Contents

[1. The linked list 4](#_Toc86592542)

[The linked allocation has the following draw backs: 4](#_Toc86592543)

[1. Singly Linked List 4](#_Toc86592544)

[Operations on Singly linked list: 4](#_Toc86592545)

[2. Doubly Linked List 4](#_Toc86592546)

[Operations on Doubly linked list: 5](#_Toc86592547)

[3. Circularly Linked List 5](#_Toc86592548)

[Advantages: 5](#_Toc86592549)

[2. STACK Pointer:- 5](#_Toc86592550)

[Applications of Stack: 5](#_Toc86592551)

[3. QUEUE ADT 6](#_Toc86592552)

[Representing a Queue: (dynamic array) 7](#_Toc86592553)

[A queue have two basic operations: 7](#_Toc86592554)

[Application of Queue: 7](#_Toc86592555)

[4. CIRCULAR QUEUE 8](#_Toc86592556)

[5. Priority Queue 9](#_Toc86592557)

[APPLICATIONS: 9](#_Toc86592558)

[Various operations that can be performed on priority queue are: 9](#_Toc86592559)

[6. HEAPS 9](#_Toc86592560)

[7. ALGORITHMS 14](#_Toc86592561)

[Searching 15](#_Toc86592562)

[1. Linear searching 15](#_Toc86592563)

[2. Binary Search 15](#_Toc86592564)

[Sorting 16](#_Toc86592565)

[1 Bubble sort 16](#_Toc86592566)

[3 Quick sort 18](#_Toc86592567)

[4 Merge sort 19](#_Toc86592568)

[5 Heap sort 20](#_Toc86592569)

[8. Graphs 20](#_Toc86592570)

[Euler Circuit and Euler Path 21](#_Toc86592571)

[1. Adjacency Matrix 21](#_Toc86592572)

[2. Incidence Matrix 21](#_Toc86592573)

[3. Adjacency List 21](#_Toc86592574)

[Depth First Search (DFS) 22](#_Toc86592575)

[Breadth First Search (BFS); 22](#_Toc86592576)

[9. BINARY TREES 23](#_Toc86592577)

[Binary Tree Representation 23](#_Toc86592578)

[a) Sequential Representation 23](#_Toc86592579)

[b) Linked Representation: 24](#_Toc86592580)

[Advantages of linked representation: 24](#_Toc86592581)

[Disadvantages of linked representation: 25](#_Toc86592582)

[TRAVERSING A BINARY TREE 25](#_Toc86592583)

[Deletion of a node from binary search tree. 26](#_Toc86592584)

[Searching for a node in binary search tree. 27](#_Toc86592585)

[AVL TREES 27](#_Toc86592586)

[Definition of Balance Factor: 27](#_Toc86592587)

[Representation of AVL Tree 28](#_Toc86592588)

[Insertion of a node. 28](#_Toc86592589)

[*Deletion* 30](#_Toc86592590)

[Searching: 31](#_Toc86592591)

[BTREES 31](#_Toc86592592)

[Definition: 31](#_Toc86592593)

[Insertion 32](#_Toc86592594)

[B+ Trees 36](#_Toc86592595)

[B+ Tree Internal Node Structure 37](#_Toc86592596)

[B+ Tree Leaf Node Structure 37](#_Toc86592597)

[10. DICTIONARIES: 39](#_Toc86592598)

[1. Linear List Representation 39](#_Toc86592599)

[2. SKIP LIST REPRESENTATION 39](#_Toc86592600)

[Insertion: 40](#_Toc86592601)

[10. HASH TABLE REPRESENTATION 40](#_Toc86592602)

[HASH FUNCTION 40](#_Toc86592603)

[Bucket and Home bucket: 40](#_Toc86592604)

[TYPES OF HASH FUNCTION 40](#_Toc86592605)

[1. Division Method: 40](#_Toc86592606)

[2. Mid Square: 41](#_Toc86592607)

[3. Multiplicative hash function: 41](#_Toc86592608)

[4. Digit Folding: 41](#_Toc86592609)

[5. Digit Analysis: 41](#_Toc86592610)

[11. COLLISION RESOLUTION TECHNIQUES 41](#_Toc86592611)

[1. CHAINING 42](#_Toc86592612)

[2. OPEN ADDRESSING – LINEAR PROBING 42](#_Toc86592613)

[3. QUADRATIC PROBING: 44](#_Toc86592614)

[4. DOUBLE HASHING 45](#_Toc86592615)

[Comparison of Quadratic Probing & Double Hashing 45](#_Toc86592616)

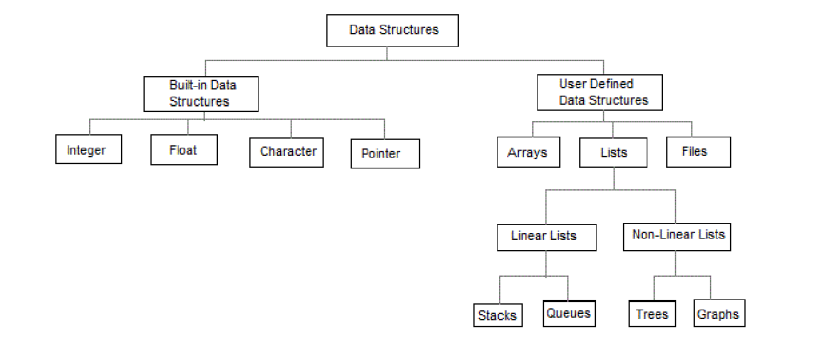
[5. REHASHING 46](#_Toc86592617)

[Advantages: 46](#_Toc86592618)

[6. EXTENSIBLE HASHING 47](#_Toc86592619)

[Deletion Operation: 48](#_Toc86592620)

[Applications of hashing: 49](#_Toc86592621)



# The linked list

To store collections of information limited only by the total amount of memory that the OS will allow us to use.

There is no need to specify our needs in advance.

The linked list is very flexible dynamic data structure.

The items may be added to it or deleted from it at will.

## The linked allocation has the following draw backs:

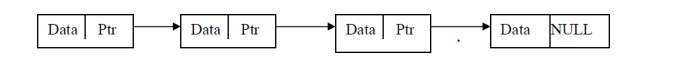
1. No direct access to a particular element.

2. Additional memory required for pointers.

Linked list are of 3 types:

## Singly Linked List

Conceptual view of Singly Linked List



## Operations on Singly linked list:

1. Insertion of a node

2. Deletions of a node

3. Traversing the list

A singly linked list has the disadvantage that we can only traverse it in one direction.

## 2. Doubly Linked List

The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields.

a) One to store data

b) Remaining are self referential pointers which points to previous and next nodes in the list

### Operations on Doubly linked list:

1. Insertion of a node

2. Deletions of a node

3.Traversing the list



## 3. Circularly Linked List

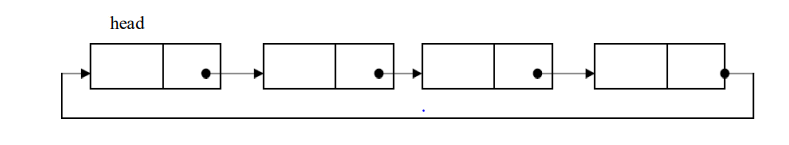
A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.

### Advantages:

1. Any node can be traversed starting from any other node in the list.

2. There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.

3. In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that item itself.



# 2. STACK Pointer:-

1. When an element is placed on the stack, it is said to be pushed on the stack.

2. When an object is removed from the stack, it is said to be popped off the stack.

overflow, which occurs when we try to push more information on a stack that it can hold, and

underflow, which occurs when we try to pop an item off a stack which is empty.

## Applications of Stack:

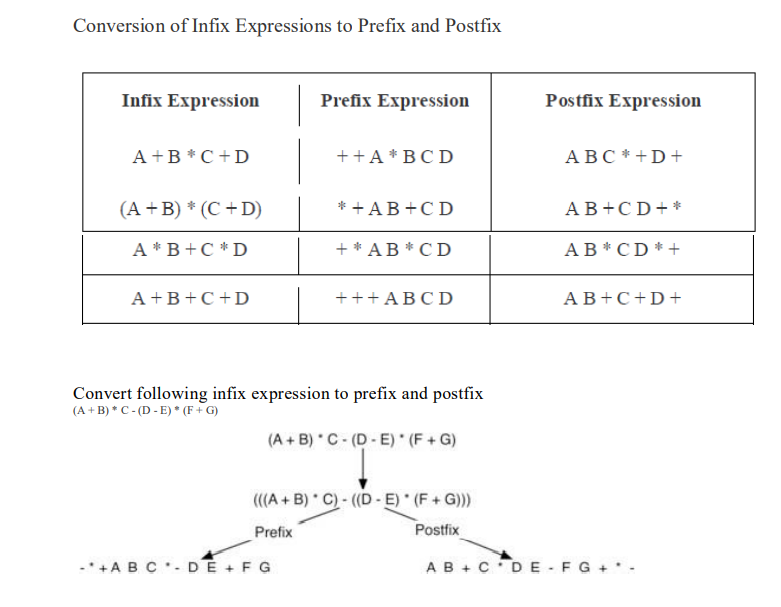
1. Stacks are used in conversion of (in between operator) infix to (after operator) postfix expression.

2. Stacks are also used in evaluation of postfix expression.

3. Stacks are used to implement recursive procedures.

4. Stacks are used in compilers.

5. Reverse String

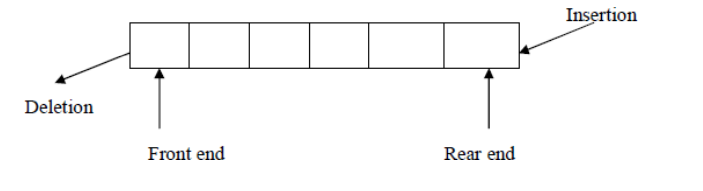


Bracket as differentiator

# 3. QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end.

The key difference when compared stacks is that in a queue the information stored is processed first-in first-out or FIFO. In other words the information receive from a queue comes in the same order that it was placed on the queue.



## Representing a Queue: (dynamic array)

One of the most common way to implement a queue is using array.

An easy way to do so is to define an array

Queue, and two additional variables front and rear.

The rules for manipulating these variables are simple:

1. Each time information is added to the queue, increment rear.

2. Each time information is taken from the queue, increment front.

3. Whenever front >rear or front=rear=-1 the queue is empty.

Array implementation of a Queue do have drawbacks. The maximum queue size has to be set at compile time, rather than at run time. Space can be wasted, if we do not use the full capacity of the array

## A queue have two basic operations:

a) The operation of adding new item on the queue occurs only at one end of the queue called the rear or back.

b)The operation of removing items of the queue occurs at the other end called the front.

1. For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is maxSize.

2. The variable front will hold the index of the item that is considered the front of the queue, while the rear variable will hold the index of the last item in the queue. Assume that initially the front and rear variables are initialized to -1.

3. Like stacks, underflow and overflow conditions are to be checked before operations in a queue

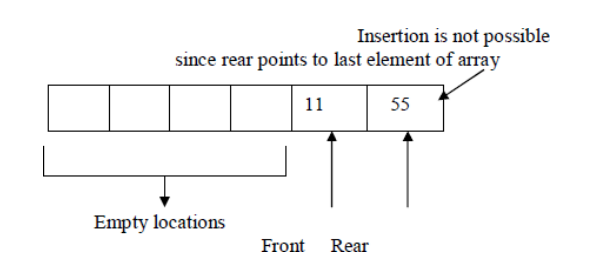
## Application of Queue:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

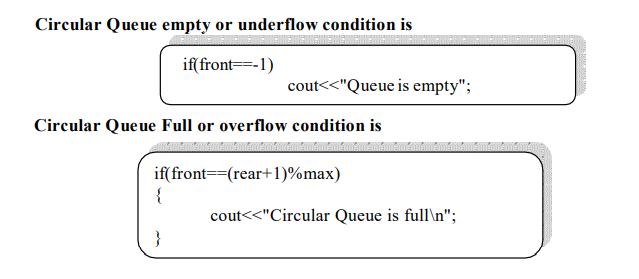
Once the queue gets filled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted.

