Heap Tree/Sort

A heap is based on the notion of a **complete binary tree,** formally a binary tree is **completely full** if it is of height, h, and has 2^{h+1}-1 nodes. A binary tree of height, h, is complete iff

(a) it is empty or
(b) its left subtree is complete of height h-1 and its right subtree is completely full of height h-2 or
(c) its left subtree is completely full of height h-1 and its right subtree is complete of height h-1.

A binary tree has the **heap property**iff

(a) it is empty or
(b) the key in the root is larger than that in either child and both subtrees have the heap property.

A binary tree has the **heap property**iff

(a) it is empty or
(b) the key in the root is larger than that in either child and both subtrees have the heap property. This type of heap is called max heap. A heap is an ordered balanced binary tree (complete binary tree) in which the value of the node at the root of any sub-tree is greater than or equal to the value of either of its children.

N_j (Node Key value/Subtree key value) $\leq N_i$ (Root node Key value) for $2 \leq j \leq n$ and $i = \lfloor j / 2 \rfloor$ This is max heap condition.

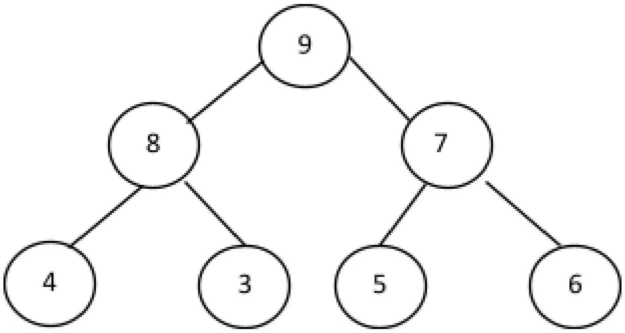


Figure 1. Max Heap Binary Tree

(c) the key in the root is smaller than that in either child and both subtrees have the heap property. This type of heap is called min heap. A heap is an ordered balanced binary tree (complete binary tree) in which the value of the node at the root of any sub-tree is less than or equal to the value of either of its children.

N_j (Node Key Subtree key value) $\geq N_i$ (Root node Key) for $2 \leq j \leq n$ and $i = \lfloor j / 2 \rfloor$ This is min heap condition.

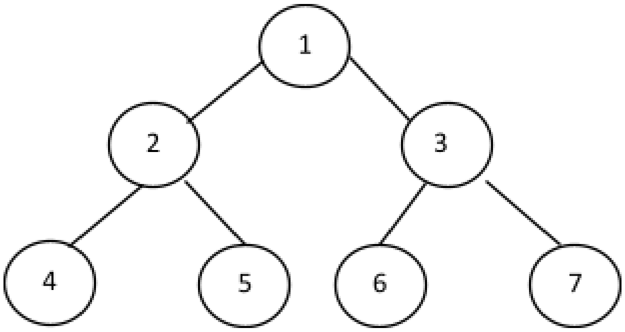
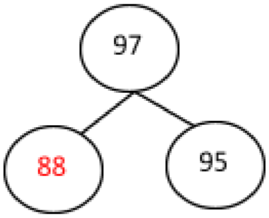


Figure 2. Min Heap

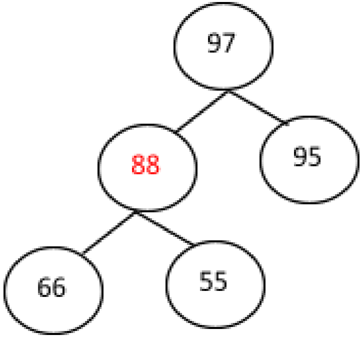
In a balanced tree the number of links traversed between the root of the tree and any leaf node will differ by only one. The binary tree is allocated sequentially such that the indices of the left and right children (if exist) of elements **i** are **2i** and **2i + 1,** respectively. Similarly, the index of the parent of element **i** (if exists) is **i / 2**. In other words, let the root of any sub-tree be at position **i**in the list then left child will be at position **2i** and the right child will be at position **2i + 1**. Use this property and construct a Binary Heap Tree from the following data list.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
97	88	95	66	55	95	48	60	45	48	55	62	77	25	38	18	40	38

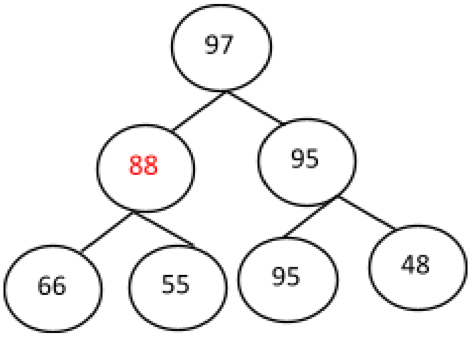
Step 1: i=1 then 2i (left) = 2 and 2i+1 (right) =3



Step 2: 2i (root) = 2 and 2i*2(left) =2*2=4 and 2i*2+1 (right)=5



Step 3: 2i +1(root) = 3 and (2i+1)*2(left) =6 and (2i+1)*2+1 (right)=5



If continue this process we will get the final complete binary tree as shown in Figure 3.

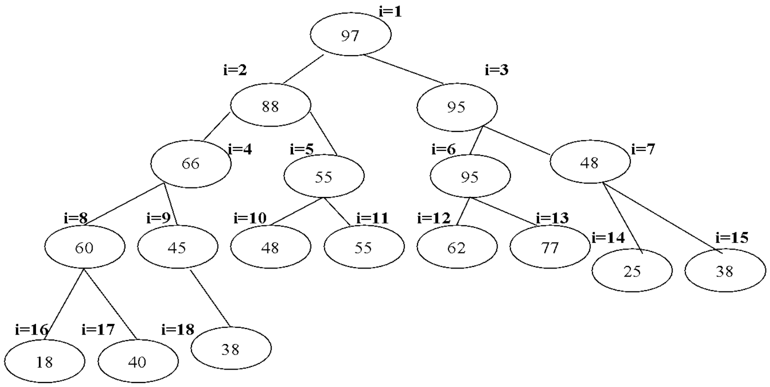


Figure 3. Binary Heap

Implementation of Priority Queue Using Binary Heap Tree (Assignment)

One important variation of the queue is the **priority queue**. A priority queue acts like a queue in that items remain in it for some time before being dequeued. However, in a priority queue the logical order of items inside a queue is determined by their “priority”. Specifically, the highest priority items are retrieved from the queue ahead of lower priority items. A couple of easy ways to implement a priority queue using sorting functions and arrays or lists.

The classic way to implement a priority queue is using a data structure called a **binary heap**. A binary heap will allow us to enqueue or dequeue items in $O(\log n)$. The binary heap has two common variations: the **min heap**, in which the smallest key is always at the front, and the **max heap**, in which the largest key value is always at the front.

Binary Search Tree (BST)

The value stored at a node is greater than the value stored at its left child and less than the value stored at its right child.

