UNIT III NON LINEAR DATA STRUCTURES – TREES

Tree ADT – Tree traversals - Binary Tree ADT – Expression trees – Applications of trees – Binary search tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap.

3.1 INTRODUCTION TO TREE – TREE ADT

Trees are non-liner data structure, which represents any hierarchical relationship between any data Item.

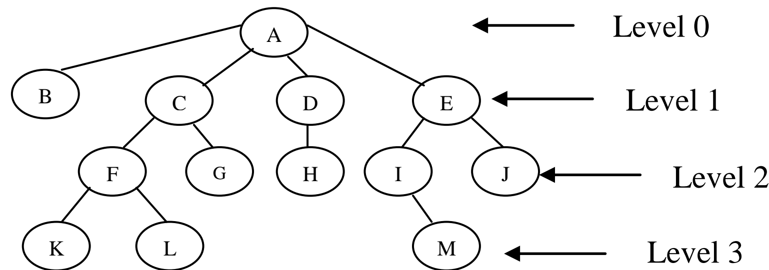


Fig: 3.1 Tree ADT

Root: The node at the top of the tree or the node that has no parent is called the root. There is only one root in the tree, In Fig 4.1, the root is A.

Node: Item of information.

Parent: The node having further sub-branches is called parent node. In Fig 4.1, the C is the parent node of F and G

Leaf: A node which doesn't have children is called leaf or terminal node. Here B, K, L, G, H, M, J are leaf.

Siblings: Children of the same parents are said to be siblings or brothers, here node B, C, D, E are siblings of each other and node F, G are siblings of each other.

Path: A path from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, n_3,…n_k such that n_i is the parent of n_{i+1}, for $I \leq i< k$. There is exactly only one path from each node to root. In fig:3.1 path from A to L is A,C,F,L. Where A is the parent for C, C is the parent of F and F is the parent of L.

Length: The length is defined as the number of edges on the path. In Fig 4.1 the length for the path A to L is 3.

Degree of node: The total number of sub trees attached to that node is called the degree of a node. In Fig 4.1 degree of A is 4, degree of C is 2, degree of D is 1, and degree of H is 0.

Degree of tree: The degree of the tree is the maximum degree of any node in the tree. In Fig 4.1 the degree of the tree is 4.

Level: The root node is always considered at level zero. The adjacent node to rot are supposed to be at level1 and so on.

Level of A is 0; Level of B, C, D is 1; Level of F, G, H, I, J is 2

Level of K, L, M, is 3.

Depth: The maximum level is the depth of the tree. In Fig 4.1 the depth of tree is 3. Sometimes the depth of the tree is also called as **height of the tree.**

3.2 TREE TRAVERSALS (INORDER, PREORDER AND POSTORDER)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

Following are the generally used ways for traversing trees.

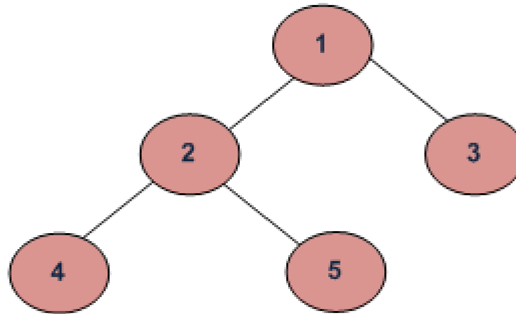


Fig: 3.2 Tree traversals

Example Tree

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Inorder Traversal:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder itraversal s reversed, can be used.

Example: Inorder traversal for the above given figure is 4 2 5 1 3

Preorder Traversal:

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal:

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder

Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree.

3.3 BINARY TREE ADT

A binary tree is a tree in which no node can have more than two children. The maximum degree of any node is two. This means the degree of a binary tree is either zero or one or two.

Types of Binary Tree

i. Strictly binary tree

Strictly binary tree is a binary tree where all the nodes will have either zero or two children. It does not have one child in any node.

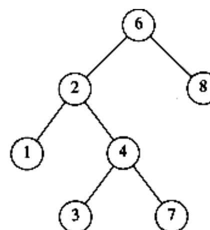


Fig: 3.3 Binary Tree

ii. Skew tree

A skew tree is a binary tree in which every node except the leaf has only one child node. There are two types of skew tree, they are left skewed binary tree and right skewed binary tree.

- **Left skewed binary tree:** A left skew tree has node with only the left child. It is a binary tree with only left sub-trees.
- **Right skewed binary tree :** A right skew tree has node with only the right child. It is a binary tree with only right sub-trees.

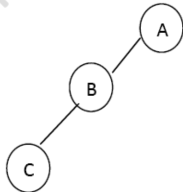


Fig: 3.4 Left skew Binary tree

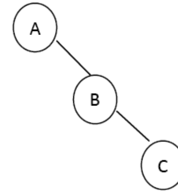


Fig: 3.5 Right skew binary tree

iii. Full binary tree or proper binary tree

A binary tree is a full binary tree if all leaves are at the same level and every non leaf node has exactly two children and it should contain maximum possible number of nodes in all levels. A full binary tree of height h has **2h+1 – 1** node.

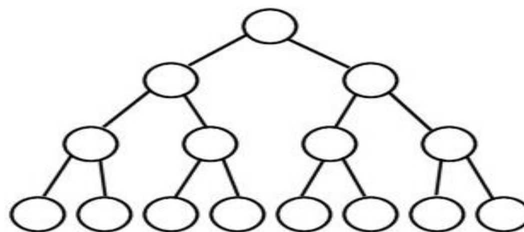


Fig: 3.6 Proper Binary Trees

iv. Complete binary tree

Every non leaf node has exactly two children but all leaves are not necessary at the same level. A complete binary tree is one where all levels have the maximum number of nodes except the last level. The last level elements should be filled from left to right.

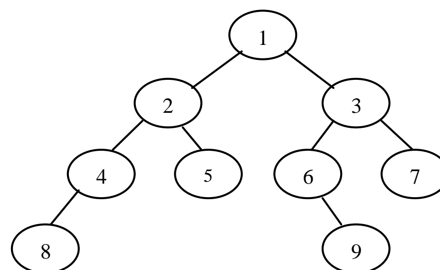


Fig: 3.7 Complete Binary Tree

Representation of binary Trees:

In binary tree each node will have left child, right child and data field.

Left Child	Data	Right Field

Fig: 3.8 Representation of binary Trees

The left child is nothing but the left link which points to some address of left sub-tree whereas right child is also a right link which points to some address of right sub-tree. And the data field gives the information about the node. Let us see the 'C' structure of the node in a binary tree.

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

3.4 EXPRESSION TREE

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:

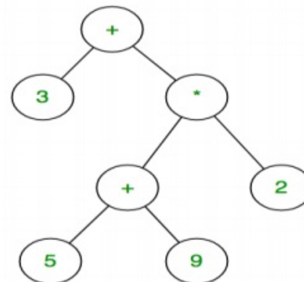


Fig: 3.9 Expression Tree

Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

Let t be the expression tree

```
If  t is not null then
    If t.value is operand then
        Return  t.value
    A = solve(t.left)
    B = solve(t.right)
  // calculate applies operator 't.value'
```

// on A and B, and returns value

Return calculate(A, B, t.value)

Construction of Expression Tree:
Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.
1) If character is operand push that into stack
2) If character is operator pop two values from stack make them its child and push current node again.
At the end only element of stack will be root of expression tree.

3.5 APPLICATION OF TREES

i. Manipulation of arithmetic expression
ii. Symbol table construction
iii. Syntax Analysis
iv. Grammar
v. Expression Tree

3.6 BINARY SEARCH TREE ADT

A **Binary Search Tree** is a binary tree (has atmost 2 children) with the following properties:

- All items in the left sub-tree are less than the root.
- All items in the right sub-tree are greater or equal to the root.
- Each sub-tree is itself a binary search tree.