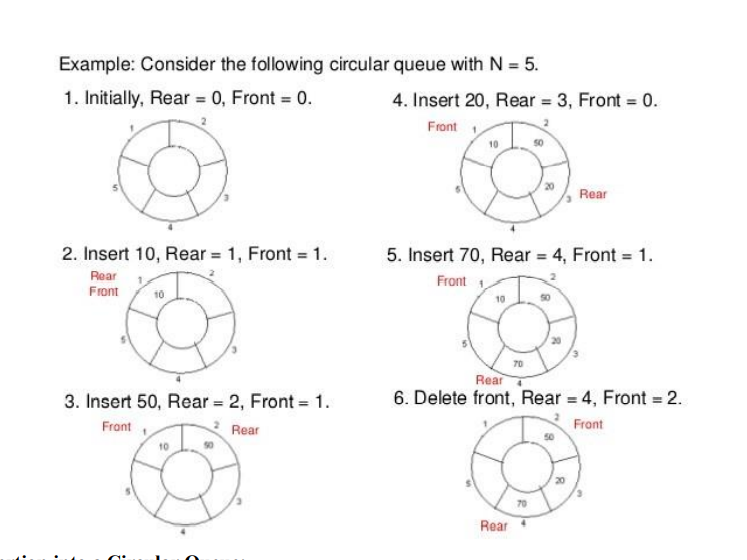


When the queue contains very few items and the rear pointer points to last element. i.e. rear=maxSize-1, we cannot insert any more items into queue because the overflow condition satisfies. That means a lot of space is wasted. .Frequent reshuffling of elements is time consuming. Solution is to arrange them in circular fashion.

# 4. CIRCULAR QUEUE

A circular queue is a queue in which all locations are treated as circular such that the first location CQ[0] follows the last location CQ[max-1].





# 5. Priority Queue

A priority queue is a collection of zero or more elements. Each element has a priority or value.

Unlike the queues, which are FIFO structures, the order of deleting from a priority queue is determined by the element priority.

Elements are removed/deleted either in increasing or decreasing order of priority rather than in the order in which they arrived in the queue.

There are two types of priority queues:

1. Min priority queue (smallest first)

2. Max priority queue (largest first)

In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.

The implementation of priority queue can be done using arrays or linked list.

## APPLICATIONS:

The typical example of priority queue is scheduling the jobs in operating system.

In OS there are 3 jobs- real time jobs, foreground jobs and background jobs.

The OS always schedules the real time jobs first.

If there is no real time jobs pending then it schedules foreground jobs.

Lastly if no real time and foreground jobs are pending then OS schedules the background jobs.

2. In network communication, the manage limited bandwidth for transmission the priority queue is used.

3. In simulation modeling to manage the discrete events the priority queue is used.

## Various operations that can be performed on priority queue are:

1. Find an element

2. Insert a new element

3. Remove or delete an element The abstract data type specification for a max priority queue is given below.

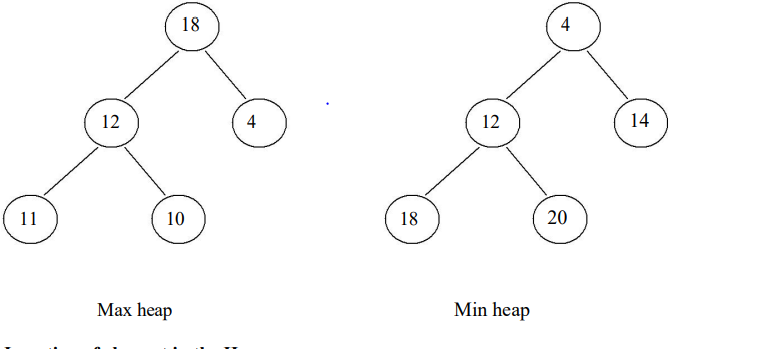
The specification for a min priority queue is the same as ordinary queue except while deletion, find and remove the element with minimum priority.

# 6. HEAPS

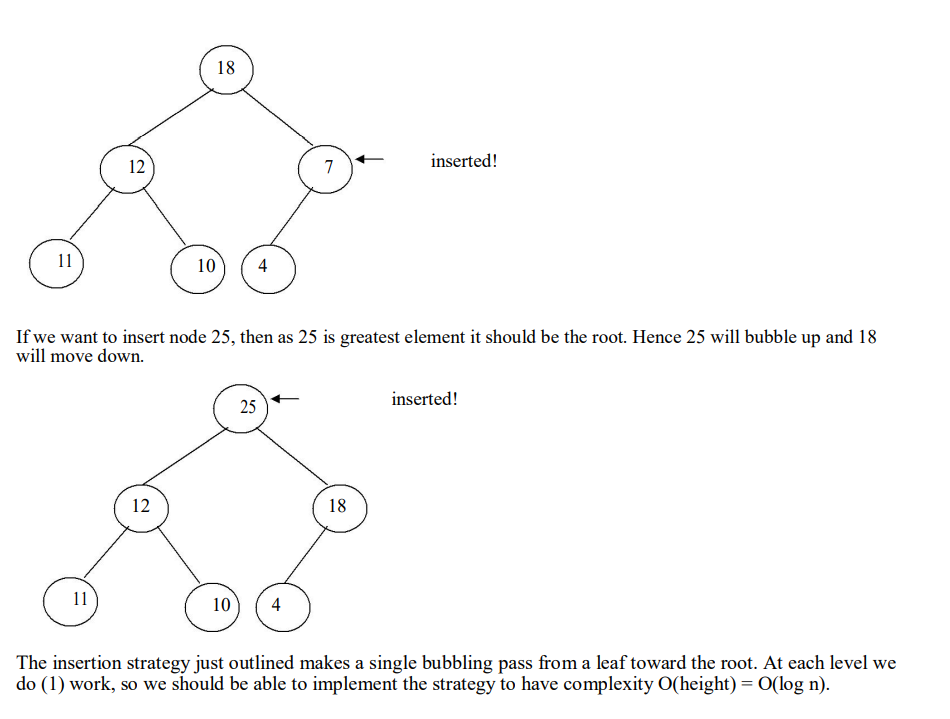
Heap is a tree data structure denoted by either a max heap or a min heap.

A max heap is a tree in which value of each node is greater than or equal to value of its children nodes.

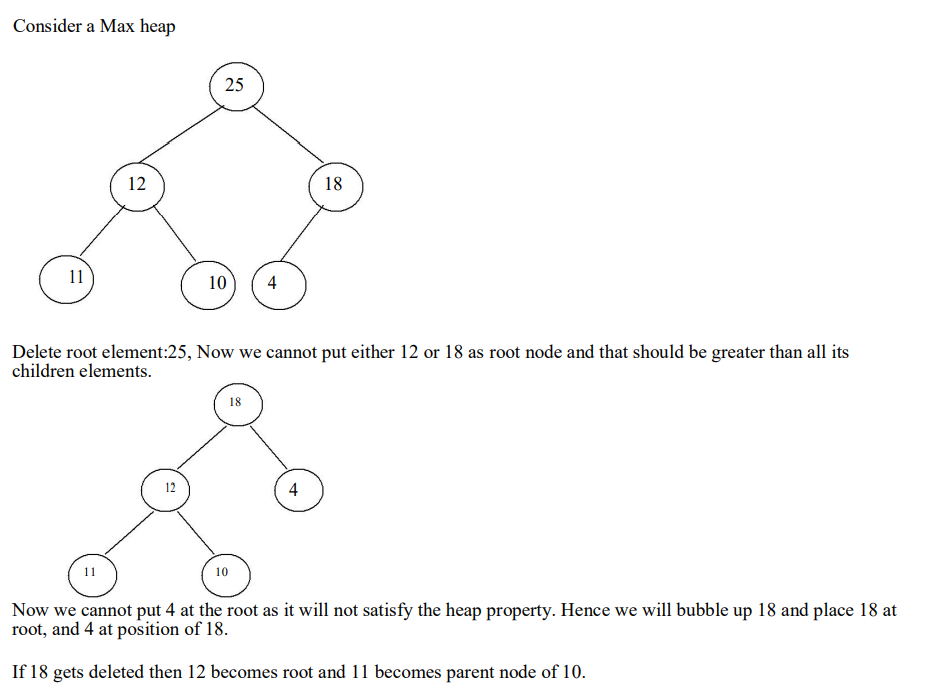
A min heap is a tree in which value of each node is less than or equal to value of its children nodes.

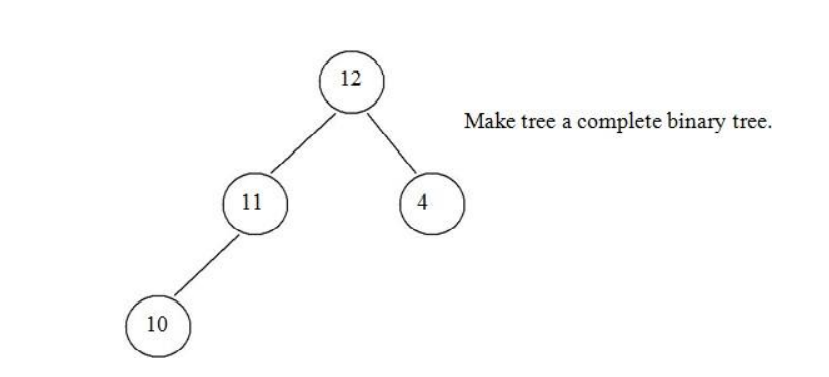


Now if we want to insert 7. We cannot insert 7 as left child of 4. This is because the max heap has a property that value of any node is always greater than the parent nodes. Hence 7 will bubble up 4 will be left child of 7. Note: When a new node is to be inserted in complete binary tree we start from bottom and from left child on the current level. The heap is always a complete binary tree.



For deletion operation always the maximum element is deleted from heap. In Max heap the maximum element is always present at root. And if root element is deleted then we need to reheapify the tree.





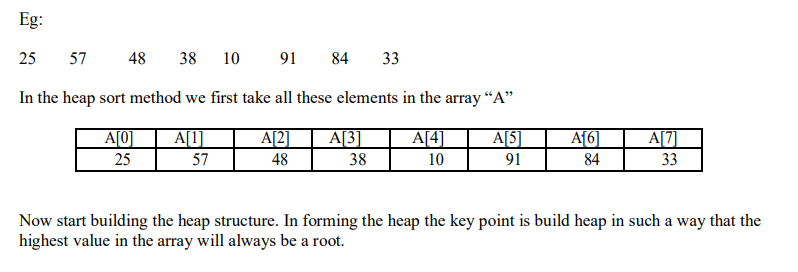
Thus deletion operation can be performed. The time complexity of deletion operation is O(log n).