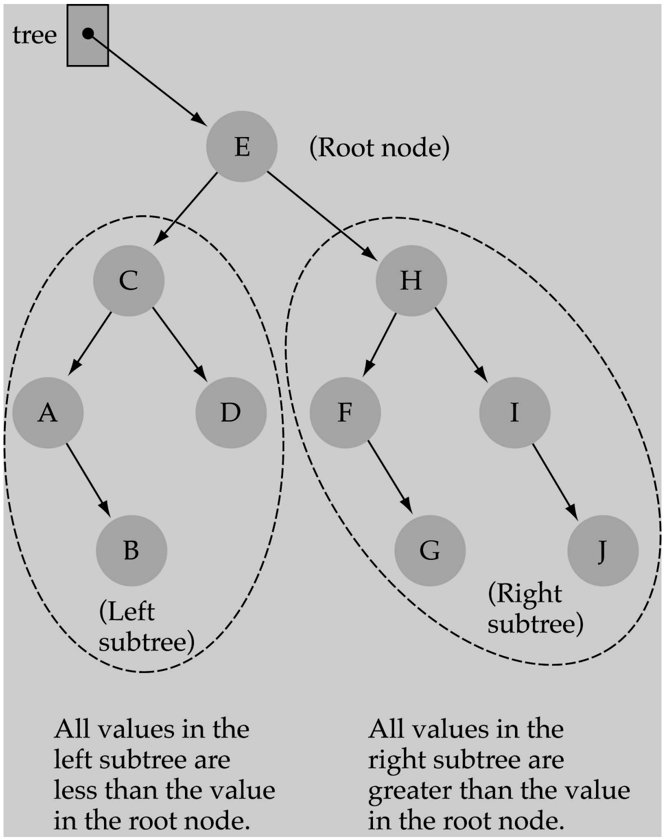

tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

Figure 1.BST or Not

Suppose we have a list L and in this list we are searching for a value X. If L has no special properties, if it might be any old list, then there is no better way to search it than to start at the beginning and go through the list one step at a time comparing each element to X in turn. This is called linear search - the time it takes (on average, and in the worst case) is linear, or O(n), in the length of the list. But if L is a **sorted** list, there is a much faster way to search for X:

1. Compare X to the middle value (M) in L.
2. if X = M we are done.
3. if X < M we continue search, but we can confine search to the first half of L and entirely ignore the second half of L.
4. if X > M we continue, but confine ourselves to the second half of L.

For example, consider the list, L = 1, 3, 4, **6**, 8, 9, 11 and search for X = 4. The size of list n=7. Now middle index value=n/2= 7/2=3.5=>4. Here 6 is considered as middle element. And x<6, i.e., 4<6 means we take first half of list to search for x. the first half of list is 1, 3, 4. Now find middle element here size of sublist is 3. Middle index value is 3/2=1.5=>2. Now value at index 2 is 3 and this 3<4 means we take up send half of the sublist. The second half of sublist is 4. Size of sublist is 1. No more division is possible. Here 4=4.

This is called Binary Search,each iteration of (1)-(4) the length of the list we are looking in gets cut in half. Therefore, the total number of iterations cannot be greater than $\log_2 n$. The difference between $O(\log_2 n)$ and O(n) is extremely significant when n is large: for any practical problem it is crucial that we avoid O(n) searches.

Binary search works perfectly if lists are implemented with arrays. But, there is a problem with linked list implementations. With a linked implementation we cannot find the middle of the list in constant time: it takes linear time and this takes away all the speedup we get from binary search. We need linked implementations for their memory efficiency and flexibility, but we cannot afford to do linear search. Let us design a linked structure specifically for doing binary search very efficiently. It will contain exactly the pointers needed for binary search. What is the first thing we need? A pointer to the middle element.

Using our previous example (L→ 6), what should the node ‘6’ point to? Well, when we compare X to 6 there are two possible outcomes that involve further search: X < 6 and X > 6. In the first case we need a pointer from ‘6’ to the middle of the front/first half of L, i.e., from ‘6’ to ‘3’. In the second case we need a pointer from ‘6’ to the middle of the second/last half of L, i.e., to ‘9’. Blue marked in L = 1, 3, 4, **6**, 8, 9, 11.

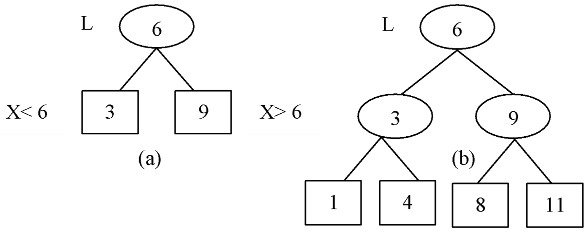


Figure 2. Binary Search Tree

What pointers do we need emanating out of ‘3’? Again there are two, one for each possible outcome of comparing X to 3 that requires further search. If X < 3 we need a pointer to the middle of the portion of L less than 3. If X > 3 we need a pointer to be careful here this pointer should not be to the middle of the portion of L bigger than 3. We have already eliminated all the elements bigger than or equal to 6. So, this pointer should point to the middle of the portion of the list between 3 and 6, similarly for 9. As we can see this structure is a binary tree (Figure 2). But is not an arbitrary binary tree; it has two special properties:

1. Every element in a node’s left subtree is smaller than the node’s value
2. Every element in a node’s right subtree is bigger than the node’s value

These hold for every node in the tree. A binary tree with these properties is called a Binary Search Tree. In other words we can say that a binary search tree is a binary tree that is either empty or in which each node possesses a key that satisfies the following conditions:

1. All keys (if any) in the left sub-tree of the root precede the key in the root.
2. The key in the root precedes all keys (if any) in its right sub-tree.
3. The left and right sub-trees of the root are again search trees.

To search for the target, we first compare it with the key at the root of the tree. If it is not same, we go to either the left sub-tree or right sub-tree as appropriate and repeat the search in that sub-tree. What condition will terminate the execution? Clearly if we find the key, the function succeeds. If not, then we continue searching until we find an empty sub-tree.

In a binary search tree we can search for a value in O($\log_2$ n) time, where n is the number of nodes in the tree. The algorithm is exactly the binary search algorithm, but translated in terms of root and subtrees:

1. Compare item to be searched is X compare with key at the root (i.e., middle key/value) (M) in L (List of elements or keys).
2. If X = M we are done.
3. If X < M we recursively search for X in L’s left subtree.
4. If X > M we recursively search for X in L’s right subtree.

In which subtree smallest key will be kept and where will be biggest key value in the tree?

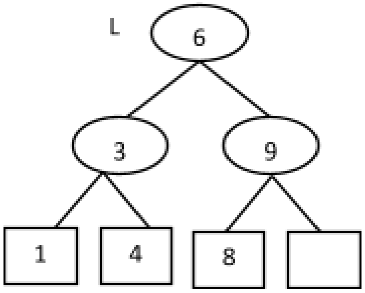


Figure 3. BST

The properties defining a BST, mean that it is very easy to traverse the tree in increasing/ascending order (in-order traversal algorithm can used) (or in decreasing order). What algorithm(s) can be applied for each type of traversal?

Assignment 5: Write algorithm to generate descending order list from a binary search tree(BST)

Insertion In Binary Search Tree (BST)

It is process/operation to add to an item/key into a BST without disturbing the property of binary search tree. **For example consider the list** L = 1, 3, 4, **6**, 8, 9, 11 and construct a BST for this list. If BST is empty then very fist element is added into the tree and that will be root of initial tree. Here 1 is added to T. now we go for 2nd element, i.e., 3 then 4 and so on.

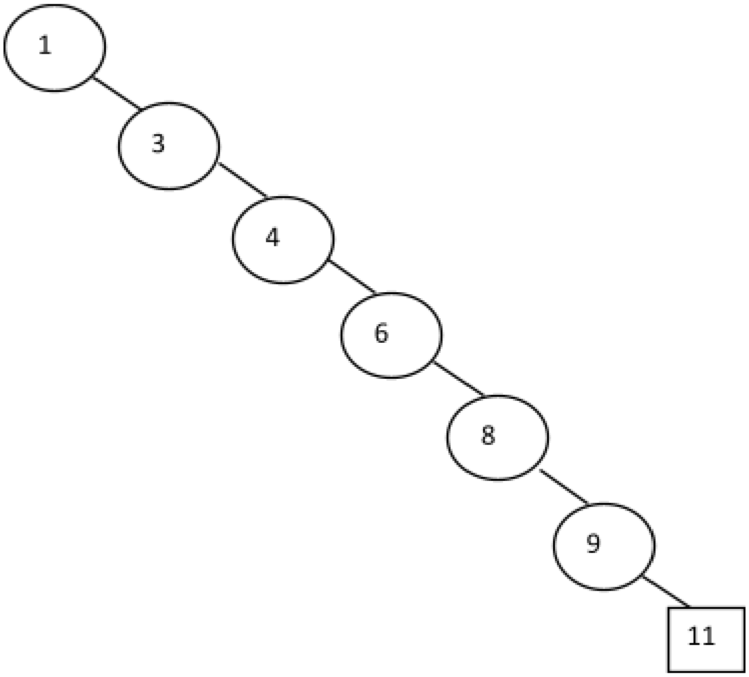


Figure 4.Right skewed Binary Search tree with height (n=7)

If we look into Figure 4 we find that here search time will be O(n) because it is right skewed binary tree. The above process to build a BST is not acceptable. We should always observe Binary Search Algorithm if we are building a BST from initial/scratch stage. Thus, we get the BST when applied Binary Search Algorithm for above data list.

