Data Structures Notes

Contributor: Abhishek Sharma

[Founder at TutorialsDuniya.com]

Computer Science Notes

INDEX

Data Structures

- Data structure is a method of organising large amount of data more efficiently so that any operation on that data becomes easy.

* Types

1. Linear Data Structure: A Data Structure organised the data in sequential order.

Ex: 1) Arrays
2) Linked List
3) Stack
4) Queue

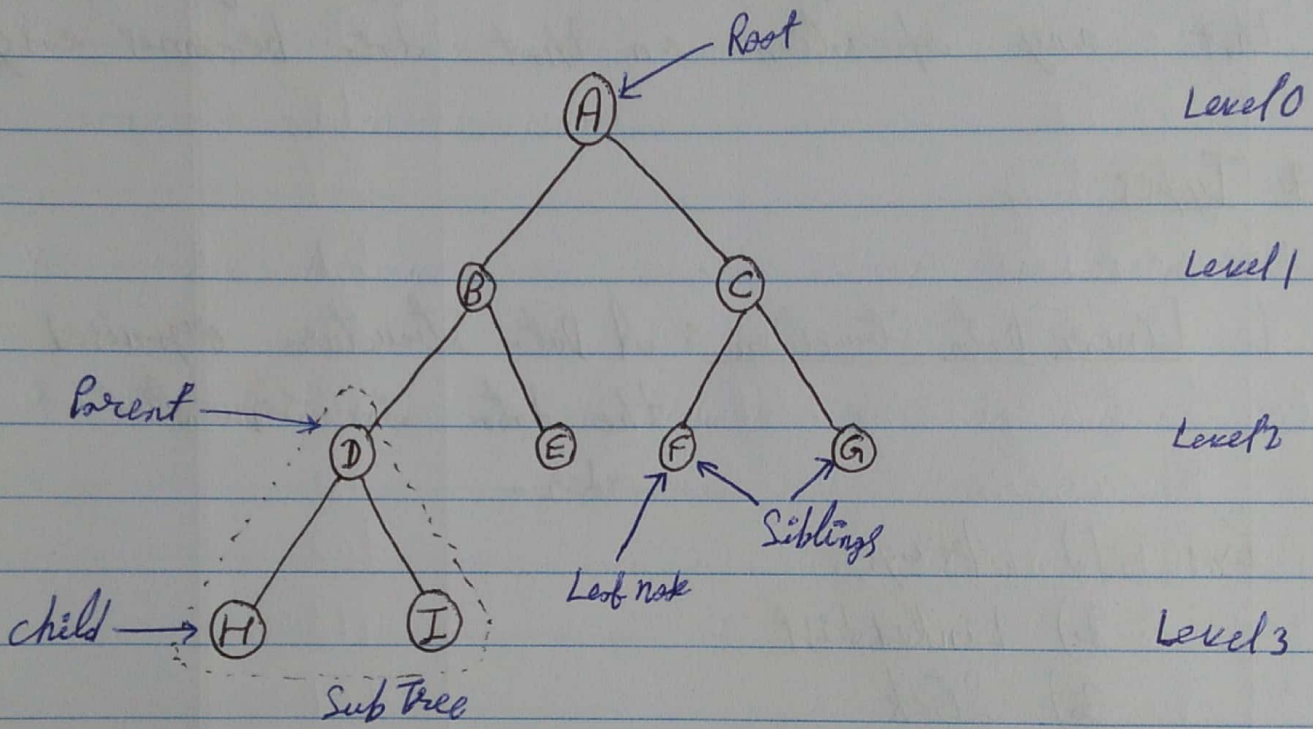
2. Non-Linear Data Structure: A data structure organising the data in Random order.

Ex: 1) Trees
2) Graphs

(2)

TREE

- Tree is a non linear data structure which organises data in hierarchial structure.



Each element of a Tree is called Node.

Root: Node at top of Tree.
There is only one root per Tree & one path from root to any node.

Parent: Node which is predecessor of any node.

Child: Node which is descendent of any node.

Leaf: Node which does not have any child.

Subtree: Each child from a node forms a subtree.

Traversing: Passing through nodes in a specific order.

Siblings: Nodes which belong to same parent.

Level: Total no. of edges from root to a leaf in longest path.

Depth: Total no. of Nodes from root to a leaf in the longest path.

Keys: Key represents a value of a node based on which a search operation is to be carried out for a node.

- Binary Tree: A tree in which every node can have a maximum of 2 children.

- In a binary tree, every node can have either ~~maximum~~ of 0 children or 1 child or 2 children but not more than 2 children.

(4)

- **Strictly Binary Tree:** A binary tree in which every node has either 0 or 2 number of children.

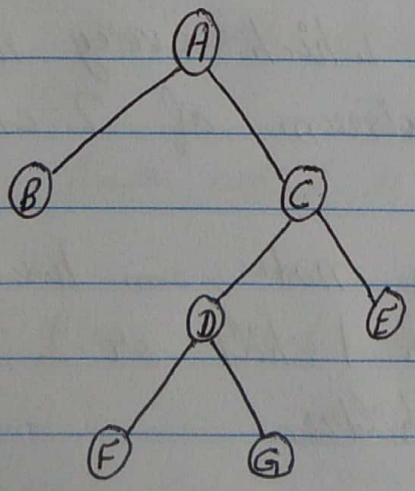
- A strictly binary tree with n leaves always contains $2n-1$ nodes.

* Strictly binary tree is also called Full Binary Tree.

- **Complete Binary Tree:** A binary tree in which every internal node has exactly 2 children and all leaf nodes are at same level.

- At level d, tree contains 2^d leaves and $2^d - 1$ non leaf nodes.

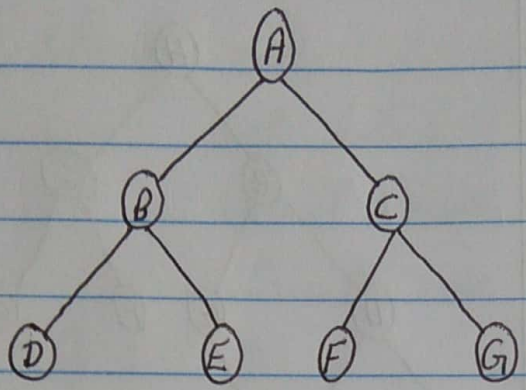
* Complete Binary Tree is also called Perfect Binary Tree.



Level = 3
depth/height = 4

Strictly Binary Tree

Not complete Binary Tree



Level : 2
depth / height = 3

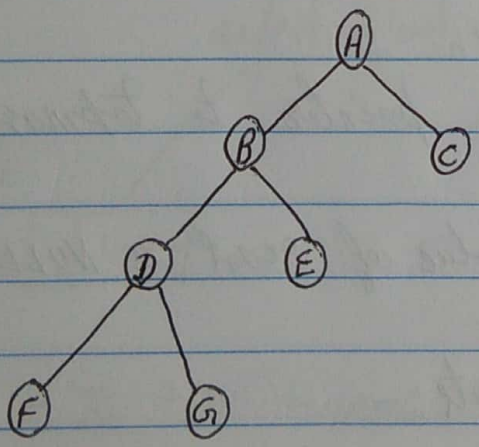
Strictly Binary Tree
Complete Binary Tree

- **Almost Complete Binary Tree**

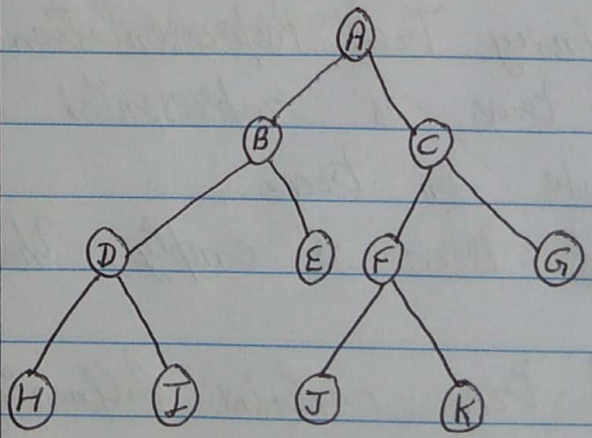
A binary tree of depth d is an almost complete binary tree if:

(i) Any node 'nd' at level less than $d-1$ has 2 childs.

(ii) For any node 'nd' in tree with a right descendant at level d, 'nd' must have a left child and every left descendant of 'nd' is either a leaf at level 'd' or has 2 children.

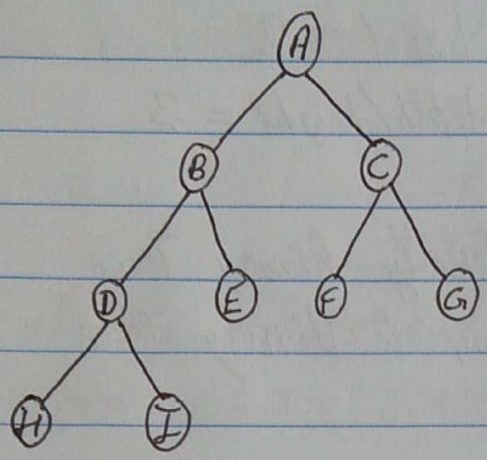


Not Almost Complete
(Leaf at level 1, [C])
Strictly binary Tree

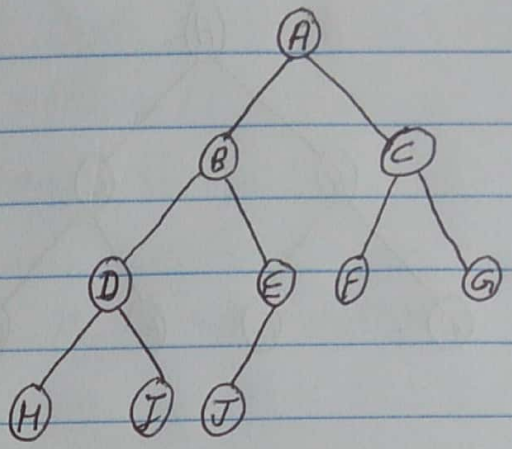


Not almost complete
[Leaf at level 2 (E)]
Strictly Binary Tree

(6)



Almost complete Binary Tree
Strictly Binary Tree



Almost complete Binary Tree
Not Strictly binary Tree

⇒ An almost complete strictly binary tree with 'n' leaves has $2n-1$ leaves.

⇒ An almost complete binary tree with 'n' leaves that is not strictly binary has $2n$ nodes.

★ Binary Tree Representation

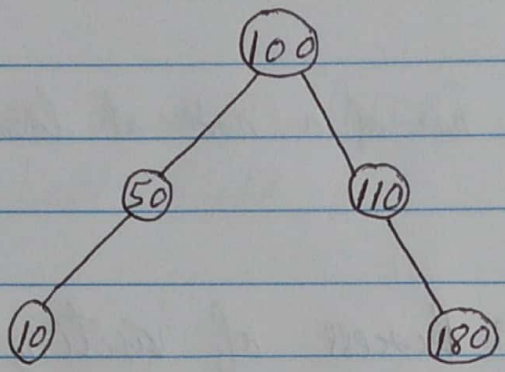
A tree is represented by a pointer to topmost node in tree.

- If tree is empty, then value of root is NULL.

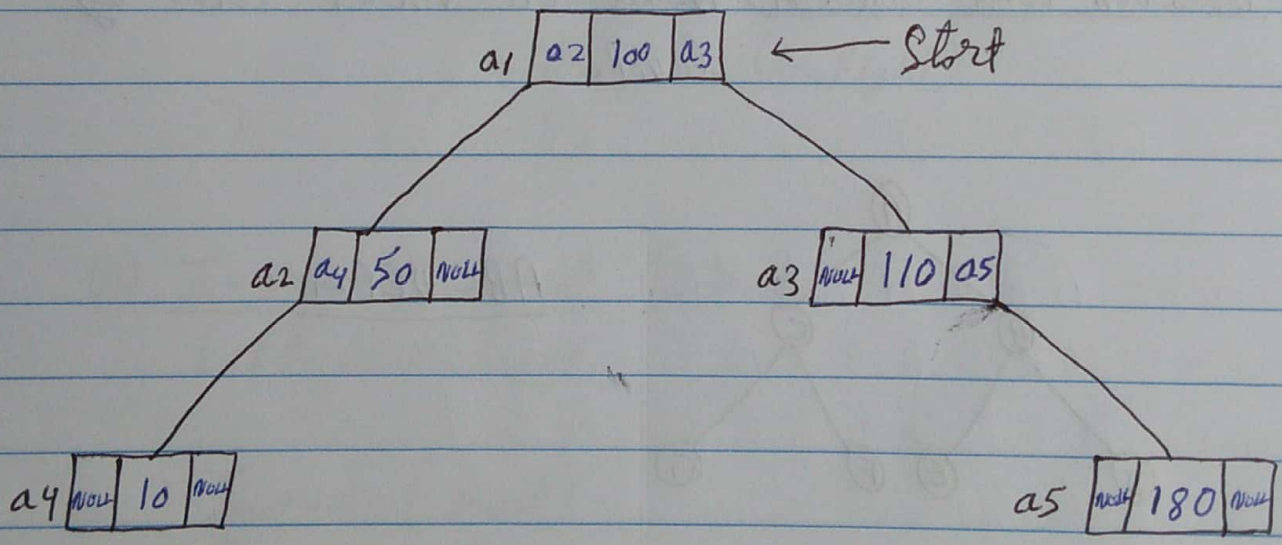
⇒ A tree contains following parts:-

(i) Data
(ii) Pointer to left child
(iii) Pointer to right child

```
struct node
{       int data;
        struct node *left;
        struct node *right;
};
```



1) Linked Representation



- In this representation, we can only move from top to bottom.
- We can not move from bottom to top.

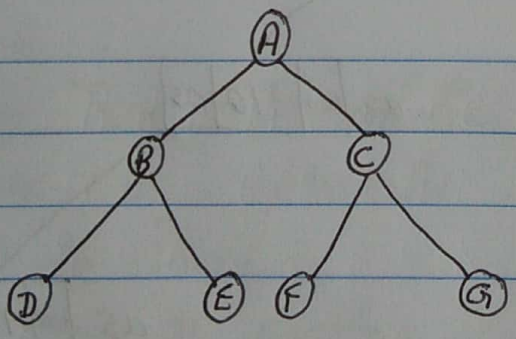
2) Sequential Representation

	1	2	3	4	5	6	7
Tree	100	50	110	10			180

- Left and Right position of a node at location K are 2*K and 2*K+1.
- Root node location no. of a node at location K = $\lceil K/2 \rceil$

★ **Tree Traversal**: Process of visiting each node exactly once.

1) **Breadth First Traversal**: Nodes are visited level by level.



ABCDEFG

2) Depth First Traversal

In depth first traversal of a tree for every node, first left subtree is visited then right subtree is visited.

- Based on order in which root node is visited there are 3 possible traversal of binary tree:-

(i) Pre-order Traversal [NLR]

Node, Left, Right

```
Preorder (N)
{       if (N≠NULL)
            print (N.key)
            Preorder (N.left)
            Preorder (N.Right)
}
```

(ii) In-order Traversal [LNR]

Left Node Right

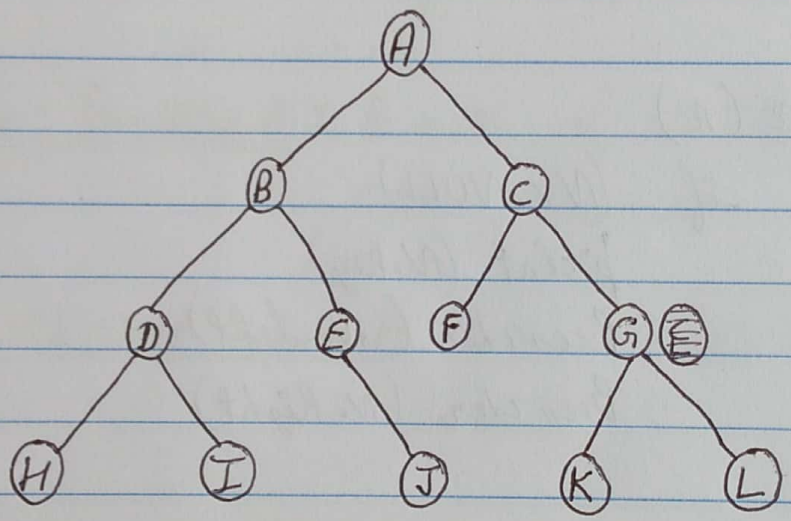
```
Inorder (root)
{       if (root≠NULL)
            Inorder (root→left)
            print (root→data)
            Inorder (root→right)
}
```

(10)

(iii) Post-order Traversal [LRN]
Left Right Node

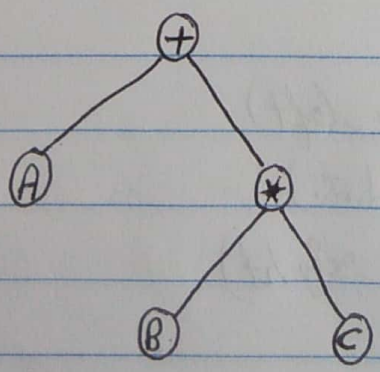
```
Postorder(root)
{  if (root≠NULL)
       Postorder(root→left)
       Postorder(root→right)
       print(root→data)
}
```

Ex=



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

Ex=



Preorder: +A*BC
Inorder: A+B*C
Postorder: ABC*+

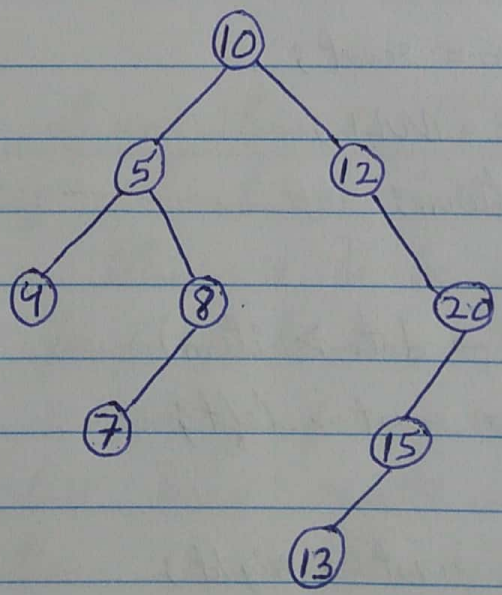
☆ Binary Search Tree [BST]

Binary Search Tree is a binary tree in which for any node 'n', all values in its left subtree will be lesser than n and all values in its right subtree will be greater than n.

⇒ No value in BST will be repeated.

Ex= Construct a BST by inserting the following sequence of numbers.

10, 12, 5, 4, 20, 8, 7, 15, 13



⇒ If a BST is traversed in-order, the output will produce key values in Ascending order.

In-order : 4, 5, 7, 8, 10, 12, 13, 15, 20.

☆ Searching an item in BST

Whenever an item is to be searched, we start searching from root node.

- If item is less than root's data, search for item in root's left subtree.

Otherwise, search for item in right subtree.

```
int bst_search(root, item)
{   if (root == NULL)
    {   loc = Null;
        par = NULL;
        return;
    }
    else if ( root→data == item)
    {   loc = root;
        par = NULL;
        return;
    }
    if ( root→data > item)
        ptr = root→left;
    else
        ptr = root→right;
    save = root;
    while ( ptr ≠ NULL)
    {   if ( ptr→data == item)
```

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

```
{       loc = ptr;
        par = save;
        return;
    }
    save = ptr;
    if ( ptr → data > item)
        ptr = ptr → left;
    else
        ptr = ptr → right;
}
loc = NULL;                    //unsuccessful search
par = save;
return;
}
```

☆ Insertion of an item in BST

A new key is always inserted at Leaf.

- Whenever an item is to be inserted, we first find its proper location.
- We start searching from root node, Then if item is less than root's data. We search for empty location in root's left subtree and insert the item.
- Otherwise, we search for an empty location in root's right subtree and insert the item.

(14)

```
void bst_insert (root, item)
{
        bst_search (root, item);
    if (loc ≠ NULL)
    { cout << "Item already exists";
      return;
    }
    newnode = getnode();
    newnode → data = item;
    newnode → right = NULL;
    newnode → left = NULL;
    if (par == NULL)
    {   root = newnode;
        return;
    }
    else if (par → data > item)
            par → left = new node;
    else
            par → right = newnode;
    return;
}
```

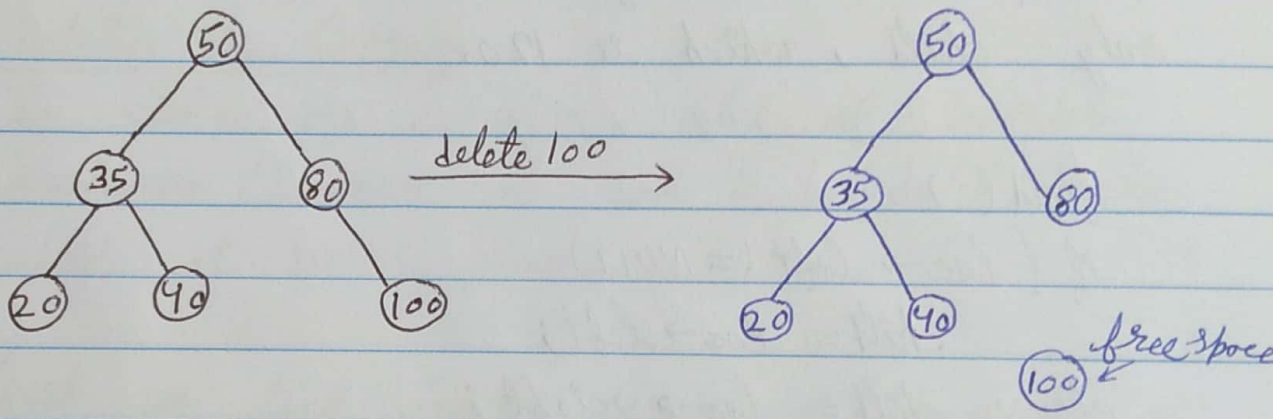
- Time complexity of :-
  (i) Searching — O(n)
  (ii) Insertion — O(1)
  (iii) Deletion — O(n)

☆ Deletion of an item from BST

1) Node to be deleted is leaf

Since the node is a leaf, it has no children.

- The appropriate pointer of its parent is set to NULL and the node is disposed off by delete.

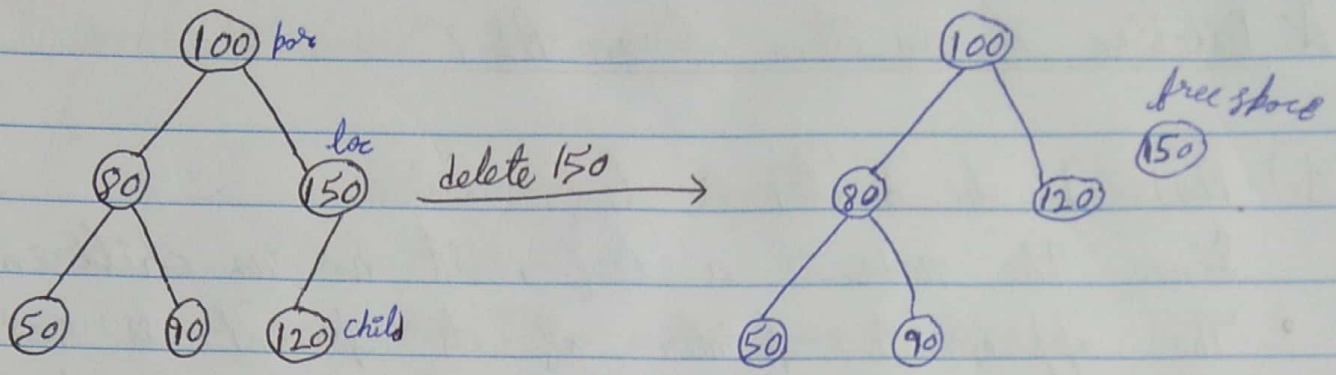



```
find()
if (loc->left==NULL && loc->right==NULL)
{       if (loc = par->left)
                par->left = NULL;
        else    par->right = NULL;
}
```

2) Node to be deleted has only 1 child

The parent's pointer to node is reset to point to node's child.

(16)



- Node 150 is deleted by setting right pointer of its parent containing 100 to point to 150's only child, which is 120.

```
find ( )
if ( loc→ left != NULL)
        child = loc→ left;
else    child = loc→ right;

if ( par ≠ NULL)
{       if  (loc == par→ left)
                par→ left = child;
        else  par→ right = child;
}
else  root = child ;
return;
```

3) Node to be deleted has 2 children

Since the node has 2 children, so no one step operation can be performed because parent's right or left pointer cannot point to both the nodes children at some time.

- We can delete the node having 2 children by 2 ways:-

(i) Deletion by Copying

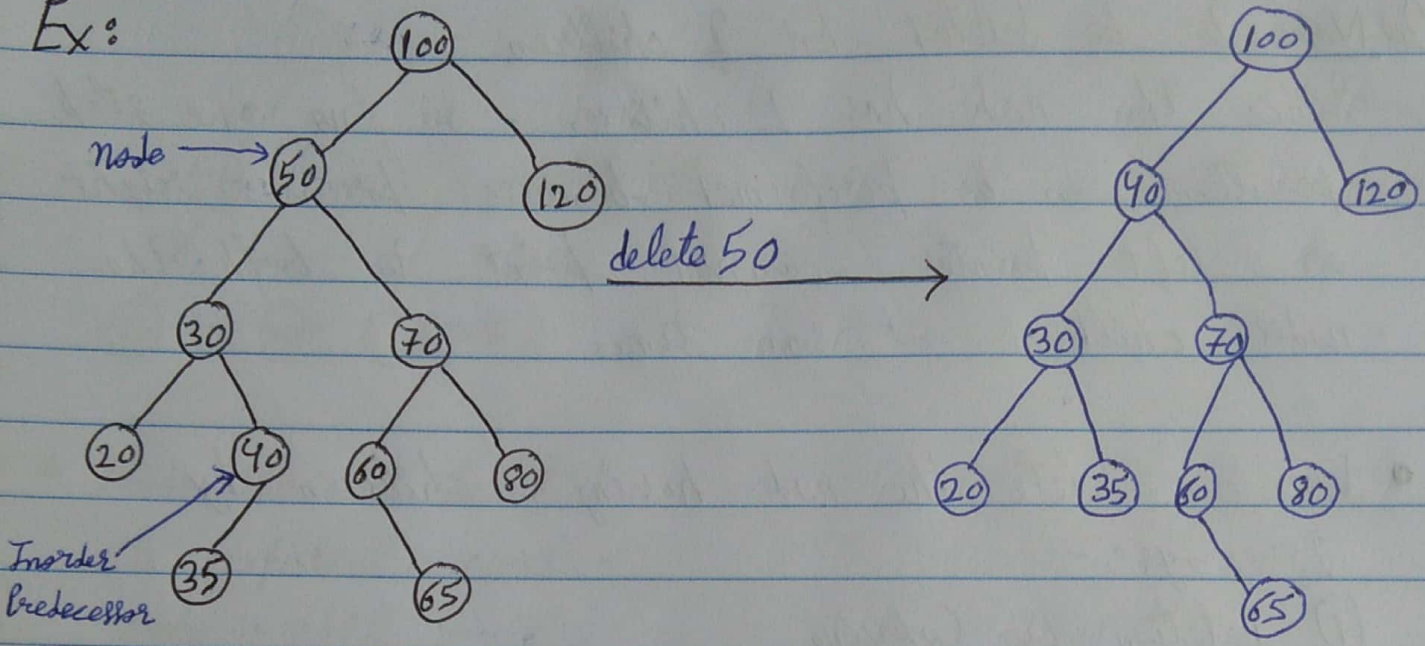
In deletion by copying, value of inorder predecessor/Successor of node to be deleted are copied at node's place.

- First we find inorder predecessor/Successor of node. Then copy value of inorder predecessor/Successor at key value of node & then delete inorder predecessor/successor.

⇒ In deletion by copying, height of tree remain balanced.

⇒ While in deletion by merging may result in increasing the height of tree. Sometimes the height may be reduced.

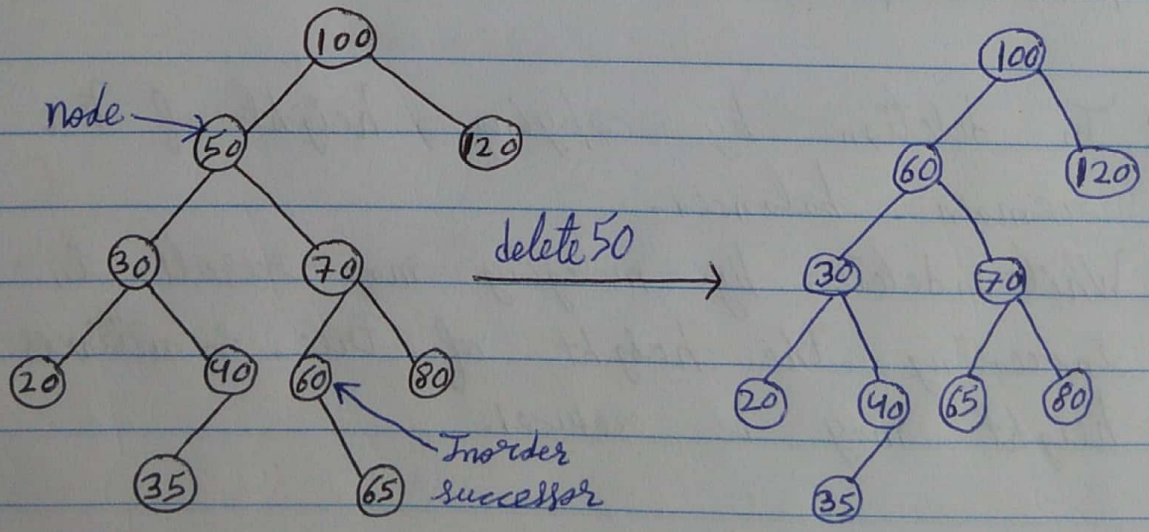
Ex:



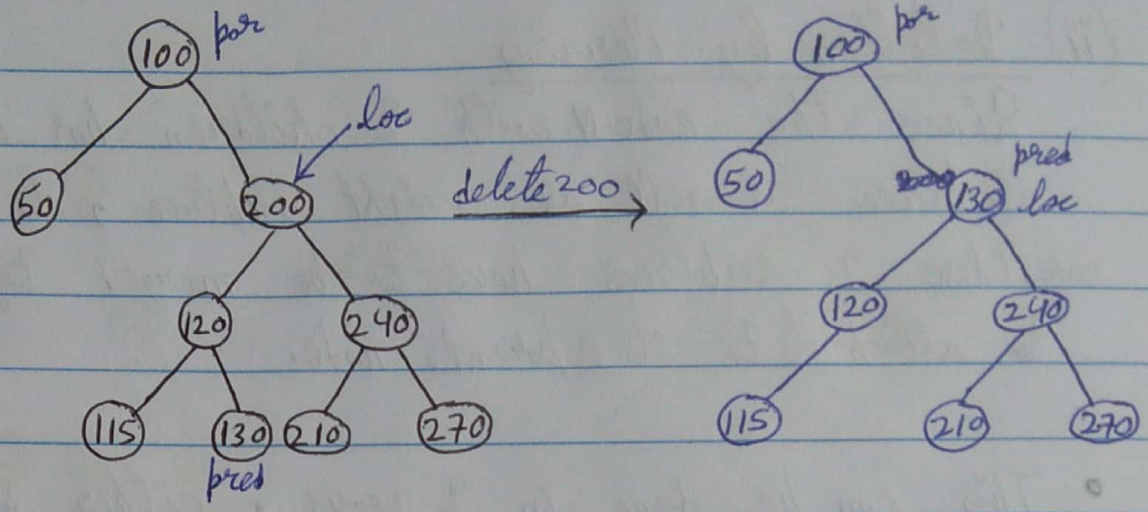
Deletion by Copying [Inorder predecessor]

- To delete node 50, we copy its inorder predecessor that is 40 at place of 50 & then delete 40. If its [50's] inorder predecessor i.e. 40 has any left child, then the inorder predecessor's parent will point to inorder predecessor's child.

