Searching methods

Def: *Searching operation* is the process of finding an item in the array that meets some specified criterion.

Some common searching methods

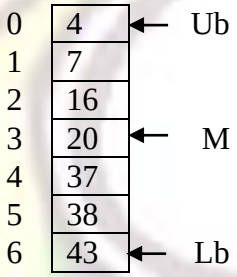
1. Linear Search (Sequential Search) (in an unordered array)
2. Binary Search

1. Linear Search

- Look at each item in the array in turn, and check whether that item is the one you are looking for.
- If so, the search is finished and the method returns it's index.
- If it is not found, then the method returns -1.
- This method is benefit if nothing is known about the order of the items in the array.

2. Binary Search

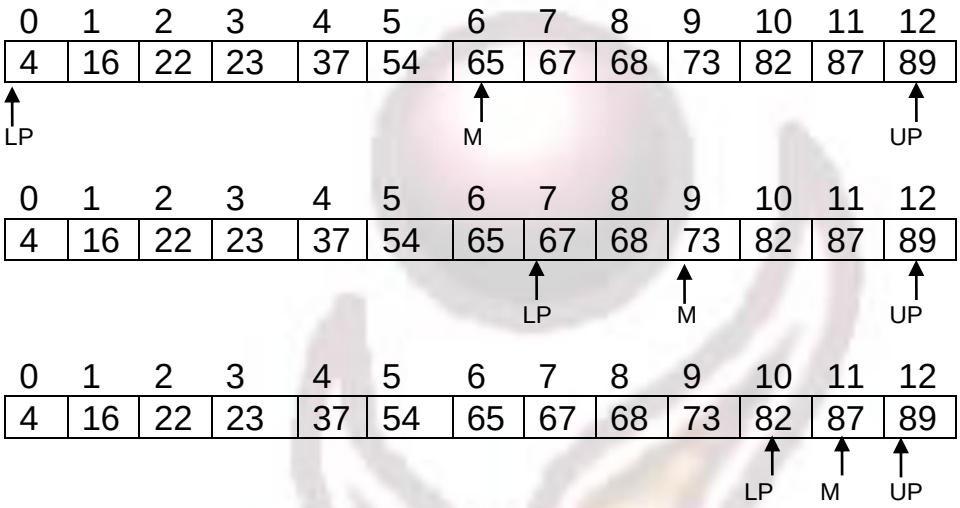
Is a method for searching for a given item in a sorted array.



- Variables *Lb* and *Ub* keep track of the *lower bound* and *upper bound* of the array, respectively.
- Begin searching by examining the middle element of the array.
- If the key we are searching for is equal to the middle element then returns M, else if it is less than the middle element, then set *Ub* to (*M* – 1). Else set *Lb* to (*M* + 1).
- In this way, each iteration halves the size of the array to be searched.

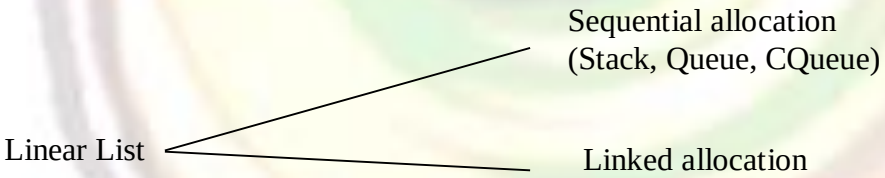
This is a powerful method. Given an array of 1023 elements, we can narrow the search to 511 elements in one comparison. After another comparison, and we're looking at only 255 elements. In fact, we can search the entire array in only 10 comparisons.

Ex: search for 87 in the array below:



algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		**efficiently support ordered operations?**
	search	**insert**	**search hit**	**insert**	
sequential search (unordered linked list)	N	N	N/2	N	no
binary search (ordered array)	Log N	2N	Log N	N	yes

Note: $\log_2 N$: (x such that $2^x = N$)



An array is a very useful data structure provided in programming languages. However, it has at least two limitations: ***memory*** or ***time***.

- It wastes memory by allocating an array that is large enough to store what you estimate to be maximum number of elements a list will ever hold.
- The insertion and deletion operations require shifting elements back and forth within the array.

So, arrays had certain disadvantages as data storage structures. In an unordered array, searching is slow, whereas in an ordered array, insertion is slow. In both kinds of arrays deletion is slow. Also, the size of an array can't be changed after it's created.

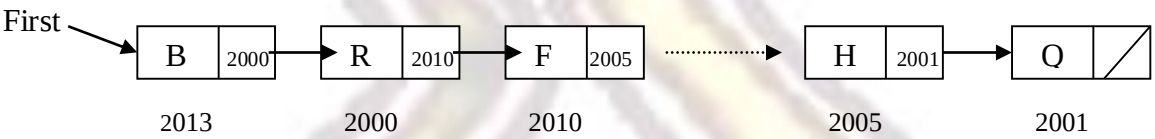
This limitation can be overcome by using *linked structure*. A linked structure is collection of nodes storing data and links to other nodes.

In this way, nodes can be located anywhere in memory, and passing from one node of the linked structure to another is accomplished by storing the addresses of other nodes in the linked structure.

Link list: is a data structure composed of nodes, each node hold some information and a pointer to another node in the list.

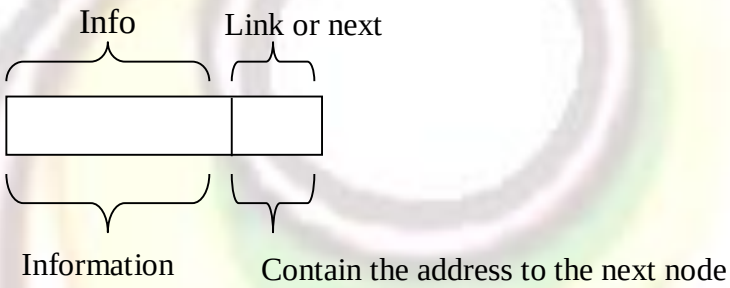
Linear List and Linked Allocation:-

Single Link Liner List (S.L.L.L): (one-way chain)



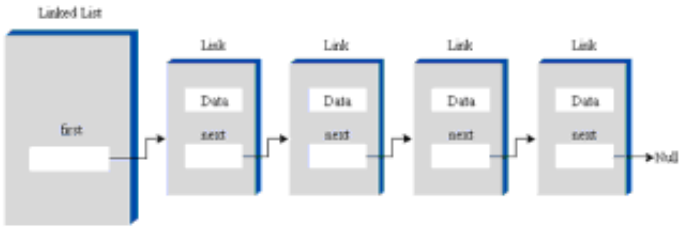
First: is a pointer var. to denote to the first node in the list.

Typical node



A node includes 2 data members: **info** and **next**. The info member is used to store information. The next member is used to link together nodes to form a linked list.

```
class Link
  {
  public  int     iData;          // data
  public  double dData;           // data
  public  Link    next;           // reference to next link
  }
```



- *note* that The ***next*** field of type Link is only a reference to another link, not an object.
- A *reference* is a number that refers to an object. It's the object's address in the computer's memory

To create an object you must always use ***new***:

```
Link  someLink  =  new Link( );
```

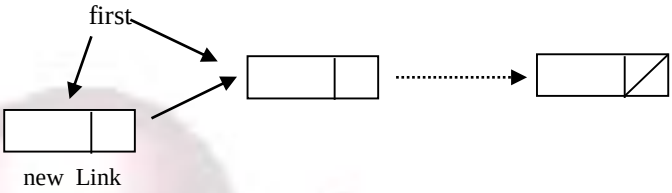
Ex:

```
class LinkList
{
        private Link first;          // reference  to first link on list
        // -------------------------------------------------------------
        public void LinkList( )     // constructor
        {
                first = null;               // no items on list yet
        }
        // -------------------------------------------------------------
        public boolean isEmpty( )    // true if list is empty
        {
                return (first==null);
        }
        // -------------------------------------------------------------
                ... // other methods go here
}
```

The insertFirst() Method

The insertFirst( ) method of LinkList inserts a new link at the beginning of the list by using the data passed as arguments. To insert the new link, we need only set the next field in the newly created link to point to the old first link, and then change first so it points to the newly created link.

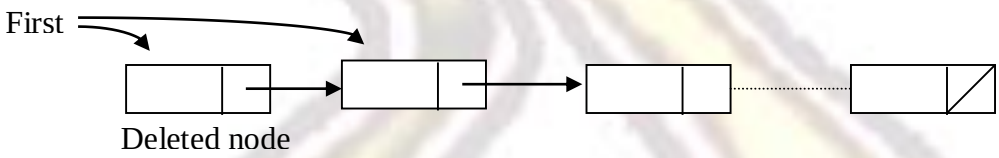
```
Link  newLink = new Link(id, dd);    // make new link
newLink.next = first;                 // newLink --> old first
first = newLink;                      // first --> newLink
```



The deleteFirst() Method

The deleteFirst() method is the reverse of insertFirst(). It disconnects the first link by rerouting first to point to the second link. This second link is found by looking at the next field in the first link.

```
                    // (assumes list not empty)
Link temp = first;      // save reference to link
first = first.next;     // delete it: first-->old next
return temp;            // return deleted link
```



The displayList() Method

To display the list, you start at first and follow the chain of references from link to link. A variable current points to (or technically refers to) each link in turn. It starts off pointing to first, which holds a reference to the first link. The statement

```
current = current.next;
```

changes current to point to the next link.

```
System.out.print ("List (first-->last): ");
Link current = first;           // start at beginning of list
while (current != null)                 // until end of list,
{       current.displayLink();          // print data
        current = current.next;         // move to next link
}
System.out.println("");
```

To delete a given key:

```
// delete link with given key     // (assumes non-empty list)

Link current = first;               // search for link
Link previous = first;
while (current.iData != key)
{
        if (current.next == null)
                return null;      // didn't find it
        else
        {
                previous = current;       // go to next link
                current = current.next;
        }
} // found it
if (current == first)               // if first link,
        first = first.next;         // change first
else // otherwise,
        previous.next = current.next;  // bypass it
```

Now try to
1) ***Add a node at the tail*** of the list,
2) ***Change the data*** for a specific node,
3) ***Display the list***.
4) ***Delete a node at the tail*** of the list.
5) ***Count the nodes*** in any list.
6) A ***Recursive function to count number of nodes*** in any list.

