Trees

- A tree is a non-linear data structure, which consists of collection of nodes connected by edges. A tree consists of a distinguished node r, called the root. In the tree of Figure 4.1, the root is A.
- In a tree with n number of nodes, the number of edges will be n-1.
- Each node except the root has one parent. Node C has A as a parent and H, I, and J as children.

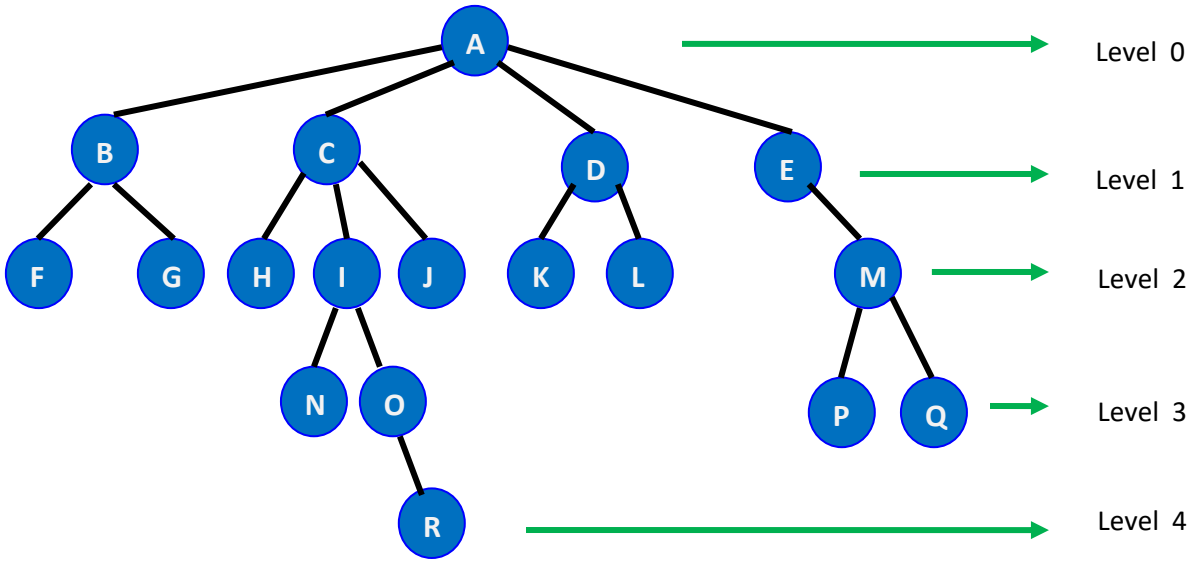


Figure 4.1: An example tree

- Each node may have an arbitrary number of children, possibly zero. Nodes with no children are known as leaves; the leaves in the tree above are F, G, H, N, R, J, K, L, P, Q, and R.
- Nodes with at least one child are sometimes called internal or non-terminal nodes. In the above figure, B, C, D, E, I, M, and O are internal nodes.
- Nodes with the same parent are siblings; thus H, I, and J are all siblings.
- A is grandparent of I and I is a grandchild of A.
- A path from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, . . . , n_k such that n_i is the parent of n_{i+1} . The length of the path is the number of edges on the path. In a tree there is exactly one path from the root to each node.
- For any node n_i, the depth of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0.
- The height of n_i is the longest path from n_i to a leaf. Thus all leaves are at height 0. The height of a tree is equal to the height of the root. For the tree in Figure 4.1, E is at depth 1 and height 2; F is at depth 2 and height 0; the height of the tree is 4. The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the

tree. If there is a path from n_1 to n_2, then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1.

- The ***level*** of a node is the number of nodes on the path from the node to the root node.
- Degree of a node is the number of its children/subtrees. Degree of node A in figure 4.1 is 4. Degree of node M is 2. The degree of a tree is the maximum degree of any of its nodes. Degree of tree shown in figure 4.1 is 4.
- The sum of the depths of all nodes in a tree is known as the internal path length.

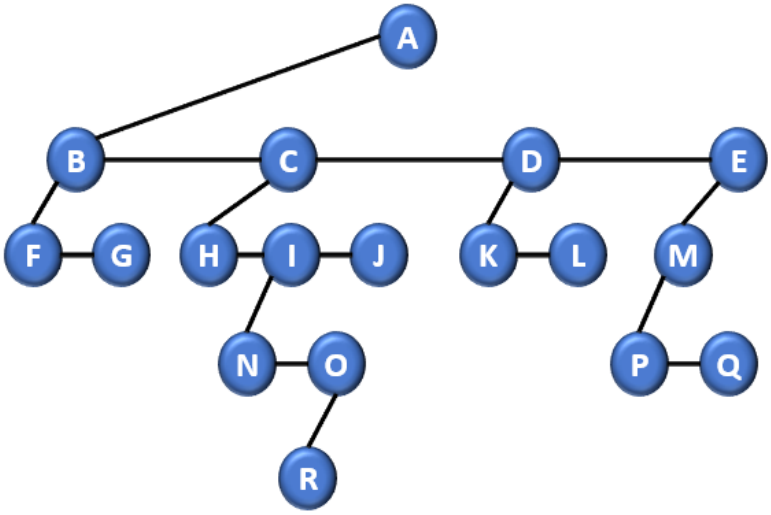
Representation of Trees

1. **List Representation**

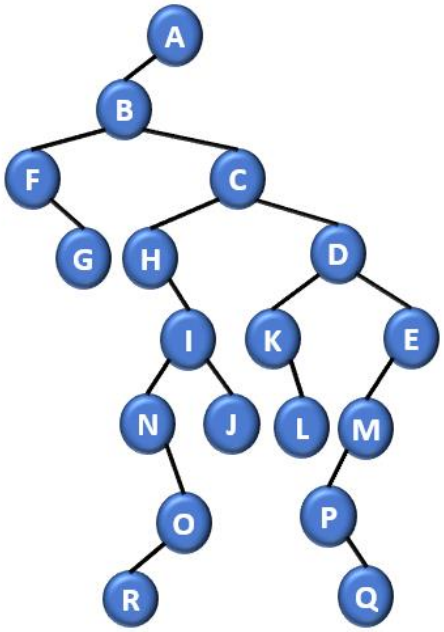
   Tree can be represented as a list. Information in the root node comes first, followed by a list of the subtrees of that node. Tree in Figure 4.1 can be represented as a list (A (B(F, G), C(H, I(N, O(R)), J), D(K, L), E(M(P, Q))

2. **Left Child-Right Sibling Representation**

   Since it is often easier to work with nodes of a fixed size, this representation requires exactly two link or pointer fields per node, the left child and right sibling pointers.
   To convert a tree into this representation, note that every node has only one leftmost child and one closest right sibling. For example, in Figure 4.1, the leftmost child of A is *B,* and the leftmost child of *D* is *K.* Similarly, the closest right sibling of *B* is C, and the closest right sibling of *H* is I. Following figure shows the tree of Figure 4.1 redrawn using the left child-right sibling representation.



3. **Representation as a degree two tree**
   To obtain the degree two tree representation, simply rotate the left child-right sibling tree clockwise by 45 degrees. This gives the degree two tree displayed in following figure.



Binary Trees

A binary tree is a tree in which no node can have more than two children. The average depth of a binary tree is O(√n). In the worst case the depth can be as large as n-1.

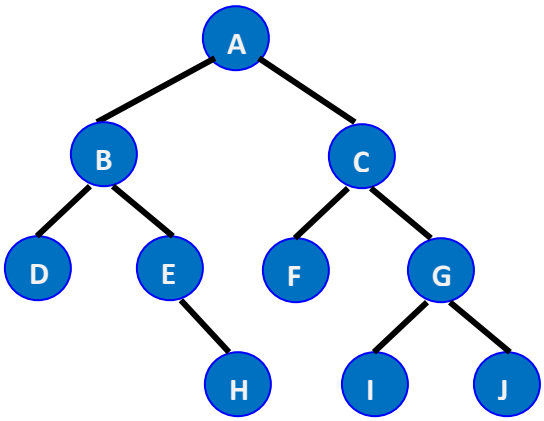


Figure 4.2: An example Binary Tree

Types of Binary trees

- A full binary tree (sometimes proper binary tree or 2-tree or strictly binary tree) is a tree in which every node other than the leaves has two children. Or, perhaps more clearly, every node in a binary tree has exactly (strictly) 0 or 2 children.

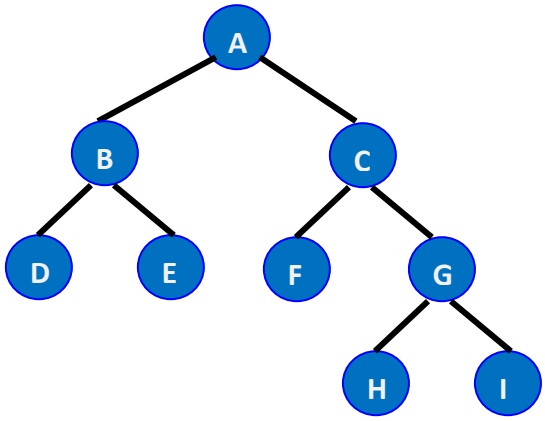


Figure 4.3: An example Full Binary Tree

- A **perfect binary tree** is a *full binary tree* in which all *leaves* are at the same *depth* or same *level*, and in which every parent has two children. (This is ambiguously also called a *complete binary tree* (see next).) An example of a perfect binary tree is the ancestry chart of a person to a given depth, as each person has exactly two biological parents (one mother and one father); note that this reverses the usual parent/child tree convention, and these trees go in the opposite direction from usual (root at bottom).

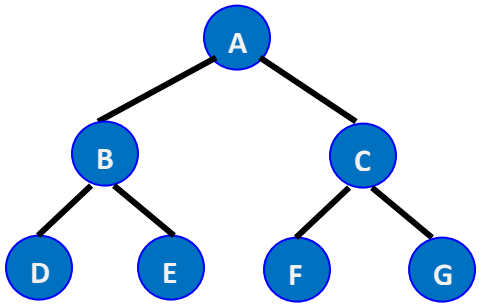


Figure 4.4: An example Perfect Binary Tree

- A **complete binary tree** is a binary tree in which every level, *except possibly the last*,(the last level is not completely filled) is completely filled, and all nodes are as far left as possible. . This type of tree is used as a specialized data structure called a heap.

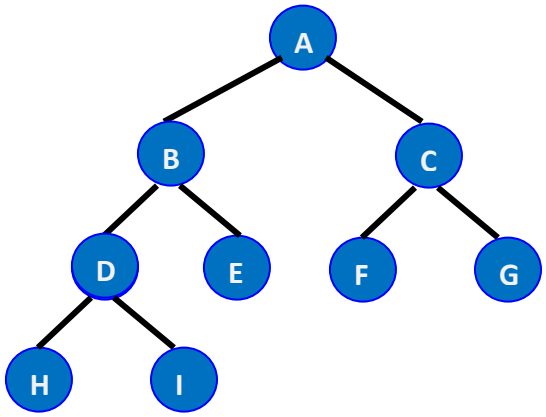


Figure 4.5: An example Complete Binary Tree

- A **balanced binary tree** is commonly defined as a binary tree in which the depth of the left and right subtrees of every node differ by 1 or less. Binary trees shown in above figures 4.2 to 4.5 are all balanced. Binary tree shown in the following figure 4.6 is an unbalanced binary tree, since the difference between the heights of left subtree and right subtree of node B is 2 – 0 = 2, which is greater than 1.

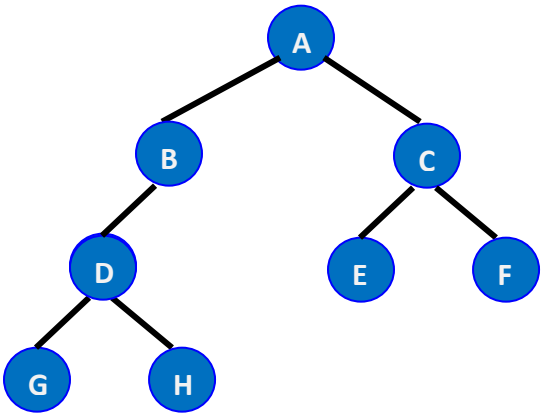


Figure 4.6: An example Binary Tree that is not balanced

Tree traversal

Tree traversal (also known as **tree search**) refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree.

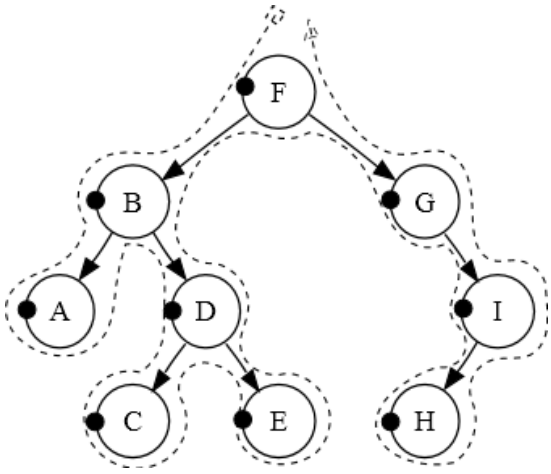


Figure 4.7: Tree Traversal

Pre-order Traversal

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

Application of these steps on the binary tree of figure 4.7 will visit the nodes in the order F, B, A, D, C, E, G, I, and H.

In-order Traversal

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

Application of In-order steps on the binary tree of figure 4.7 will visit the nodes in the order A, B, C, D, E, F, G, H, and I.

Post-order Traversal

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Application of Post-order steps on the binary tree of figure 4.7 will visit the nodes in the order A, C, E, D, B, H, I, G, and F.

Threaded binary trees

A **threaded binary tree** is a binary tree variant that allows fast traversal: given a pointer to a node in a threaded tree, it is possible to cheaply find its in-order successor (and/or predecessor).

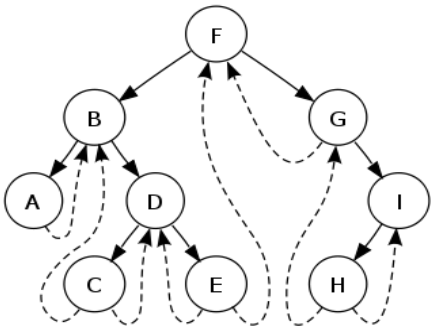
A threaded binary tree is defined as follows:
"A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node (**if** it exists), and all left child pointers that would normally be null, point to the inorder predecessor of the node.