Ex:



Deletion by Copying [Inorder Successor]



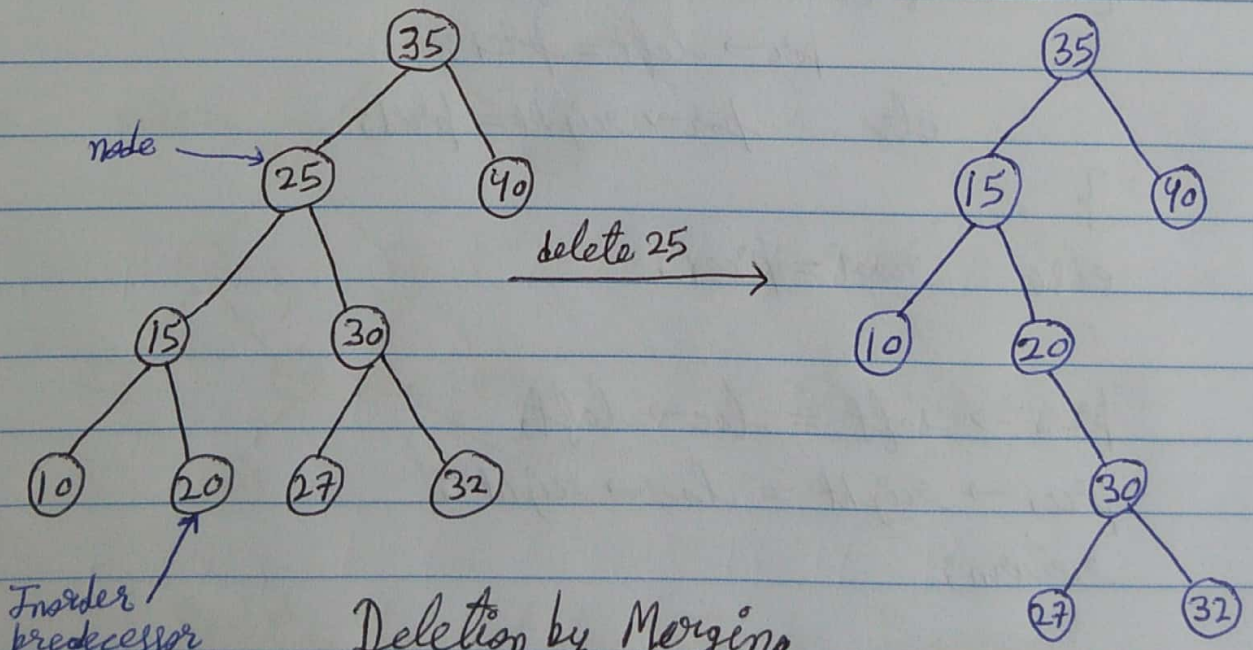
Deletion by copying
[Inorder predecessor]

```
find()
ptr = loc→left;
~~save = loc;~~
while (ptr→right != NULL)
{  pred ~~save~~ = ptr;
      ptr = ptr→right;
}
~~pred = ptr;~~
if (par ≠ NULL)
{      if (loc == par→left)
          par→left = pred;
       else    par→right = pred;
}
else    root = pred;

pred→left = loc→left;
pred→right = loc→right;
return;
```

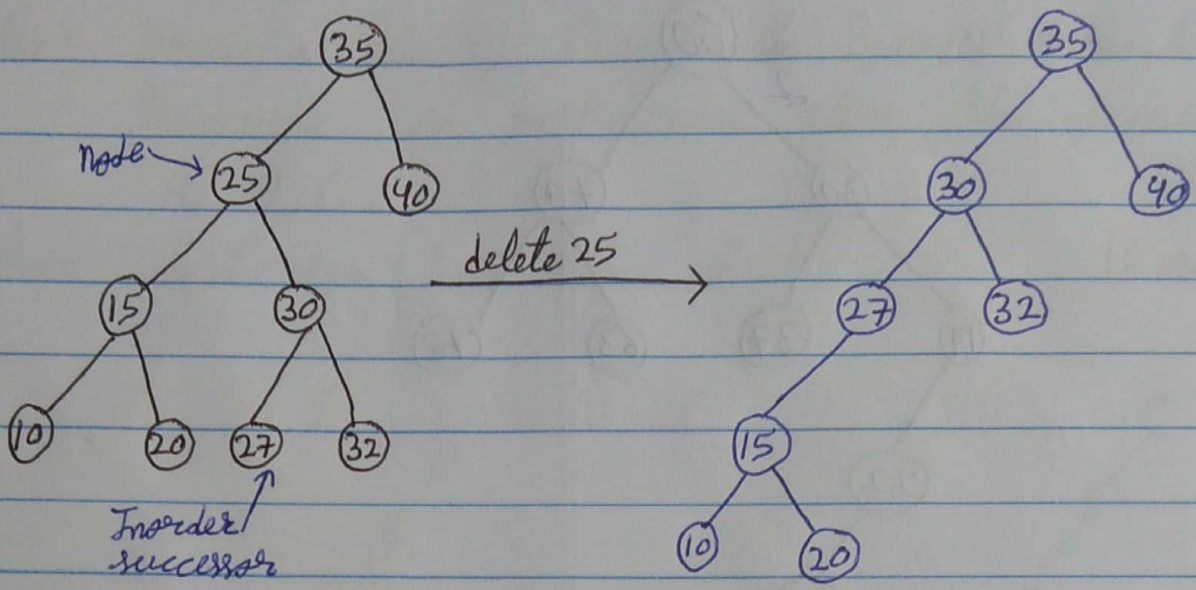
(ii) Deletion By Merging

Since the node N with 2 children has a left subtree as well as right subtree, to delete this node, these 2 subtrees have to be merged together to attach to its parent node.

- This can be done in 2 ways, either we merge right subtree into left subtree or we merge left subtree into right subtree.
- To merge a right subtree into a left subtree we find the largest value in left subtree & make it a parent of right subtree of node N.
- The largest value in left subtree will be rightmost child in the subtree.



Deletion by Merging [Inorder predecessor]



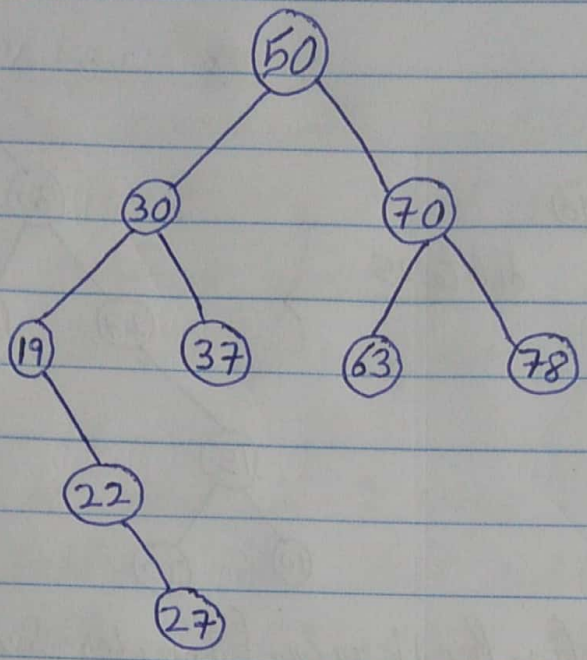
Deletion By Merging [Inorder Successor]

- In order to merge left subtree into right subtree find smallest value in right subtree & make it a parent of left subtree.
The smallest value in a right subtree will be leftmost node in the subtree.

Ex = Preorder traversal of a BST is
Preorder: 50 30 70 37 78 19 63 22 27
Write Postorder Traversal ?

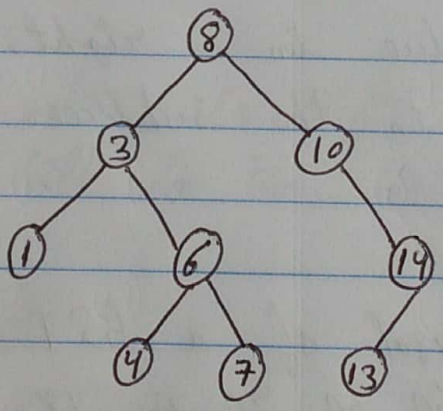
Ans = In BST, the inorder traversal is simply data in ascending order.
So, Inorder: (19 22 27 30 37) 50 (63 70 78)
Preorder: 50 30 70 37 78 19 63 22 27

22



Postorder: 27 22 19 37 30 63 78 70 50

Ex= Give the preorder, inorder and postorder traversals? [3 marks]



Ans= Preorder: 8 3 1 6 4 7 10 14 13

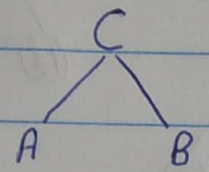
Inorder: 1 3 4 6 7 8 10 13 14

Postorder: 1 4 7 6 3 13 14 10 8

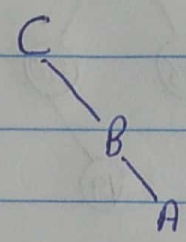
Ex= Drow a binary tree with 3 nodes which when traversed in post order gives the sequence A, B, C.

Ans= Postorder [Left Right Root] [5 marks]

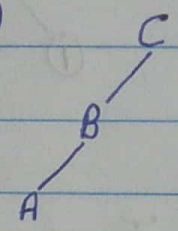
(i)



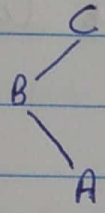
(ii)



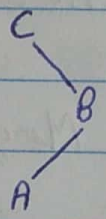
(iii)



(iv)

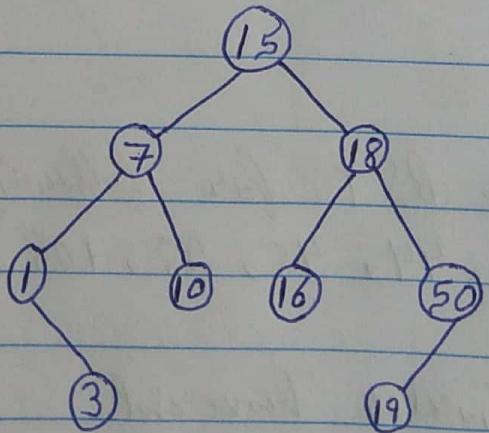


(v)



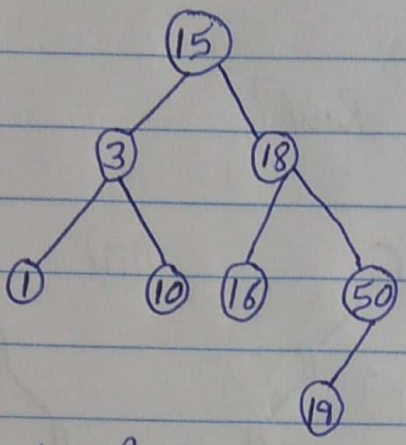
Ex= Create a Binary Search Tree using following data:
15, 7, 1, 18, 50, 19, 3, 10, 16
Perform deletion of node 7 using
(i) deletion by copying
(ii) deletion by merging [5 marks]

Ans=

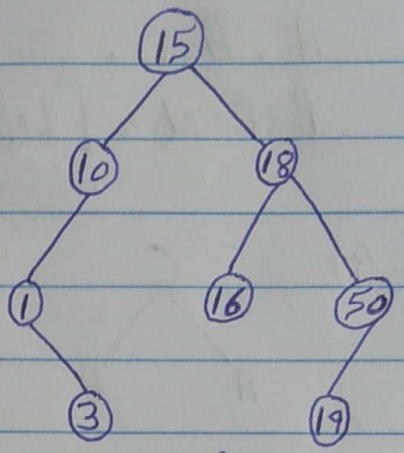


BST

(i) Delete 7 by Copying

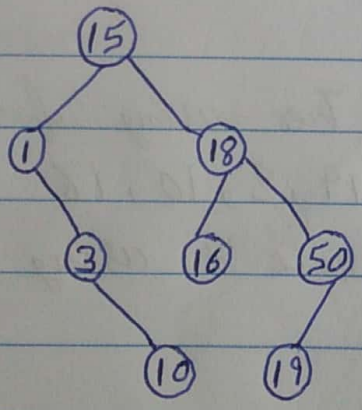


Inorder Predecessor

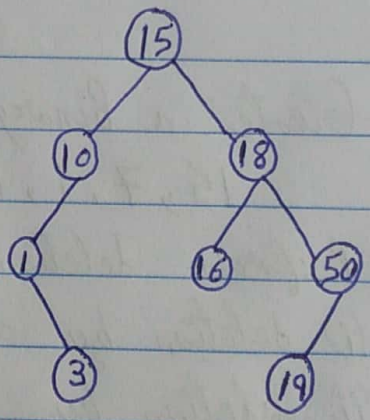


Inorder Successor

(ii) Delete 7 by Merging



Inorder Predecessor



Inorder Successor

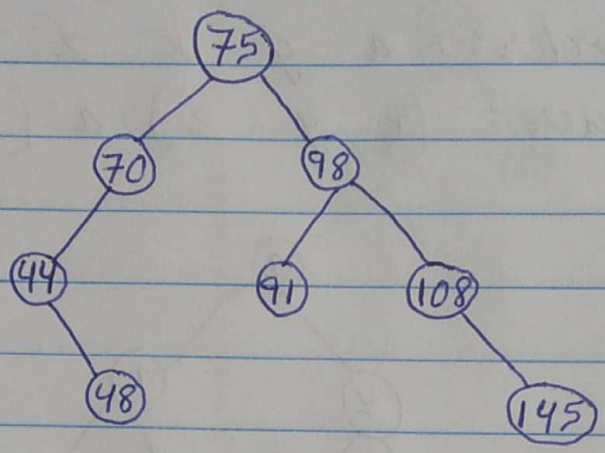
2016

Ex= Construct a BST for following keys:
75, 70, 44, 48, 98, 108, 91, 145

Show: (i) Inorder Traversal
(ii) Postorder Traversal

(iii) Tree after deleting 98. Use deletion by merging.

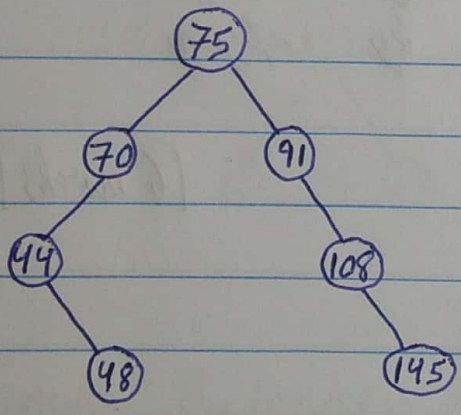
Ans = (i) ~~Inorder~~ [5 marks]



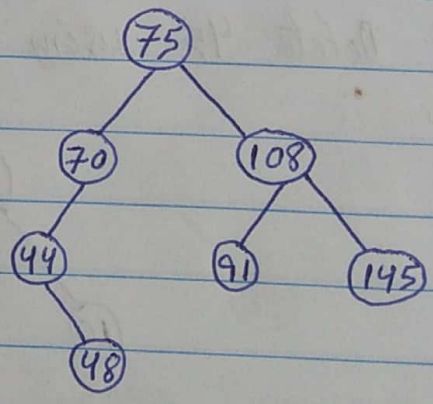
(i) Inorder : 44 48 70 75 91 98 108 145

(ii) Postorder : 48 44 70 91 145 108 98 75

(iii) delete 98 by merging,



Inorder Predecessor



Inorder Successor

(26)

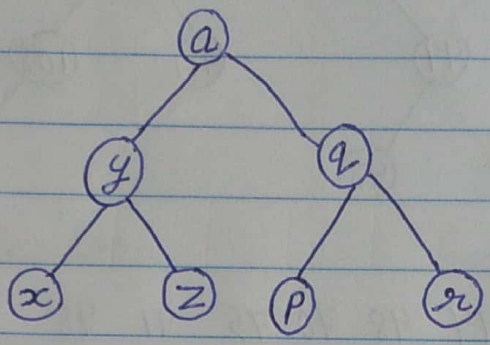
2017

Ex= Construct a binary tree whose following traversals are given:

Preorder: a y x z q P r

Inorder: (x y z) a (P q r)

Ans= [2 marks]



2017

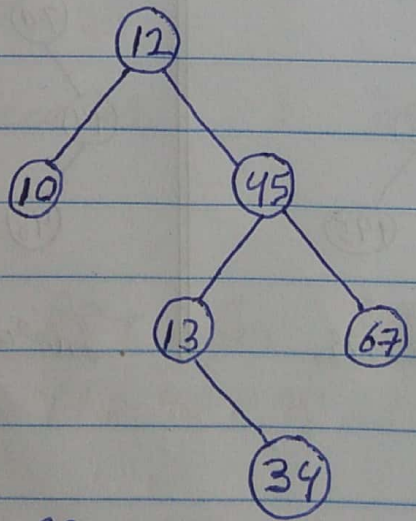
Ex= Create a BST using following values:

12, 45, 13, 67, 10, 34

(i) Delete 12 using deletion by merging

(ii) Delete 45 using deletion by copying

Ans= [6 marks]



BST

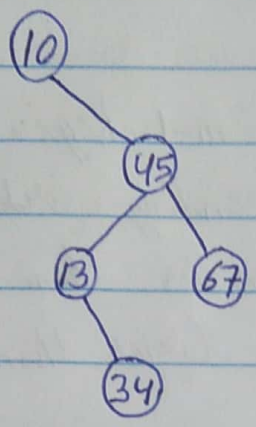
TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

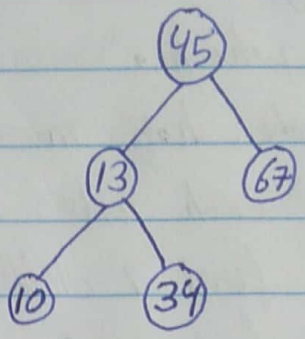
- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

(i) Delete 12 by merging,

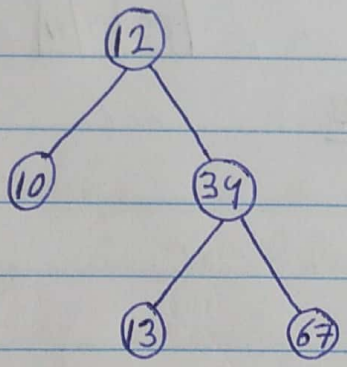


Inorder Predecessor

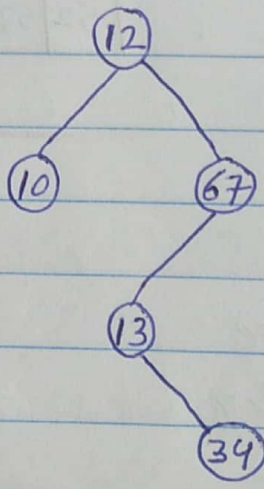


Inorder Successor

(ii) Delete 45 using copying,



Inorder Predecessor



Inorder Successor

☆ Multiway Trees

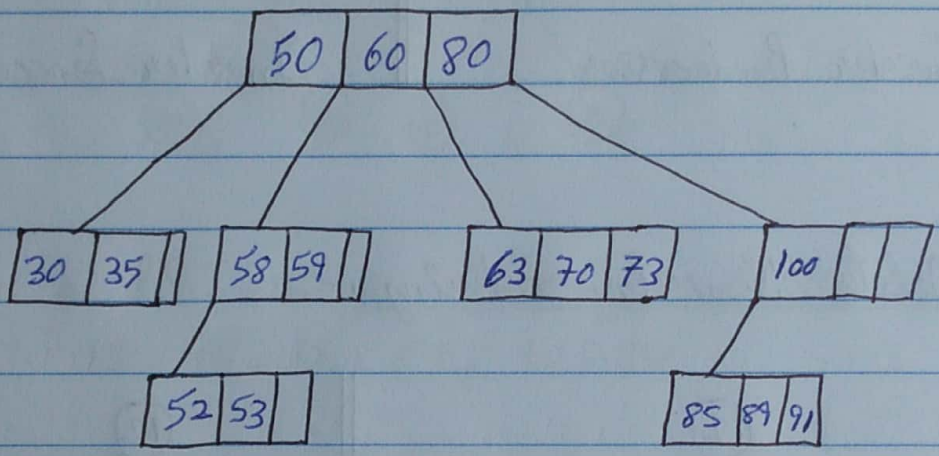
A multiway search tree of order m is a tree in which,

(i) Each node has m children & m-1 keys.

(ii) Keys in each node are in ascending order.

(iii) Keys in first i children are smaller than i^{th} key.

(iv) Keys in last m-i children are larger than i^{th} key.



4-way Tree

Types

1. B tree
2. B^+ tree

☆ B Trees

B tree is a self-balanced tree with multiple keys in every node and more than 2 children for every node.

☆ Properties of B tree

1. All leaves must be at some level.
2. All nodes except root must have atleast $\lceil m/2 - 1 \rceil$ keys & maximum of m-1 keys.
3. All key values within a node must be in Ascending order.
4. A non leaf node with 'n' keys must have 'n' number of children.
5. All non leaf nodes except root [i.e. internal nodes] must have atleast m/2 children.
6. If the root node is a non leaf node, then it must have at least 2 children.
7. Root node may contains minimum 1 key

☆ Searching In a B tree

Searching in a B tree is similar to that of BST.

- Instead of choosing b/w a left & right child as in a binary tree, a B tree search must make an n-way choice where n is the total no. of children that node has.
- In a B tree also, the search starts from the root node.
- The correct child is chosen by performing a liner search of values in node.

- After finding value greater than the desired value, left child pointer is followed.

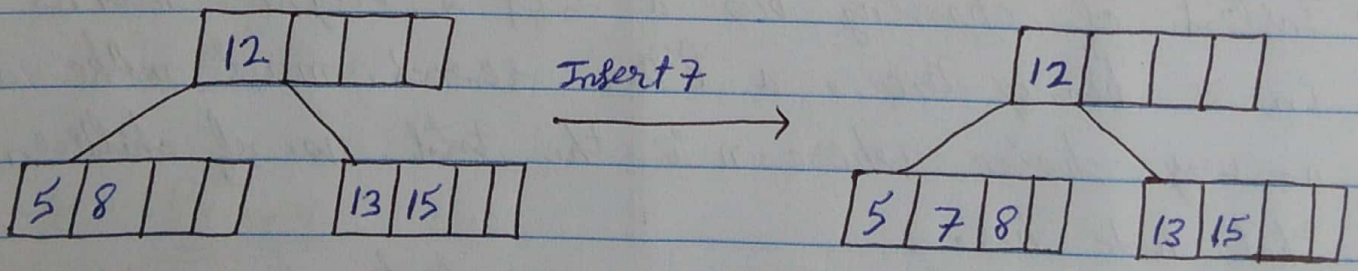
- Since the running time of search operation depends upon height of tree, search time complexity of B tree search is $O(\log n)$.

☆ Inserting a key into a B tree

In a B tree, the tree is build from bottom up & we have to keep all leaves at some level.

First a search for proper location is made & then insertion is done at that position.
Every new key is first inserted in leaf & then adjustments are done to upper level.

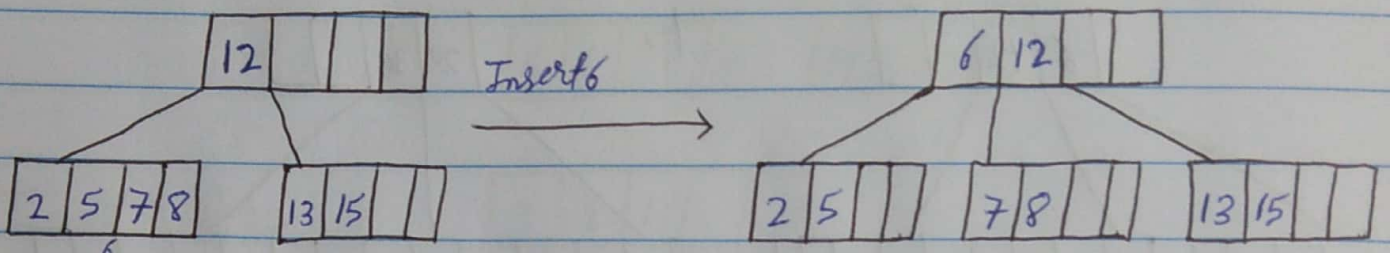
- **Inserting a key into a non-full leaf**



- **Leaf in which a key should be placed is full**

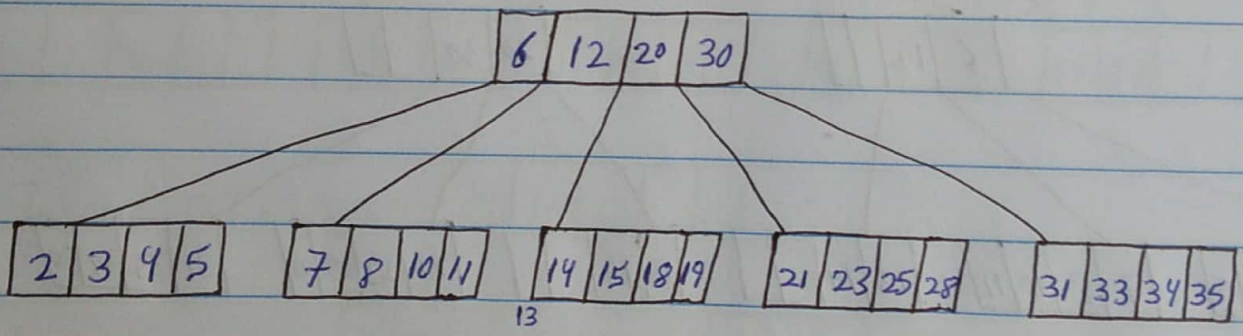
In this case, the leaf is split, creating a new leaf & half of keys are moved from full leaf to new leaf.

The middle key is moved to parent and a pointer to new leaf is placed in parent as well.

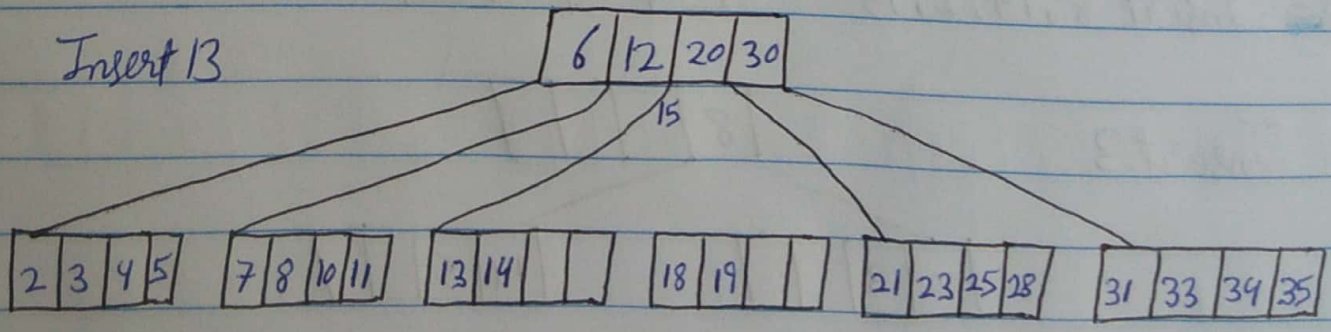


- Root of B tree is full
  In this case, a new root and a new sibling of existing root have to be created.

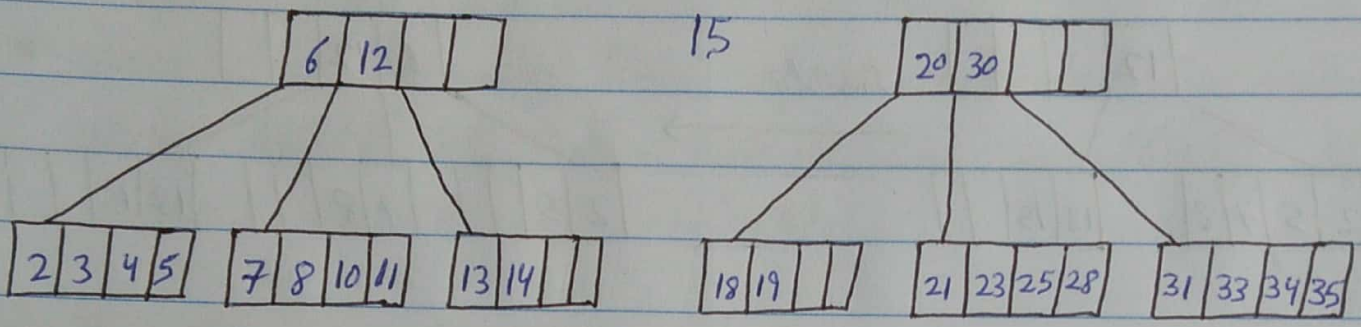
- After inserting key 13 in third leaf,



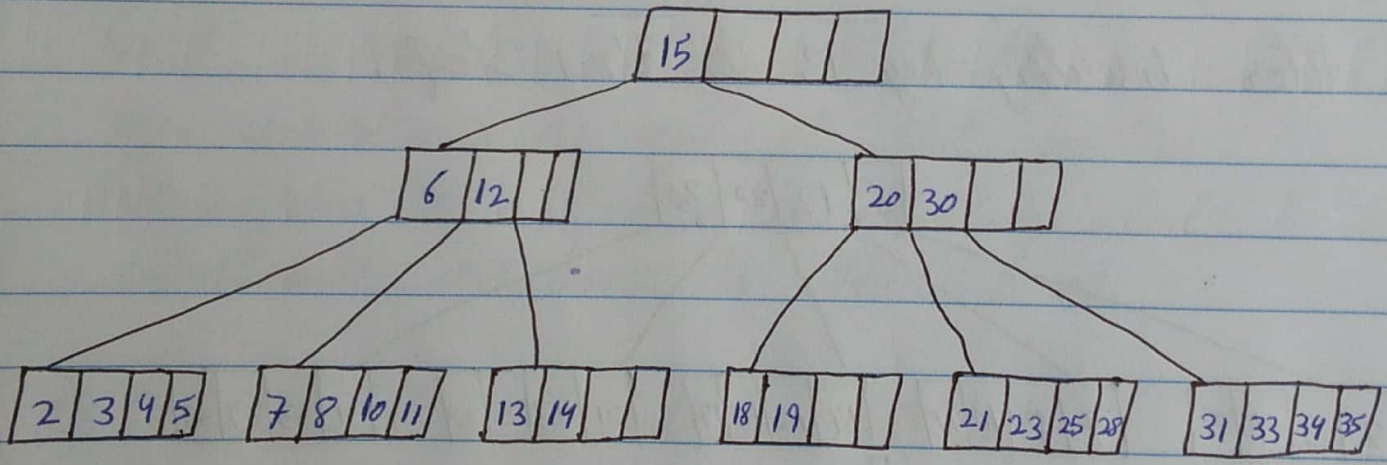
- The leaf is split, a new leaf is created & key 15 is about to be moved to parent but parent has no room for it.



So the parent is split but now 2 B trees have to be combined into one.

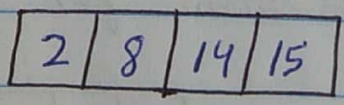


This is achieved by creating a new root & moving middle key to it.

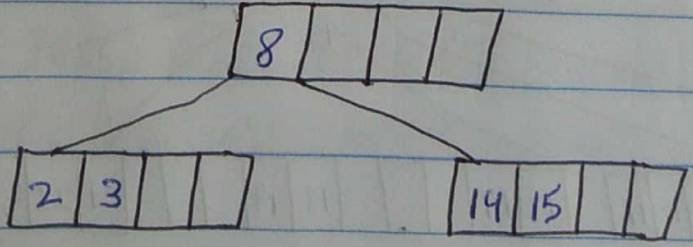


Ex: Insert 8, 14, 2, 15, 3, 1, 16, 6, 5, 27, 37, 18, 25, 7, 13, 20, 22, 23, 24 into a B tree of order 5.