Of course you can insert a new link at the end of an ordinary single-ended list by iterating through the entire list until you reach the end, but this is inefficient.

Double-Ended Lists

A double-ended list is similar to an ordinary linked list, but it has one additional feature: a reference to the last link as well as to the first. Figure below shows what this looks like.

The reference to the last link permits you to insert a new link directly at the end of the list as well as at the beginning.

Linked-List Efficiency

Insertion and ***deletion*** at the beginning of a linked list are very fast. They involve changing only one or two references, which takes **O(1)** time.

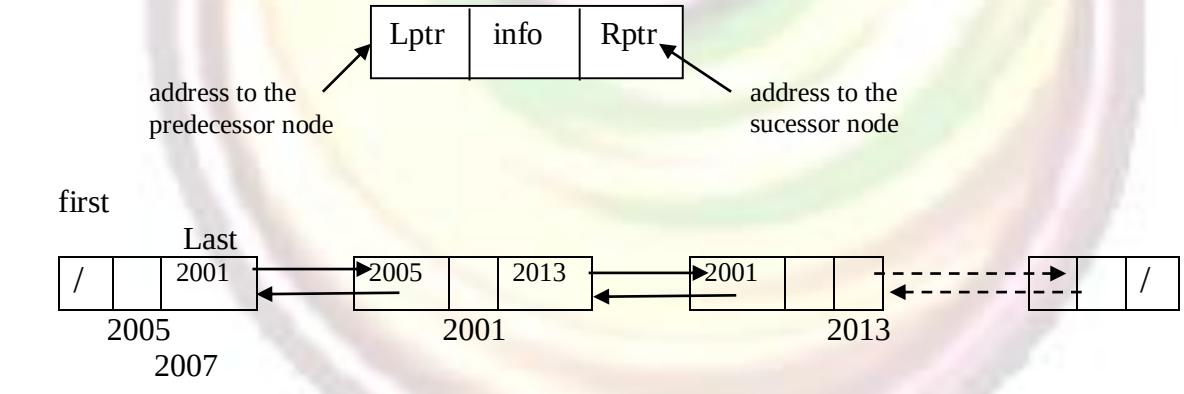
Finding, ***deleting***, or ***insertion next to a specific item*** requires searching through, on the average, half the items in the list. This requires **O(N)** comparisons. An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or deleted.

Another important advantage of linked lists over arrays is that the linked list uses exactly as much memory as it needs, and can expand to fill all of the available memory. The size of an array is fixed when it's created; this usually leads to inefficiency because the array is too large, or to running out of room because the array is too small. Vectors, which are expandable arrays, may solve this problem to some extent, but they usually expand in fixed-sized increments (such as doubling the size of the array whenever it's about to overflow). This is still not as efficient a use of memory as a linked list.

Double linked linear list (D.L.L.L.)

A potential problem with ordinary linked lists is that it's *difficult to traverse backward* along the list.

The doubly linked list provides this capability. It allows you to traverse backward as well as forward through the list. The secret is that each link has two references to other links instead of one. The first is to the next link, as in ordinary lists. The second is to the previous link. This is shown in Figure below



The beginning of the specification for the Link class in a doubly linked list looks like this:

```
class Link
{
        public double dData;  // data item
        public Link next;     // next link in list
        public link previous; // previous link in list
   ...
}
```

The **downside of doubly linked** lists is that every time you insert or delete a link you must deal with 4 links instead of 2; 2 attachments to the previous link and 2 attachments to the following one.

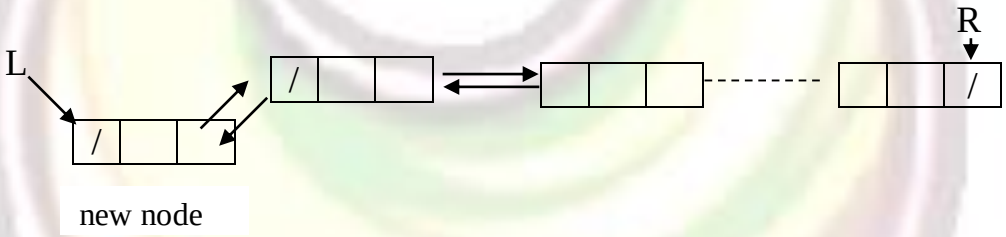
Traversal

Two display methods demonstrate traversal of a doubly linked list. The *displayForward()* method is the same as in ordinary linked lists. The *displayBackward()* method is similar, but starts at the last element in the list and proceeds toward the start of the list, going to each element's previous field:

```
Link current = last;              // start at end
while (current != null)           // until start of list,
   current = current.previous;    // move to previous link
```

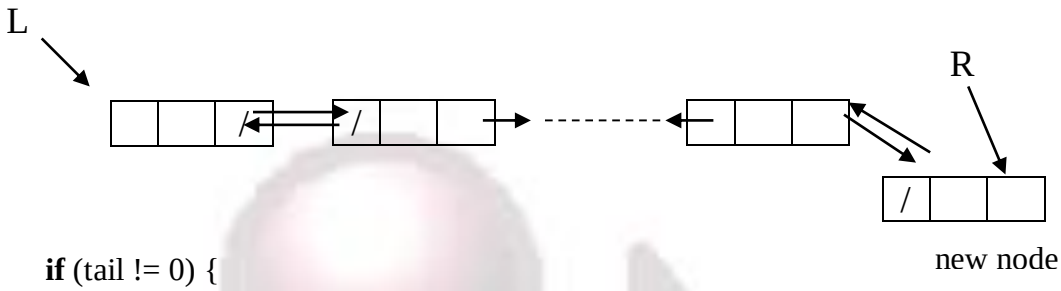
1. Insertion in D.L.L.L.

a) Insert at the left most node.



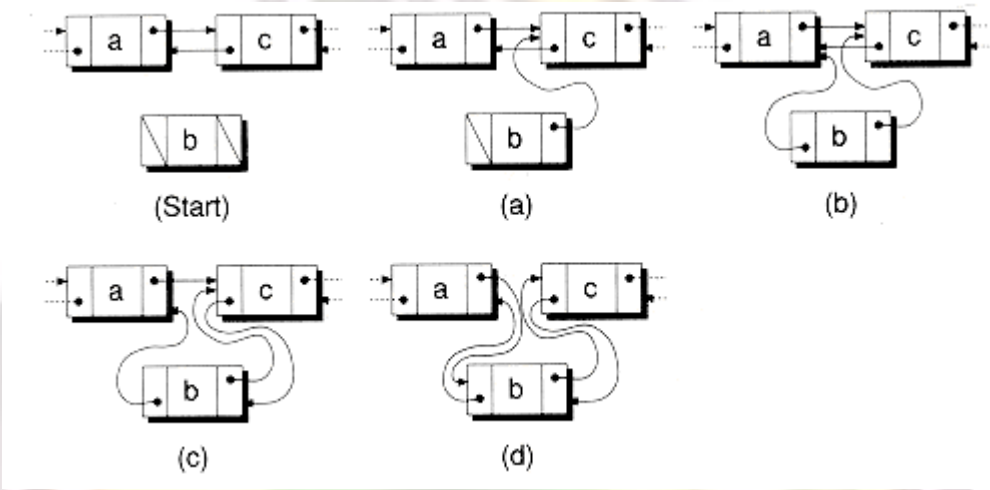

```
if (first = 0) {
        first = last = newLink;
else
        first.prev = newLink;          // newlink <-- old first
        newLink.next = first;          // newLink --> old first
        first = newLink;               // first --> newLink
```

b) Insert at the right most node.




```
if (tail != 0) {
        newlink.prev = last;
        last.next = newlink ;
        last = newlink;
}
else
        first = last = newLink;
```

c) Insert in the middle of D.L.L.L. // insert after a node



3. Deletion in D.L.L.L.

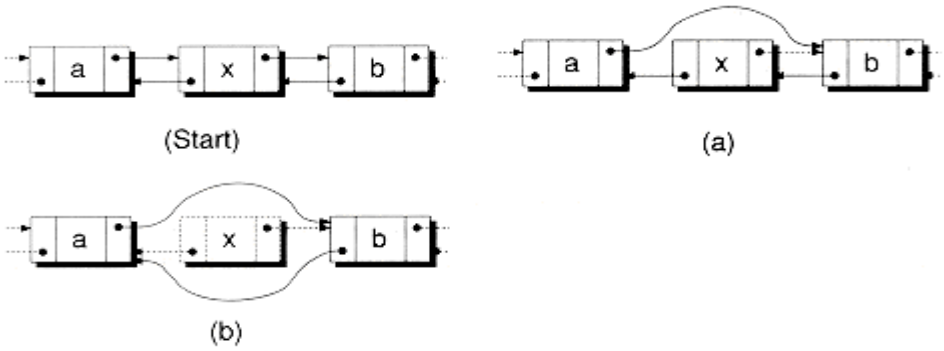
a) Delete from the right most node.

```
if (first= last) {  // if only one node in the list;
        first = last = 0;
}
else {
        last = last.prev;
        last.next= 0;
}
```

b) Delete from the left most node.

