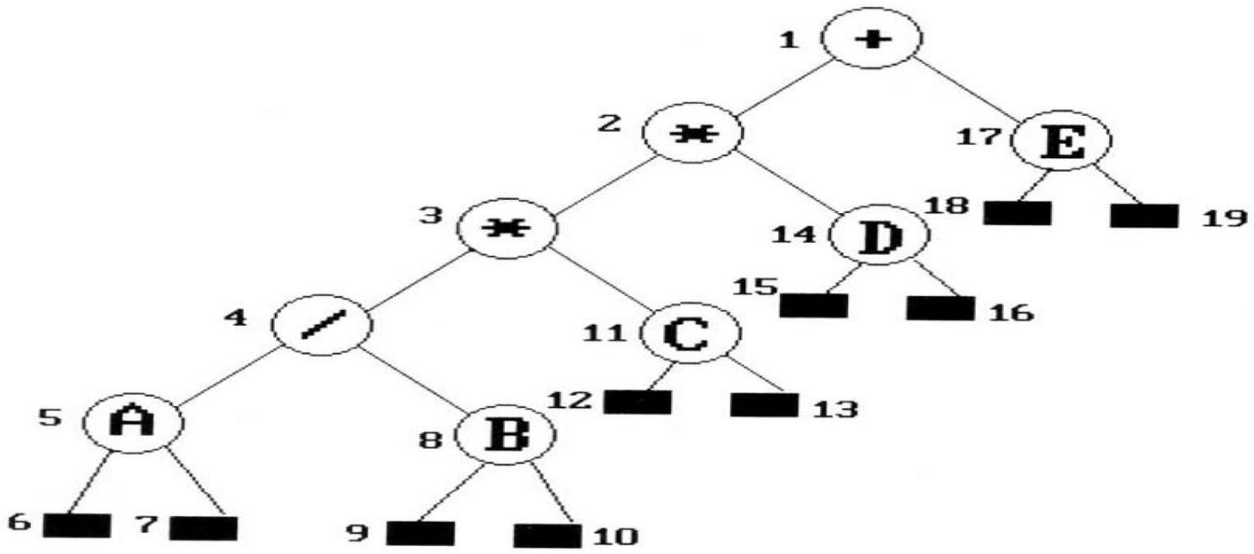
try



Exercise



Expression Tree



The above figure is an example for an Expression Tree.

Definition:

It's a Binary tree in which the non-terminals are operators and the Leaves or Terminals are operands.

Construction of an Expression Tree

Input : Post fix Notation

Output : Expression Tree

Procedure:

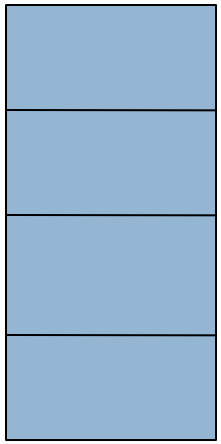
1. Read the post fix notation character by character from left to right.
2. When an operand is encountered push it on to stack.
3. When an operator is encountered pop the top most two character from the stack. A new tree is formed with operator as root and operand1 as RST and operand2 as LST. The newly formed tree is once again pushed on to the stack.
4. Repeatedly perform the operation until all the characters are read. The final resultant tree which is popped out from the stack would be an expression Tree.

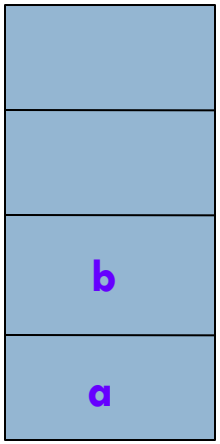
Construction of an Expression Tree

Input:

a b + c d e + * *</u>

1. **Read the character a and b – Both are operands so push it on to the stack. Initially the stack is empty.**





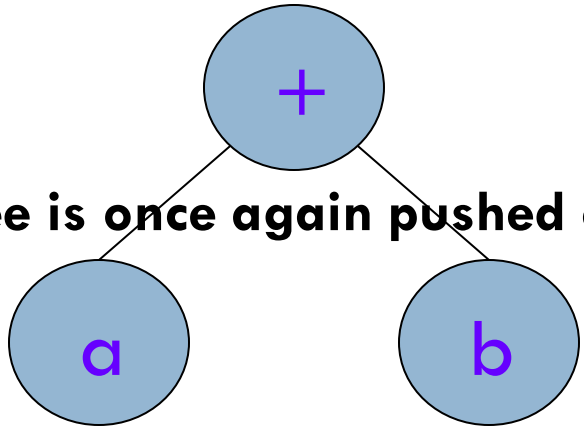
Initial Status of Stack **After Reading a & b**

(empty)

Construction of an Expression Tree

2. The Third Character is an operator so we pop out the two characters from the stack. Assign the 1st popped item as T1 and Second popped out character as T2.

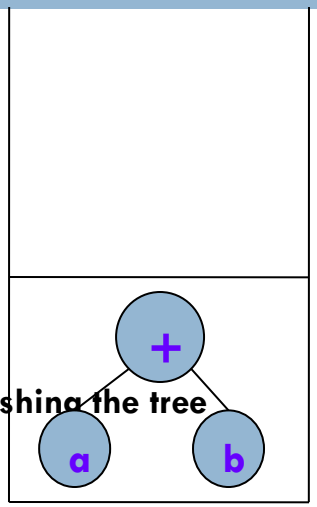
3. Assign T1 as RST and T2 as LST with the operator as Root node.