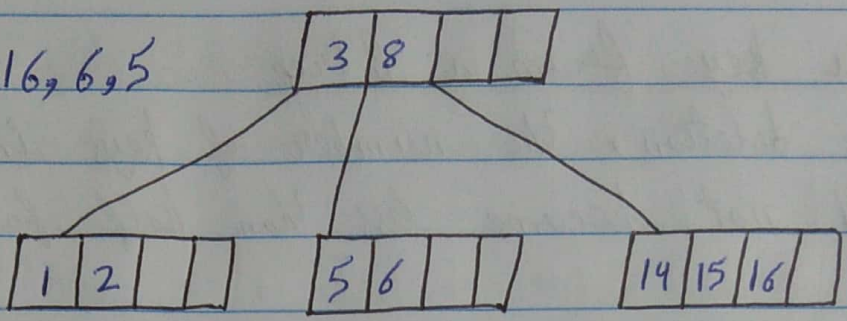
Ans= Insert 8, 14, 2, 15



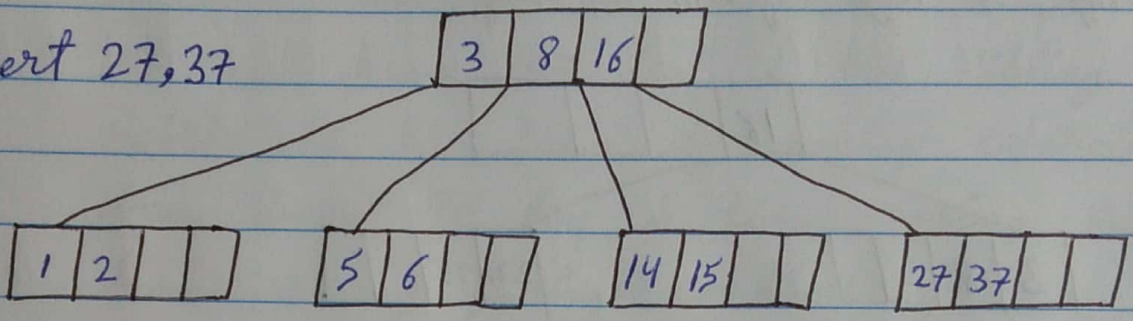
Insert 3



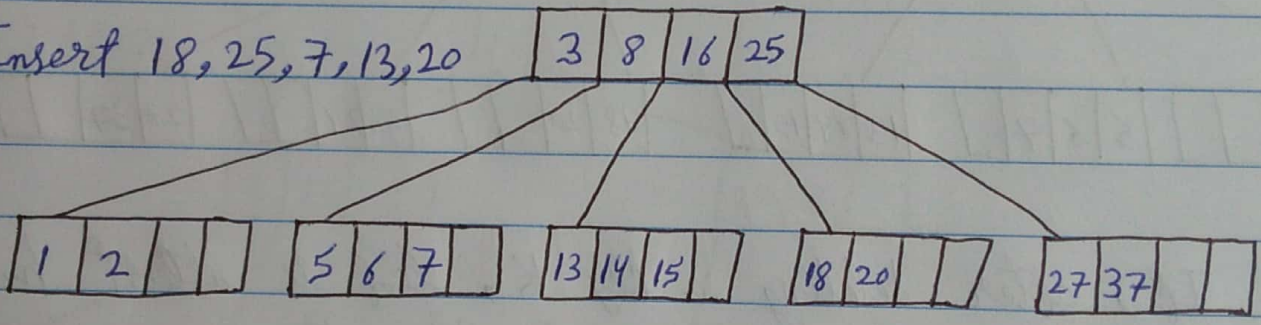
Insert 1,16,6,5



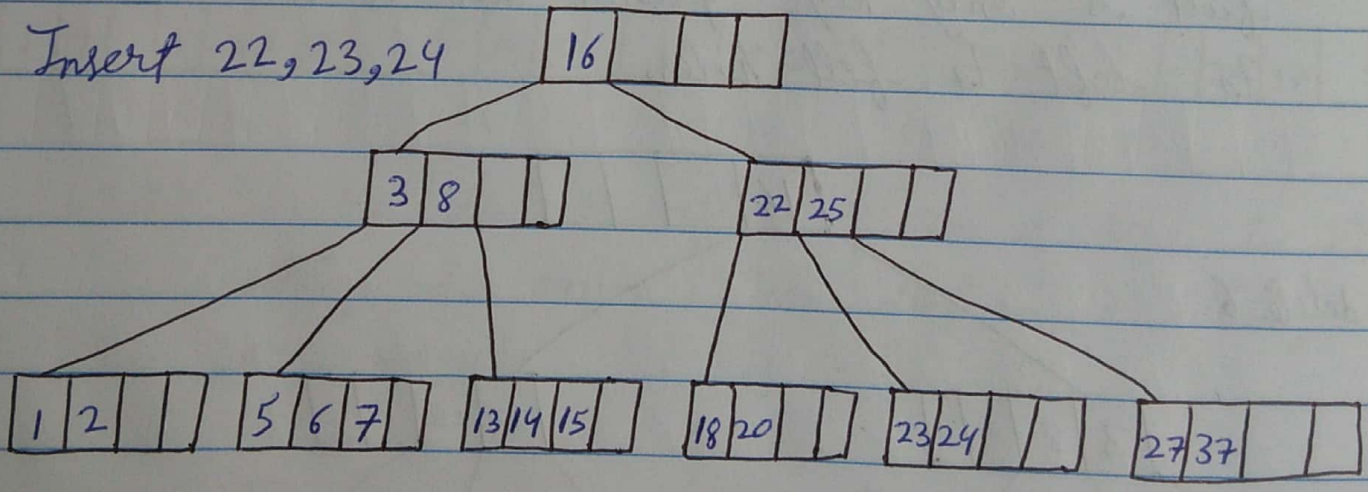
Insert 27,37



Insert 18,25,7,13,20



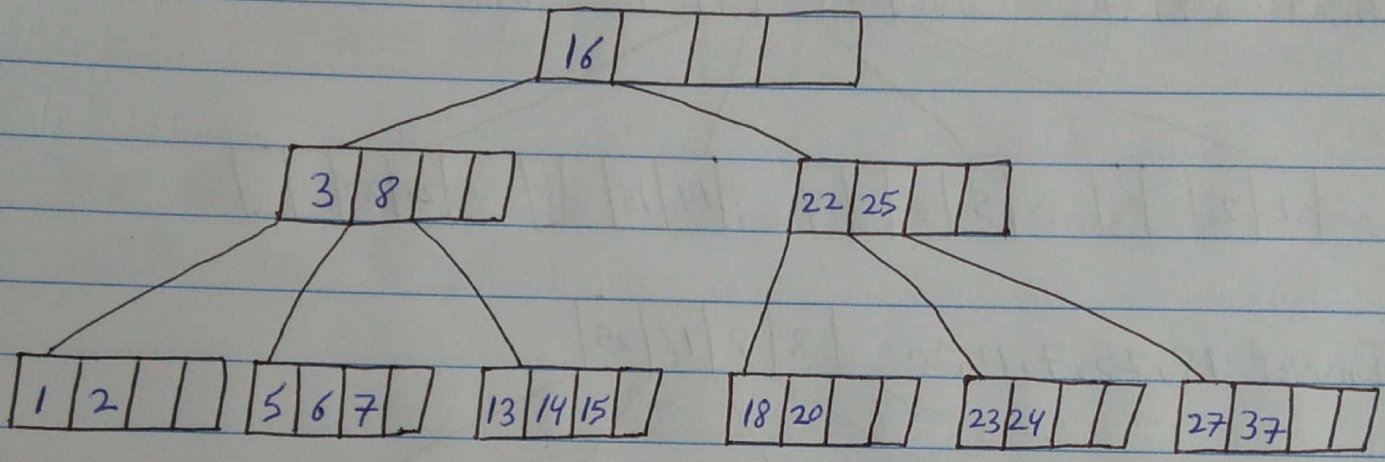
Insert 22,23,24



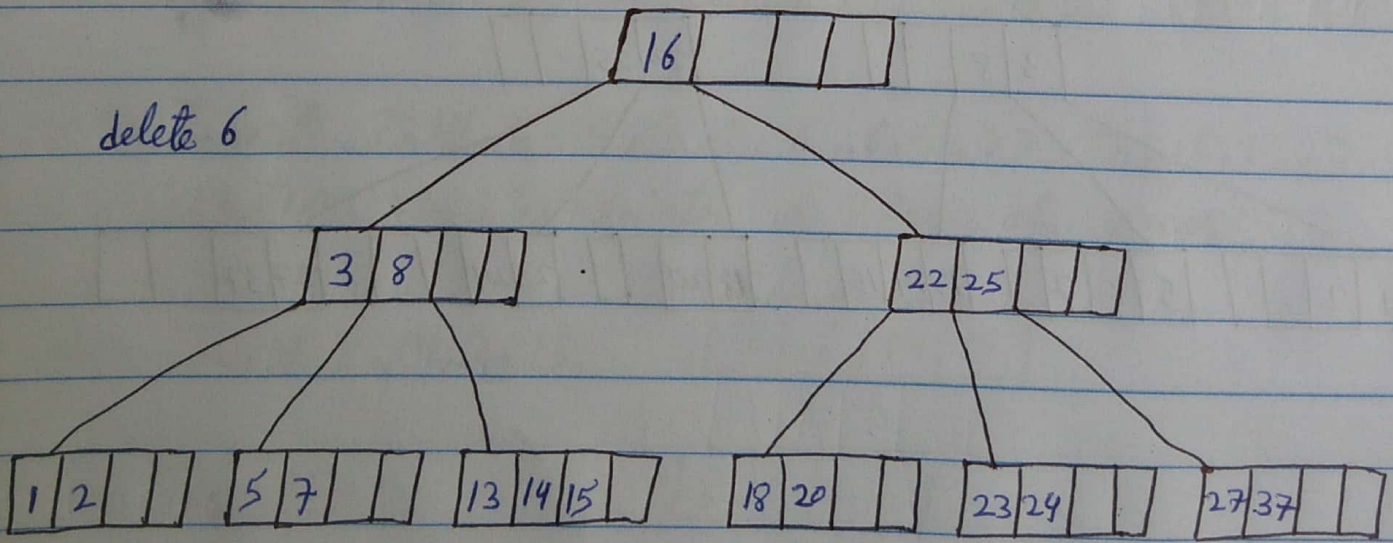
* Deleting a key from a B Tree

While doing deletion, the number of keys in any node should not become less than half full.

1.) Deleting a key from a leaf

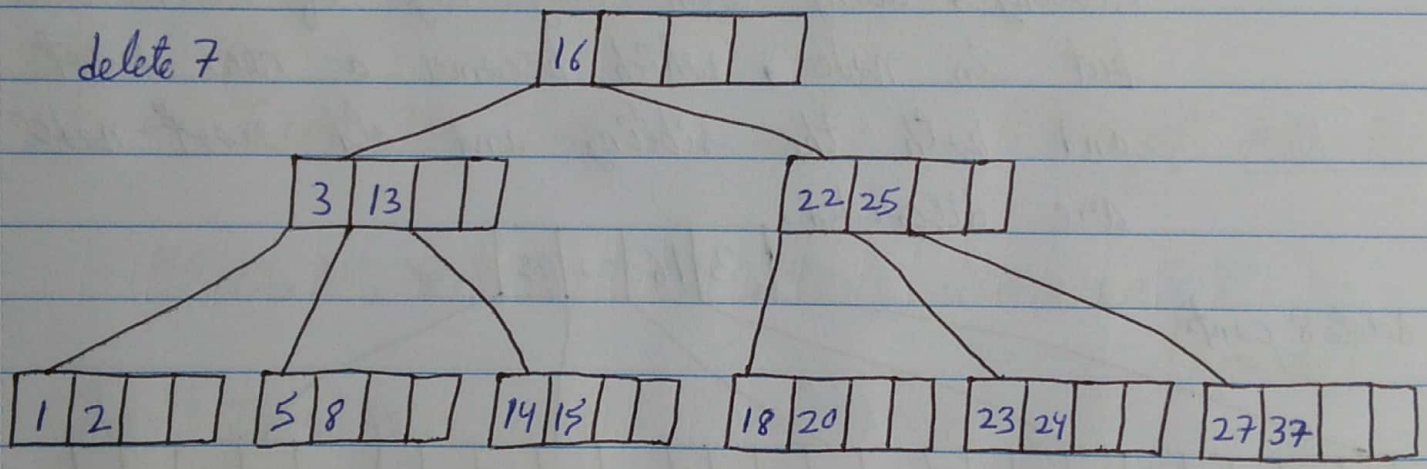


1.1) If, after deleting a key K, leaf is atleast half full & only keys greater than K are moved to left to fill holes.



1.2) If, after deleting K, no. of keys in left is less than $\lceil m/2 \rceil - 1$, causing an underflow.

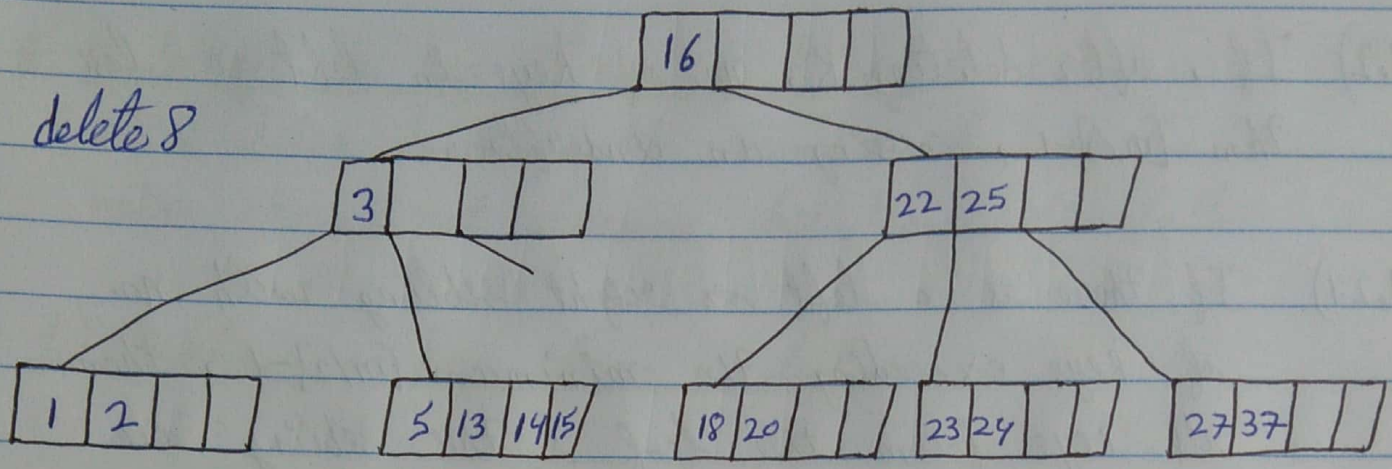
1.2.1) If there is a left or right sibling with no. of keys exceeding the minimum $\lceil m/2 \rceil - 1$, then all keys from this leaf & this sibling are redistributed b/w them by moving separator key from parent to leaf & then moving middle key from node & siblings combined to parent.



1.2.2) If the leaf underflows & no. of keys in its sibling is $\lceil m/2 \rceil - 1$, then leaf & a sibling are merged.

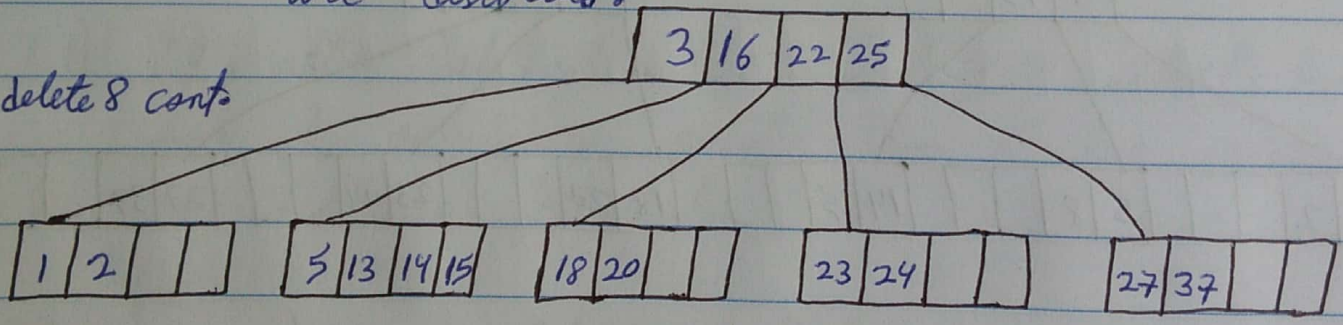
The keys from leaf, its sibling and separator key from the parent are all put in leaf and sibling is discarded.

delete 8



1.2.2.1) When its parent is root with only one key. In this case, keys from node & its sibling, along with only key of root are put in node, which becomes a new root and both the sibling and old root nodes are discarded.

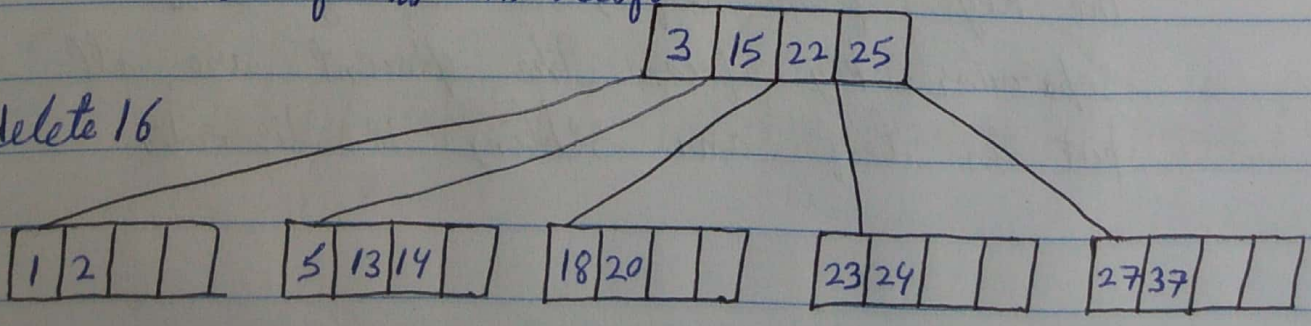
delete 8 cont.



Case 2: Deleting a key from a non-leaf

The key to be deleted is replaced by its immediate predecessor / Successor, which can only be found in leaf.

delete 16



☆ Advantages of B Tree

- B Trees are one of the approach to reduce the time penalty for accessing secondary storage, since most of the info is stored on disk or tapes.
- The imp. property of B trees is the size of each node which can be made as large as size of block memory.
- The number of keys in one node can vary depending on size of keys, organisation of data & size of a memory block.

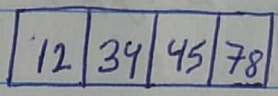
2017

Ex = Insert the following values in B tree of order 5.

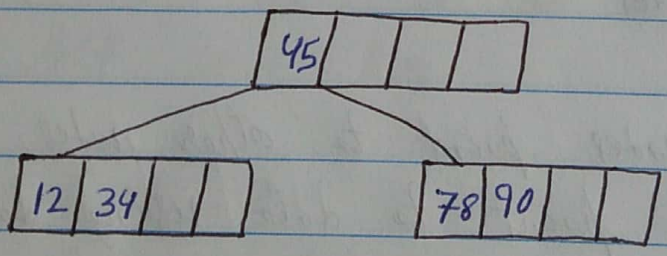
45, 12, 34, 78, 90, 22, 88, 96, 40, 82, 55, 100

[6 marks]

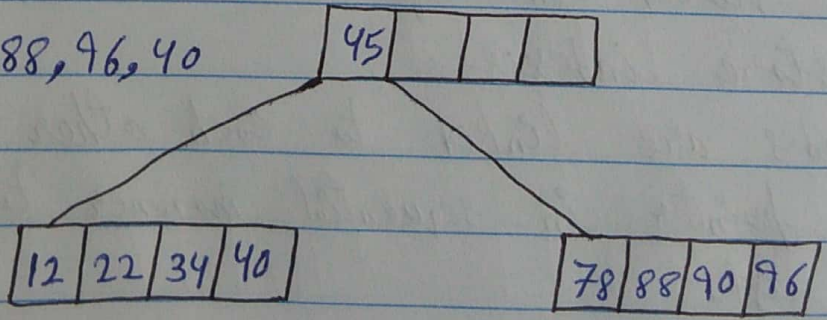
Ans = Insert 45, 12, 34, 78



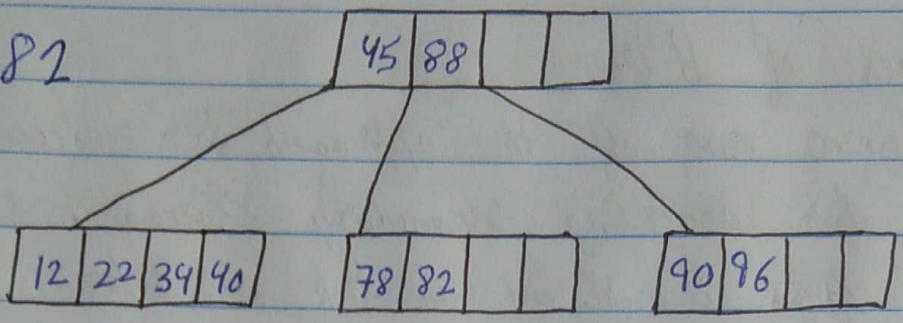
Insert 90



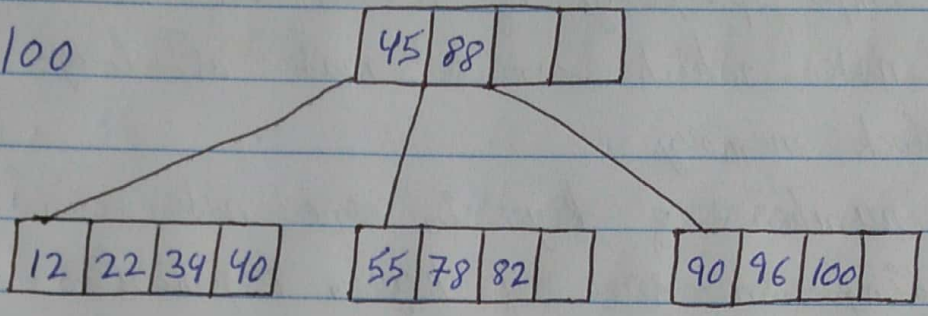
Insert 22, 88, 96, 40



Insert 82



Insert 55, 100



★ B+ Tree

B+ trees are in extension to B trees.
It consists of 2 types of nodes:
(i) Internal nodes
(ii) Leaf nodes

- Internal nodes point to other nodes in tree.
- Leaf nodes point to data using data pointer.
- Data is stored in leaf nodes & all other nodes store indexes.
- Leaf nodes are linked to each other using sibling pointer in sequential manner to form a linked list.

Advantages of B^+ trees

- Only leaf nodes needs to be traversed to scan entire tree as data is present only in leaf nodes without visiting higher nodes at all reducing block accesses to a great extent.

- Traversal is faster as compared to B trees in which data is present in all nodes which in turn would require more number of block accesses.

- Like B trees, B^+ trees are also balanced trees and every node except root must be atleast half full.

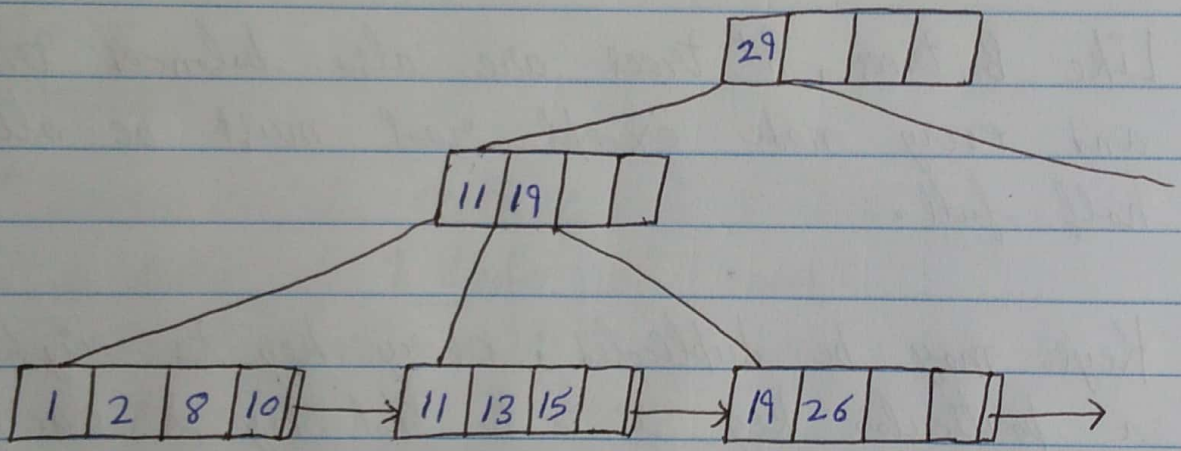
- Keys may be duplicated; every key to right of a particular key is > to that key & every key to left is <= key.

- Used for purpose of indexing in relational database.

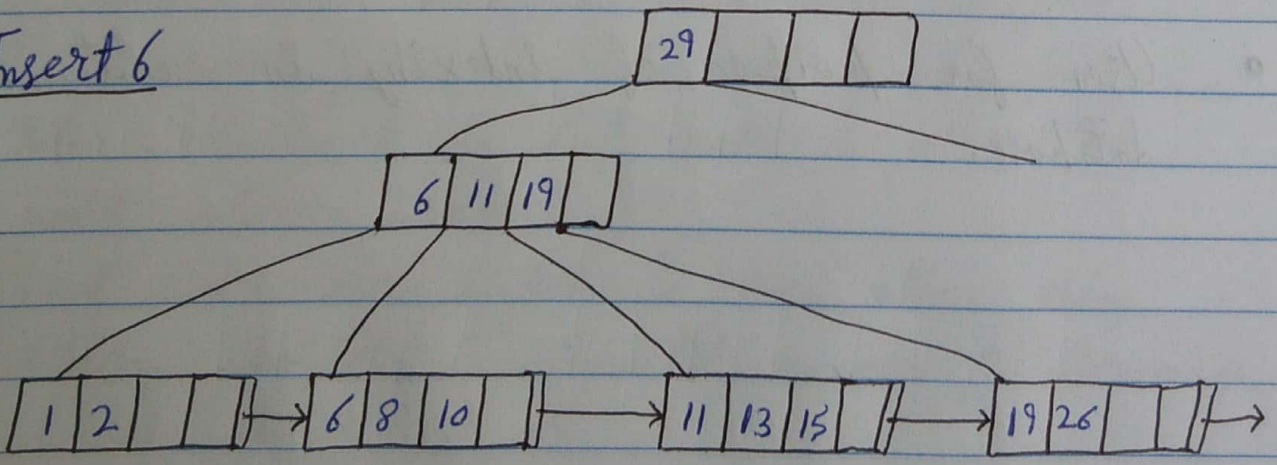
☆ Insertion in a B^+ tree

Inserting a key into a leaf that still has some room requires putting the keys of this leaf in order. No changes are made in index set.

- If a key is inserted into a full leaf, the leaf is split, the new leaf node is included in sequence set, all keys are distributed evenly b/w old & new leaves, and first key from new node is copied [not moved, as in a B tree] to parent.



Insert 6



TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

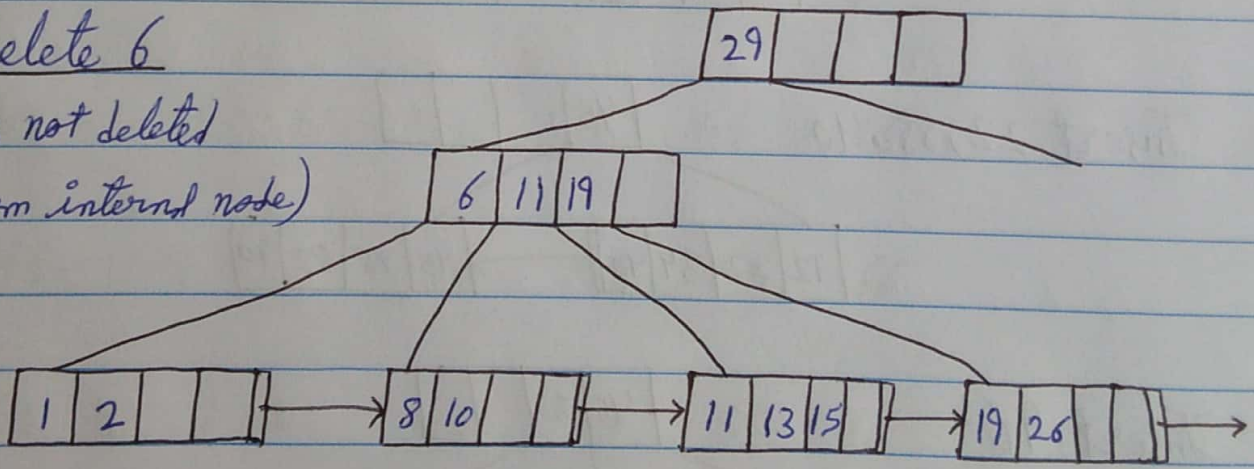
☆ Deletion of a key in B^+ tree

Deletion of a key from a leaf leading to no underflow requires putting the remaining keys in order. No changes are made to index set.

- If a key that occurs only in a leaf is deleted, then it is simply deleted from leaf But can remain in the Internal node.

- The reason is that it still serves as a proper guide when navigating down the B^+ tree because it still properly seperate keys b/w 2 adjacent children, even if seperator itself does not occur in either of children.

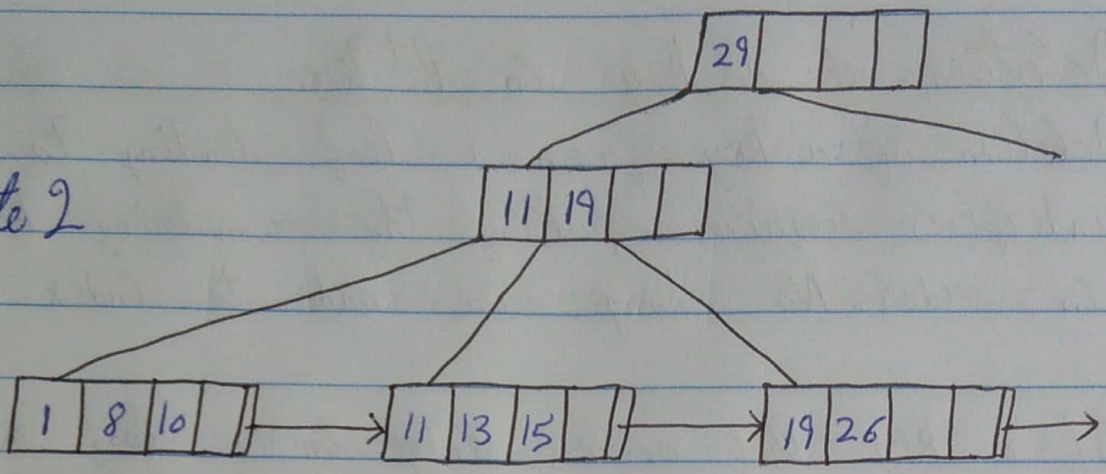
delete 6
(6 is not deleted from internal node)



- When deletion of a key from a leaf causes an underflow, then either the keys from this leaf & keys of a sibling are redistributed b/w this leaf and its sibling.