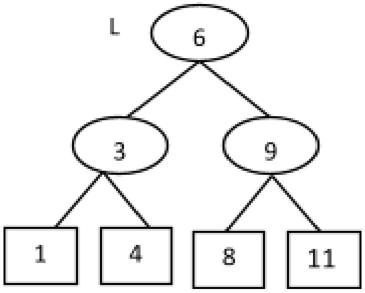
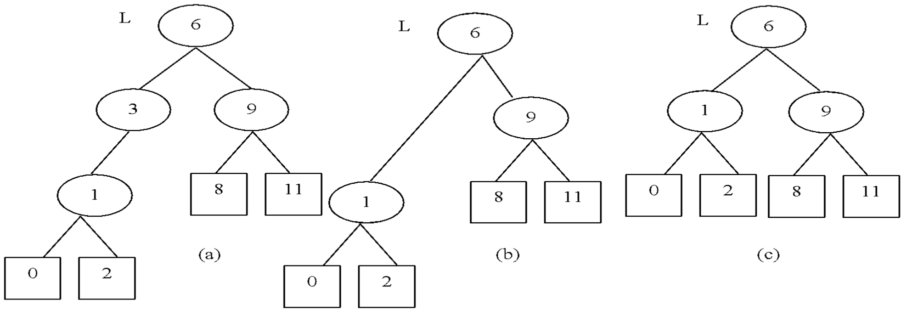


Figure 5. BST using Binary Search Algorithm

Now insert item 7 to the BST shown in Figure 5.

Deletion From a Binary Search Tree (BST)

If we are trying to delete a leaf, there is no problem. We just delete it and rest of the tree is exactly as it was, so it is still a BST. There is another simple situation: suppose the node we are deleting has only one subtree. In the three of **Figure7(a)**, '3' has only 1 subtree.



(a) Binary tree (b)After deletion of 3 form (c) Final resulting tree

Figure 7. Deletion in Binary Search Tree

Finally, let us consider the only remaining case: how to delete a node having two subtrees. For example, how to delete '6' {**Figure** 8(a)}? We had like to do this with minimum amount of work and disruption to the structure of the BST.

The standard solution is based on this idea: we leave the node containing '6' exactly where it is, but we get rid of the value 6 and find another value to store in the '6' node. This value is taken from a node below the '6's node, and it is that node that is actually removed from the tree. So, here is the plan. Starting with **Figure** 8(a). Erase 6, but keep its node is given in **Figure** 8(b).

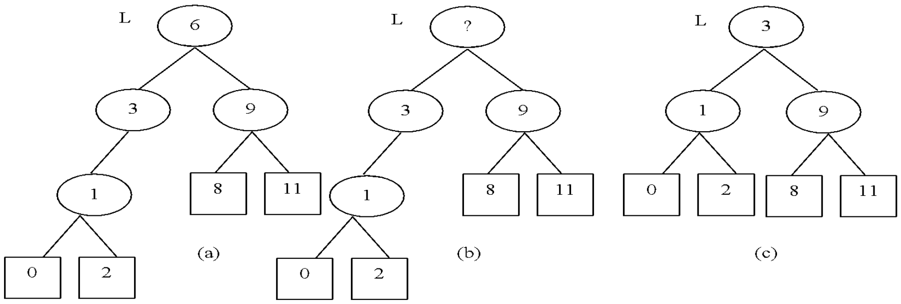


Figure 8 (a) Binary tree (b) Binary tree after removal of 6 (c) Binary tree after deleting a node having left and right subtree.

Now, what value can we move into the vacated node and have a binary search tree? Well, here’s how to figure it out. If we choose value X, then:

1. every element in the left subtree must be smaller than X.
2. every element in the right subtree must be bigger than X.

Binary Search Trees: A Problem

To finish off the discussion of binary search trees, we would like to return to the issue that originally motivated them - we wanted a data structure that could be searched in O($\log_2$ n) time, where n is the number of values in the data structure. Do BSTs actually meet this requirement? In other words, is it true that we can find any value X in any BST in O($\log_2$ n) time? What does one think? Yes, no?

It is does not fulfill all the above property.

AVL Tree

To address this issue Balanced Binary Tree/AVL Tree is invented.

Two Russian mathematicians, G. M. ADELSON-VELSKII **and** E.M. LANDIS**, gave a technique in 1962. It is feasible to balance the height of binary trees using this technique and the resulting tree is called** AVL **trees in their honor. Difference of height of left and right subtree of a node should not be more than 1 for a balanced binary search tree. It means following Binary tree does not contain the property of BST.**

With the help of **AVL** trees searching, insertions and deletions in a tree with **n** nodes achieved in time of order **O (n),** even in the worst case. The height of an **AVL** tree with **n** nodes can never exceed by **1.44 $\log_2$n** and thus the behavior of an **AVL** tree could not be much below that of a random binary search tree. The actual length of a search is near to **$\log_2$n** and thus the behavior of **AVL** trees are closed to ideal balanced binary tree (BBT).

A tree is called **AVL** tree (Balanced binary tree), if each node possesses one of the following properties:

1. A node is called left heavy, if the longest path in its left sub-tree is one longer than the longest path of its right sub-tree.
2. A node is called right heavy, if the longest path in the right sub-tree is one longer than the longest path in its left sub-tree.
3. A node is called balanced, if the longest paths in both the right and left sub-trees are equal. For example, **Figure 9(a)** illustrates a balanced binary tree.

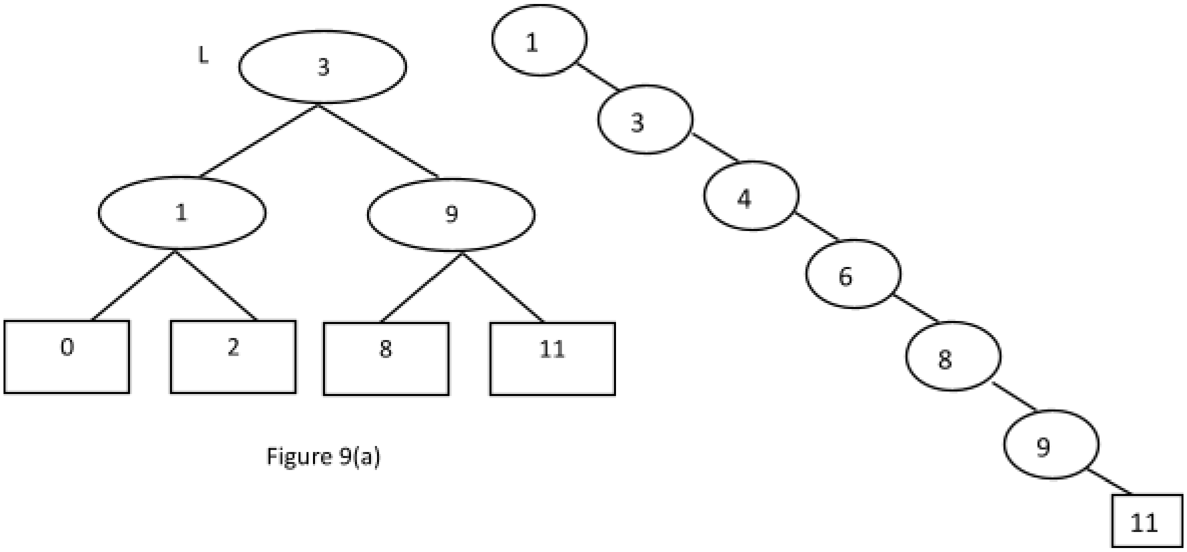


Figure 9(a)

Now it is required to insert 12 and 13 into the BST shown in Figure 9(a). What will get. If we insert these keys into this figure it violet AVL Tree conditions.

