**JAVA WRAPPER CLASSES**  
  
Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.

The wrapper classes in java servers two primary purposes.

* To provide a mechanism to ‘wrap’ primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, Hashset, HashMap etc. collection.
* To provide an assortment of utility functions for primitives like converting primitive types to and from string objects, converting to various bases like binary, octal or hexadecimal, or comparing various objects.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

int x = 25;

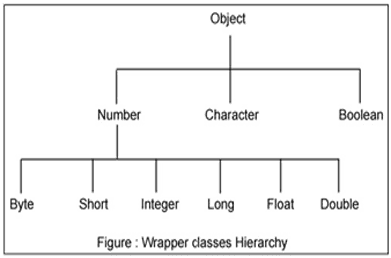
Integer y = new Integer(33);

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Below table lists wrapper classes in Java API with constructor details.

|  |  |  |
| --- | --- | --- |
| **Primitive** | **Wrapper Class** | **Constructor Argument** |
| boolean | Boolean | boolean or String |
| byte | Byte | byte or String |
| char | Character | char |
| int | Integer | int or String |
| float | Float | float, double or String |
| double | Double | double or String |
| long | Long | long or String |
| short | Short | short or String |

Below is wrapper class hierarchy as per Java API



As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in the constructor. For example to create Integer object we can have the following syntax.

Integer intObj = new Integer (25);

Integer intObj2 = new Integer ("25");

Here in we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

Integer intObj3 = new Integer ("Two");

The following discussion focuses on the Integer wrapperclass, but applies in a general sense to all eight wrapper classes.

The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

**BOXING AND UNBOXING**

Boxing in Java is the process of converting a primitive data type into its corresponding wrapper class. The wrapper classes are:

Byte, Short, Integer, Long, Float, Double, Character, and Boolean.

For example, boxing an int to an Integer is done as follows:

Java

int a = 5;

Integer b = a;

Unboxing is the opposite operation of boxing, and it converts a wrapper class object back to its primitive data type. For example, unboxing an Integer to an int is done as follows:

Java

Integer b = 5;

int a = b;

Boxing and unboxing are automatic in Java, and the compiler will perform them as needed. However, it is important to be aware of when boxing and unboxing occur, as they can affect the performance of your code.

Here are some of the benefits of boxing and unboxing:

They allow primitive data types to be used where objects are required.

They make code more readable and maintainable.

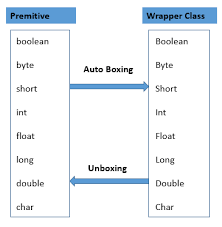
They can improve the performance of code by avoiding unnecessary type conversions.

Here are some of the drawbacks of boxing and unboxing:

They can make code slower and more memory intensive.

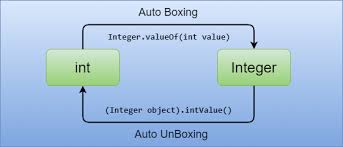
They can lead to unexpected behaviour if not used correctly.

Overall, boxing and unboxing are powerful features of Java that can be used to improve the readability, maintainability, and performance of your code. However, it is important to be aware of their drawbacks and use them carefully.



**AUTOBOXING AND AUTOUNBOXING**

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrappers whenever an object of the type is needed. There is no need to construct an object explicitly. Autoboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when the program requires its value. Furthermore, autoboxing and auto-unboxing significantly streamline the code of several algorithms, removing the tedium of manually boxing and unboxing values.