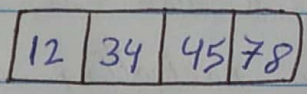
(42)

delete 2

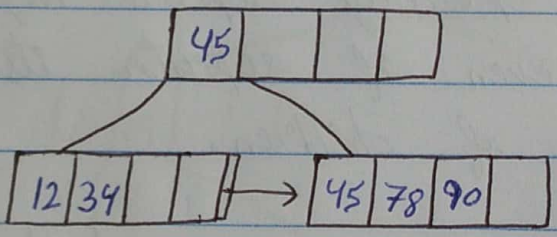


Ex= Insert the following values in a B^+ tree of order 5.
45, 12, 34, 78, 90, 22, 88, ~~90~~ 40, 96, ~~82~~.

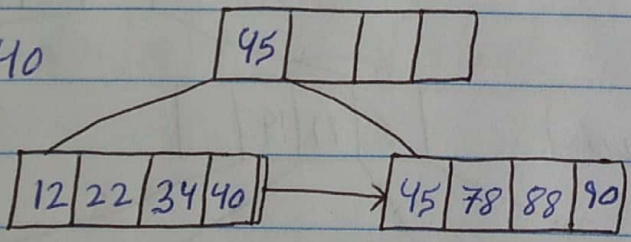
Ans= Insert 45, 12, 34, 78



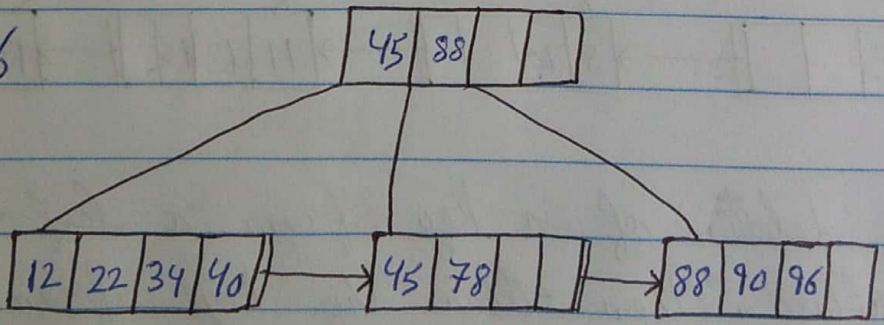
Insert 90



Insert 22, 88, 40



Insert 96



Sorting

- Sorting is the process of arranging a list of elements in a particular order.

1) Bubble Sort

This sorting algorithm is comparison-based algorithm in which each pair of adjacent element is compared & elements are swapped if they are not in order.

Data:	11	55	22	66	33	44
Pass 1:	11	22	55	33	44	66
Pass 2:	11	22	33	44	55	66
Pass 3:	11	22	33	44	55	66
Pass 4:	11	22	33	44	55	66
Pass 5:	11	22	33	44	55	66

- Since each iteration places a new element into its proper position, a file of 'n' elements require no more than 'n-1' iterations.

Total no. of comparisons, $T(n) = (n-1) + (n-2) + \dots + 2 + 1$

$= \frac{n(n-1)}{2}$

Complexity = $O(n^2)$

```
algo bubble_sort(a[], n)
for(i=1; i≤n-1; i++)
{   for(j=1; j≤n-i; j++)
    {   if(a[j] > a[j+1])
            swap a[j], a[j+1]
    }
}
```

Modified Bubble Sort

It includes a flag that is set if an exchange is made after an entire pass over the array.

- If no exchange is made, then the array is already sorted because no 2 elements need to be swapped. In that case, sorting should end.

Data:	11	55	22	66	33	44
Pass 1:	11	22	55	33	44	66
Pass 2:	11	22	33	44	55	66

- Sorting will end after 2nd pass as the array is sorted.

```
algo modified_bubble_sort(a[], n)
    flag = 1
    for(i=1; i<=(n-1) && flag==1; i++)
    {   flag = -1
        for(j=1; j<=(n-i); j++)
        {   if (a[j] > a[j+1])
            {   swap a[j], a[j+1]
                flag = 1
            }
        }
    }
```

2.) Insertion Sort

In insertion sort, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Data:	11	55	22	66	33	44
Pass 1:	11	55	22	66	33	44
Pass 2:	11	22	55	66	33	44
Pass 3:	11	22	55	66	33	44
Pass 4:	11	22	33	55	66	44
Pass 5:	11	22	33	44	55	66

Complexity = $O(n^2)$

```
algo insertion_sort (a, n)
for (i=1 ; i <= (n-1) ; i++)
{   if (a[i] > a[i+1])
    {   temp = a[i+1]
        loc = i+1
        while (loc != 1 && a[loc-1] > temp)
        {   a[loc] = a[loc-1]
            loc = loc-1
        }
        a[loc] = temp
    }
}
```

3) Selection Sort

In selection sort, the list is divided into 2 parts: Sorted part and unsorted part.

- Initially, sorted part is empty & unsorted part is the entire list.

- The smallest element is selected from unsorted array & swapped with leftmost element & that element becomes a part of sorted array. This process continues moving unsorted array boundary by one element to right.

Complexity : $O(n^2)$

```
algo  minpos ( a , lb, n)
      pos = lb
      for ( i = lb+1 ; i ≤ n ; i++)
      {     if ( a[pos] > a [i])
                  pos = i ;
      }
      return pos ;

algo  selection_sort ( a, n)
      for (i=1 ;  i ≤ n-1 ; i++)
      {       pos = minpos ( a , i , n)
              if ( pos ≠ i)
                    swap  a[pos] , a[i]
      }
```

Data :	11	55	22	66	33	44
Pass 1:	11	55	22	66	33	44
Pass 2:	11	22	55	66	33	44
Pass 3:	11	22	33	66	55	44
Pass 4:	11	22	33	44	55	66
Pass 5:	11	22	33	44	55	66

4) Quick Sort (Partition exchange sort)

This algorithm is based on partitioning of array of data into smaller array.

- A large array is partitioned into 2 arrays one of which holds values smaller than the specified value, pivot, based on which the partition is made, another array holds values greater than the pivot element.

- Quick sort partitions an array & then calls itself recursively twice to sort the 2 resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its average & worst case complexity are of $O(n^2)$.

```
algo quicksort ( a[], lb, ub)
    if ( lb < ub)
    {     pos = partition ( a, lb, ub)
          quicksort ( a, lb, pos-1)
          quicksort ( a, pos+1, ub)
    }
```

```
alg  partition ( a[] , lb, ub)
    pivot = a [lb]
    start = lb ,  end = ub
    while ( start < end)
    {     while ( a[start] ≤ pivot && start < end )
                start = start + 1 ;

          while ( a[end] > pivot)
                end = end - 1 ;

          if ( start < end)
                swap ( a [start] , a[end])
    }
    a [lb] = a [end]
    a [end] = pivot
    return end
```

Data :

LB				UB
33 (pivot) start→	11	22	55	44 end
33	11 start→	22	55	44 end
33	11	22 start→	55	44 end
33	11	22	55 start	44 ←end
33	11	22	55 ←end/start	44

swap

33 (pivot) 11 22 55 44

end start

22 11 33 55 44

recursively sort left subarray — recursively sort right subarray

LB UB LB UB

22 (pivot) 11 33 55 (pivot) 44

start → end start → end

swap swap

22 11 33 55 44

start/end start/end

Sorted array: 11 22 33 44 55

5.) Merge Sort

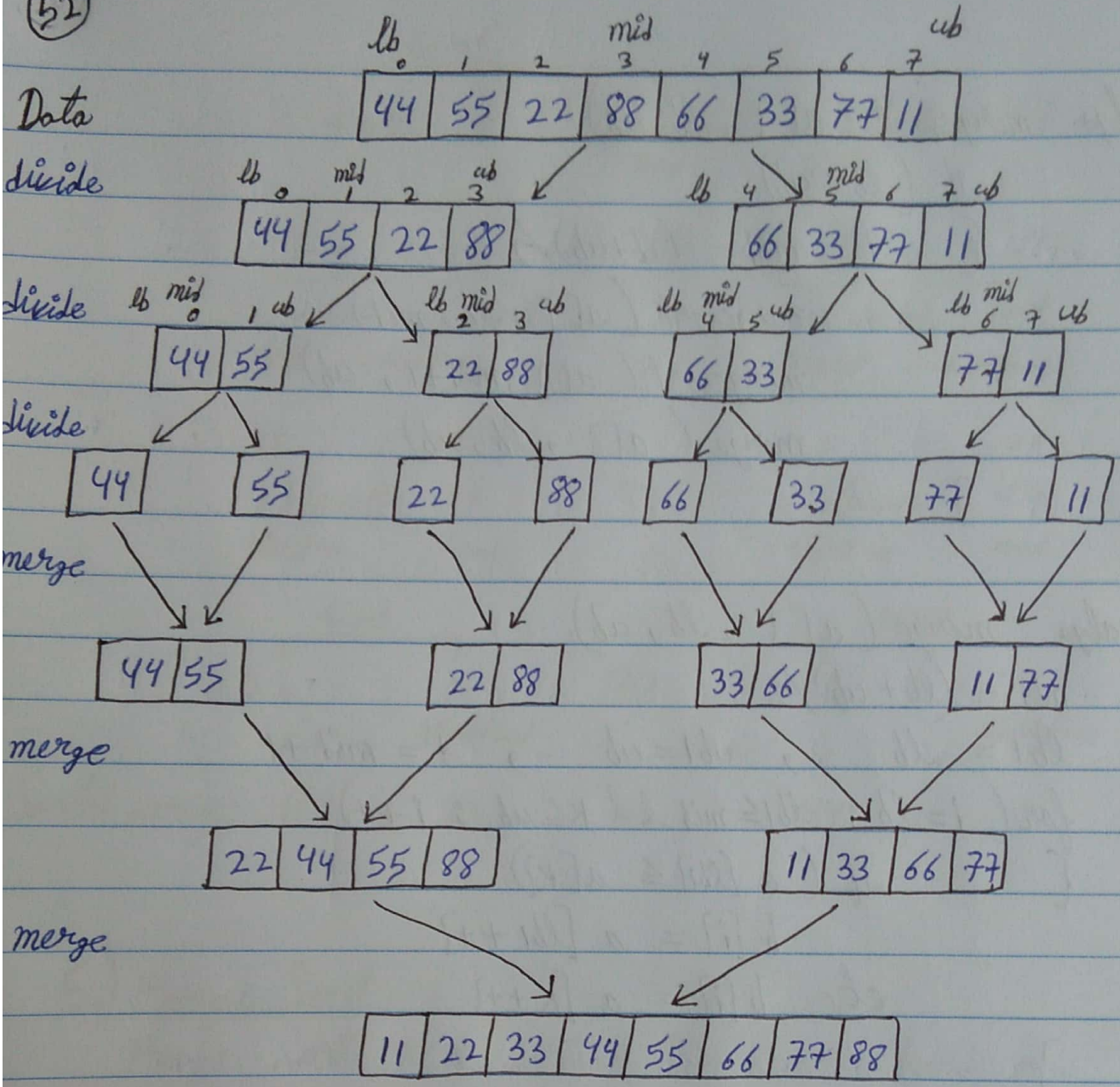
Merge Sort is a sorting technique based on Divide & Conquer technique.

- Merge sort keeps on dividing the list into equal halves until it can no more be divided. A list of 1 element is considered sorted.
- It repeatedly merge sublists to produce new sorted sublist until there is only 1 sublist remaining. This will be sorted list.

Complexity : $O(n \log n)$

```
algo mergesort (a[], lb, ub)
    if (lb < ub)
    {       mid = (lb+ub)/2
            mergesort (a[], lb, mid)
            mergesort (a[], mid+1, ub)
            merge (a[], lb, ub)
    }

algo merge (a[], lb, ub)
  mid = (lb+ub)/2
  lb1 = lb , ub1 = ub , K = mid+1
  for (i=lb; lb1 <= mid && K <= ub ; i++)
  {       if (a[lb1] <= a[K])
              b[i] = a[lb1++]
          else b[i] = a[K++]
  }
  while (lb1 <= mid)
          b[i++] = a[lb1++];

  while (K <= ub)
          b[i++] = a[K++];

  for (i = lb ; i <= ub ; i++)
          a[i] = b[i];
```

(52)



Ex= Consider following list of 9 numbers:

66, 33, 44, 22, 55, 88, 11, 77, 99

Suppose the list is to be sorted. Use quick sort to find find position of first number 66 in sorted list. [5 marks]

Ans =

Pass 1: 11 33 44 22 55 66 88 77 99

So, find position of 66 in sorted list is 6th.

2016

Ex= Apply Bubble sort on following array of integers:
26, 45, 13, 23, 12, 7, 38, 42
Show the content of array after every pass.
[6 marks]

Ans= Pass 1: 26 13 23 12 7 38 42 45
Pass 2: 13 23 12 7 26 38 42 45
Pass 3: 13 12 7 23 26 38 42 45
Pass 4: 12 7 13 23 26 38 42 45
Pass 5: 7 12 13 23 26 38 42 45

- Sorting will terminate after 5th pass as no swapping is required to sort the array.

2017

Ex= Sort the following set of elements using selection sort. Show the content of array after every pass
34, 56, 12, 8, 92, 9, 44, 23
[5 marks]

Ans= Pass 1: 8 56 12 34 92 9 44 23
Pass 2: 8 9 12 34 92 56 44 23
Pass 3: 8 9 12 34 92 56 44 23
Pass 4: 8 9 12 23 92 56 44 34
Pass 5: 8 9 12 23 34 56 44 92
Pass 6: 8 9 12 23 34 44 56 92
Pass 7: 8 9 12 23 34 44 56 92

Searching

1) Sequential or Linear Search

This search is applicable to a table organised either as an array or as a linked list.

- When using linear search we check each & every element of list, comparing it with given key, one element at a time.

- The process continues until the desired element is found or we reach the end of list.

$$\boxed{\text{Complexity} = O(n)}$$

Linear Search using Recursion

```
int RecLinearSearch(int arr[], int size, int key)
{   int loc;
    if (arr[size] == key)
        return size;

    else if (size == -1)
        return -1;

    else return (loc = RecLinearSearch(arr, size-1, key));
}
```

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Linear Search using Iteration

```
int LinearSearch(int arr[], int size, int key)
{
    for (int i=0; i<n; i++)
        if (key == arr[i])
            return (i);
    return (-1);
}
```

2) Binary Search

For binary search, list of elements should be sorted.

- Binary search looks for the searched element by comparing the middle most element of the list.
- If a match occurs, then index of item is returned.
- If middle element is greater than item, then item is searched in sub array to left of middle element.
- Otherwise, the item is searched ~~for~~ in the sub array to right of middle element.
- This process continues on sub array as well until size of sub array reduces to zero.

Complexity : O(log n)

Binary Search using Iteration

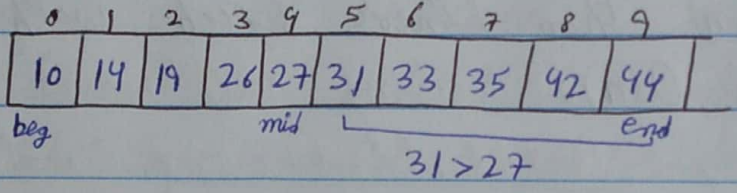
```
int BinarySearch( a[], lb, ub, item)
{       int beg = lb;
        int end = ub;
        int mid = (beg+end)/2;
        while ( a[mid] ≠ item && beg ≤ end)
        {       if ( a [mid] > item)
                        end = mid -1;
                else  beg = mid+1;
                mid = (beg + end)/2;
        }
        if ( a[mid] == item)
           int loc = mid;
        else   int loc = -1;
        return (loc);
}
```

Binary Search using Recursion

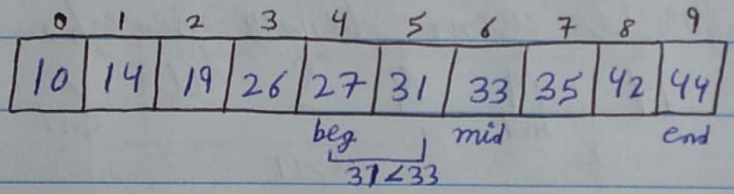
```
int RecBinarySearch(int a[], int lb, int ub, int item)
{       if ( lb ≤ ub)
        {   int mid = (lb+ub)/2;
            if ( item == a[mid])
                    return mid;
```

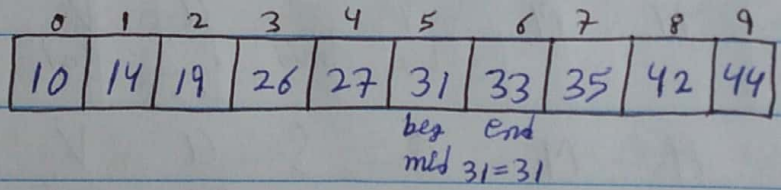
```
    else if ( item < a[mid])
        return RecBinarySearch(a, lb, mid-1, item);
    else return RecBinarySearch(a, mid+1, ub, item);
  }
  return -1;
}
```

Ex= Search for 31 using binary search.



Ans=





Hence, the searched element, i.e. 31 is found at position 5.

- Binary Search is not suitable when data is stored in linked list.

2016

Ex = Consider following list S of alphabetic characters:
B, F, H, M, Q, S, U, V

(58)

An application requires to perform search operations on above list. Which of the search technique is appropriate & why? Apply the technique suggested by you to search an element U on the list S and count the number of comparisons performed in searching U.

Ans = As the list is already sorted, binary search will be more appropriate with time complexity of $O(\log n)$ than Linear search with time complexity $O(n)$.

Searching for U using binary search:

1	2	3	4	5	6	7
B	F	H	M	Q	S	U	V
beg			mid				end

U > M

B	F	H	M	Q	S	U	V.
				beg	mid		end

U > S

B	F	H	M	Q	S	U	V
						beg mid	end

- Hence, U is found at position 6 and no. of comparisons in searching U are 3.

2017

Skip List

- A skip list is a data structure that allows fast search within an ordered sequence of elements by making non sequential search.

- Linked list have one drawback that it requires sequential scanning to locate a searched-for element.

- This drawback can be overcome by using skip list where (i) Elements are ordered in linked list.
  (ii) Makes non-sequential search (skips some element).

$$\boxed{\text{complexity} = O(\log n)}$$

Properties

The node in position points to node in position that is every second node points to 2 positions ahead.

- If n is no. of nodes in skip list then half of nodes have 1 pointer, one-fourth of nodes have 2 pointers, one-eighth of nodes have 3 pointers & so on.

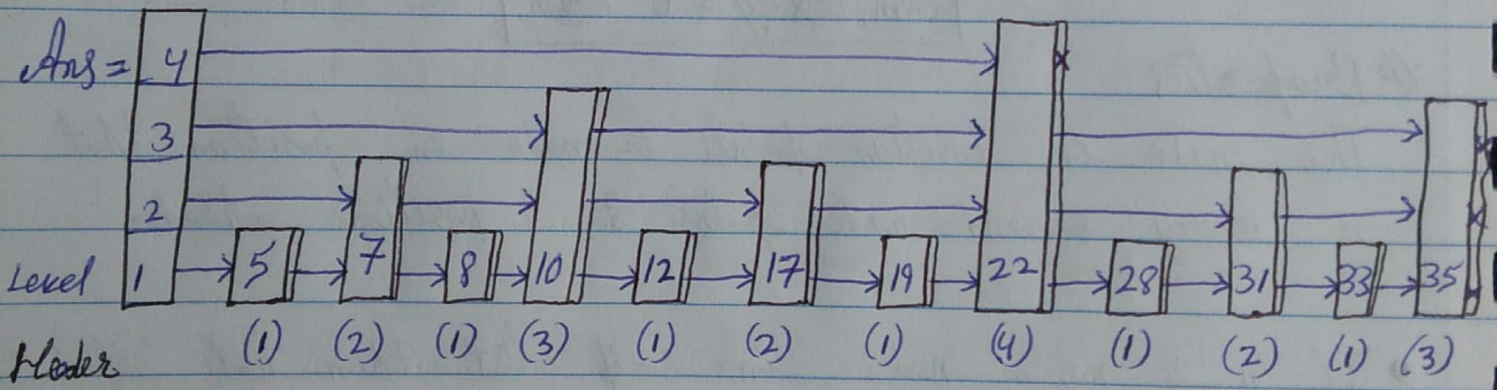
- The no. of pointers indicates level of each node.

Searching in a skip list

Searching for an element 'el' consists of following pointers on highest level until an element is found that finishes the search successful.

- In case of reaching the end of list or encountering an element key that is greater than 'el', search is restarted from node preceding the one containing key, but this time starting from a pointer on a lower level than before.
- The search continues until 'el' is found or first level pointers are followed to reach end of list or to find an element greater than 'el'.

Ex= Search for 16 using skip list,
5, 7, 8, 10, 12, 17, 19, 22, 28, 31, 33, 35

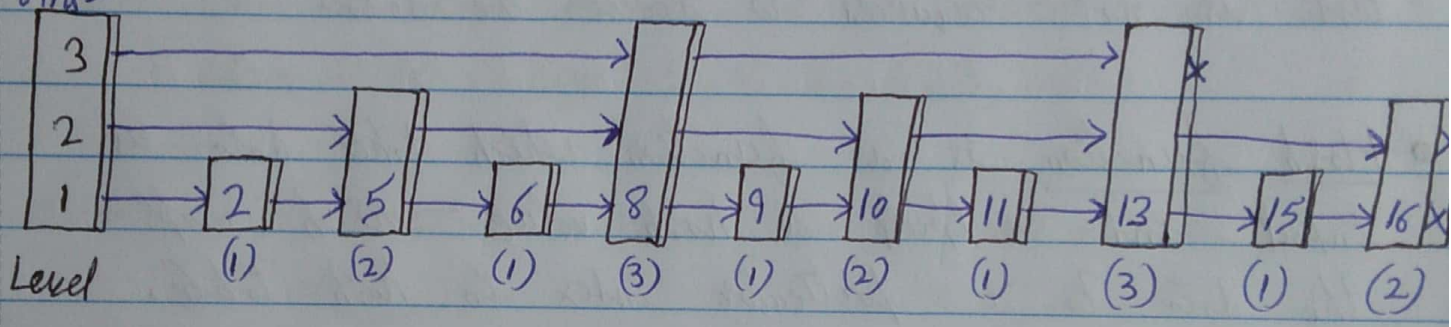


- If we look for number 16 in list, then level 4 is tried first, which is unsuccessful because first node on this level has 22.
- Next, we try the 3rd level sublist starting from root: It first leads to 10 & then to 22.
- Hence, we try second level sublist that originates from node holding 7. It then leads to 10 & then again to 17.

- The last try is by starting first level sublist which begins in node 5.
- This sublist's first node has 5 and then 7, 8, 10, 12, 17, as 17 is greater than 16 and there is no lower level, the search is unsuccessful.

Ex= Search for 10 using skip list,
2, 5, 6, 8, 9, 10, 11, 13, 15, 16

Ans=



- If we search for 10 in the list, then level 3 is tried first, which is unsuccessful because 10 is greater than 8 but less than 13.
- Next, we try second level which starts from node 5 and then it leads to 8 and then to 10.

- Hence, the search is successful and the searched element i.e. 10 is found in level 2.

Hashing

- Hashing approach to searching calculate the position of key in table based on value of data which indicates actual position of data being searched in the hash table.
- The data can be directly accessed using value of key.
- Thus, in this method comparison of keys or any other test is not required to access the data.
- Hash function is a function which takes data as input and outputs a hash value which maps the data to a particular index in hash table.

Complexity: O(1)

- It is independent of no. of elements (n).
- Perfect hash function: If hash function transforms different data into different index with no collision.

☆ Hash Functions

1.) Division

Each key to be inserted will be divided by size of table in which values are to be stored.
Then, modulus or remainder of this division will

be used to index into the table.

$$h(K) = K \bmod Tsize$$

where, $h(K)$ = hash function to be used
K = Key to be inserted
$Tsize$ = Size of table

Ex = Let the hash function be $2K^2+3K+2$ and table size is 10. Keys: 5, 8, 2, 6, 3, 7.

Ans = (i) $h(5) = (2\times5^2+3\times5+2)\%10$
$= 7$

(ii) $h(8) = (2\times8^2+3\times8+2)\%10$
$= 4$

(iii) $h(2) = (2\times2^2+3\times2+2)\%10$
$= 6$

(iv) $h(6) = (2\times6^2+3\times6+2)\%10$
$= 2$

(v) $h(3) = (2\times3^2+3\times3+2)\%10$
$= 9$

(vi) $h(7) = (2\times7^2+3\times7+2)\%10$
$= 1$

2) Folding

In this method, the key is divided into several parts. These parts are combined or folded together & are often transformed in a certain way to create the target address.

- There are 2 types of folding:
(i) Shift folding
(ii) Boundary folding

(i) Shift Folding

In shift folding, the key is divided into parts & put underneath one another and processed.

Ex= Number 123456789 can be divided into 3 parts:

```
  123
  456
+ 789
 1368
```

- This sum value cannot be used to index into the table. The resulting no., 1368, can be divided modulo Tsize, or if size of table is 1000, the first 3 digits can be used for address.

(ii) Boundary Folding

In boundary folding, the divided parts are written on a piece of paper & this paper is folded on boundaries of these parts.

- When we do that, every alternate part of key appears reversed.

```
  1 2 3
  6 5 4
+ 7 8 9
 15 6 6
```

- The sum will be divided modulo the size of table into which the keys are to be inserted.

3.) Mid Square function

In mid square method, the key is squared & middle part of result is used as address.

Ex = 3121

$(3121)^2 = 9740641$

for 1000 cell table, $h(3121) = 406$

4.) Extraction

In extraction method, only a part of key is used to compute the address.

Ex = 12345 6789

- This method might use first four digits 1234, last four digits 6789, first 2 combined with last 2, 1289 and so on.

5.) Radix Transformation

In this method, key is transformed from current radix to a different radix in the number system.

Ex: If key is 322 in decimal number system, then its value will be 423 in nonal number system i.e. with base 9.

The value will be divided modulo of the size of table & resulting value is to be used as index into the table.

- **Collision Resolution**

Any hash function can lead to collision, which means that 2 different key value can be given some index value in the table.

- Thus, collision resolution methods are required which allocates a different location in table each time a new value is to be inserted.
- There are methods to resolve this collision;

1.) Open Addressing

When 2 keys collide for some index value, open addressing method is used to resolve the collision.

- In this method, when a collision occurs the table is searched for the next available cell in it & key is placed in that cell, if it is empty.
  This process is repeated until these some positions are searched repeatedly or table is full.

- Open addressing is also known as closed hashing because collided keys are stored within the table.

- Open addressing is of 3 types :-

(i) Linear Probing

In linear probing, the position in which a key can be stored is found by sequentially searching all positions

starting from position calculated by hash function until an empty cell is found.

- If end of table is reached & no empty cell has been found, the search is continued from beginning of table & stops in cell preceding the one from which the search started.

$$p(i) = (h(k) + i) \bmod Tsize$$

Ex = Insert: $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$; Tsize = 10.

Ans =

0			B_9
1			
2	A_2	A_2	A_2
3	A_3	A_3	A_3
4		B_2	B_2
5	A_5	A_5	A_5
6		B_5	B_5
7			
8			
9		A_9	A_9

Insert: A_5, A_2, A_3 B_5, A_9, B_2 B_9, C_2

2017

Ex = A hash table of length 10 uses open addressing with hash function $h(k) = k \bmod 10$, and linear probing.
Keys: 85, 63, 11, 32, 71, 52.

[3 marks]

(68)

hash table →

0	1	2	3	4	5	6	7	8	9
	11	32	63	71	85	52			

(i) $h(85) = 85 \bmod 10 + 0$
$= 5$

(ii) $h(63) = 63 \bmod 10 + 0$
$= 3$

(iii) $h(11) = 11 \bmod 10 + 0$
$= 1$

(iv) $h(32) = 32 \bmod 10 + 0$
$= 2$

(v) $h(71) = 71 \bmod 10 + 0$
$= 1$ (collision)
$= 71 \bmod 10 + 1$
$= 2$ (collision)
$= 71 \bmod 10 + 2$
$= 3$ (collision)
$= 71 \bmod 10 + 3$
$= 4$

(vi) $h(52) = 52 \bmod 10 + 0$
$= 2$ (collision)
$= 52 \bmod 10 + 1$
$= 3$ (collision)
$= 52 \bmod 10 + 2$
$= 4$ (collision)
$= 52 \bmod 10 + 3$
$= 5$ (collision)
$= 52 \bmod 10 + 4$
$= 6$

* Cluster

A cluster is a collection of index positions which contain collided key values.

- Linear probing has a tendency to create primary cluster in a hash table.

- Primary Cluster occurs after a collision causes 2 of records in a hash table to hash to some position and causes one of the record to be moved

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

to next location in its probe sequence.

- In order to avoid primary clustering, we use quadratic probing.

(ii) Quadratic Probing

$$p(i) = (h(k) \pm i^2) \bmod Tsize$$

if $i = +i^2 \rightarrow i = 0, 1, 2, \ldots, Tsize - 1$

if $i = \pm i^2 \rightarrow i = 0, 1, 2, \ldots, (Tsize - 1)/2$

Ex = Insert: $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$; $Tsize = 10$.

0		0		0	B_9	0	B_9	0	B_9
1		1	B_2	1	B_2	1	B_2	1	B_2
2	A_2	2	A_2	2	A_2	2	A_2	2	A_2
3	A_3	3	A_3	3	A_3	3	A_3	3	A_3
4		4		4		4		4	
5	A_5	5	A_5	5	A_5	5	A_5	5	A_5
6		6	B_5	6	B_5	6	B_5	6	B_5
7		7		7		7		7	
8		8		8		8		8	C_2
9		9	A_9	9	A_9	9	A_9	9	A_9

Insert: A_5, A_2, A_3 B_5, A_9, B_2 B_9, C_2

Ex= Load the keys into a hash table of size 7 using quadratic probing.

Keys : 23, 13, 21, 14, 7, 8 [3 marks]

Ans= hash table

0	1	2	3	4	5	6
21	14	23		7	8	13

(i) $h(23) = ((23 \bmod 7) \pm 0^2) \bmod 7$
$= 2$

(ii) $h(13) = (13 \bmod 7 + 0^2) \bmod 7$
$= 6$

(iii) $h(21) = (21 \bmod 7 + 0^2) \bmod 7$
$= 0$

(iv) $h(14) = (14 \bmod 7 + 0^2) \bmod 7$
$= 0$ (collision)
$= (14 \bmod 7 + 1^2) \bmod 7$
$= 1$

(v) $h(7) = (7 \bmod 7 + 0^2) \bmod 7$
$= 0$ (collision)
$= (7 \bmod 7 + 1^2) \bmod 7$
$= 1$ (collision)
$= (7 \bmod 7 - 1^2) \bmod 7$
$= 6$ (collision)
$= (7 \bmod 7 + 2^2) \bmod 7$
$= 4$

(vi) $h(8) = (8 \bmod 7 + 0^2) \bmod 7$
$= 1$ (collision)
$= (8 \bmod 7 + 1^2) \bmod 7$
$= 2$ (collision)
$= (8 \bmod 7 - 1^2) \bmod 7$
$= 0$ (collision)
$= (8 \bmod 7 + 2^2) \bmod 7$
$= 5$

- Although using quadratic probing gives much better results than linear probing, the problem of cluster buildup is not avoided altogether because for keys hashed to same location, the same probe sequence is used. Such clusters are called Secondary Cluster.

- To avoid problem of secondary clustering, we use the concept of double hashing.

(iii) Double Hashing

This methods utilises 2 hash functions one for accessing the primary position of a key, h, and a secondary function, h_p, for resolving conflict.

$$p(k,i) = [h(k) + i\,h_p(k)] \bmod Tsize$$

Ex = Load the keys into a hash table of size 13. using double hashing.