1. Remove the maximum element which is present at the root. Then a hole is created at the root.

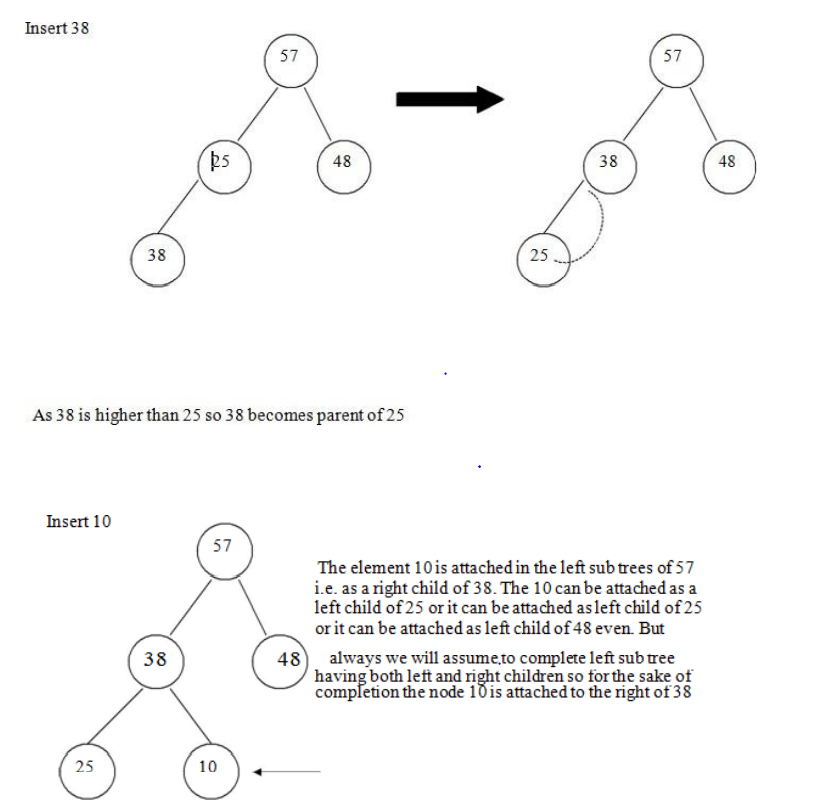
2. Now reheapify the tree. Start moving from root to children nodes. If any maximum element is found then place it at root. Ensure that the tree is satisfying the heap property or not.

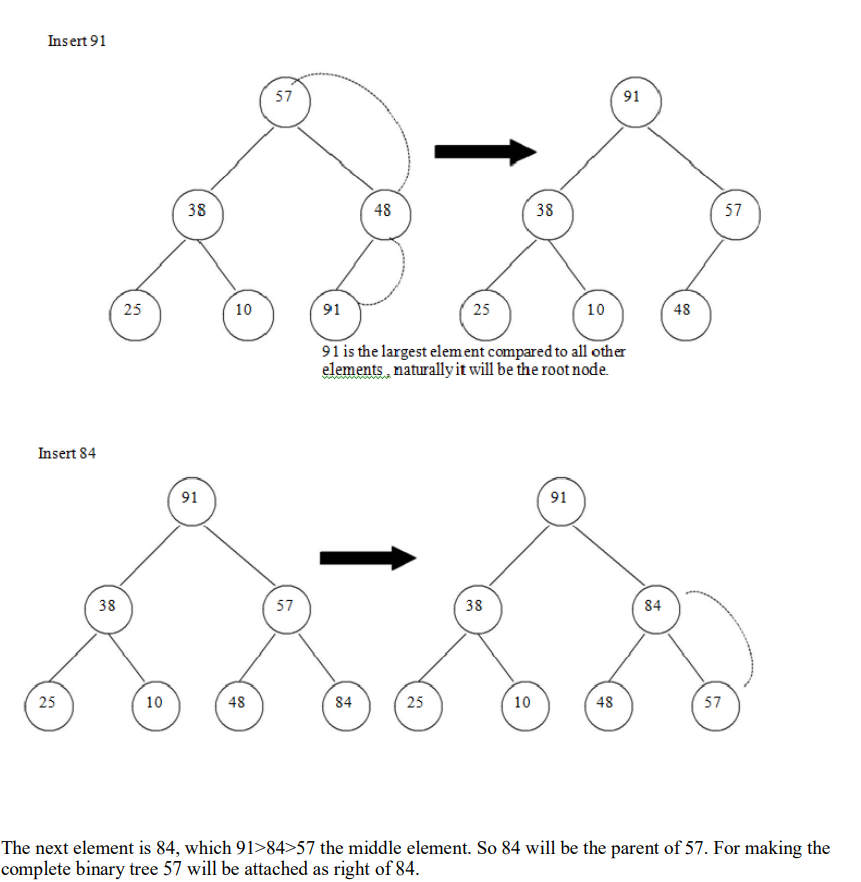
3. Repeat the step 1 and 2 if any more elements are to be deleted.

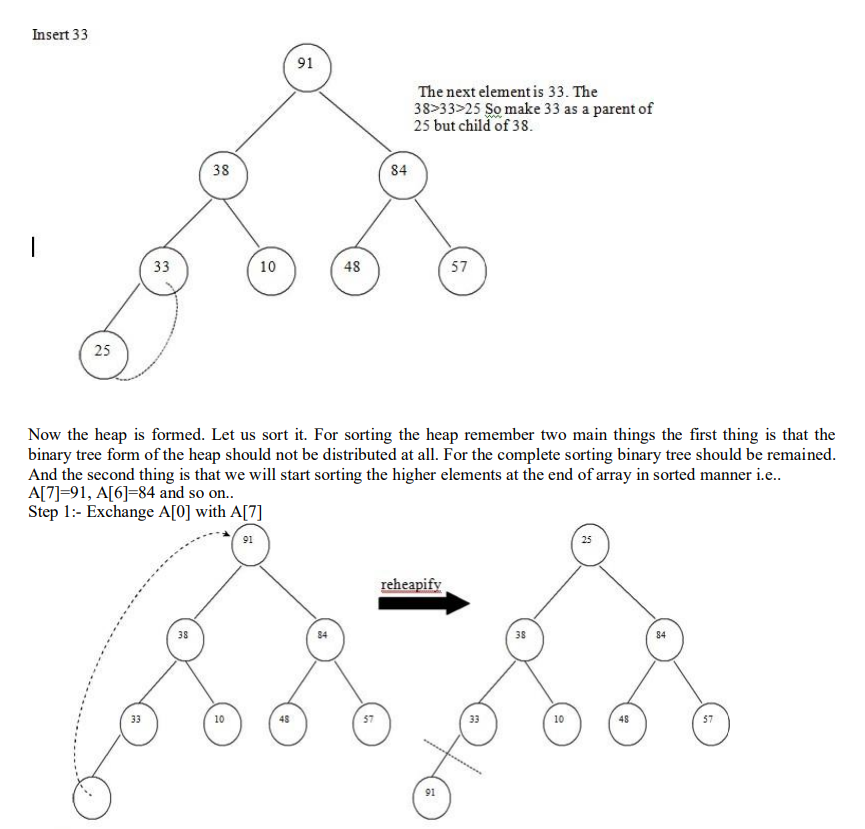
HEAP SORT Heap sort is a method in which a binary tree is used. In this method first the heap is created using binary tree and then heap is sorted using priority queue

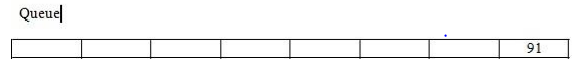


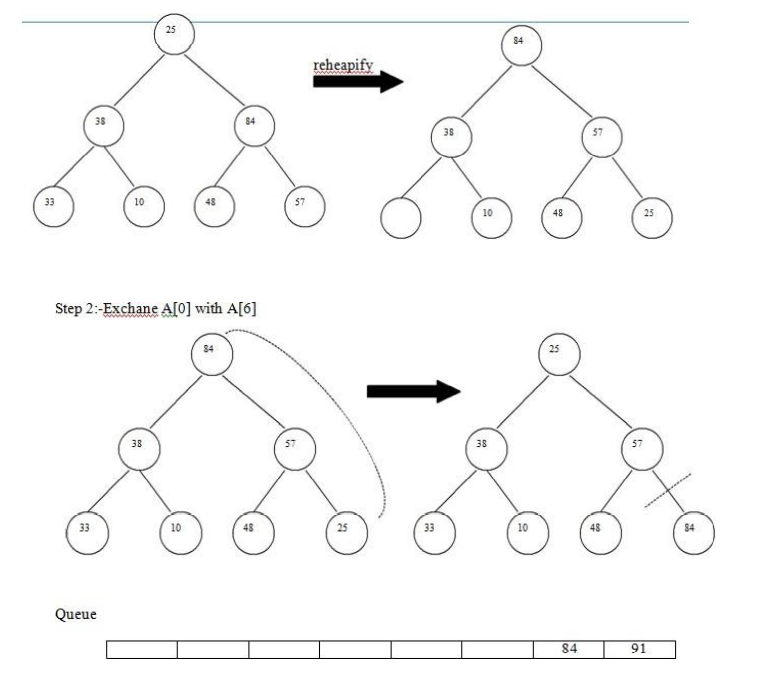


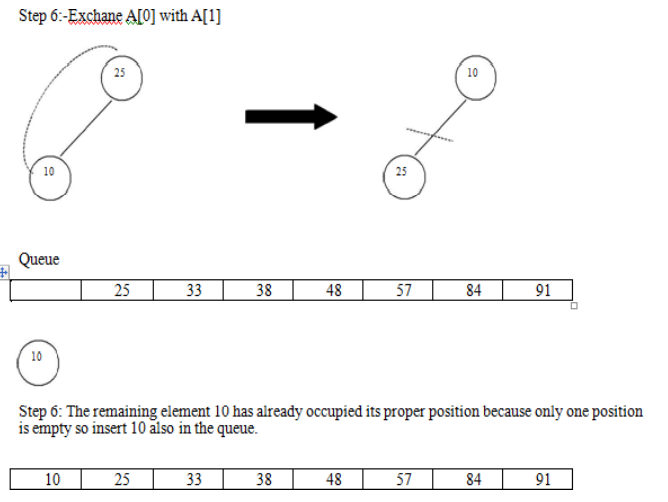












# 7. ALGORITHMS

An Algorithm is a method of representing the step-by-step procedure for solving a problem. It is a method of finding the right answer to a problem or to a different problem by breaking the problem into simple cases. It must possess the following properties:

1. Finiteness: An algorithm should terminate in a finite number of steps.

2. Definiteness: Each step of the algorithm must be precisely (clearly) stated.

3. Effectiveness: Each step must be effective.i.e; it should be easily convertible into program statement and can be performed exactly in a finite amount of time.

4. Generality: Algorithm should be complete in itself, so that it can be used to solve all problems of given type for any input data.

5. Input/Output: Each algorithm must take zero, one or more quantities as input data and gives one of more output values. An algorithm can be written in English like sentences or in any standard representations. The algorithm written in English language is called Pseudo code.

## Searching

Searching is the technique of finding desired data items that has been stored within some data structure.

Data structures can include linked lists, arrays, search trees, hash tables, or various other storage methods.

The appropriate search algorithm often depends on the data structure being searched. Search algorithms can be classified based on their mechanism of searching.

### 1. Linear searching

Linear or Sequential searching: Linear Search is the most natural searching method and It is very simple but very poor in performance at times. In this method, the searching begins with

searching every element of the list till the required record is found.

We begin search by comparing the first element of the list with the target element.

If it matches, the search ends and position of the element is returned.

Otherwise, we will move to next element and compare.

In this way, the target element is compared with all the elements until a match occurs.

If the match do not occur and there are no more elements to be compared, we conclude that target element is absent in the list by returning position as -1.

Linear search can be implemented in two ways. 1)Non recursive 2)recursive

### Binary Search

Binary search is a fast search algorithm with run-time complexity of Ο(log n).

This search algorithm works on the principle of divide and conquer.

Binary search looks for a particular item by comparing the middle most item of the collection.