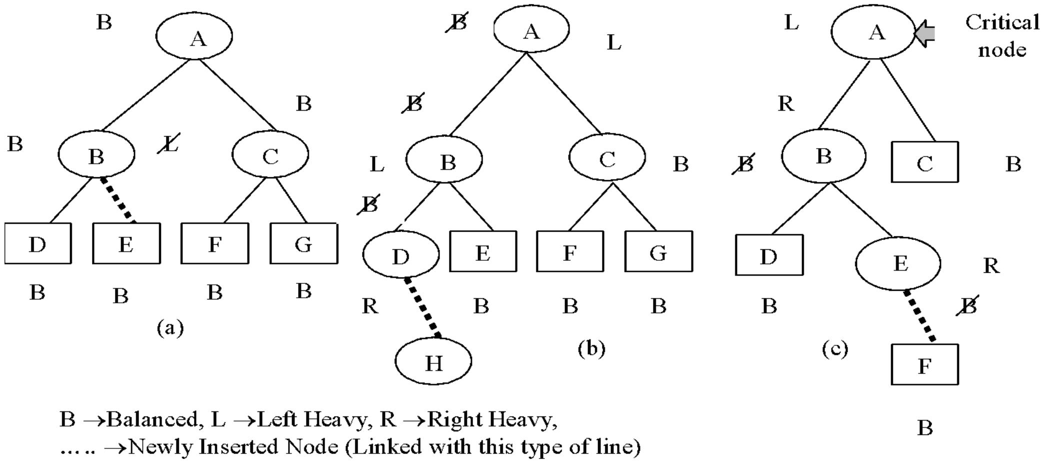
So, we required to find out solutions for these issues. These issues can be addressed with the help of balancing height of left and right subtrees of a node. To balance the height of left and right subtrees of a node, we required to find the balance factor of each node in a tree. **Balanced factor of a node can be defined as a difference of height of left and right subtrees.**If one inserts a new node into a balanced binary tree at the leaf, then the possible changes in the status of a node on the path (balanced indicator), which can occur are as follows:

Condition 1: The node was either left or right heavy and has now become balanced.

Condition 2: The node was balanced and has now become left or right heavy.

Condition 3: The node was heavy and the new node has been inserted in the heavy sub-tree, thus creating an unbalanced sub-tree, such a node is called a **critical node**.

Continued… AVL Tree



B
A
B
B
B
~~L~~
C
D
E
F
G
B
B
B
B
(a)
~~B~~
A
L
~~B~~
L
B
C
B
~~B~~
D
E
F
G
R
B
B
B
H
(b)
L
A
Critical node
R
~~B~~
B
C
B
D
E
R
B
~~B~~
(c)
F
B
B →Balanced, L →Left Heavy, R →Right Heavy,
….. →Newly Inserted Node (Linked with this type of line)

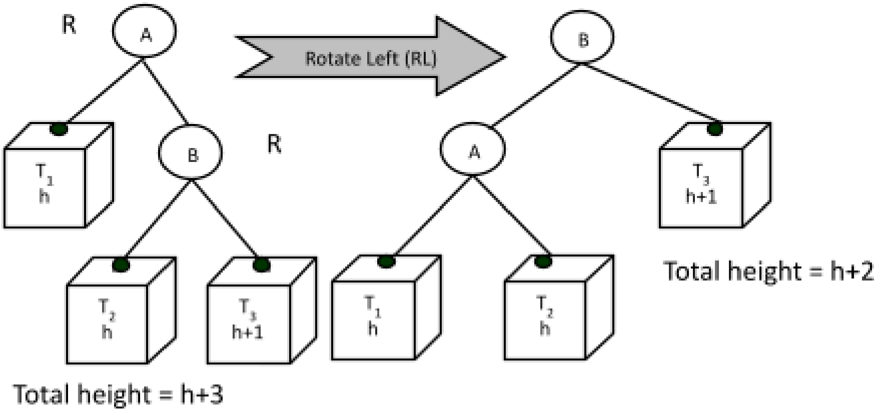
Figure 10.Balancing Binary Tree (a) Condition 1st (b) Condition 2nd (c) Condition 3rd

B (Balance Factor)= (Hieght of Left Subtree) L_h-(Hieght of Right Subtree) R_h

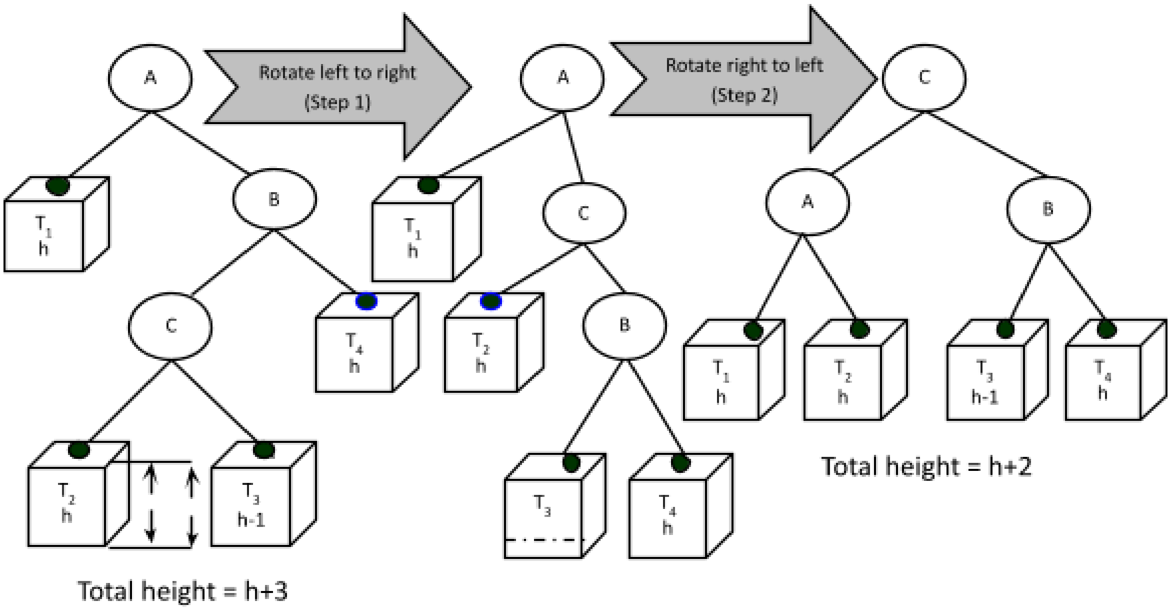
Balancing AVL Tree

When a tree does not satisfy the AVL requirements, it is required to rebuild the part of the tree to restore its balance. For example, assume if we insert a new node into the right sub-tree, its height will increase and the original tree will become right high.

Right High (height of the right sub-tree is greater than left sub-tree):If **B** is right high as illustrated in **Figure 11** then the action needed in this case is called a left rotation. We rotate the node **B** upward to the root, dropping **A** down into the left sub-tree of **B**; the sub-tree **T_2**of node's with keys between **A** and **B** now becomes the right sub-tree of **A** rather than the left sub-tree of **B**.