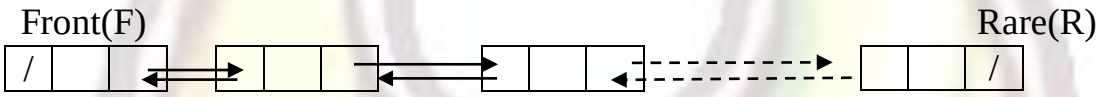
```
first = first.next;
first.prev = 0
```

c) Delete from the Middle.




```
current. prev.next = current.next;
current.next.prev = current.prev;
```

D.L.L.L. as a queue:-



1. Create ⟵ same as create to any D.L.L.L.

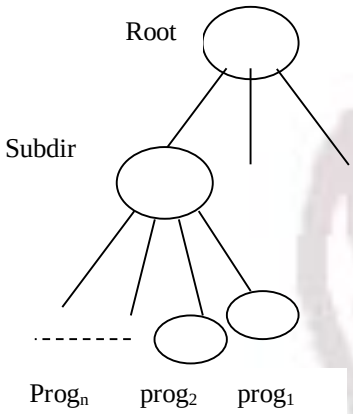
2. Insertion (insert at the right most location)

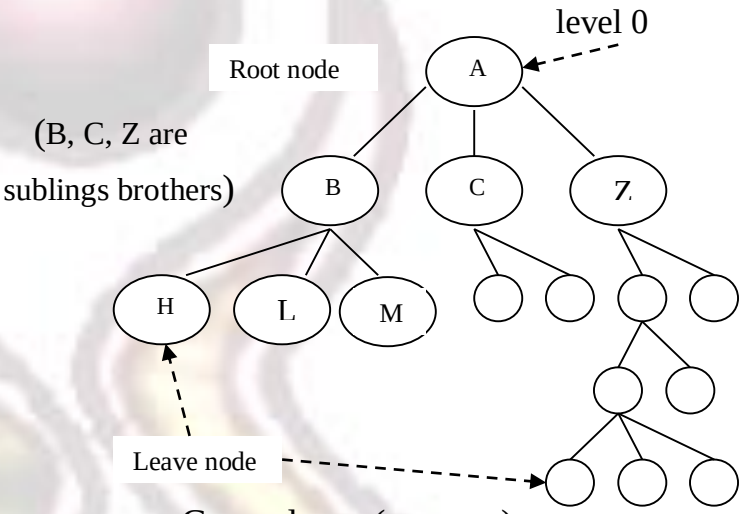
3. Deletion (delete the left most node)

Non linear Data Structure

1. Tree
2. Graph
3. Network

Tree (Branching Structure)



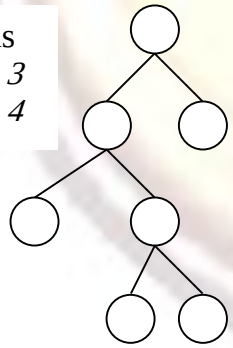


General tree (concept)

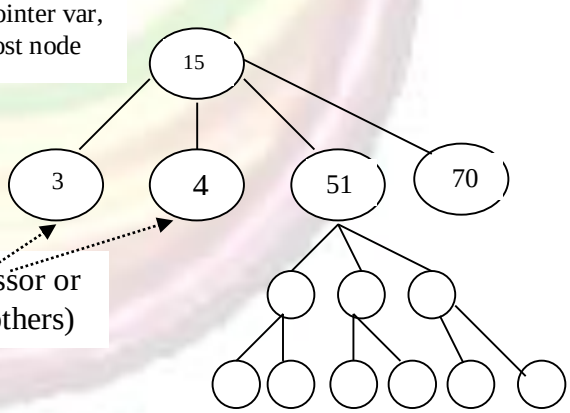
notes

- (H, L, M are the **children** of B)
- A-B-M is path (**descending**) /* descending from top to down i.e. from root to leaves */
- **Climbing**. Is a bottom up approach, e.g., a program combines subprograms in one unit.
- Two nodes are **siblings** if they have the same parent.
- **Ancestor.** A node's parent is its first ancestor, the parent of the parent is the next ancestor, and so on. The root is an ancestor of each other node.
- (70) is a leave node because it has no sons, whatever its level, it has no successor

4 levels
0, 1, 2, 3
1, 2, 3, 4



Root : is a pointer var,
to the top most node

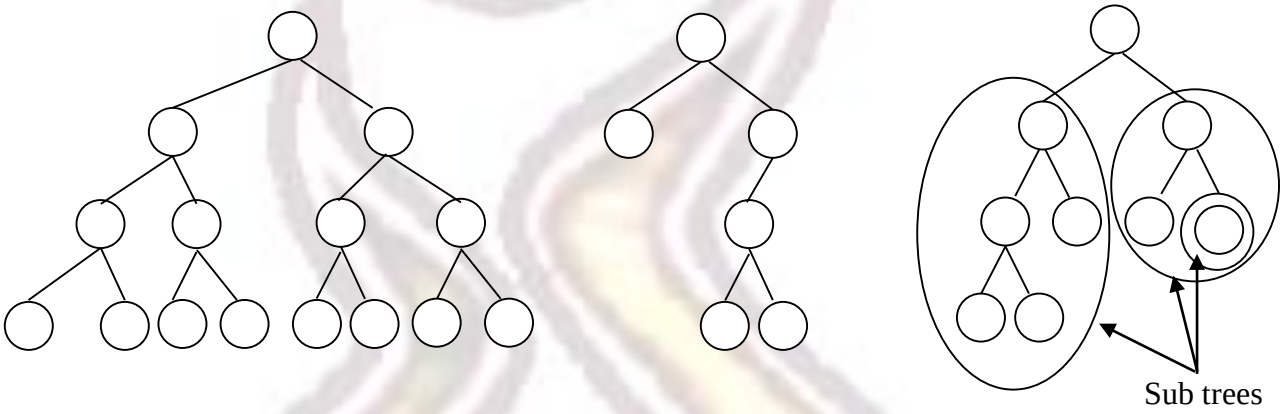


Same predecessor or
(twins) or (brothers)

Binary tree

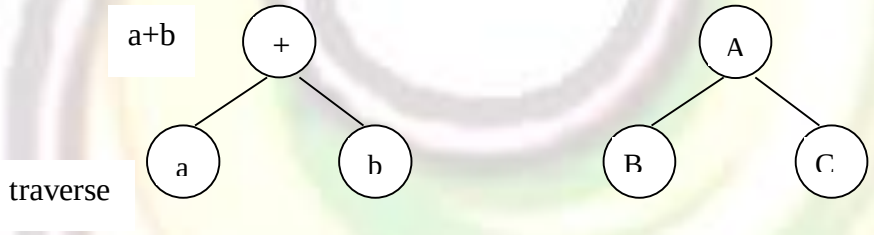
A **binary tree** is a finite set of "nodes". The set might be empty (no nodes, which is called **empty tree**). But if the set is not empty, it follows these rules:

1. There is one special node called the root.
2. Each node may be associated with up to two other different nodes, called its **left child** and its **right child**. If a node *c* is the child of another node *p*, then we say that "*p* is *c*'s **parent**".
3. Each node, except the root, has exactly one parent; the root has no parent.



Application of binary tree:-

To represent any arithmetic expression



traverse

Tree traversals

Means visit all the nodes in a tree only once.

There are 3 common ways for traversals of binary tree: in-order traversal, pre-order traversal, and post-order traversal.

1- Inorder traversal
   a. left subtree
   b. root
   c. right subtree

B A C
a + b (infix expression)

2- Preorder traversal

a. Root
b. left subtree
c. right subtree

A B C
\+ a b (prefix expression)

3- Postorder traversal
   a. left subtree
   b. right subtree
   c. root

B C A
a b + (suffix expression)

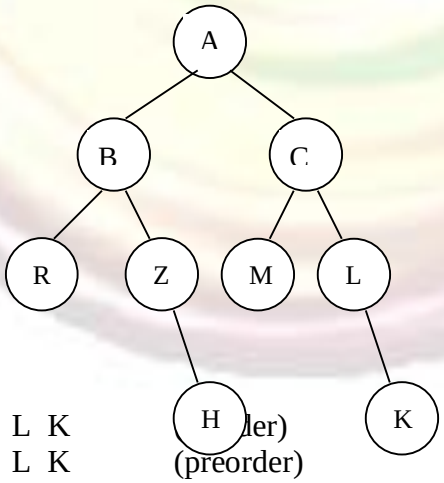
Note that there are another methods to traverse general tree, a tree if we convert left by right.

1- Convert inorder
   a. right subtree
   b. root
   c. left subtree

2- Convert preorder
   a. Root
   b. right subtree
   c. left subtree

3- Convert postorder
   a. right subtree
   b. left subtree
   c. root

Example:

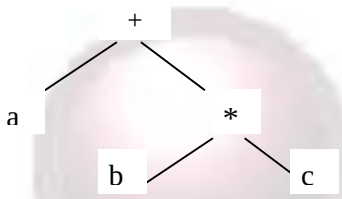


R B Z H A M C L K (inorder)
A B R Z H C M L K (preorder)

R H Z B M K L C A (postorder)

Example2:

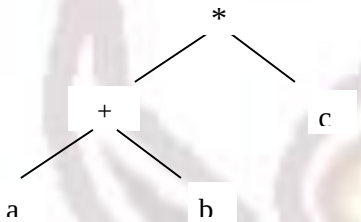
a + b * c



1. a + b * c
2. a b c * +
3. + a * b c

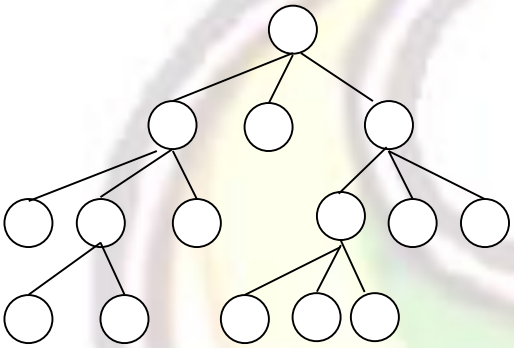
Example3:

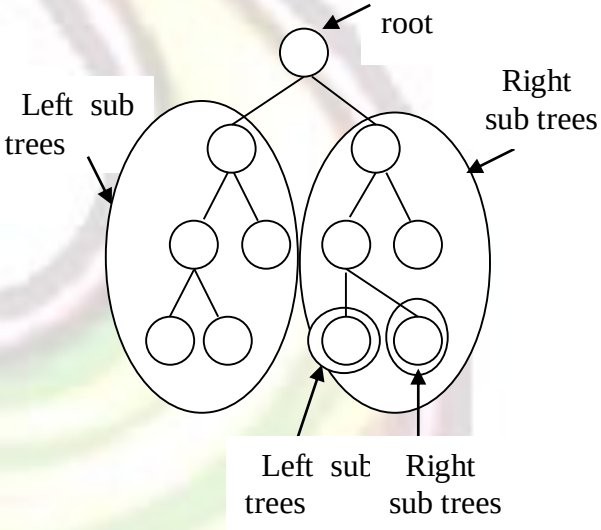
(a + b) * c



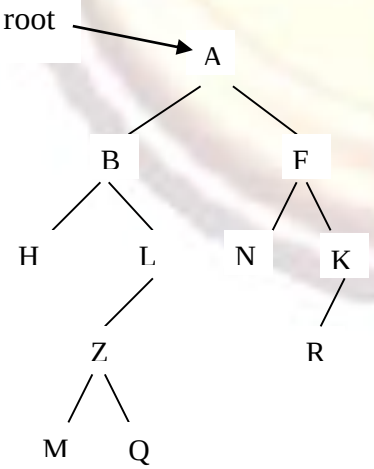
1. * + a b c
2. a b + c *

Non Linear Data Structure (Tree):





General Tree



Traversal

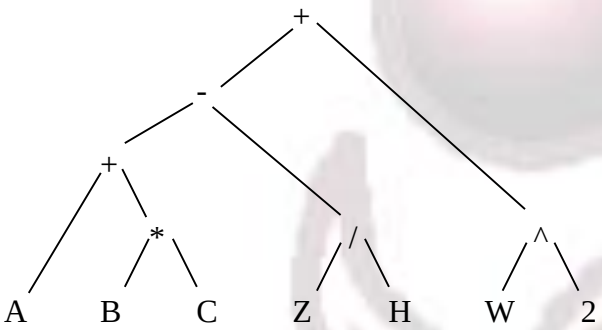
1. Inorder traversal
   H B M Z Q L A N F R K
2. Preorder traversal
   A B H L Z M Q F N K R
3. Postorder traversal
   H M Q Z L B N R K F A

Application of binary tree:-

1. Using BT to convert any infix exp. into suffix and prefix form, such trees called expression trees.

Example

A + B * C – Z / H + W ^ 2



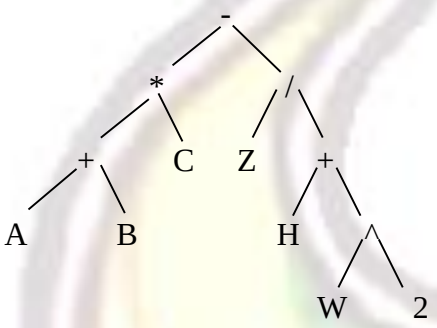
Expression Tree

Traversal

1. Inorder traversal
   A + B * C - Z / H + W ^ 2
2. Preorder traversal
   + - + A * B C / Z H ^ W 2
3. Postorder traversal
   A B C * + Z H / - W 2 ^ +

Example2

(A + B) * C – (Z / (H + W ^ 2))



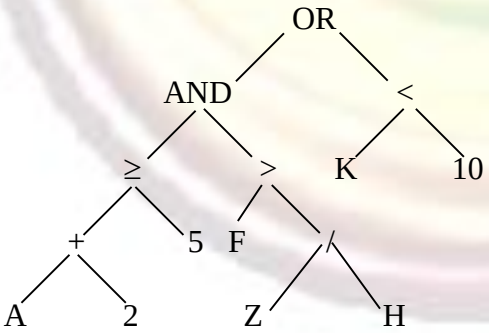
Traversal

1. Prefix form (preorder)
   - * + A B C / Z + H ^ W 2
2. Suffix form (postorder)
   A B + C * Z H W 2 ^ + / -

How to draw a mathematical expression tree

- Every subtree has a root (operation) and its children (operands)

((A + 2 ≥ 5) AND (F > Z / H)) OR (K < 10)

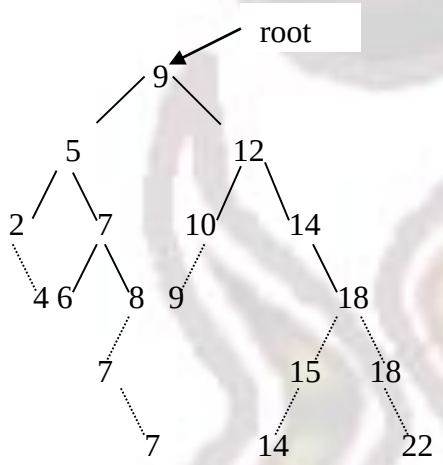


4. Infix (inorder)
   A + 2 ≥ 5 AND F > Z / H OR K < 10
5. Preorder
   OR AND ≥ + A 2 5 > F / Z H < K 10
6. Postorder
   A 2 + 5 ≥ F Z H / > AND K 10 < OR

2. Using BT to find all the duplicated in any given values by built a **Binary Search Tree**.

Binary Search Tree (BST)

9, 12, 5, 7, 2, 14, 18, 10, 8, 6

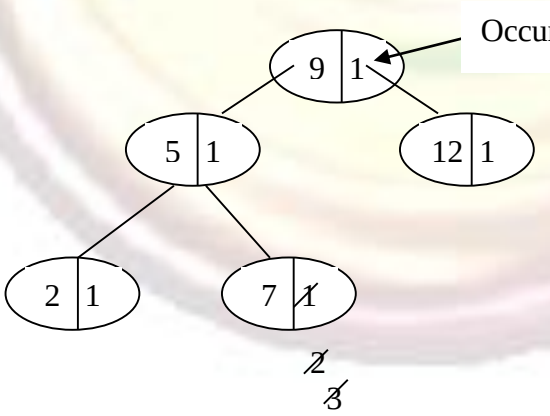


insert (7)
insert (7)
insert (4)
insert (15)
insert (9)
insert (14)
insert (18)
insert (22)

3. Obtain sorted data by building a BST then traverse the tree using inorder traversal.

notes

- if we want to insert duplicated data to the tree, then the duplicated means greater than operation.
- Another way to insert duplicate data, by put an integer number denoting the times any data are duplicated.

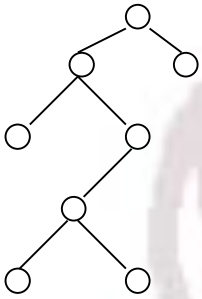


Representation of Binary Tree in a Computer Memory:

1. Sequential allocation (store a dimension greater than or equal to the tree size)
2. Linked allocation

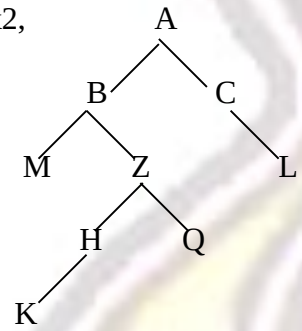
- In sequential allocation, if we have 5 levels, the no. of locations (array size) = 2^5-1, but this will occupy spaces without using it.

Ex.,



If we have 5 levels,
Then there is 31 location
and 8 locations are used only

Ex2,



A	B	C	M	Z		L			H	Q			K				
1	2	3	4	5		7			10	11			20				31

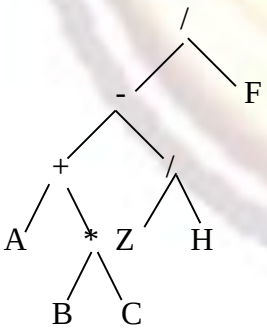
- If the parents in location P, then
  left son in location 2P and
  right son in location 2P+1

- If **n** is a no. of level then the array size = $\mathbf{2^n-1}$

  - 4-0+1 → 5 levels if root in level zero
  - 5-1+1 → 5 levels if root in level one.

Ex3: (A + B * C – Z / H) / F

/	-	F	+	/			A	*	Z	H	…….	B	C		
1	2	3	4	5			8	9	10	11		18	19		31



- In sequential allocation, the returning to the root is easy.
- While in linked allocation, it is hard to return to the root, also there is increasing in reserving space since there is a need to reserve the left and right pointers.

Rebalancing

The tree is called balanced if different subtrees below a node are guaranteed to have nearby the same height.

This will reduce the reserved array size, the searching time since it reduces the level for the above example from 5 to level 3.

Create a Binary Search Tree

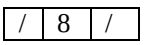
(* *function CreateTree* *)

```
void CreateTree(LRp     *Ptr)
{           LRp     Root;
            CreateNode(&Root);
            *Ptr = Root;
}

void CreateNode(LRp    *Ptr)
{
  po=(Node *)malloc(sizeof(Node));
  if(po != NULL)
     {     po→info = x;    po→Lptr = NULL;
  po→Rptr = NULL;      }
  *Ptr=po;
}
```

Actual parameter

Root=2015

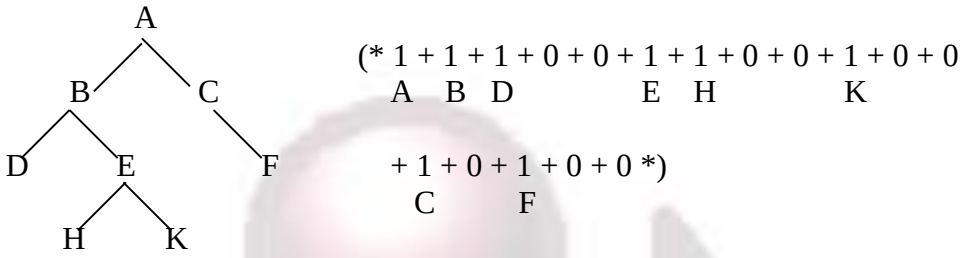


Algorithm Rinsert(Root, Node)

(* *This algorithm to insert a new node to any BST its Root given by a pointer variable Root* *)

```
if (current→info < (*Root) →info)
            if ((*Root) →Lptr == NULL)
                        (*Root) →Lptr = current;
            else
                        Rinsert(&(*Root) →Lptr, &current);
else
            if (current→info > (*Root)→info)     // insert right subtree
                        if ((*Root)→Rptr == NULL)
                                    (*Root)→Rptr = current;
                        else
                                    Rinsert(&(*Root)→Rptr, &current);
            else
            {           cout<< "the no. is found";
                        (*Root)→occ=(*Root)→occ+1;
            }
```

