THREADED BINARY TREE

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position. In any binary tree linked list representation, there are more number of NULL pointer than actual pointers. Generally, in any binary tree linked list representation, if there are **2N** number of reference fields, then **N+1** number of reference fields are filled with NULL (**N+1 are NULL out of 2N**). This NULL pointer does not play any role except indicating there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called **threads**.

A threaded binary tree defined as follows:

"A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

Why do we need Threaded Binary Tree?

Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals. Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.

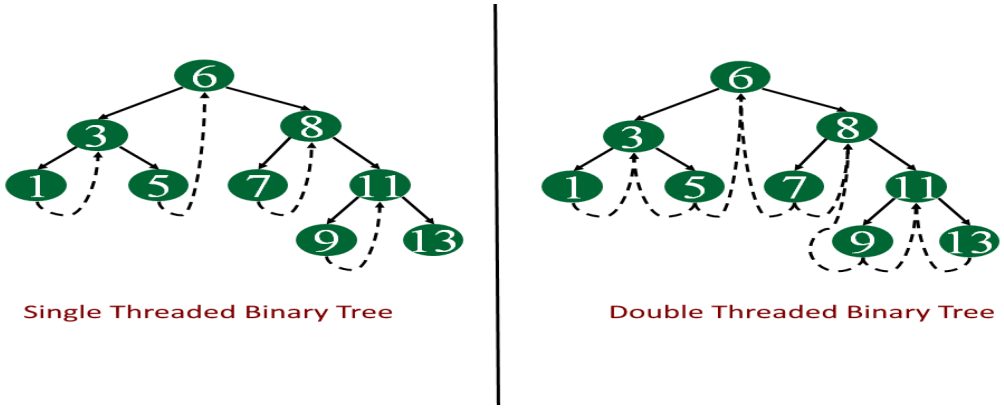
Comparison between a normal binary tree and threaded binary tree

Threaded Binary Trees	Normal Binary Trees
• In threaded binary trees, The null pointers are used as thread.	• In a normal binary trees, the null pointers remains null.
• We can use the null pointers which is a efficient way to use computers memory.	• We can't use null pointers so it is a wastage of memory.
• Traversal is easy. Completed without using stack or reccursive function.	• Traverse is not easy and not memory efficient.
• Structure is complex.	• Less complex than Threaded binary tree.
• Insertion and deletion takes more time.	• Less Time consuming than Threaded Binary tree.

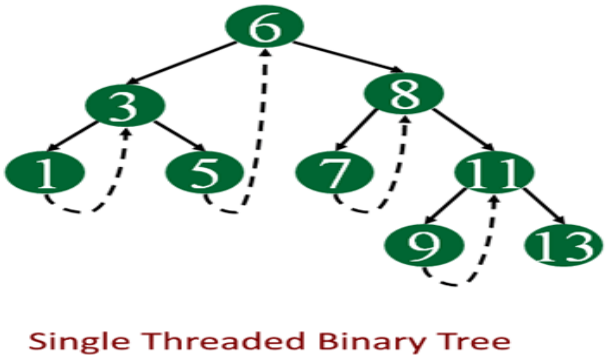
Types of threaded binary trees:

Single Threaded: each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

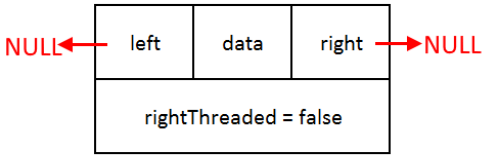
Double threaded: each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor AND all left null pointers will point to inorder predecessor.



Single Threaded: each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.



Implementation:
Let's see how the Node structure will look like




```
class Node{
   Node left;
   Node right;
   int data;
```

```
    boolean rightThread;
    public Node(int data){
        this.data = data;
        rightThread = false;
    }
}
```

In normal BST node we have left and right references and data but in threaded binary tree we have boolean another field called “rightThreaded”. This field will tell whether node’s right pointer is pointing to its inorder successor, but how, we will see it further.

Operations:

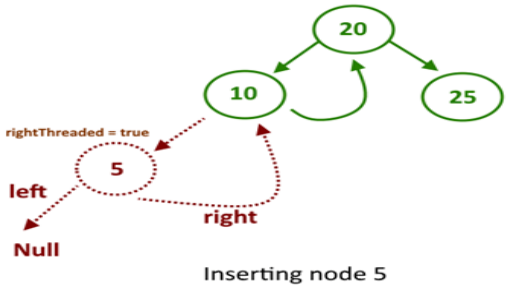
Insert node into tree

Print or traverse the tree.

Insert():

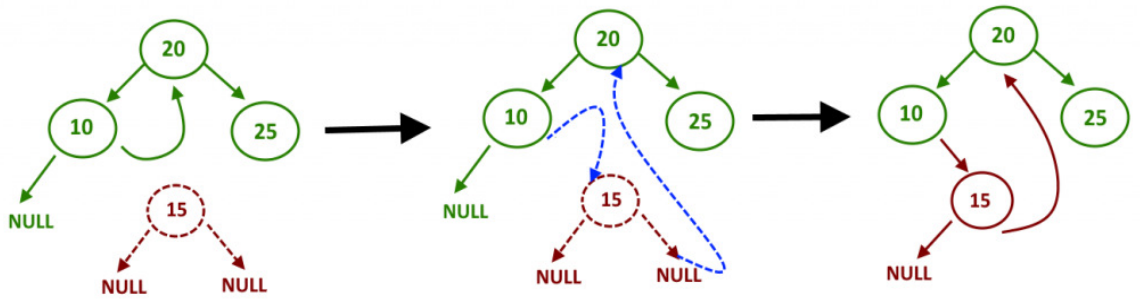
The insert operation will be quite similar to Insert operation in Binary search tree with few modifications.To insert a node our first task is to find the place to insert the node.

- Take current = root .
- Start from the current and compare root.data with n.
- Always keep track of parent node while moving left or right.
- if current.data is greater than n that means we go to the left of the root, if after moving to left, the current = null then we have found the place where we will insert the new node. Add the new node to the left of parent node and make the right pointer points to parent node and rightThread = true for new node.



Inserting node 5

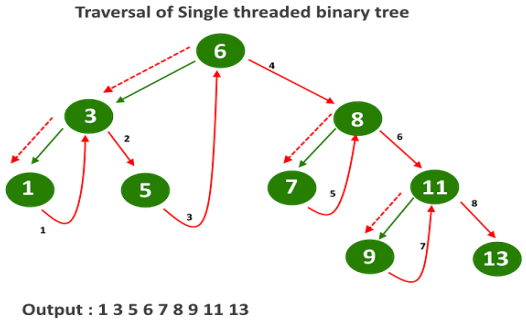
- if current.data is smaller than n that means we need to go to the right of the root, while going into the right sub tree, check rightThread for current node, means right thread is provided and points to the in order successor, if rightThread = false then and current reaches to null, just insert the new node else if rightThread = true then we need to detach the right pointer (store the reference, new node right reference will be point to it) of current node and make it point to the new node and make the right reference point to stored reference.



Inserting Node 15 into threaded binary tree

Traverse():

Traversing the threaded binary tree will be quite easy, no need of any recursion or any stack for storing the node. Just go to the left most node and start traversing the tree using right pointer and whenever rightThread = false again go to the left most node in right sub-tree.




```
Node leftMost(Node n) {
   Node ans = n;
   if (ans == null) {
       return null;
   }
   while (ans.left != null) {
      ans = ans.left;
   }
   return ans;
}
void inOrder(Node n) {
   Node cur = leftmost(n);
   while (cur != null) {
      print(cur);
      if (cur.rightThread) {
         cur = cur.right;
      } else {
         cur = leftmost(cur.right);
      }
   }
}
```

HEIGHT BALANCED TREES (AVL TREES)

The Height balanced trees were developed by researchers Adelson-Velskii and Landis. Hence these trees are also called AVL trees. Height balancing attempts to maintain the balance factor of the nodes within limit.

Height of the tree: Height of a tree is the number of nodes visited in traversing a branch that leads to a leaf node at the deepest level of the tree.

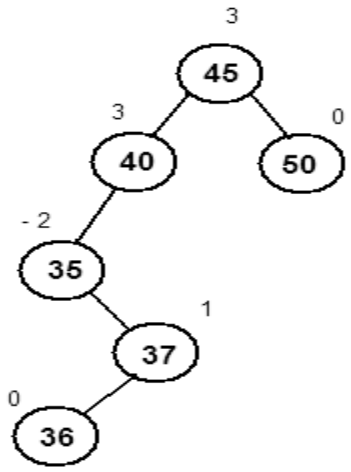
Balance factor: The balance factor of a node is defined to be the difference between the height of the node's left subtree and the height of the node's right subtree.

Consider the following tree. The left height of the tree is 5, because there are 5 nodes (45, 40, 35, 37 and 36) visited in traversing the branch that leads to a leaf node at the deepest level of this tree.

Balance factor = height of left subtree – height of the right subtree

In the following tree the balance factor for each and every node is calculated and shown. For example, the balance factor of node 35 is (0 – 2) = - 2.

The tree which is shown below is a binary search tree. The purpose of going for a binary search tree is to make the searching efficient. But when the elements are added to the binary search tree in such a way that one side of the tree becomes heavier, then the searching becomes inefficient. The very purpose of going for a binary search tree is not served. Hence we try to adjust this unbalanced tree to have nodes equally distributed on both sides. This is achieved by rotating the tree using standard algorithms called the AVL rotations. After applying AVL rotation, the tree becomes balanced and is called the AVL tree or the height balanced tree.