(a) **Restoring balance by left rotation**



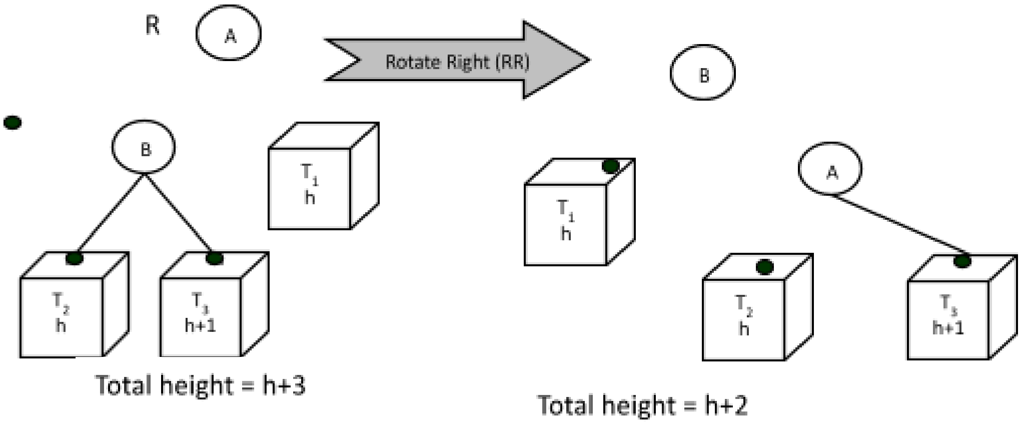
(b) Double Rotation

Figure 11. AVL Tree Balancing

Left High

Rotate Left Right

Rotate Left Right and Right Left (LR)



Operations in AVL Tree

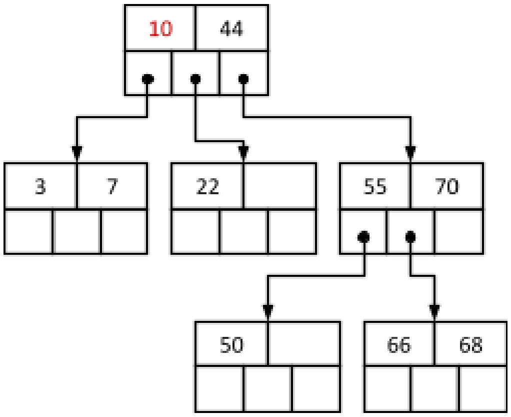
Insertion in AVL Tree

Deletion in AVL Tree

m-Way Search Trees

A binary search tree (BST) has one value in each node and twosubtrees. This notion easily generalizes to an m-way search tree, which has (m-1) values per node and m subtrees. Here m is called the degree of the tree. A BST, therefore, has degree 2. In fact, it is not necessary for every node to contain exactly (m-1) values and have exactly m subtrees. In an m-way tree a node can have anywhere from 1 to (m-1) values, and the number of (non-empty) subtrees can range from 0 (for a leaf) to 1+(the number of values). Thus, m is a fixed upper limit on how much data can be stored in a node.

The values in a node are stored in ascending order, $V_1 < V_2 < \ldots V_k$ ($k \leq m-1$) and the subtrees are placed between adjacent values, with one additional subtree at each end. We can thus associate with each value a 'left' and 'right' subtree, with the right subtree of V_i being the same as the left subtree of V_{i+1}. All the values in V_1's left subtree are less than V_1; all the values in V_k's right subtree are greater than V_k; and all the values in the subtree between V_i and V_{i+1} are greater than V_i and less than V_{i+1}



Here degree on tree is 3 and maximum number of keys stored in a node are 2.

Figure 1. 3-way Search tree

It will be convenient to illustrate m-way trees using a small value of m. But bear in mind that in practice m is usually very large. Each node corresponds to a physical block on disk, and m represents the maximum number of data items that can be stored in a single block. The m is maximized inorder to speedup processing: to move from one node to another involves reading a block from disk - a very slow operation compared to moving around a data structure stored in memory. The algorithm for searching for a value in an m-way search tree is the obvious generalization of the algorithm for searching in a binary search tree.

If we are searching for value X and currently at node consisting of values $V_1 \ldots V_k$, there are four possible cases that can arise:

1. If $X < V_1$, recursively search for X in V_1's left subtree.
2. If $X > V_k$, recursively search for X in V_k's right subtree.
3. If $X = V_i$, for some i, then we are done (X has been found).
4. The only remaining possibility is that, for some i, $V_i < X < V_{i+1}$. In this case recursively search for X in the subtree that is in between $V_i$and V_{i+1}.

If inserting (in BST) values in ascending order will result in a **degenerate** m-way search tree, i.e., a tree whose height is O(n) instead of O($\log_2$ n), if degree of tree is 2. This is a problem because all the important operations are O(height), and it is aim to make them O($\log_2$ n). One solution to this problem is to force the tree to be height-balanced as discussed. For example List: 1, 2, 3, 4, 5, 6, 7. A BST will be created.

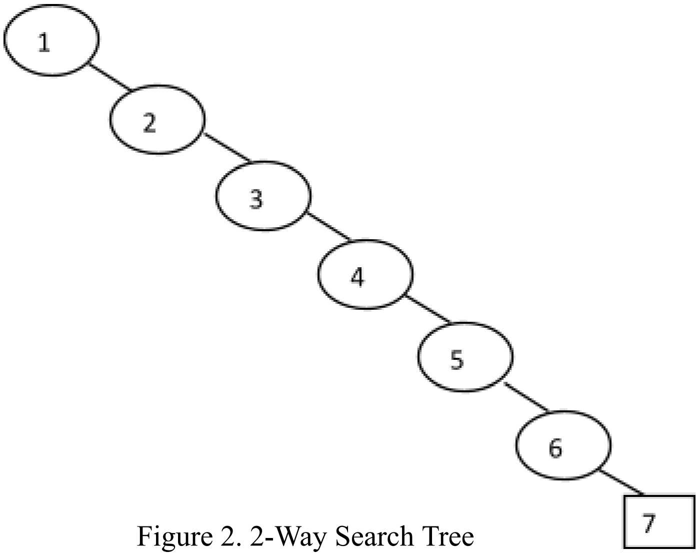


Figure 2. 2-Way Search Tree

What are the limitations of m-way search tree? (a) All leaf nodes are not being at the same level (b) since all the operations (insertion, deletion and search) are based on height of tree means these operations time may vary for different subtrees in the m-way tree.

Repeat the above example for m=3. We get 3-way search Tree as shown in Figure 3.

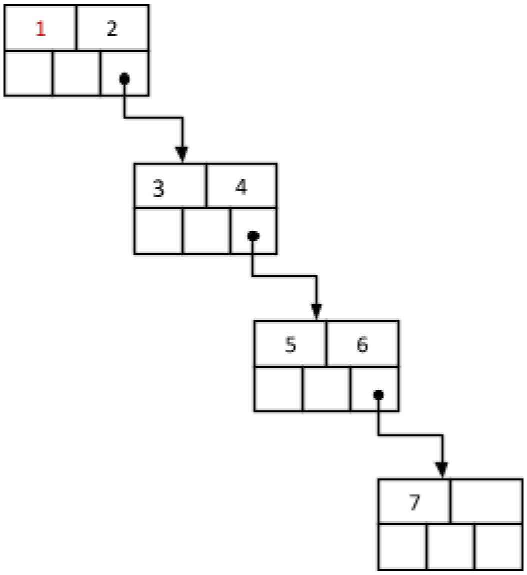


Figure 3.3-way Search Tree.