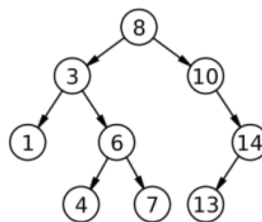


Fig: 3.10 Binary Search Tree

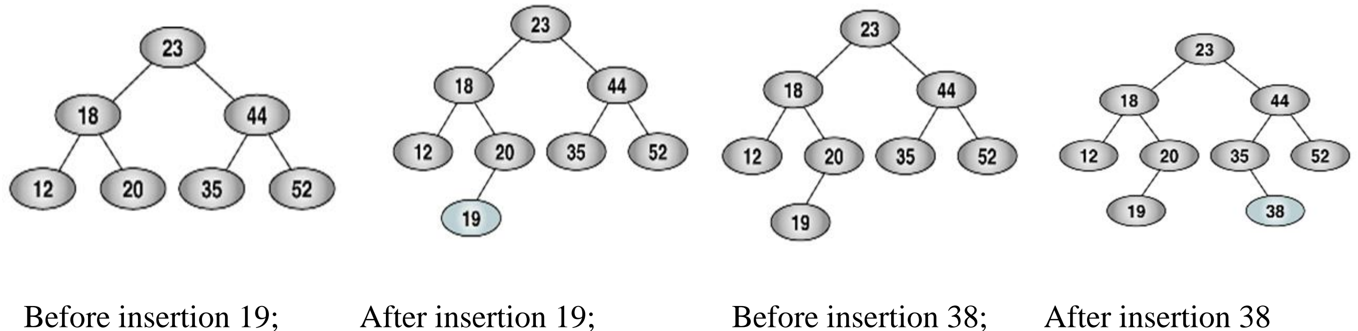
Binary Tree operation:

- Insertion of a node in a binary tree.
- Deletion of some element from the binary search tree.
- Searching of an element in the binary tree

Insertion in BST

- Read the value for the node which is to be created, and store it in a node called **New.**
- Initially if (root!=**Null**) then root=New
- Again read the next value of node created in New.

- If (New->value < root ->value) then attach New node as a left child of root otherwise attach New node as a right child of root.
- Repeat step 3 and 4 for constructing required binary search tree completely.
- Time complexity, Average: O(log n), Worst O(n)



Before insertion 19; After insertion 19; Before insertion 38; After insertion 38

Fig: 3.11 Binary Tree Operations-Insertions

Deletion in BST:

There are three possible cases to consider:

- Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree.
- Deleting a node with one child: Remove the node and replace it with its child.
- Deleting a node with two children: It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced trees.
- Rather than simply delete the node, we try to maintain the existing structure as much as possible by finding data to take the place of the deleted data. This can be done in one of two ways.
- We can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.
- We can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.
- Either of these moves preserves the integrity of the binary search tree.

Deletion of node has two children:

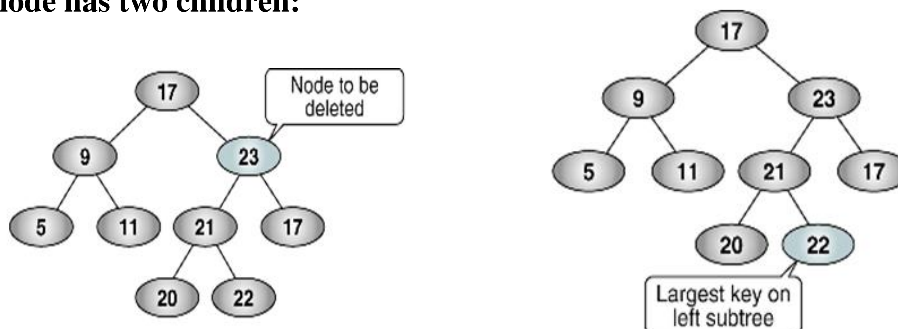


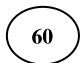
Fig: 3.11 Binary Tree Operations-Deletion

Searching in a BST

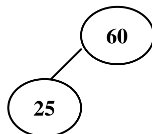
- Examine the root node. If tree is NULL value doesn't exist.
- If value equals the key in root search is successful and return.
- If value is less than root, search the left sub-tree.
- If value is greater than root, search the right sub-tree.
- Continue until the value is found or the sub tree is NULL.
- Time complexity. Average: O(log n), Worst: O(n) if the BST is unbalanced and resembles a linked list.

Draw a binary search tree for the following input list 60, 25, 75, 15, 50, 66, 33, 44. Trace the algorithm to delete the nodes 25, 75, 44 from the tree.

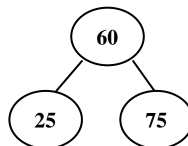
Step: 1



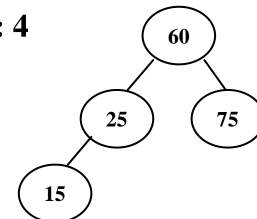
Step: 2



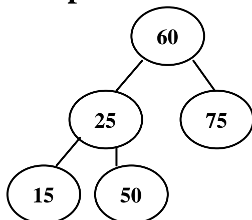
Step: 3



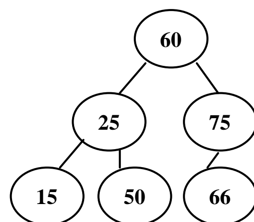
Step: 4



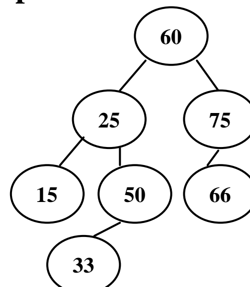
Step 5:



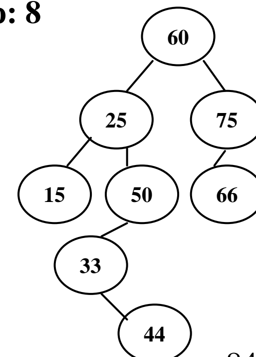
Step: 6



Step: 7

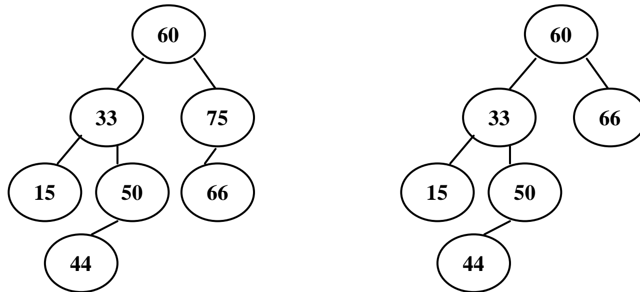


Step: 8



Now we will delete 25 from above tree. To delete 25, first find in-order successor of 25 and copy it at the place of 25.

To delete node 75, copy node 66 at the place of 75.



Delete node 44, i.e. simple set left pointer of node 50 to Null.

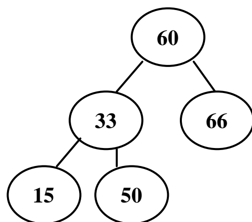


Fig: 3.12 Binary Search Tree Operations

3.7 THREADED BINARY TREE

- The idea of threaded binary trees is to make Inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the Inorder successor of the node (if it exists).
- There are two types of threaded binary trees.
  Single Threaded: Where a NULL right pointers is made to point to the Inorder successor (if successor exists)
- **Double Threaded:** Where both left and right NULL pointers are made to point to Inorder predecessor and Inorder successor respectively.
- The predecessor threads are useful for reverse Inorder traversal and Postorder traversal.
- The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree.

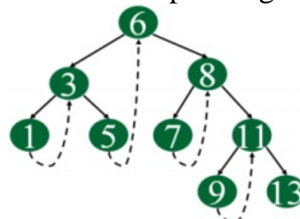


Fig: 3.13 Threaded Binary Trees

C REPRESENTATION OF A THREADED NODE

Following is C representation of a single threaded node.

```
struct Node
{
   int data;
   Node *left, *right;
   bool rightThread;
}
```

The following diagram demonstrates inorder order traversal using threads.