If a match occurs, then the index of item is returned.

If the middle item is greater than the item,

then the item is searched in the sub-array to the left of the middle item.

Otherwise, the item is searched for in the sub-array to the right of the middle item.

This process continues on the sub-array as well until the size of the subarray reduces to zero.

Before applying binary searching, the list of items should be sorted in ascending or descending order.

Best case time complexity is O(1)

Worst case time complexity is O(log n)



Arranging the elements in a list either in ascending or descending order. various sorting algorithms are

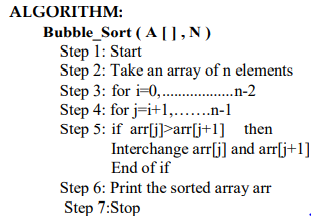
## Sorting

### 1 Bubble sort

The bubble sort is an example of exchange sort.

In this method, repetitive comparison is performed among elements and essential swapping of elements is done.

Bubble sort is commonly used in sorting algorithms. It is easy to understand but time consuming i.e. takes more number of comparisons to sort a list.

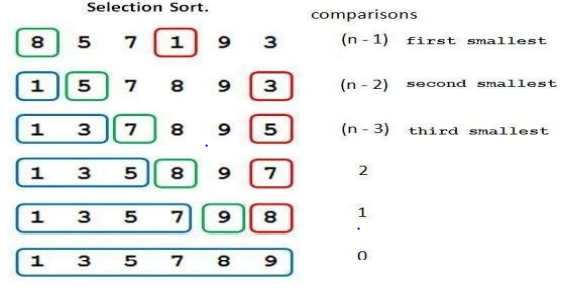


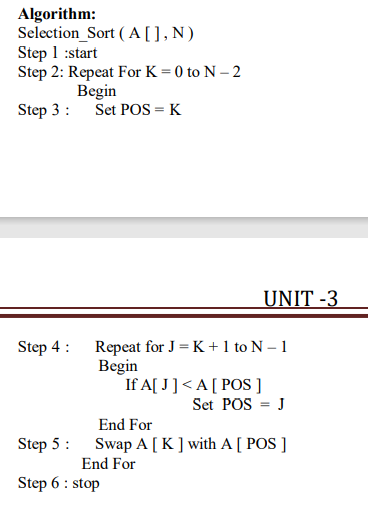
• selection sort

( Select the smallest and Exchange ):

The first item is compared with the remaining n-1 items, and whichever of all is lowest, is put in the first position.

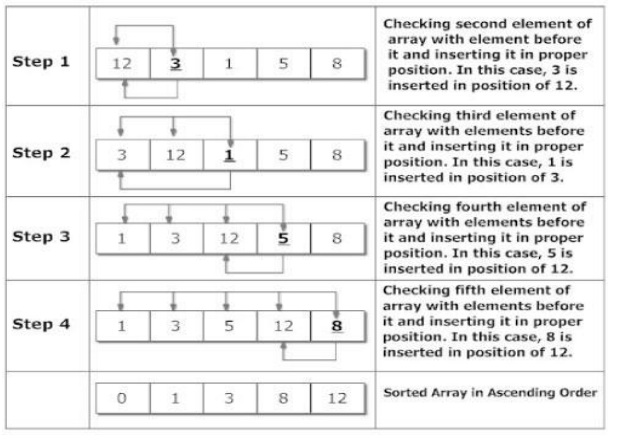
Then the second item from the list is taken and compared with the remaining (n-2) items, and so on

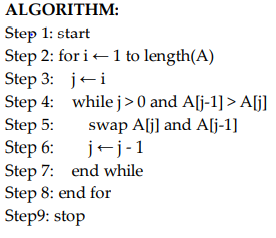




2 Insertion sort

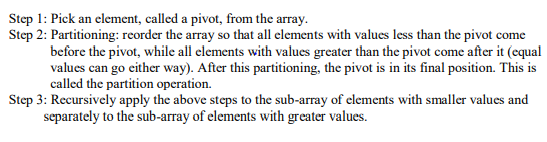
It iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain

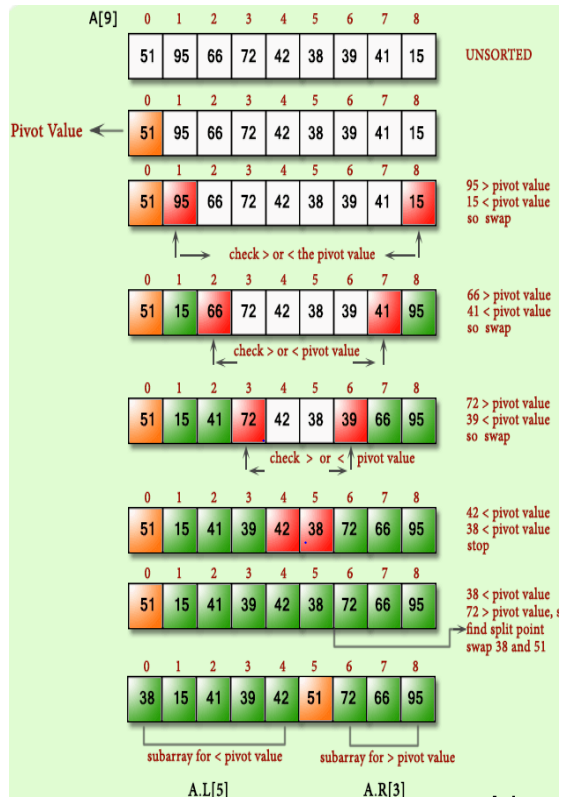


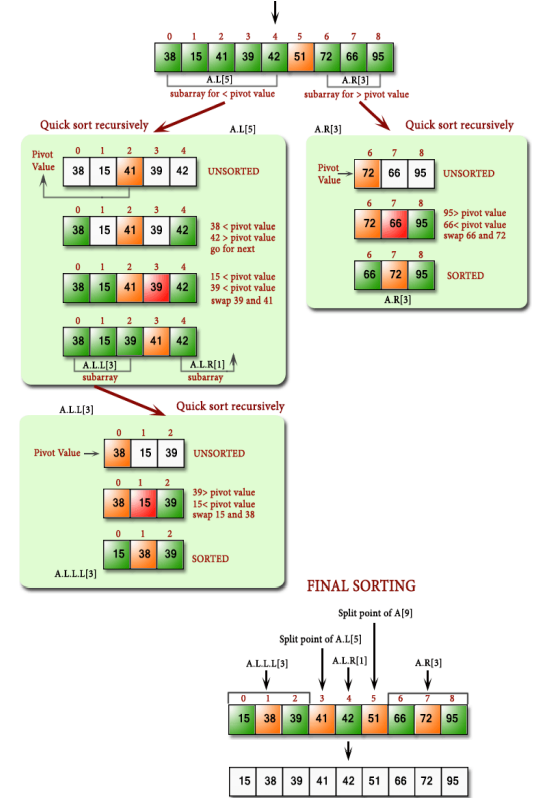


### 3 Quick sort

It is a divide and conquer algorithm. Developed by Tony Hoare in 1959. Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays.

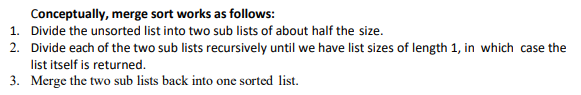


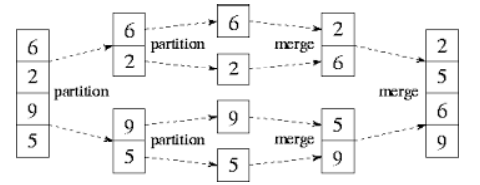




### 4 Merge sort

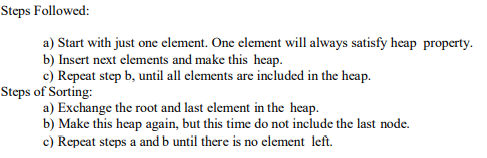
Merge sort is a sorting technique based on divide and conquer technique. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced. Merge Sort is quite fast, and has a time complexity of O(n log n).

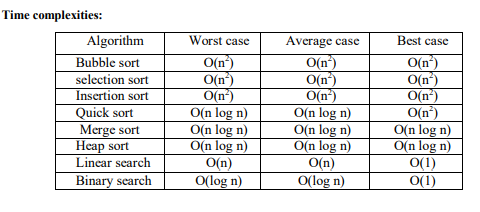




### 5 Heap sort

It is a completely binary tree with the property that a parent is always greater than or equal to either of its children (if they exist). first the heap (max or min) is created using binary tree and then heap is sorted using priority queue.





Sort Priority ((hs, ms)>(qs)>(is,ss,bs)

Search Priority (bs>ls)

# 8. Graphs

A graph G is a discrete structure consisting of nodes (called vertices) and lines joining the nodes (called edges).

Two vertices are adjacent to each other if they are joint by an edge.

The edge joining the two vertices is said to be an edge incident with them.

We use V (G) and E(G) to denote the set of vertices and edges of G respectively.

Graph Representations Graph data structure is represented using following representations... 1. Adjacency Matrix

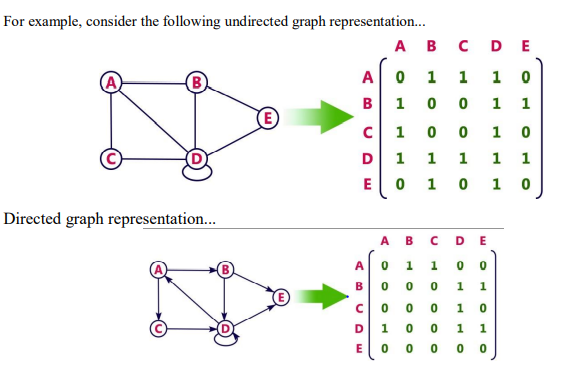
2. Incidence Matrix

3. Adjacency List

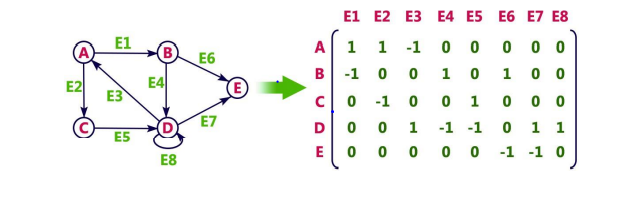
## Euler Circuit and Euler Path

An Euler circuit in a graph G is a simple circuit containing every edge of G. An Euler path in G is a simple path containing every edge of G.

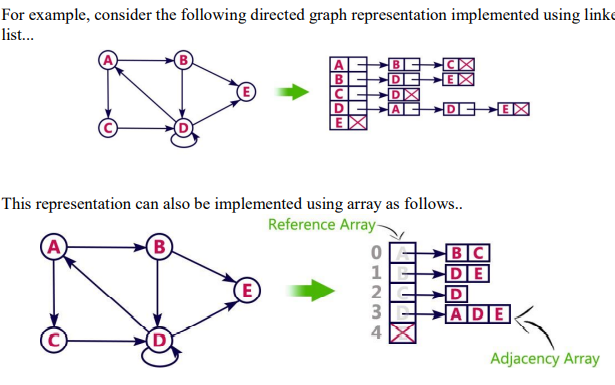
## Adjacency Matrix



## Incidence Matrix



## Adjacency List



## Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse.

All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

DFS can be implemented using stacks.

The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

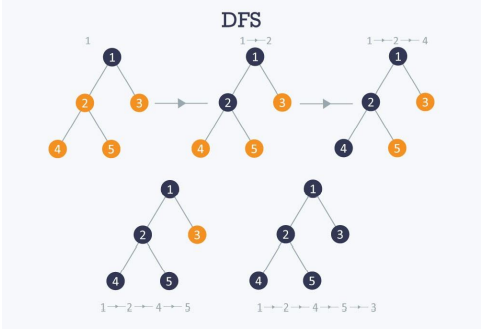
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty.