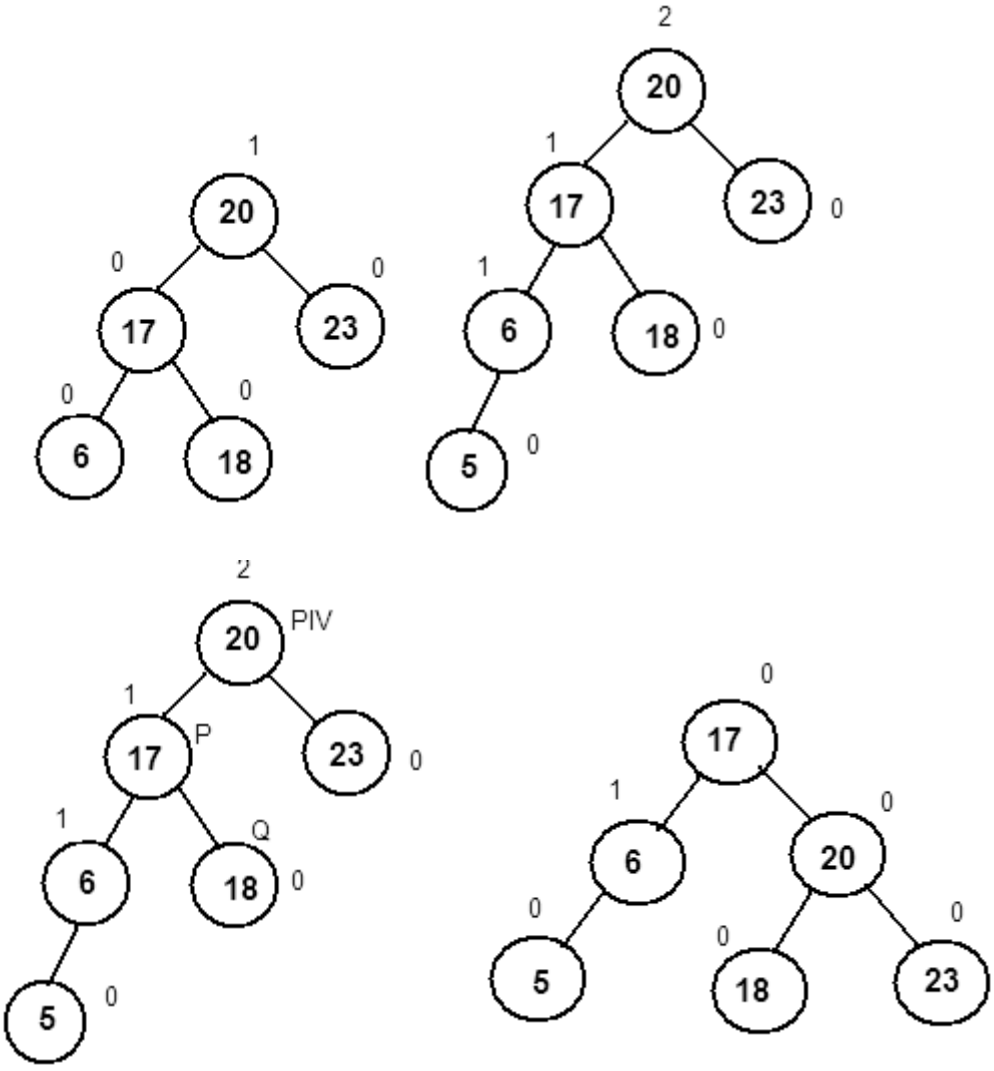


The tree is said to be balanced if each node consists of a balance factor either -1 or 0 or 1. If even one node has a balance factor deviated from these values, then the tree is said to be unbalanced. There are four types of rotations. They are:

1. *Left-of-Left rotation.*
2. *Right-of-Right rotation.*
3. *Right-of-Left rotation.*
4. *Left-of-Right rotation.*

Left-of-Left Rotation

Consider the following tree. Initially the tree is balanced. Now a new node 5 is added. This addition of the new node makes the tree unbalanced as the root node has a balance factor 2. Since this is the node which is disturbing the balance, it is called the pivot node for our rotation. It is observed that the new node was added as the left child to the left subtree of the pivot node. The pointers P and Q are created and made to point to the proper nodes as described by the algorithm. Then the next two steps rotate the tree. The last two steps in the algorithm calculates the new balance factors for the nodes and is seen that the tree has become a balanced tree.

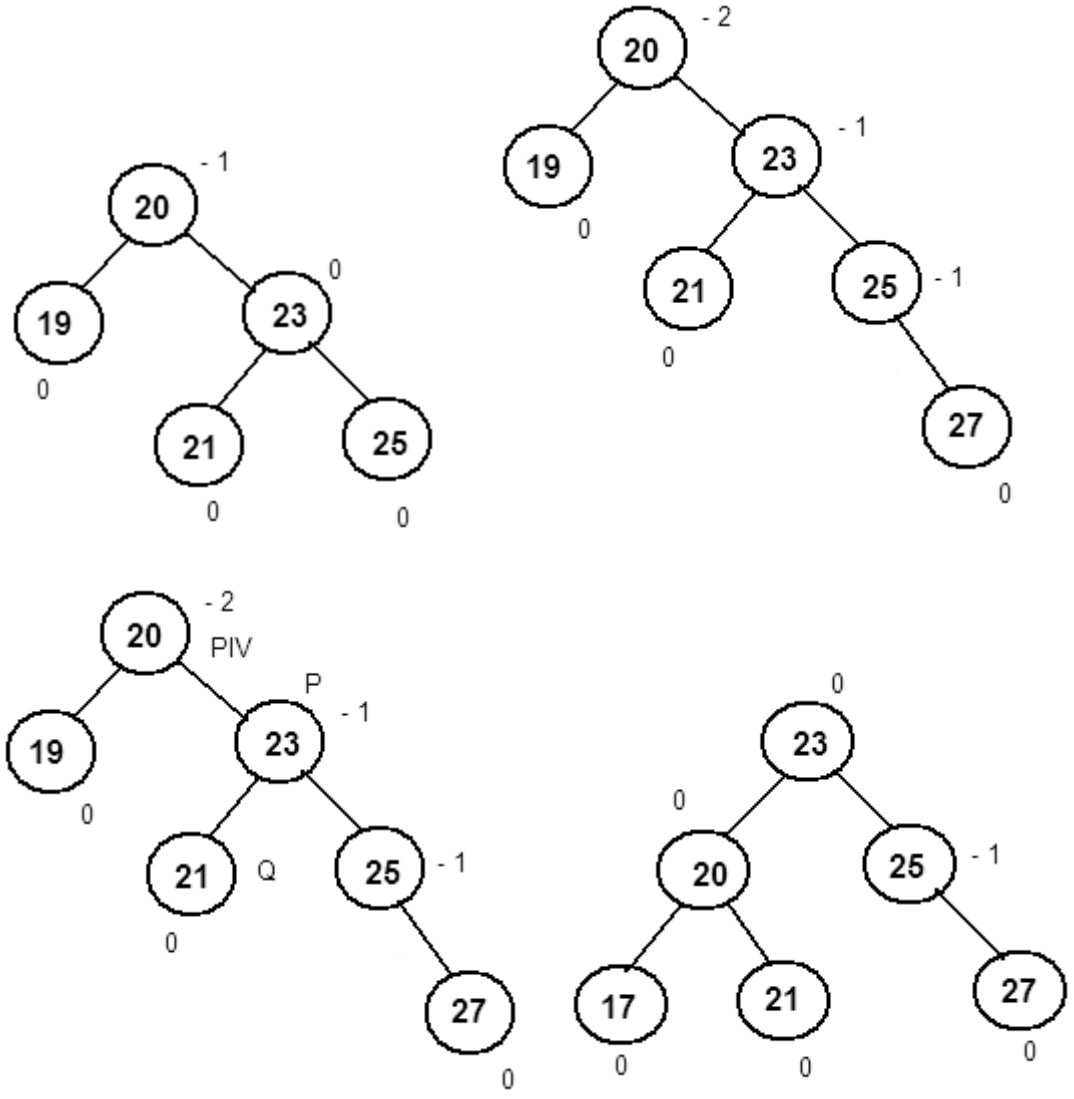


Algorithm

LEFT-OF-LEFT(pivot)
P = left(pivot)
Q = right(P)
Root = P
Right(P) = pivot
Left(pivot) = Q
Bal(pivot) = 0
Bal(right(pivot)) = 0
End LEFT-OF-LEFT

Right-of- Right Rotation

In this case, the pivot element is fixed as before. The new node is found to be added as the right child to the right subtree of the pivot element. The first two steps in the algorithm sets the pointer P and Q to the correct positions. The next two steps rotate the tree to balance it. The last two steps calculate the new balance factor of the nodes.