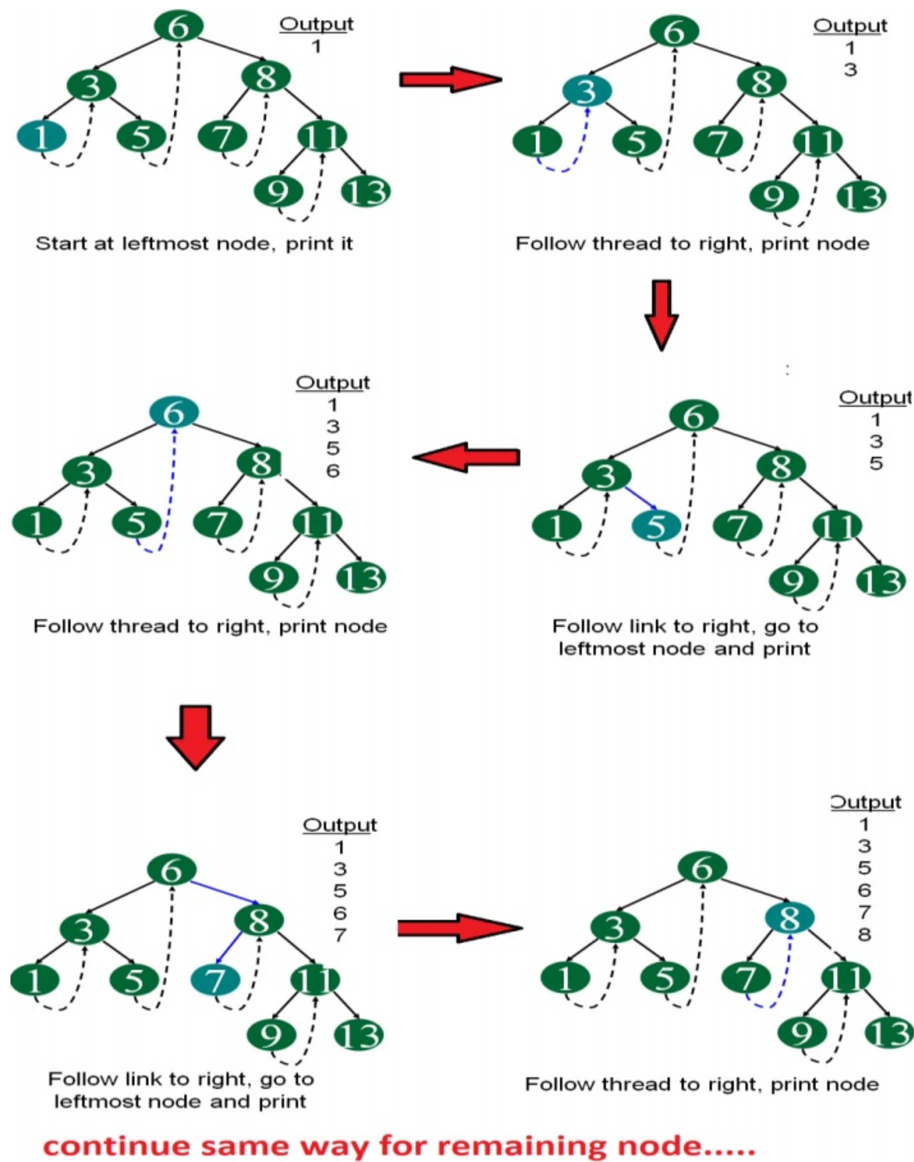


Fig: 3.14 Threaded Binary Tree-Inorder Traversal

3.8 AVL TREE

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

Balance Factor of AVL Tree

- Adelsion Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to height of sub-trees.
- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - › height(left sub-tree) - height(right sub-tree)
- An AVL tree has balance factor calculated at every node
  - › For every node, heights of left and right sub-tree can differ by no more than 1
  - › Store current heights in each node

Representation of AVL Tree

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

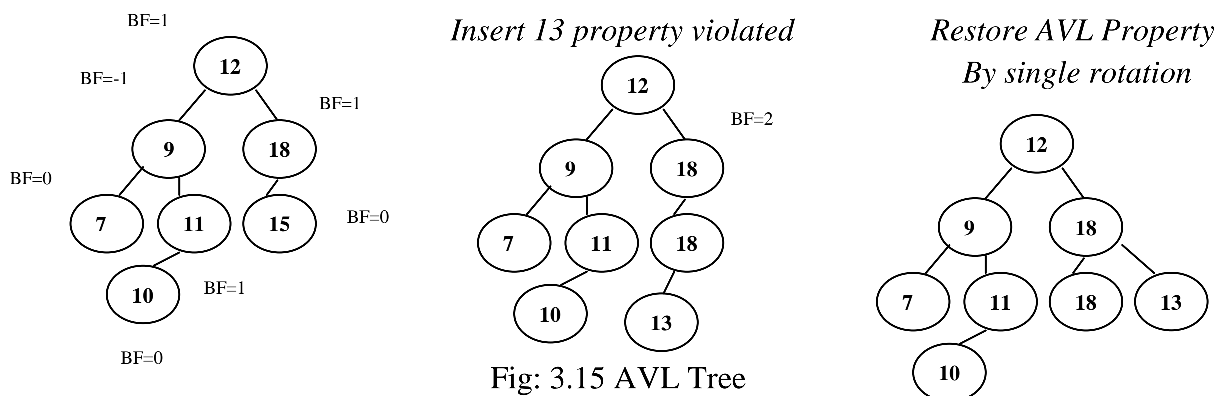


Fig: 3.15 AVL Tree

Insertion

- There are four different cases when rebalancing is required after insertion of new node.
  1. An insertion of new node into left sub-tree of left child (LL).
  2. An insertion of new node into right sub-tree of left child (LR).
  3. An insertion of new node into left sub-tree of right child (RL).
  4. An insertion of new node into right sub-tree of right child (RR).
- There are two types of rotations.
  - Single rotation (LL rotation) or (RR rotation)
  - Double rotation (LR rotation) or (RL rotation)

Insertion algorithm

1. Insert a new node as new leaf just as in ordinary binary search tree.
2. Now trace the path from insertion point (new node inserted as leaf) towards root. For each node 'n' encountered, check if heights of left (n) and right (n) differ by at most 1.
   a) If yes, move towards parent (n)
   b) Otherwise restructure by doing either a single rotation or a double rotation.
   Thus once we perform a rotation at node 'n' we do not require to perform any rotation at any ancestor on 'n'.

Different rotations in AVL tree

LL rotation:

When node '1' gets inserted as a left child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2.

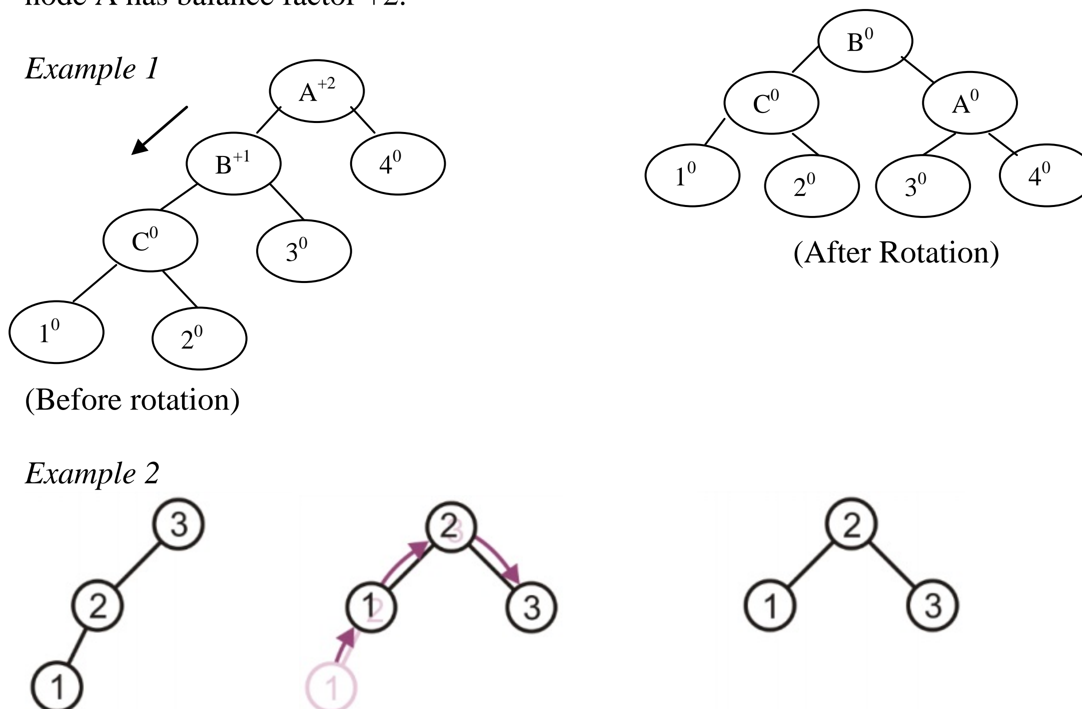


Fig: 3.16 AVL Tree [LL rotation]

RR rotation:

When node '4' gets inserted as a right child of node 'C' then node 'A' gets unbalanced. The rotation which needs to be applied is RR rotation as shown below

Example 1:

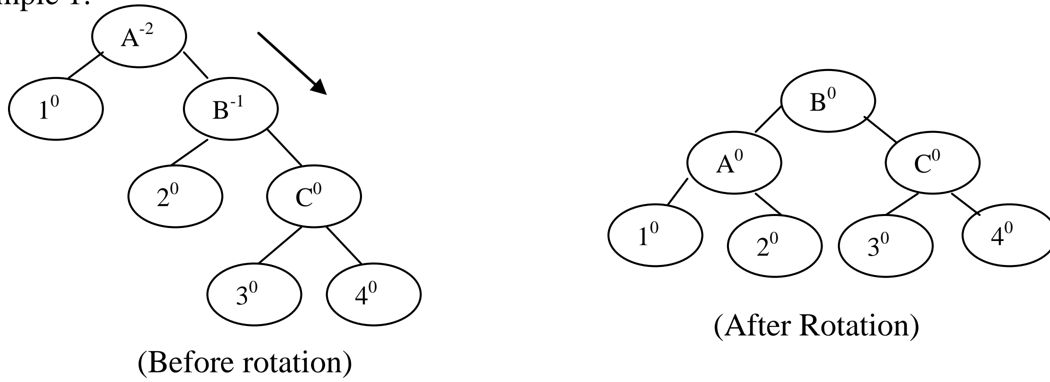
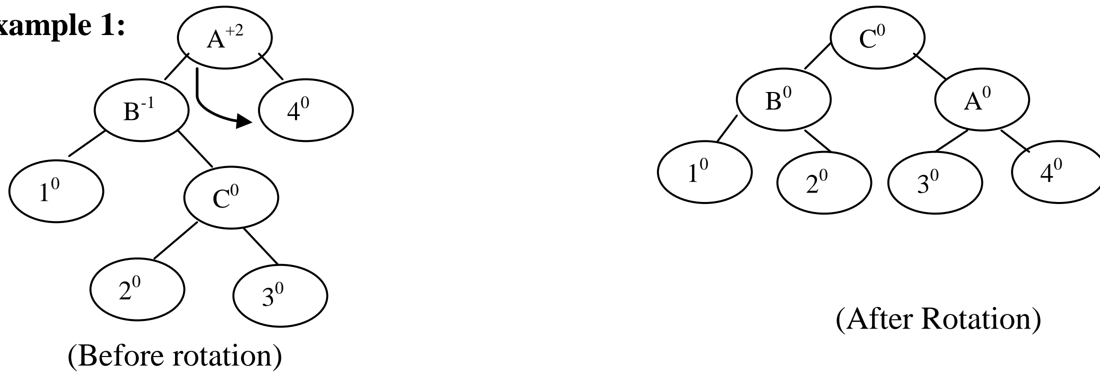


Fig: 3.17 AVL Tree [RR rotation]

LR rotation:

When node '3' is attached as a right child of node 'C' then unbalancing occurs because of LR. Hence LR rotation needs to be applied.

Example 1:



Example 2:

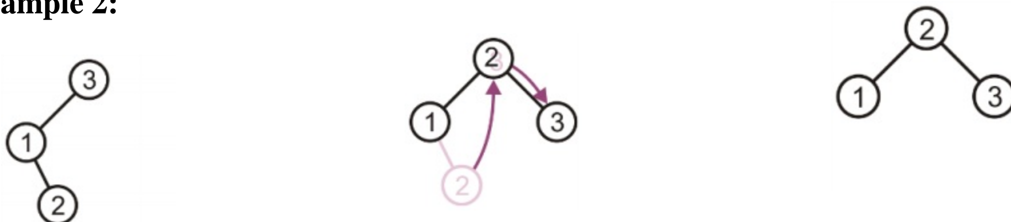


Fig: 3.18 AVL Tree [LR rotation]

RL rotation:

When node ‘2’ is attached as a left child of node ‘C’ then node ‘A’ unbalanced as its balance factor become -2. Then RL rotation needs to be applied to rebalance the AVL tree.

Fig: 3.19 AVL Tree [RL rotation]

Deletion

Even after deletion of any particular node from AVL tree, the tree has to be restricted in order to preserve AVL property. And thereby various rotations need to be applied.

Algorithm for deletion

1. Search the node which is to be deleted.
2. a.) If the node to be deleted is a leaf node then simply make it NULL to remove.
   b.) If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove this node.
3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some sub-tree then balance that sub-tree using appropriate single or double rotations.

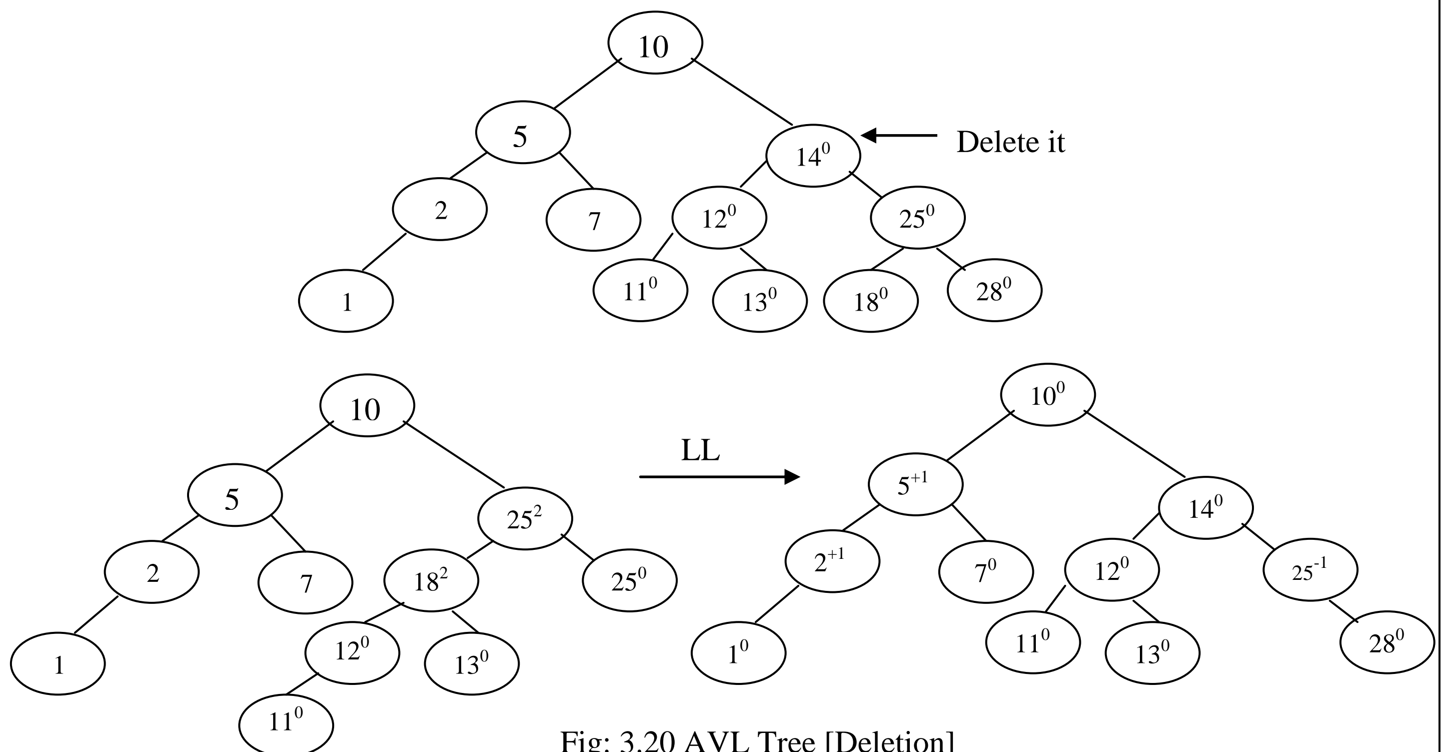


Fig: 3.20 AVL Tree [Deletion]

Construct an AVL tree with the value 3, 1, 4, 5, 9, 2, 8, 7, 0 into an initially empty tree., inserting into an AVL tree.

insert 3

insert 1

insert 4

insert 5

insert 9

RR

insert 2

insert 8

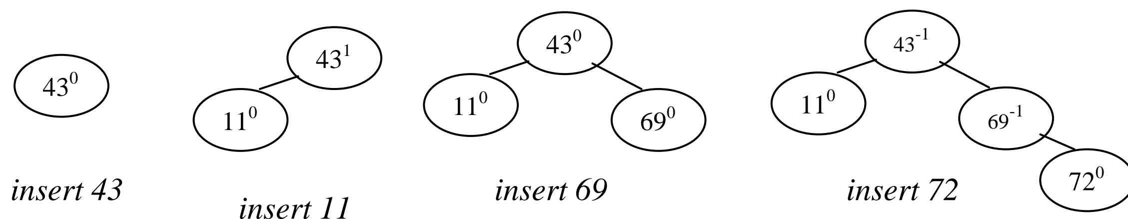
insert 7

LL

insert 0

Fig: 3.21 AVL Tree [Insertion]

Show the result of inserting 43, 11, 69, 72 and 30 into an initially empty AVL tree. Show the result of deleting the nodes 11 and 72 one after the other of the constructed tree.