However, ensure that the nodes that are visited are marked.

This will prevent you from visiting the same node more than once.

If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.



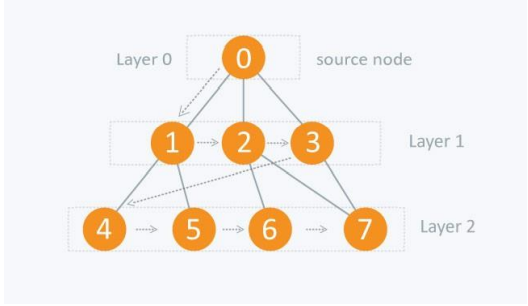
## Breadth First Search (BFS);

.BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node).

You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows: 1.First move horizontally and visit all the nodes of the current layer

2.Move to the next layer



# 9. BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

A binary tree is either empty or consists of

a) a node called the root

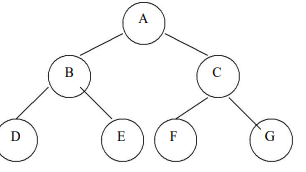
b)left and right sub trees are themselves binary trees.

Complete binary tree:

The tree in which degree of each node is at the most two is called a complete binary tree.

In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on.

So we can say that a complete binary tree depth d will contain exactly 2l nodes at each level l, where l is from 0 to d.



Note: 1. A binary tree of depth n will have maximum 2n -1 nodes.

2. A complete binary tree of level l will have maximum 2l nodes at each level, where l starts from0.

3. Any binary tree with n nodes will have at the most n+1 null branches.

4. The total number of edges in a complete binary tree with n terminal nodes are 2(n-1).

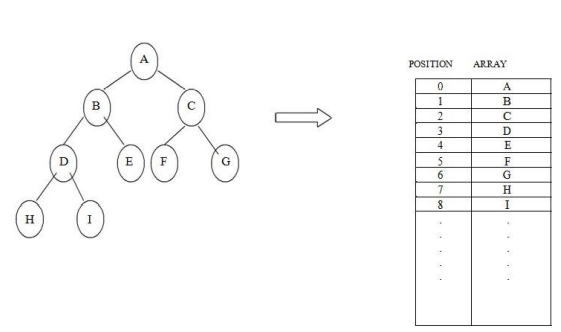
## Binary Tree Representation

### a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.

1) The root of binary tree is stored in the 1st location of array

2) If a node is in the j th location of array, then its left child is in the location 2J+1 and its right child in the location 2J+2 The maximum size that is required for an array to store a tree is [(2^d+1) -1], where d is the depth of the tree.



#### Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular node is fast because of the random access.

#### Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.

2. In this type of representation the maximum depth of the tree has to be fixed. Because we have decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we coose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.

3. The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary tree can be preserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

## b) Linked Representation:

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer.

In linked list each node will look like this:



### Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.

2. Insertions and deletions which are the most common operations can be done without moving the nodes.

### Disadvantages of linked representation:

1. This representation does not provide direct access to a node and special algorithms are required.

2. This representation needs additional space in each node for storing the left and right sub-trees

## TRAVERSING A BINARY TREE

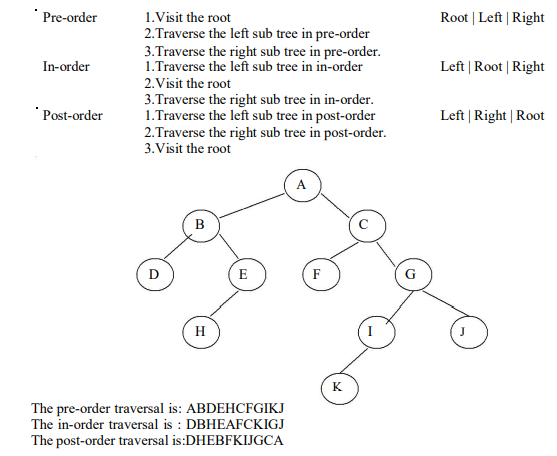
Traversing a tree means that processing it so that each node is visited exactly once. A binary tree can be traversed a number of ways.

The most common tree traversals are

1 In-order

2 Pre-order and

3 Post-order





Operations On Binary Search Tree:

1. Insertion of a node in binary search tree.

2. Deletion of a node from binary search tree.

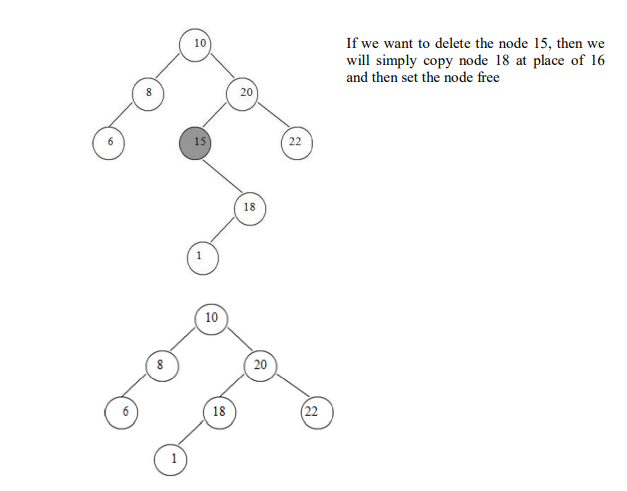
3. Searching for a particular node in binary search tree.

### Deletion of a node from binary search tree.

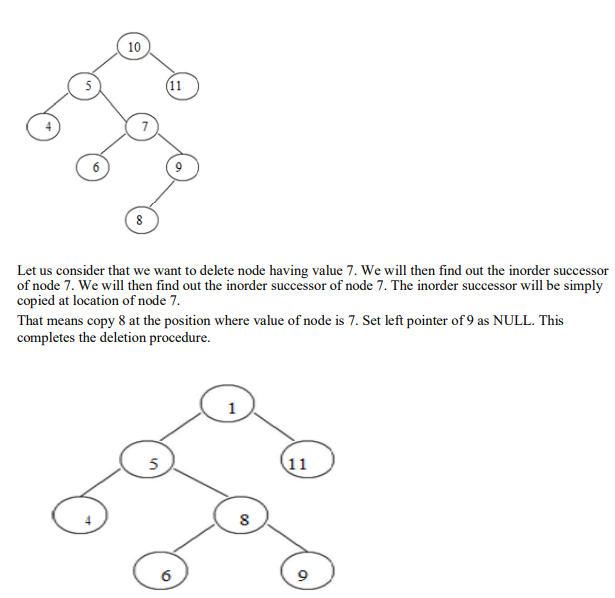
i. Deletion of leaf node.

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.

ii. Deletion of a node having one child.

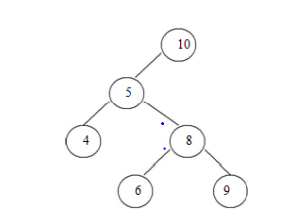


iii. Deletion of a node having two children



### Searching for a node in binary search tree.

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare “node is not present in the tree”



In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub tree. Now compare 9 with 8 but 9 is greater than 8 we will move on right sub branch. As the node we will get holds the value 9. Thus the desired node can be searched

## AVL TREES

An empty tree is height balanced if T is a non empty binary tree with TL and TR as its left and right sub trees. The T is height balanced if and only if

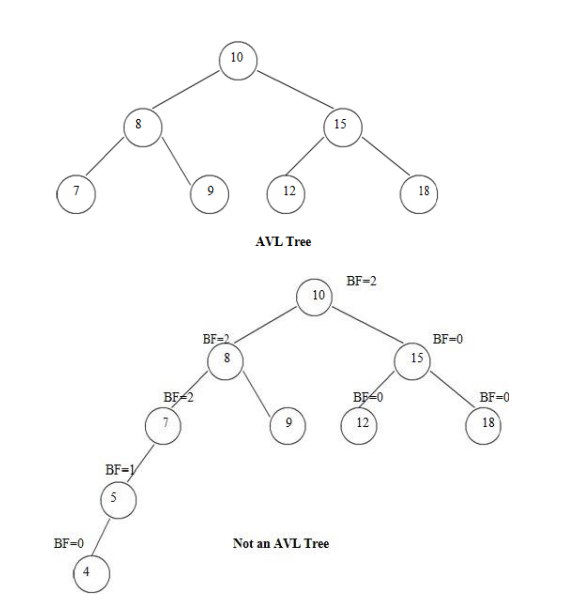
TL and TR are height balanced.

ii. hL-hR <= 1 where hL and hR are heights of TL and TR. The idea of balancing a tree is obtained by calculating the balance factor of a tree.

### Definition of Balance Factor:

The balance factor BF(T) of a node in binary tree is defined to be hL-hR where hL and hR are heights of left and right sub trees of T.

For any node in AVL tree the balance factor i.e. BF(T) is -1, 0 or +1.



The height of AVL tree with n elements (nodes) is O(log n)

### Representation of AVL Tree

The AVL tree follows the property of binary search tree. In fact AVL trees are basically binary search trees with balance factors as -1, 0, or +1.

After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0, or +1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition. The balance factor is denoted at right top corner inside the node.

### Insertion of a node.

There are four different cases when rebalancing is required after insertion of new node.

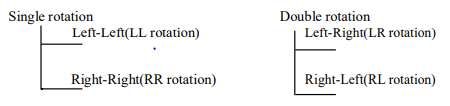
1. An insertion of new node into left sub tree of left child. (LL).

2. An insertion of new node into right sub tree of left child. (LR).

3. An insertion of new node into left sub tree of right child. (RL).

4. An insertion of new node into right sub tree of right child.(RR).

Some modifications done on AVL tree in order to rebalance it is called rotations of AVL tree There are two types of rotations:



#### Insertion Algorithm:

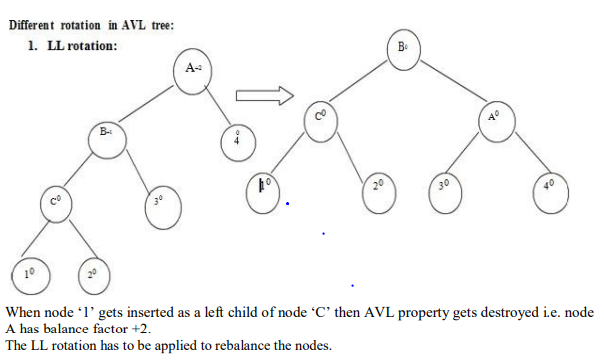
1.Insert a new node as new leaf just as an ordinary binary search tree.

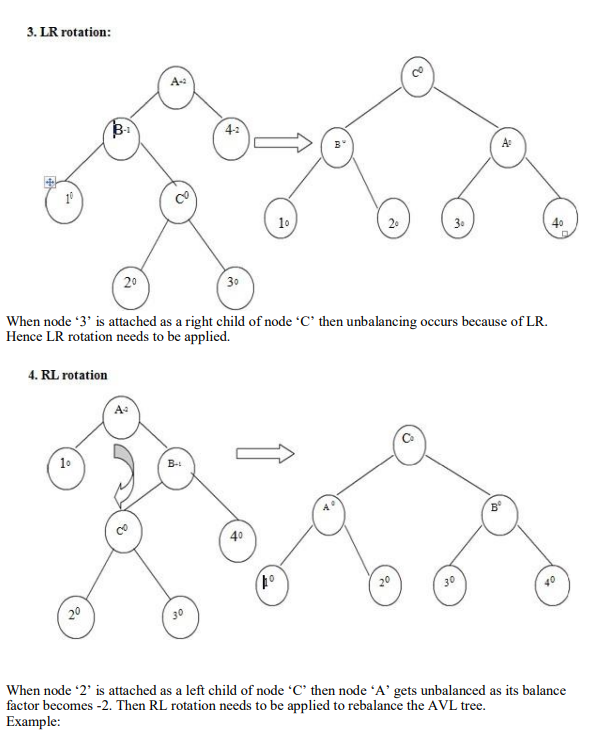
2.Now trace the path from insertion point(new node inserted as leaf) towards root. For each node ‘n’ encountered, check if heights of left (n) and right (n) differ by at most 1.

a)If yes, move towards parent (n).

b)Otherwise restructure by doing either a single rotation or a double rotation.