$h(k) = k \bmod Tsize$, $h_p(k) = 1 + k \bmod 12$

Keys : 18, 26, 35, 64, 9, 96

Ans = hash table ⇒

0	1	2	3	4	5	6	7	8	9	10	11	12
26					18	9	96		35			64

(i) $h(18) = 18 \bmod 13$
$= 5$

$h_p(18) = 1 + 18 \bmod 12$
$= 7$

$p(18,0) = (5 + 0 \times 7) \bmod 13$
$= 5$

(ii) $h(26) = 26 \bmod 13$
$= 0$

$h_p(26) = 1 + 26 \bmod 12$
$= 3$

$p(26,0) = (0 + 0 \times 3) \bmod 13$
$= 0$

(iii) $h(35) = 35 \bmod 13$
$= 9$

$h_p(35) = 1 + 35 \bmod 12$
$= 12$

(iv) $h(64) = 64 \bmod 13$
$= 12$

$h_p(64) = 1 + 64 \bmod 12$
$= 5$

$p(35,0) = (9 + 0 \times 12) \bmod 13$
$= 9$ //

(v) $h(9) = 9 \bmod 13 = 9$
$h_e(9) = 1 + 9 \bmod 12 = 10$
$p(9,0) = (9 + 0 \times 10) \bmod 13$
$= 9$ (collision)
$p(9,1) = (9 + 1 \times 10) \bmod 13$
$= 6$ //

$p(64,0) = (12 + 0 \times 5) \bmod 13$
$= 12$ //

(vi) $h(96) = 96 \bmod 13 = 5$
$h_e(96) = 1 + 96 \bmod 12 = 1$
$p(96,0) = (5 + 0 \times 1) \bmod 13$
$= 5$ (collision)
$p(96,1) = (5 + 1 \times 1) \bmod 13$
$= 6$ (collision)
$p(96,2) = (5 + 2 \times 1) \bmod 13$
$= 7$ //

2) Chaining

In chaining, keys are not necessarily stored in the table. Each position in the table is a pointer which links to first key value, which hashes to this index value, say i.

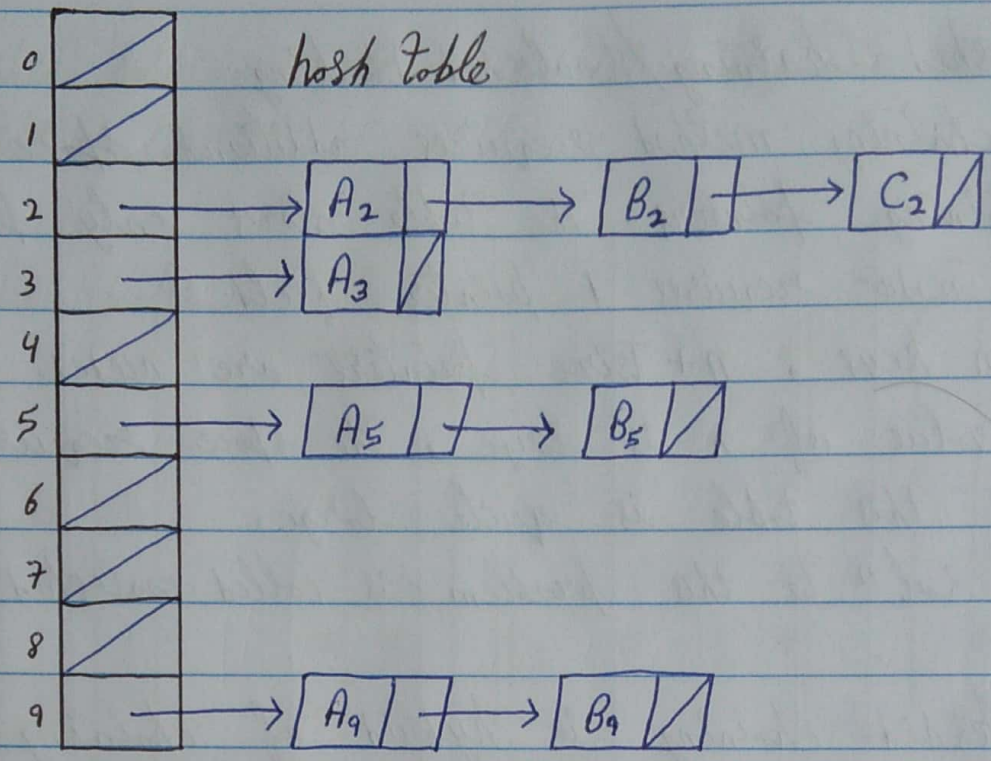
- When a new key value hashes to index position, i, this key value is attached to end of linked list placed at this index position.

Ex = Insert the keys $A_5, A_2, A_3, B_5, A_9, B_2, C_2, B_9$ into a hash table of size 10.

Ans = Keys: $A_5, A_2, A_3, B_5, A_9, B_2, C_2, B_9$

Tsize = 10

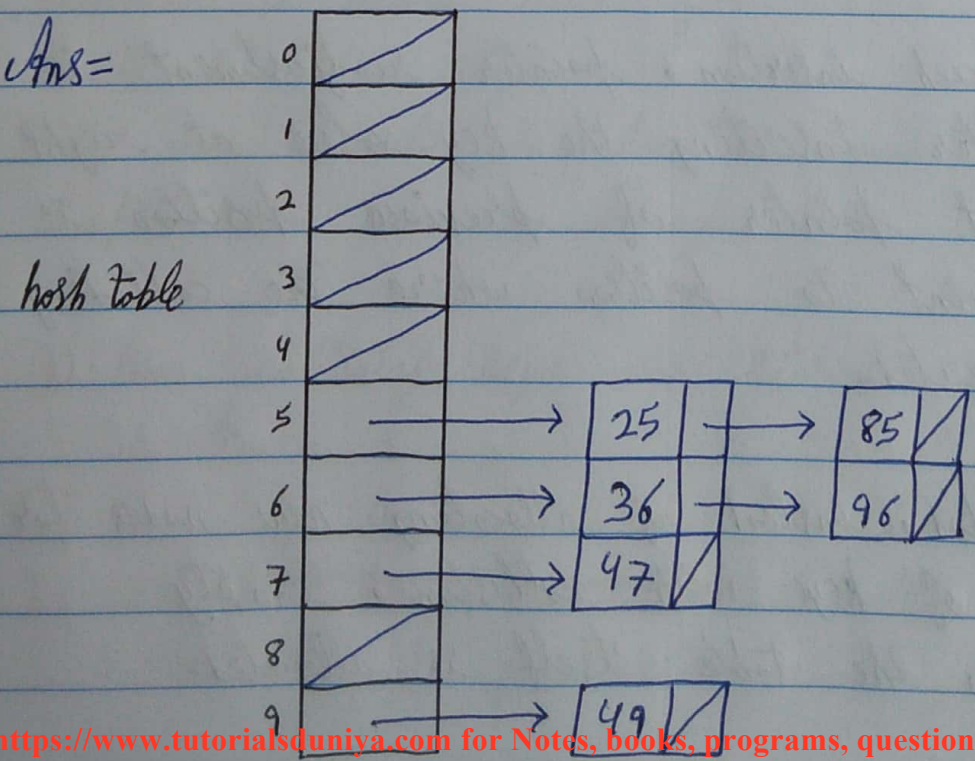
Ans=



- Null indicates that no key values are stored at these index positions.

Ex= Insert: 25, 36, 47, 85, 96, 49. Consider max. size of table is 10 & hash function division modulo.

Ans=



* Coalesced Chaining / Coalesced Hashing

The chaining method requires additional space for maintaining pointers. The table stores only pointers & each node requires 1 pointer field.
For n keys, n+Tsize pointers are needed.

- If value of n is large, the space required to store this table is quite large.
The sol^n to this problem is called coalesced chaining.

- Coalesced chaining is hybrid of chaining and open addressing.
- Each index position in the table stores key values & a pointer to next index position.

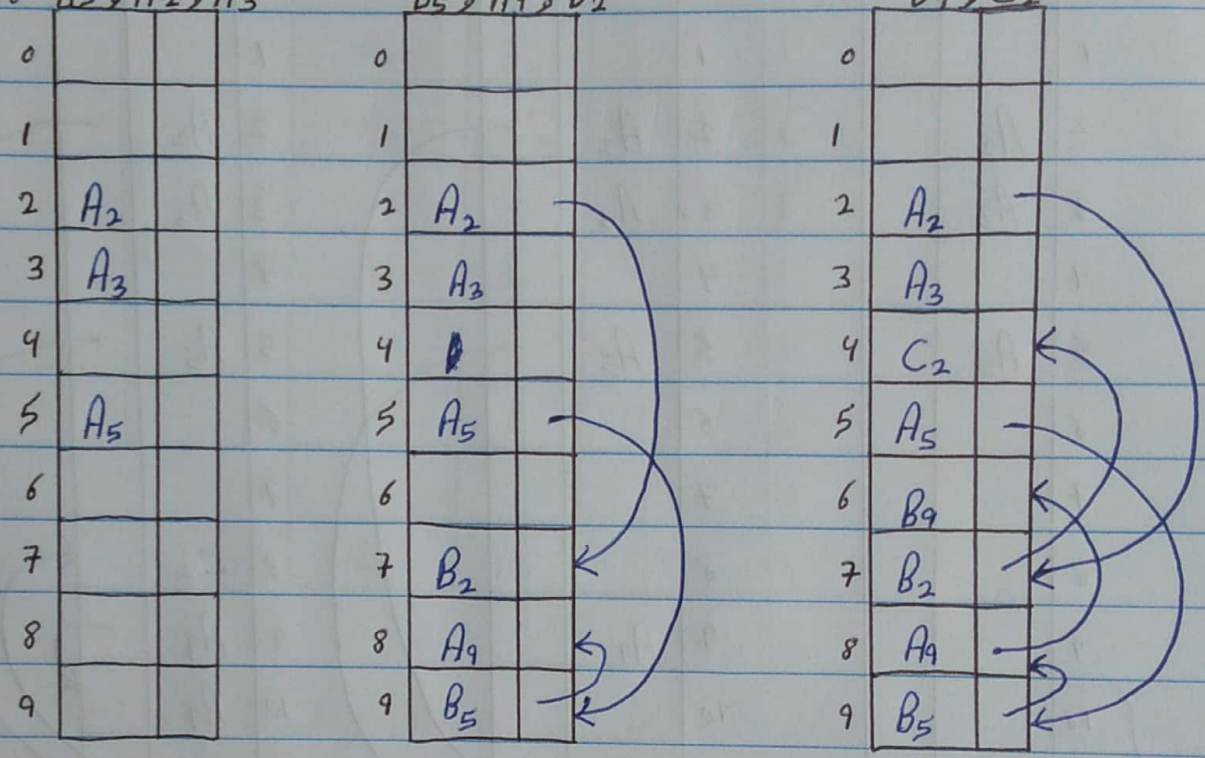
- In this method, next available position is searched for a colliding key & is placed in that position.

- After each such insertion, pointer readjustment is required. After inserting the key value at right place, next pointer of previous position is made to point to position where the colliding key is inserted.

- In this method, instead of allocating new nodes for linked list of keys with collision, empty position from the table itself is allocated.

Ex: Coalesced hashing puts a colliding key in the last available position of table.

Insert: A_5, A_2, A_3 | B_5, A_9, B_2 | B_9, C_2



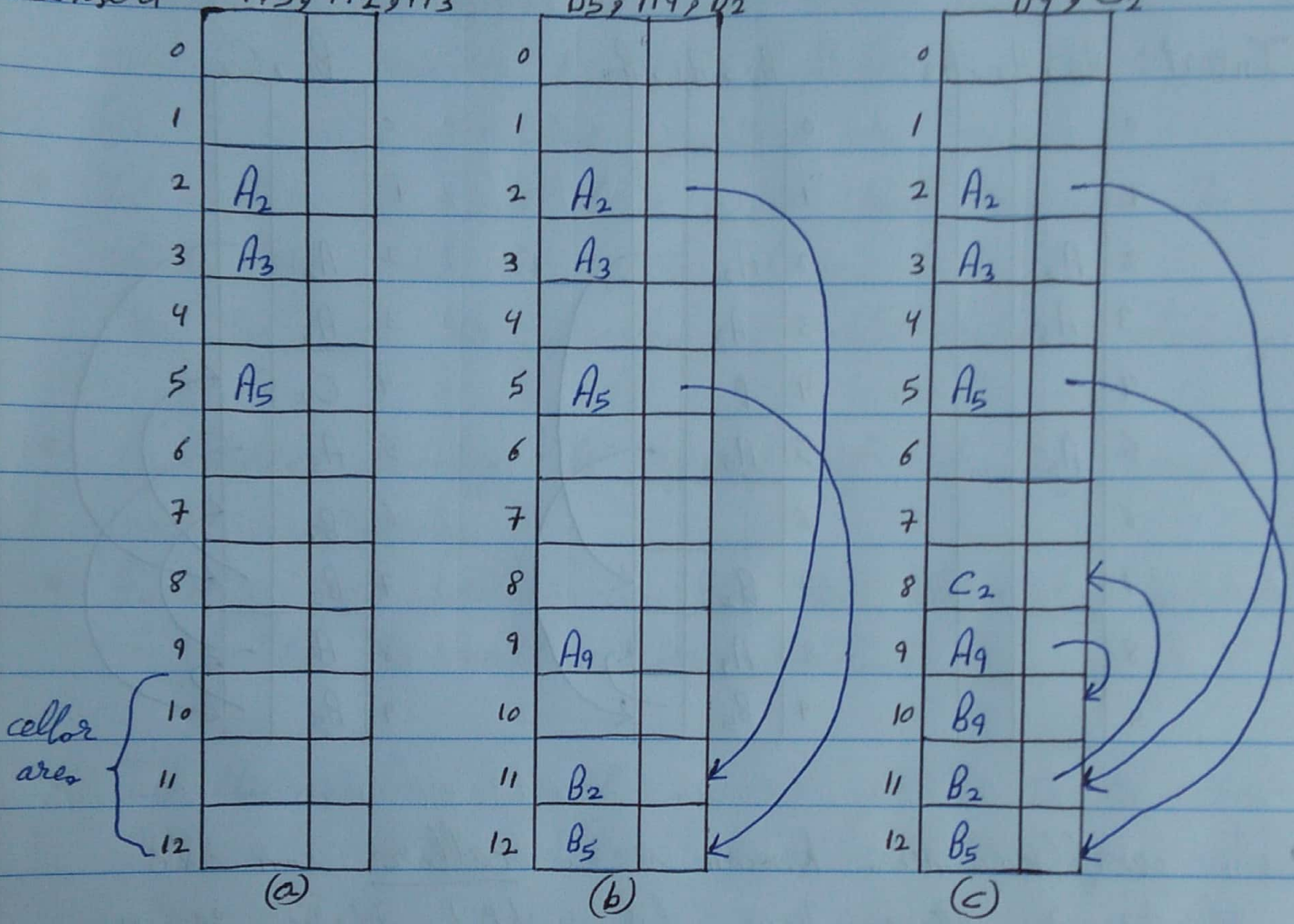
- An overflow area known as a <u>cellar</u> can be allocated to store keys for which there is no room in the table.
This area should be located dynamically if implemented as an array.

(a) Non colliding keys are stored in their home positions.

(b) Colliding keys are put in last available ~~cell~~ slot is taken from the table of cellar & added to list starting from their home position.

Ex= Coalesced hashing that uses a cellar

Insert: A_5, A_2, A_3 B_5, A_9, B_2 B_9, C_2



(c) The cellar is full, so an available cell is taken from the table when C_2 arrives.

3.) Bucket Addressing

Bucket addressing for collision resolution allows the table to store all colliding key values at some position.

- A bucket is a block of space large enough to hold multiple key values.

Ex = Collision resolution with buckets & linear probing

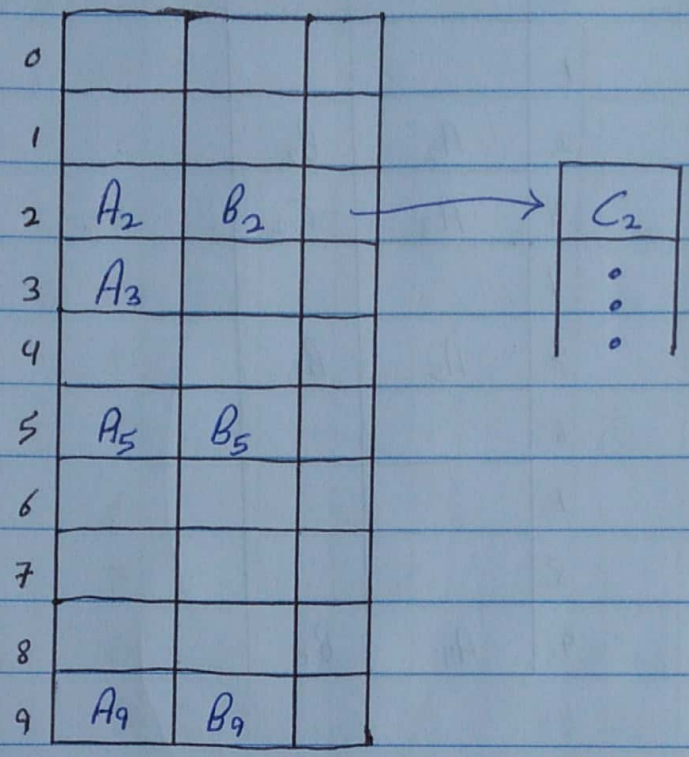
Insert : $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$; Tsize = 10.

0		
1		
2	A_2	B_2
3	A_3	C_2
4		
5	A_5	B_5
6		
7		
8		
9	A_9	B_9

- When the first key value is to be inserted into the table, the key value is hashed to determine the index where it will be placed.
- If a collision occurs, the particular key value is stored in a bucket provided for this key value.
- If this key value is also full, then this key value has to be stored in <u>overflow bucket.</u>
- In this case, each bucket includes a field that indicates whether the search should be continued in this area or not. It can be a "Yes/No" marker.

Ex= Collision resolution with buckets & overflow area

Insert: $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$



★ Deletion

In chaining method, deleting an element leads to deletion of a node from a linked list holding the element.

Ex= Insert: A_1, A_4, A_2, B_4, B_1 and then delete A_4 & A_2.

Ans= After A_4 is deleted & position 4 is freed, we try to find B_4 by first checking position 4, But this position is now empty, so we may conclude that B_4 is not in the table.

Insert: A_1, A_4, A_2, B_4, B_1

0	
1	A_1
2	A_2
3	B_1
4	A_4
5	B_4
6	
7	
8	
9	

Delete A_4 and A_2

0	
1	A_1
2	
3	B_1
4	
5	B_4
6	
7	
8	
9	

Purging →

Purged hash Table

0	
1	A_1
2	B_1
3	
4	B_4
5	
6	
7	
8	
9	

- The same result occurs after deleting A_2 & marking cell 2 as empty. The search for B_1 is unsuccessful because if we are using linear probing, the search terminates at position 2.

- If we leave deleted keys in the table with markers indicating that they are not valid elements of table, any subsequent search for an element does not terminate prematurely. When a new key is inserted, it overwrites a key that is only a space filler.

- The table should be <u>purged</u> after a certain no. of deletions by moving undeleted elements to the cells occupied by deleted elements.

Self-Organising List

- A self-organising list is a list that reorders its elements dynamically based on some self-organising heuristic to improve average access time.

- This organisation depends on the configuration of data; thus, the stream of data requires reorganising the nodes already on the list.
  Only configuration of nodes is changed.

- The aim of a self-organising list is to improve efficiency of linear search by moving more frequently accessed element at or close to head of the list.

- The simplest implementation of a self-organising list is as a linked list.

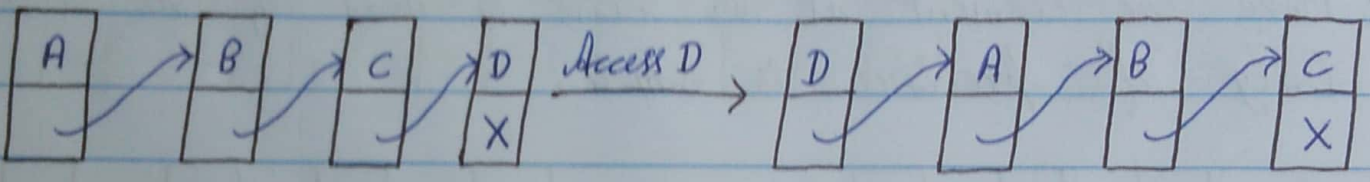
- In a self organising list, after the desired element is accessed, the list is automatically organised in such a way that next time when that element is accessed, the time required to reach it, is less than the previous search.

* The 4 different ways to self-organise the list are:-

1.) Move to front method

After the desired element is located, put it at the beginning of the list.

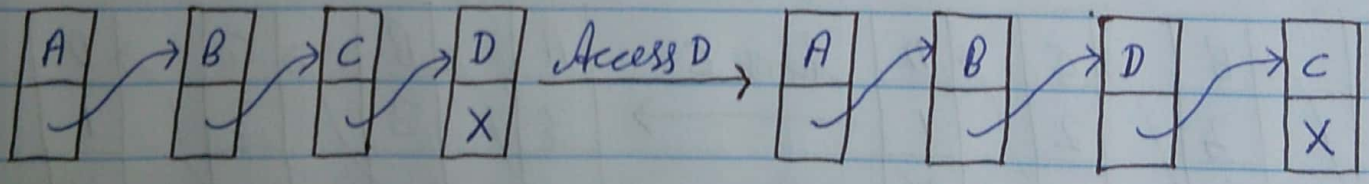
- This technique moves the element which is accessed to the head of the list.



2.) Transpose method

After the desired element is located, swap it with its predecessor unless it is of the head of the list.

- This technique involves swapping on accessed node with its predecessor.



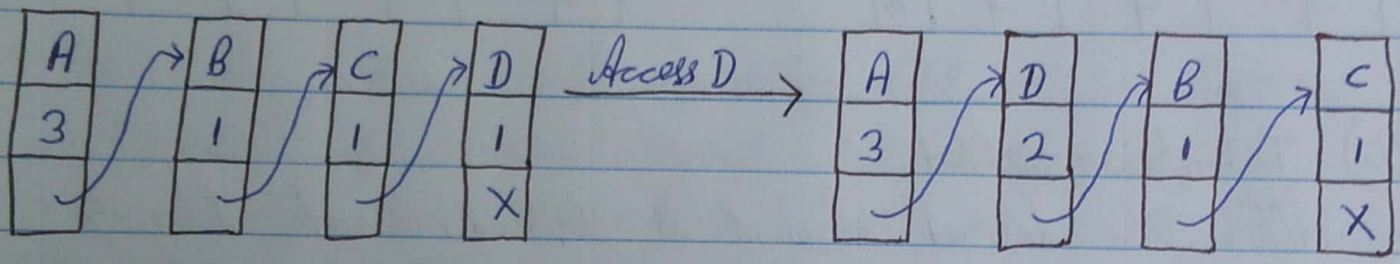
3.) Count method

Order the list by the number of times elements are being accessed.

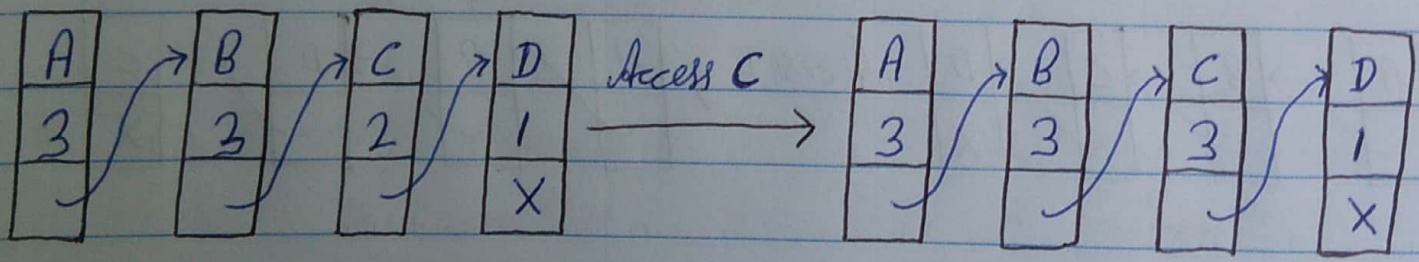
- In this technique, the number of times each element was searched for is counted.

- Every node keeps a seperate counter variable which is incremented every time it is called.
- The nodes are then rearranged according to decreasing count.
- Thus, the nodes of highest count i.e. most frequently accessed nodes are kept at the head of the list.

- Move the element at or close to head of the list, Only if previous is small.



- If previous is equal or greater, then no change is made to the list.



4.) Ordering method

Order the list using certain criteria natural for the information.

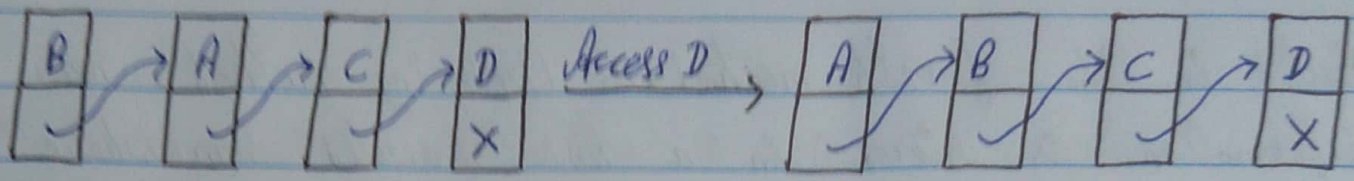
- This technique orders a list with a particular criterion such as alphabetical, ascending or descending order.

TutorialsDuniya.com

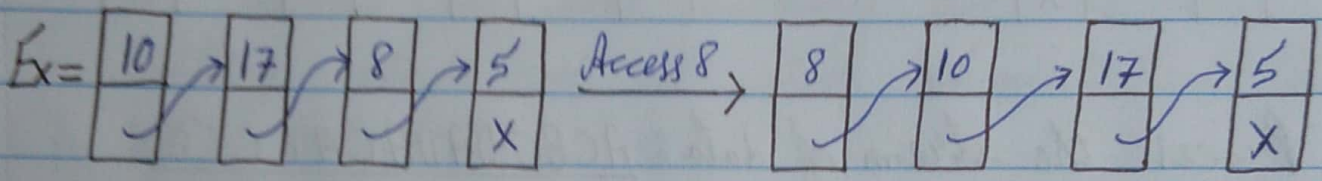
Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well



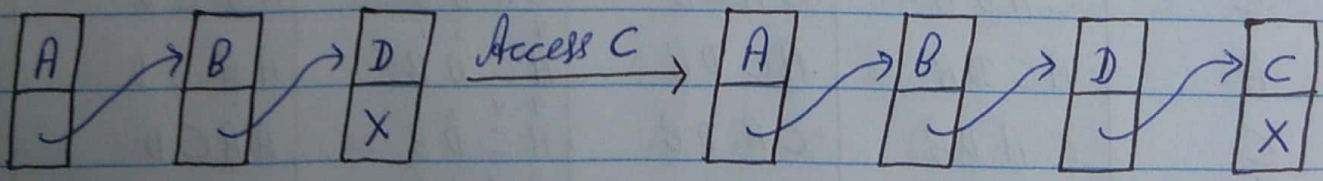
* In ordering method, once the desired element is accessed, the search can terminate without scanning the entire list.



→ If we want to access node 8, then all nodes upto 8 are organised to maintain the order of the list i.e. Increasing order.

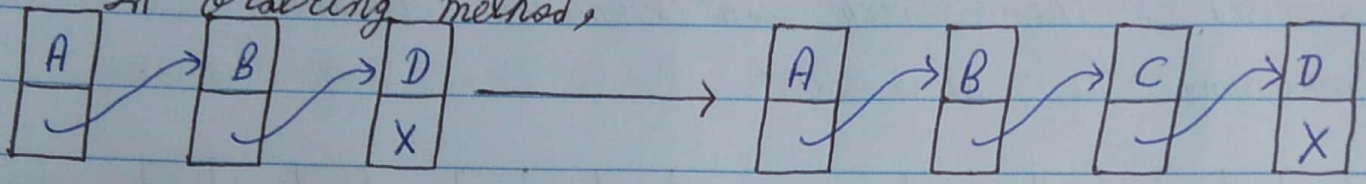
* In case when the desired element is not found, the first 3 methods i.e. move to front, Transpose, Count; stores new information in a node added to the end of the list.

In move to front, Transpose, Count method



- But in the fourth method i.e. Ordering method, new information is stored in a node inserted somewhere in the list to maintain the order of the list.

In ordering method,



Ex= Process the stream of data: ACBCDADACACCEE

Element Searched for	Move-to-front	Transpose	Count	Ordering
A	A	A	$\acute{A}$	A
C	A C	A C	$\acute{A}\,\acute{C}$	A C
B	A C B	A C B	$\acute{A}\,\acute{C}\,\acute{B}$	A B C
C	C A B	C A B	$^{2}C\,\acute{A}\,\acute{B}$	A B C
D	C A B D	C A B D	$^{2}C\,\acute{A}\,\acute{B}\,\acute{D}$	A B C D
A	A C B D	A C B D	$^{2}C\,^{2}A\,\acute{B}\,\acute{D}$	A B C D
D	D A C B	A C D B	$^{2}D\,^{2}C\,^{2}A\,\acute{B}$	A B C D
A	A D C B	A C D B	$^{3}A\,^{2}D\,^{2}C\,\acute{B}$	A B C D
C	C A D B	C A D B	$^{3}C\,^{3}A\,^{2}D\,\acute{B}$	A B C D
A	A C D B	A C D B	$^{4}A\,^{3}C\,^{2}D\,\acute{B}$	A B C D
C	C A D B	C A D B	$^{4}A\,^{4}C\,^{2}D\,\acute{B}$	A B C D
C	C A D B	C A D B	$^{5}C\,^{4}A\,^{2}D\,\acute{B}$	A B C D
E	C A D B E	C A D B E	$^{5}C\,^{4}A\,^{2}D\,\acute{B}\,\acute{E}$	A B C D E
E	E C A D B	C A D E B	$^{5}C\,^{4}A\,^{2}E\,^{2}D\,\acute{B}$	A B C D E

Recursion

- A recursive definition is used to define an object in terms of itself.

A recursive definition consists of 2 parts:-

1) Anchor or ground statement
   It defines the basic elements that are the building blocks of all other elements of the set are listed.
   - It contains a condition which helps the recursive loop breaks.

2) Inductive or Recursive statement
   This is the actual body of the function which recursively call itself again and again.
   - These rules are applied again and again to generate the final output.

* Recursive definitions serves 2 purposes:
  ① Generating new elements
  ② Testing, whether an element belongs to a set.

$$n! = \begin{cases} 1 & \text{if } n=0 \quad \text{(anchor)} \\ n(n-1)! & \text{if } n>0 \quad \text{(Inductive)} \end{cases}$$

☆ Application of Recursion

1.) Recursive definitions are used extensively in specification of grammers of programming language.
2.) Recursive definitions on most computers are eventually implemented using a run time stack.

☆ Activation Record

An activation record usually contains the following info:

1.) Values for all parameters to the function.
2.) Local variables that can be stored elsewhere.
3.) Return address to resume control by caller.
4.) A dynamic link, which is a pointer to the caller's activation record.
5.) Returned value for a function not declared as void.

Activation record of a function ⇒

Parameter & local variables
Dynamic link
Return address
Return value

- Activation record usually have a short lifespan because they are dynamically allocated at function entry & deallocated upon exiting.
- The run-time stack always reflects the current state of a function.