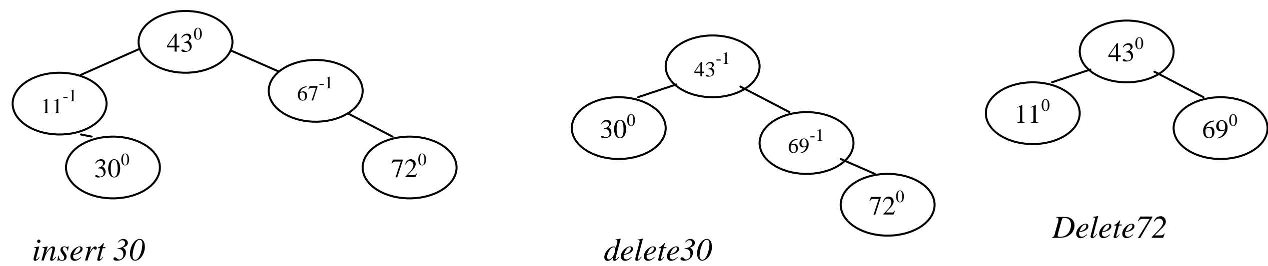


Fig: 3.22 AVL Tree [Deletion]

ROUTINES FOR AVL TREE OPERATIONS

Node Declaration of AVL tree

```
trypedef struct avl_node *avl-ptr;
struct avl_node
{
element_type element;
avl_ptr left;
avl_ptr right;
int height;
};
Trpedefavl_ptr SEARCH_TREE;
```

Computing the height of AVL tree

```
int height(avl_ptr p)
{
if(p==NULL)
return -1;
else return p->height;
}
```

Routine for Insertion

```
SEARCH_TREE insert(element_type x, SEARCH_TREE T)
{
return insert1(x,T,NULL);
}
insert1(element_type x, SEARCH_TREE T, avl_ptr parent)
{
avl_ptr rotated_tree;
if(T==NULL)
```

```
{ //Create and return a one-node tree
T=(SEARCH_TREE)malloc(sizeof (struct avl_node));
if(T==NULL)
fatal_error("Out of Space");
else
{
T->element=x;
T->height=0;
T->left=T->right=NULL;
}
}
else
{
If(x<T->element)
{
T->left=insert1(x,T->left,T);
if(height(T->left)-height(T->right))==2)
{
if(x< T->left->element)
rorated_tree = s_rotate_left(T);
rorated_tree = d-rotate_left(T);
if(parent->left == T)
parent->left = rotated_tree;
else
parent->right=rotated_tree;
}
else
T->height = max(height(T->left),height(T->right))+1;
}
else
}
return T;
}
```

Routine for Left rotation

```
Acl_ptr s_rotate_left(avl_ptr k2)
{
Avl_ptr k1;
K1 = k2->left;
K2->left=k1->right;
K1->right=k2;
```

```
K2->height=max(height(k2->left),height(k2->right))+1;
K1->height=max(height(k1->left),k2->height)+1;
return k1;
}
```

Routine for Double left rotation

```
avl_ptrd_rotate_left(avl_ptr k3)
{
K2->left=s_rotate_right(k3->left);
return(s_rotate_left(k3));
}
```

3.9 B TREE

A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:

- the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
- all leaves are on the same level
- all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
- the root is either a leaf node, or it has from two to *m* children
- a leaf node contains no more than $m - 1$ keys

Inserting into a B-Tree

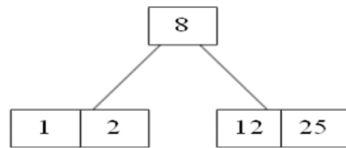
- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Constructing a B-tree

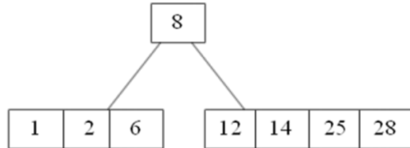
- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- The first four items go into the root:

1	2	8	12

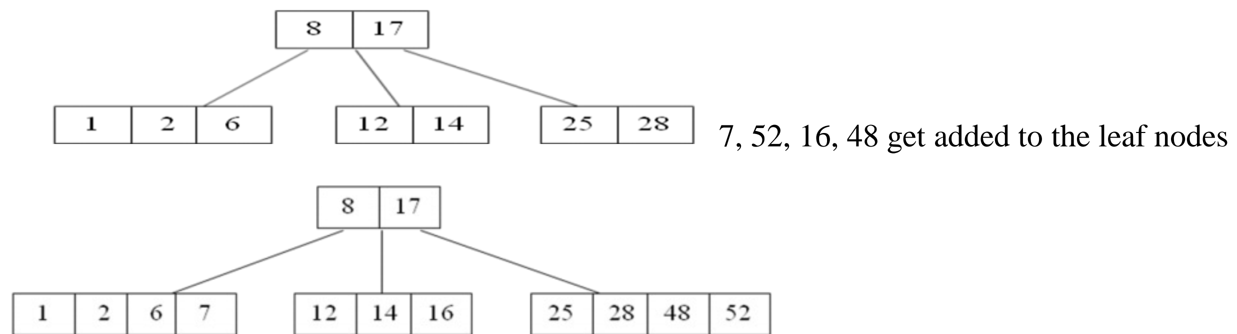
- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root



6, 14, 28 get added to the leaf nodes:

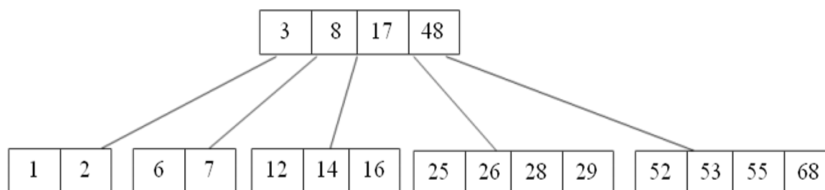


Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



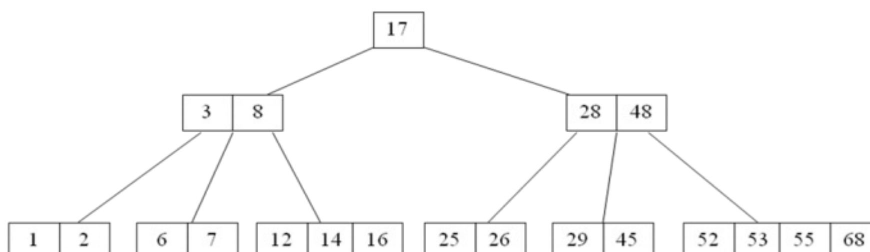
7, 52, 16, 48 get added to the leaf nodes

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves



Adding 45 causes a split of | 25 | 26 | 28 | 29 |

and promoting 28 to the root then causes the root to split



Removal from a B-tree

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.
- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
- 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
- 4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

3.10 B+-trees

B+-tree

A node of a binary search tree uses a small fraction of that, so it makes sense to look for a structure that fits more neatly into a disk block.

Hence the B+-tree, in which each node stores up to d references to children and up to $d - 1$ keys.

Each reference is considered "between" two of the node's keys; it references the root of a subtree for which all values are between these two keys.Here is a fairly small tree using 4 as our value for d.

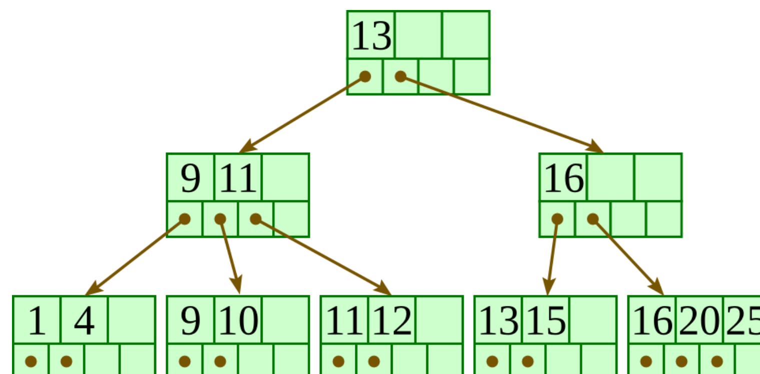


Fig: 3.23 B+ Tree

A B+-tree requires that each leaf be the same distance from the root, as in this picture, where searching for any of the 11 values (all listed on the bottom level) will involve loading three nodes from the disk (the root block, a second-level block, and a leaf).

In practice, d will be larger — as large, in fact, as it takes to fill a disk block. Suppose a block is 4KB, our keys are 4-byte integers, and each reference is a 6-byte file offset. Then we'd choose d to be the largest value so that $4\,(d - 1) + 6\,d \leq 4096$; solving this inequality for d, we end up with $d \leq 410$, so we'd use 410 for d. As you can see, d can be large.