Thus once we perform a rotation at node ‘n’ we do not require to perform any rotation at any ancestor on ‘n’





#### Deletion:

Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property.

And thereby various rotations need to be applied.

#### Algorithm for deletion:

The deletion algorithm is more complex than insertion algorithm.

1. Search the node which is to be deleted.

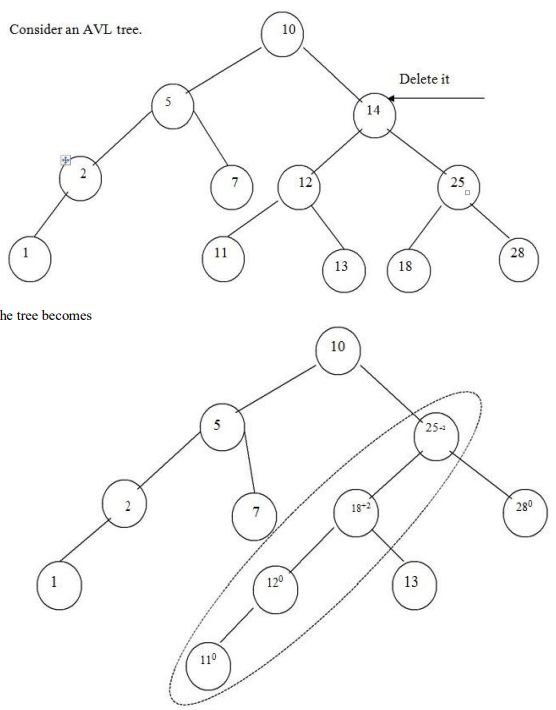
2. a) If the node to be deleted is a leaf node then simply make it NULL to remove.

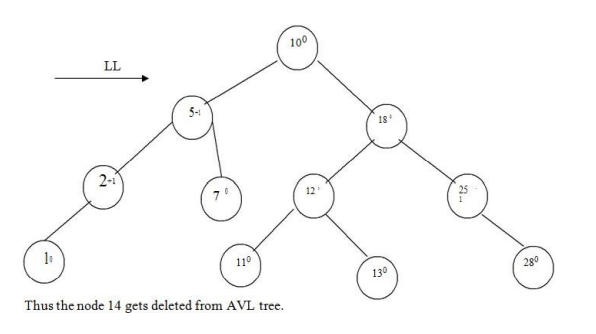
b) If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove this node.

3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path.

If we encounter unbalancing in some sub tree then balance that sub tree using appropriate single or double rotations.

The deletion algorithm takes O(log n) time to delete any node





### Searching:

The searching of a node in an AVL tree is very simple.

As AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is the same one is used to search a node from AVL tree.

The searching of a node from AVL tree takes O(log n) time.

## BTREES

Multi-way trees are tree data structures with more than two branches at a node. The data structures of m-way search trees, B trees and Tries belong to this category of tree structures.

AVL search trees are height balanced versions of binary search trees, provide efficient retrievals and storage operations. The complexity of insert, delete and search operations on AVL search trees id O(log n).

Applications such as File indexing where the entries in an index may be very large, maintaining the index as m-way search trees provides a better option than AVL search trees which are but only balanced binary search trees.

While binary search trees are two-way search trees, m-way search trees are extended binary search trees and hence provide efficient retrievals.

B trees are height balanced versions of m-way search trees and they do not recommend representation of keys with varying sizes.

Tries are tree based data structures that support keys with varying sizes.

### Definition:

A B tree of order m is an m-way search tree and hence may be empty. If non empty, then the following properties are satisfied on its extended tree representation:

i. The root node must have at least two child nodes and at most m child nodes.

ii. All internal nodes other than the root node must have at least |m/2 | non empty child nodes and at most m non empty child nodes.

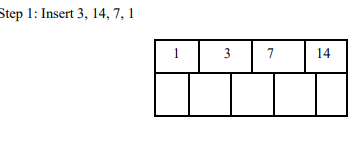
iii. The number of keys in each internal node is one less than its number of child nodes and these keys partition the keys of the tree into sub trees.

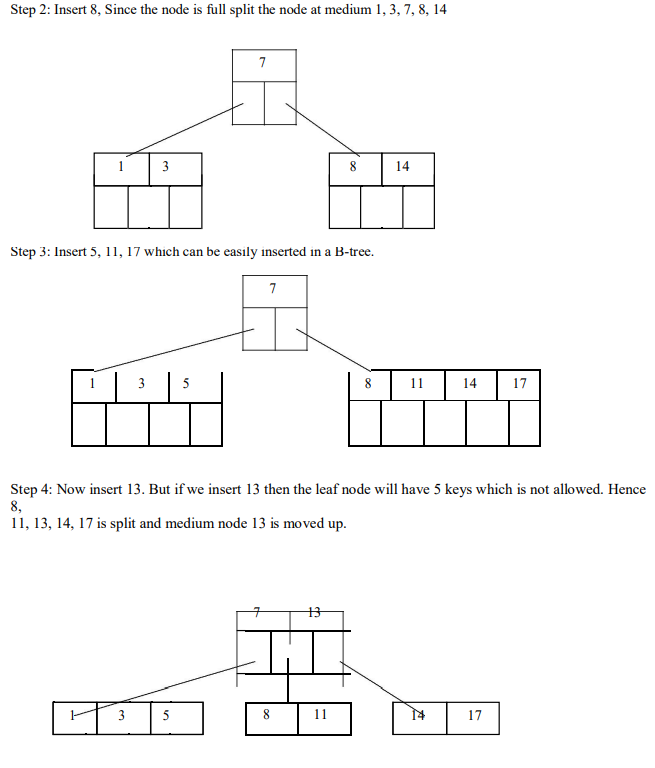
iv. All external nodes are at the same level.

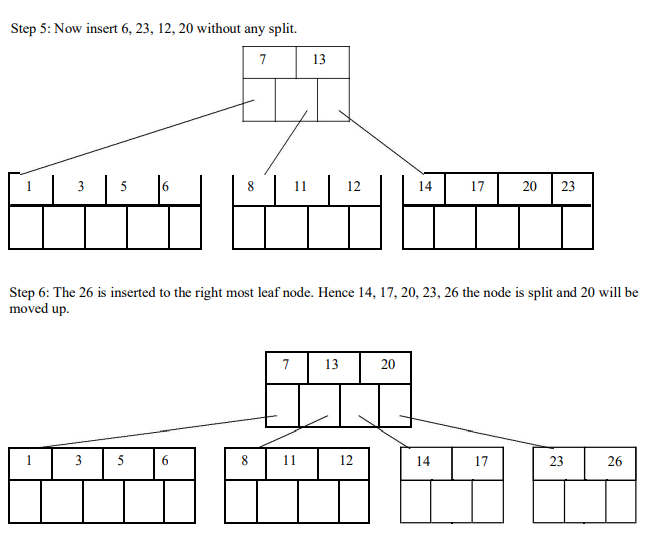
### Insertion

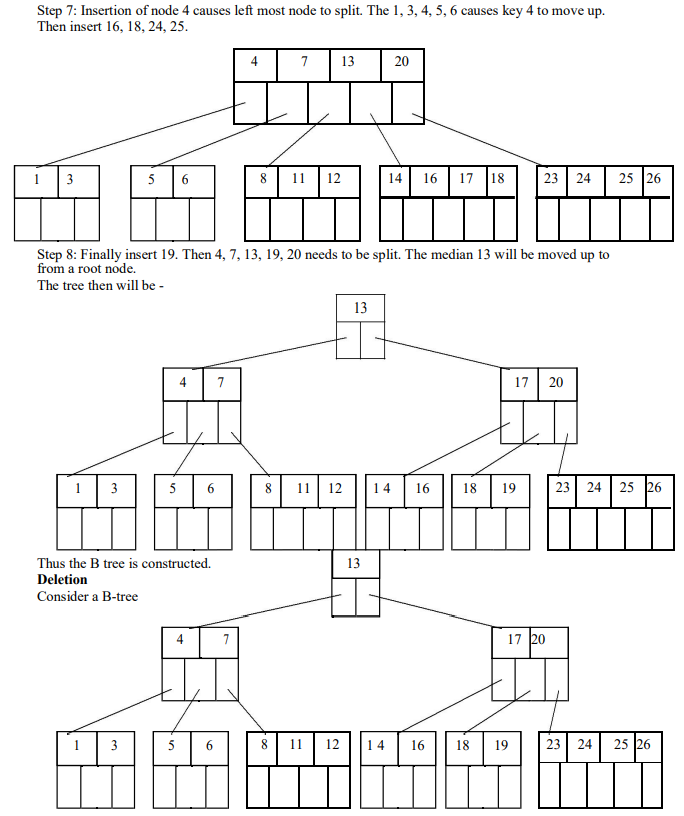
For example construct a B-tree of order 5 using following numbers. 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19 The order 5 means at the most 4 keys are allowed.

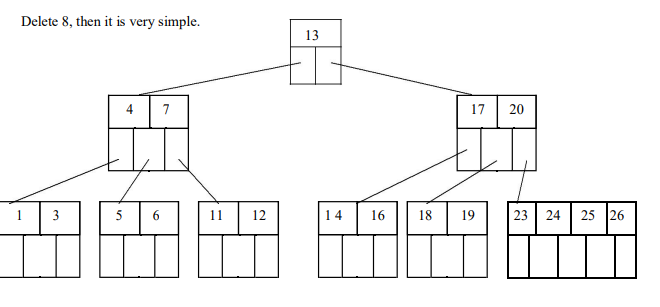
The internal node should have at least 3 non empty children and each leaf node must contain at least 2 keys.