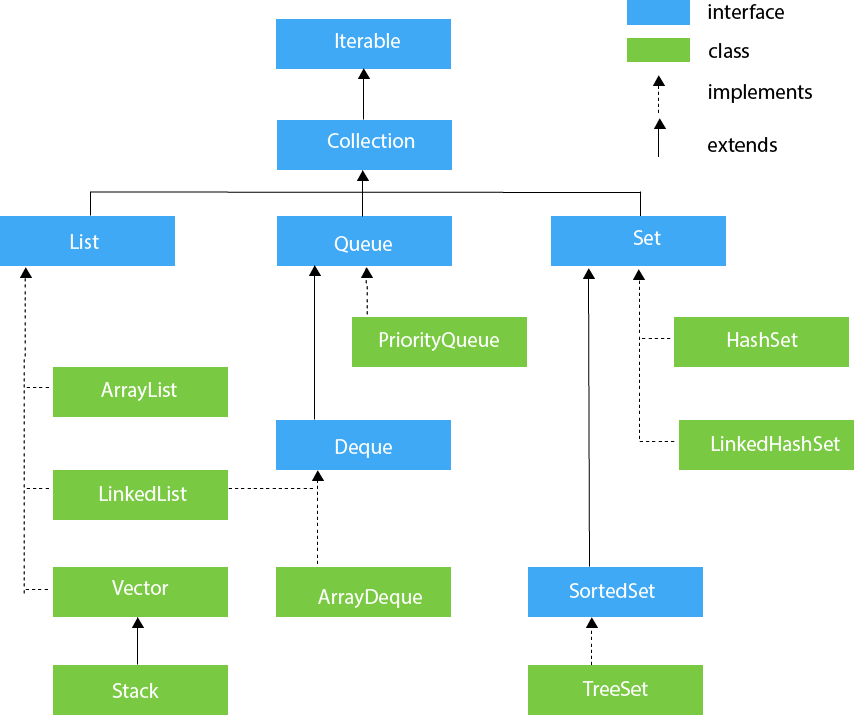
**AUTOBOXING**

* AutoBoxing is when Java compiler performs the automatic conversion of the primitive data types to the object of their corresponding wrapper classes
* Ex: converting an int to Integer
* The Java compiler applies autoboxing when a primitive value is: Passed as a parameter to a method that expects an object of the corresponding wrapper class.
* Assigned to a variable of the corresponding wrapper class.

**UNBOXING**

* It is just the opposite process of autoboxing,
* Unboxing is automatically converting an object of a wrapper type(Integer, for example) to its corresponding primitive (int) value.
* The Java compiler applies unboxing when an object of a wrapper class is: Passed as a parameter to a method that expects the value of the corresponding primitive data type,
* Assigned to a variable of the corresponding primitive type.

**HIERARCHY OF COLLECTION FRAMEWORK**

****

**List ADT**

Methods:

* size(): Returns the number of elements in a list
* isEmpty(): Returns a Boolean indicating whether the list is empty
* get(i): Returns the element of the list having index i; an error condition occurs if I is not in range [0,size()-1]
* set(i,e): Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range [0,size()-1]
* add(i,e): Inserts a new element e into the list so that it has index i ,moving all subsequent elements one index later in the list; an error condition occurs if i is not in range[0,size()-1]
* remove(i): Removes an element at index i, moving all the subsequent elements one index back in the list; an error condition occurs if i is not in range [0,size()-1.

**CREATING ARRAY LIST**

* Syntax: ArrayList<Type> arrayList = new ArrayList<>();
* By defining size: ArrayList<Type> arrayList = new ArrayList<>(3);

**List ADT Example**

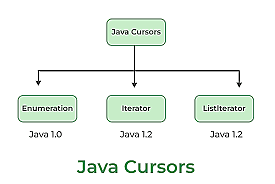
|  |  |  |
| --- | --- | --- |
| **Method** | **Return Value** | **List Contents** |
| Add(0,A) | - | (A) |
| Add(0,B) | - | (B,A) |
| Get(1) | A | (B,A) |
| Set(2,C) | “error” | (B,A) |
| Add(2,C) | - | (B,A,C) |
| Add(4,D) | “error” | (B,A,C) |
| Remove(1) | A | (B,C) |
| Add(1,D) | - | (B,D,C) |
| Add(1,E) | - | (B,E,D,C) |
| Get(4) | “error” | (B,E,D,C) |
| Add(4,F) | - | (B,E,D,C,F) |
| Set(2,G) | D | (B,E,G,C,F) |
| Get(2) | G | (B,E,G,C,F) |

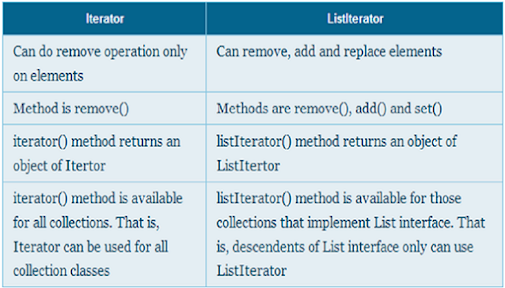
**UPDATED METHODS OF POSITIONAL LIST**

* addFirst(e): Inserts a new element a at the front of the list, returning the position of the new element.
* addLast(e): Inserts the element e at the back of the list, returning the position of the new element.
* addBefore(e)

**ITERATORS**

A Java cursor is an iterator, which is used to iterate or traverse or retrieve a Collection or Stream Object’s elements one by one.