A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node N with k being the number of keys in N: all keys in the first child's subtree are less than N's first key; and all keys in the ith child's subtree ($2 \leq i \leq k$) are between the $(i - 1)$th key of n and the ith key of n.
- The root has at least two children.
- Every non-leaf, non-root node has at least *floor*(d / 2) children.
- Each leaf contains at least *floor*(d / 2) keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

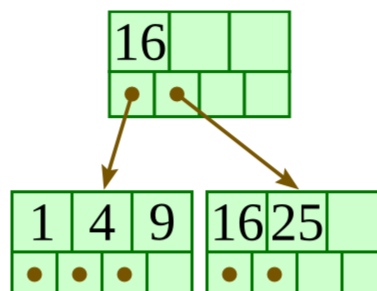
In our examples, we'll continue to use 4 for d. Looking at our invariants, this requires that each leaf have at least two keys, and each internal node to have at least two children (and thus at least one key).

Insertion algorithm

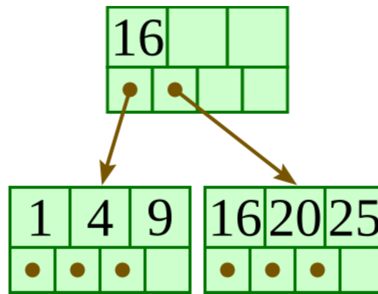
Descend to the leaf where the key fits.

1. If the node has an empty space, insert the key/reference pair into the node.
2. If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node. If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

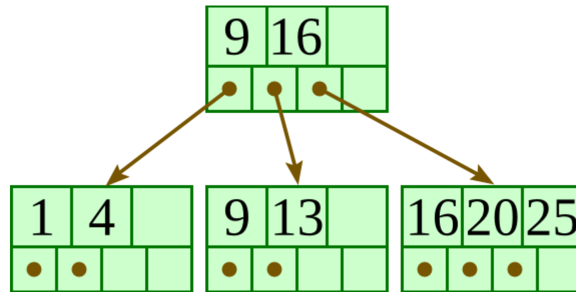
Initial:



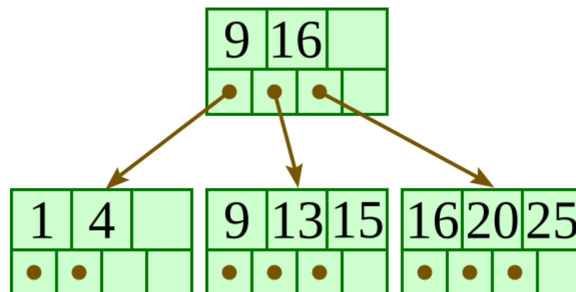
Insert 20:



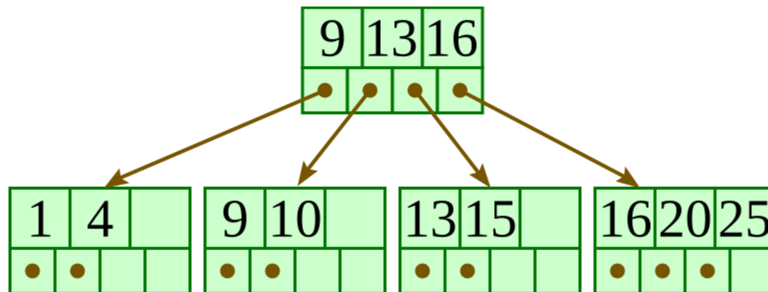
Insert 13:



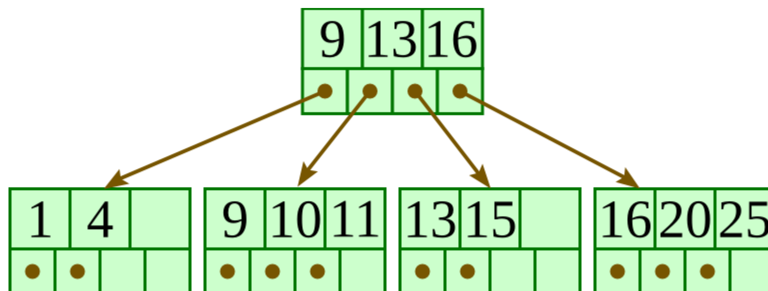
Insert 15:



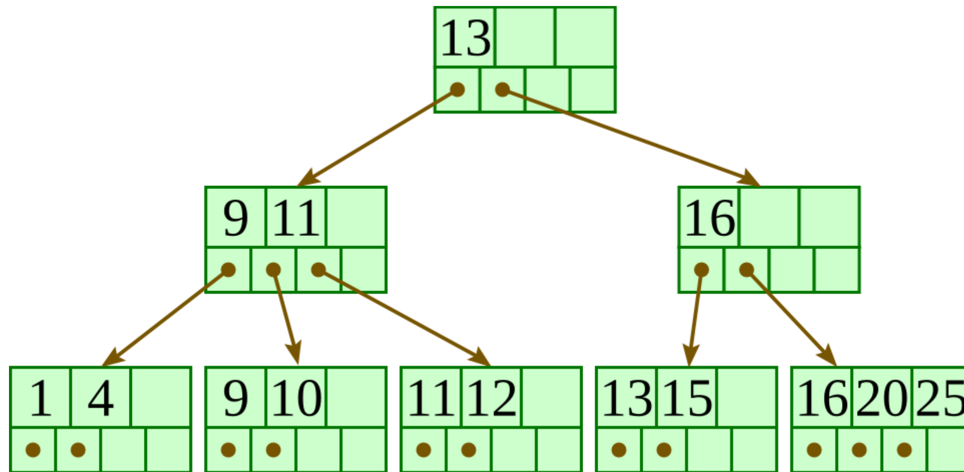
Insert 10:



Insert 11:



Insert 12:

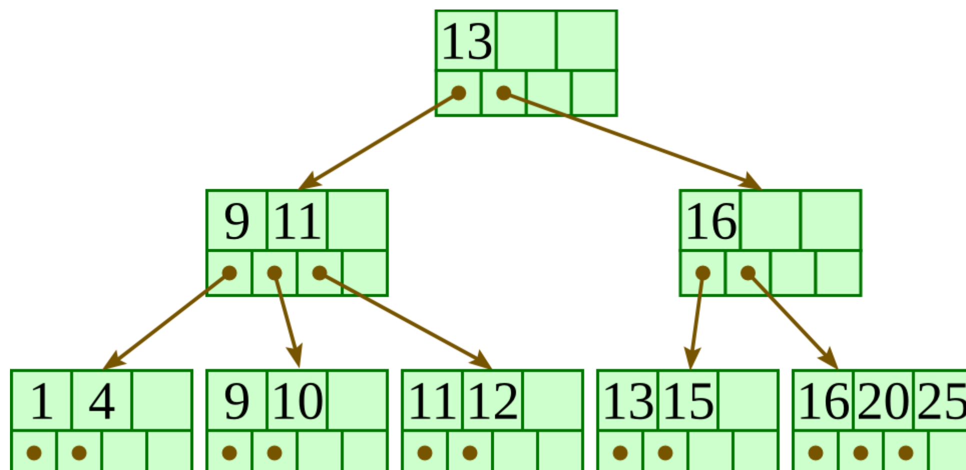


Deletion algorithm

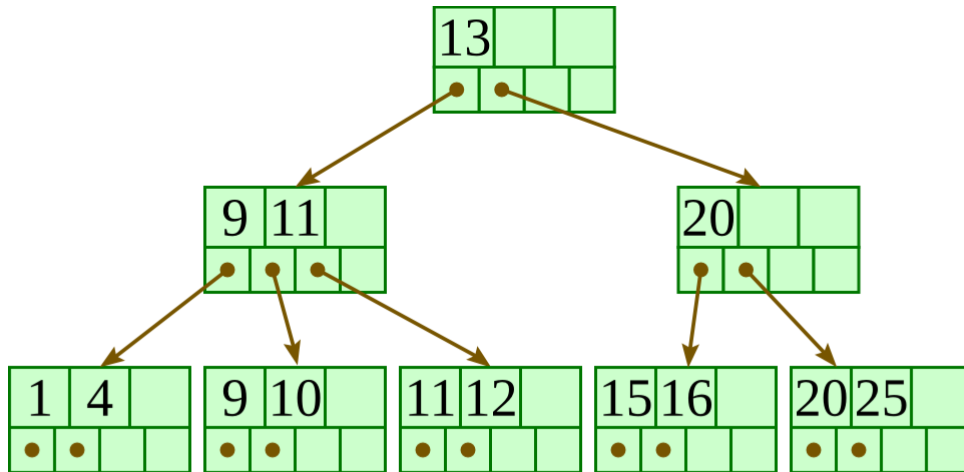
Descend to the leaf where the key exists.