What are the advantages of threaded tree?

- A simple tree traversal algorithm (inorder/preorder/postorder traversal algorithm) has a problem that, because of its recursion, it uses stack space proportional to the height of a tree. If the tree is fairly balanced, this amounts to $O(\log n)$space for a tree containing n elements. In the worst case, when the tree is skewed, the height of the tree is n so the algorithm takes $O(n)$ space. One of the solutions to this problem is tree threading.
- It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack
- A binary tree with n nodes would have unused (NULL) pointers for which lot of memory gets wasted. Threads could be set up on those unused pointers allowing one to traverse the tree iteratively.

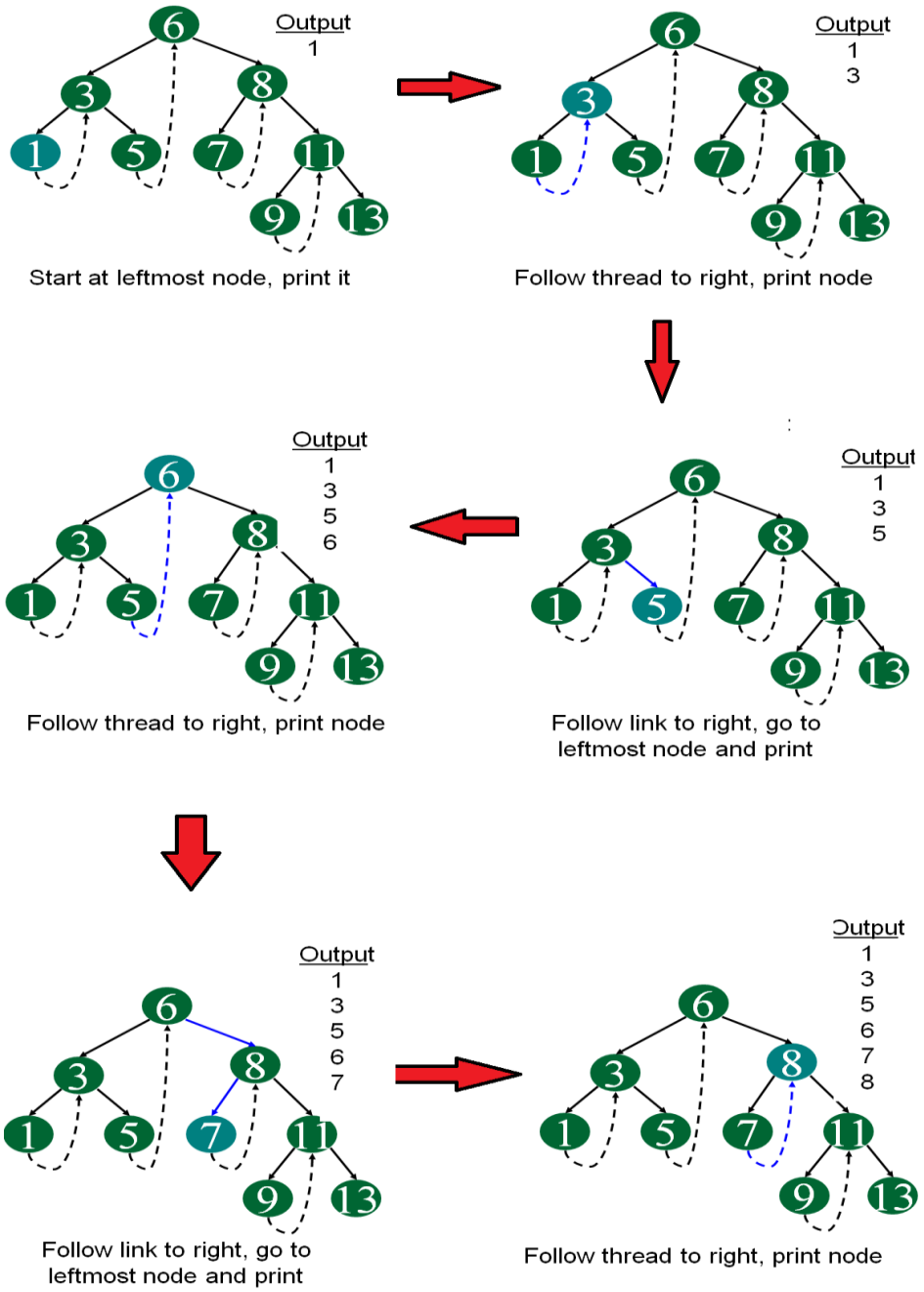
Conversion of binary search tree to threaded binary search tree

Consider the following threaded BST.



The inorder traversal of this tree yields: A B C D E F G H I. Since C's inorder predecessor is B, the left thread from C points to B. The right thread from C points to D which is the inorder successor of D. Since A has no predecessor, A's left thread doesn't exist (dangling thread) as does for G whose right thread doesn't exist.

Following diagram demonstrates in order traversal using threads.



continue same way for remaining node.....

Image Source: www.cs.berkeley.edu/~kamil/teaching/su02/080802.ppt

Expression Tree

Binary trees have many important uses. One of the principal uses of binary trees is in the area of compiler design, in the form of an expression tree. Figure 4.8 shows an example of an expression tree. The leaves of an expression tree are operands, such as constants or variable names, and the other nodes contain operators. This particular tree happens to be binary, because all of the operations are binary. We can evaluate an expression tree, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees. In our example, the left subtree evaluates to a + (b * c) and the right subtree evaluates to ((d *e) + f)*g. The entire tree therefore represents (a + (b*c)) + (((d * e) + f)* g).

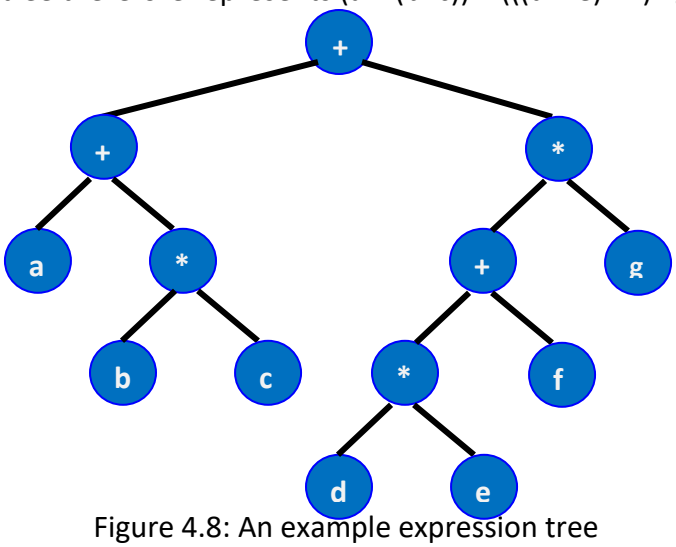


Figure 4.8: An example expression tree

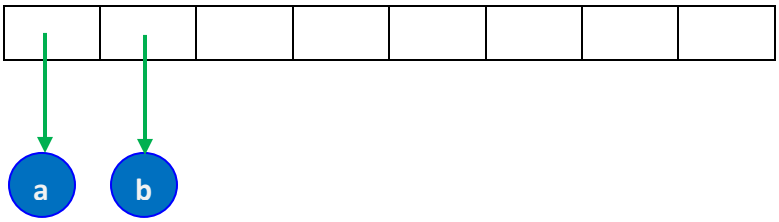
Application of Pre-order traversal on the expression tree would produce prefix expression, In-order traversal would produce infix expression, and Post-order traversal would produce postfix expression.

Algorithm to convert a postfix expression into an expression tree

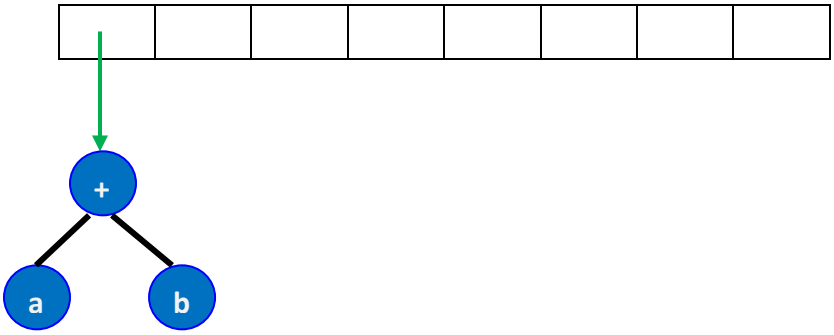
```
Step 1: Read a symbol from the postfix expression.
Step 2:  If symbol is an operand, create a tree node for the operand and push
      the node onto the stack.
Step 3:  If symbol is an operator,
      a. Create a tree node for the operator.
      b. Pop top two elements from the stack, and form a new tree by making
         them right and left subtrees of operator node. Push this newly
         formed tree onto the stack.
Step 4: Repeat the steps 1 to 3 until there are symbols to be processed in
      the postfix expression.
```

As an example, suppose the input is a b + c d e + * *

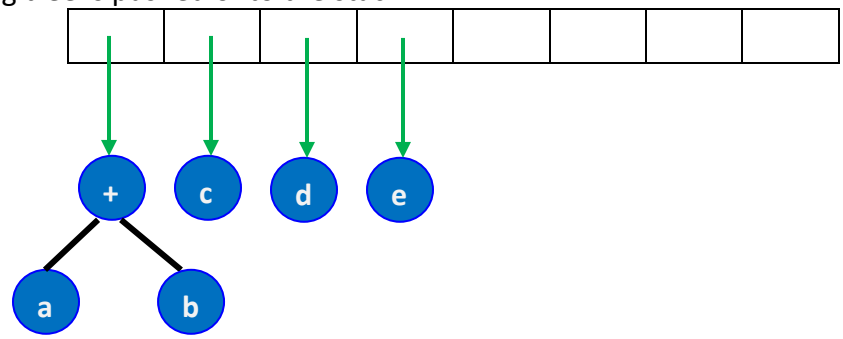
The first two symbols are operands, so create one-node trees and push pointers to them onto a stack.



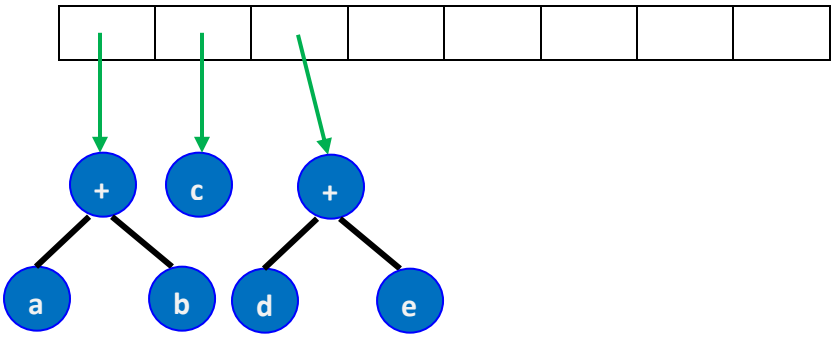
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



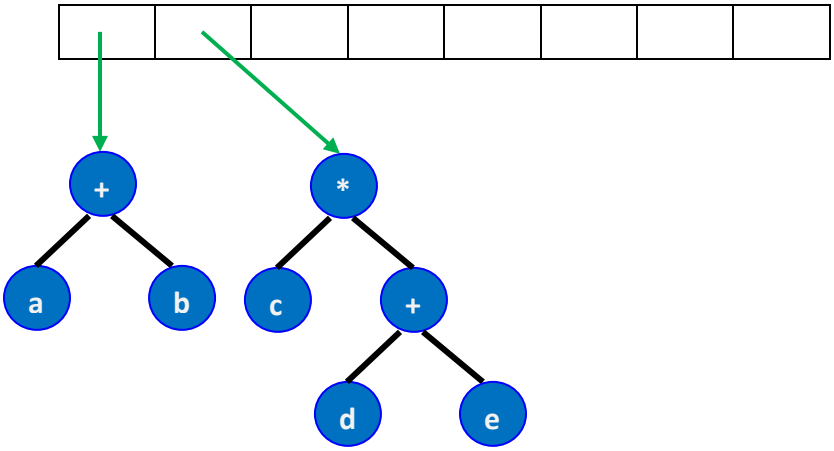
Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



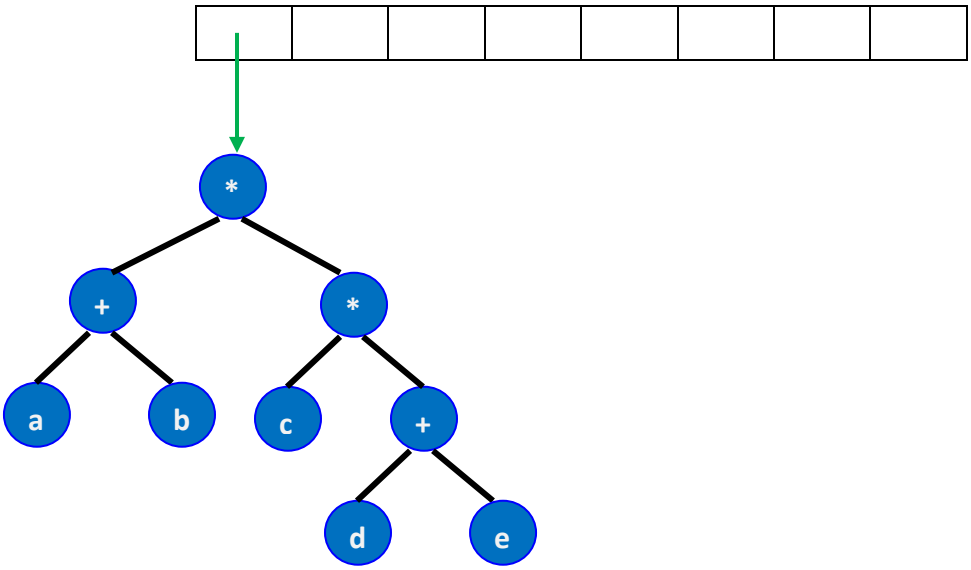
Now a '+' is read, so top two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



Skewed Binary trees : A skewed binary tree means a binary tree with either only right sub trees or only left subtrees. Right skewed binary tree will have only right sub trees. Left skewed binary tree will have only left sub trees. Figure 4.9 shows left skewed binary tree and Figure 4.10 shows right skewed binary tree.