4. The resultant tree is once again pushed on to stack

Construction of an Expression Tree

4. The resultant tree is once again pushed on to stack

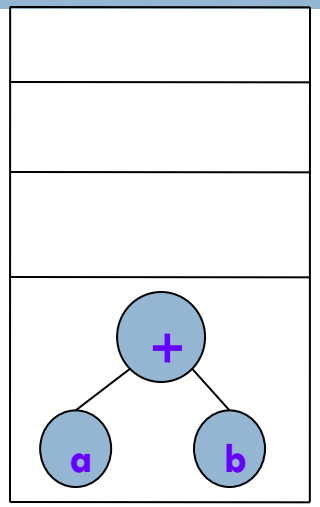


Stack after pushing the tree

e

d

c



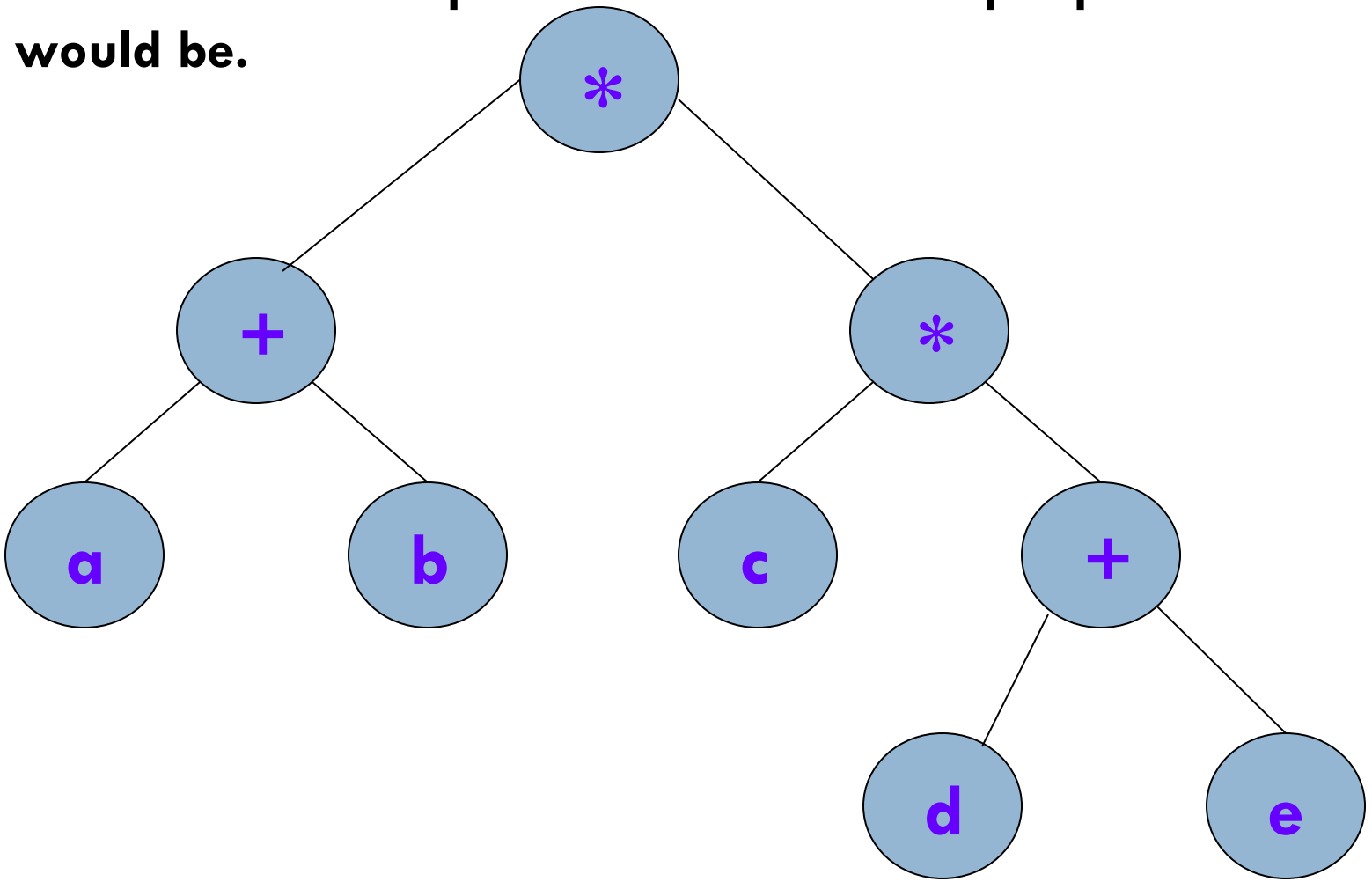
Stack after reading c, d & e

5. The next character is an operator so repeat step2, 3 and 4.

6. Repeatedly read and do the corresponding operation until the last character is read.

Construction of an Expression Tree

The Final Resultant Expression Tree for the input postfix notation would be.



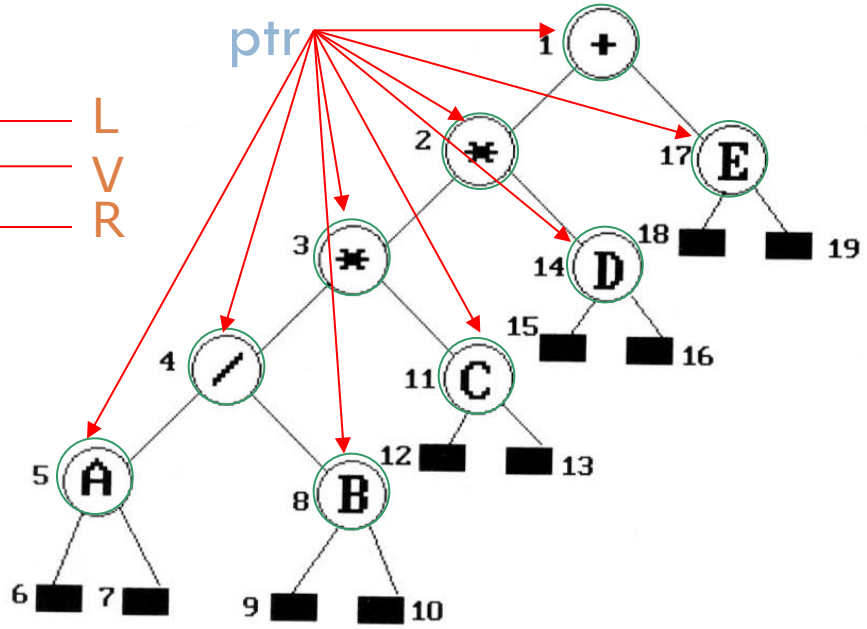
Binary Tree Traversals (1/7)