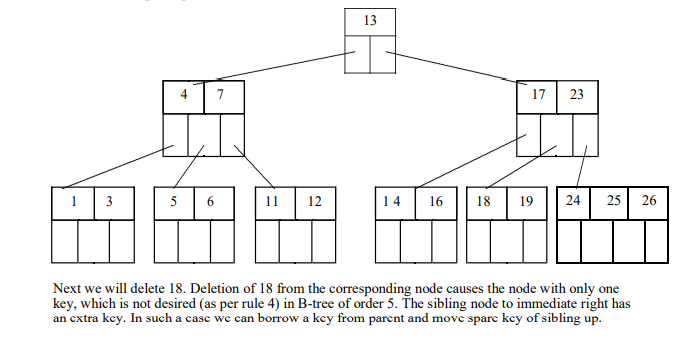


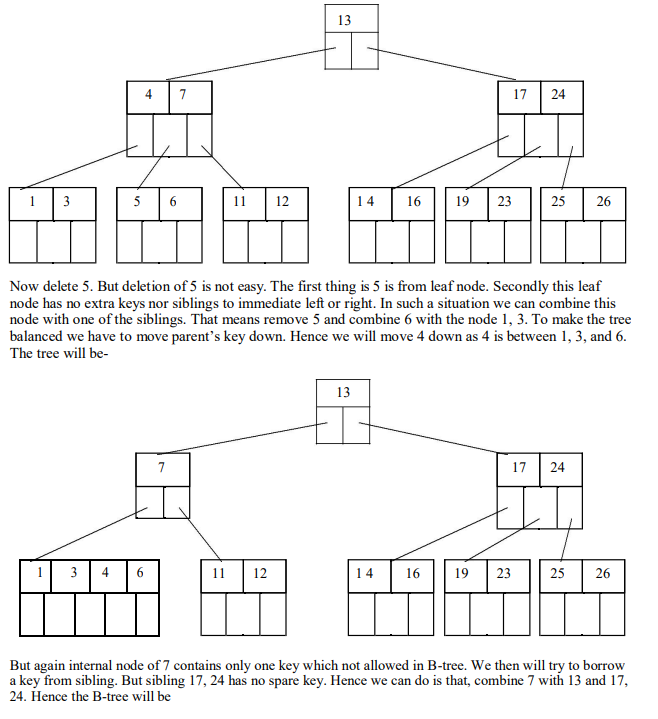


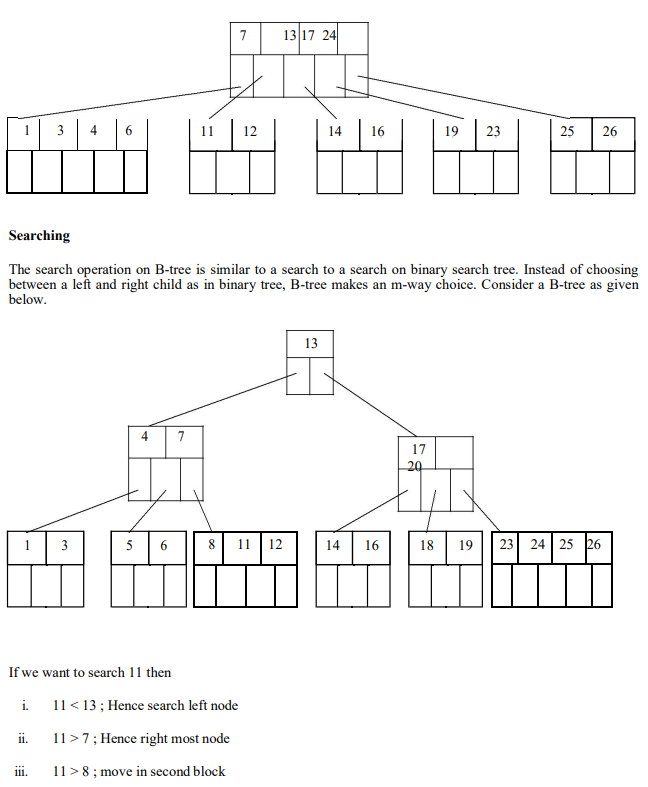




Now we will delete 20, the 20 is not in a leaf node so we will find its successor which is 23, Hence 23 will be moved up to replace 20







iv. node 11 is found

The running time of search operation depends upon the height of the tree. It is O(log n)

## B+ Trees

• Most implementations use the B-tree variation, the B+-tree.

• In the B-tree, every value of the search field appears once at some level in the tree, along with the data pointer to the record, or block where the record is stored.

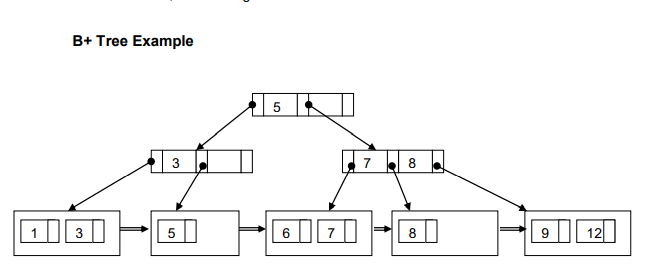
• In a B+ tree, data pointers are stored only at the leaf nodes, therefore the structure of the leaf nodes vary from the structure of the internal (non leaf) nodes.

• If the search field is a key field, the leaf nodes have a value for every value of the search field, along with the data pointer to the record or block.

• If the search field is a non key field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection (similar to option 3 for the secondary indexes)

• The leaf nodes of the B+ Trees are linked to provide ordered access on the search field to the record. The first level is similar to the base level of an index.

• Some search field values in the leaf nodes are repeated in the internal nodes of the B+ trees, in order to guide the search.



### B+ Tree Internal Node Structure

1. Each internal node is of the form , where q<=p and each Pi is a tree pointer.

2. Within each internal node, K1 < K2 < ….< Kq-1.

3. For all search field values X in the subtree pointed at by Pi, we have:

• Ki-1 < X <= Ki for 1 < i < q;

• X <=K for i = 1;

• and Ki-1 < X for i = q.

4. Each internal node has at most, p tree pointers.

5. Each internal node, except the root, has at least ⎡p/2⎤ tree pointers. The root node has at least two tree pointers if it is an internal node.

6. An internal node with q pointers, q <=p, has q-1 search field values.

### B+ Tree Leaf Node Structure

1. Each leaf node is of the form, <, ,… ,, Pnext> where q<=p, each Pri is a data pointer, and Pnext points to the next leaf node of the B+ tree.

2. Within each leaf node, K1 < K2 < …< Kq-1, q<=p

3. Each Pri is a data pointer that points to the record whose search field value is Ki, or to a file block containing the record (or a block of pointers if the search field is not a key field)

4. Each leaf node has at least ⎡p/2⎤ values.

5. All leaf nodes are at the same level. B+ Tree Information

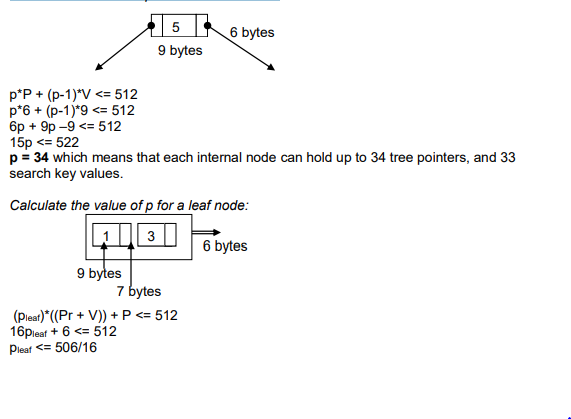
• By starting at the leftmost block, it is possible to traverse leaf nodes as a linked list using the Pnext pointers. This provides ordered access to the data records on the indexing field.

• Entries in internal nodes of a B+ tree include search values and tree pointers, without any data pointers, more entries can be stored into an internal node of a B+ tree, than for a B-tree.

• Therefore the order p will be larger for a B+ tree, which leads to fewer B+ tree levels, improving the search time.

• The order p can be different for the internal and leaf nodes, because of the structural differences of the nodes. Example 6 from Text To calculate the order p of a B+ Tree. suppose the search key field is V = 9 bytes long, the block size is B = 512 bytes, a record pointer is Pr = 7 bytes and a block pointer is P = 6 bytes. An internal node of the B+ trees can have up to p tree pointers and p – 1 search field values, which must fit into a single block.

Calculate the value of p for an internal node:



pleaf = 31 which means each leaf node can hold up to pleaf = 31

value/data pointer combinations, assuming data pointers are record pointers.

Example 7 from Text

Suppose that we construct a B+ tree on the field of Example 6. To calculate the approximate number of entries of the B+ tree we assume that each node is 69 percent full. On average, each internal node will have 34 \* 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average will hold 0.69\*pleaf = 0.69\*31 or approximately 21 data record pointers. A B+ tree will have the following average number of entries at each level.



When we compare this result with the previous B-tree example (Example 5), we can see that the B+ tree can hold up to 255,507 record pointers, whereas a corresponding B-tree can only hold 65,535 entries. Insertion and Deletion with B+-trees. The following example has p = 3, and pleaf = 2

Points to Note:

• Every key value must exist at the leaf level, because all data pointers are at the leaf level,

• Every value appearing in an internal node, also appears as the rightmost value in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

• When a leaf node is full, and a new entry is inserted there, the node overflows and must be split. The first j = (pleaf+1)/2 entries (in the example 2 entries) in the original node are kept there, and the remaining entries are moved to the new leaf node. The entry at position j is copied/replicated and moved to the parent node.

• When an internal node is full, and a new entry is to be inserted, the node overflows and must be split into 2 nodes. The entry at position j is moved to the parent node. The first j-1 entries are kept in the original node, and the last j+1 entries are moved to the new node.

# 10. DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key. Basic operations that can be performed on dictionary are:

1. Insertion of value in the dictionary

2. Deletion of particular value from dictionary

3. Searching of a specific value with the help of key

## 1. Linear List Representation

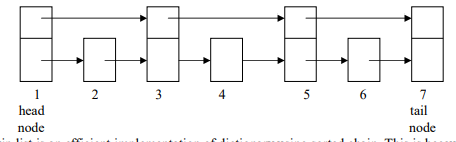
The linear list is a collection of pair and value. There are two method of representing linear list.

1. Sorted Array- An array data structure is used to implement the dictionary.

2. Sorted Chain- A linked list data structure is used to implement the dictionary

## 2. SKIP LIST REPRESENTATION

Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level. There are two special nodes in the skip list one is head node which is the starting node of the list and tail node is the last node of the list



The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time Eg:



Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the key\_val. If the node key is less than the key\_val, the search continues moving forward at the same level. If o the other hand, the node key is equal to or greater than the key\_val, the search drops one level and continues forward. This process continues until the desired key\_val has been found if it is present in the skip list. If it is not, the search will either continue at the end of the list or until the first key with a value greater than the search key is found.

### Insertion:

There are two tasks that should be done before insertion operation:

1. Before insertion of any node the place for this new node in the skip list is searched. Hence before any insertion to take place the search routine executes. The last[] array in the search routine is used to keep track of the references to the nodes where the search, drops down one level.

2. The level for the new node is retrieved by the routine randomelevel()

# 10. HASH TABLE REPRESENTATION

1 Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.

2 Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table.

3 The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

## HASH FUNCTION

1 Hash function is a function which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table.

2 The integer returned by the hash function is called hash key.

For example: Consider that we want place some employee records in the hash table The record of employee is placed with the help of key: employee ID. The employee ID is a 7 digit number for placing the record in the hash table. To place the record 7 digit number is converted into 3 digits by taking only last three digits of the key.

If the key is 496700 it can be stored at 0th position. The second key 8421002, the record of those key is placed at 2nd position in the array. Hence the hash function will be- H(key) = key%1000 Where key%1000 is a hash function and key obtained by hash function is called hash key.

## Bucket and Home bucket:

The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket. The function H(key) is home bucket for the dictionary with pair whose value is key.

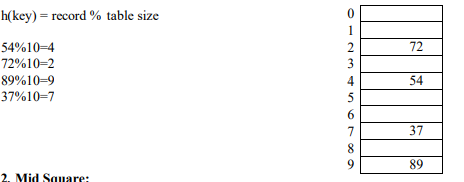
## TYPES OF HASH FUNCTION

There are various types of hash functions that are used to place the record in the hash table.

### Division Method:

The hash function depends upon the remainder of division. Typically the divisor is table length. For

eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then



2. Mid Square:

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number.

Consider that if we want to place a record 3111 then 31112 = 9678321 for the hash table of size 1000 H(3111) = 783 (the middle 3 digits)

### 3. Multiplicative hash function:

The given record is multiplied by some constant value. The formula for computing the hash key is

H(key) = floor(p \*(fractional part of key\*A))

where p is integer constant and A is constant real number.