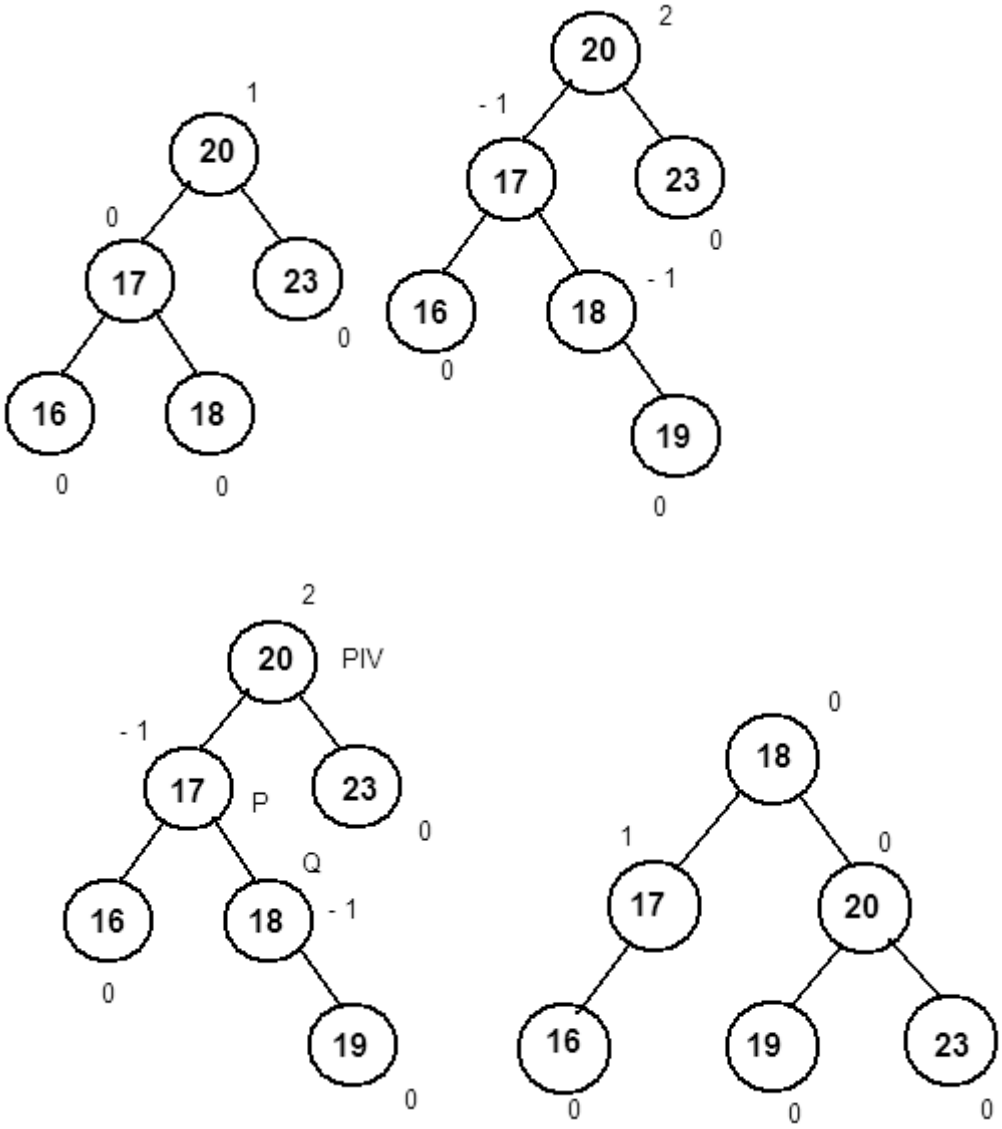


Algorithm

RIGHT-OF-RIGHT(pivot)
P = right(pivot)
Q = left(P)
Root = P
Left(P) = pivot
Right(pivot) = Q
Bal(pivot) = 0
Bal(left(pivot)) = 0
End RIGHT-OF-RIGHT

Right-of-Left Rotation

In this following tree, a new node 19 is added. This is added as the right child to the left subtree of the pivot node. The node 20 fixed as the pivot node, as it disturbs the balance of the tree. In the first two steps the pointers P and Q are positioned. In the next four steps, tree is rotated. In the remaining steps, the new balance factors are calculated.

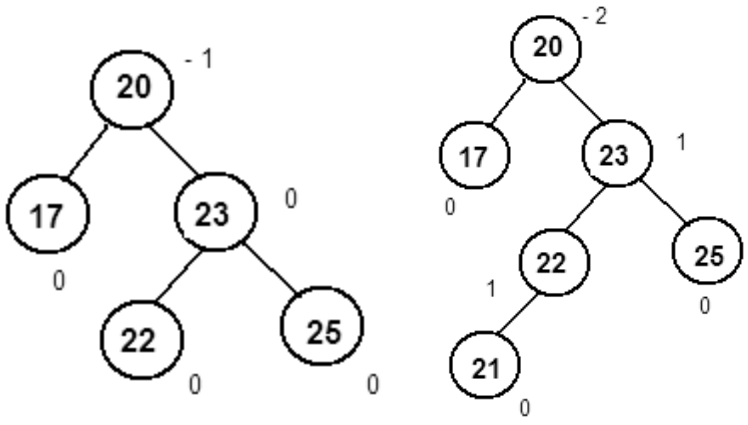


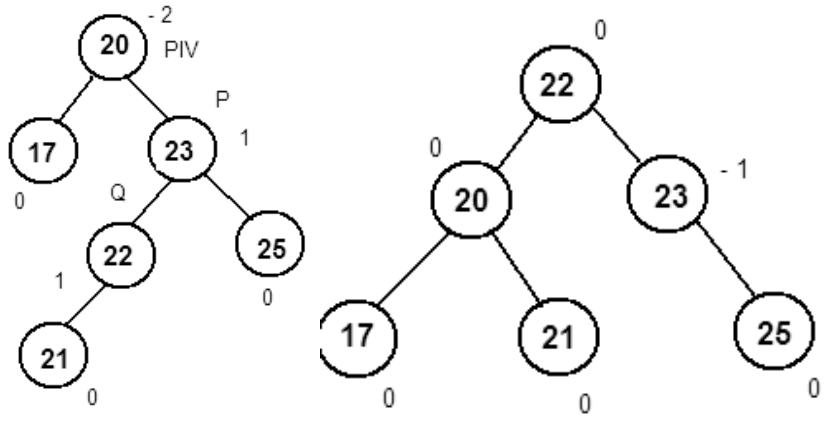
Algorithm

```
RIGHT-OF-LEFT(pivot)
P = left(pivot)
Q = right(P)
Root = Q
Left(Q) = P
Right(Q) = Pivot
Left(pivot) = right(Q)
Right(P) = left(Q)
If Bal(pivot) = 0
        Bal(left(pivot)) = 0
        Bal(right(pivot)) = 0
Else
        If Bal(pivot) = 1
                Bal(pivot) = 0
                Bal(left(pivot)) = 0
                Bal(right(pivot)) = -1
        Else
                Bal(pivot) = 0
                Bal(left(pivot)) = 1
                Bal(right(pivot)) = 0
        End if
End if

End RIGHT-OF-LEFT
```

Left-of-Right

In the following tree, a new node 21 is added. The tree becomes unbalanced and the node 20 is the node which has a deviated balance factor and hence fixed as the pivot node. In the first two steps of the algorithm, the pointers P and Q are positioned. In the next 4 steps the tree is rotated to make it balanced. The remaining steps calculate the new balance factors for the nodes in the tree.





Algorithm

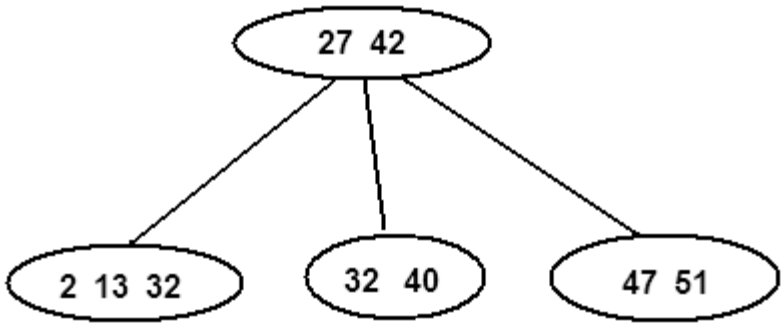
LEFT-OF-RIGHT(pivot)

```
P = right(pivot)
Q = left(P)
Root= Q
Right(Q) = P
Left(Q) = Pivot
Right(pivot) = left(Q)
Left(P) = right(Q)
If Bal(pivot) = 0
        Bal(right(pivot)) = 0
        Bal(left(pivot)) = 0
Else
        If Bal(pivot) = 1
                Bal(pivot) = 0
                Bal(right(pivot)) = 0
                Bal(left(pivot)) = -1
        Else
                Bal(pivot) = 0
                Bal(right(pivot)) = 1
                Bal(left(pivot)) = 0
        End if
End if
End LEFT-OF-RIGHT
```

B – TREES

Multiway search tree (m-way search tree): *Multiway search tree of order n is a tree in which any node may contain maximum n-1 values and can have maximum n children.*

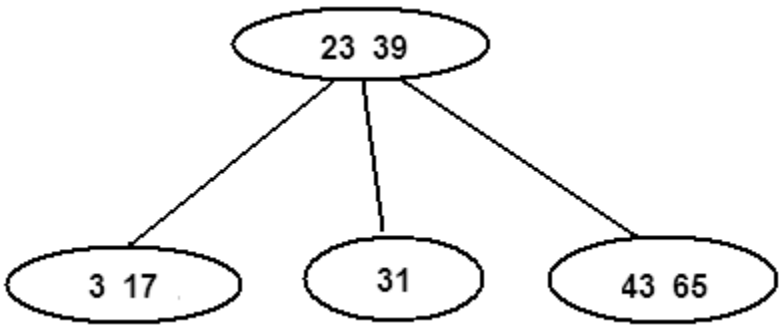
Consider the following tree. Every node in the tree has one or more than one values stored in it. The tree shown is of order 3. Hence this tree can have maximum 3 children and each node can have maximum 2 values. Hence it is an m-way search tree.