Inorder traversal (*LVR*) (recursive version)

output: A / B * C * D + E

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
   if (ptr) {
      inorder(ptr->left_child);
      printf("%d",ptr->data);
      inorder(ptr->right_child);
   }
}
```

L
V
R



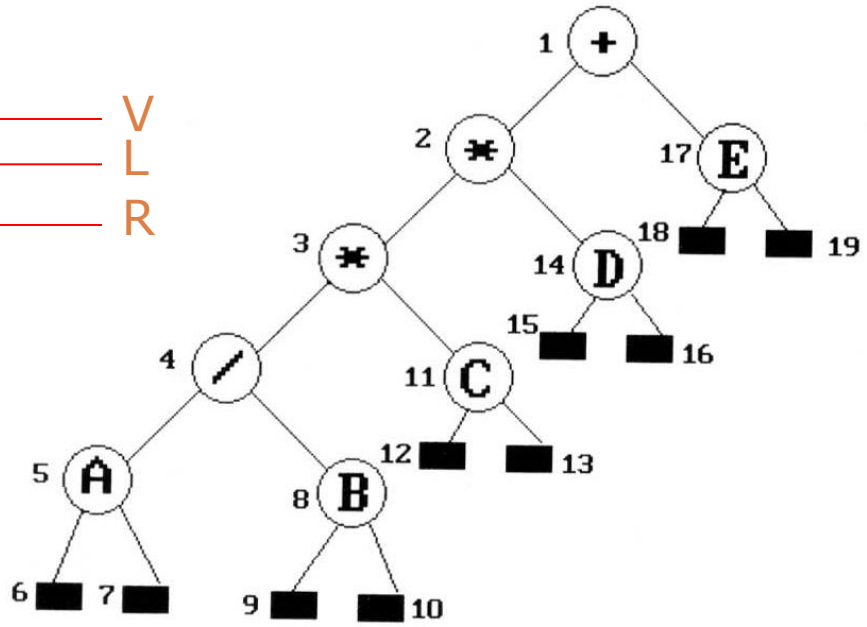
Binary Tree Traversals (2/7)

- Preorder traversal (*VLR*) (recursive version)

output: + * * /A B C D E

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
   if (ptr) {
      printf("%d",ptr->data);
      preorder(ptr->left_child);
      preorder(ptr->right_child);
   }
}
```

V
L
R



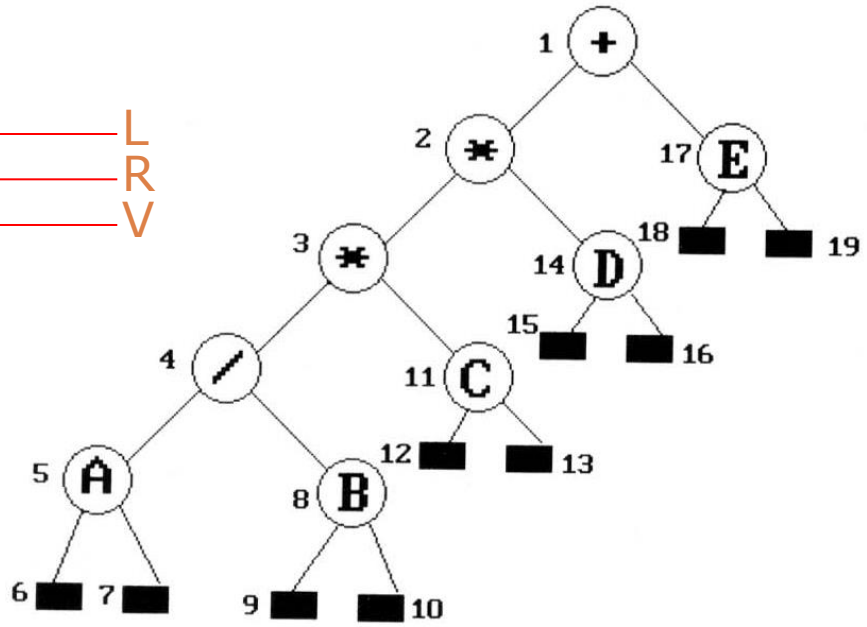
Binary Tree Traversals (3/7)

- Postorder traversal (*LRV*) (recursive version)