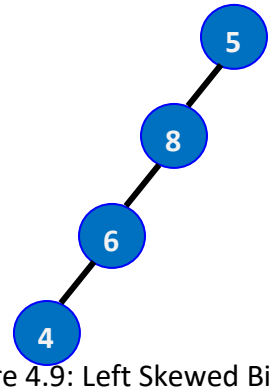


Figure 4.9: Left Skewed Binary Tree

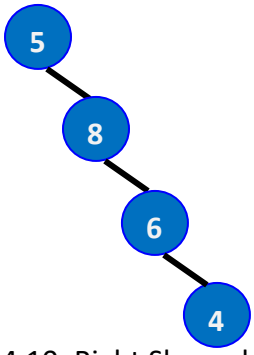


Figure 4.10: Right Skewed Binary Tree

Search Tree ADT – Binary Search Tree

Binary Search Tree is a binary tree in which, for every node, X, in the tree, the values of all the keys in the left subtree are smaller than the key value in X, and the values of all the keys in the right subtree are larger than the key value in X.

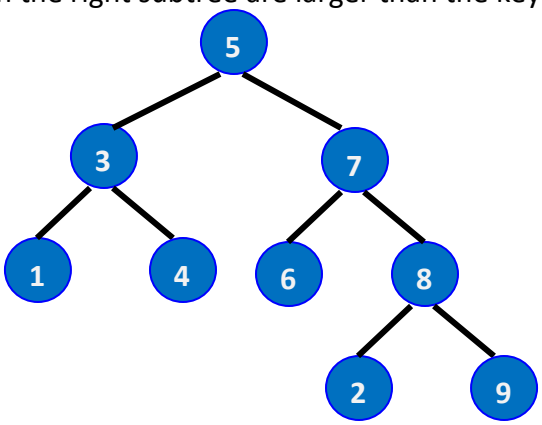


Figure 4.11: Binary tree

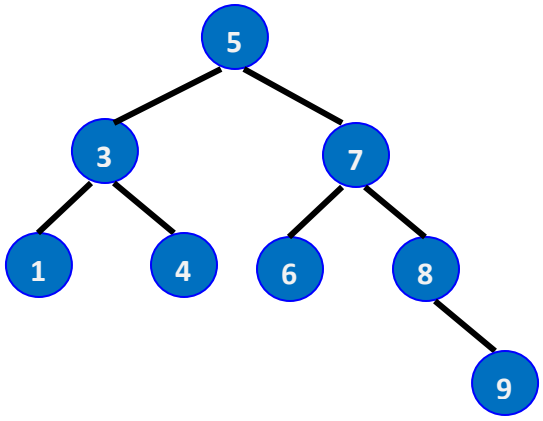


Figure 4.12: Binary Search Tree

The tree in Figure 4.12 is a Binary Search Tree but the one shown in Figure 4.11 is not a Binary Search Tree, since the tree has a node with key 2 in the right subtree of a node with key 7. The average depth of a binary search tree is O(log n), where n is the number of nodes in the tree.

Operations on BST

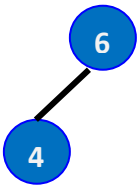
1. **Insert :** Inserting a node into Binary Search Tree can be divided into two steps:

- Search for a place to put a new element;
- Insert the new element to this place.

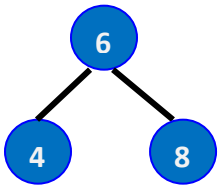
To search for place to insert new element, algorithm should follow binary search tree property. If a new value is less than the current node's value, go to the left subtree, else if it is greater than the current node's value, go to the right subtree. Following this simple rule, the algorithm reaches a leaf node, to which the new element is to be inserted as left child or right child depending on its value.

As an example, assume that the elements 6, 4, 8, and 7 are to be inserted into a BST. While inserting 6, because the tree is not existing, place to insert 6 is root node. The resulting tree after insertion would be as shown below

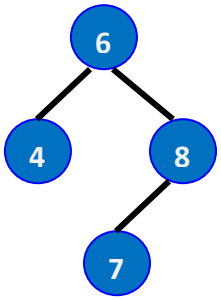
While inserting 4, inorder to find the place for inserting 4, 4 is compared with 6. Since 4 is less than 6 and 6 has no left child, 4 should be inserted as the left child of 6.



While inserting 8, inorder to find the place for inserting 8, 8 is compared with 6. Since 8 is greater than 6 and 6 has no right child, 8 should be inserted as the right child of 6.



While inserting 7, inorder to find the place for inserting 7, 7 is compared with 6. Since 7 is greater than 6, by following right subtree, 7 is compared with 8. Since 7 is less than 8 and 8 has no left child, 7 should be inserted as the left child of 8.



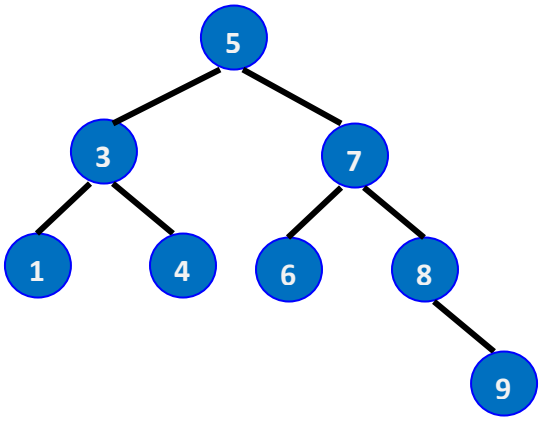
2. Find : Find searches for a node whose value matches with the element searching for, and returns a pointer to that node. It returns NULL if there is no matching node.

3. FindMin and FindMax: These routines return pointer to the smallest and largest elements in the tree, respectively. To perform a FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element. The FindMax routine is the same, except that branching is to the right child.

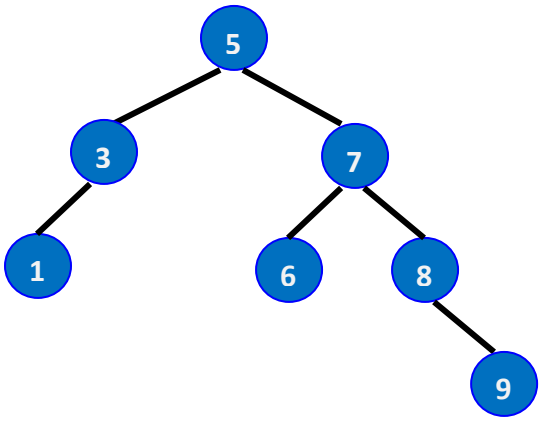
4. Delete: Once the node to be deleted is found, the following possibilities should be considered for deleting the node.

- If the node is a leaf, it can be deleted immediately.
- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node.
- If the node has two children, following strategy should be followed
  a) Find the smallest element in the right subtree of node to be deleted
  b) Replace the value of node to be deleted with the value of smallest element found in previous step.
  c) Delete the smallest element in the right subtree of node to be deleted

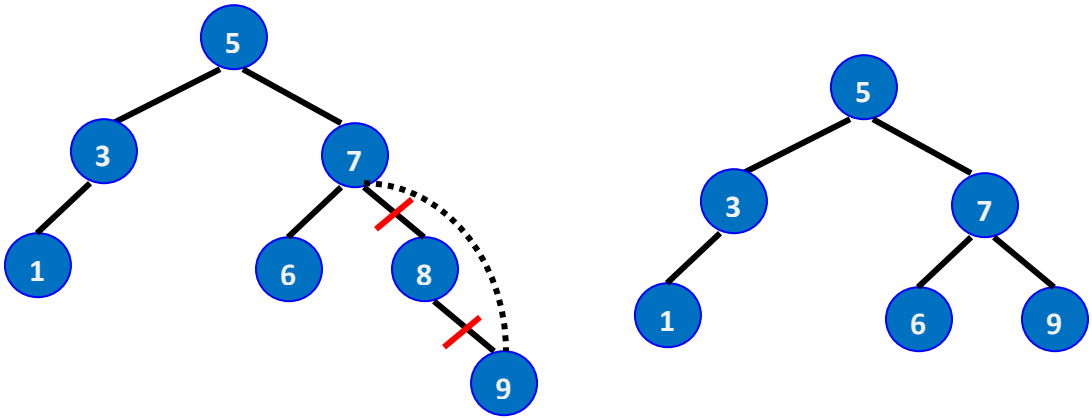
Consider the following Binary Search Tree. Assume the element to be deleted is 4.



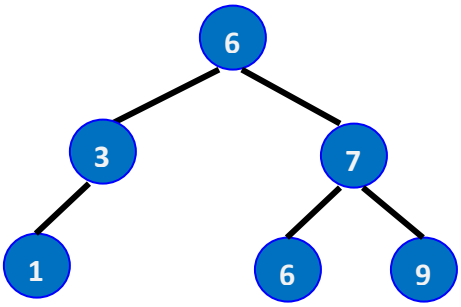
Since 4 is leaf node, it can be deleted immediately. Resulting tree would be as shown below:



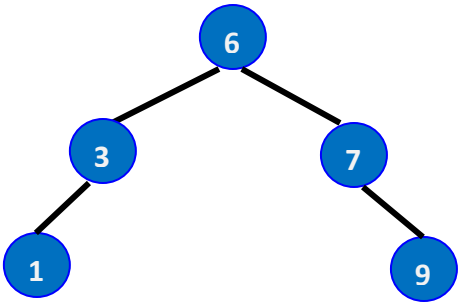
Next, if the node to be deleted is 8, the node can be deleted after node 7 adjusts a pointer to bypass the node 8.



Next, if the node to be deleted is 5, then follow the strategy given above.
First find the smallest element in the right subtree of 5. Smallest element is 6. Copy 6 into node 5.



Now, delete element 6 which is in the right subtree of 5.



The average time complexity of all the operations discussed above is O(log n), where n is number of nodes in the binary search tree.

C implementation of operations on a Binary Search Tree

```
#include<stdio.h>

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

struct node * insert (int ele,struct node *t)
{
      if(t==NULL)
      {
            t=(struct node *)malloc(sizeof(struct node));
            t->data=ele;
            t->left=NULL;
            t->right=NULL;
      }
      else
      {
            if(ele < t->data)
                  t->left = insert(ele, t->left);
            else if(ele > t->data)
                  t->right = insert(ele, t->right);
      }
      return t;
}

void preorder (struct node *t)
{
      if(t!=NULL)
      {
            printf("%d\t",t->data);
            preorder(t->left);
            preorder(t->right);
      }
}
void postorder (struct node *t)
{
        if(t!=NULL)
        {
            postorder(t->left);
            postorder(t->right);
            printf("%d\t",t->data);
      }
}
void inorder (struct node *t)
{
        if(t!=NULL)
        {
            inorder(t->left);
            printf("%d\t",t->data);
            inorder(t->right);
      }
}
```

```
struct node * find (int ele, struct node *t)
{
	if(t!=NULL)
	{
		if(t->data == ele)
		{
			printf("Element found\n");
			return t;
		}
		else if(t->data < ele)
			return find(ele, t->right);
		else
			return find(ele, t->left);
	}
	else
	{
		printf("Element not found\n");
		return NULL;
	}
}

struct node * findmin (struct node *t)
{
	if(t == NULL)
		return NULL;
	while(t->left != NULL)
		t = t->left;
	return t;
}

struct node * findmax (struct node *t)
{
	  if(t == NULL)
			return NULL;
	  while(t->right != NULL)
			t = t->right;
	  return t;
}

struct node * delete(int ele, struct node *t)
{
	struct node *temp;
	if(t == NULL)
	{
		printf("No element to delete...\n");
		return NULL;
	}
	if(ele < t->data)
		t->left = delete(ele, t->left);
	else if(ele > t->data)
		t->right = delete(ele,t->right);
	else
	{
		if(t->left != NULL && t->right != NULL)
		{
			temp = findmin(t->right);
```

```
                t->data = temp->data;
                t->right = delete(t->data, t->right);
            }
            else if(t->left == NULL)
                t = t->right;
            else if (t->right == NULL)
                t = t->left;
        }
        return t;
}

main()
{
        int ch,n;
        struct node *m, *root=NULL;
        while(1)
        {
            printf("\n1.Insert\n2.Preorder\n3.Inorder\n4.Postorder\n5.Find
element\n6.Find max\n7.Find min\n8.Delete an element\n9.Exit:\nEnter your
choice:\n");
            scanf("%d",&ch);
            switch(ch)
            {
                case 1: printf("Enter the element to insert:\n");
                        scanf("%d",&n);
                        root=insert(n,root);
                        break;
                case 2: preorder(root);
                        break;
                case 3: inorder(root);
                        break;
                case 4: postorder(root);
                        break;
                case 5: printf("Enter the number to find:\n");
                        scanf("%d",&n);
                        m=find(n,root);
                        break;
                case 6: m=findmax(root);
                        if(m!=NULL)
                            printf("Maximum element is %d\n",m->data);
                        break;
                case 7: m=findmin(root);
                        if(m!=NULL)
                            printf("Minimum element is %d\n",m->data);
                        break;
                case 8: printf("Enter the element to delete:\n");
                        scanf("%d",&n);
                        m=delete(n,root);
                        break;
                case 9: exit(0);
                default: printf("Invalid choice\n");
            }
        }
}
```

AVL trees

An AVL tree is a binary search tree, with the property that for every node in the tree, the height of the left and right subtrees can differ by at most 1.
The tree in Figure 13 is an AVL tree, but the tree in figure 14 is not. Height information is kept for each node in the node structure.