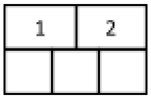
B-Tree

B-tree is solution for m-way search tree issue of height(degeneration property) and leaf nodes. A B-tree is an m-way search tree with the following special properties:

1. The root may have any number of values (from 1 to m-1) where m is the degree of the B-Tree.
2. It is perfectly balanced: every leaf node is at the same depth/level.
3. Every node, except the root is at least half-full, i.e., contains m/2 or more values but cannot contain more than m-1 values.

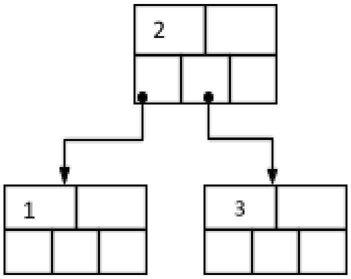
For example consider the List: 1, 2, 3, 4, 5, 6, 7 and construct/build a B-Tree of Degree 3. Here m=3 means number of keys/values in a node will be 2.

Step 1. Insert 1 and 2 into the tree we get.

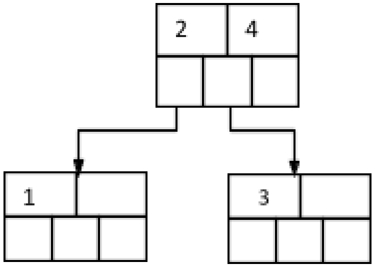


Step 2:

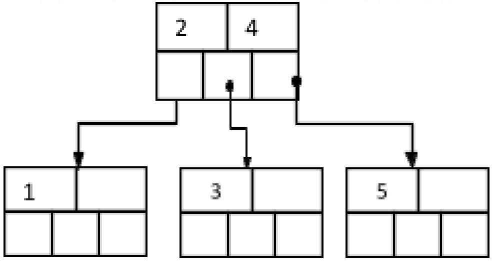
Take next value, i.e., 3. Since there is only 1 node in tree and there is also no room to accept 3 means splitting of node will be done. Central key/value of the node will be promoted to 1 level upward and two subtrees are produced.



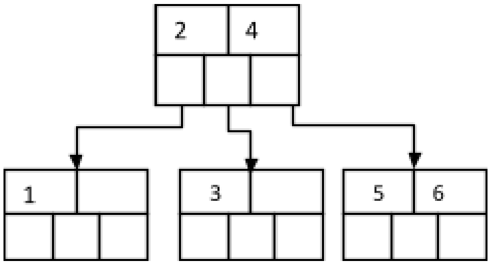
Step 3: Insert 4 into the tree. Since there is room in root then this will inserted into the root.



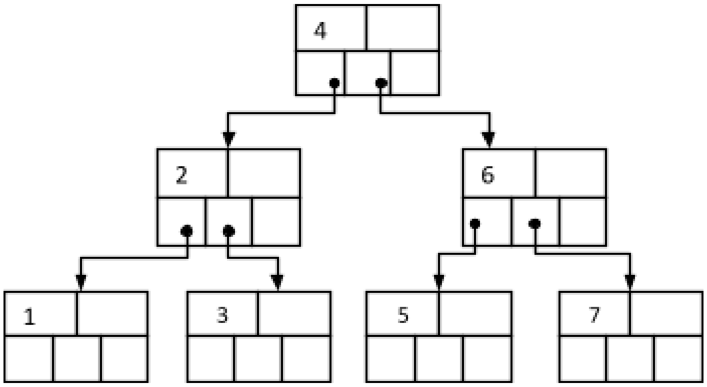
Step 4: Take 5 and insert it into the tree shown in Step 3.



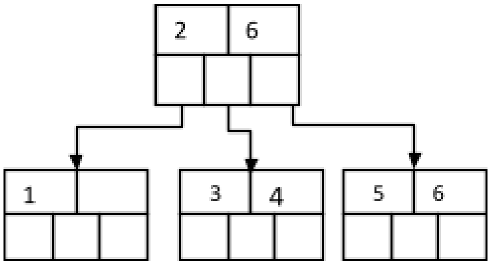
Step 5: Take 6 and insert it into the tree shown in Step 4.



Step 6: Take 7 and insert it into the tree shown in Step 5. Root is full right subtree of 4 is also full means 7 will be added in the right subtree node and this node will split and pivot/central value will be promoted upward. Since root is also full then root will split and pivot move upward and form new root. As per this statement the following B-Tree will be created.



Since left subtree of 4 and right subtree of 2 are not full/only half field. Means to adjust the height of the tree pivot of the root will be downgraded into the left subtree of newly added key into the root and result B-tree will be.



Insertion in B-Tree

To insert value X into a B-tree, there are 3 steps:

1. Using the search procedure for m-way trees find the leaf node to which X should be added.

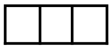
2. Add X to this node in the appropriate place among the values already there. Being a leaf node there are no subtrees to worry about.
3. If there are m-1 or fewer keys/values in the node after adding X, then we are finished. If there are m-1 (keys/values) and after adding X there will be m keys/values, we say the node has overflowed. To repair this, we split the node into three parts: for

Left: the first (m-1)/2 values

Middle: the middle value (position 1+((m-1)/2)

Right: the last (m-1)/2 values

For example consider the List: 1, 2, 3, 4, 5, 6, 7 and construct/build a B-Tree of Degree 3. Here m=3 means number of keys/values in a node will be 2. Starting the construction of B-tree of degree3 .first we insert 1 and 2 in the tree after that Insertion 3. Insertion of 3 violate the property of B-tree.As per the steps mentioned above we split the node into three parts. Same is discussed in the example Above.



Note that Left and Right have subtrees have just enough values to be made into individual nodes. That's what we do, they become the left and right children of Middle element, which we add in the appropriate place in this node's parent node. But what if there is no room in the parent? If it overflows we do the same thing again: split it into Left-Middle-Right, make Left and Right into new nodes and add Middle (with Left and Right as its children) to the node above. We continue doing this until no overflow occurs, or until the root itself overflows. If the root overflows, we split it, as usual, and create a new root node with Middle as its only value and Left and Right as its children (as usual).

For example, let us do a sequence of insertions into B-tree **Figure**2 (m=5, so each node other than the root must contain between 2 and 4 values).

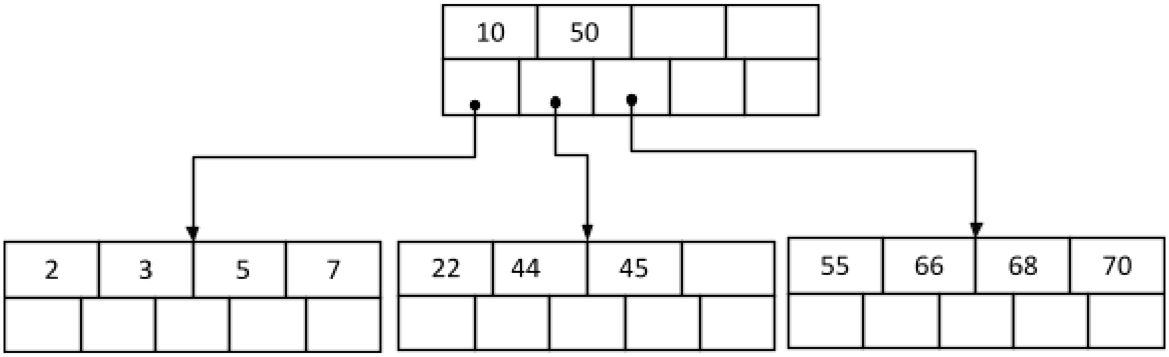


Figure 25-way B-tree

Insert 17: Add it to the middle leaf {**Figure 2**}. No overflow, so we are done {**Figure 3**}.