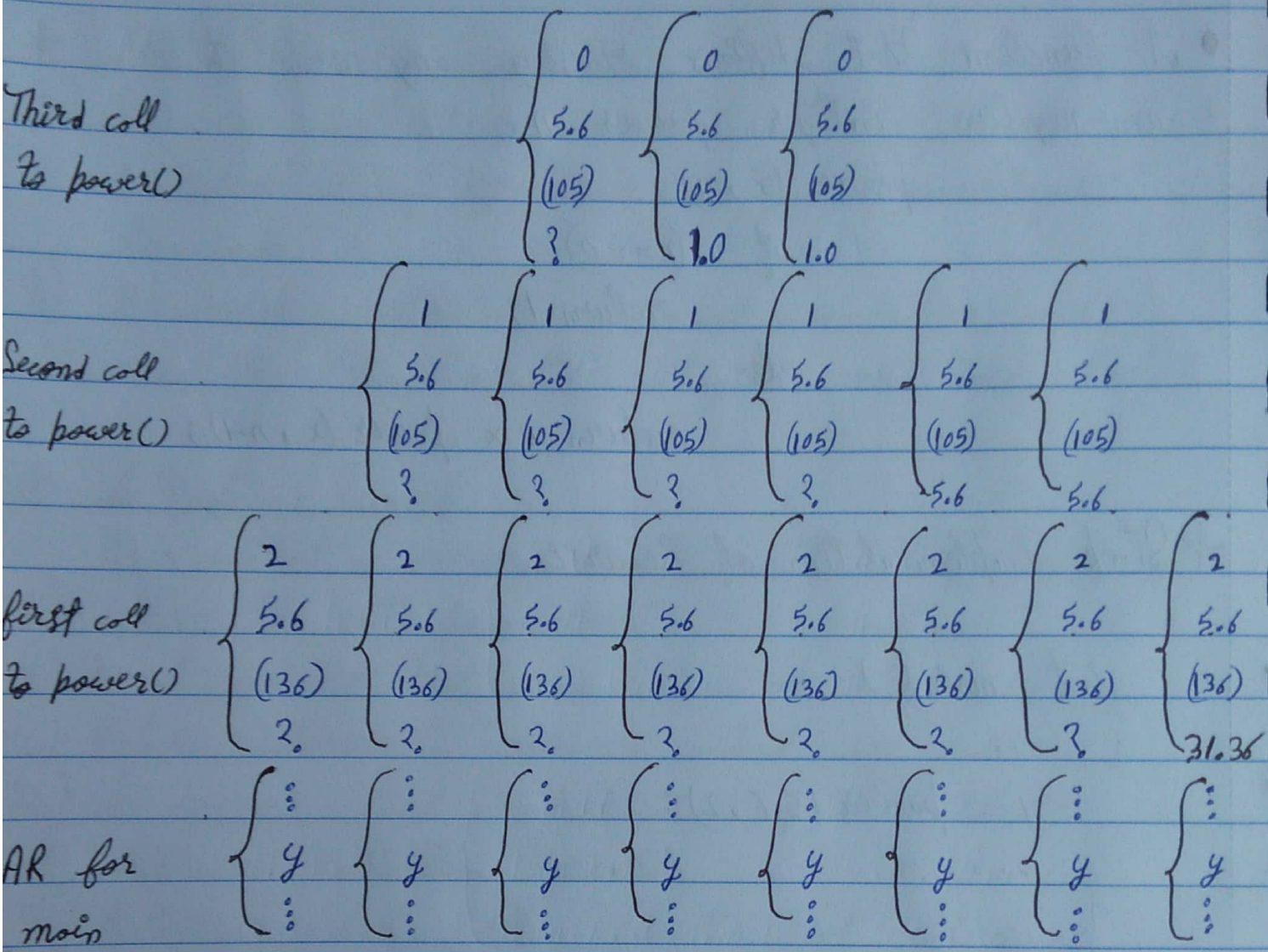
- A functions that defines raising any no. x to a non-negative integer power n.

```
power (x, n)
        if (n==0)
                return 1;
        else
                return x * power (x, n-1);
```

☆ Stack implementation of Recursion

```
int main()
{ ...
   y = power (5.6, 2);
   ...
}
```

```
call 1          power (5.6, 2)
call 2                power (5.6, 1)
call 3                      power (5.6, 0)
call 3                        1
call 2                  5.6
call 1          31.36
```



☆ Recursive Function

A function that calls itself directly or indirectly and approaches towards the terminating condition or base condition.

- Everytime the function calls itself, the called function should be smaller than caller function.
- Finally the called function becomes so small that it gives a direct solution.
- The smallest instance of the function is called base condition.

★ Types of Recursive functions

1) Direct Recursion

In direct recursion, the function body itself will have a base case and a recursive call to some function with smaller inputs value.

```
int factorial (fact, n)
{   if (n==0)                          // base case
    {  fact=1;
       return fact;
    }
    fact = n x factorial (fact, n-1)   // recursive call
    return fact;
}
```

2) Indirect Recursion

In indirect recursion, one recursive call will call another function, this recursive function call will call another, and so on, and finally the last function call calls the first one.

- It occurs when there is a chain of calls of the form,

$$f1() \xrightarrow{\text{calls}} f2() \xrightarrow{\text{calls}} f3() \xrightarrow{\text{calls}} f4() \xrightarrow{\text{calls}} f1()$$

3.) Nested Recursion

In nested recursion, one recursive call is nested within itself, and the recursive function call may be used as one of the parameter for the subsequent calls.

- One of the arguments to the recursive function is the recursive function itself.

$$h(n) = \begin{cases} 0 & \text{if } n=0 \\ n & \text{if } n>4 \\ h(2+h(2n)) & \text{if } n \leq 4 \end{cases}$$

$$A(n,m) = \begin{cases} m+1 & \text{if } n=0 \\ A(n-1,1) & \text{if } n>0, m=0 \\ A(n-1, A(n,m-1)) & \text{othermise} \end{cases}$$

Ex= Calculate A(1,2);

Ans= $A(1,2) = A(0, A(1,1))$

$A(1,1) = A(0, A(1,0))$

$A(1,0) = A(0,1)$

$A(0,1) = 1+1 = 2$

$A(0,2) = 2+1 = 3$

$A(0,3) = 3+1 = 4$

$A(1,2) = 4$

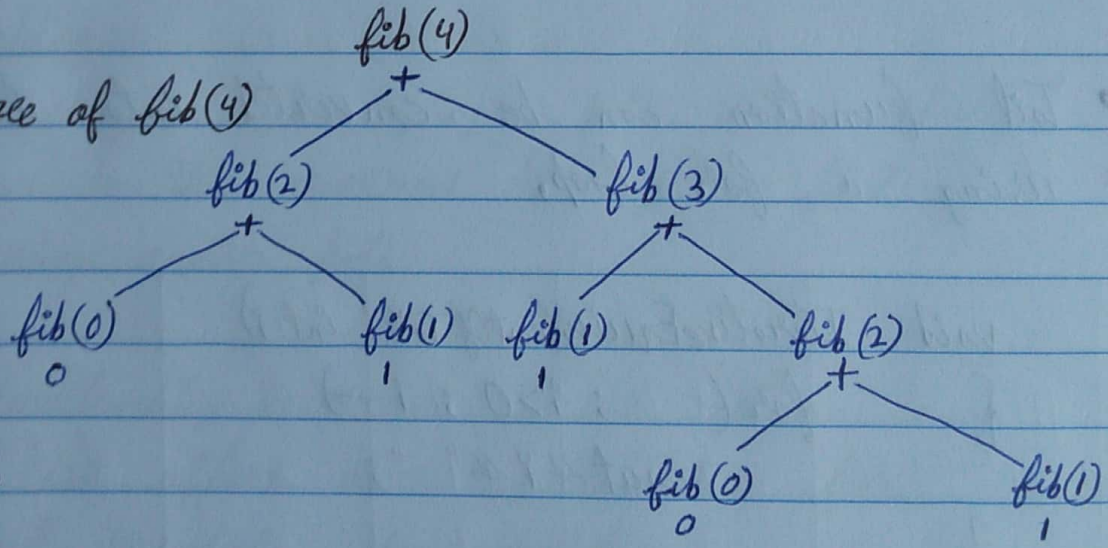
4.) Excessive Recursion

If recursion is too ~~to~~ deep (ex 5.6^{1000}) then we can run out of space on stack & program crashes.

- In order to avoid this, the system uses concept of trees to implement recursion.

```
unsigned int fib (int n)
{     if (n<2)
            return n;
      else  return (fib(n-2) + fib(n-1))
}
```

Recursion tree of fib(4)



fib(4) = 3

5.) Tail and Non Tail Recursion

A tail recursive call will only have 1 recursive call & this recursive call has to be at the end of the function.

- When the call is made, there are no statements left to be executed by the function & there are no earlier recursive calls.

- In Non-Tail recursion, the recursive call is not at the end of the recursive function definition & the recursive function can contain multiple recursive statements.

```
void Tail (int i)
{   if (i>0)
    {   cout << i << " ";
        Tail (i-1);
    }
}
```

```
void nonTail (int i)
{   if (i>0)
    {   nonTail (i-1);
        cout << i << " ";
        nonTail (i-1);
    }
}
```

- Tail function can be converted into iterative using a for loop,

```
void IterativeEquivalentOfTail (int i)
{   for ( ; i>0 ; i--)
        cout << i << " ";
}
```

- return func (x,y); is a Tail recursive, as no work is pending after recursive call, just return the value of the call.
- return func(x,y) +1; is non Tail recursive, as after recursive call, 1 has to be added to the result.

☆ Application of Non-Tail Recursion

1) Reversal of elements

```
void reverse()
{       char ch;
        cin.get(ch);
        if (ch != '\n')
        {   reverse();
            put(ch);
        }
}
```

☆ Backtracking

In backtracking, after trying 1 path unsuccessfully, we return to this crossroad & try to find a solution using another path.

- Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem.

Ex = 8 queens problem

- The 8 queens problem attempts to place 8 queens on a chess board in a way that no queens is stuck attacking any other.

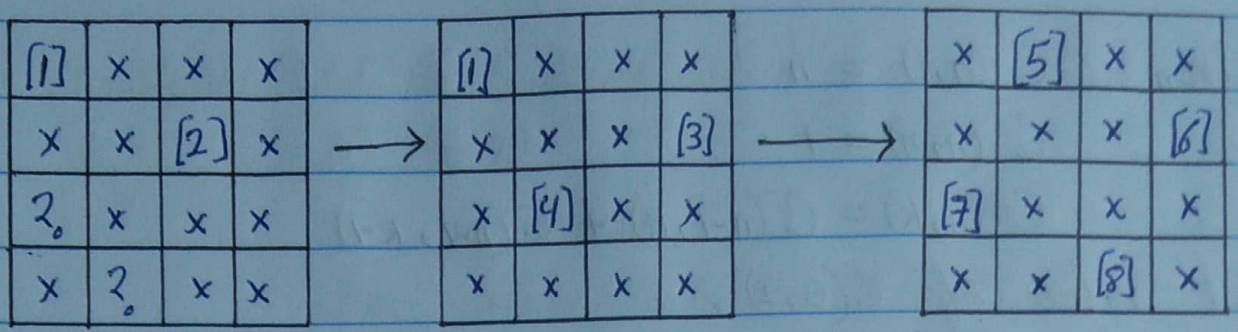
- The rules of chess says that a queen can take another piece if it lies on same row, on same column or on same diagonal as queen.

* Backtracking Algorithm

```
putQueen (row)
    for every position col on same row
        if position col is available
            place the next queen in position col;
            if (row<8)
                putQueen (row+1);
            else success;
            remove the queen from position col;
```

- 4 Queen problem

Move	Queen	Row	Col	
[1]	1	0	0	
[2]	2	1	2	failure
[3]	2	1	3	
[4]	3	2	1	failure
[5]	1	0	1	
[6]	2	1	3	
[7]	3	2	0	
[8]	4	3	2	



Successfully placed 4 queens

- Trace of calls to putQueen() to place 4 queens

putQueen(0)
  col=0;
  putQueen(1)
    col=0;
    col=1;
    col=2;
    putQueen(2)
      col=0;
      col=1;
      col=2;
      col=3;
    col=3;
    putQueen(2)
      col=0;
      col=1;
      putQueen(3)
        col=0;
        col=1;
        col=2;
  col=3;
  col=2;
  col=3;
col=1;
putQueen(1)
  col=0;
  col=1;
  col=2;
  col=3;
  putQueen(2)
    col=0;
    putQueen(3)
      col=0;
      col=1;
      col=2;
      Success

Ex = Given: $C(n,1) = n$

$C(n,n) = 1$

$C(n,k) = C(n-1,k) + C(n-1,k-1)$

Find value of $C(5,2)$.

[4 marks]

Ans = $C(5,2) = C(4,2) + C(4,1)$

$= C(3,2) + C(3,1) + 4$

$= C(2,2) + C(2,1) + 3 + 4$

$= 1 + 2 + 3 + 4$

$= 10$ //

Ex = Given: $C(n,k) = 1$ if $k=0$ or $k=n$

$C(n,k) = C(n-1,k-1) + C(n-1,k)$ otherwise

Find the value of $C(3,2)$ and $D(4,4)$.

[5 marks]

Ans = $C(3,2) = C(2,1) + C(2,2)$

$= C(1,0) + C(1,1) + C(2,2)$

$= 1 + 1 + 1$

$= 3$ //

$D(4,4) = 1$ // [$\because$ as $n=k$]

Ex = Consider the following recursive function:

This function produces 0, 1, 1, 2, 3, 5, 8, 13, ...

```
unsigned int Fib (unsigned int n)
{       if (n<2)
            return n;
        else return (Fib (n-2) + Fib (n-1));
}
```

How many recursive calls & additions will be performed to compute Fib(5).

Draw the tree showing all calls generated by Fib(5).

[5 marks]

Ans = Fib(5) = fib(3) + fib(4)

= fib(2) + fib(1) + fib(3) + fib(2)

= fib(0) + fib(1) + fib(1) + fib(2) + fib(1) + fib(1) + fib(0)

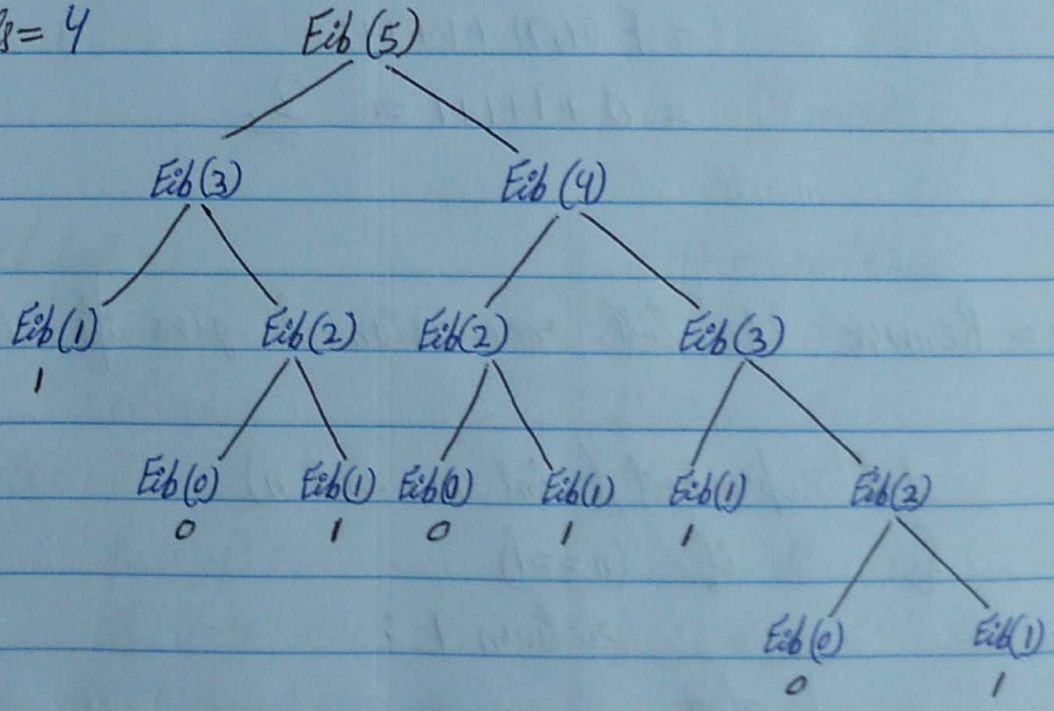
= 0 + 1 + 1 + fib(1) + fib(0) + 1 + 1 + 0

= 0 + 1 + 1 + 1 + 0 + 1 + 1 + 0

= 5

Recursive calls = 4

Additions = 8



2016

Ex= $$F(a,b) = \begin{cases} 0 & \text{if } a < b \\ F(a-b,b)+1 & \text{if } b \le a \end{cases}$$

Find the value of (i) F(2,3)
(ii) F(14,3)
(iii) F(25,7) [6 marks]

Ans= (i) F(2,3) = 0 // [∵ 2<3]

(ii) F(14,3) = F(11,3) + 1
= F(8,3) + 1 + 1
= F(5,3) + 1 + 1 + 1
= F(2,3) + 1 + 1 + 1 + 1
= 0 + 1 + 1 + 1 + 1 = 4 //

(iii) F(25,7) = F(18,7) + 1
= F(11,7) + 1 + 1
= F(4,7) + 1 + 1 + 1
= 0 + 1 + 1 + 1 = 3 //

Ex= Remove the tail recursion & give its iterative fn.

```
int product (int m, int n)
{   if (n==1)
        return 1;
    else
        return (m + product (m, n-1));
}
```

[4 marks]

```
int product (int m, int n)
{ int i, sum = 1;
  if (n==1)
      return sum;
  else
  { for (i=1; i<=n; i++)
        sum += m;
    return sum;
  }
}
```

product (2,1) = 1
product (2,2) = 3
product (2,3) = 5
product (2,4) = 7
product (3,2) = 4
product (3,3) = 7

2017

Ex= Write its recursive function;

```
void f (int n)
{   for (int i=1; i<=n; i++)
    {   if (i%2==0)
            cout<<i*i*i;
    }
}
```

[4 marks]

Ans=

```
void f(int n)
{   if (n==1)
        return;
    else
    { f(n-1);
      if (n%2==0)
          cout<<n*n*n;
    }
}
```

~~f(1) = nothing~~
f(2) = 8
f(3) = 8
f(4) = 8 64
f(5) = 8 64
f(6) = 8 64 216

ADTs and Arrays

- An array is a variable that holds multiple values of the same data type.

- **Abstract data type:** Each instance of the data object "array" is a set of pairs of the form (index, value)

- create(): Creates an empty array.
- Store(index, value): Adds a pair to set S if a pair with some index already exists, deletes the old pair.
- Retrieve(index): Retrieves the pair with this index.

These 3 functions together with the data object array define the abstract data type Array.

★ One dimensional Array

$$\boxed{loc[a] = ba + w\,(a - lb)}$$

ba = base address
w = data size
lb = lower limit
a = subscript of element whose address is to be found.

Ex = Given the base address of an array B[1300.... 1900] as 1020 & size of each element is 2 bytes in memory. Find the address of B[1700].

Ans = Address of B[1700] = 1020 + 2[1700 - 1300]
= 1020 + 2 × 400
= 1020 + 800
= 1820 //

☆ Two dimensional Array

1) Row Major Order

Within each row, numbers are assigned from left to right.

Row 0	1	2	3
Row 1	4	5	6
Row 2	7	8	9

3×3 =

1	2	3	4	5	6	7	8	9

Row 0 Row 1 Row 2

$$A[i][j] = B + W[N(i - Lr) + (j - Lc)]$$

2) Column Major Order

Within a column, numbers are assigned from top to bottom.

1	2	3
4	5	6
7	8	9

3×3 =

1	4	7	2	5	8	3	6	9

column 0 column 1 column 2

C1 C2 C3

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

$$A[i][j] = B + W((i - L_r) + M(j - L_c))$$

B = Base address
W = Data size
M = No. of rows in given matrix
N = No. of columns in given matrix
L_r = Lower limit of row
L_c = Lower limit of column
i = Row subscript of element whose address is to be found
j = Column subscript of element whose address is to be found

Ex = An array A[-15....10, 15....40] requires 1 byte of storage. If beginning location is 1500. Determine the location of A[15][20].

[3 marks]

Ans = $M = (U_r - L_r) + 1$
$= 10 - (-15) + 1$
$= 26$

$N = (U_c - L_c) + 1$
$= 40 - 15 + 1$
$= 26$

RMO $A[15][20] = 1500 + 1[26(15 - (-15)) + (20 - 15)]$
$= 1500 + 1[780 + 5]$
$= 2285$

CMO $A[15][20] = 1500 + 1[(15 - (-15)) + 26(20 - 15)]$
$= 1500 + 1[30 + 130]$
$= 1660$

* The lower bound is always 0 & upper bound is size-1.

2016

Ex = The beginning address of the array is 100 and every element requires 4 bytes of storage. The 2D array is defined as int A[6][6]. Calculate the address of A[2][4].

[4 marks]

Ans = B = 100, W = 4, A[6][6], $L_r = 0$, $L_c = 0$

$M = U_r - L_r + 1$ $\quad N = U_c - L_c + 1$

$= 6 - 0 + 1$ $\quad = 6 - 0 + 1$

$= 7$ $\quad = 7$

RMO $\quad A[2][4] = 100 + 4[7(2-0) + (4-0)]$

$= 100 + 4[14 + 4]$

$= 172$

CMO $\quad A[2][4] = 100 + 4[(2-0) + 7(4-0)]$

$= 100 + 4[2 + 28]$

$= 220$

2017

Ex = Calculate the address of A[3][2] of the 2D array defined as A[5][5], if beginning address of array is 400 and every element requires 4 bytes of storage.

Ans = B = 400, W = 4, M = 6, N = 6 [4 marks]

RMO $A[3][2] = 400 + 4[6(3-0) + (2-0)]$

$= 400 + 4[18 + 2]$

$= 480$

CMO $A[3][2] = 400 + 4[(3-0) + 6(2-0)]$

$= 400 + 4[3 - 12]$

$= 460$

✱ Three dimensional Array

RMO $A[i][j][k] = B + W[L_3(E_1 L_2 + E_2) + E_3]$

CMO $A[i][j][k] = B + W[L_1(E_3 L_2 + E_2) + E_1]$

$L = (ub - lb) + 1$

$E_1 = i - lb$

$E_2 = j - lb$

$E_3 = k - lb$

Ex = A 3-D array is declared using (2...8, -4...1, 6...10), Find the address of A[5][1][8].
If the base address is 200 & every element requires 4 bytes of storage.

Ans = $E_1 = 5-2 = 3$ | $L_1 = 8-2+1 = 7$

$E_2 = -1-(-4) = 3$ | $L_2 = 1-(-4)+1 = 6$

$E_3 = 8-6 = 2$ | $L_3 = 10-6+1 = 5$

RMO $A[5][-1][8] = 200 + 4[5(3\times6+3)+2]$

$= 200 + 4[105+3]$

$= 200 + 428$

$= 628$

CMO $A[6][-1][8] = 200 + 4[7(2\times6+3)+3]$

$= 200 + 4[105+3]$

$= 200 + 432$

$= 632$

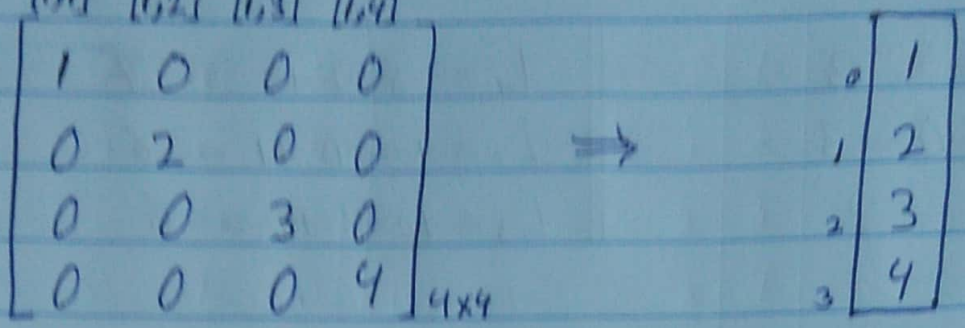
★ Special Matrices

① Diagonal Matrix

A diagonal matrix M is diagonal if and only if $M(i,j) = 0$ for $i \neq j$.

- That is, the main diagonal of the matrix where i (row of the matrix) is equal to j (column of matrix) has values & all other elements of the matrix are 0.

- Number of elements in a $n \times n$ diagonal matrix are n.



- Store()
  if (i==j)
    D[i-1] = x;

Retrieve()
if (i==j)
  return D[i-1];

where x is the element to be inserted.

2.) Tridiagonal Matrix

A matrix M is tridiagonal if and only if $M(i,j) = 0$ for $|i-j| > 1$.

- That is, the main diagonal of the matrix where i is equals to j and the diagonal above the main diagonal and diagonal below the main diagonal has values; all other elements of the matrix are 0.

- Number of elements in a $n \times n$ tridiagonal matrix are $3n-2$.

- For elements on lower diagonal: T[i-2]
- For elements on main diagonal: T[n+i-2]
- For elements on upper diagonal: T[2n+i-2]

$$\begin{bmatrix} 1 & 8 & 0 & 0 \\ 5 & 2 & 9 & 0 \\ 0 & 6 & 3 & 10 \\ 0 & 0 & 7 & 4 \end{bmatrix}_{4\times4} \Longrightarrow$$

0	5
1	6
2	7
3	1
4	2
5	3
6	4
7	8
8	9
9	10

lower, main, upper

- Store()

```
if (i-j==1)        // lower
    T[i-2] = x;
if (i-j==0)        // main
    T[n+i-2] = x;
if (i-j== -1)      // upper
    T[2n+i-2] = x;
```

- Retrieve()

```
if (i-j==1)    return T[i-2];
if (i-j==0)    return T[n+i-2];
if (i-j==-1)   return T[2n+i-2];
```

3) Lower Triangular Matrix

A matrix M is lower triangular if and only if $M(i,j) = 0$ for $i < j$.

That is, the area below the main diagonal has values & all other elements are 0.

Number of elements in a $n \times n$ lower triangular matrix are $\frac{n(n+1)}{2}$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix}_{4\times4}$$

$\Rightarrow$

	RMO
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10

	CMO
0	1
1	2
2	4
3	7
4	3
5	5
6	8
7	6
8	9
9	10

- Store ()

if $(i \geq j)$

$$L\left[\frac{i(i-1)}{2} + (j-1)\right] = x;$$ // RMO

// CMO $$L\left[\frac{j(j-1)}{2} + (i-1)\right] = x;$$

- Retrieve ()

if $(i \geq j)$

return $L\left[\frac{i(i-1)}{2} + (j-1)\right];$ // RMO

return $L\left[\frac{j(j-1)}{2} + (i-1)\right];$ // CMO

4.) Upper Triangular Matrix

A matrix M is upper triangular matrix if and only if $M(i,j) = 0$ for $i > j$.

- That is, the area above the main diagonal has values & other elements are 0.

- Number of elements in a $n \times n$ upper triangular matrix are $\frac{n(n+1)}{2}$.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{bmatrix}_{4\times4} \Longrightarrow$$

	RMO
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10

	CMO
0	1
1	2
2	5
3	3
4	6
5	8
6	4
7	7
8	9
9	10

- Store ()
  if $(i \le j)$
    $A\left[\frac{i(i-1)}{2} + (j-1)\right] = x$; // RMO

- Retrieve ()
  if $(i \le j)$
    return $A\left[\frac{i(i-1)}{2} + (j-1)\right]$; // RMO

5.) Symmetric Matrix

A matrix M is symmetric if and only if $M(i,j) = M(j,i)$ for all i and j.

- We can store a symmetric matrix in a 1-D array by storing either the lower or upper triangle of the matrix using one of the triangular matrix.

- Number of elements in a $n \times n$ symmetric matrix are $\frac{n(n+1)}{2}$.

$$\begin{bmatrix} 1 & 6 & 0 & 8 \\ 6 & 2 & 5 & 0 \\ 0 & 5 & 3 & 7 \\ 8 & 0 & 7 & 4 \end{bmatrix}_{4\times4} \Longrightarrow$$

	RMO
0	1
1	6
2	2
3	8
4	5
5	3
6	5
7	0
8	7
9	4

	CMO
0	1
1	6
2	0
3	8
4	2
5	5
6	0
7	3
8	7
9	4

- Store ()

  if $(i \geq j)$

  $S\left[\frac{i(i-1)}{2} + (j-1)\right] = x;$

  // lower triangular matrix

- Retrieve ()

  if $(i \geq j)$

  return $S\left[\frac{i(i-1)}{2} + (j-1)\right];$

☆ Sparse Matrix

A sparse matrix is a matrix which contains very few non-zero elements.

- A matrix that is not sparse is dense matrix.

- There is no need to store the zero values of the sparse matrix. Only the non-zero elements of the sparse matrices need to be stored.

- A sparse matrix can be represented by 2 ways:

(i) Array representation

(ii) Linked representation

1) Array Representation

In this representation, we consider only the non-zero values along with their row, column and index values.

- The 0^{th} row in the table stores total rows, total columns and total non-zero values in the table.

Ex = Consider a matrix of size 4x8 containing 9 non-zero value.

$$\begin{bmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 \\ 0 & 6 & 0 & 0 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 9 & 0 & 8 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{4\times8}$$

Total rows: 4
Total columns: 8
Total non-zero values: 9

RMO

Row	Column	Value
4	8	9
1	4	2
1	7	1
2	2	6
2	5	7
2	8	3
3	4	9
3	6	8
4	2	4
4	3	5

CMO

Row	column	Value
4	8	9
2	2	6
4	2	4
4	3	5
1	4	2
3	4	9
2	5	7
3	6	8
1	7	1
2	8	3

2) Linked Representation

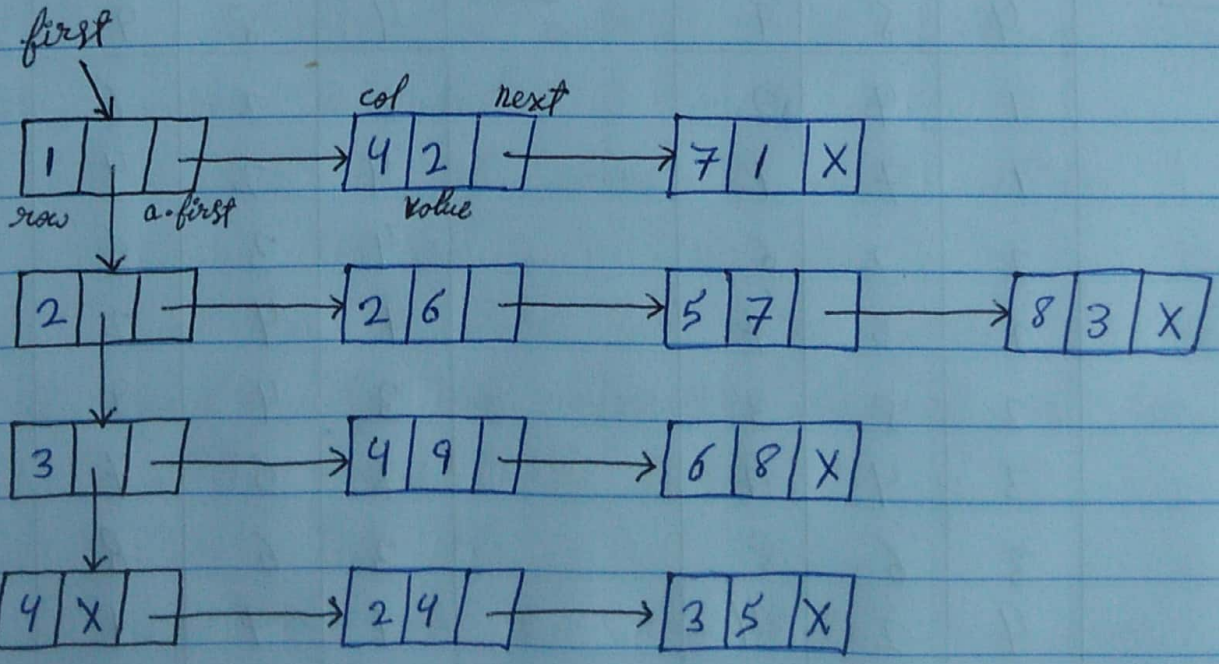
In linked representation, we use linked list data structure to represent a sparse matrix.