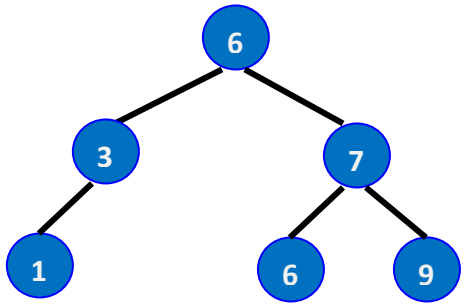


Figure 13: An example AVL Tree

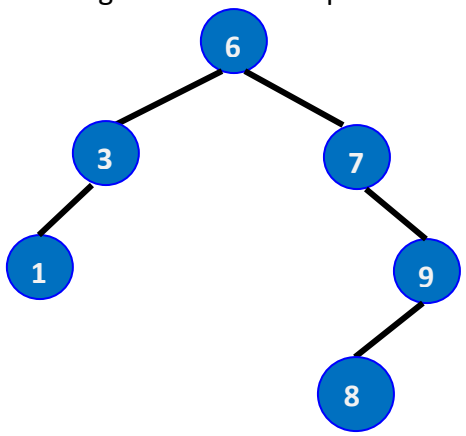


Figure 14: A Binary Search Tree but not an AVL Tree

Height of an AVL tree is log(n + 1) + 0.25. The minimum number of nodes, N(h), in an AVL tree of height h is given by N(h) = N(h - 1) + N(h - 2) + 1. For h = 0, N(h) = 1. For h = 1, N(h) = 2. All the tree operations except insert and delete can be performed in O(log n) time.

Rotations

Inserting a node into the AVL tree or deleting a node from AVL tree could violate the AVL tree property. In order to restore the AVL property, a simple modification must be made to the tree. This modification is called rotation.

During insertion, AVL property violation might occur at a node (call t) in four cases:

1. An insertion into the left subtree of the left child of t.
2. An insertion into the right subtree of the left child of t.

3. An insertion into the left subtree of the right child of t.
4. An insertion into the right subtree of the right child of t.

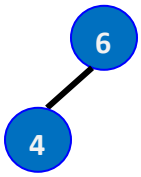
If the case is 1 or 4 violation can be fixed by single rotation else, if the case is 2 or 3 violation can be fixed by double rotation.

- If the case is 1, LL Rotation can fix the violation.
- If the case is 2, LR Rotation can fix the violation.
- If the case is 3, RL Rotation can fix the violation.
- If the case is 4, RR Rotation can fix the violation.

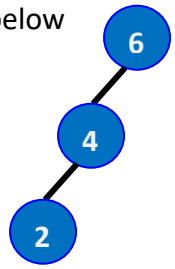
LL and RR rotations are single rotations, while LR and RL rotations are double rotations.

LL Rotation

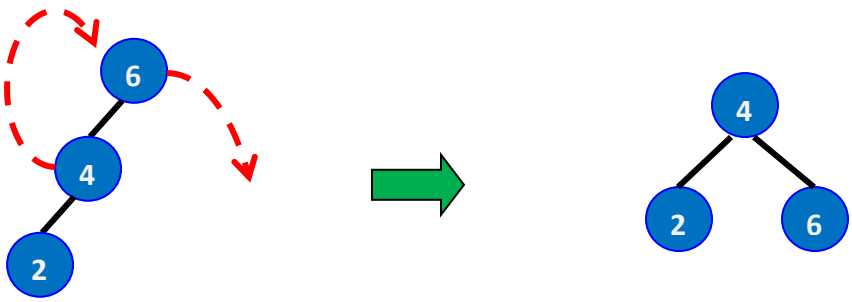
If the AVL property gets violated at a node t due to the insertion of a node in the left subtree of the left child of t, LL Rotation should be applied to fix the violation. Consider the following AVL tree consisting of 2 nodes as shown below.



Now assume that the next node to be inserted into the tree has a value 2. After inserting the node, tree would be as shown below

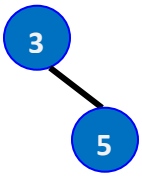


This insertion causes violation of AVL property at node 6. (Difference between heights of left subtree and right subtree is 2, which is greater than the maximum allowed height difference, 1). Since the insertion is in the left subtree of left child of 6, LL Rotation can fix the violation. After applying LL Rotation on node 6, the resulting tree would be:

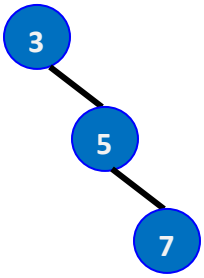


RR Rotation

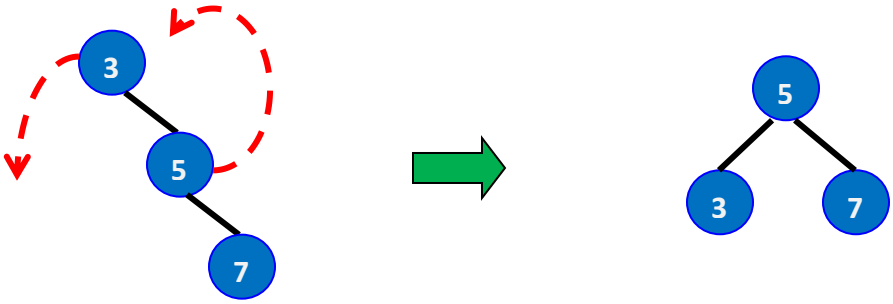
If the AVL property gets violated at a node t due to the insertion of a node in the right subtree of the right child of t, RR Rotation should be applied to fix the violation. Consider the following AVL tree consisting of 2 nodes as shown below.



Now assume that the next node to be inserted into the tree has a value 7. After inserting the node, tree would be as shown below

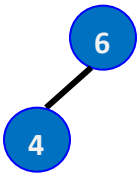


This insertion causes violation of AVL property at node 3. Since the insertion is in the right subtree of right child of 3, RR Rotation can fix the violation. After applying RR Rotation on node 3, the resulting tree would be as shown below.

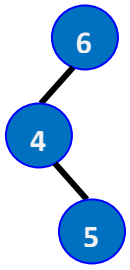


LR Rotation

If the AVL property gets violated at a node t due to the insertion of a node in the right subtree of the left child of t, LR Rotation should be applied to fix the violation. Consider the following AVL tree consisting of 2 nodes as shown below.



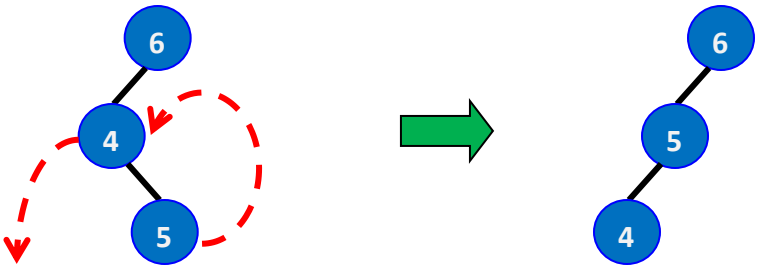
Now assume that the next node to be inserted into the tree has a value 5. After inserting the node, tree would be as shown below



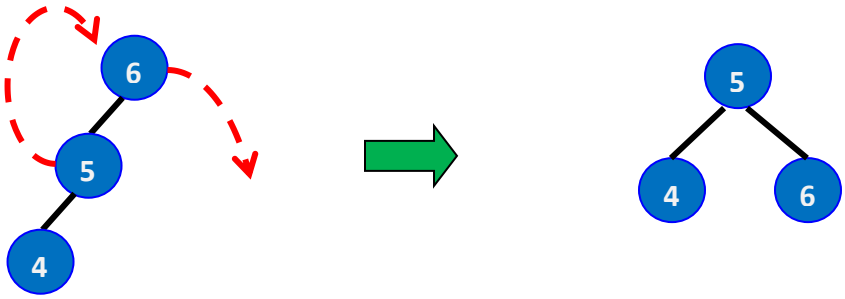
This insertion causes violation of AVL property at node 6. Since the insertion is in the right subtree of left child of 6, LR Rotation can fix the violation. LR rotation is a double rotation. It involves

1. Application of RR rotation on child node
2. Followed by application of LL rotation on parent node.

In this example, parent is node 6 and child is node 4. After applying RR rotation on node 4, resulting tree would be as shown below.

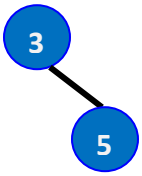


After applying LL Rotation on node 6, the resulting tree would be as shown below.

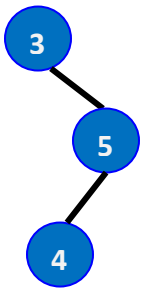


RL Rotation

If the AVL property gets violated at a node t due to the insertion of a node in the left subtree of the rightt child of t, RL Rotation should be applied to fix the violation. Consider the following AVL tree consisting of 2 nodes as shown below.



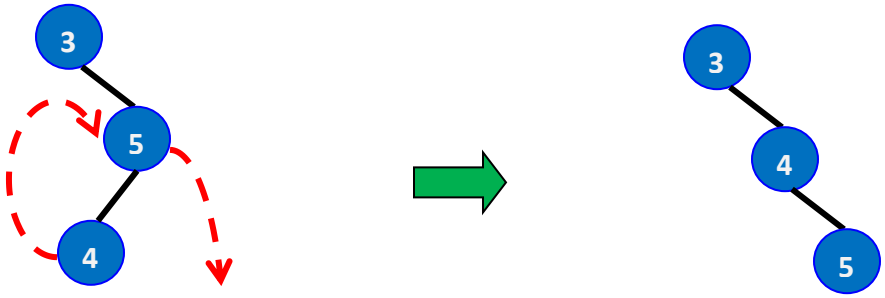
Now assume that the next node to be inserted into the tree has a value 4. After inserting the node, tree would be as shown below:



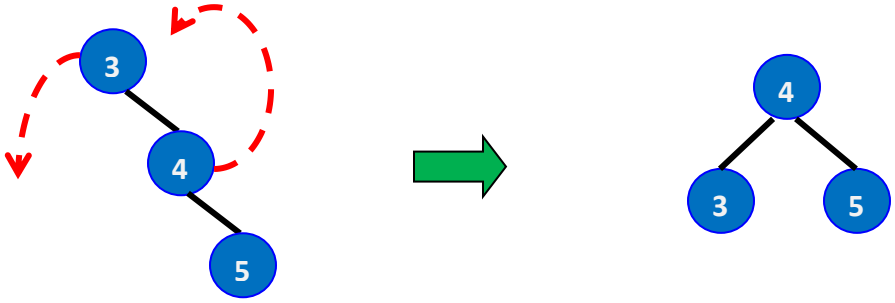
This insertion causes violation of AVL property at node 3. Since the insertion is in the left subtree of right child of 3, RL Rotation can fix the violation. RL rotation is a double rotation. It involves

1. Application of LL rotation on child node
2. Followed by application of RR rotation on parent node.

In this example, parent is node 3 and child is node 5. After applying LL rotation on node 5, resulting tree would be as shown below.



After applying RR Rotation on node 3, the resulting tree would be as shown below.



```
#include<stdio.h>

struct Avlnode
{
        int value, height;
        struct Avlnode *left, *right;
};

int Height(struct Avlnode *t)
{
        if (t == NULL) return -1;
        else
        return t->height;
}

int Max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
}

struct Avlnode * LLRotate(struct Avlnode *t)
{
        struct Avlnode *t1;
        t1 = t->left;
        t->left = t1->right;
        t1->right = t;
        t->height = Max(Height(t->left), Height(t->right)) + 1;
        t1->height = Max(Height(t1->left), t->height) + 1;
        return t1;
}

struct Avlnode * RRRotate(struct Avlnode *t)
{
        struct Avlnode *t1;
        t1 = t->right;
        t->right = t1->left;
        t1->left = t;
        t->height = Max(Height(t->left), Height(t->right)) + 1;
        t1->height = Max(t->height, Height(t1->right)) + 1;
        return t1;
}

struct Avlnode * LRRotate(struct Avlnode *t)
{
        t->left = RRRotate(t->left);
        t = LLRotate(t);
        return t;
}

struct Avlnode * RLRotate(struct Avlnode *t)
{
        t->right = LLRotate(t->right);
        t = RRRotate(t);
        return t;
```

```
}

struct Avlnode * insert(int elt, struct Avlnode *t)
{
        if(t == NULL)
        {
                t = (struct Avlnode *) malloc(sizeof(struct Avlnode));
                t->value = elt;
                t->height = 0;
                t->left = t->right = NULL;
        }
        else
        if(elt < t->value)
        {
                t->left = insert(elt, t->left);
                if ( Height(t->left) - Height(t->right) == 2)
                        if(elt < t->left->value)
                                t = LLRotate(t);
                        else
                                t = LRRotate(t);
        }
        else
        if(elt > t->value)
        {
                t->right = insert(elt, t->right);
                if ( Height(t->right) - Height(t->left) == 2)
                        if(elt > t->right->value)
                                t = RRRotate(t);
                        else
                                t = RLRotate(t);
        }
        t->height = Max(Height(t->left), Height(t->right)) + 1;
        return t;
}
```

Priority Queue (Heaps)

A priority queue is a data structure that allows at least the following two operations: insert, which does the obvious thing, and delete_min, which finds, returns and removes the minimum element in the heap.

Heap is a complete binary tree with heap property. A heap can be max heap or min heap. In max heap, the value at each node is bigger than the values at its children. In min heap, the value at each node is smaller than the values at its children.