output: A B / C * D * E +

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
   if (ptr) {
      postorder(ptr->left_child);
      postorder(ptr->right_child);
      printf("%d",ptr->data);
   }
}
```

L
R
V



Binary Tree Traversals (4/7)

- Iterative inorder traversal
  - we use a stack to simulate recursion

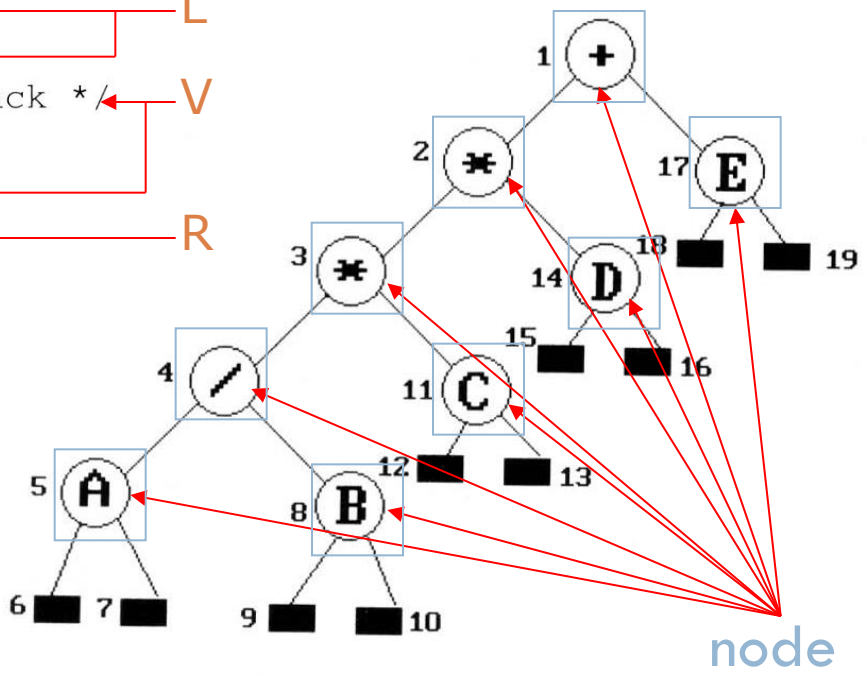
```
void iter_inorder(tree_pointer node)
{
   int top = -1; /* initialize stack */
   tree_pointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node->left_child)
         add(&top, node); /* add to stack */
      node = delete(&top); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node->data);
      node = node->right_child;
   }
}
```

5	8	11	14	17
A	B	C	D	E

L

V

R



output: A /B *C *D +E

Binary Tree Traversals (5/7)

- Analysis of inorder2 (Non-recursive Inorder traversal)
  - Let *n* be the number of nodes in the tree
  - Time complexity: O(*n*)
    - Every node of the tree is placed on and removed from the stack exactly once
  - Space complexity: O(*n*)
    - equal to the depth of the tree which (skewed tree is the worst case)

Binary Tree Traversals (6/7)

- Level-order traversal
  - method:
    - We visit the root first, then the root's left child, followed by the root's right child.
    - We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost nodes
  - This traversal requires a queue to implement

Binary Tree Traversals (7/7)

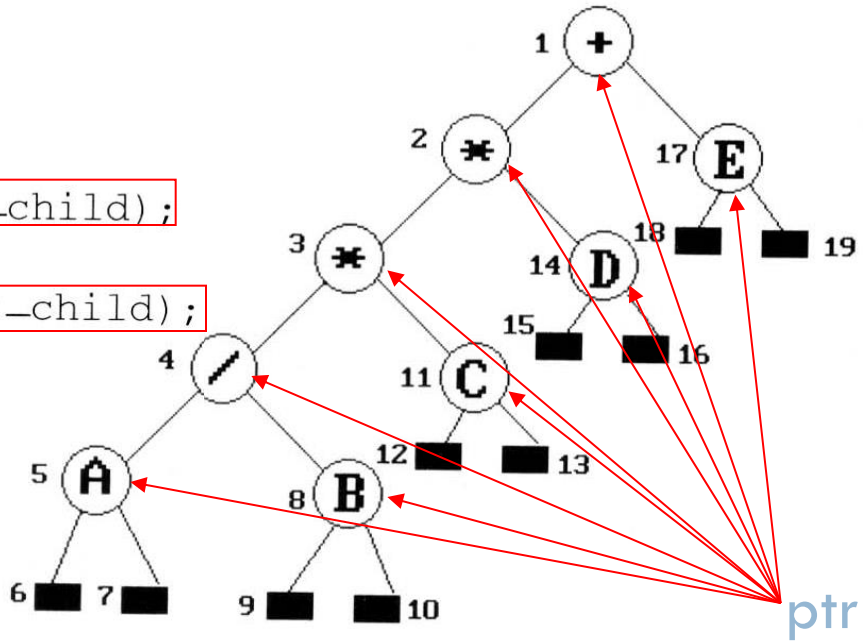
- Level-order traversal (using queue)

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
   int front = rear = 0;
   tree_pointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty tree */
   addq(front, &rear, ptr);
   for (;;) {
      ptr = deleteq(&front, rear);
      if (ptr) {
         printf("%d",ptr->data);
         if(ptr->left_child)
            addq(front,&rear,ptr->left_child);
         if (ptr->right_child)
            addq(front,&rear,ptr->right_child);
      }
      else break;
   }
}
```