B – Tree: *B – tree is a m-way search tree of order n that satisfies the following conditions.*

(i) *All non-leaf nodes (except root node) have at least n/2 children and maximum n children.*
(ii) *The non-leaf root node may have at least 2 children and maximum n children.*
(iii) *B-Tree can exist with only one node. i.e. the root node containing no child.*
(iv) *If a node has n children then it must have n-1 values.*
(v) *All the values that appear on the left most child of a node are smaller than the first value of that node. All values that appears on the right most child of a node are greater that the last values of that node.*
(vi) *If x and y are any two i^{th} and $(i+1)^{th}$ values of a node, where $x < y$, then all the values appearing on the $(i+1)^{th}$ sub-tree of that node are greater than x and less than y.*
(vii) *All the leaf nodes should appear on the same level. All the nodes except root node should have minimum n/2 values.*

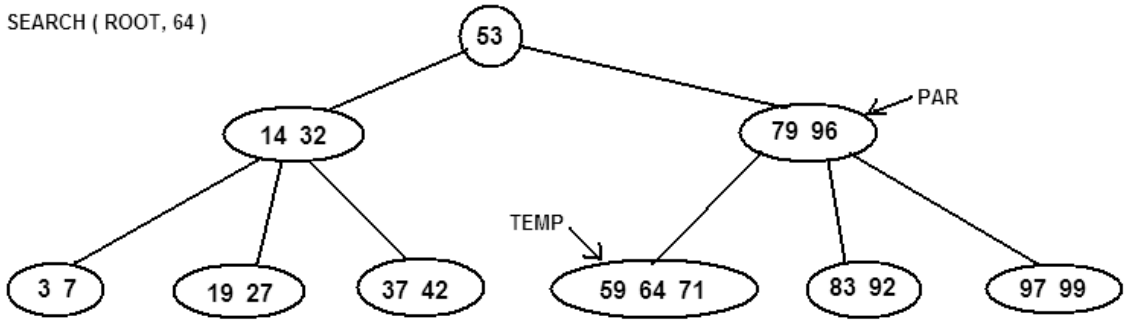
Consider the following tree. Clearly, it is a m-way search tree of order 3. Let us check whether the above conditions are satisfied. It can be seen that root node has 3 children and therefore has only 2 values stored in it. Also it is seen that the elements in the first child (3, 17) are lesser than the value of the first element (23) of the root node. The value of the elements in the second child (31) is greater than the value of the first element of the root node (23) and less than the value of the second element (39) in the root node. The value of the elements in the rightmost child (43, 65) is greater than the value of the rightmost element in the root node. All the three leaf nodes are at the same level (level 2). Hence all the conditions specified above is found to be satisfied by the given m-way search tree. Therefore it is a B-Tee.



Search Operation in a B-Tree

Let us say the number to be searched is k = 64. A temporary pointer temp is made to initially point to the root node. The value k = 64 is now compared with each element in the node pointed by temp. If the value is found then the address of the node where it is found is returned using the temp pointer. If the value k is greater than i^{th} element of the node, then the temp is moved to the $i+1^{th}$ node and the search process is repeated. If the k value is lesser than the first value in the node, then the temp is moved to the first child. If the k value is greater than the last value of the node, then temp is moved to the rightmost child of the node and the search process is repeated.

After the particular node where the value is found is located (now pointed by temp), then a variable LOC is initialized to 0, indicating the position of the value to be searched within that node. The value k is compared with each and every element of the node. When the value of the k is found within the node, then the search comes to an end position where it is found is stored in LOC. If not found the value of LOC is zero indicating that the value is not found.



Algorithm

SEARCH(ROOT, k)

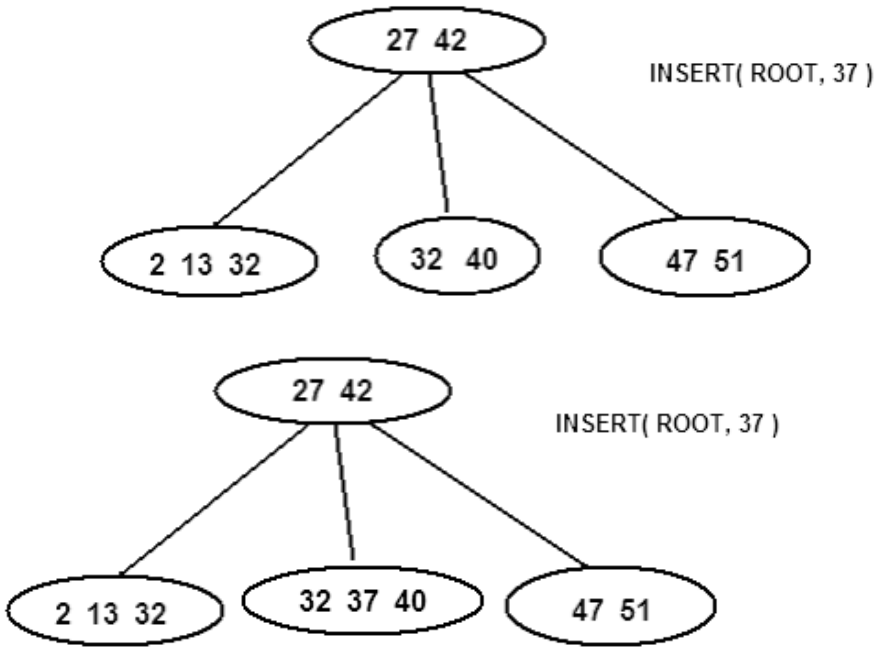
```
Temp = ROOT, i = 1, pos = 0
While i ≤ count(temp) and child[i](temp) ≠ NULL
        If k = info(temp[i])
                Pos = i
                Return temp
        Else
                If k < info(temp[i])
                        SEARCH(child[i](temp), k)
                Else
                        If i = count(temp)
                                Par = temp
                                Temp = child[i+1](temp)
                        Else
                                i = i + 1
                        End if
                End if
        End if
```

```
End While

While i ≤ count(temp)
        If k = info(temp[i])
                Pos = i
                Return temp
        End if
End while

End SEARCH
```

Insert Operation in a B-Tree

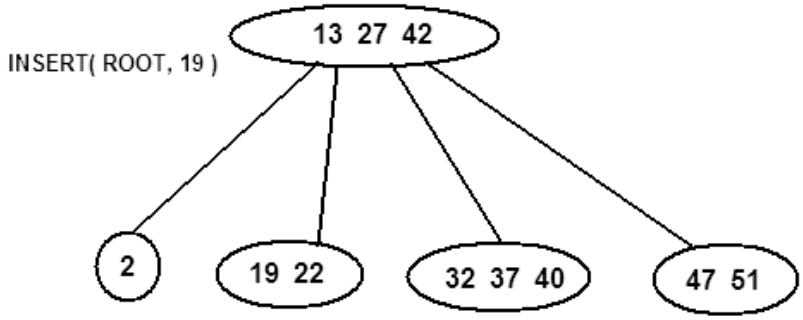
One of the conditions in the B-Tree is that, the maximum number of values that can be present in the node of a tree is n – 1, where n is the order of the tree. Hence, it should be taken care that, even after insertion, this condition is satisfied. There are two cases: In the first case, the element is inserted into a node which already had less than n- 1 values, and the in the second case, the element is inserted into a node which already had exactly n-1 values. The first case is a simple one. The insertion into the node does not violate any condition of the tree. But in the second case, if the insertion is done, then after insertion, the number values exceeds the limit in that node.

Let us take the first case. In both the cases, the insertion is done by searching for that element in the tree which will give the node where it is to be inserted. While searching, if the value is already found, then no insertion is done as B-Tree is used for storing the key values and keys do not have duplicates. Now the value given is inserted into the node. Consider the figure which shows how value 37 is inserted into correct place.



In the second case, insertion is done as explained above. But now, it is found that, the number of values in the node after insertion exceeds the maximum limit. Consider

the same tree shown above. Let us insert a value 19 into it. After insertion of the value 19, the number of values (2, 13, 19, 22) in that node has become 4. But it is a B-Tree of order 4 in which there should be only maximum 3 values per node. Hence the node is split into two nodes, the first node containing the numbers starting from the first value to the value just before the middle value (first node: 2). The second node will contain the numbers starting just after the mid value till the last value (second node: 19, 22). The mid value 13 is pushed into the parent. Now the adjusted B-Tree appears as shown.



Algorithm

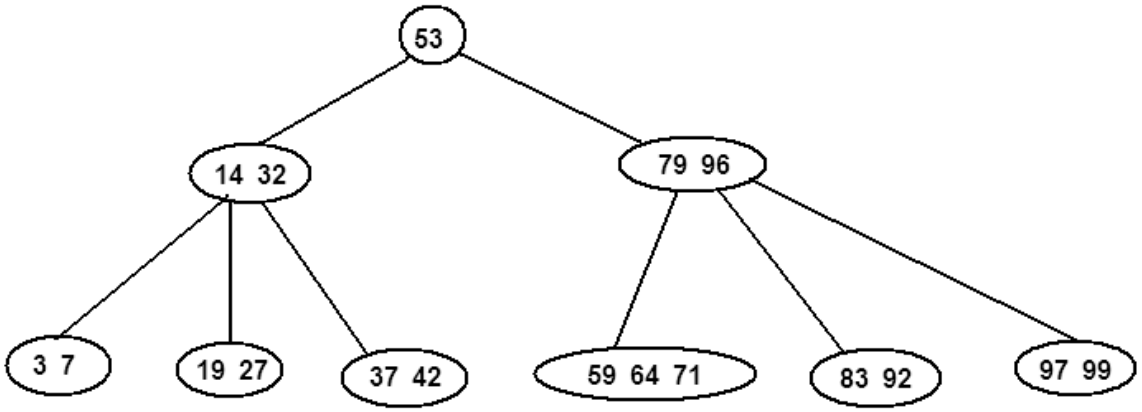
```
INSERT( ROOT, k )

Temp = SEARCH( ROOT, k )
If count(temp) < n-1
        Ins(temp, k)
        Return
Else
        Repeat for i = n/2 +1 to n-1
                Info(R[i-n/2]) = info(temp[i])
                Count(R) = count(R) + 1
        End repeat
        Count(temp) = n/2 – 1
        Ins(par, info(temp[m/2])
End if
INSERT( ROOT, k )

End INSERT
```

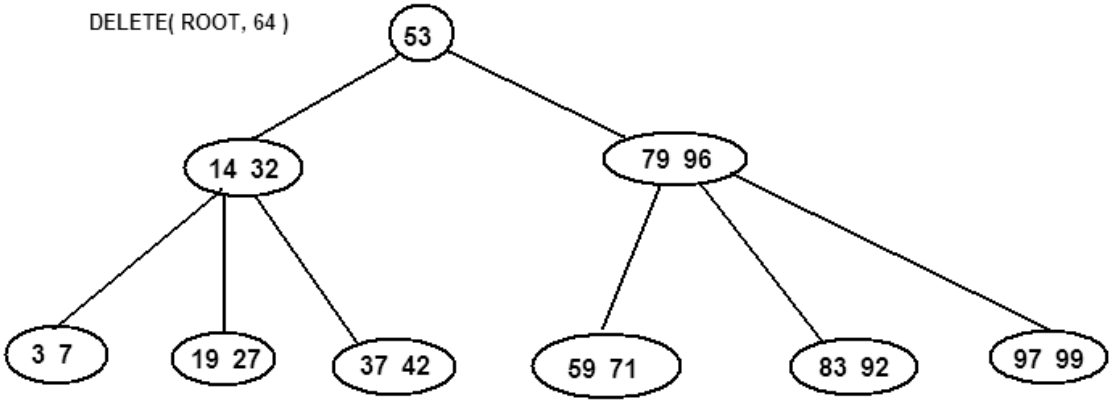
Delete Operation in B-Tree

When the delete operation is performed, we should take care that even after deletion, the node has minimum n/2 value in it, where n is the order of the tree.