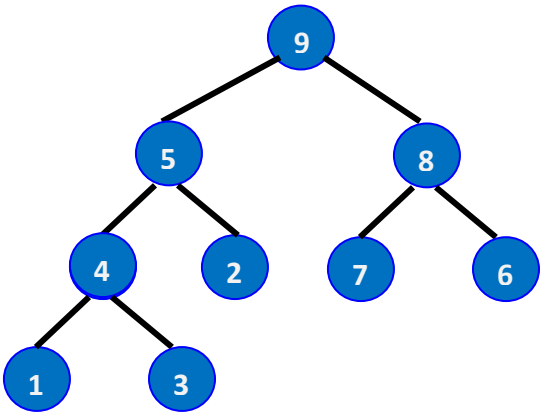


Figure 15: A Max Heap

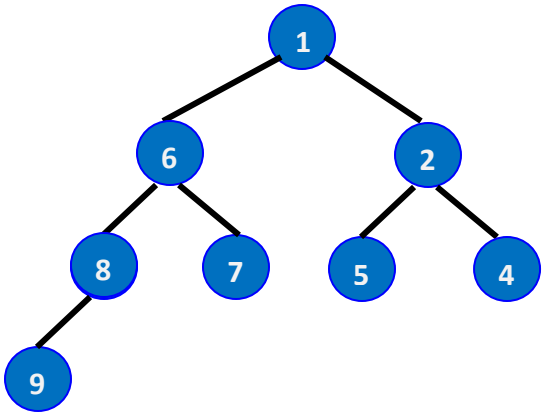
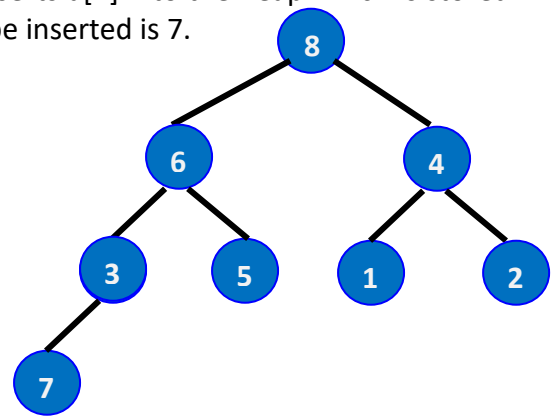


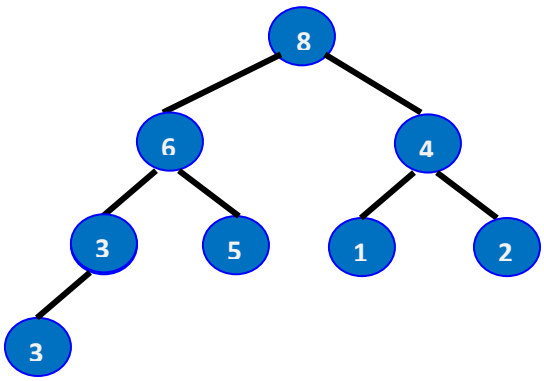
Figure 16: A Min Heap

Following operations can be performed on a heap.

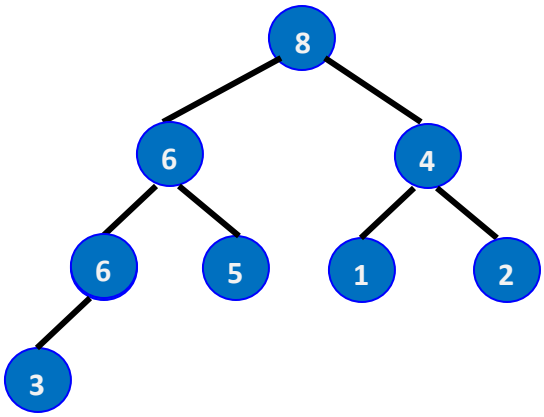
1. Insert: Insert operation inserts a[n] into the heap which is stored in an array whose size is n-1. Assume that the node to be inserted is 7.



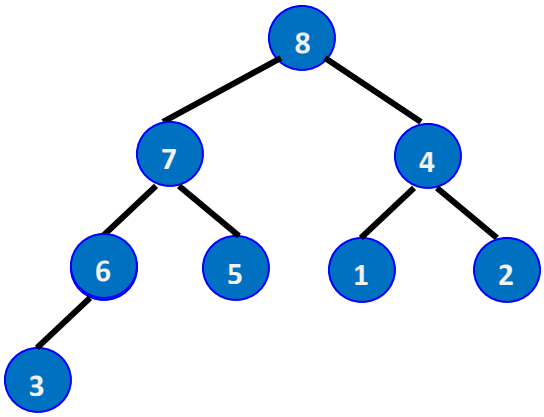
Compare 7 with its parent 3, in this example. Since 3 < 7, move 3 down to one level.



Now, 7 is compared with 6. Since 6 < 7, move 6 down to one level.



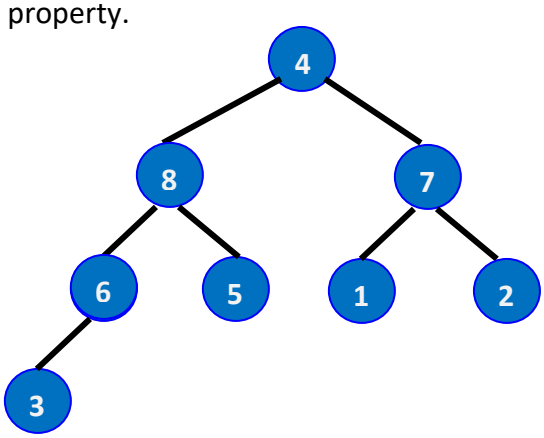
Next 7 is compared with 8. This time 8 > 7. 7 is placed below 8.



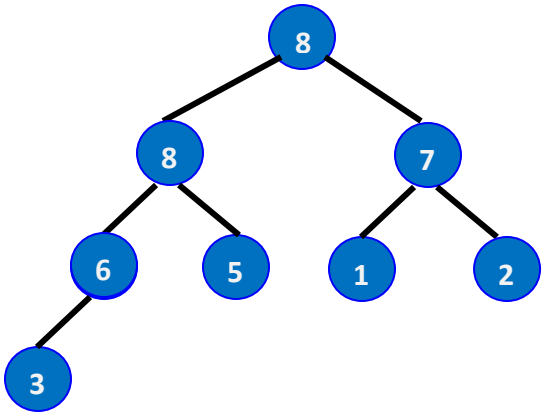
Following is the C code for insert function

```
void insert(int a[], int n)
{
            i = n;
            item = a[n];
            while ( (i>1) && (a[ i/2 ] < item )
            {
                        a[i] = a[ i/2 ];
                        i = i/2;
            }
           a[i] = item;
}
```

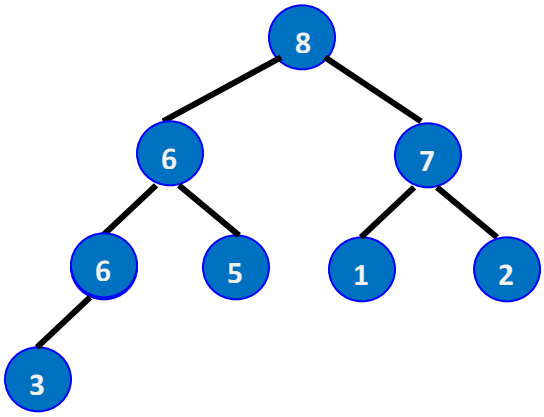
2. Adjust: Rearranges the elements of heap such that the tree rooted at index i is also a max heap. Consider the following complete binary tree which is not a heap because the element at root is violating heap property.



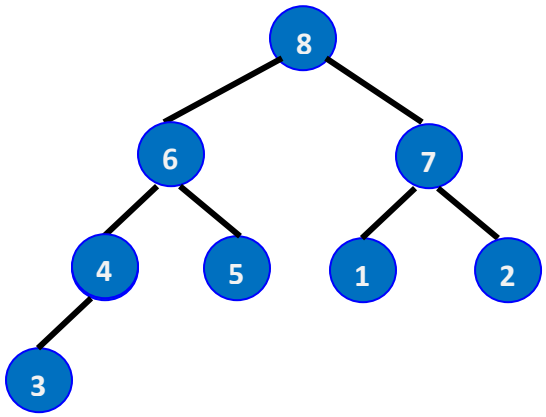
Adjust compares 4 with the biggest of its children (8 and 7) i.e., 8. since 8 > 4, 8 is copied into its parent's position



Next, 4 is compared with the biggest child of 8 i.e., 6. since 6 > 4, 6 is copied into its parent's position.



Next, 4 is compared with the biggest child of 6 i.e., 3. Since 3 < 4, 4 is copied into position of node 6.



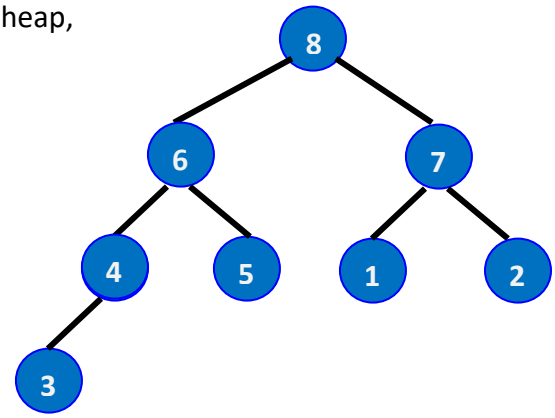
The resulting tree is a heap.

```
void Adjust(int a[],int i,int n)
{
      j = 2*i;
      item = a[i];
      while (j <= n) do
      {
            if ((j < n) && (a[j] < a[j+1])) then
                  j = j + 1;
            if (item >= a[j]) then
                  break;
            a[ j/2 ] = a[j]; j = 2j;
      }
a[ j/2 ] = item;
}
```

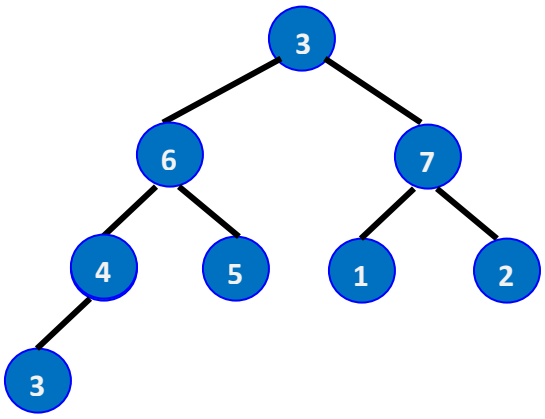
3. Delete Max: Delete Max deletes the biggest element of heap, which is at root node. It operates by the following steps:

1. The value of last node is copied into the root node
2. Adjust is invoked on root, to rearrange the elements of tree such that the resulting tree is a heap.

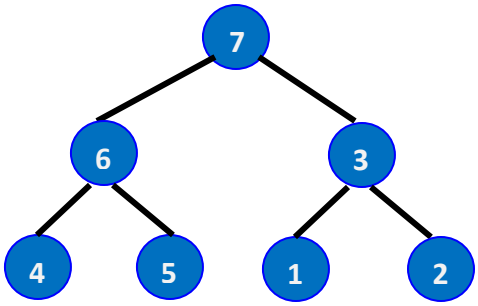
Consider the following heap,



Delete Max will first copy the value of last node into root node of tree. Resulting tree is:

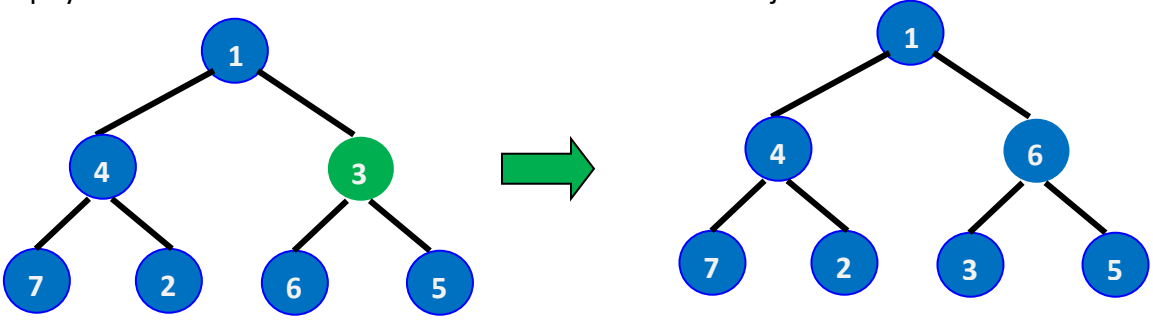


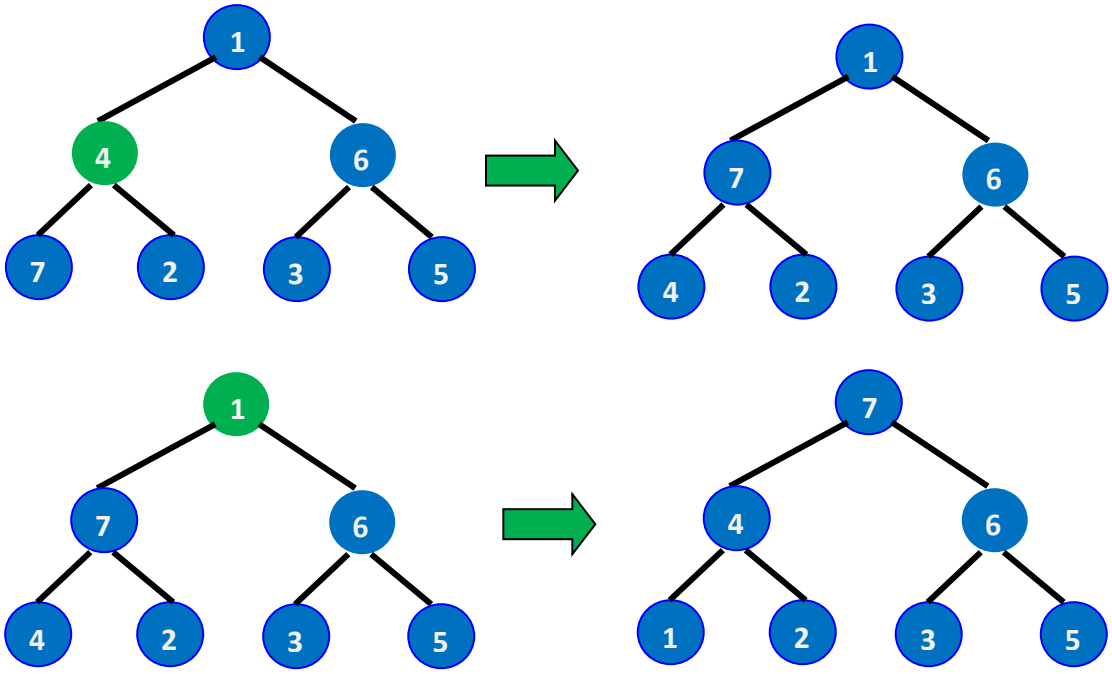
The above tree is not a heap because heap property is violated at root. Next, Delete Max invokes adjust on root by excluding last node, yielding a tree which is a heap.




```
void DelMax(int a[], int n)
{
            if (n ==0) then
            {
                  printf("Heap is empty");
                  return;
            }
            a[1] = a[n];
            Adjust(a, 1, n-1);
}
```