****



**HELPER CLASSES IN INTERFACES IN JAVA**

A helper class in Java is a class that contains methods that are used by other classes to perform common tasks. Helper classes are often used to encapsulate common functionality, such as input/output, logging, or database access. Helper classes can also be used to provide utility methods, such as mathematical functions or string manipulation.

Helper classes can be either static or non-static. Static helper classes are classes that do not have any instance variables or methods. Non-static helper classes can have both instance variables and methods.

Helper classes can be used in interfaces in Java. An interface is a reference type that defines a set of abstract methods. Classes that implement an interface must implement all of the methods that are defined in the interface.

Helper classes can be used to provide implementations of the methods that are defined in an interface. This can be useful if the methods in the interface are complex or difficult to implement.

Here is an example of a helper class that implements an interface:

Java

public interface ICalculator {

int add(int a, int b);

int subtract(int a, int b);

int multiply(int a, int b);

int divide(int a, int b);

}

public class CalculatorHelper implements ICalculator {

@Override

public int add(int a, int b) {

return a + b;

}

@Override

public int subtract(int a, int b) {

return a - b;

}

@Override

public int multiply(int a, int b) {

return a \* b;

}

@Override

public int divide(int a, int b) {

return a / b;

}

}

The CalculatorHelper class implements the ICalculator interface. The CalculatorHelper class can be used to provide implementations of the methods that are defined in the ICalculator interface.

Here is an example of how to use the CalculatorHelper class:

Java

ICalculator calculator = new CalculatorHelper();

int result = calculator.add(1, 2);

System.out.println(result); // 3

The CalculatorHelper class is a useful way to encapsulate common functionality and provide utility methods. Helper classes can also be used to provide implementations of the methods that are defined in an interface.  
  
**CREATING VECTORS**

* Synatx: Vector<Type> vector\_name = new Vector<type> ();//declaring without size.