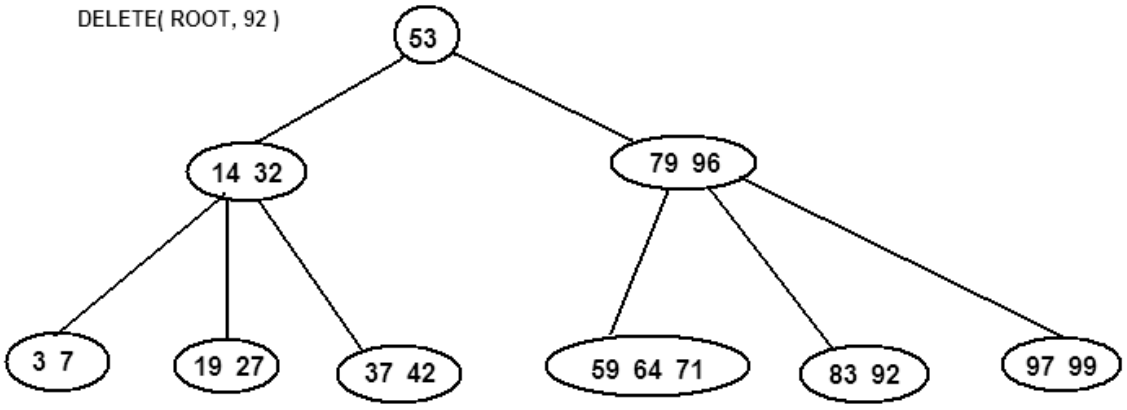
There are three cases:

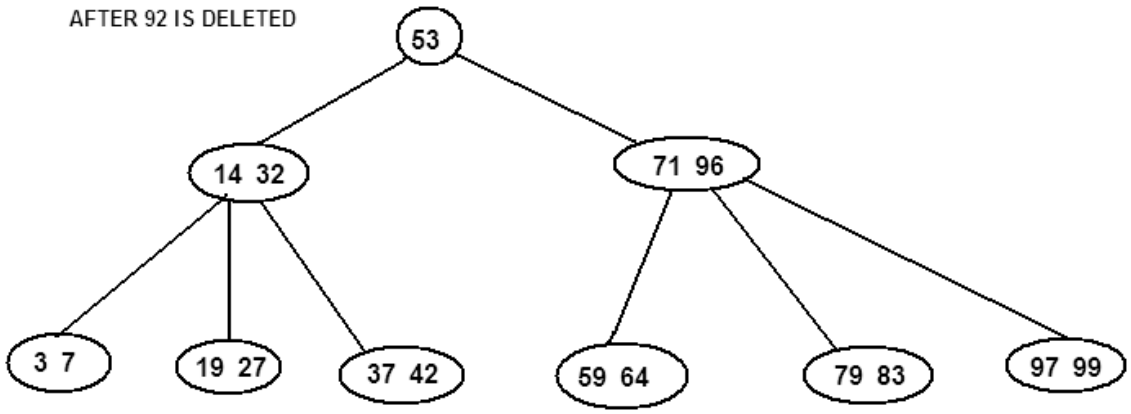
Case 1: The node from which the value is deleted has minimum n/2 values even after deletion. Let us consider the following B-Tree of order 5. A value 64 is to be deleted. Even after the deletion of the value 64, the node has minimum n/2 values (i.e., 2 values). Hence the rules of the B-Tree are not violated.



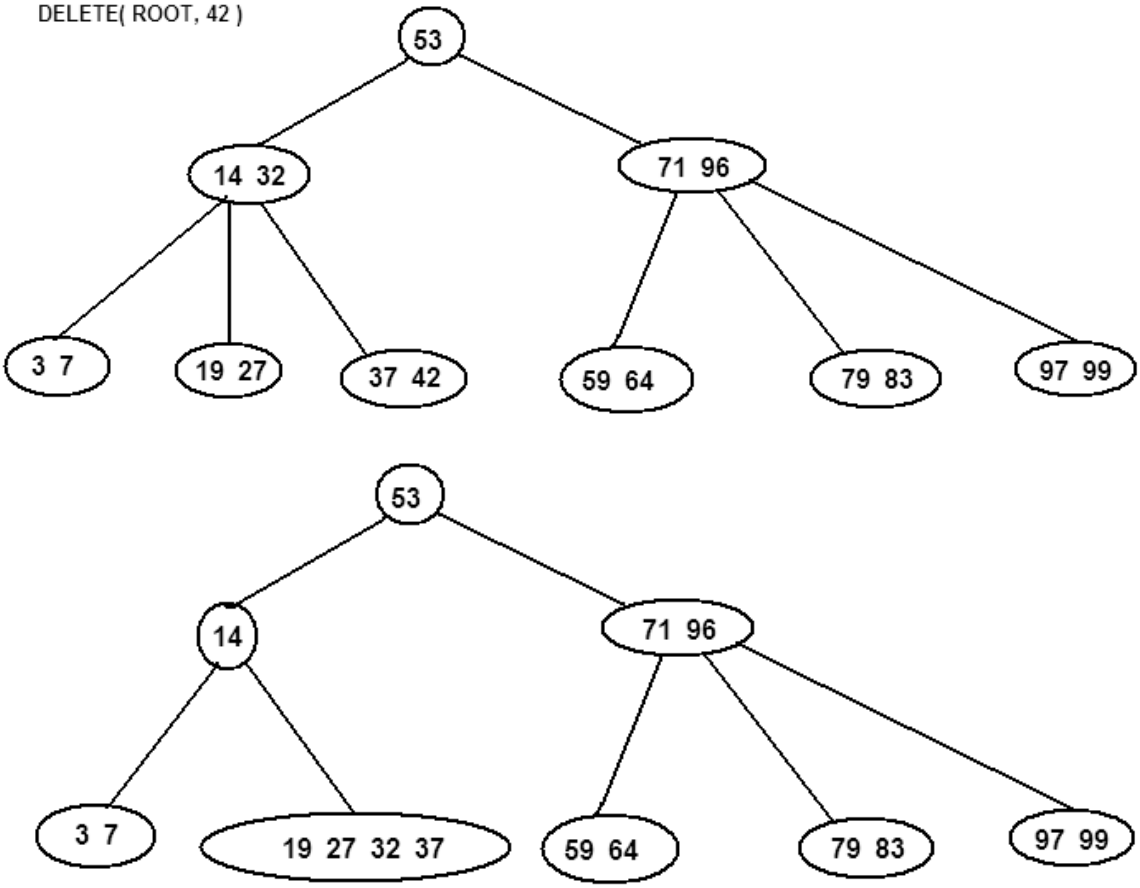


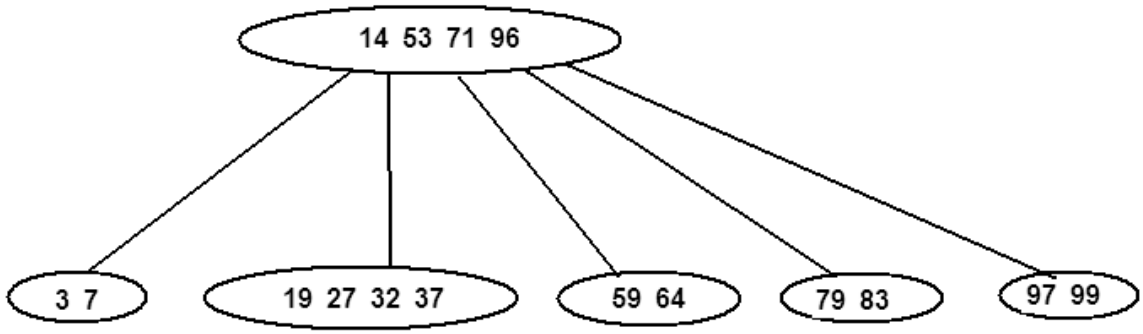
Case 2: In the second case, after the deletion the node has less than minimum n/2 values. Let us say we delete 92 from the tree. After 92 is deleted, the node has only one value 83. But a node adjacent to it consist 3 values (i.e., there are extra values in the adjacent node). Then the last value in that node 71 is pushed to its parent and the first value in the parent namely 79 is pushed into the node which has values less than minimum limit. Now the node has obtained the minimum required values.