FIFO

output: + * E * D / C A B

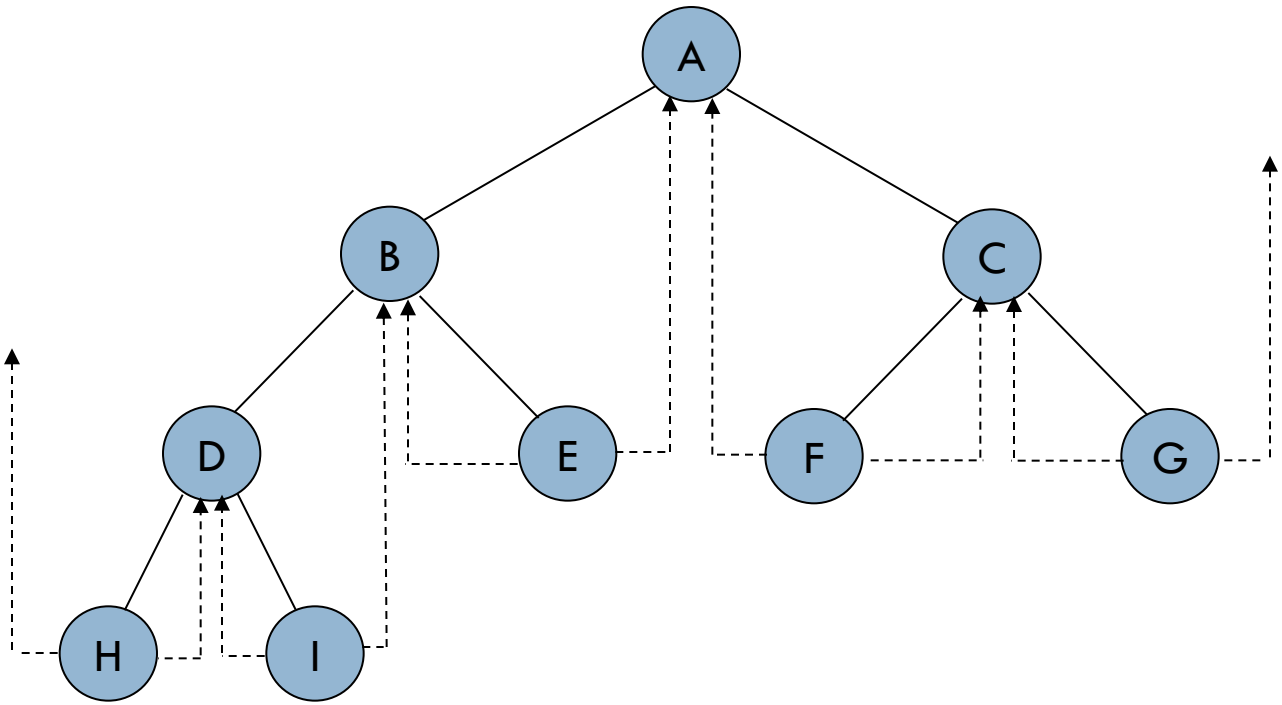
2	17	3	14	4	11	5	8	
*	E	*	D	/	C	A	B	



Threaded Binary Tree

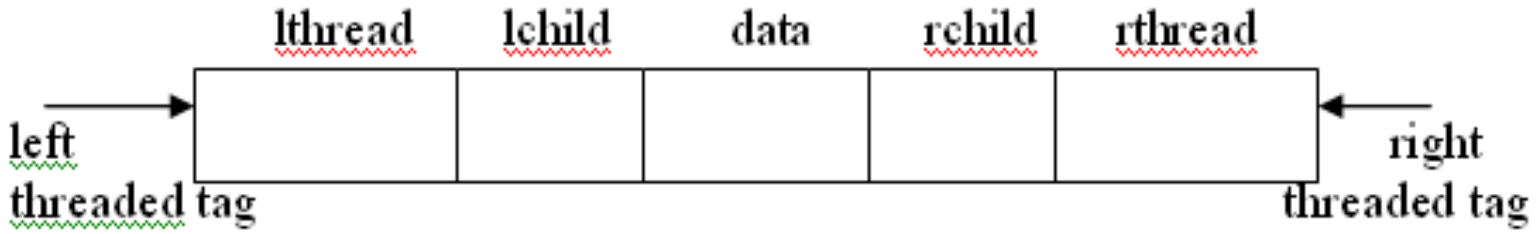
- Threading Rules
  - A 0 RightChild field at node p is replaced by a pointer to the node that would be visited after p when traversing the tree in inorder. That is, it is replaced by the inorder successor of p.
  - A 0 LeftChild link at node p is replaced by a pointer to the node that immediately precedes node p in inorder (i.e., it is replaced by the inorder predecessor of p).

Threaded Tree



Inorder sequence: H, D, I, B, E, A, F, C, G

- Therefore we have an alternate node representation for a threaded binary tree which contains five fields as show bellow:



For any node p, in a threaded binary tree.

lthred(p)=1 indicates lchild (p) is a thread pointer
lthred(p)=0 indicates lchild (p) is a normal
rthred(p)=1 indicates rchild (p) is a thread
rthred(p)=0 indicates rchild (p) is a normal pointer

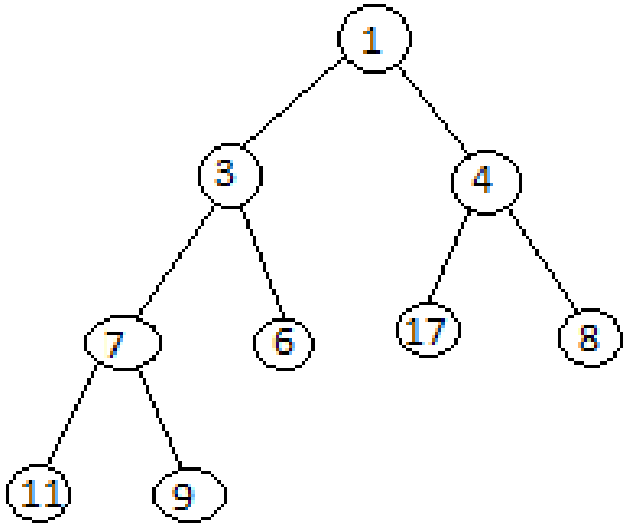
Heap

Heap Property

All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children.

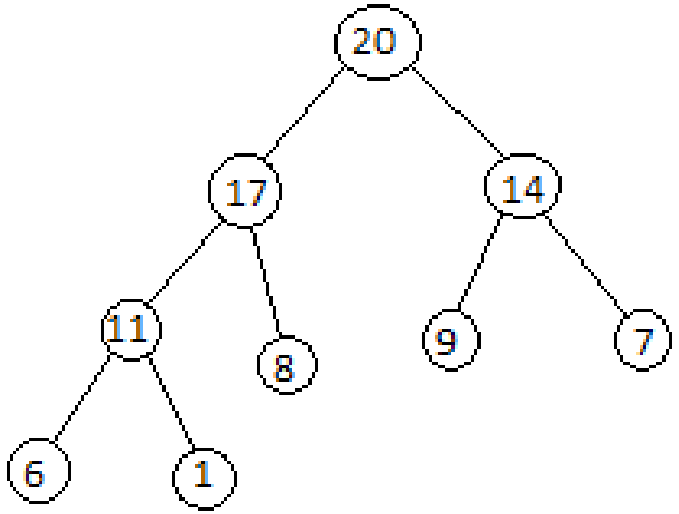
If the parent nodes are greater than their children, heap is called a **Max-Heap**.

If the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Algorithms

- The HEAPIFY procedure, which runs in $O(\log n)$ time, is the key to maintaining the heap property.
- The BUILD-HEAP procedure, which runs in linear time, produces a heap from an unordered input array, which runs in $O(n \log n)$ time.
- The HEAPSORT procedure, which runs in $O(n \log n)$ time, sorts an array in place.
- The EXTRACT-MAX and INSERT procedures, which run in $O(1\log n)$ time, allow the heap data structure to be used as a priority queue.