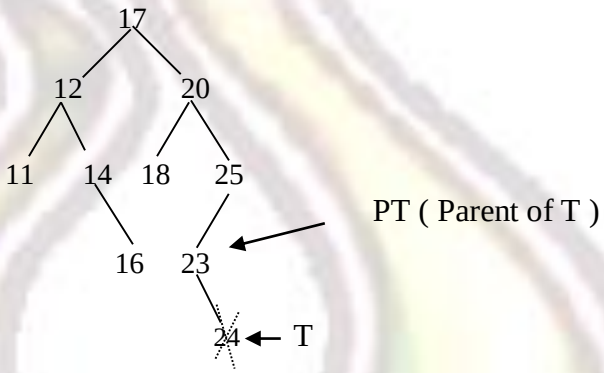
```
//(* This function to count no. of nodes in any given BT its root R  *)
    int count(LRp R)
    {
        if (R == NULL)      co=0;
        else                co = 1 + count(R->Lptr) + count(R->Rptr);
        return (co);
    }
```



Deletion in a Binary Tree

There are 3 cases to consider the deletion operation

1. If the deleted node has <u>no sons</u> (it is a leaf)



The deleted node called T

<u>right</u>

PT→Rptr = NULL
free(T) (* Return T to the avail *)

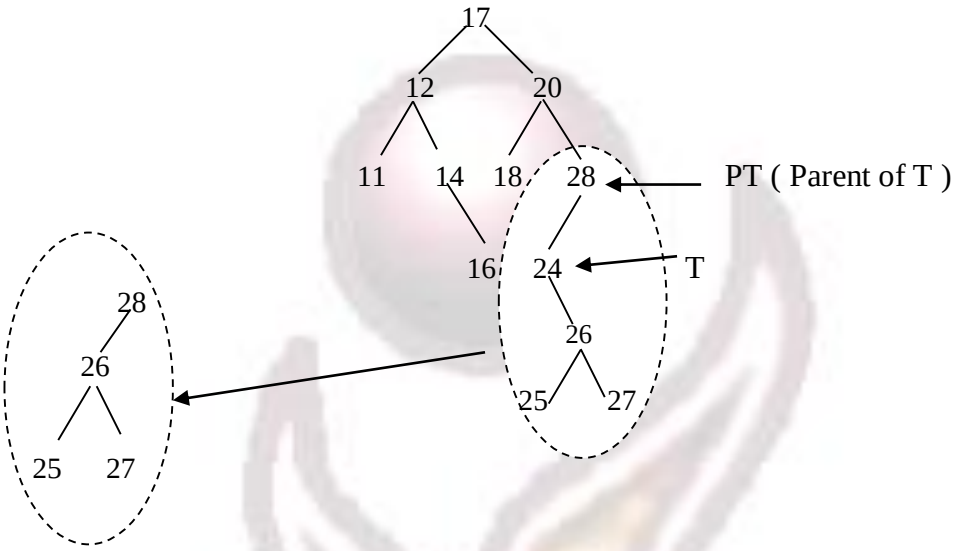
<u>left</u>

PT→Lptr = NULL
free(T) (* Return T to the avail *)