- A drawback of using 1-D array representation is that the number of non-zero elements should be known in advance. This problem can be overcome by using linked representation of sparce matrix

- This representation links together the non-zero entries in each row to form a chain.

- The first node in each row represents the Row number and other nodes represents a non zero term of sparse matrix & has 3 fields — col, value, next.

Linked Representation of previous matrix



Stack

- A stack is a linear data structure containing data items of similar type in which the elements are added and removed only from 1 end called "Top".

Push: Insertion of an element on Top of the stack.
Pop: Removal of an element ~~on~~ from top of the stack.

- A stack is a Last In First Out [LIFO] structure.

- A stack can be implemented as an array or as a linked list.

☆ Stack as an Array

```
int Push(int Stack[], int &top, int ele)
{       top=-1;
        if (top == size-1)
                cout<<" Overflow";
        else
        {       top++;
                Stack [top] = ele;
        }
        return 0;
}
```

In case the array is full & no new element can be accomodated, this is called Overflow.

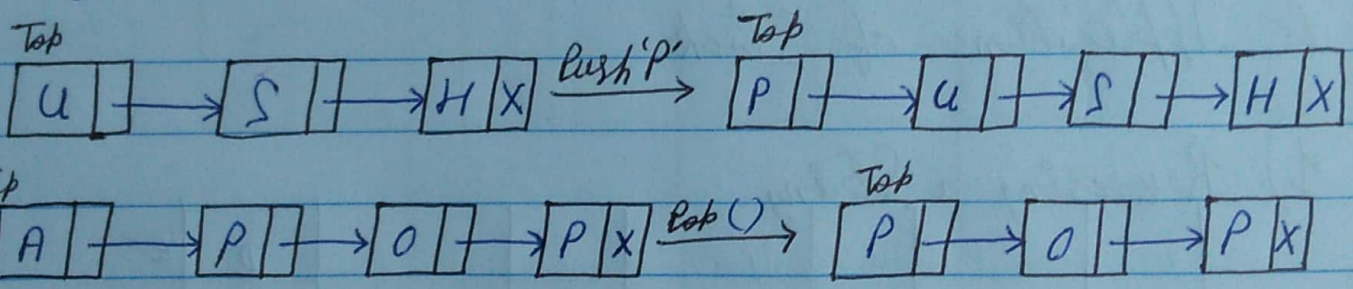
In case the last element is popped, the stack becomes empty. If one tries to delete an element from an empty stack, this is called Underflow.

```
int Pop(int Stack[], int &Top)
{     int save;
      if (Top==-1)                    // underflow
            return -1;
      else
      {     ~~int~~ save = Stack[Top];
            Top--;
      }
      return save;
}
```

★ Stack as a Linked list [Linked Stack]

```
void Push(int ele)
{     node *p;
      p = new node(ele);
      if (top==NULL)
            Top=p;
      else {      p->next = Top;
                  Top=p;
            }
```

```
int pop()
{       node *temp;
        int save = top->info;
        temp = top;
        if ( top == NULL)
                exit();
        else if ( top->next == NULL)
                        top = NULL;
        else    top = top->next;
        delete temp;
        return save;
}
```



☆ Generic Stack

A generic stock is a C++ language construct that allows the compiler to generate multiple versions of a class type or a function by allowing parameterised type. This can be achieved using template parameters.

```
template <class stacktype>
class stack
{       stacktype elements[10];
```

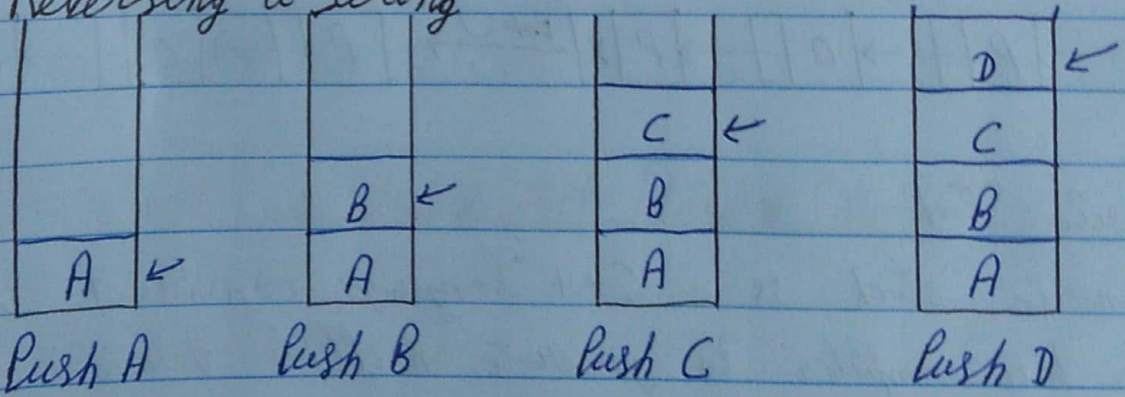
```
public: void push (stacktype ele)
        stacktype pop();
};
```

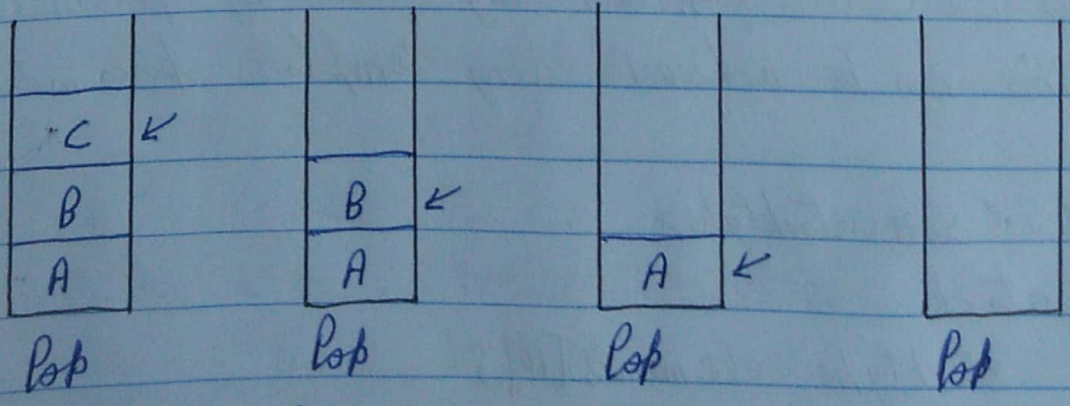
```
template <class stacktype>
void stack<stacktype> :: push (stacktype ele)
{ ..... }
```

```
template <class stacktype>
stacktype stack<stacktype> :: pop()
{ ..... }
```

☆ Applications of Stack

1) Reversing a String

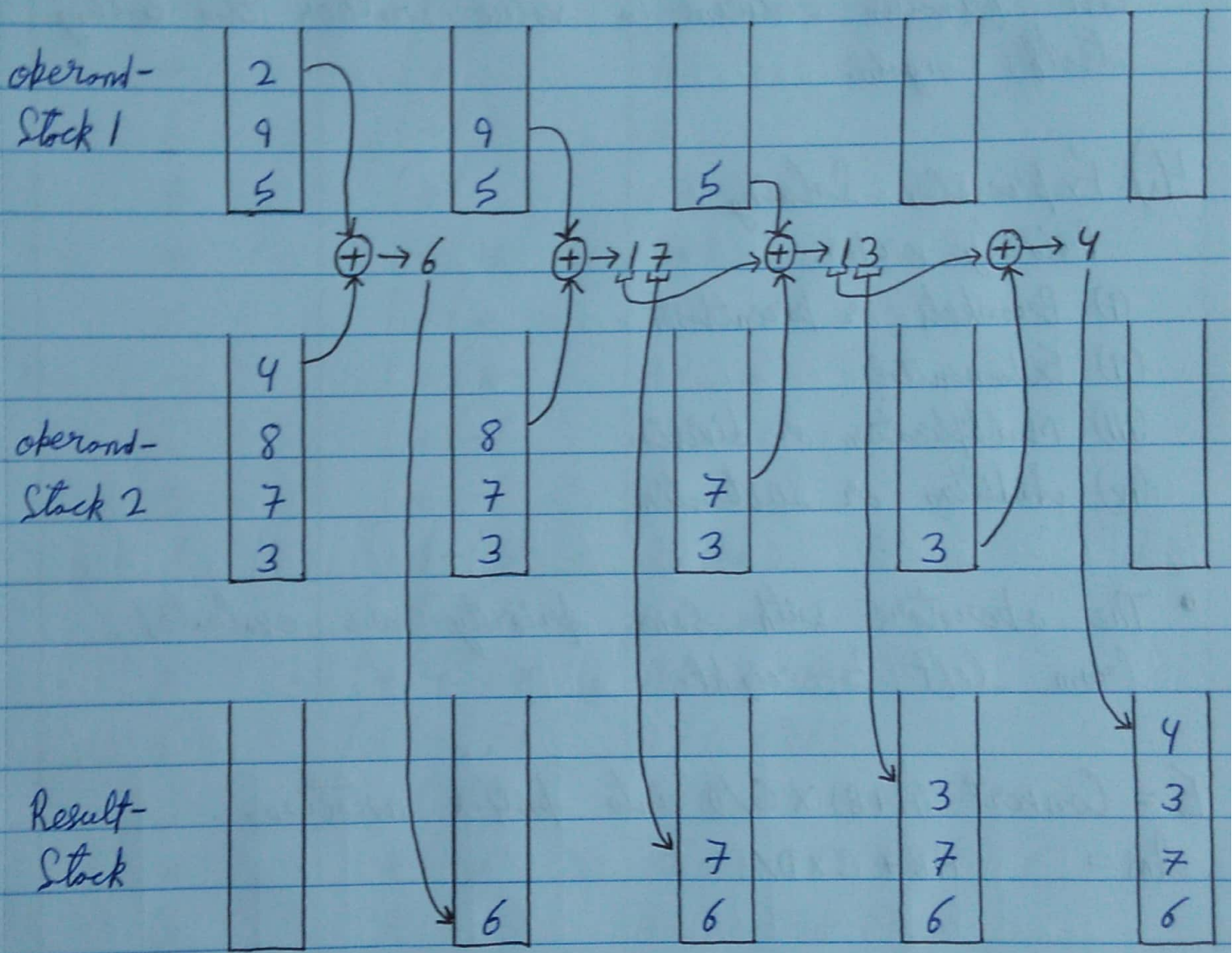




ABCD → DCBA

2.) Adding 2 large numbers

			①	
592	2	9	5	1
+3784	+4	+8	+7	+3
4376	6	17	13	4



3.) Stack in Recursion

When functions call one another, they are added to the call stack. The call stack contains all of its functions currently being called. When a function completes its execution or returns a value;

it is removed from the call stack and the function which called it resumes executing.

- Recursive functions add themselves to the call stack repeatedly until some condition is reached that the last function returns a value rather than calling itself again.

4.) Expression Solving

Priority order

(i) Brackets or parenthesis
(ii) Exponential
(iii) Multiplication or division
(iv) Addition or Subtraction

- The operators with same priority are evaluated from left to right.

Ex = Convert (A+B) x C/D into postfix notation.
Ans = AB+CxD/

Ex = Evaluate the postfix exp: AB+CxD/ if A=2, B=3, C=4 & D=5.
Ans = 4

Ex = Convert A+(B*C-(D/E↑F)*G)*H) into postfix form showing stack status after every step in tabular form.

Ans= (A+(B*C-(D/E↑F)*G)*H)

Symbol Scanned	Stack	Expression
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
↑	(+(-(/↑	ABC*DE
F	(+(-(/↑	ABC*DEF
)	(+(-	ABC*DEF↑/
*	(+(-*	ABC*DEF↑/
G	(+(-*	ABC*DEF↑/G
)	(+	ABC*DEF↑/G*-
*	(+*	ABC*DEF↑/G*-
H	(+*	ABC*DEF↑/G*-H
)		ABC*DEF↑/G*-H*+

Postfix form: ABC*DEF↑/G*-H*+

Ex = Convert into postfix notation.

(i) $(A+(B*C))/(C-(D*B))$

(ii) $((A+B)*C/D+E\uparrow F)/G$

(iii) $A*(B+(C+D)*(E+F)/G)*H$

(iv) $A+[(B+C)+(D+E)*F]/G$

(v) $-A+B*C\uparrow(D*E)/F$

Ans = (i) $ABC*+CDB*-/$

(ii) $AB+C*D/EF\uparrow+G/$

(iii) $ABCD+EF+*G/+*H*$

(iv) $ABC+DE+F*+G/+$

(v) $A-BCDE*\uparrow*F/+$

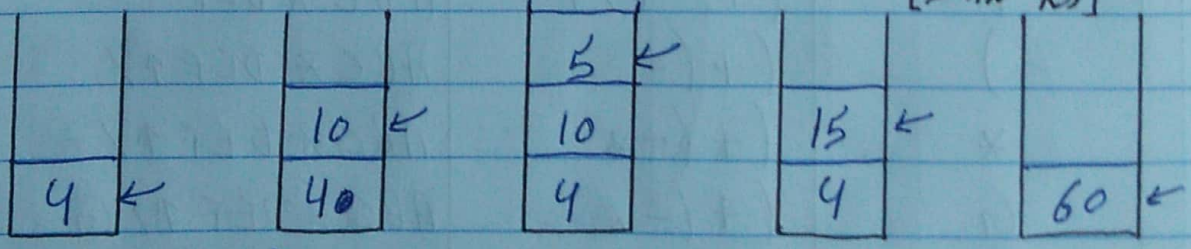
2017

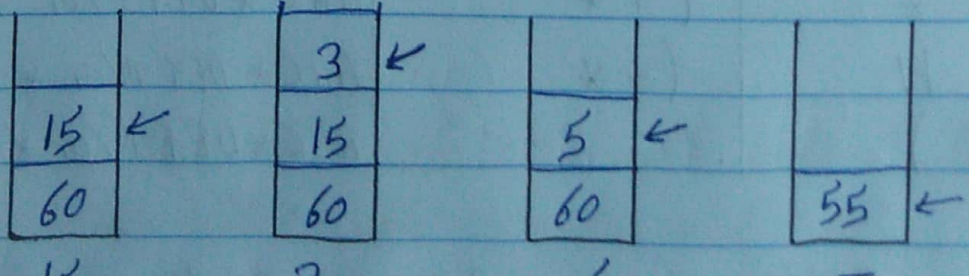
Ex = Evaluate the following postfix exp. using a stack. Show the contents of stack after every step.

4 10 5 + * 15 3 / −

[5 marks]

Ans =





TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Queue

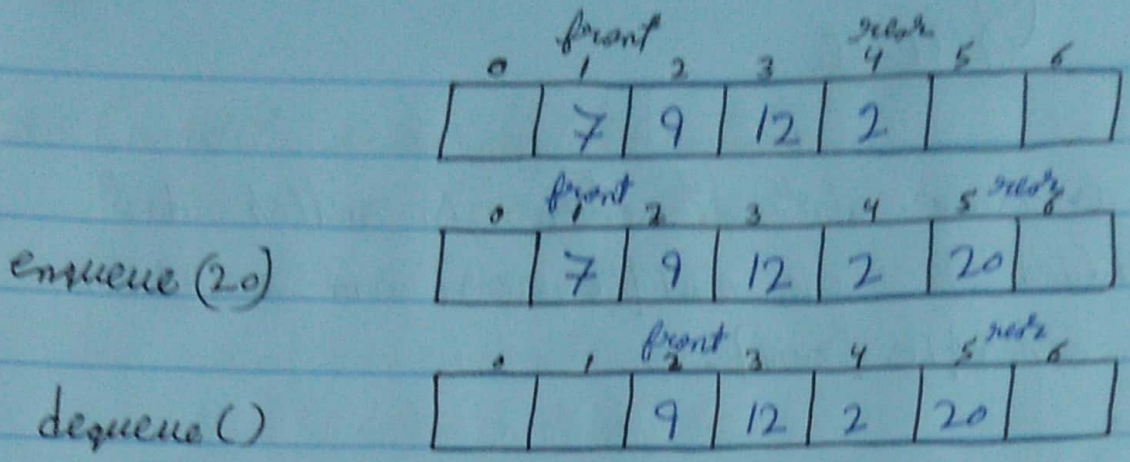
- Queue is a linear data structure in which data can be added to one end [i.e. rear] and retrieved from the other [i.e. front].
- Insertion takes place at rear end and deletion at front end.

Enqueue(): Insertion of an element at the rear of the list queue.
Dequeue(): Deletion of an element from the front of the queue.

- A queue is a First In First Out [FIFO] structure.

☆ Queue as an Array

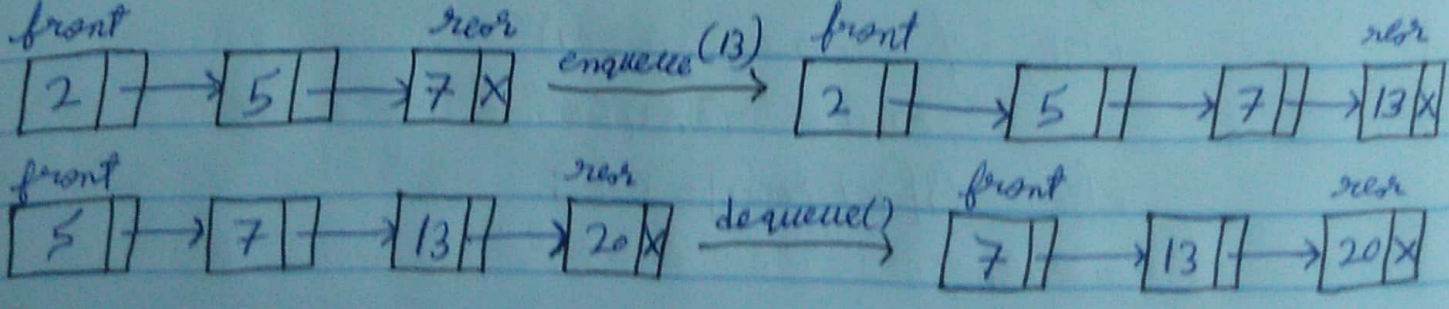
```
* Enqueue (Queue[], front, rear, ele)
{       if (rear == NULL)
        {       rear = front = 0;
                Q[0] = ele;
        }
        else if (rear == size-1)
                cout << "Overflow";
        else    Q[++rear] = ele;
        return;
}
```




```
* int Dequeue(Queue[], front, rear)
  {   if (front == NULL)
          cout << "Underflow";
      else
      {   ele = Q[front];
          if (front == rear)
              front = rear = NULL;
          else front = front + 1;
      }
      return ele;
  }
```

☆ Queue as a Linked list [Linked Queue]

Linked queue can be represented as singly linked list and doubly linked list with 2 pointers "front" and "rear".



* void Enqueue (int ele)

```
{
    node *p;
    p = new node (ele);
    p->next = NULL;
    if (front = NULL)
        front = rear = p;
    else
    {
        rear->next = p;
        rear = p;
    }
}
```

* int Dequeue ()

```
{
    node *temp;
    temp = front;
    int save = temp->info;
    if (front == NULL)
        cout << "Underflow";
    else if (front == rear)
        front = rear = NULL;
    else    front = front->next;
    delete temp;
    return save;
}
```

☆ Generic Queue

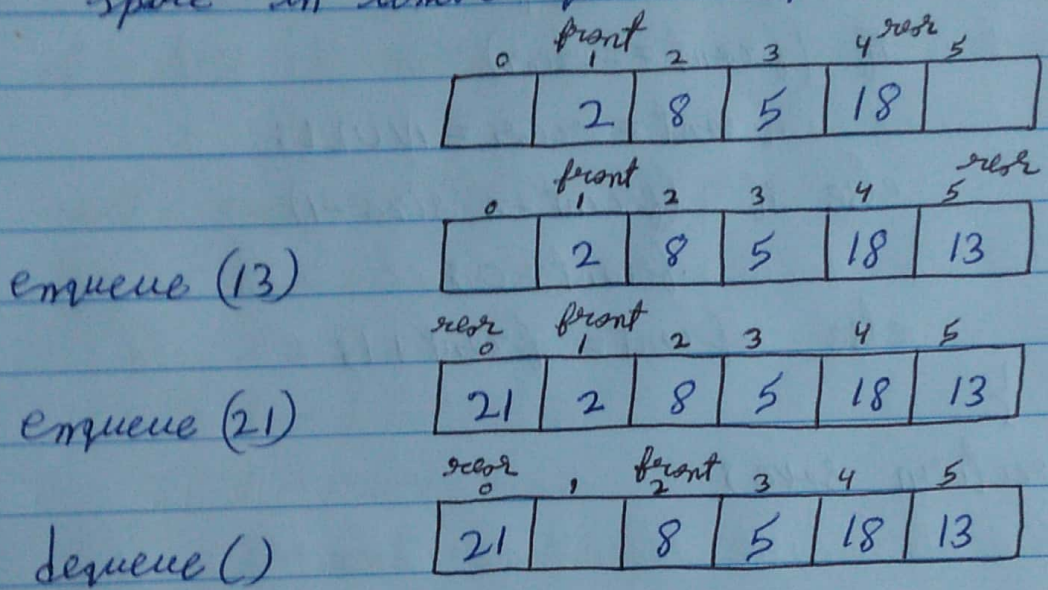
When some queue can be used to hold different type of data.

```
Template < class Queuetype >
class queue
{       Queuetype elements[10];
        public : void enqueue (Queuetype);
                 Queuetype dequeue ();
                 queue ();
};

// Enqueue
  Template < class Queuetype >
  void queue < Queuetype > :: enqueue (Queuetype)
  { ..... }

// Dequeue
  Template < class Queuetype >
  Queuetype queue < Queuetype > :: dequeue ()
  { ..... }

// Queue constructor
  Template < class Queuetype >
  queue < Queuetype > :: queue ()
  { ..... }
```

★ Circular Queue

Circular queues are queues implemented in circular form rather than a straight line.

- Circular queues overcome the problem of unutilised space in linear queues implemented as arrays.



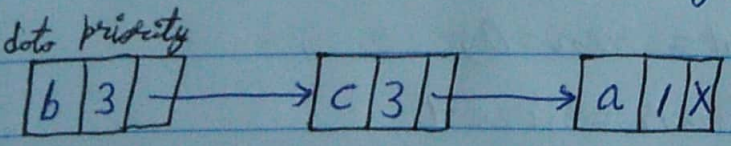

```
* void Enqueue_CQ ( int CQueue[], int ele)
  { if ((front == 0 && rear == size-1) || (front = rear+1))
        cout << "Overflow";
    else
    { if (front == NULL)
          front = rear = 0;
      else if (rear == size-1)
          rear = 0;
      else rear = rear + 1;
    }
    CQueue[rear] = ele;
  }
```

```
* int dequeue-CQ( int CQueue[])
  {
      int save;
      if (front==NULL)
          cout << "Underflow";
      else
      {
          save = CQueue[front];
          if (front == rear)
              front = rear = NULL;
          else if (front == size-1)
              front=0;
          else front = front+1;
      }
      return save;
  }
```

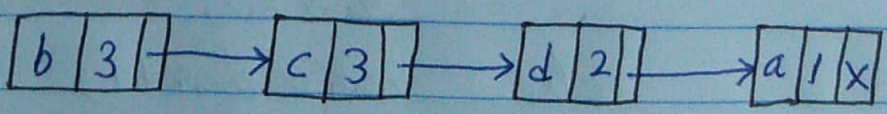
☆ Priority Queue

A priority queue is an abstract data type (ADT) which have elements arranged in the queue according to their priority.

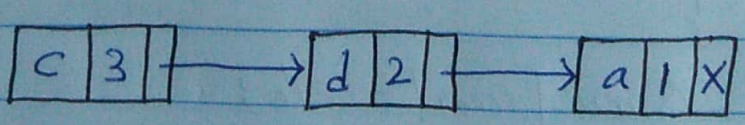
- Each element has a "priority" associated with it.



enqueue (d, 2)



dequeue()



* Enqueue in priority Queue

enqueue() : Inserts a value to the queue with the specified priority.

- To enqueue, we start at the front of the queue & keep moving back until we find some element of lesser priority or we reach end of queue.

```
void enqueue (int ele, int p)
{
    node *t, *temp;
    t = new node();
    t->data = ele;
    t->priority = p;
    t->next = NULL;
    temp = front;
    if ((front==NULL) || (p > front->priority))
    {
        t->next = front;
        front = t;
    }
    else { while(temp->next!=NULL && temp->priority ≥p)
            temp = temp->next;
        temp->next = t;
    }
}
```

```
Struct node
{       int data, priority;
        node *next;
};
```

* Dequeue in Priority Queue

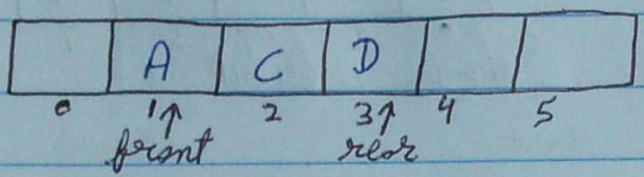
dequeue(): Deletes an element from the queue having highest priority.

- In a priority queue, an element with high priority is deleted before an element with low priority.
- If 2 elements have the same priority, they are deleted according to their order in queue.

```
int dequeue()
{       node *temp;
        temp = front;
        int save = front -> data;
        if ( front == NULL)
                cout << "Empty queue";
        else    front = front -> next;

        delete temp;
        return save;
}
```

2016

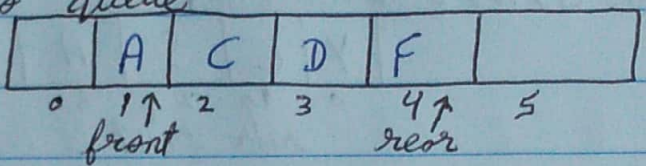
Q = Consider the following Queue of characters of size 6:
This Queue is implemented as a circular array.
Show the contents of Queue with position of Front & Rear after each of the following operations:



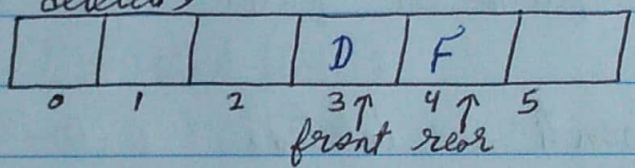
[5 marks]

(i) F is added to queue

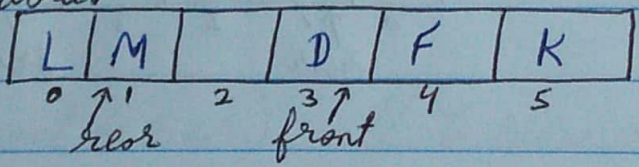
Ans=



(ii) 2 letters are deleted,

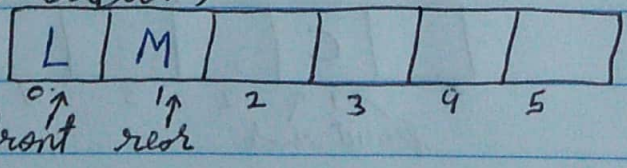


(iii) K, L, M are added

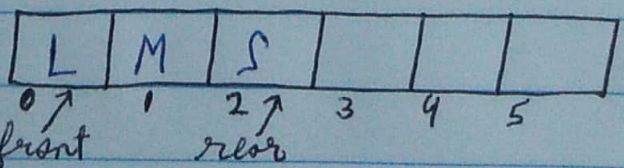


(iv) 3 letters are deleted,

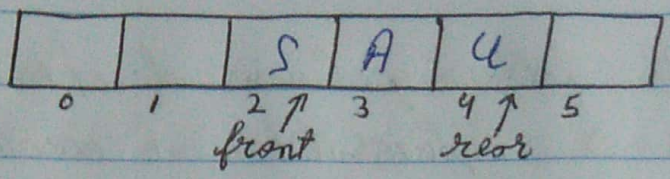
Ans=



(v) S is added,

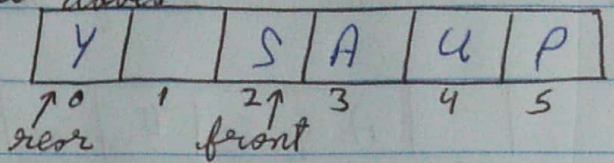


Ex=

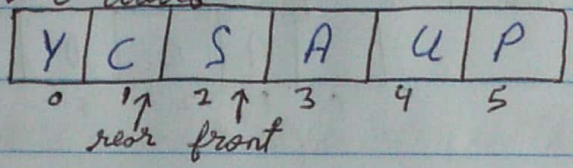


(4 marks)

(i) Elements P, Y are added

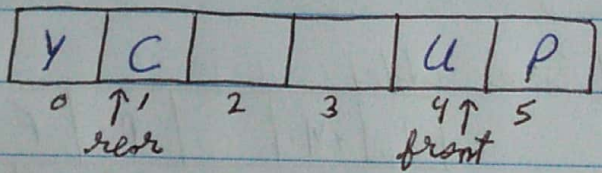


(ii) Elements C, N, B are added

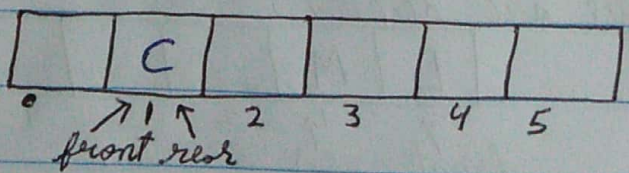


Elements N and B can not be added into the queue as the queue is full.

(iii) 2 elements are deleted,

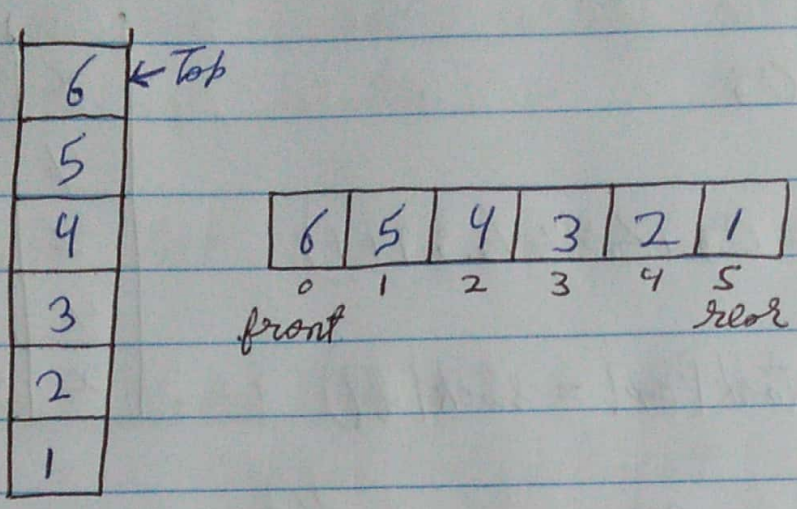


(iv) 3 elements are deleted,



Ex = W.A.P. to reverse the order of elements in a stock using 1 additional ~~S~~ queue.

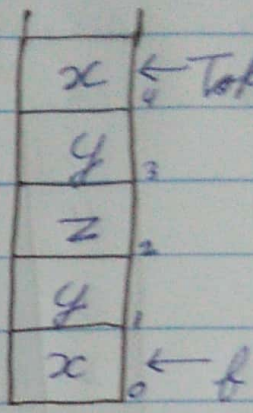
Ans =




```
while (top > -1)          // To store elements of a stock
{     rear = rear + 1;    // into a queue
      q[rear] = stock[top];
      top--;
}
```

```
while ( front ≤ rear)     // To store elements of the
{     top++;              // queue back into the stock
      stock[top] = queue[front];
      front++;
}
```

Ex= Write a function to check whether a string is Polindrome or not using stack.