HEAPIFY

- HEAPIFY is an important subroutine for manipulating heaps.
- Its inputs are an array *A* and an index *i* into the array.

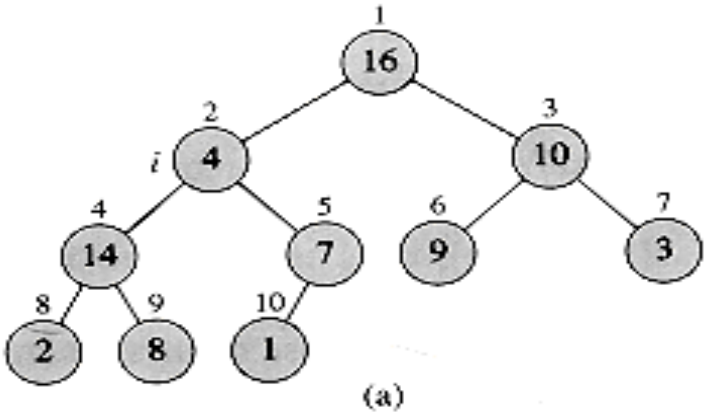
- When HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are heaps, but that *A*[*i*] may be smaller than its children, thus violating the heap property.

- The function of HEAPIFY is to let the value at *A*[*i*] "float down" in the heap so that the subtree rooted at index *i* becomes a heap.

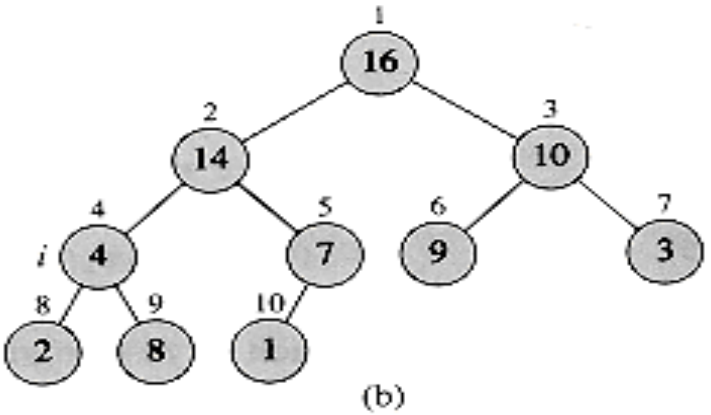
Heapify

```
HEAPIFY(A, i)
1 l ← LEFT(i)
2 r ← RIGHT(i)
3 if l ≤ heap-size[A] and A[l] > A[i]
4       then largest ← l
5       else largest ← i
6 if r ≤ heap-size[A] and A[r] > A[largest]
7       then largest ← r
8 if largest ≠ i
9       then exchange A[i] ↔ A[largest]
10            HEAPIFY(A, largest)
```

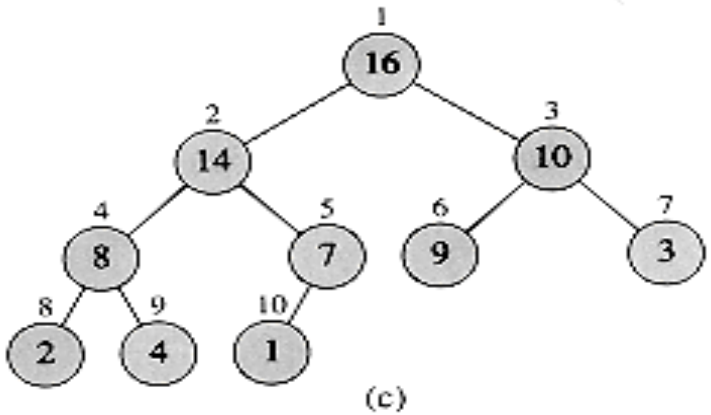
Heapify(A,2)



(a)



(b)



(c)

BUILD-HEAP

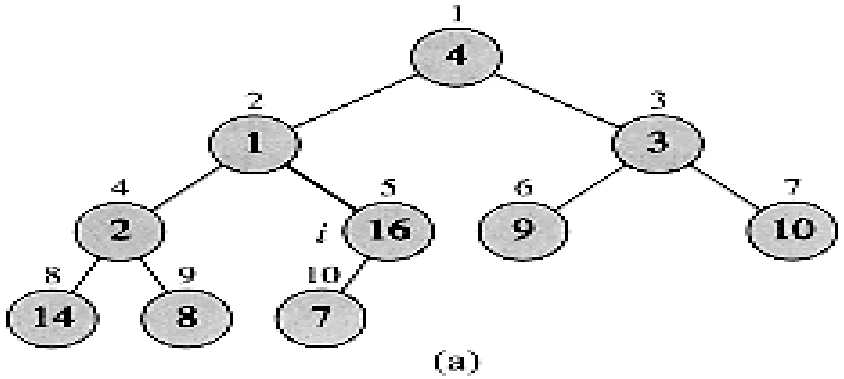
The procedure BUILD-HEAP goes through the remaining nodes of the tree and runs HEAPIFY on each one. It runs in $O(n \log n)$ time

we can build a heap from an unordered array

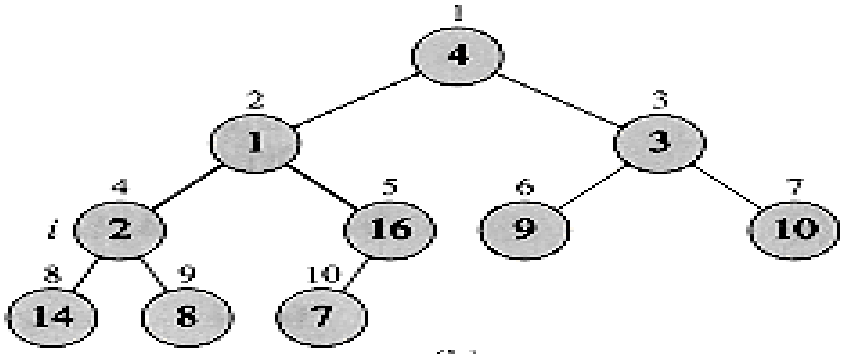
```
BUILD-HEAP(A)

1  heap-size[A] ← length[A]

2  for i ← ⌊length[A]/2⌋ downto 1

3        do HEAPIFY(A, i)
```