Donald Knuth suggested to use constant

A = 0.61803398987 If key 107 and

p=50 then H(key) = floor(50\*(107\*0.61803398987)) = floor(3306.4818458045) = 3306

At 3306 location in the hash table the record 107 will be placed.

### 4. Digit Folding:

The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.

For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together

H(key) = 123+654+12 = 789

The record will be placed at location 789

### 5. Digit Analysis:

The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix, r.

Then examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

# 11. COLLISION RESOLUTION TECHNIQUES

If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.

1.Chaining

2.Open addressing (linear probing)

3.Quadratic probing

4.Double hashing

5. Double hashing

6.Rehashing

## 1. CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain.

A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

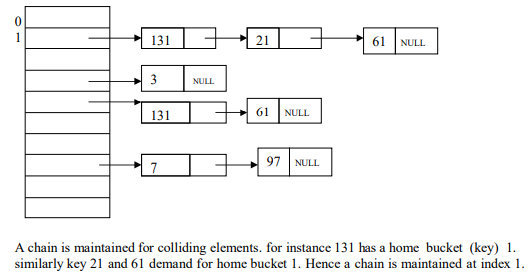
For eg; Consider the keys to be placed in their home buckets are

131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as H(key) = key % D

Where D is the size of table. The hash table will be-

Here D = 10



## 2. OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example: Consider that following keys are to be inserted in the hash table

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.

We will use Division hash function.

That means the keys are placed using the formula

H(key) = key % tablesize

H(key) = key % 10

For instance the element 131 can be placed at

H(key) = 131 % 10 = 1

Index 1 will be the home bucket for 131.

Continuing in this fashion we will place 4, 8, 7.

Now the next key to be inserted is 21.

According to the hash function

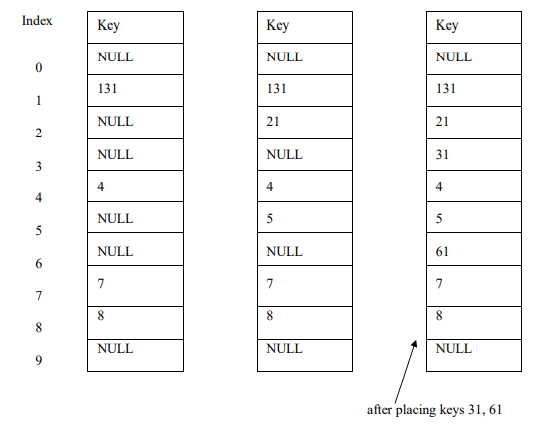
H(key)=21%10

H(key) = 1

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element.

Therefore 21 will be placed at the index 2.

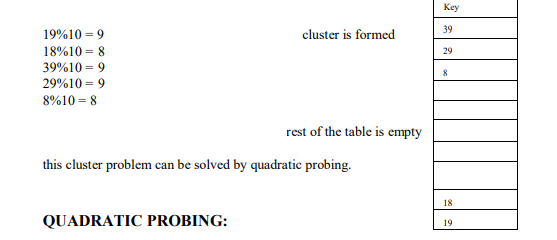
If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5



The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0th index.

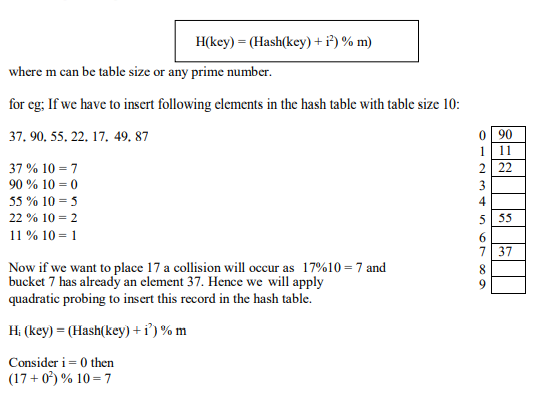
Problem with linear probing:

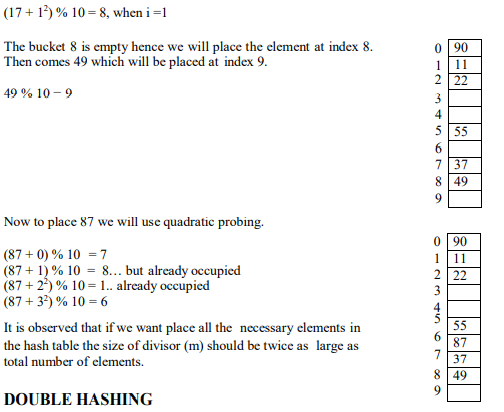
One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved



## 3. QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.





## 4. DOUBLE HASHING

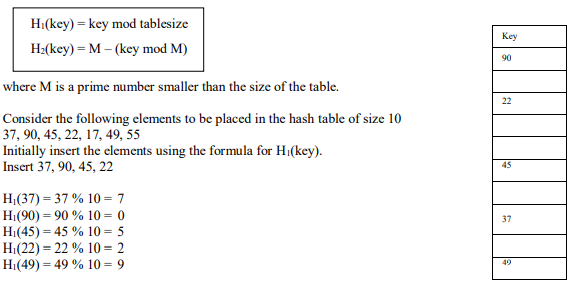
Double hashing is technique in which a second hash function is applied to the key when a collision occurs.

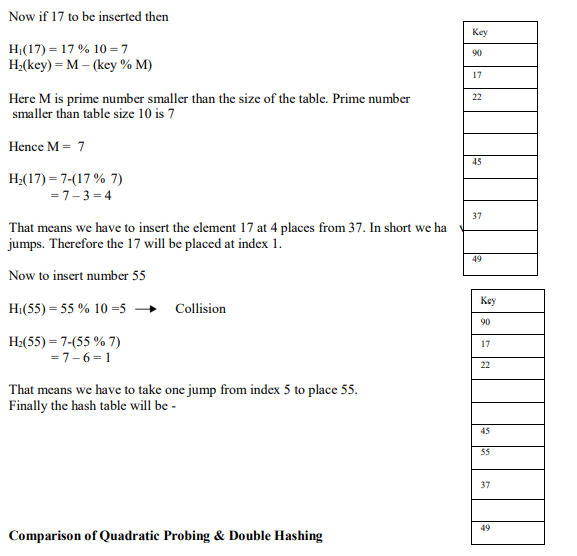
By applying the second hash function we will get the number of positions from the point of collision to insert.

There are two important rules to be followed for the second function:

• it must never evaluate to zero.

• must make sure that all cells can be probed. The formula to be used for double hashing is





### Comparison of Quadratic Probing & Double Hashing

The double hashing requires another hash function whose probing efficiency is same as some another hash function required when handling random collision.

The double hashing is more complex to implement than quadratic probing. The quadratic probing is fast technique than double hashing.

## 5. REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.

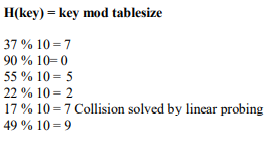
• When table is completely full

• With quadratic probing when the table is filled half.

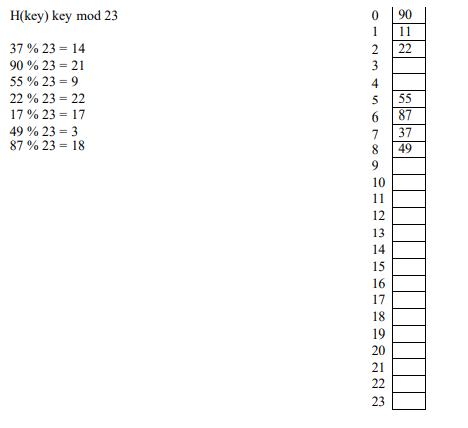
• When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,



Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be



Now the hash table is sufficiently large to accommodate new insertions.

### Advantages:

1. This technique provides the programmer a flexibility to enlarge the table size if required.

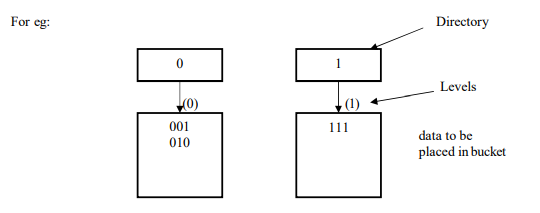
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

## 6. EXTENSIBLE HASHING

1 Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.

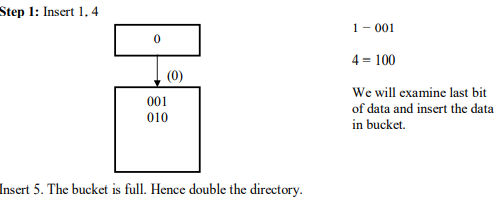
2 Extensible hashing grow and shrink similar to B-trees.

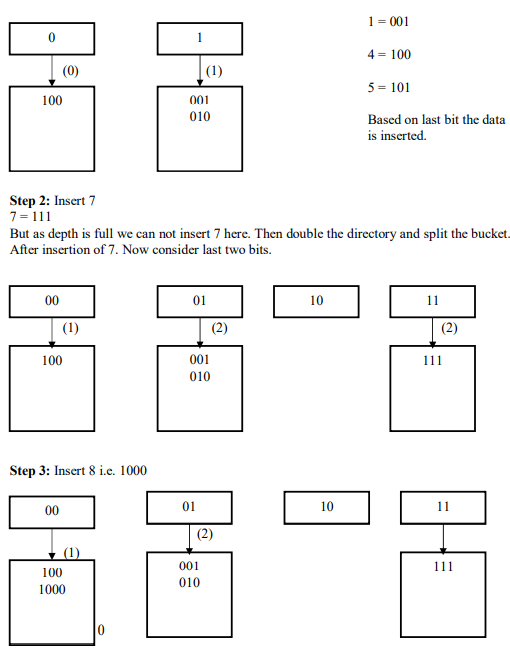
3 In extensible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in parenthesis.



The bucket can hold the data of its global depth. If data in bucket is more than global depth then, split the bucket and double the directory.

Consider we have to insert 1, 4, 5, 7, 8, 10. Assume each page can hold 2 data entries (2 is the depth).

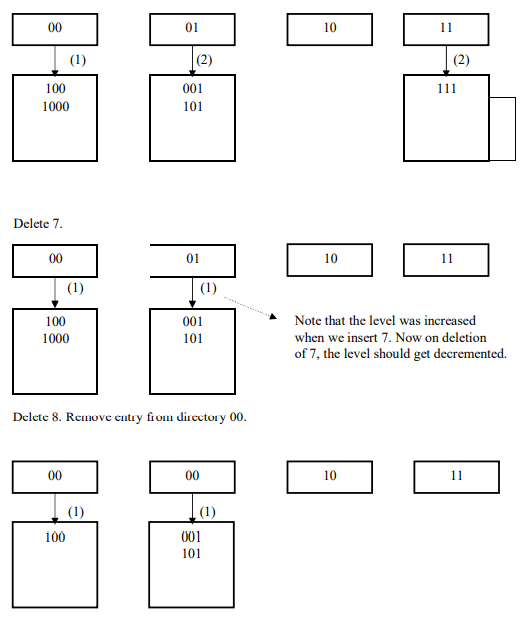




Thus the data is inserted using extensible hashing.

### Deletion Operation:

If we want t0 delete 10 then, simply make the bucket of 10 empty



## Applications of hashing:

1. In compilers to keep track of declared variables.

2. For online spelling checking the hashing functions are used.

3. Hashing helps in Game playing programs to store the moves made.

4. For browser program while caching the web pages, hashing is used.

5. Construct a message authentication code (MAC)

6. Digital signature.

7. Time stamping

8. Key updating: key is hashed at specific intervals resulting in new key