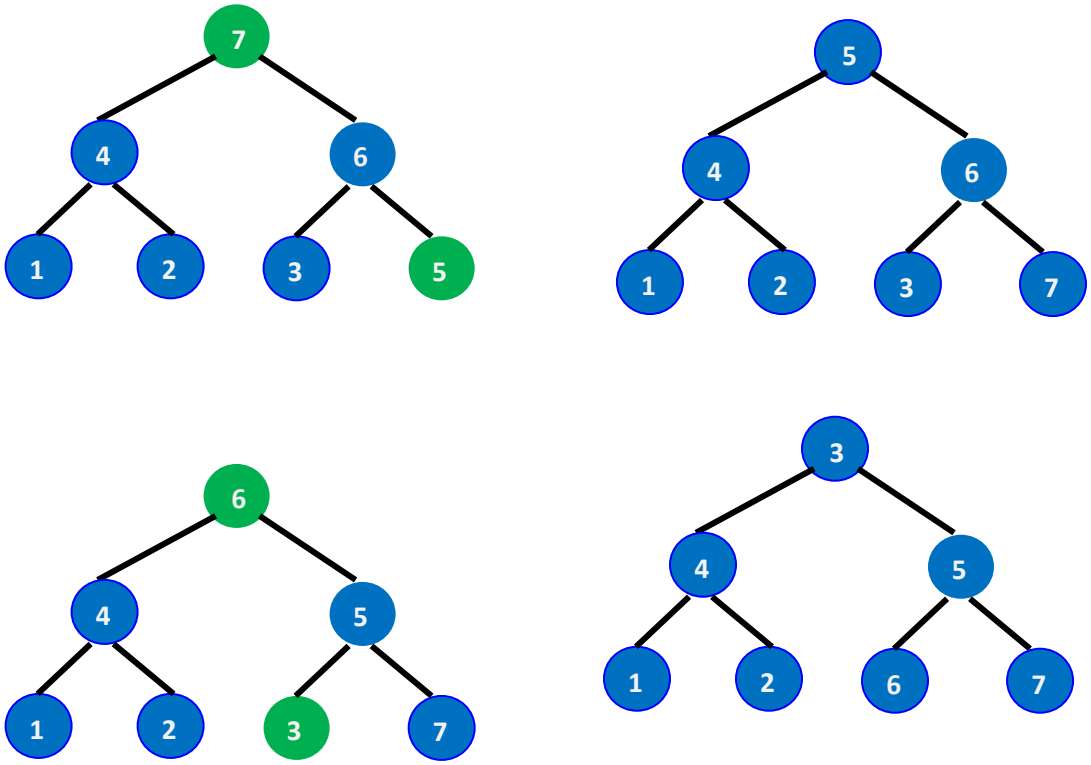
4. Heapify: Heapify creates a heap out of the given array/binary tree. Heapify begin by calling adjust for the parents of leaf nodes and then work its way up, level by level, until the root is reached. Following figures shows the sequence of changes made to the initial tree as a result of applying heapify. Green coloured node denotes the node on which adjust is called.

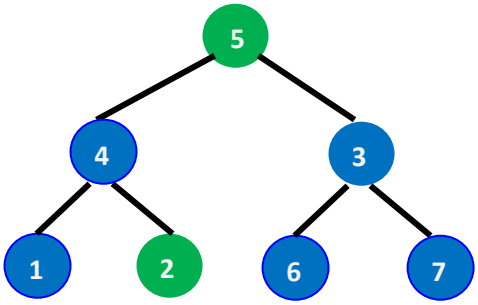


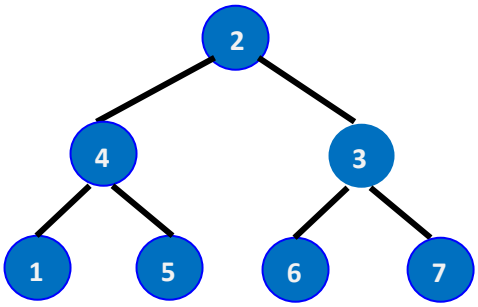


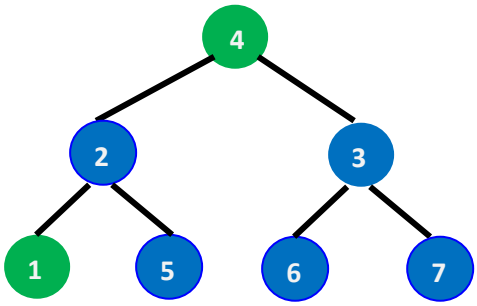
Heap Sort

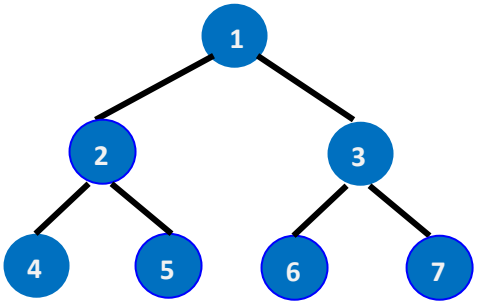
This algorithm requires the input array to be a heap. The algorithm swaps the first element (element at root)with the last element in the array, then invokes adjust on the first element by excluding last element. This process is repeated till the second element of the array. Heap sort strategy is shown in following figures. green colored nodes represent nodes that are to be swapped.

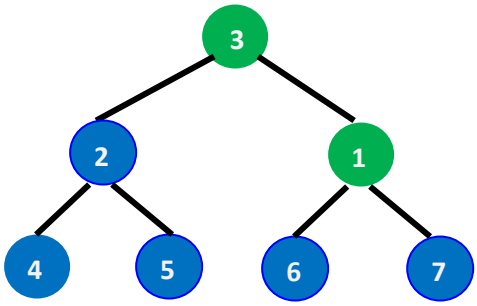


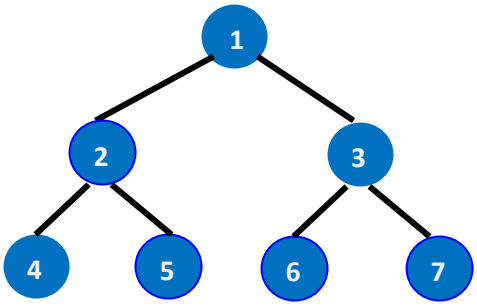

5
4
3
1
2
6
7

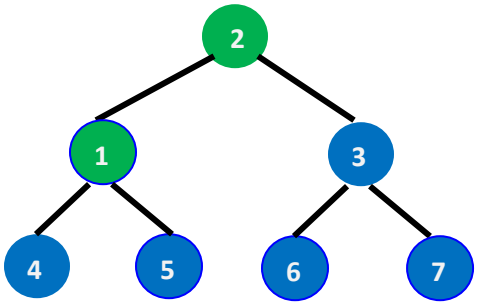

2
4
3
1
5
6
7

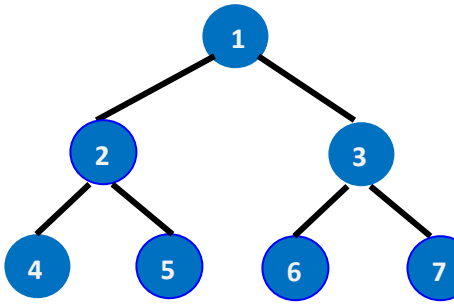

4
2
3
1
5
6
7


1
2
3
4
5
6
7


3
2
1
4
5
6
7


1
2
3
4
5
6
7


2
1
3
4
5
6
7


1
2
3
4
5
6
7

C program for Heap Sort is given below.

```
#include<stdio.h>

void Adjust(int a[],int i,int n)
{
      int j, item;
      j = 2*i;
      item = a[i];
      while (j <= n) do
      {
            if ((j < n) && (a[j] < a[j+1])) then
                  j = j + 1;
            if (item >= a[j]) then
                  break;
            a[ j/2 ] = a[j]; j = 2*j;
      }
a[ j/2 ] = item;
}

void Heapify(int a[], int n)
{
      int i;
      for(i = n/2; i >= 1; i--)
           Adjust(a,i,n);
}

void Heapsort(int a[], int n)
{
      int i, t;
      Heapify(a,n);
      for( i = n; i > 1; i--)
      {
           t = a[i];
           a[i] = a[1];
           a[1] = t;
           Adjust(a, 1, i-1);
      }
}

void main()
{
      int i, n, a[25];
      printf("\nEnter size of Heap:");
      scanf("%d",&n);
      printf("\nEnter elements of heap:");
      for(i = 1; i<= n; i++)
            scanf("%d",&a[i]);
      Heapsort(a,n);
      printf("\n Sorted Array: \n");
      for(i = 1; i<= n; i++)
            printf("%d  ",a[i]);
}
```

Time complexity of Heap Sort is O(n log n).

B-Tree

- Each internal node in the B-tree is of the form $<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$ where $q \leq p$.
  Each P_i is a pointer to a successor. Each Pr_i is a pointer to the data record containing key K_i .
- Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
- For all key values X in the subtree pointed at by P_i, we have:
  $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i=1$; and $K_{i-1} < X$ for $i = q$.
- Each node has at most p successors and p-1 key values.
- Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ successors and ($\lceil (p/2) \rceil$ -1) keys . The root node has at least two successors unless it is the only node in the tree
- A node with q successors, $q \leq p$, has q-1 key values
- All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their successor pointers P_i are null.

1. Insertion

To insert value X into a B-tree of order p, following are the steps:

1. 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.
3. If there are p-1 or fewer values/keys in the node after adding X, then we are finished.

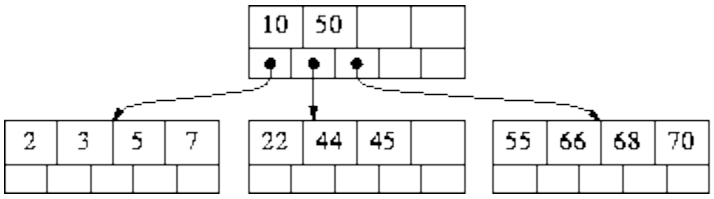
   If there are p keys after adding X, we say the node has *overflowed*. To repair this, we split the node into three parts:

   Left:

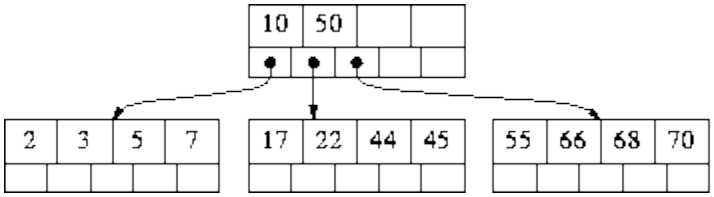
   The first $\boxtimes (p-1) / 2 \boxtimes$ values
   Middle:

   the middle value (which is at position $\boxtimes (p-1)/2 \boxtimes + 1$)
   Right:
   The remaining values
4. Move middle value to appropriate place in parent and make Left and Right the left and right children of middle value.
5. If there is no room in the parent, that is if it overflows, 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. Continue doing this until no overflow occurs, or until the root itself overflows. If the root overflows, 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's do a sequence of insertions into the following B-tree (p=5, so each node other than the root must contain between 2 and 4 values):



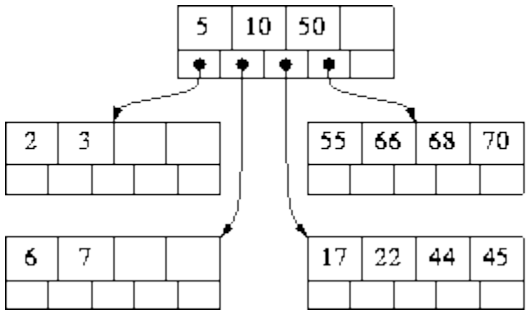
Insert 17: Add it to the middle leaf. No overflow, so we're done.



Insert 6: Add it to the leftmost leaf. That overflows, so we split it:

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

Left and Right become nodes; Middle is added to the node above with Left and Right as its children.

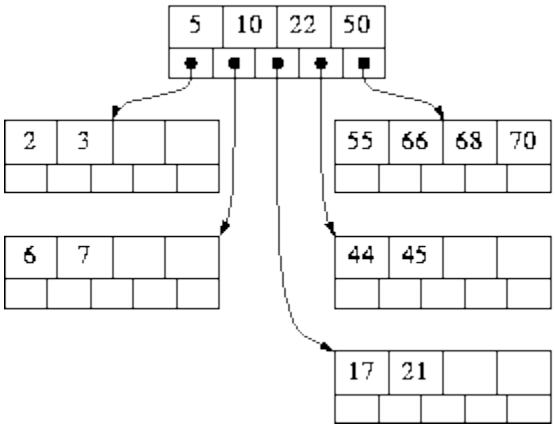


The node above (the root in this small example) does not overflow, so we are done.

Insert 21: Add it to the middle leaf. That overflows, so we split it:

- 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.

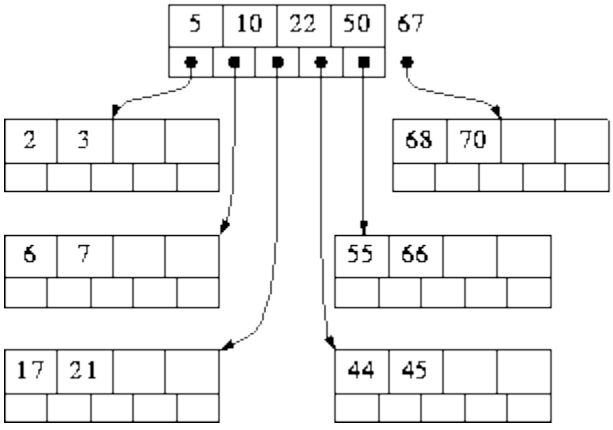


The node above (the root in this small example) does not overflow, so we are done.

Insert 67: Add it to the rightmost leaf. That overflows, so we split it:

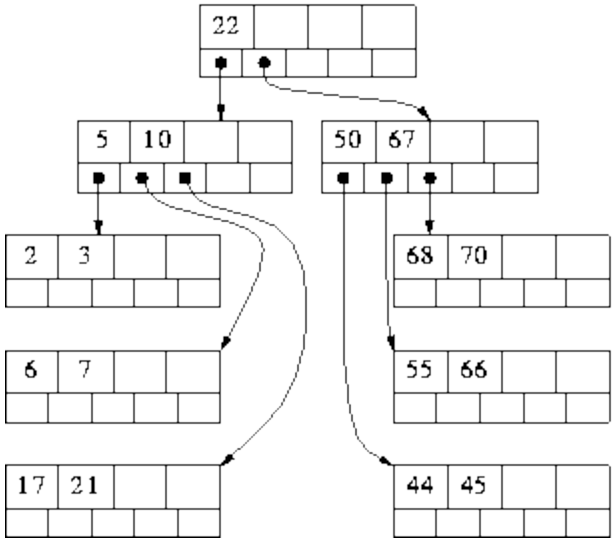
- Left = [55 66]
- Middle = 67
- Right = [68 70]

Left and Right become nodes; Middle is added to the node above with Left and Right as its children.