Ans=

```
f=0;
for(int i=0; i<size/2; i++)
{
    if(stack[top] = stack[f])
    {
        top--;          // Polindrome
        f++;
    }
    else cout<<"Not Polindrome";
}
```

Linked List

- A linked list is a liner collection of data elements, called nodes pointing to the next node by means of pointers.
- A linked list is a dynamic data structure.

★ Singly Linked List

It is a sequence of elements in which every element has link to its next element in the sequence.

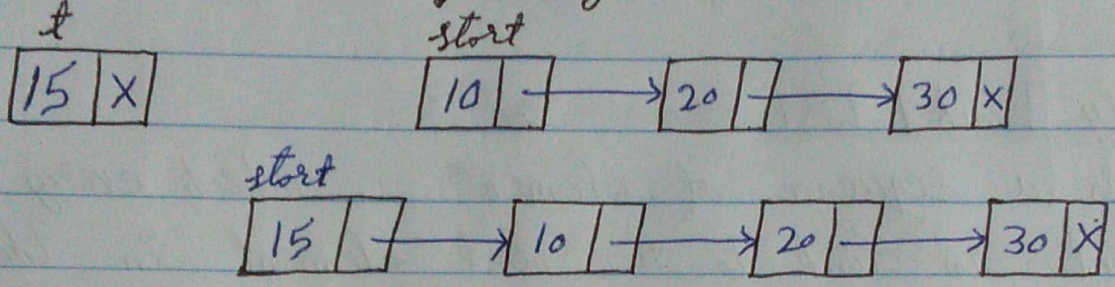
```
struct node
{
        int info;
        struct node *next;
};
```

- In above statement we defines a new data type called node, consisting of 2 fields: one is "info" of type integer and second is "next" which is a pointer of type struct node.

```
class node
{   public: int info;
            node *next;
};
```

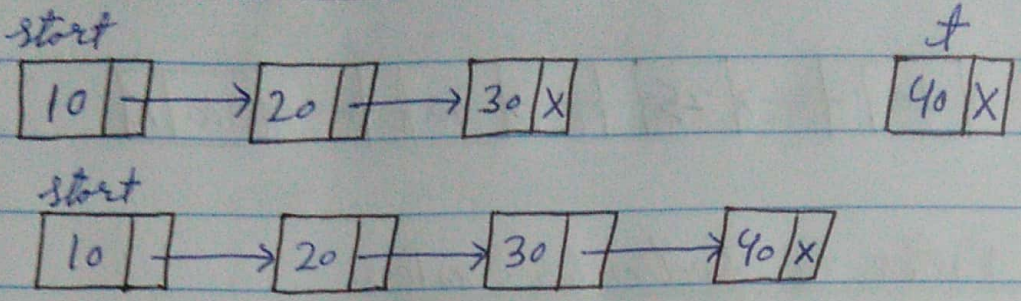
- There is no difference between the way "class" and "struct" keywords are used except for the fact that by default the members of a class are private and that of "struct" is public.

* Insertion at Beginning




```
void InsertAtBeg(int value)
{
    struct node *t;
    t->data = value;
    if (start == NULL)
    {
        start = t;
        t->next = NULL;
    }
    else
    {
        t->next = start;
        start = t;
    }
}
```

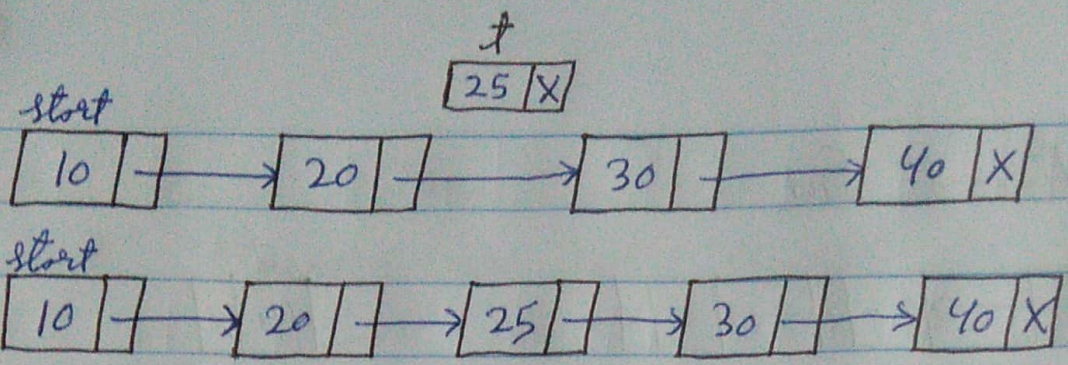
* Insertion at End




```
void InsertAtEnd(int value)
{
        struct node *t, *temp;
        t -> data = value;
        t -> next = NULL;
        temp = start;
        while (temp -> next != NULL)
                temp = temp -> next;
        temp -> next = t;
}
```

* Insertion between 2 nodes

```
void InsertBetween(int value, int loc1, int loc2)
{
        struct node *t, *temp;
        t -> data = value;
        temp = start;
        while(temp -> data != loc1 && temp -> data != loc2)
                temp = temp -> next;
        t -> next = temp -> next;
        temp -> next = t;
}
```



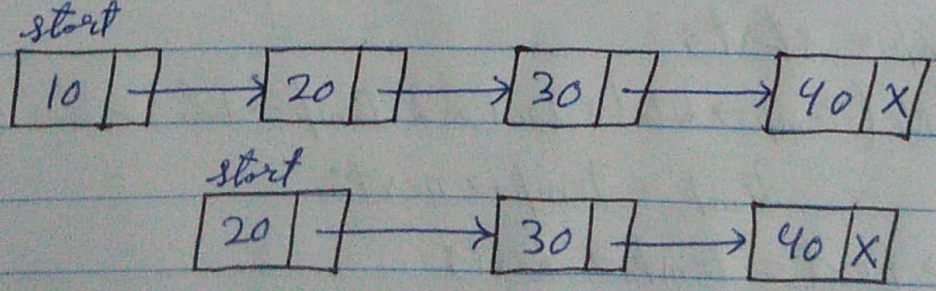
* Insertion after a particular node

```
t1 = start;
while (t1 -> data != 20)          // After 20
        t1 = t1 -> next;
t -> next = t1 -> next;
t1 -> next = t;
```

* Insertion before a particular node

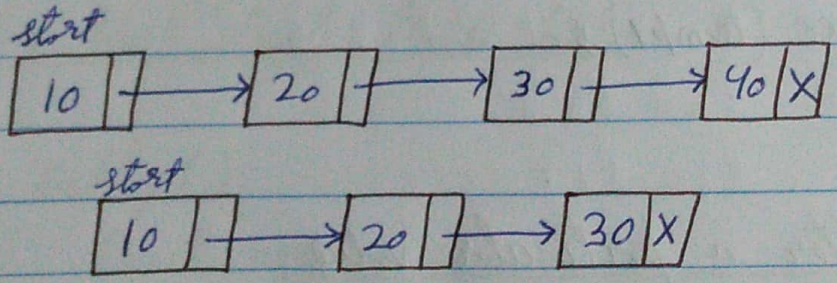
```
t1 = t2 = start;
while (t1 -> data != 30)          // before 30
{       t2 = t1;
        t1 = t1 -> next;
}
t2 -> next = t;
t -> next = t1;
```

* Deletion at Beginning

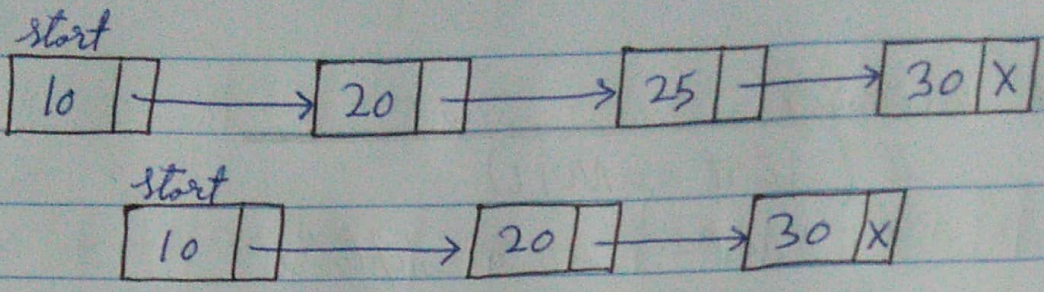


```
void DeleteBeg()
{       if (start == NULL)
            cout << "List is empty";
        else
        {   struct node *temp = start;
            if (start -> next == NULL)
                    start = NULL;
            else  start = temp -> next;
            free (temp);
        }
}
```

* Deletion at End




```
void DeleteEnd()
{       struct node *temp1 = start, *temp2;
        while (temp1 -> next != NULL)
        {   temp2 = temp1;
            temp1 = temp1 -> next;
        }
        temp2 -> next = NULL;
        free (temp1);
}
```



* Deletion of a particular value

```
void DeleteBetween (int delValue)
{
    struct node *temp1 = start, *temp2;
    while (temp1->data != delValue)
    {
        temp2 = temp1;
        temp1 = temp1->next;
    }
    temp2->next = temp1->next;
    free (temp1);
}
```

* Deletion after a particular value

```
void DeleteAfter (int value)
{
    struct node *t, *t1, *t2 = start;
    while (t->data != value)
    {
        t1 = t;  t = t->next;
        t2 = t->next;
    }
    t1->next = t2;
    free (t);
}
```

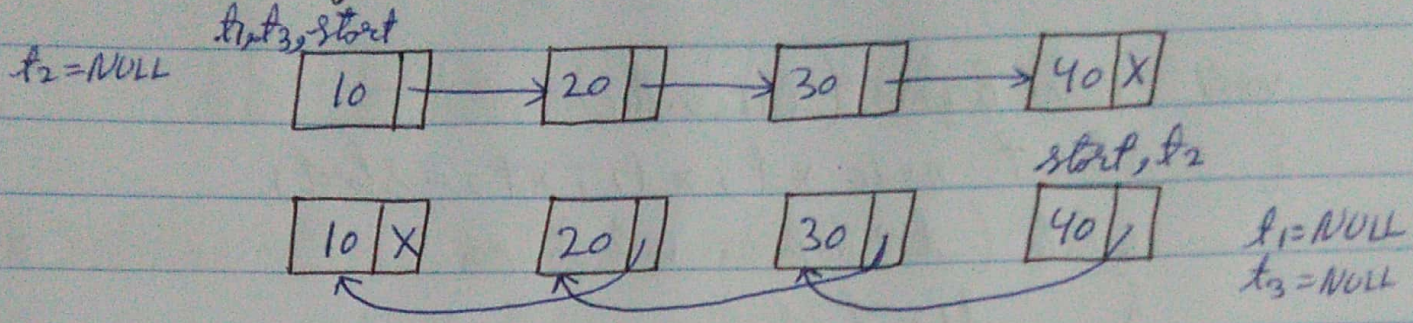
* Deletion before a particular value

```
void DeleteBefore (int value)
{   struct node *t, *t1, *t2 = start;
    while ( t→data != value)
    {   t1 = t2;
        t2 = t1→next;
        t = t2→next;
    }
    t1→next = t;
    free (t2);
}
```

* Display the Singly linked list

```
void display()
{   struct node *temp = start;
    cout << "The elements are :";
    while ( temp→next != NULL)
    {   cout << temp→data;
        temp = temp→next;
    }
    cout << temp→data;
}
```

Ex = Reversing a Singly Linked List

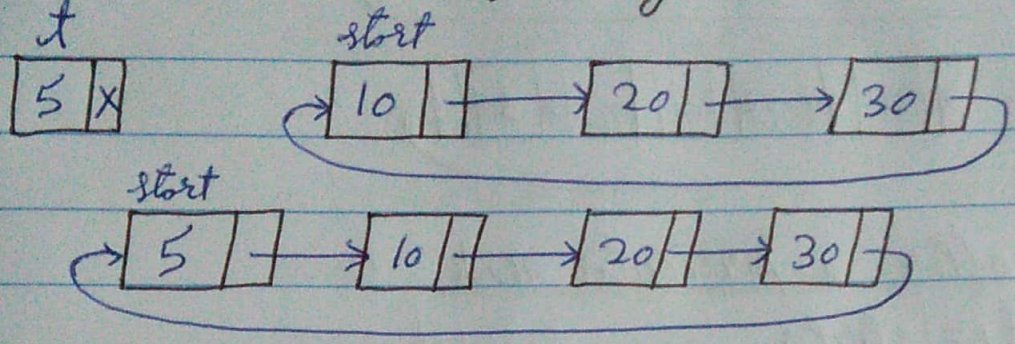



```
void Reverse()
{
    struct node *t1, *t2, *t3;
    t1 = start;
    t2 = NULL;
    t3 = start;
    while (t1 != NULL)
    {
        t1 = t1 -> next;
        t3 -> next = t2;
        t2 = t3;
        t3 = t1;
    }
    start = t2;
}
```

☆ Circular Linked List

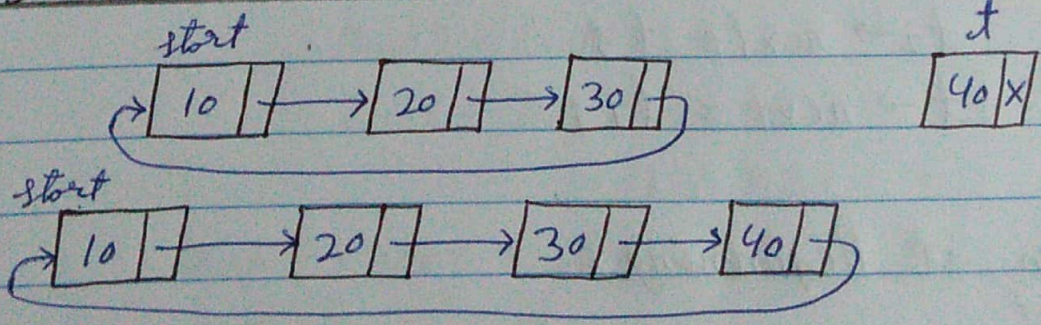
Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and last element has a link to the first element in sequence.

* Insertion at Beginning

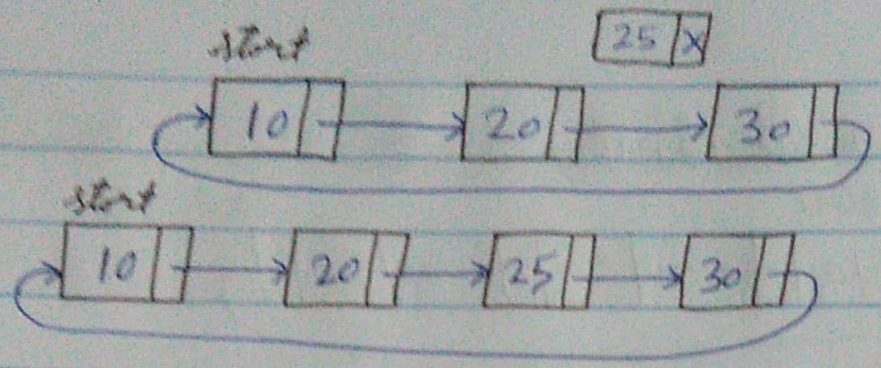



```
void InsertAtBeg(int value)
{
    struct node *t, *temp;
    t->data = value;
    temp = start;
    while( temp->next != start)
        temp = temp->next;
    t->next = start;
    start = t;
    temp->next = start;
}
```

* Insertion at End




```
while ( temp->next != start)
    temp = temp->next;
temp->next = t;
t->next = start;
```



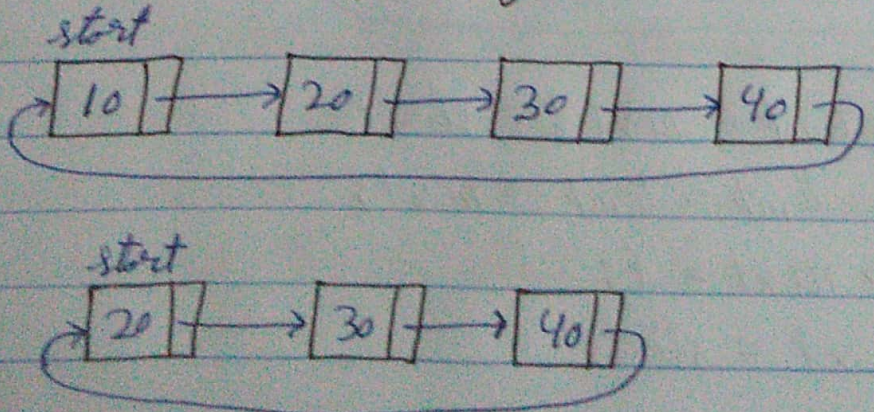
* Insertion after a particular node

```
t1 = start;
while (t1→data != value)
        t1 = t1→next;
t→next = t1→next;
t1→next = t;
```

* Insertion before a particular value

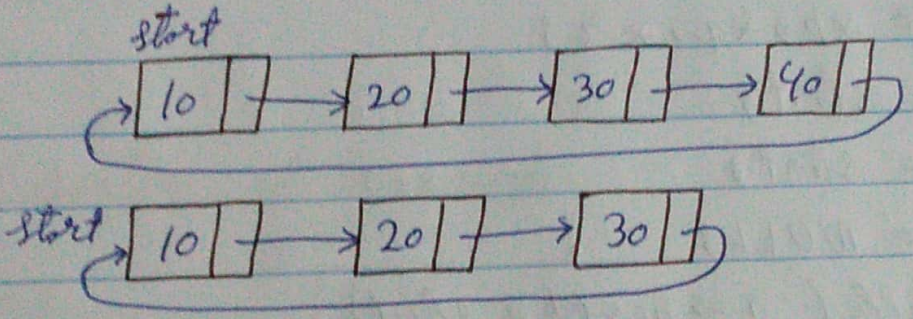
```
t1 = t2 = start;
while (t1→data != value)
{   t2 = t1;
    t1 = t1→next;
}
t2→next = t;
t→next = t1;
```

* Deletion at Beginning



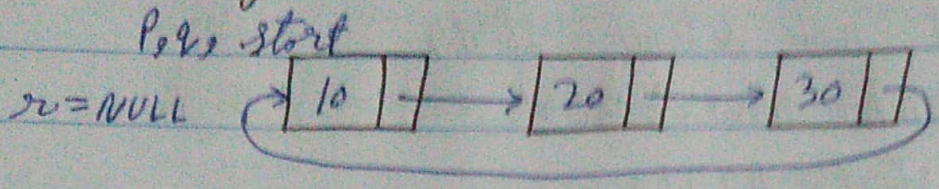
```
void DeleteAtBeg()
{
    struct node *temp = start;
    if (temp->next == start)
        start = NULL;
    else start = start->next;
    free(temp);
}
```

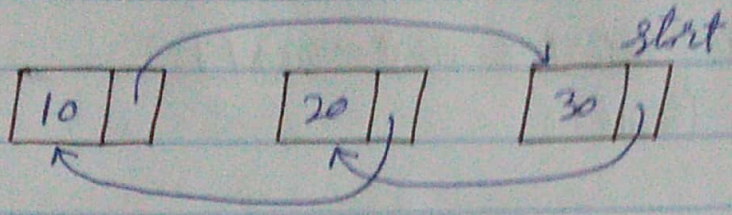
* Deletion at End




```
void DeleteAtEnd()
{
    struct node *temp1 = start, *temp2;
    while(temp1->next != start)
    {
        temp2 = temp1;
        temp1 = temp1->next;
    }
    temp2->next = start;
    free(temp1);
}
```

★ Reversing a Circular linked list





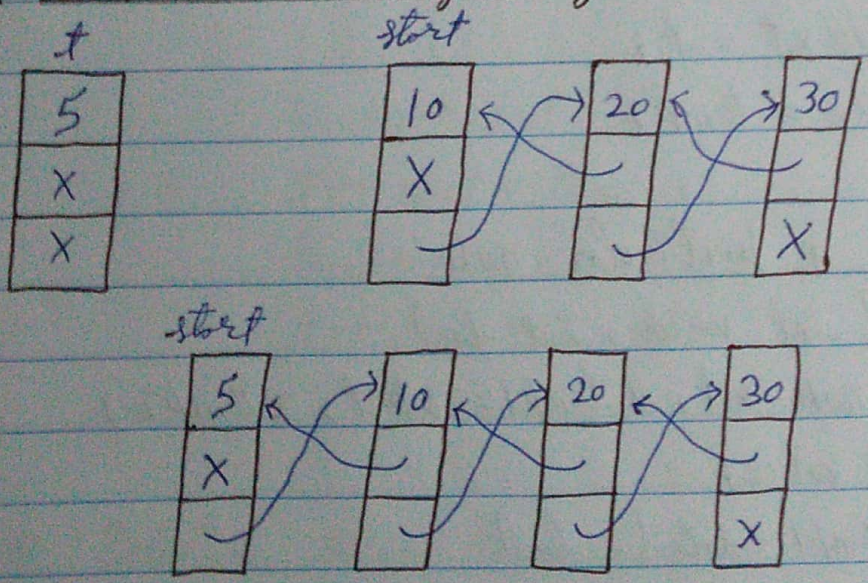

```
void ReverseCLL()
{
    node *p, *q, *r;
    p = start;
    q = start;
    r = NULL;
    while ( q -> next != start)
    {
        p = p -> next;
        q -> next = r;
        r = q;
        q = p;
    }
    q -> next = r;
    start -> next = p;
}
```

☆ Doubly Linked List

Doubly Linked List is a sequence of elements in which every element has links to its previous element and next element in the sequence.

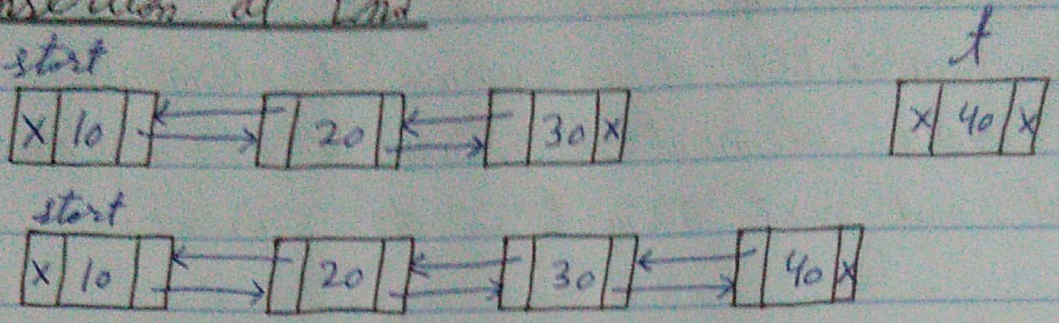
```
Struct node
{   int data;
    struct node *prev;
    struct node *next;
}
```

* Insertion at Beginning




```
void InsertAtBeg (int value)
{   struct node *t;
    t → data = value;
    t → prev = NULL;
    t → next = start;
    start = t;
}
```

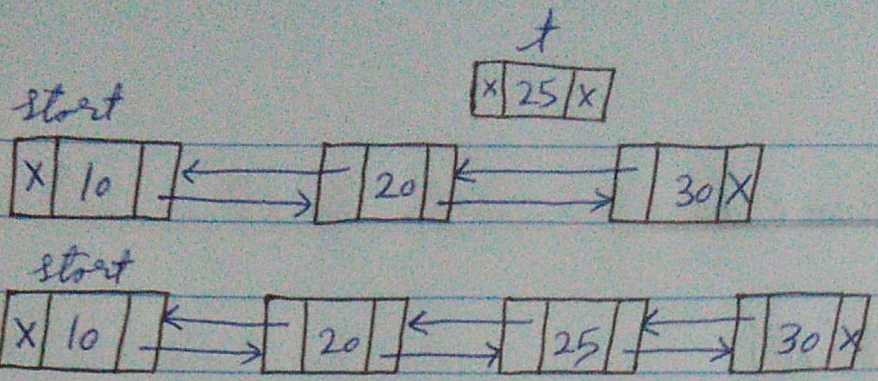
* Insertion at End




```
void InsertAtEnd (int value)
{
    struct node *t, *temp = start;
    t→data = value;
    t→next = NULL;
    while (temp→next != NULL)
        temp = temp→next;
    temp→next = t;
    t→prev = temp;
```

* Insertion after a particular node

```
void InsertAfter (int value, int loc)
{
    struct node *t, *temp1 = start, *temp2;
    t→data = value;
    while (temp1→data != loc)
        temp1 = temp1→next;
    temp2 = temp1→next;
    temp1→next = t;
    t→next = temp2;
    t→prev = temp1;
    temp2→prev = t;
}
```



* Insertion before a particular node

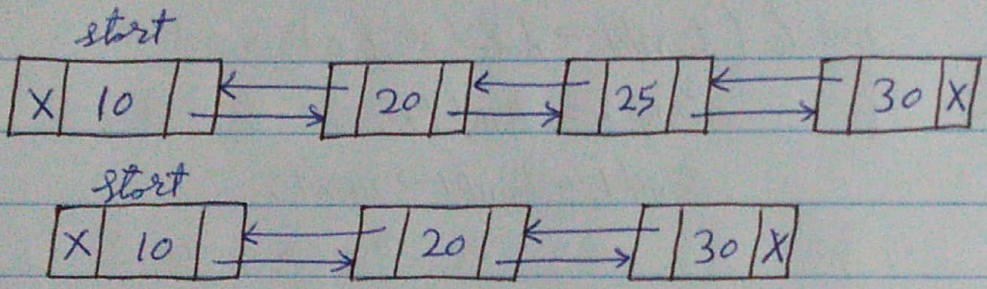
```
void InsertBefore (int value, int loc)
{       struct node *t, *temp1 = start, *temp2;
        t -> data = value;
        while (temp1 -> data != loc)
        {       temp2 = temp1;
                temp1 = temp1 -> next;
        }
        temp2 -> next = t;
        t -> next = temp1;
        temp1 -> prev = t;
        t -> prev = temp2;
}
```

* Deletion from Beginning

```
void DeleteBeg ()
{       struct node *temp = start;
        start = temp -> next;
        start -> prev = NULL;
        free (temp);
}
```

* Deletion from End

```
void DeleteEnd()
{
    struct node *temp = start;
    while (temp->next != NULL)
        temp = temp->next;
    temp->prev->next = NULL;
    free (temp);
}
```



* Deletion of a particular value

```
void DeleteSpecific (int delValue)
{
    struct node *temp = start;
    while (temp->data != delValue)
        temp = temp->next;

    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
    free (temp);
}
```

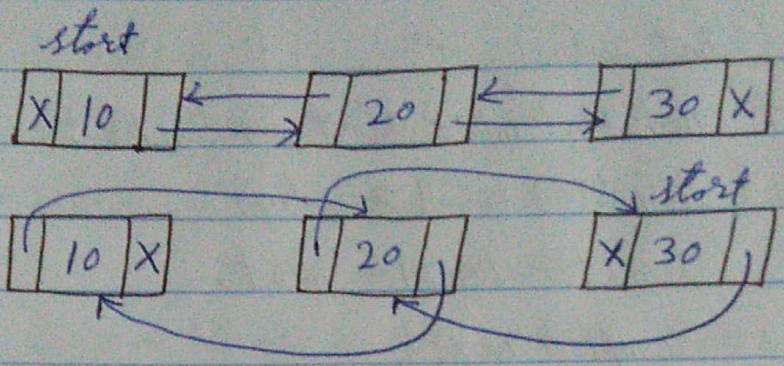
* Deletion after a particular value

```
void DelAfter(int value)
{
    struct node *temp1 = start, *temp2;
    while(temp1->data != value)
        temp1 = temp1->next;
    temp2 = temp1->next;
    temp1->next = temp2->next;
    temp2->next->prev = temp1;
    free(temp2);
}
```

* Deletion before a particular value

```
void DelBefore(int value)
{
    struct node *temp1 = start, *temp2;
    while(temp1->data != value)
    {
        temp2 = temp1;
        temp1 = temp1->next;
    }
    temp2->prev->next = temp1;
    temp1->prev = temp2->prev;
    free(temp2);
}
```

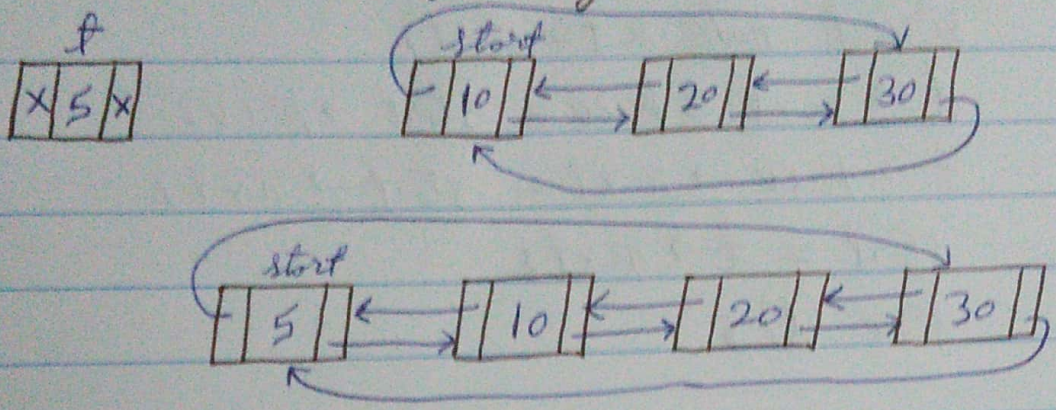
☆ Reversing a Doubly Linked List




```
void ReverseDLL()
{
    node *q, *s, *temp;
    q = s = start;

    while (q != NULL)
    {
        q = q -> next;
        temp = s -> next;
        s -> next = s -> prev;
        s -> prev = temp;
        if (q != NULL)
            s = q;
    }

    start = s;
}
```

A Circular Doubly Linked List

* Insertion at beginning




```
void InsertAtBeg(int value)
{
    struct node *t = start;
    t -> info = value;
    t -> prev = start -> prev;
    t -> next = start;
    start -> prev -> next = t;
    start -> prev = t;
    start = t;
}
```

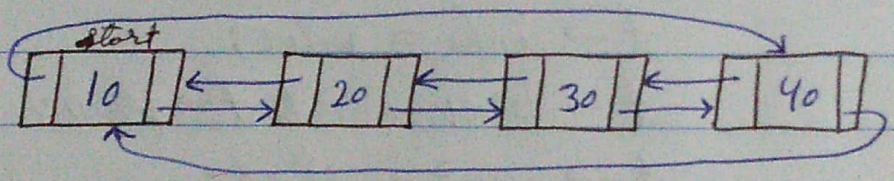
* Insertion at End

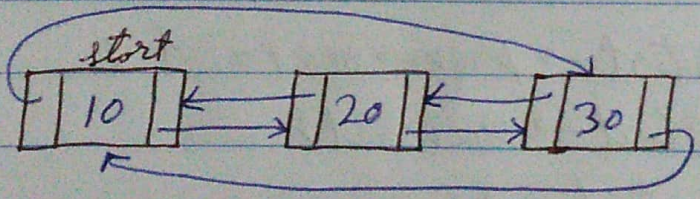
```
void InsertAtEnd(int value)
{
    struct node *t = start;
    t -> info = value;
    t -> prev = start -> prev;
    t -> next = start;
    start -> prev -> next = t;
    start -> prev = t;
}
```

* Deletion from Beginning

```
void DeleteBeg ()
{
        struct node *t = start;
        t -> next -> prev = start -> prev;
        t -> prev -> next = start -> next;
        start = t -> next;
        free (t);
}
```

* Deletion from End






```
void DeleteEnd ()
{
        struct node *t = start;
        while ( t -> next != start)
                t = t -> next;

        t -> prev -> next = start;
        start -> prev = t -> prev;
        free (t);
}
```

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well