2. If the deleted node has only <u>one subtree</u> left or right then its son can be moved up to take its place.

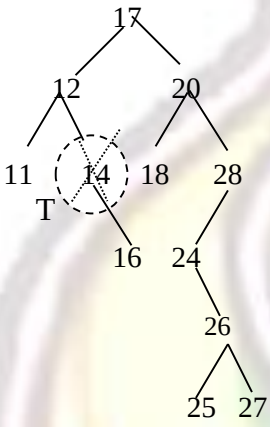
2.a

PT→Lptr = T→Rptr



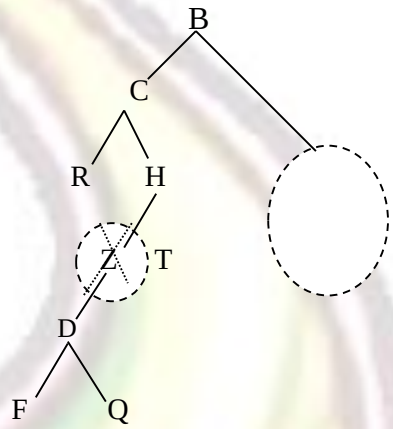
2.b

PT→Rptr = T→Rptr



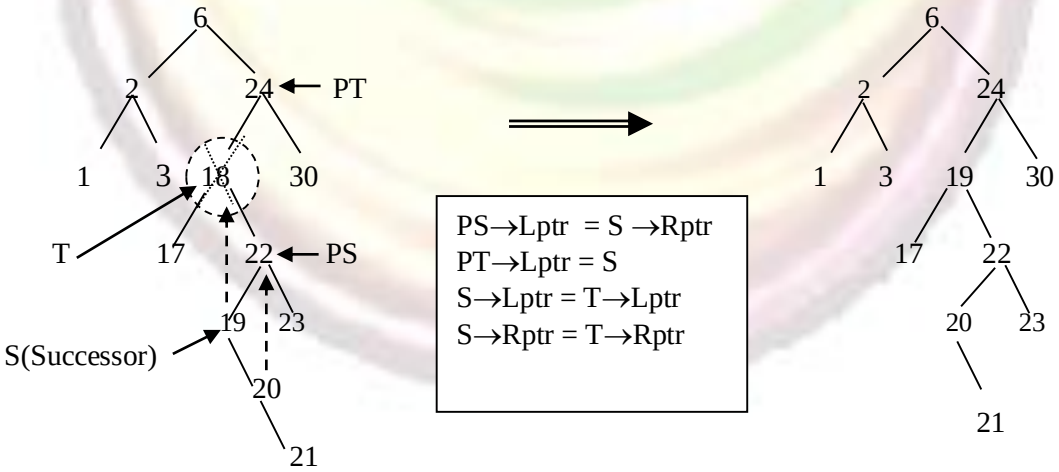
2.c

PT→Lptr = T→Lptr

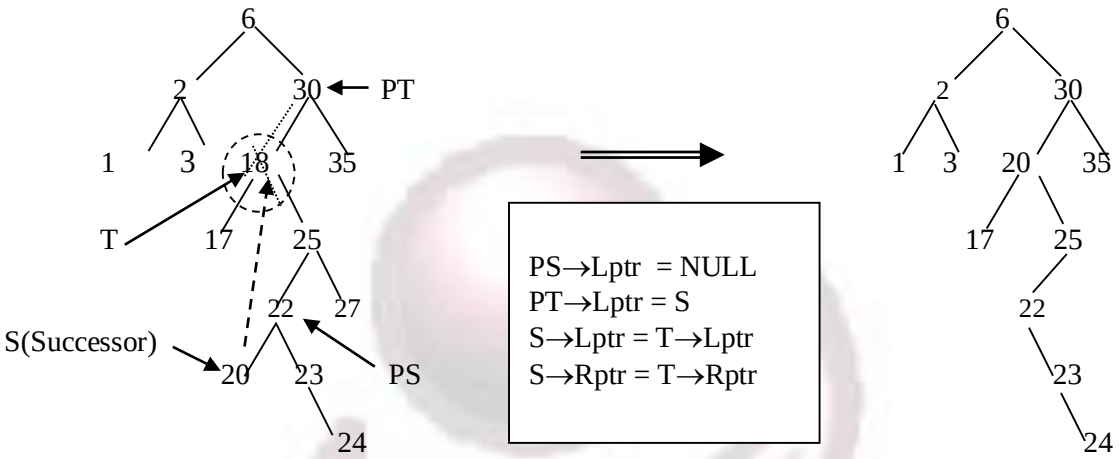


3. If the deleted node has 2 subtrees, then its inorder successor must take its place.

3.a

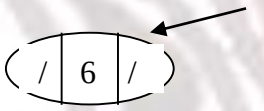


3.b

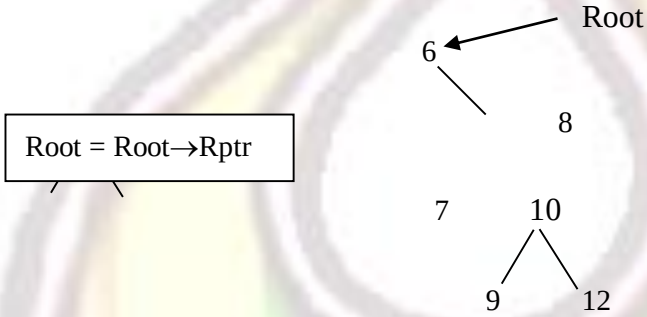


4. If the deleted node is the root

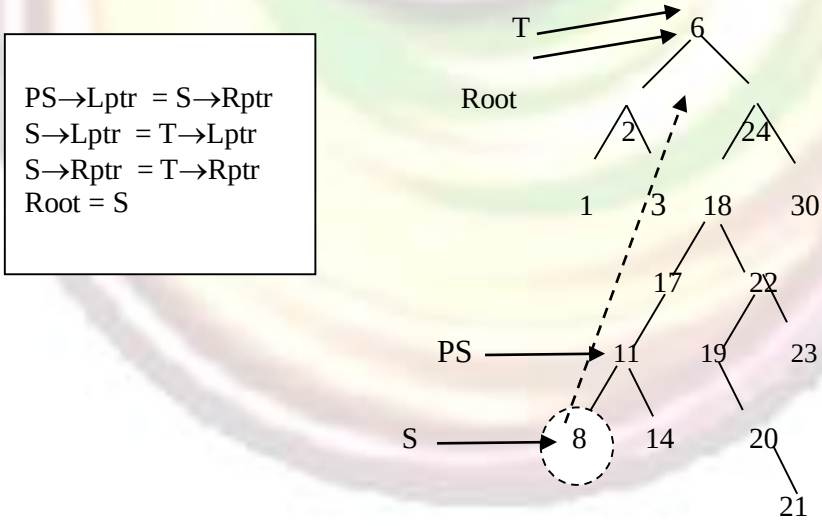
4.a



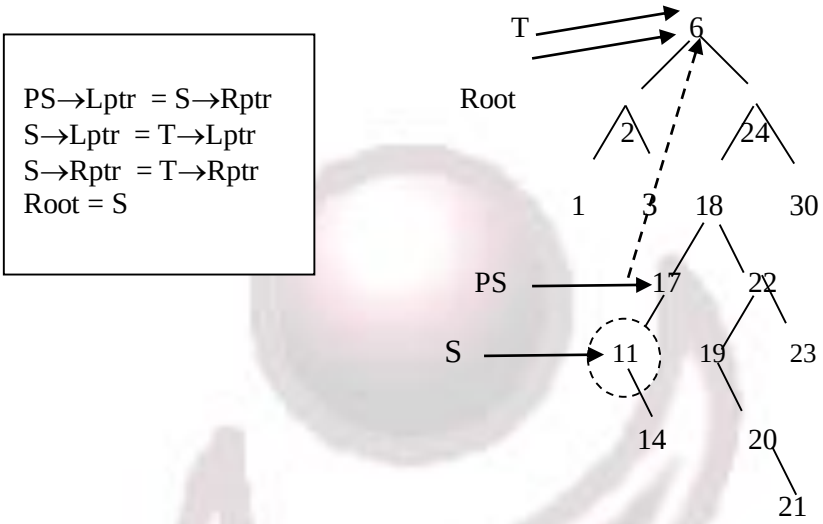
4.b



4.c



4.d



Height of Tree

Dynamic tree

```
int height (LRp  Root)
{
    if  (Root ==NULL)
          Height=0;
    else
          height = 1 + Max( height ( Root→Lptr ), height ( Root→Rptr));
}

int Max (int A, int B)
{
   if A>B
          return(A);
   else
          return (B);
}
```

Static Tree

If the tree as an array, then we takes the last occupied position; we start from the end of the array backward to the first element its value not NULL.

In dynamic tree, try to printout the path as well as its length.

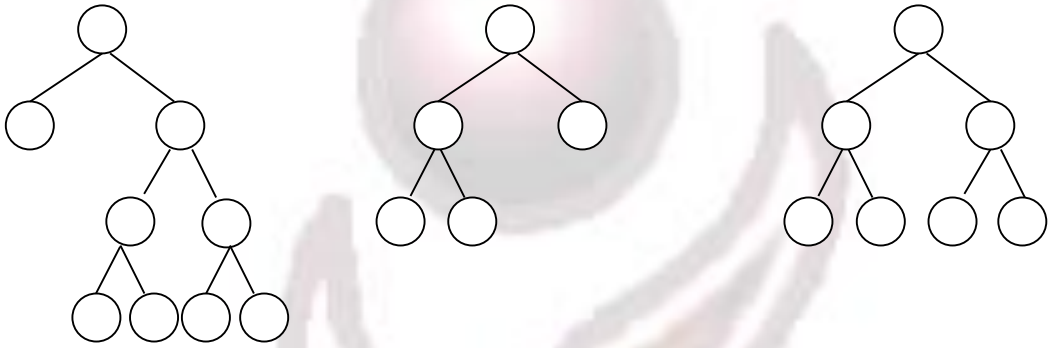
Tree Sort

Full binary trees

A binary tree is called full when every leaf has the same depth and every nonleaf has two children.

Complete binary tree

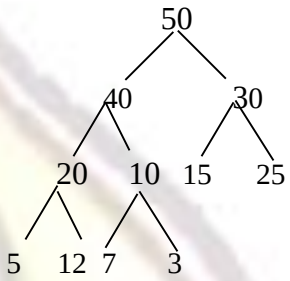
A tree with full level before starting the next level



Not complete complete full and complete

Max heap structure:

- When the father is greater than his sons.
- The greatest value in the root.
- Complete binary tree.
- Every path is in descending order.



Tree sort

Two sorting methods which are based on a tree representation of a given table of data.

1. The straight forward binary search tree (built BST then the
   inorder traversal ⇒ ascending order
   convert inorder ⇒ descending).
2. The heap sort which involve a complete binary tree in much more complex structure called Heap structure.

A Heap is a binary tree where entries of the nodes can be compared with the less-than operator of a strict weak ordering; in addition, these two rules are followed:

1. The entry contained by the node is never less than the entries of the node's children.

2. The tree is a complete binary tree, so that every level except the deepest must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.

structure is a complete binary tree with some of right most leaves removed.

A General Heap represent a table of data satisfied the following properties:

1. The Max value is in the Root (if Max Heap).
2. Each parent is greater than its children.
3. The path from the Root to any leaf node is sorted path.

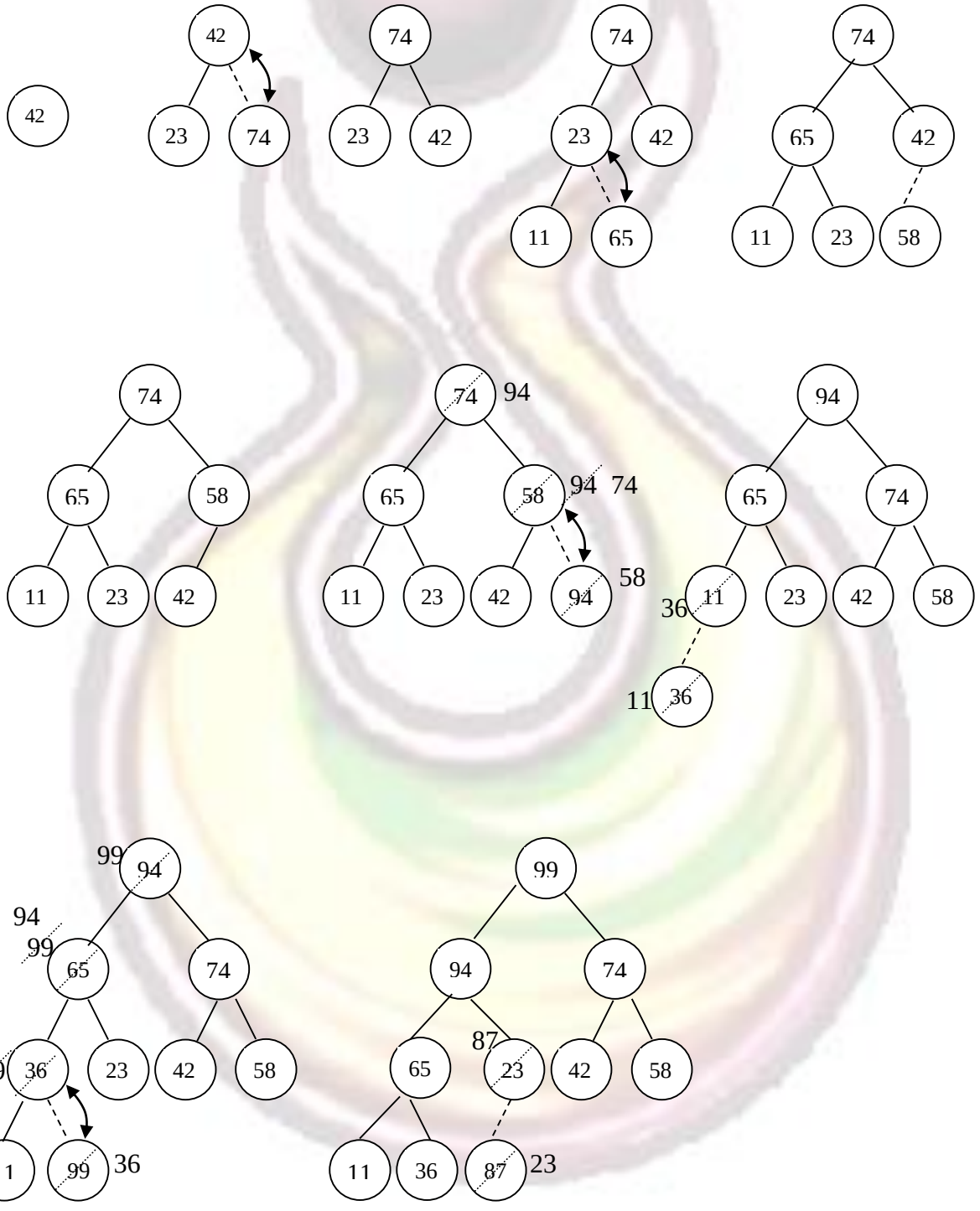
The Heap sort algorithm includes two steps:

1. Construct the Heap.
2. Sort the data.

Ex:

Use the following data to construct a Heap structure, and then use a Heap sort algorithm.