But now the node above does overflow. So it is split 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 were not the root, Middle would be 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're 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.

2. Deletion

To delete a value X from a B-tree of order p, following are the steps:

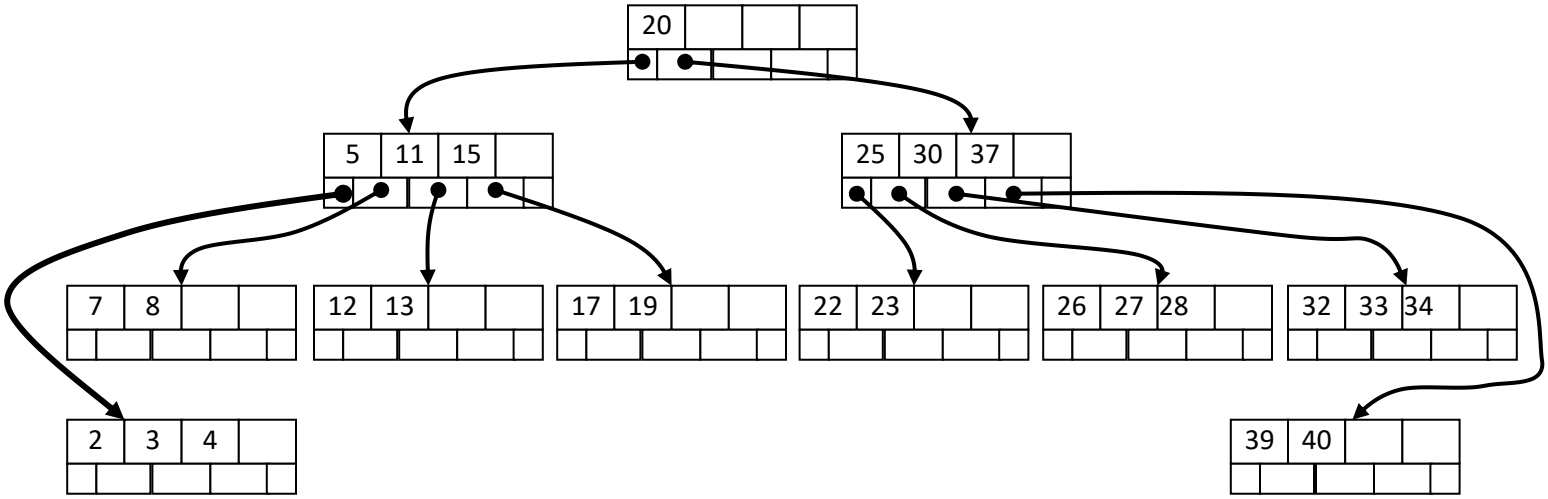
Let min = $\lceil (p/2) \rceil$ - 1 [Minimum number of keys in a node]

1. If X is a key in a leaf node that has at least min+1 number of keys, then remove X.
2. If X is a key in a leaf node that has exactly min number of keys, then
   a. If the target node from which the key is to be deleted has a right sibling with at least min+1 keys, then move a key from parent to the target node and move the smallest key of right sibling into parent. Otherwise, If the target node has a left sibling with at least min+1 keys, then move a key from parent to the target node and move the biggest key of left sibling into parent.
   b. If both siblings of target node have exactly min keys, then merge target node with any of its siblings by putting middle key from parent as median and delete X from merged node.
3. If X is a key in an internal node,
   a. If the target keys left child has at least min+1 keys, then its largest value should be moved to parent to replace the target key.
   b. If the target keys right child has at least min+1 keys, then its smallest value should be moved to parent to replace the target key.

c. If none of its children have at least min+1 keys, merge two children and key can be removed.

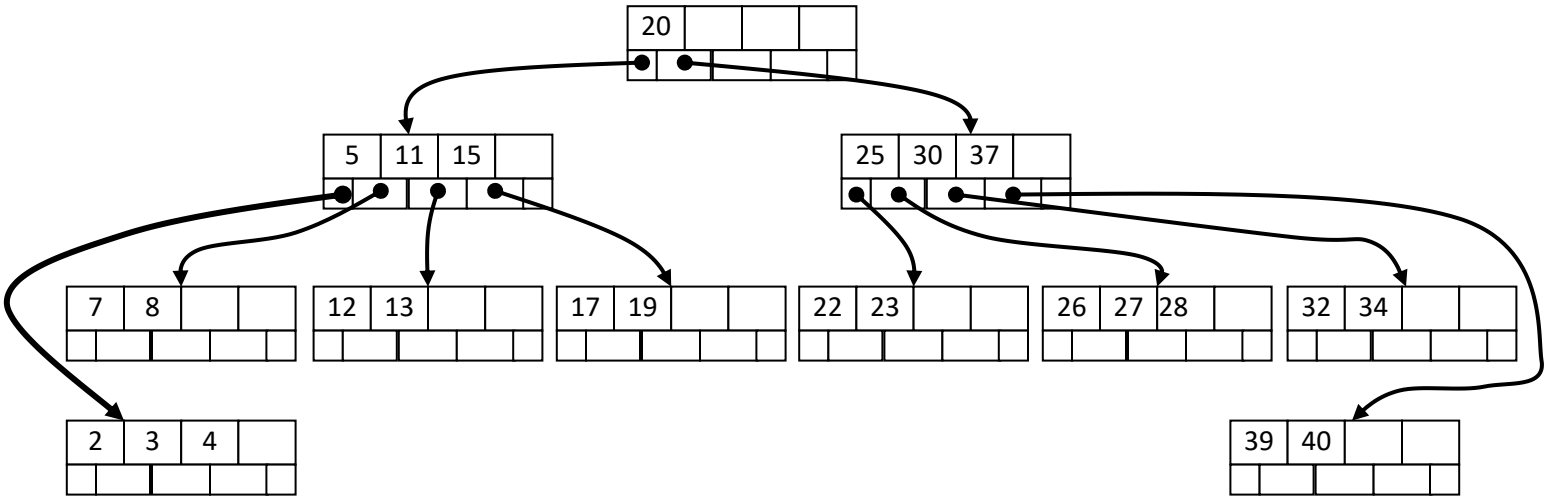
Consider the following b-tree of order 5. Following example shows how to delete the keys in the sequence 33, 32, 27, 5,4 from the tree.

Minimum number of keys in a node must be $\lceil (5/2) \rceil - 1 = 2$.

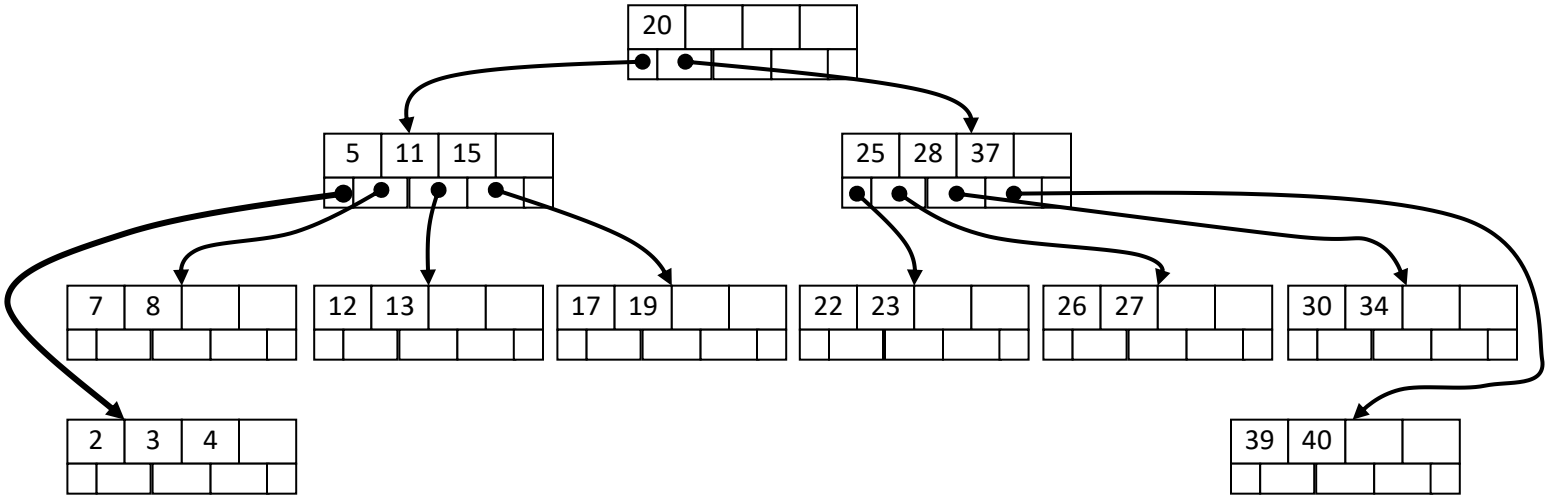


Assume that the element to be deleted is 33.

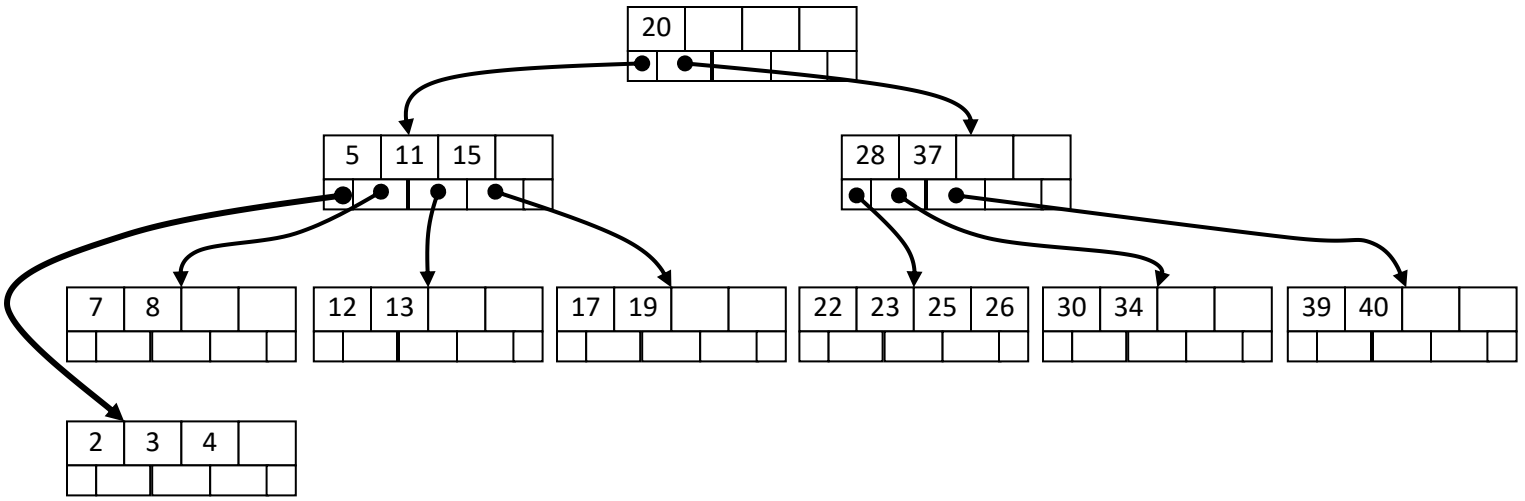
Since 33 is in a leaf node that has at least 3 (min+1) number of keys, it can be removed directly.



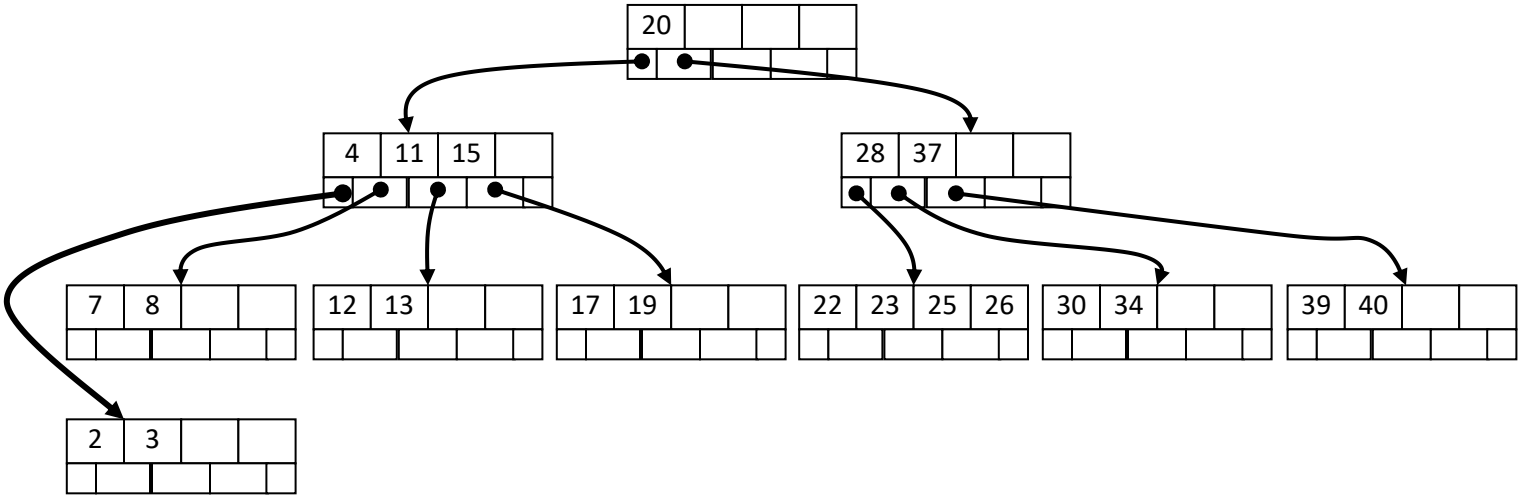
Next, element to be deleted is 32. 32 is in a node that has only 2 keys (min no of keys). According to the step 2.(a) of above given deletion algorithm, key 28 should be moved up to the parent and 30 should be moved down to the target node. Then, 32 can be removed from the target node. as shown in the following figure.