Build-Heap

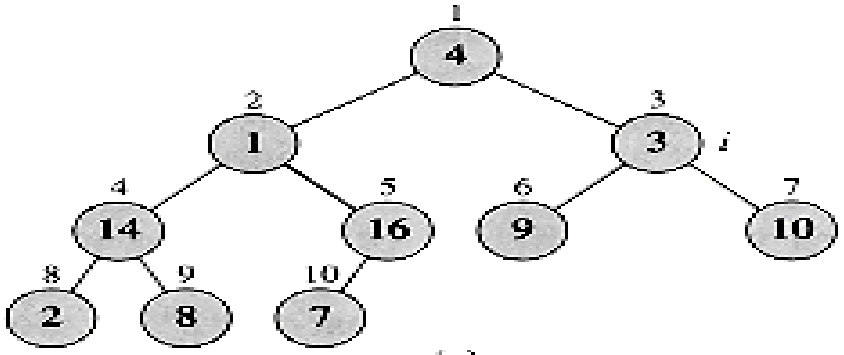
A	4	1	3	2	16	9	10	14	8	7



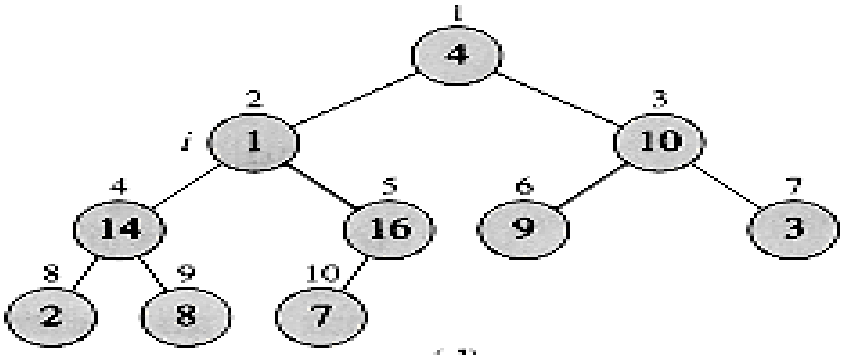
(a)



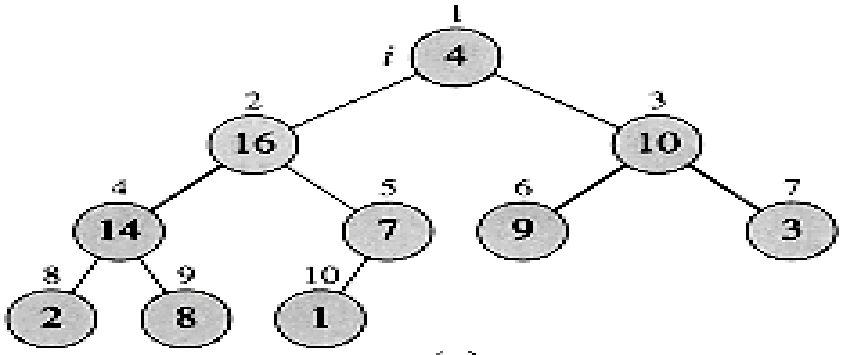
(b)



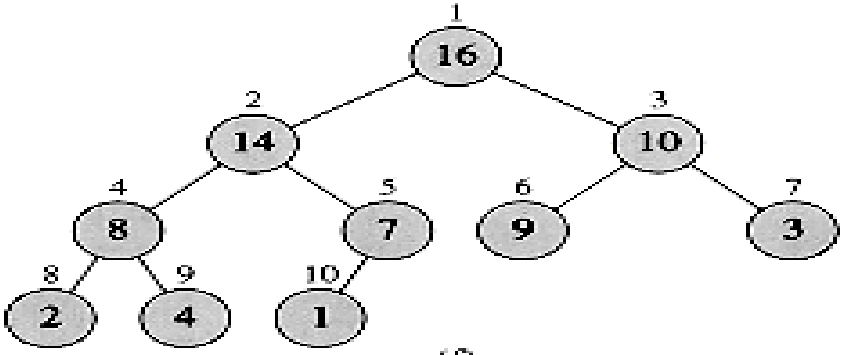
(c)



(d)



(e)



(f)

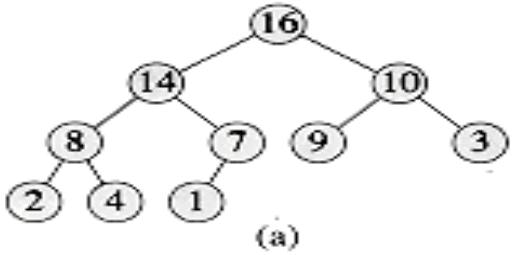
HeapSort

- The heapsort algorithm starts by using BUILD-HEAP to build a heap on the input array $A[1 . . n]$, where $n = length[A]$.
- Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$.
- If we now "discard" node n from the heap (by decrementing *heap-size*$[A]$), we observe that $A[1 . . (n - 1)]$ can easily be made into a heap.

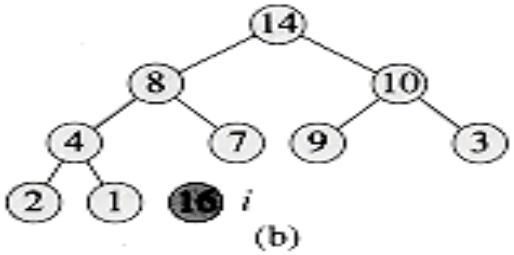
- The children of the root remain heaps, but the new root element may violate the heap property.