1. Remove the required key and associated reference from the node.
2. If the node still has enough keys and references to satisfy the invariants, stop.
3. If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different "split point" between them; this involves simply changing a key in the levels above, without deletion or insertion.
4. If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the "split key" from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the "split key" that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

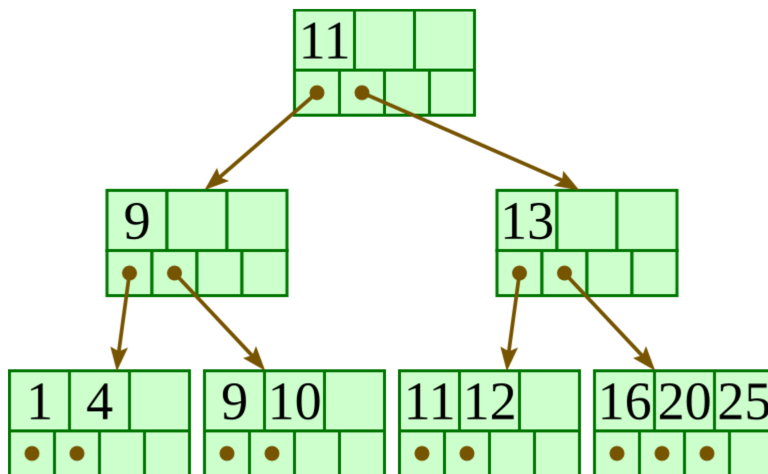
Initial:



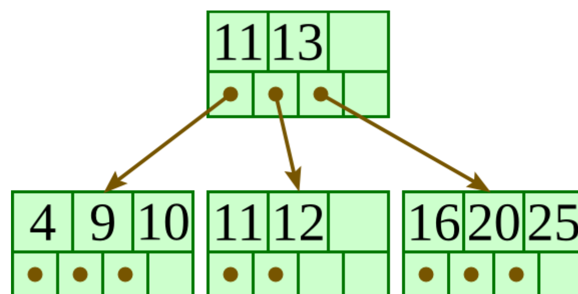
Delete 13:



Delete 15:



Delete 1:



3.11 HEAP

The main application of Binary Heap is as implement priority queue. Binomial Heap is an extension of Binary Heap that provides faster union or merge operation together with other operations provided by Binary Heap.

Representation of Binomial Heaps

- Each binomial tree within a binomial heap is stored in the left-child, right-sibling representation
- Each node *X* contains POINTERS
  - *p*[*x*] to its parent
  - child[*x*] to its leftmost child
  - sibling[*x*] to its immediately right sibling
- Each node *X* also contains the field degree[*x*] which denotes the number of children of *X*.

Operation on Binary Heap

- Merging of two binomial heaps
- Union of two binomial heaps
- Insertion of the element in the binomial heap
- Deletion of an element from the binomial heap

FIBONACCI HEAPS

Fibonacci heaps are similar to binomial heap in which collection of trees is arranged in a heap-order. The heap order means each node is smaller than each of its children. Unlike binomial heaps there may have many trees of some cardinality. It is not necessary to have exactly 2^i nodes. The Fibonacci heaps are **unordered binomial trees.**

This heap structure was originally developed for use as a priority-queue in Dijkstra's algorithm by Freedman and Tarjan in 1987.

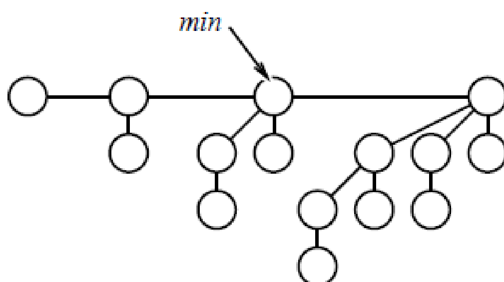


Fig:3.24(a): Fibonacci heap

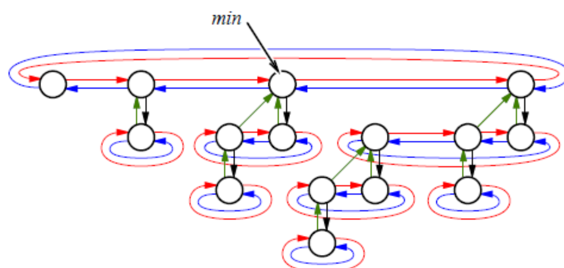


Fig3.24(b): Fibonacci heap

As shown in Fig3.24 (a) represents the simple Fibonacci heap and Fig3.24 (b) represents the linked representation of Fibonacci heap. The root list is a circular doubly linked list. Each node has a

pointer to one of its children and pointer to its parent. The child list i.e. children of a node is linked by a circular doubly linked list.

Advantage of circular doubly linked list

1. We can remove a node from circular, doubly linked list in O(1) time.
2. We can concatenate two lists into circular, doubly linked list in O(1) time.

<u>1. Insertion of a node in the Fibonacci heap</u>

```
Algorithm Create_head()
{
// Problem Description: The head of the heap is created.
// Output:  The address of head node is created.
temp[H] <- 0
min[H] <- NULL
return H
}

Algorithm Insert(H, m)
{
degree[m] <- 0
p[m] <- NULL
child[m] <- NULL
left[m]<- m
right[m] <- m
mark[m] <- FALSE
//concatenate the root list containing m with root list H
If min[H]=NULL or key[m]<key[min[H]] then
min[H]<-m
n[H]<-n[H]+1
}
```

Thus insertion of any element in Fibonacci heap is a simple concatenation of that element with the existing **root list.**

Application of Fibonacci Heaps

1. The use of Fibonacci heap improves the asymptotic running time of Dijkstra's algorithm of computing the shortest path.
2. In Prim's algorithm Fibonacci heap is helpful in finding the minimum spanning tree.

QUESTION BANK
PART-A

Tree ADT

1. Define tree. [L1]
2. Define Height of tree. [L1]
3. Define Depth of tree. [L1]
4. Define Degree of a node. [L1]
5. Define Degree of a tree. [L1]
6. Define Terminal node or leaf. [L1]
7. Define Non-terminal node. [L1]
8. Define sibling. [L1]

Tree Traversal

9. What are the different types of traversing? [L1]
10. Give the implementation of trees. [L1]
11. List the application of tree. [L1]

Binary Tree ADT

12. Define binary tree and give binary tree node structure. [L1]
13. Define complete binary tree. [L1]
14. Define full binary tree. [L1]

Binary Search Tree

15. Define binary search tree. [L1]
16. What are the various operation performed in the binary search tree. [L1]

Expression Trees

17. Define expression tree. [L1]
18. Define Construction of expression trees. [L1]

AVL Trees

19. How do we calculate the balance factor for each node in a AVL tree? [L3]
20. What are the various transformation performed in AVL tree? [L1]

21. What are the advantages of AVL trees? [L1]
22. What are the Disadvantages of AVL trees? [L1]
23. List the applications of AVL trees. [L1]
24. What is the minimum number of nodes in an AVL tree of height h? [L1]

B – Tree & B+ Tree

25. What are B-trees? [L1]
26. What are the properties of B-trees? [L1]
27. List the the various operations that can be performed on B-trees. [M/J 2016] [L1]

Heap

28. Define binary heap. [L1]
29. What are the types of heap ordering property? [L1]
30. What are binomial heap? [L1]
31. Give the representation of a node in binomial heap. [L1]
32. Define Fibonacci heap. [L1]
33. What are the advantages of Fibonacci heaps? [L1]
34. How Fibonacci heap differ from binomial heap. [L1]

PART-B

1. Explain in detail about tree data structure. **[Pg. No:77]** [L1]
2. Explain in detail about binary tree data structure. **[Pg. No:79]** [L1]
3. Explain in detail about binary search tree and its operation with example. **[Pg. No:82]** [L1]
4. Draw a binary search tree for the following input list 60, 25, 75, 15, 50, 66, 33, 44. Trace the algorithm to delete the nodes 25, 75, 44 from the tree. **[Pg. No:82]** [L2]
5. Explain in detail about AVL tree with suitable example. **[Pg. No:87]** [**Nov/Dec -2015**] [L2]
   Explain the AVL rotation with a suitable example. **[Pg. No:87]**[**May/Jun 2016**] [L2]
6. Construct an AVL tree with the value 3, 1, 4, 5, 9, 2, 8, 7, 0 into an initially empty tree. Write the code for inserting into an AVL tree. (10)**[Pg. No:87]**[**Apr/May -2015**] [L2]
   Define AVL tree and starting with an empty AVL search tree insert the following element in the given order: 2,1,4,5,9,3,6,7. (8) **[Pg. No:87]**[**May/Jun 2016**] [L2]
7. Show the result of inserting 43, 11, 69, 72, and 30 into an initially empty AVL tree.
   Show the result of deleting the nodes 11 and 72 one after the other of the constructed tree **[Pg. No:87]** [**Nov/Dec-2014**] [L2]
8. Write the routines for AVL tree operations. **[Pg. No:90]** [L2]

9. What is B-Tree? Mention the properties that a b-B-tree holds and Construct the B-Tree for the following number 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45. **[Pg. No:94]**[**Nov/Dec-2014**] [L2]

10. Construct a B-Tree with order m=3 for the key values 2, 3, 7, 9, 5, 6, 4, 8,1 and delete the value 4 and 6. Show the tree performing all operation. **[Pg. No:94]**[**May/Jun 2016**][L3]

PART-A TWO MARKS

Tree ADT

1. Define tree.

Trees are non-liner data structure, which has collections of nodes connected by directed (or undirected) edges. A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or more sub trees.