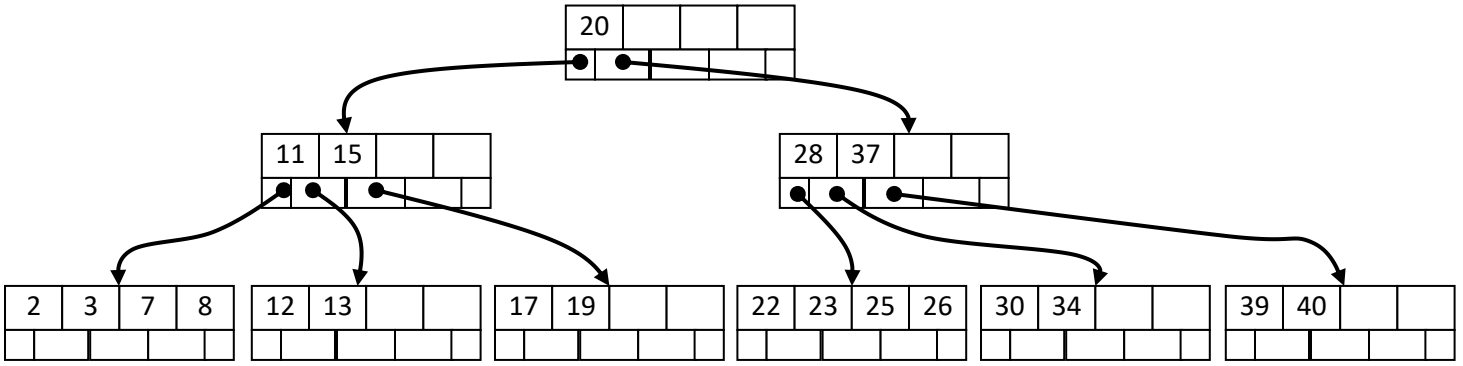
Now the element to be deleted is 27. 27 is in a leaf node which has2 keys. So, it can be removed directly. At the same time, none of its neighbor nodes have 3 keys to borrow from. According to the step 2.(b) of given deletion algorithm, merge target node with any of its siblings by putting middle key from parent as median and delete 27 from merged node. let us merge target node with its left sibling by putting the key 25 from parent as median and delete 27 from merged node.



Next, the element to be deleted is 5, which is in an internal node. As per the step 3.(a) of deletion algorithm, move the key 4from left child of target node to replace 5 in the target node.



Next element to be deleted is 4. None of the children of key 4 have 3 keys to borrow from. So, according to step 3(c), merge the child nodes of 4 and then 4 can be removed from target node.



B+ Tree

In a B-tree, pointers to data records exist at all levels of the tree. In a B+tree, all pointers to data records exist at the leaf-level nodes. A B+-tree can have less levels than the corresponding B-tree. In a B+Tree, all the keys are stored at the leaf level and the leaf nodes have pointers to connect each other. So, the data in leaf nodes can be accessed in a way similar to accessing data from a linked list.

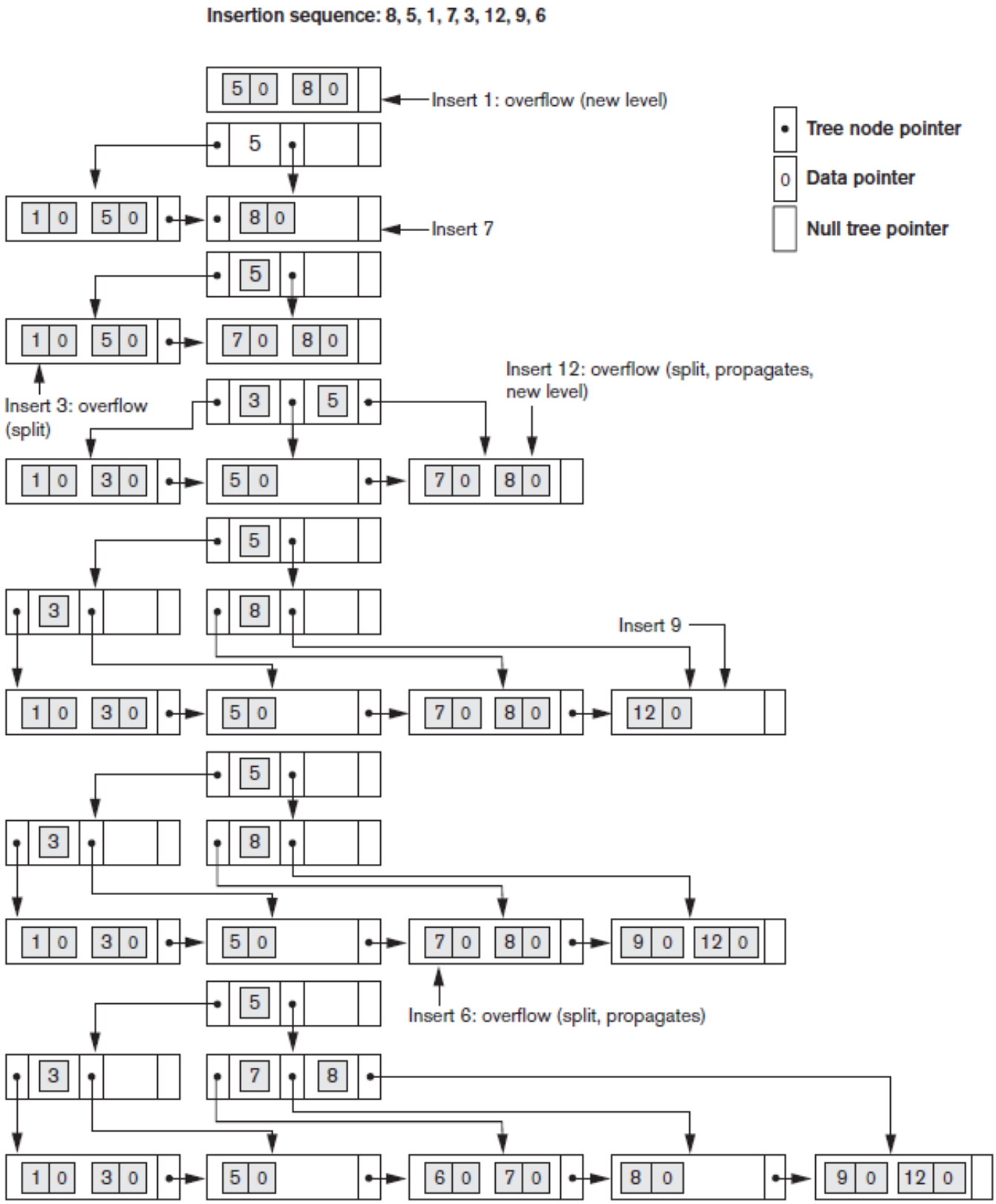
- Each internal node in the B+tree is of the form $<P_1, K_1, P_2, K_2, \ldots, K_{q-1}, , P_q>$ where $q \leq p$.
  Each P_i is a pointer to a successor. Each K_i is a key.
- Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
- For all key values X in the subtree pointed at by P_i, we have:
  $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i=1$; and $K_{i-1} < X$ for $i = q$.
- Each node has at most p successors and p-1 key values.
- Each internal node, except the root, has at least $\lceil (p/2) \rceil$ successors and $(\lceil (p/2) \rceil -1)$ keys . The root node has at least two successors unless it is the only node in the tree
- A node with q successors, $q \leq p$, has q-1 key values
- All leaf nodes are at the same level. Leaf nodes have pointers to connect to the next leaf node.
- Each leaf node is of the form <<K1, Pr1> , <K2, Pr2>, ..., <Kq-1, Prq-1>, Pnext> , whre q-1 < p. Each Pri is a pointer to the data record containing key Ki and Pnext points to the next leaf node of the B+-tree

1. Insertion

1. Find correct leaf L.
2. Put the key onto L.
   a. If L has enough space, done!
   b. else, split L into two nodes L and L1 and redistribute the entries evenly between L and L1. Copy the middle key to parent. The first $\lceil (p+1)/2 \rceil$ entries are kept in L and the remaining entries are moved to L2.

3. a. A parent needs to recursively Push-Up the middle key until the insertion is successful. i.e., If Parent has enough space, done!
b. Else split Parent and Redistribute entries evenly, and push up middle key.

Following figure illustrates the insertion into a B+-tree.

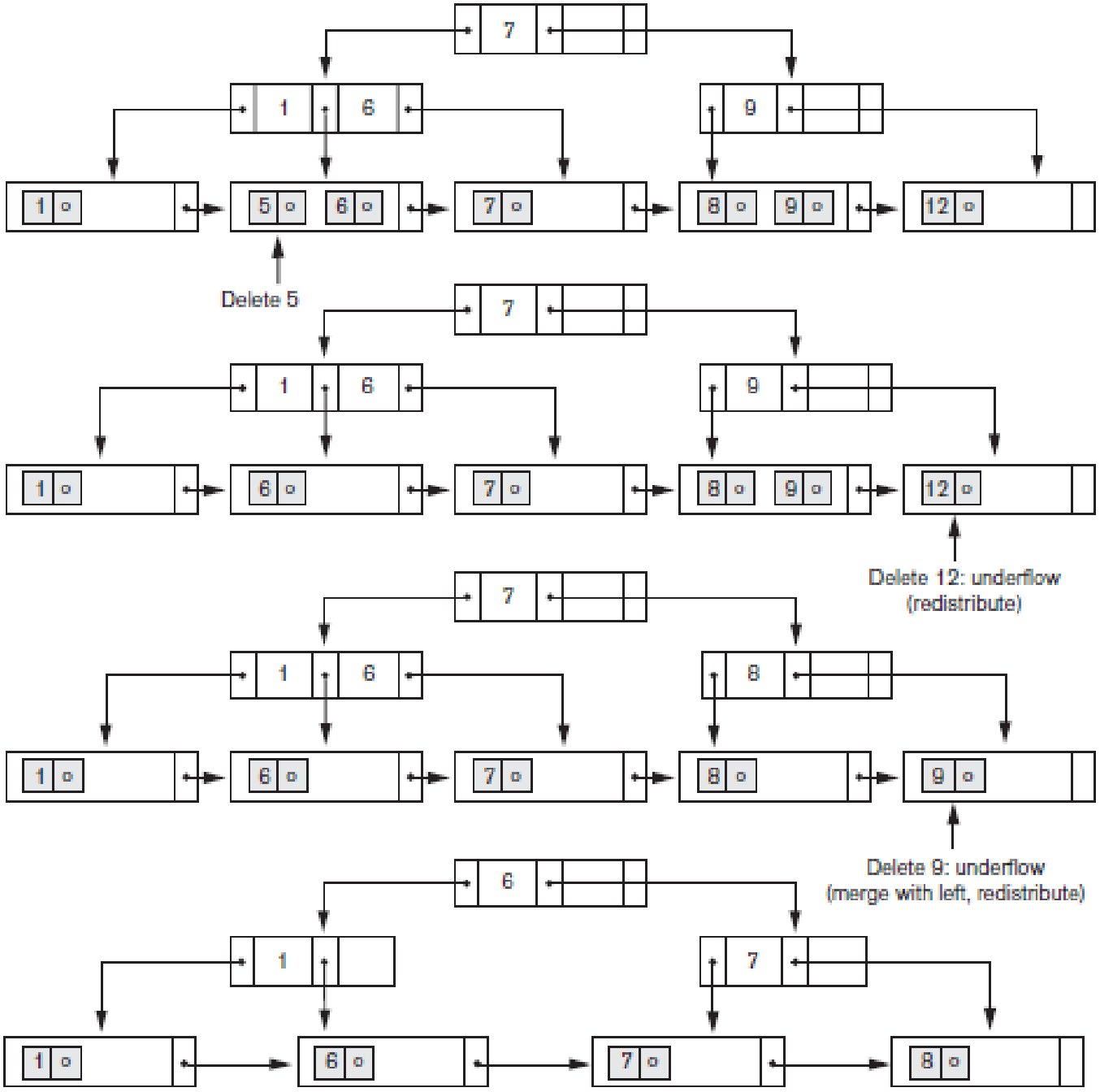

Insertion sequence: 8, 5, 1, 7, 3, 12, 9, 6
Insert 1: overflow (new level)
Insert 7
Insert 3: overflow (split)
Insert 12: overflow (split, propagates, new level)
Insert 9
Insert 6: overflow (split, propagates)
Tree node pointer
Data pointer
Null tree pointer

2. Deletion

1. Find the leaf node from which the key is to be deleted.
2. If the key to be deleted is only in the leaf node and not in the internal node
    a. If the target node has sufficient number of keys after deletion, then remove key from leaf node and stop.
    b. Else, if the target node has ($\lceil (p/2) \rceil$ -2) keys after deletion, **redistribute** entries with the left
    sibling; if this is not possible, redistribute with the right sibling.
    c. If the redistribution with siblings is not possible, merge the three nodes into two leaf nodes.
    d.If the underflow propagate to **internal** node, propagate and reduce the tree levels.
3. If the key to be deleted is in the internal node also, then the value to its left in the leaf node must replace it in the internal node. After deletion, if underflow occurs, handle it as given in step 2.

An illustration of deletion from a B+tree of order 3 is shown in the following figure.

Deletion sequence: 5, 12, 9



Average and Worst case Time complexities of operations

Binary Search Tree

Operation	Average	Worst
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

AVL Tree

Operation	Average	Worst
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

B-Tree

Operation	Average	Worst
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Heap

Operation	Average	Worst
FindMax	O(1)	O(1)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Applications of Trees

Trees are one of the most useful data structures in Computer Science. Some of the common applications of trees are:

1. Trees are used for implementing databases.
2. The file system in a computer i.e folders and all files, would be stored as a tree.
3. When searching for a word in a file, a tree is used. Because, the file would be indexed as a tree and for the same reason you get answers instantly.
4. The routers in a network heavily use trees for routing.
5. Most probably, map applications like Google maps uses trees (apart from using graphs) for all the locations in it. Then it would be very easy to answer questions like a) which is

the nearest restaurant to this place? b) List all schools which are present within 5 miles of radius from this place and so on