42, 23, 74, 11, 65, 58, 94, 36, 99, 87



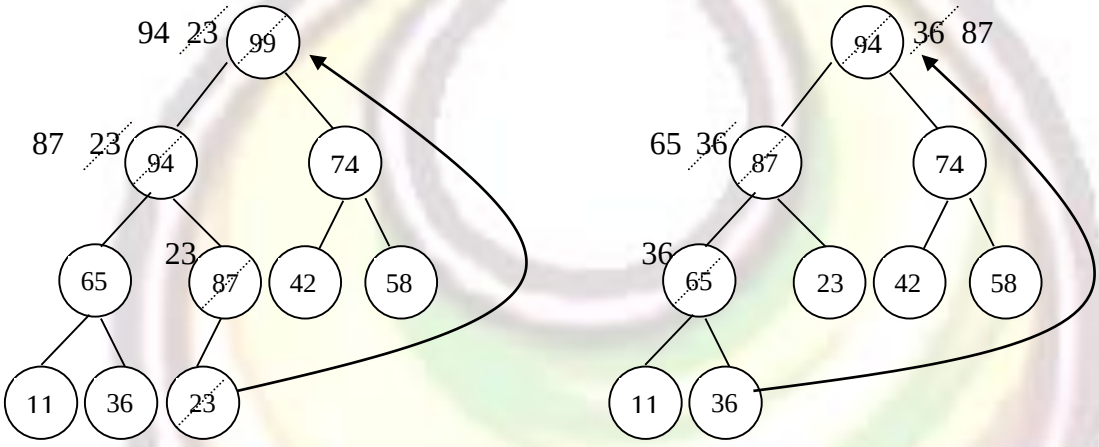
99	94	74	65	87	42	58	11	36	23

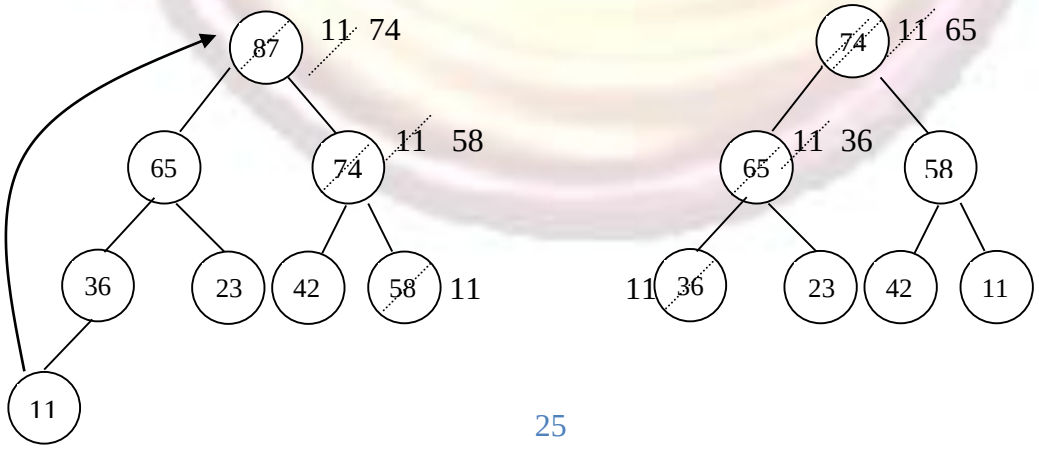
Removing an entry from a Heap

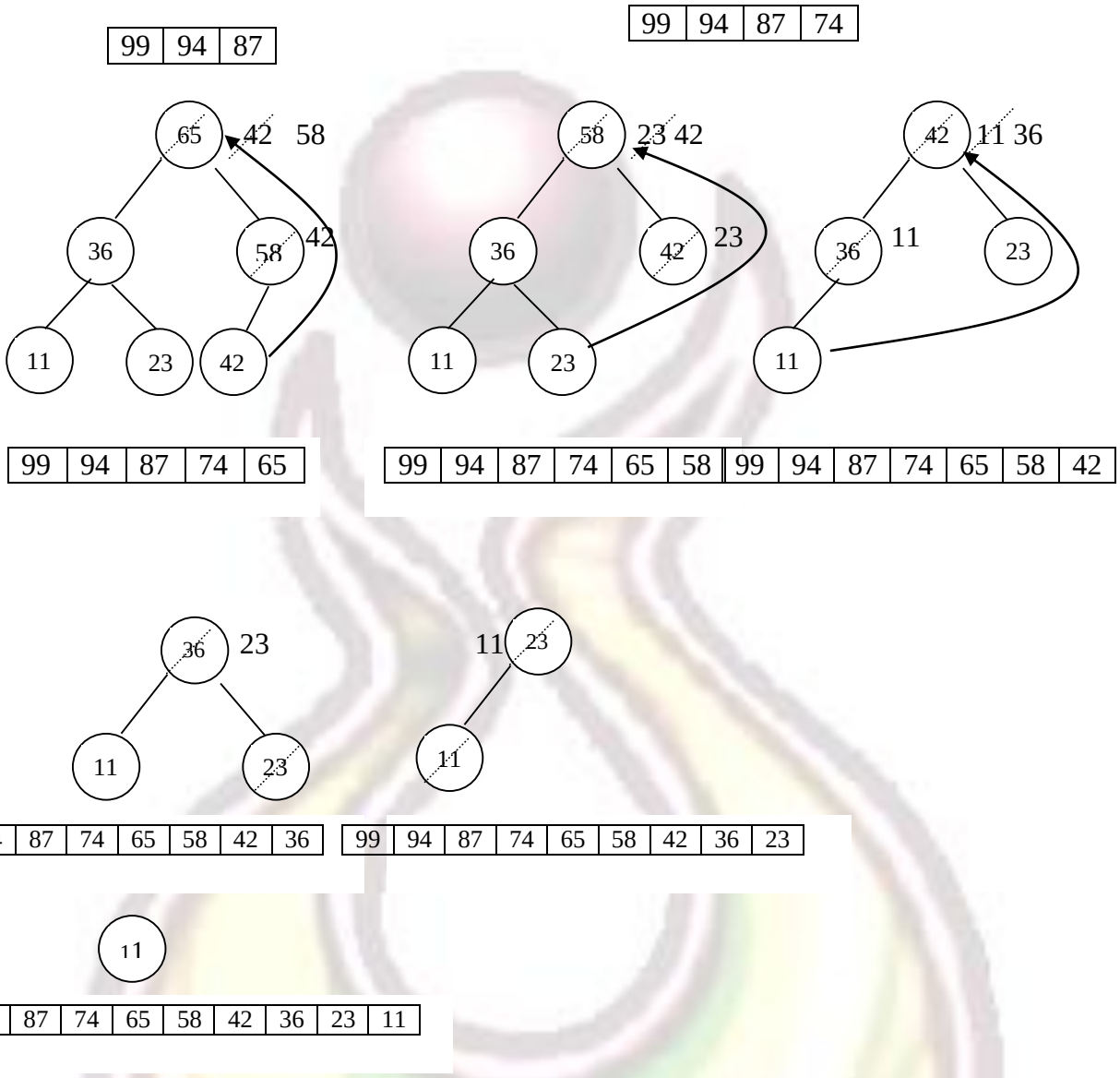
When an entry is removed from a priority queue, we must always remove the entry with the highest priority – the entry that stands "on top of the heap".

If the root entry is the only entry in the heap, then there is no more work to do except to decrement the member variable that is keeping track of the size of the heap.

Otherwise, the last entry in the last level has been moved to the root; the structure is now complete tree, but it is not heap because the root is less than its children. To fix this, we can swap the root with its larger child. The structure is not yet a heap, so again swap the out-of-place node with its larger child, and so on until we reach the leaf, then we will stop, and the structure is a heap.





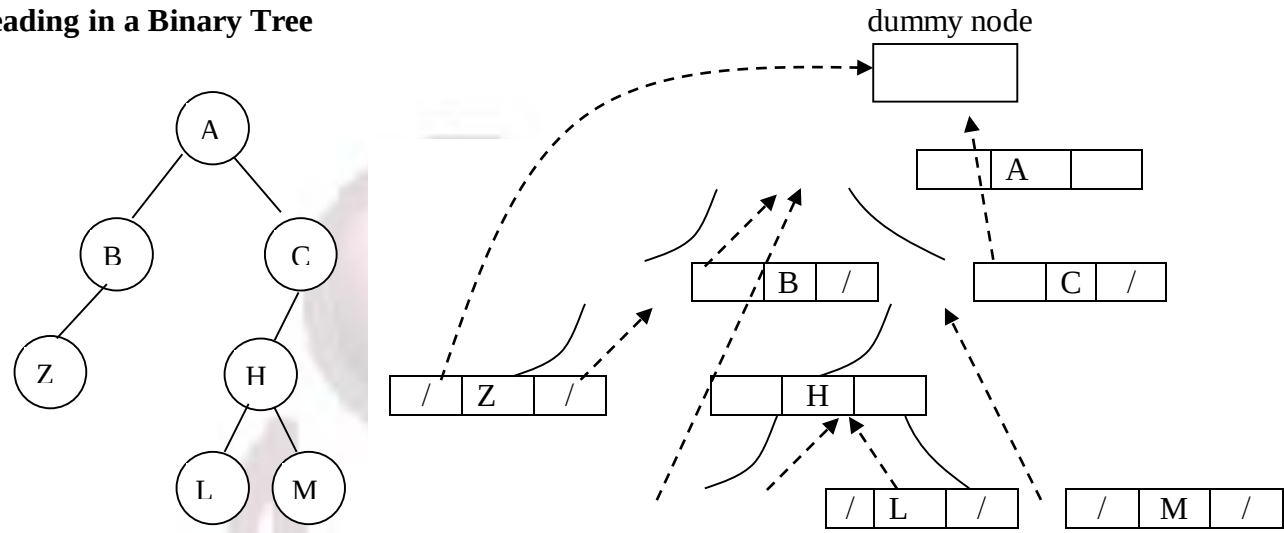


H.W: Consider the last example to construct Min Heap and sort a given data in ascending order.

Note:

- Its also possible to define ascending Heap (Min Heap) as almost a complete binary tree such that the content of each node is greater than or equal to its father.
- In Min Heap, the root contains the smallest value and any path from the root to a leaf node is on ascending order.

Threading in a Binary Tree



8 NIL pointer
7 node

In Postorder : Z B L M H C A

In Inorder traversal: Z B A N H M C

- **Thread**: is an empty pointer which became a pointer to the successor or predecessor in an inorder traverse.
- It is difficult to recognize between the original pointers or thread pointers, so there is a waste of space because there must be 2 additional fields to indicate if the node is normal or thread.

If we look carefully at the linked representation of any binary tree, we note that there is **(n+1)** NIL links for a binary tree contains (**n**) nodes. This waste of storage can be used in the reformulation of the previous representation of binary tree.

The empty link will be replaced by threads: which is a pointer to the successor or predecessor nodes in a tree.