- All that is needed to restore the heap property, however, is one call to HEAPIFY(A, 1), which leaves a heap in $A[1 . . (n - 1)]$. The heapsort algorithm then repeats this process for the heap of size n - 1 down to a heap of size 2.

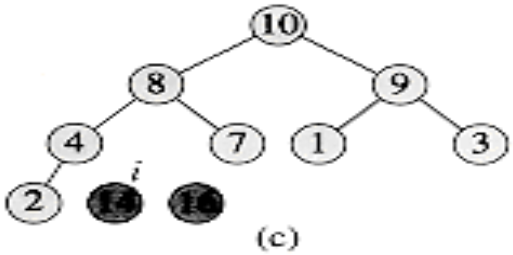
Heap Sort



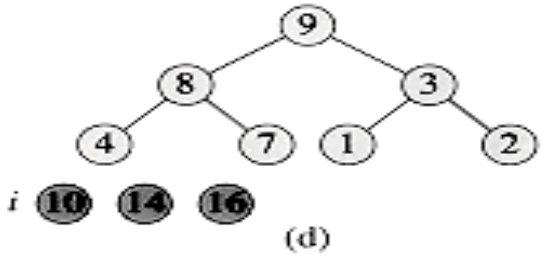
(a)



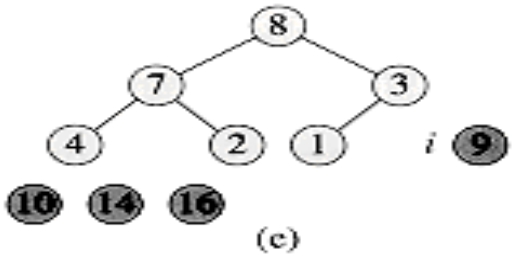
(b)



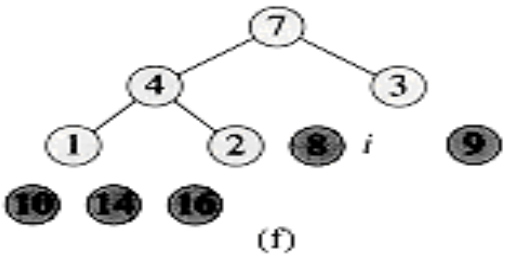
(c)



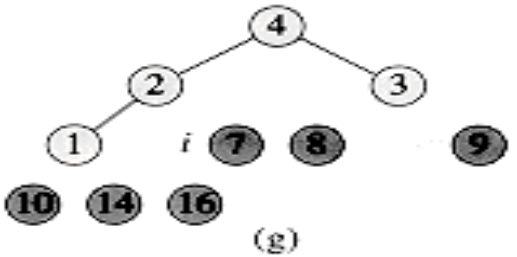
(d)



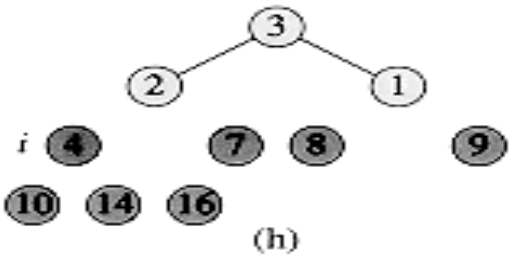
(e)



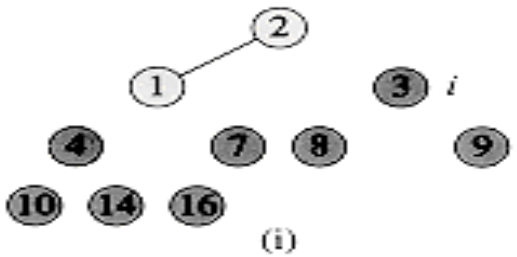
(f)



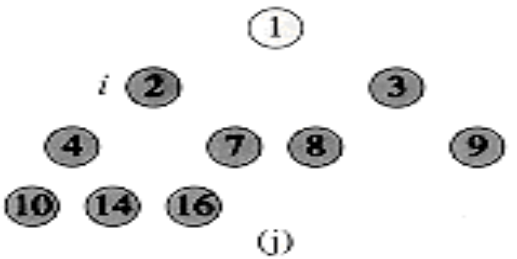
(g)



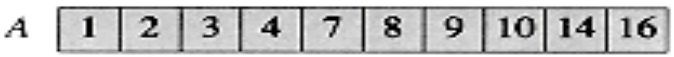
(h)



(i)



(j)



(k)

Heap Sort

```
HEAPSORT(A)

1   BUILD-HEAP(A)

2   for i ← length[A] downto 2

3         do exchange A[1] ↔ A[i]

4            heap-size[A] ← heap-size[A] -1

5            HEAPIFY(A, 1)
```

Applications of a heap:

- It can be used as an efficient priority queue.
- A ***priority queue*** is a data structure for maintaining a set *S* of elements, each with an associated value called a ***key***. A priority queue supports the following operations.
- INSERT(*S*, *x*) inserts the element *x* into the set *S*. This operation could be written as $S \leftarrow S \cup \{x\}$.
- MAXIMUM(*S*) returns the element of *S* with the largest key.
- EXTRACT-MAX(*S*) removes and returns the element of *S* with the largest key.

Extract Max

```
HEAP-EXTRACT-MAX(A)

1  if heap-size[A] < 1
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] - 1
6  HEAPIFY(A, 1)
7  return max
```

The running time of HEAP-EXTRACT-MAX is $O(\log n)$, since it performs only a constant amount of work on top of the $O(\log n)$ time for HEAPIFY.

Heap Insert

```
HEAP-INSERT(A,key)

1  heap-size[A] ← heap-size[A] + 1

2  i ← heap-size[A]

3  while i > 1 and A[PARENT(i)] < key

4       do A[i] ← A[PARENT(i)]

5          i ← PARENT(i)

6  A[i] ← key
```

The running time of HEAP-INSERT on an n-element heap is $O(\log n)$, since the path traced from the new leaf to the root has length $O(\log n)$.