2. Define Height of tree.

The height of n is the length of the longest path from root to a leaf. Thus all leaves have height zero. The height of a tree is equal to a height of a root.

3. Define Depth of tree.

For any node n, the depth of n is the length of the unique path from the root to node n. Thus for a root the depth is always zero.

4. Define Degree of a node.

It is the number of sub trees of a node in a given tree.

5. Define Degree of a tree.

It is the maximum degree of a node in a given tree.

6. Define Terminal node or leaf.

Nodes with no children are known as leaves. A leaf will always have degree zero and is also called as terminal node.

7. Define Non-terminal node.

Any node except the root node whose degree is a non-zero value is called as a non-terminal node. Non-terminal nodes are the intermediate nodes in traversing the given tree from its root node to the terminal node.

8. Define sibling.

Nodes with the same parent are called siblings.

Tree Traversal

9. What are the different types of traversing?

Traversing a tree means processing it in such a way, that each node is visited only once. The different types of traversing are

a. Pre-order traversal-yields prefix form of expression.

b. In-order traversal-yields infix form of expression.

c. Post-order traversal-yields postfix form of expression.

10. Give the implementation of tree

The tree can be implemented by two ways

1. Sequential implementation – The tree is implemented using array in this type of implementation.
2. Linked implementation – The tree is implemented or represented using linked list.

11. List the application of tree.

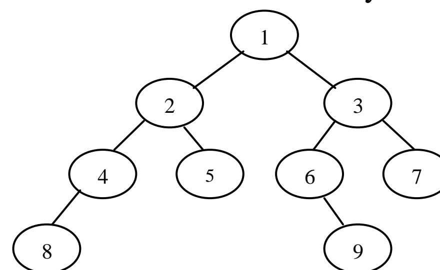
- Binary search tree
- Expression tree
- Game tree
- Threaded binary tree

Binary Tree ADT

12. Define binary tree and give binary tree node structure.

A binary tree is a tree data structure in which each node has ***at most two childern***, which are referred to as the left child and the right child. The total number of nodes at any level n in a binary tree is 2^{n+1}.

```
Struct BST
{
int data;
BST *leftchild;
BST *rightchild;
}
```

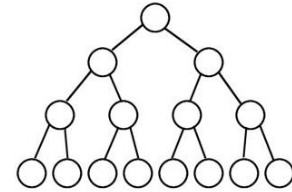


13. Define complete binary tree.

A complete binary tree is a tree in which every node except the leaf nodes should have exactly two children not necessarily on the same level.

14. Define full binary tree.

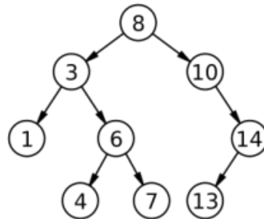
A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.



Binary Search Tree

15. Define binary search tree.

Binary search tree is a binary tree in which each node is systematically arranged i.e the left child has less value than its parent node and right child has greater value than its parent node. The searching of any node in such a tree becomes efficient in this type of tree. The time complexity of binary search tree is $O(n \log_2 n)$.



16. What are the various operation performed in the binary search tree?

- Insertion
- Deletion
- Find
- Find Min V. Find Max

Expression Trees

17. Define expression tree.

Expression tree is also a binary tree in which the leafs terminal nodes or operands and non-terminal intermediate nodes are operators used for traversal.

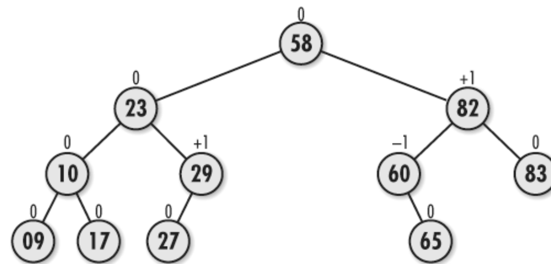
18. Define Construction of expression trees .

- Convert the given infix expression into postfix notation
- Create a stack and read each character of the expression and push into the stack, if operands are encountered.
- When an operator is encountered pop 2 values from the stack.
-

AVL Trees

19. How do we calculate the balance factor for each node in a AVL tree?

AVL is a height balanced tree in which every node will have a balancing factor of –1, 0, +1. Balancing factor of a node is given by the difference between the height of the left sub tree and the height of the right sub tree. It is named as AVL tree structure was introduced by three scientists Adelsion, Velski, and Lendis.



20. What are the various transformation performed in AVL tree?

- Single rotation
  1. Single L rotation
  2. Single R rotation
- Double rotation
  1. LR rotation
  2. RL rotation

21. What are the advantages of AVL trees?

Worst case O(log N) for insert, delete and search.

22. What are the disadvantages of AVL trees?

- Extra space for maintaining height information at each node.
- Insertion and deletion become more complicated.
- Difficult to program & debug; more space for balance factor.
- Asymptotically faster but rebalancing costs time.
- Most large search are done in database systems on disk and use other structures (E.g. B-trees).

23. List the applications of AVL trees

- They are used in quick searching applications.
- They are used when data needs to be stored in sorted or nearly sorted.

24. What is the minimum number of nodes in an AVL tree of height h?

The minimum number of nodes S(h), in an AVL tree of height h is given by S(h) = S(h-1)+S(h-2)+1. For h=0,S(h)=1.

B – Tree & B+ Tree

25. What are B-trees?

B-tree is a multi-way search tree in that node can have more than two children. If there are n numbers of children in a node then (n-1) is the number of keys in the node.

26. What are the properties of B-trees?

- All leaf nodes are at the same level.
- The root node should have at least two children.
- All non leaf nodes (except the root) have at most m and at least m/2 children.
- Each leaf node must contain at least (m/2)-1 keys.

27. List the the various operations that can be performed on B-trees. [M/J 2016]

Search. Traversal, Insertion, Deletion

Heap

28. Define binary heap.

A binary heap is a complete binary tree which satisfies the heap ordering property and structure property.

29. What are the types of heap ordering property?

- **The minimum heap property:** The value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- **The maximum-heap property:** The value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

30. What are binomial heap?

A binomial heap is a collection of binomial trees that satisfies the following binomial-heap properties:

- No two binomial trees in the collection have the same size.
- Each node in each tree has a key.
- Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.

31. Give the representation of a node in binomial heap.

Parent[X]	
Key[x]	
Degree[x]	
Child[x]	Sibling[x]

32. Define Fibonacci heap. [Nov-Dec 2014]

A Fibonacci heap H is a collection of heap-ordered trees that have the following properties:

- The roots of these trees are kept in a doubly-linked list(the root list of H)
- The root of each tree contains the minimum element in that tree (this follows from being a heap-ordered tree)
- We access the heap by a pointer to the tree root with the overall minimum key.
- For each node x, we keep track of the degree (also known as the order or rank) of x, which is just the number of children x has, we also keep track of the mark of x, which is a Boolean value.'

33. What are the advantages of Fibonacci heaps?

- The use of Fibonacci heap improves the asymptotic running time of Dijkstra's algorithm of computing the shortest path.
- In prim's algorithm Fibonacci heap is helpful in finding the minimum spanning tree.

34. How Fibonacci heap differ from binomial heap. [April-May 2015]

Sl.no	Binomial Heap	Fibonacci Heap
1	Binomial heap takes O(log n) time in all operations.	Fibonacci heap takes amortized running time O (1) in Insert, find, decrease key operation and O(log n) time in delete min, delete operations.
2	Binomial heaps use a single linked circular link list.	Fibonacci heaps use a doubly linked circular linked list.
3	Every binomial heap is a Fibonacci heap	Every Fibonacci heap isn't binomial heap.
4	Delete-min in binomial heaps involves the combining of trees	Delete-min or delete in Fibonacci heaps is performed without joining the trees obtained after deletion.