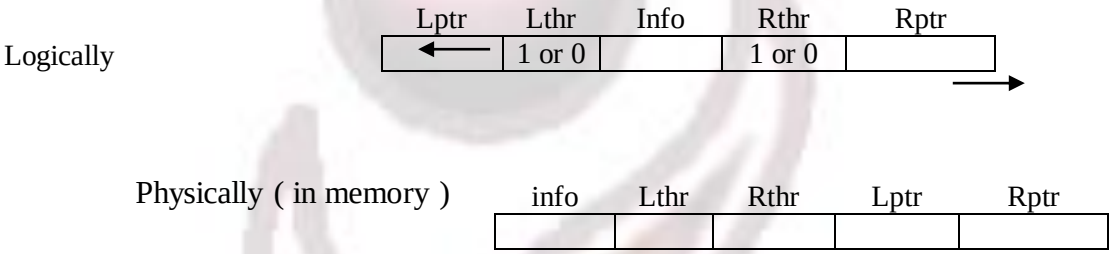
If the right pointer is nil, then it will be replaced by a pointer to the successor node in any ordered used,

and if the left pointer is NIL, it will be replaced by a pointer to the predecessor node in any ordered used.

In the memory representation, we must be able to distinguish between threads and original pointers. This is done by adding a Boolean fields to the record, so the node defined as:

```
struct Node
{
```

```
        info        any type
        boolean     Lthread, Rthread
        struct  Node    *Lptr, *Rptr;
};
typedef  struct  Node    *Lp;
```



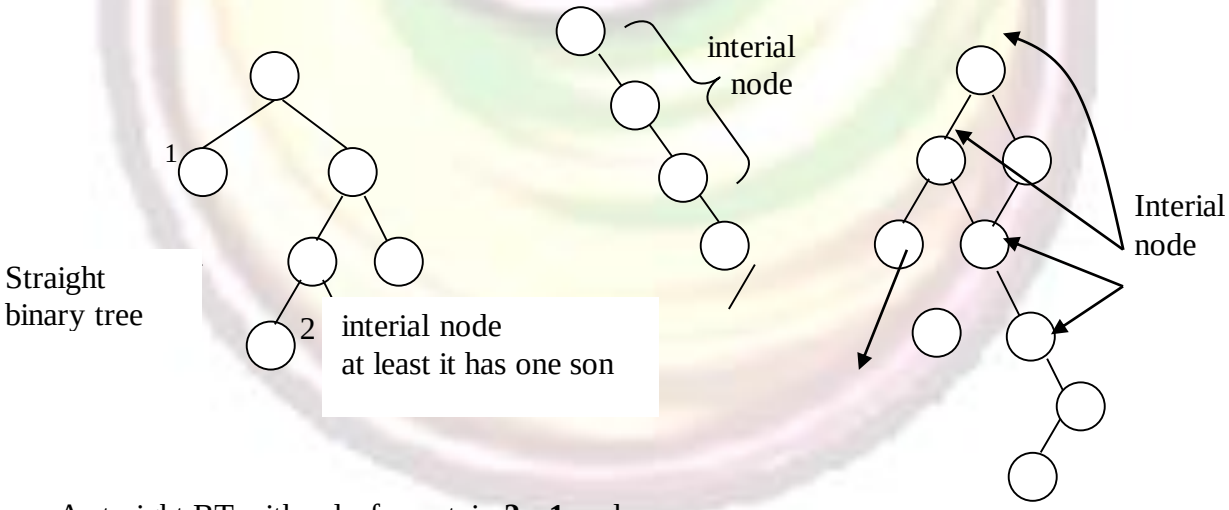
H.W: Given the following postorder and inorder traversal of a binary tree. Draw the tree.

Postorder: A B C D E F I K J G H
Inorder: C B A E D F H I G K J

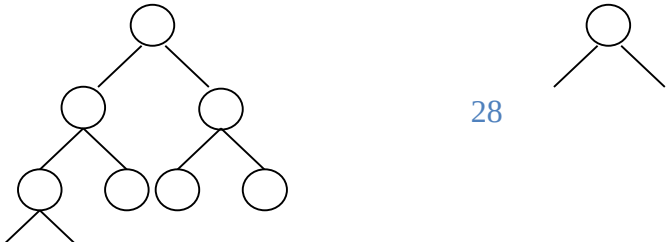
Straight Binary Tree

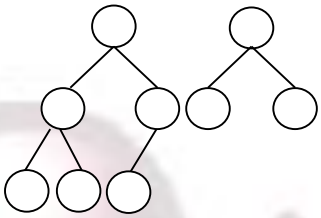
If every non leaf node in a binary tree has non empty left and right subtree, the tree is termed as straightly binary tree.

For example:



- A straight BT with **n** leafs contain **2n-1** nodes.



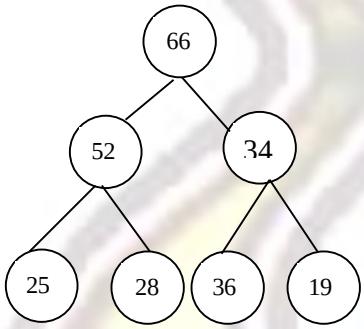


Complete and straight

Complete and not straight

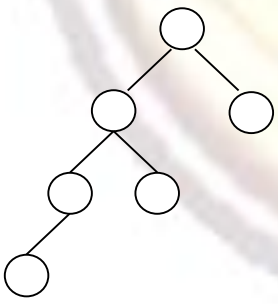
Different types of BT:

1. Binary Search Tree.
2. Expression Tree
3. Complete Binary Tree [Static Representation].
4. Heap Structure [Heap BT]
5. Thread BT.
6. Straightly Binary Tree.



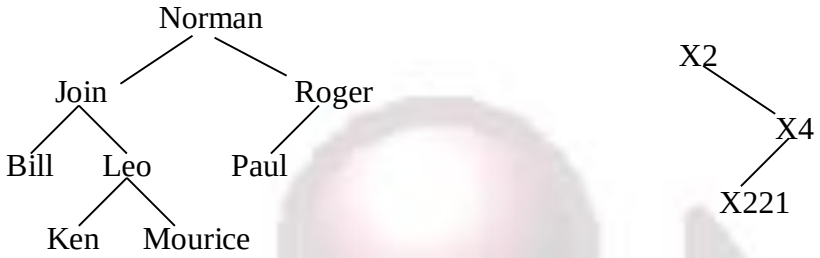
Full and complete
but not heap

Logical deletion: we put a tag field pointing if this node is deleted or not.
Physical deletion: the normal deletion by pointers.
Brunch: the connection between 2 nodes.
Spling: It is the node that came from the same father.



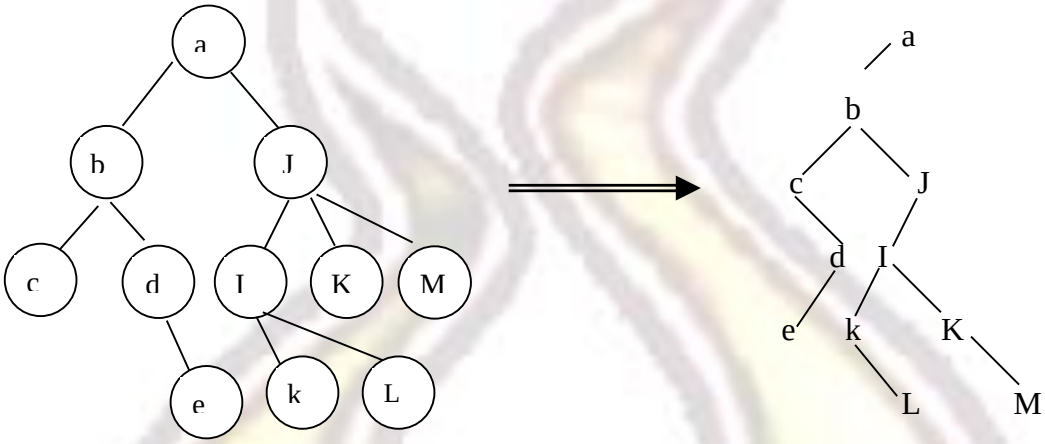
non heap
non-complete
non-full
non-straightly

Ex: Use BST to sort the following list of names in lexicographic order.
Norman, Roger, John, Bill, Leo, Paul, Ken and Mourice



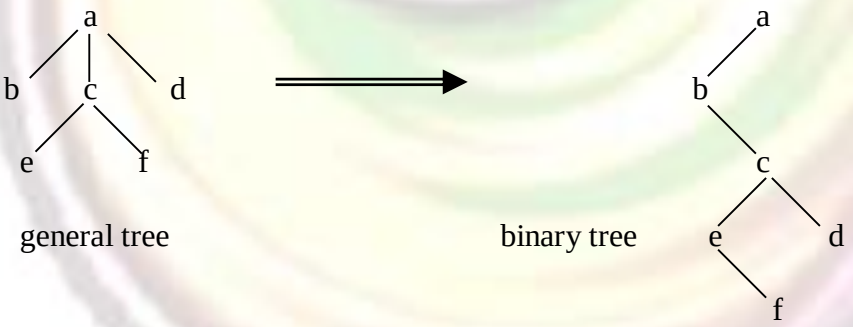
If there are small and capital letters, note that the small letters is greater than the capital letters.

Converting into Binary Tree



The son on the left
The brother on the right

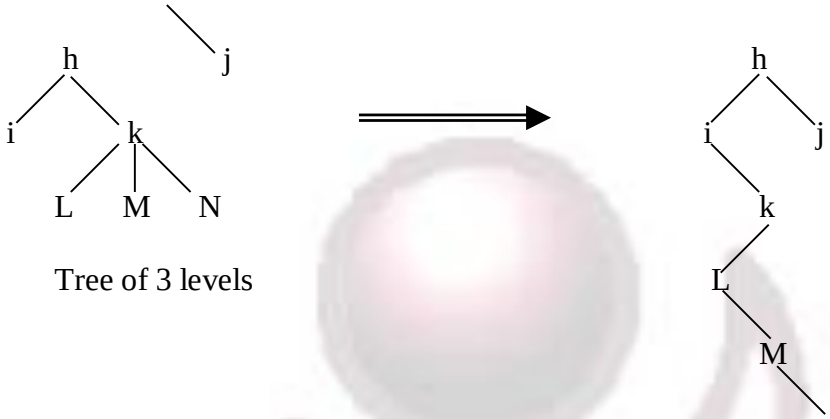
Ex1:



- No. of Links are equal.
- No of levels increased.

- Degree of tree increased by increasing the number of sons in that tree.

Ex2:



Tree of 3 levels