Case 3: In the previous case, there was an adjacent node with extra elements and hence the adjustment was made easily. But if all the nodes have exactly the minimum required, and if now a value is deleted from a node in this, then no value can be borrowed from any of the adjacent nodes. Hence as before a value from the parent is pushed into the node (in this case 32 is pushed down). Then the nodes are merged together. But we see that the parent node has insufficient number of values. Hence same process of merging takes place recursively till the entire tree is adjusted.





Algorithm

```
DELETE( ROOT, K )

Temp = SEARCH( ROOT, k ), DELETED  = 0, i = 1
While i < = count(temp)
        If (k = info(temp[i])
                DELETED = 1
                Delete temp[i]
        End if
End while

If DELETED = 0
        Print "Item not found"
        Return
Else
        If count(temp) < n / 2
                i = 1
                While i <= count(par)
                        If count(child[i](par)) > n/2
                                s = child[i](par)
                                break
                        Else
                                i = i + 1
                        End if
                End while
                If info(temp[1]) > info(s[count(s)])
                        Ins(temp, info(par[1]))
                        Ins(par, info(s[count(s)]))
                Else
                        Ins(temp, info(par[count(par)]))
                        Ins(par, info(s[1]))
                End if
        End if
End if
End DELETE
```

Splay trees

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) time.

Tree Splaying

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.

Advantages:

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly.

Tree rotations

To bring the recently accessed node closer to the tree *root*, a splay tree uses tree rotations. There are six types of tree rotations, three of which are symmetric to the other three. These are as follows:

- Left and right rotations
- Zig-zig left-left and zig-zig right-right rotations
- Zig-zag left-right and zig-zag right-left rotations

The first type of rotations, either left or right, is always a *terminal rotation*. In other words, the splaying is complete when we finish a left or right rotation.

Deciding which rotation to use

The decision to choose one of the above rotations depends on three things:

- Does the node we are trying to rotate have a grand-parent?
- Is the node left or right child of the parent?
- Is the parent left or right child of the grand-parent?

If the node does not have a grand-parent, we carry out a left rotation if it is the right child of the parent; otherwise, we carry out a right rotation.

If the node has a grand-parent, we have four cases to choose from:

If node is left of parent and parent is left of grand-parent, we do a *zig-zig right-right* rotation.

If node is left of parent but parent is right of grand-parent, we do a *zig-zag right-left* rotation.

If node is right of parent and parent is right of grand-parent, we do a *zig-zig left-left* rotation.

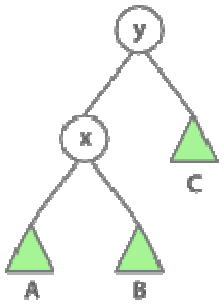
Finally, if node is right of parent but parent is left or grand-parent, we do a *zig-zag left-right* rotation.

The actual rotations are described in the following sections.

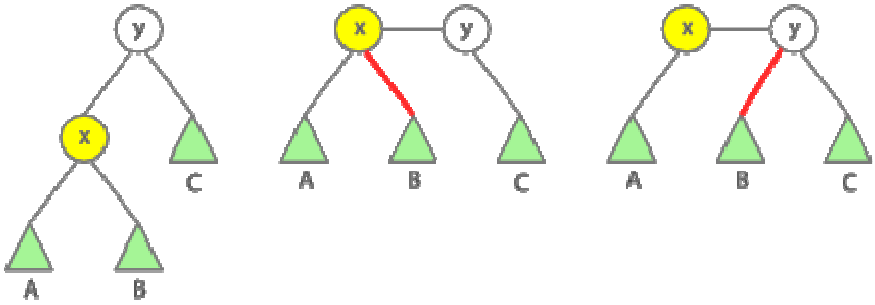
Left and right rotations

The following shows the intermediate steps in understanding a right rotation. The left rotation is symmetric to this.

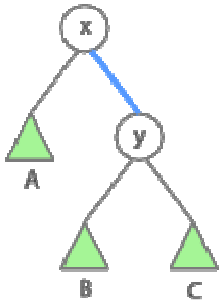
Splay x (rotate)



Right rotate x



Switch parent-child

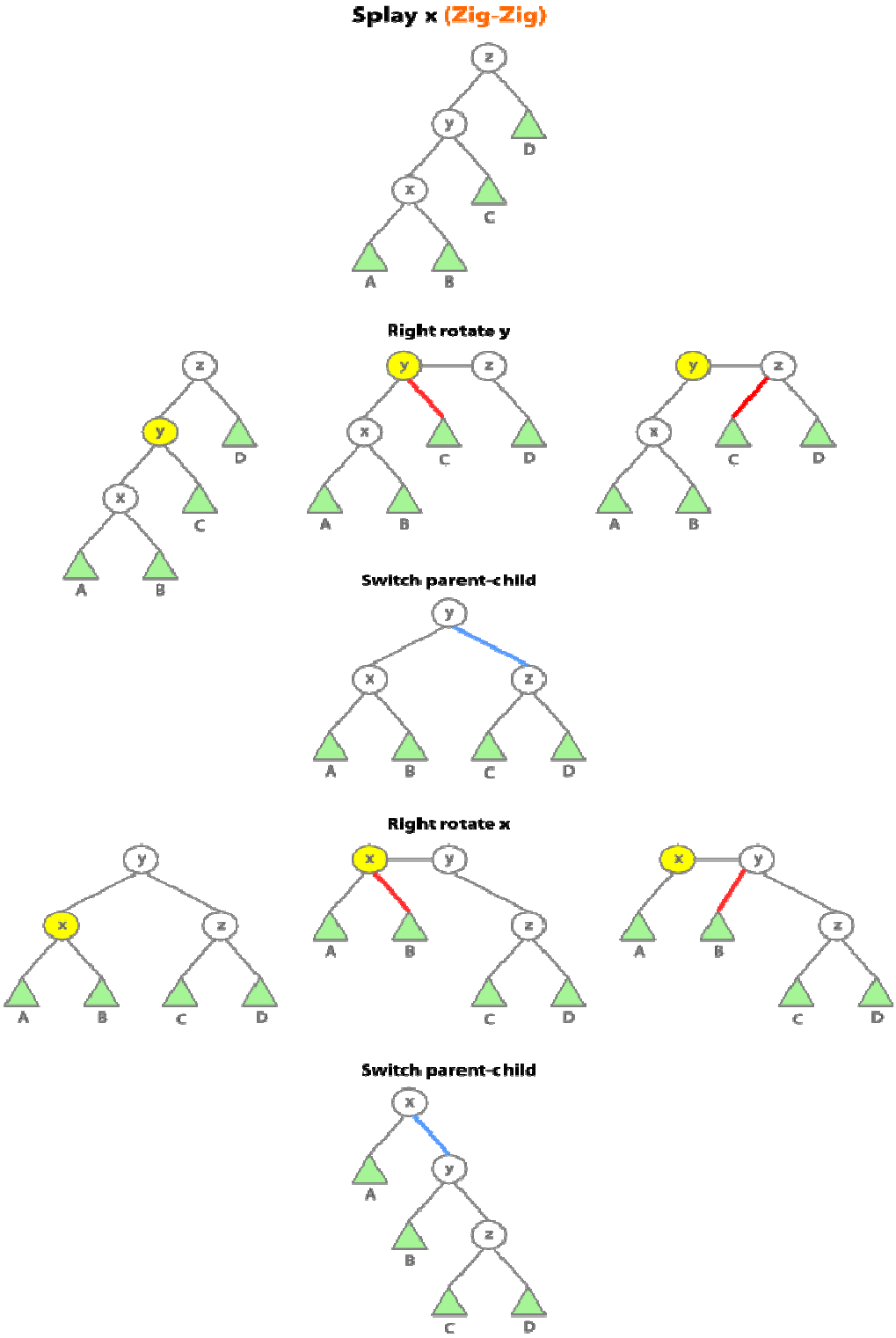


As we can see, each of the left or right rotations requires **five** pointer updates:

```
if (current == parent->left) {
   /* right rotate */
   parent->left = current->right;
   if (current->right)
      current->right->parent = parent;
   parent->parent = current;
   current->right = parent;
} else {
   /* left rotate */
   parent->right = current->left;
   if (current->left)
      current->left->parent = parent;
   parent->parent = current;
   current->left = parent;
}
current->parent = 0;
```

Zig-zig right-right and left-left rotations

The following shows the intermediate steps in understanding a zig-zig right-right rotation. The zig-zig left-left rotation is symmetric to this. Note in the following that with zig-zig rotations, we first do a right or left rotation on the **parent**, before doing a right or left rotation on the **node**.



As we can see, zig-zig right-right rotation requires **nine** pointer updates.