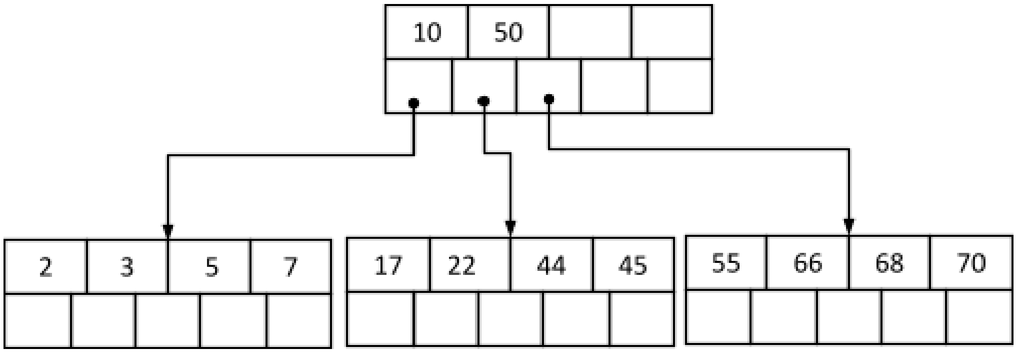


Figure 3. Insertion of 17 in 5-way B-tree

Insert 6: Add it to the leftmost leaf {**Figure 3**}. That overflows, so we split it and resulting tree is in **Figure 3.**

- Left = [2 3]
- Middle = 5
- Right = [6 7]

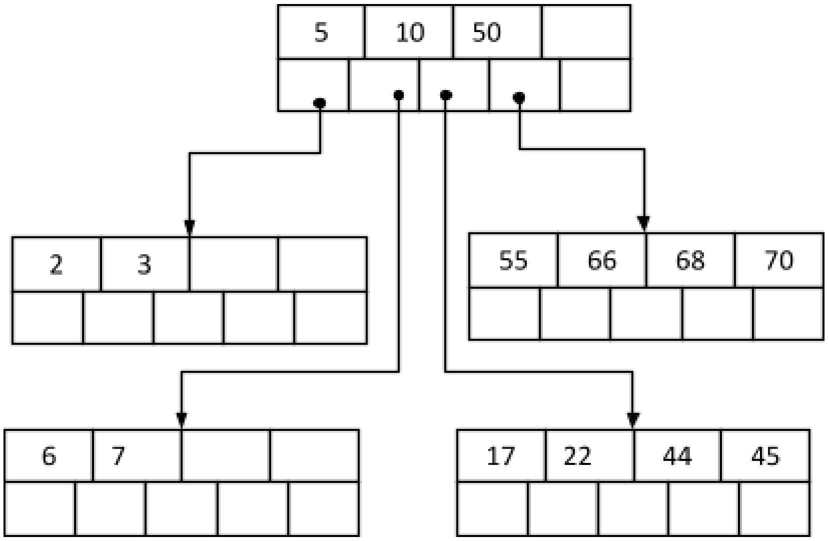


Figure 3 Insertion in 5-way B-tree

Left and Right become nodes; Middle is added to the node above with Left and Right as its children. The node above (the root in this small example) does not overflow, so we are done.

Insert 21: Add it to the middle leaf {**Figure 3**}. That overflows, so we split it the resulting tree is in **Figure 4.**

- left = [17 21]
- Middle = 22
- Right = [44 45]

Left and Right become nodes; Middle is added to the node above with Left and Right as its children. The node above (the root in this small example) does not overflow, so we are done.

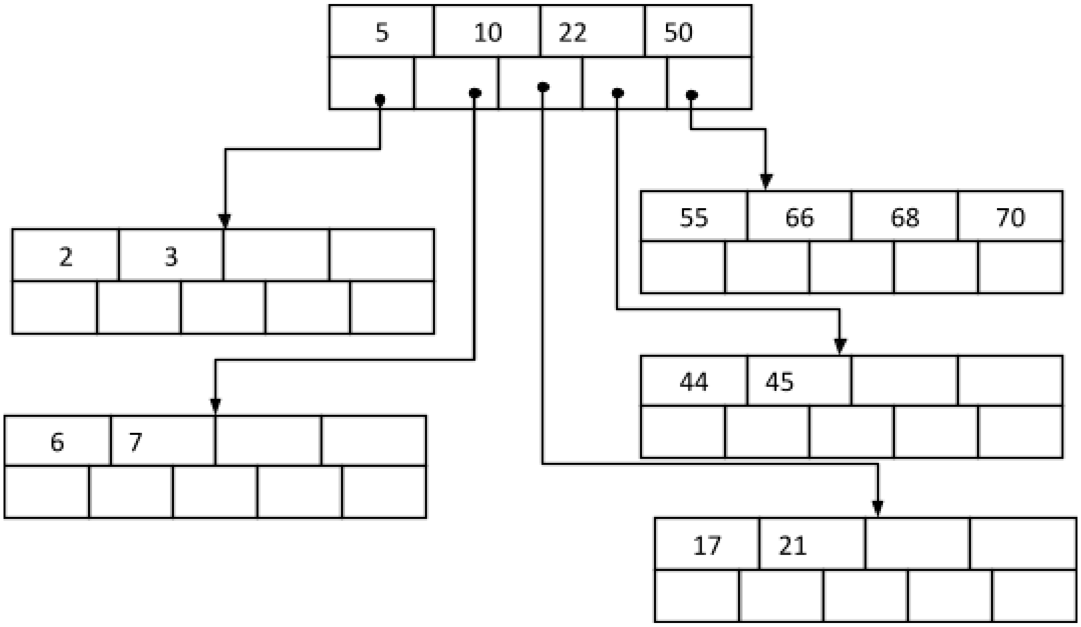


Figure 4.5-way B-tree after insertion of 21 in Figure 3

Insert 67: Add it to the rightmost leaf {**Figure 4**}. That overflows, so we split it:

- Left = [55 66]
- Middle = 67
- Right = [68 70]

Left and Right become nodes; Middle is added to the node above with Left and Right as its children {**Figure**5}.

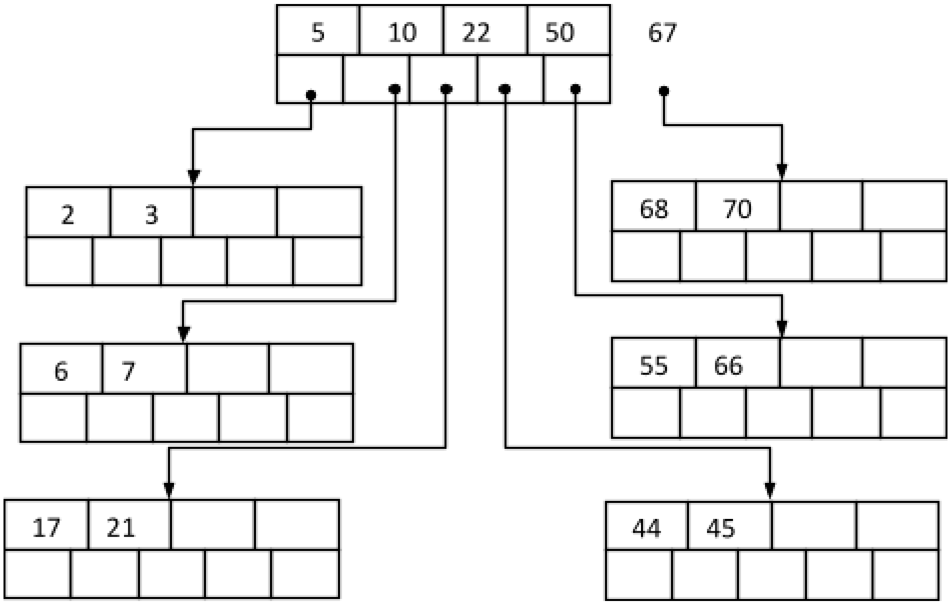


Figure 5. 5-way B-tree after insertion of 67 in **Figure** 4

But now the node above does overflow {**Figure 6**}. So it is splitting in exactly the same manner:

- Left = [5 10] (along with their children)
- Middle = 22
- Right = [50 67] (along with their children)

Left and Right become nodes, the children of Middle. If this is not the root, Middle is added to the node above and the process repeated. If there is no node above, as in this example, a new root is created with Middle as its only value.