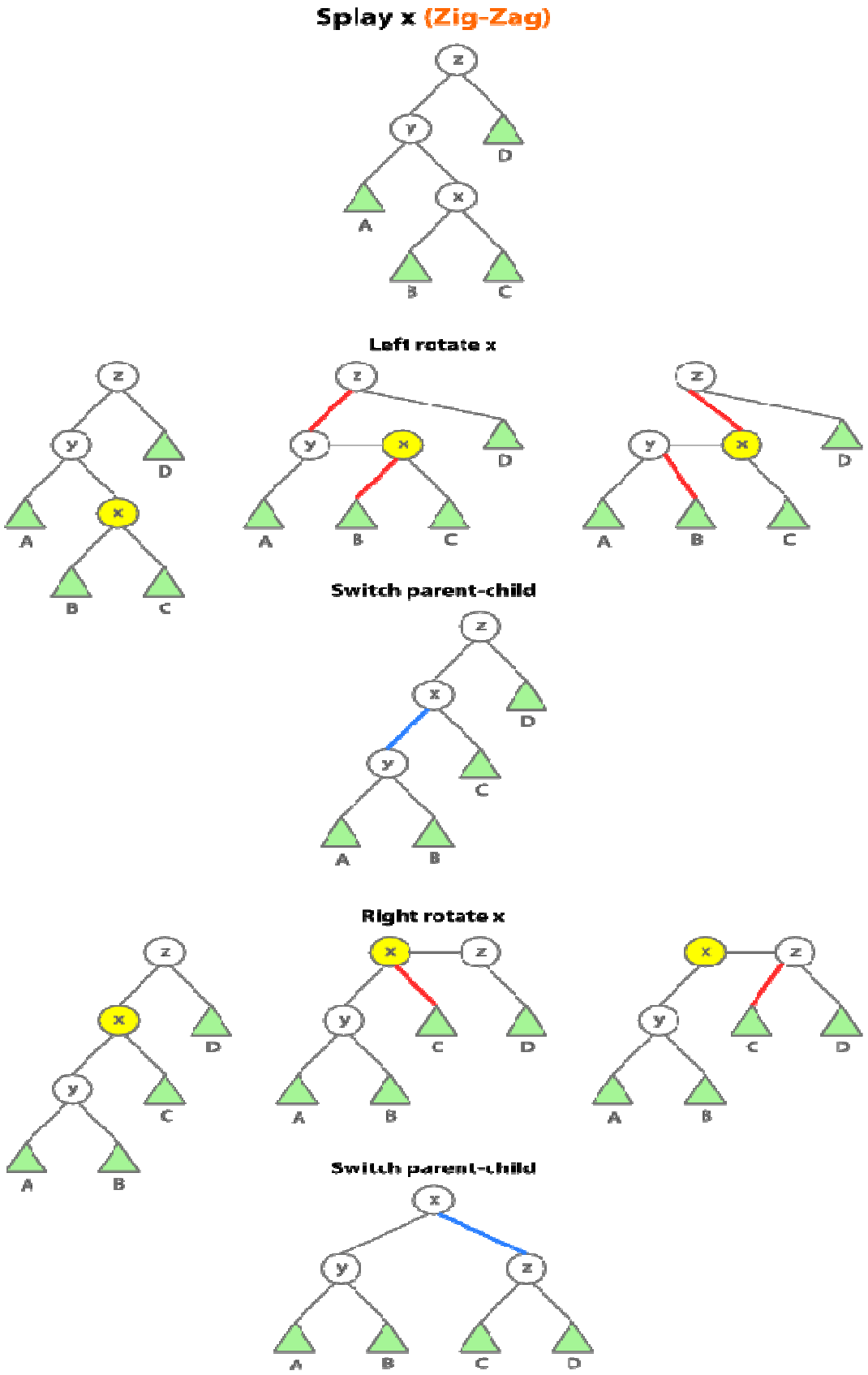
```
/* zig-zig right-right rotations */
if (current->right)
   current->right->parent = parent;
if (parent->right)
   parent->right->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = parent;
parent->parent = current;
grandParent->left = parent->right;
parent->right = grandParent;
parent->left = current->right;
current->right = parent;
```

The same number of pointer updates for zig-zig left-left rotation.

```
/* zig-zig left-left rotations */
if (current->left)
   current->left->parent = parent;
if (parent->left)
   parent->left->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = parent;
parent->parent = current;
grandParent->right = parent->left;
parent->left = grandParent;
parent->right = current->left;
current->left = parent;
```

Zig-zag left-right and right-left rotations

The following shows the intermediate steps in understanding a zig-zag left-right rotation. The zig-zag right-left rotation is symmetric to this. Note in the following that with zig-zag rotations, we do both rotations on the node, in contrast to zig-zig rotations.



As we can see, zig-zag left-right rotation requires **nine** pointer updates.

/* zig-zag right-left rotations */

if (current->left)

```
    current->left->parent = grandParent;
if (current->right)
    current->right->parent = parent;
current->parent = grandParent->parent;
grandParent->parent = current;
parent->parent = current;
grandParent->right = current->left;
parent->left = current->right;
current->right = parent;
current->left = grandParent;
```

The same number of pointer updates for zig-zag right-left rotation.

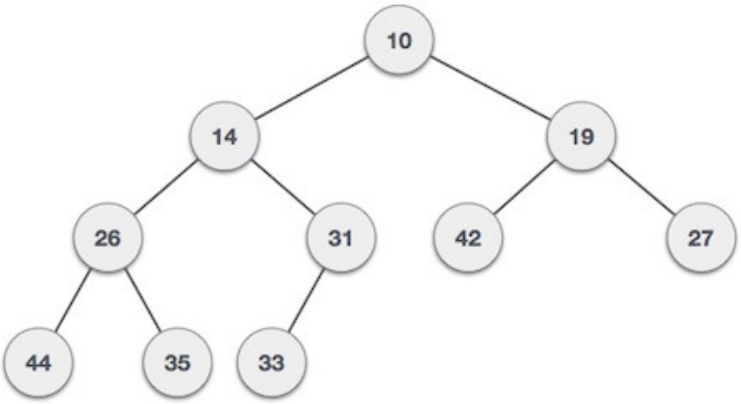
```
/* zig-zag left-right rotations */
if (current->left)
    current->left->parent = parent;
if (current->right)
    current->right->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = current;
parent->parent = current;
grandParent->left = current->right;
parent->right = current->left;
current->left = parent;
current->right = grandParent;
```

Heap Trees

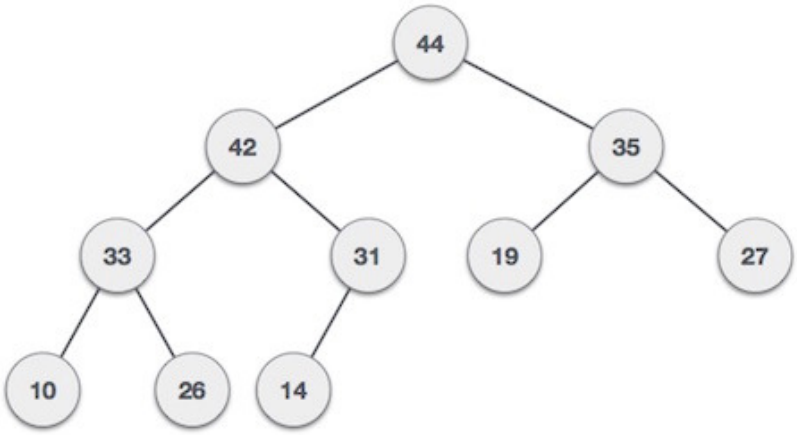
Heap is a special case of balanced binary tree data structure where root-node value is compared with its children and arranged accordingly. Heap trees are of two types- Max heap and Min heap.

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap – where the value of root node is less than or equal to either of its children.



Max-Heap – where the value of root node is greater than or equal to either of its children.



If a given node is in position I then the position of the left child and the right child can be calculated using **Left (L) = 2I** and **Right (R) = 2I + 1.** To check whether the right child exists or not, use the condition **R ≤ N.** If true, Right child exists otherwise not.The last node of the tree is **N/2.** After this position tree has only leaves.

Procedure HEAPIFY(A,N)
// A is the list of elements
//N is the number of elements
For (I = N/2 to 1)
    WALKDOWN (A,I,N)
END FOR
End Procedure
Procedure WALKDOWN(A, I,N)
//A is the list of unsorted elements
//N is the number of elements in the array
//I is the position of the node where the walkdown procedure is to be applied.

While I ≤ N/2
  L ← 2I, R ← 2I + 1

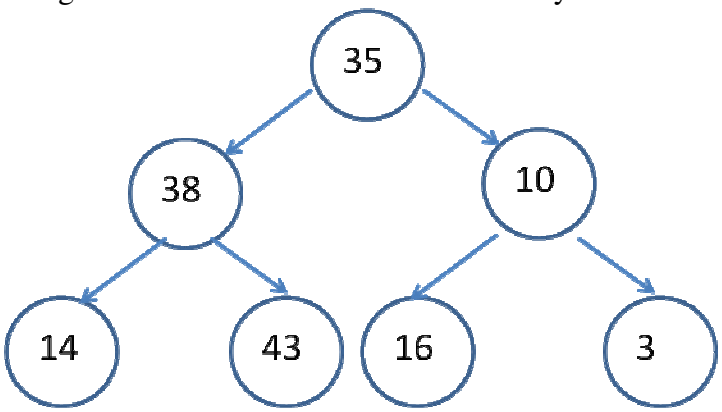
If A[L] > A[I] Then
        M ← L
Else

```
        M ← I
End If

If A[R] > A[M] and R ≤ N Then
        M ← R
End If
If M ≠ I Then
        A[I] ↔ A[M]
        I ←M

Else
        Return
End If
End While
End WALKDOWN
```

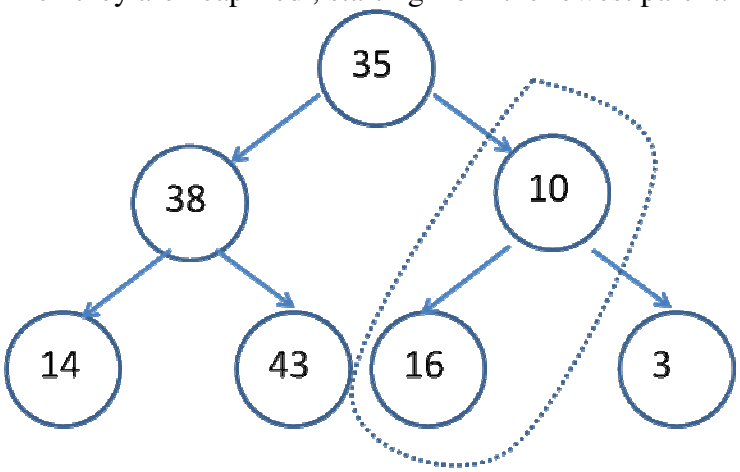
Example:
Given a list A with 8 elements:

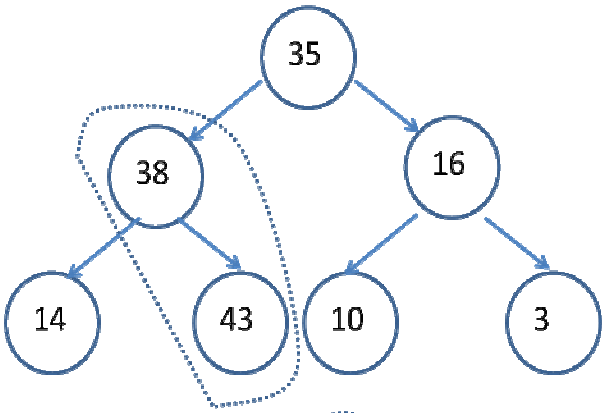
35	**38**	**10**	**14**	**43**	**16**	**3**

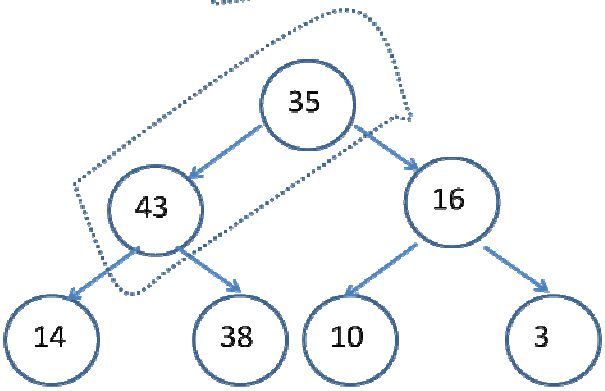
The given list is first converted into a binary tree as shown.

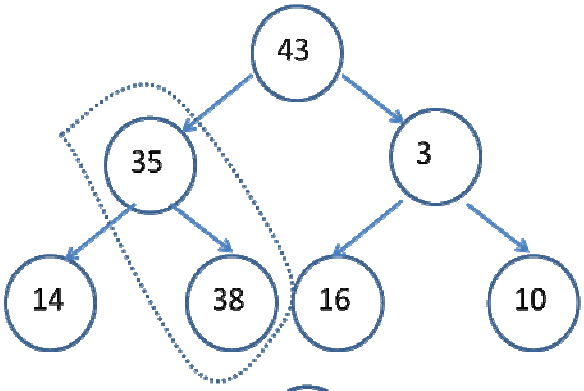


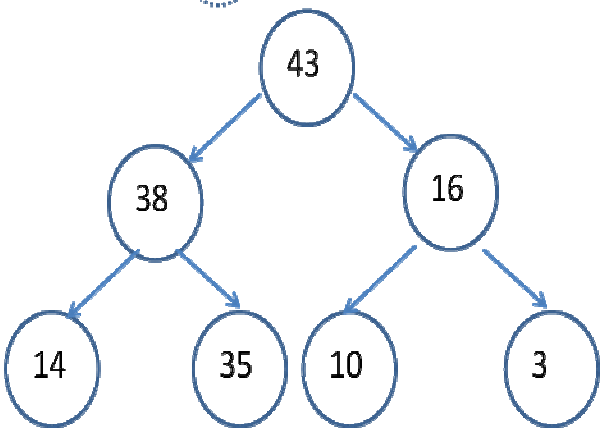
Then they are heapified , starting from the lowest parent.











The obtained tree is a Max heap tree.

TRIES

-Also called digital tree or radix tree or prefix tree.

-Tries are an excellent data structure for strings.

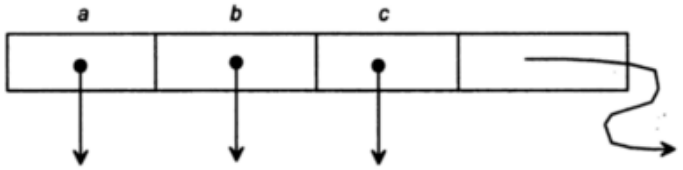
-Tries is a tree data structure used for storing collections of strings.

-Tries came from the word retrieval.

-Nodes store associative keys (strings) and values.

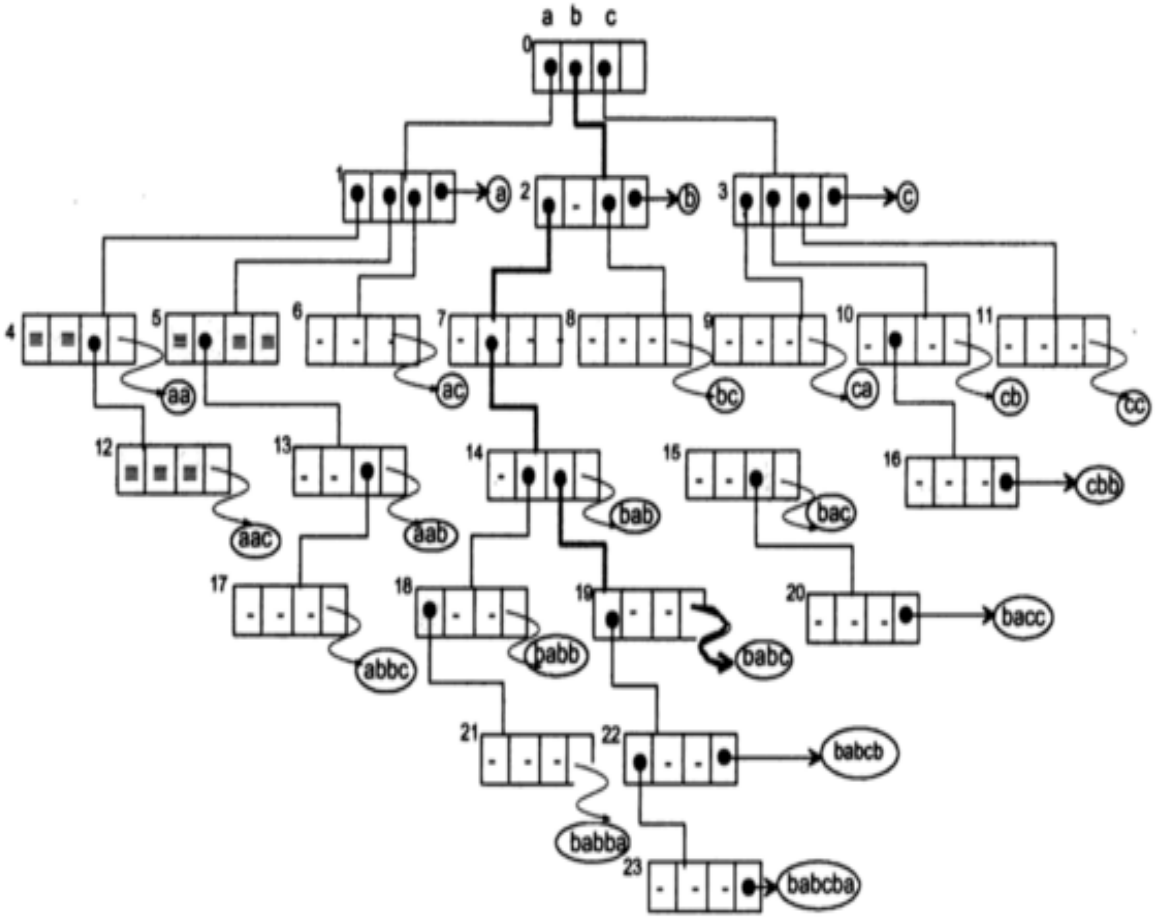
Tries Structure

Let us consider the case of a tries tree of order 3.Let the key value in this tries is constituted from three letters namely a, b and c. Each node has the following structure:




```
// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};
```

Here 3 link fields' points to three nodes in the next level and the last field are called the information field. The information field has the value either TRUE or FALSE. If the value is TRUE then traversing from the root node to this node yields some information. A tries of order 3 is given below:



For example, let us assume the key value 'bab'. Starting from the root node, and branching based on each letter, traversal will be 0-2-7-14 and in node 14, the information field TRUE implies that 'bab' is a word.

NOTE:

- Tries indexing is suitable for maintaining variable sized key values.
- Actual key value is never stored but key values are implied through links.
- If English alphabets are used, then a trie of order 26 can maintain whole English dictionary.

Operations on Trie

Searching

Searching for a key begins at the root node, compare the characters and move down. The search can terminate due to end of string or lack of key in tries. In the former case, if the *value* field of last node is non-zero then the key exists in tries. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

PSEUDOCODE. The search algorithm involves the following steps:

1. For each character in the string, see if there is a child node with that character as the content.

2. If that character does not exist, return false.
3. If that character exist, repeat step 1.
4. Do the above steps until the end of string is reached.
5. When end of string is reached and if the marker (NotLeaf) of the current Node is set to false, return true, else return false.

Insertion

Inserting a key into trie is simple approach. Every character of input key is inserted as an individual trie node. Note that the children are an array of pointers to next level trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth.

PSEUDOCODE: Any insertion would ideally be following the below algorithm:

1. Find the place of the item by following bits.
2. If there is nothing, just insert the item there as a leaf node.
3. If there is something on the leaf node, it becomes a new internal node. Build a new sub tree to that inner node depending how the item to be inserted and the item that was in the leaf node differs.
4. Create new leaf nodes where you store the item that was to be inserted and the item that was originally in the leaf node.

Deletion

Deletion procedure is same as searching and insertion with some modification. To delete a key from a trie, trace down the path corresponding to the key to be deleted, and when we reach the appropriate node, set the TAG field of this node as FALSE. If all the field entries of this node are NULL, then return this node to the pool of free storage. To do so, maintain a stack of PATH to store all the pointers of nodes on the path from the root to the last node reached.

Application of Tries

- Retrieval operation of lexicographic words in a dictionary.
- Word processing packages to support the spelling check.
- Useful for storing a predictive text for auto complete.