The tree-insertion algorithms we have previously seen add new nodes at the bottom of the tree, and then have to worry about whether doing so creates an imbalance. The B-tree insertion algorithm is just the opposite: it adds new nodes at the top of the tree (a new node is allocated only when the root splits). B-trees grow at the root, not at the leaves. Because of this, there is never any doubt that the tree is always perfectly height balanced: when a new node is added, all existing nodes become one level deeper in the tree.

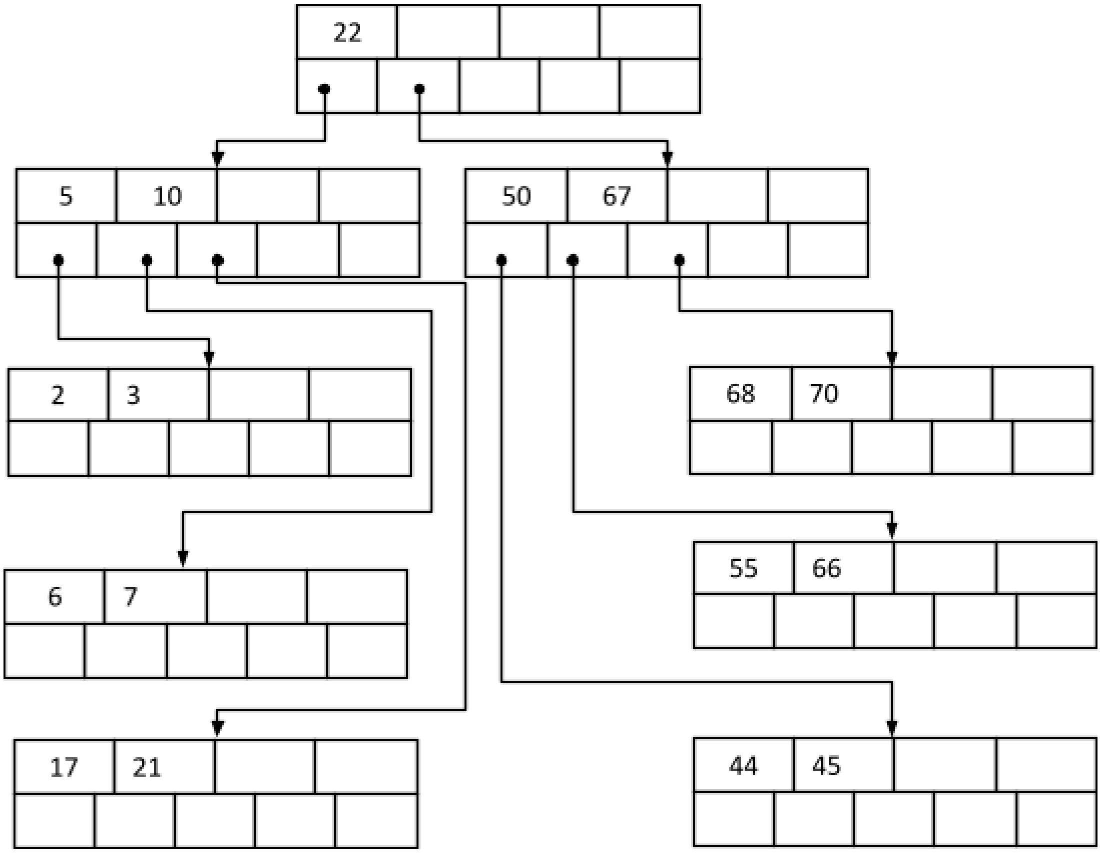


Figure 6-way B-tree (resulting)

Deleting a Key Value from a B-Tree

To delete value X from a B-tree, starting at a leaf node, there are 2 steps:

1. Remove X from the current node. Being a leaf node there are no subtrees to worry about.
2. Removing X might cause the node containing it to have too few values.
   We require to maintain in the root at least 1 value and all other nodes to have at least (m-1)/2 values in them. If the node has too few values, we say it has goneunderflow condition.

   If underflow does not occur, then we are finished the deletion process. If it does occur, it must be fixed. The process for fixing a root is slightly different than the process for fixing the other nodes. For example consider Figure 7.If deleting 6 from this B-tree. Here degree of B-tree is m=5

Removing 6 causes the node into underflow, as it now contains just one value 7. For fixing this is to try to borrow values from a neighboring node.

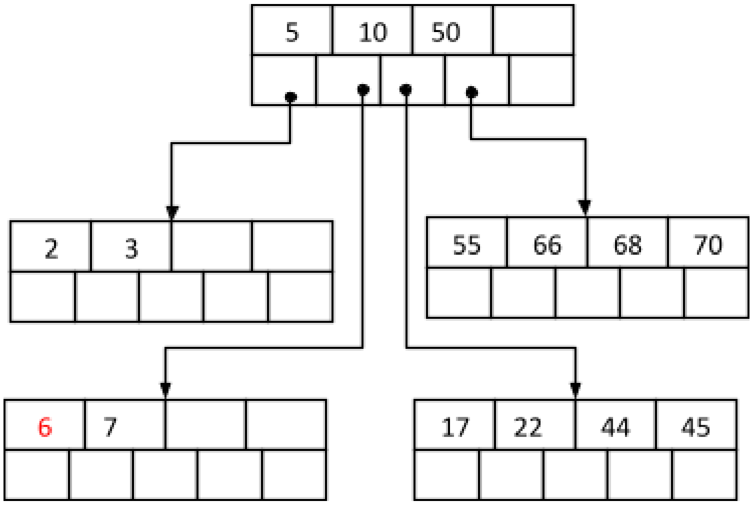


Figure 7. B-Tree with m=5

We join together the current node and its more populous neighbour to form a combined node - and we must also include in the combined node the value in the parent node that is in between these two nodes. Key of current node is **7, 10, 17, 22, 44,45**.

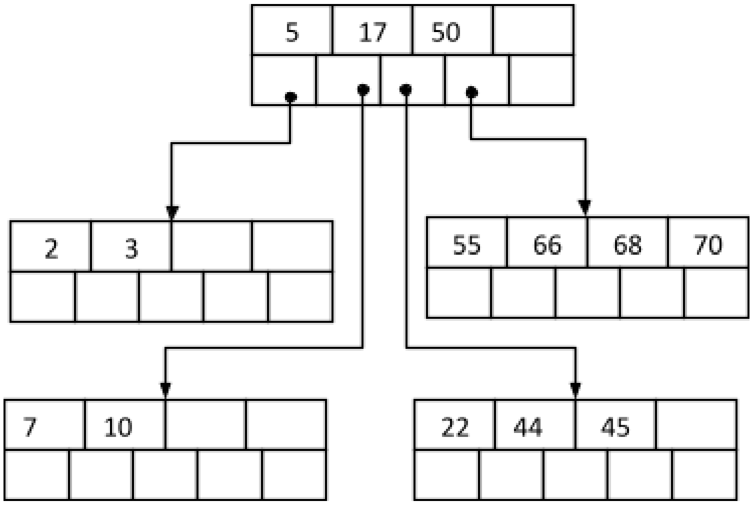


Figure 8.B-Tree after deletion of 6 from B-Tree in Figure 7.

When we delete a key from node three cases arise.

Case 1: The parent node contributes 1 value.

Case 2: The node that underflowed contributes exactly (m-1)/2-1 values.

Case 3: The neighbouring node contributes somewhere between (m-1)/2 and (m-1) values.

A variant of the B-tree, known as a **B+-tree** considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially; the entire tree may be scanned without visiting the higher nodes at all.