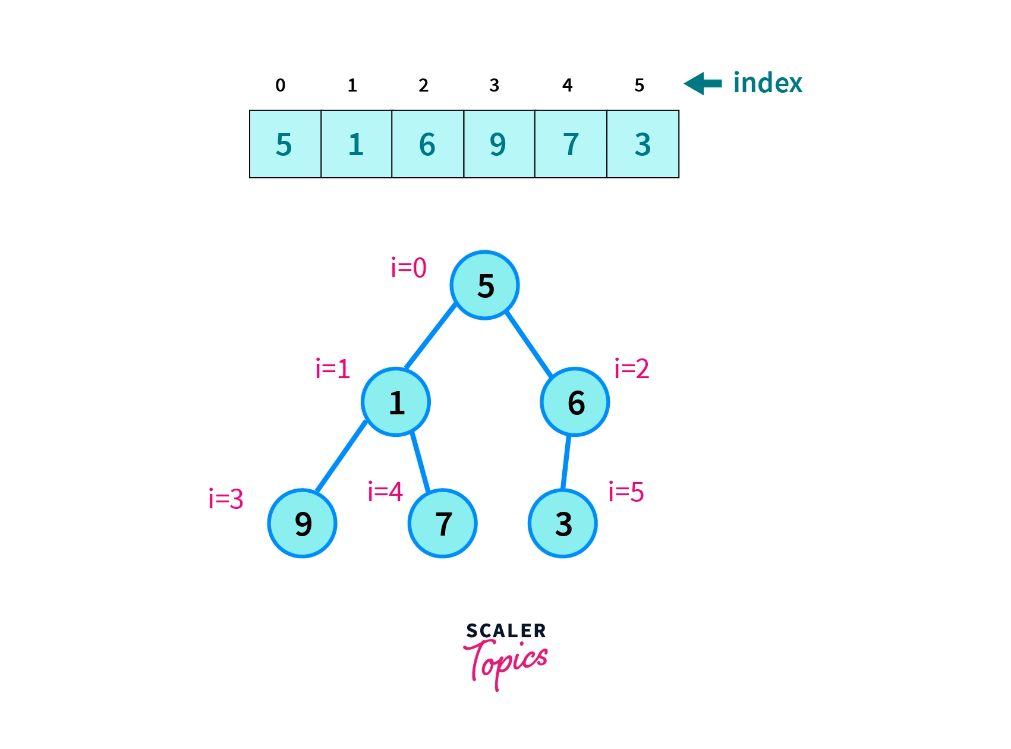
**STACKS**

* A Stack is a data structure that follows the Last-In-First-Out(LIFO) principle.
* Elements are inserted and removed from the top of the stack
* The fundamental operations are “push” to add an item to the top of the stack and “pop” to remove the top item.
* **STACK ABSTRACT DATA TYPE**
* Push(e):Adds an element e to the top of the stack
* Pop(): Removes and returns the top element from the stack
* Top():Returns the top element of the stack without removing it
* Size():Returns the number of elements in the stack
* isEmpty(): Returns a Boolean indicating whether the stack is empty or not

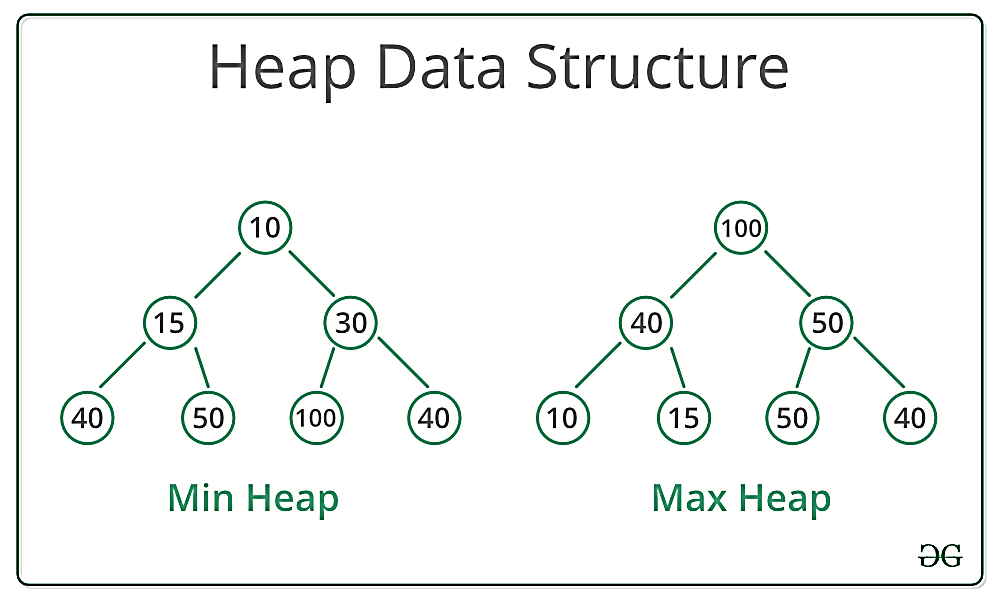
**QUEUES**

* The Queue interface is present in the java.util package and extends the Collection interface and is used to hold the elements about to be processed in FIFO.
* The Queue interface provides several methods for adding, removing, and inspecting elements in the queue.
* Methods:  
  1) add(element): Adds an element to the rear of the queue. If queue is full, it throws exception  
  2) offer(element): Adds an element to the rear of the queue. If queue is full, it throws exception  
  3) remove(): Removes and returns an element at front , if queue is empty, throws exception   
  4) poll(): Removes and returns the element at front. If queue is empty, throws exception  
  5) element()  
  6) peek()

**BINARY TREES**

* A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are   
  as far left as possible.  
  
* Max no. of nodes in a tree:2h+1-1
* Min no. of nodes in a tree:h+1 (h=level)

**HEAP (COMPLETE BINARY TREE)**

* Heap data structure is a complete binary tree that satisfies the heap property.
* 1) Min Heap: Value of the root(parent) node is the lowest amongst the values of all the other nodes(child nodes). (This pattern follows for all other nodes in the tree).  
  2) Max Heap: Value of the root(parent) node is the highest amongst the values of all the other nodes(child nodes). (This pattern follows for all other nodes in the tree).  
  
* While inserting a new node inside a heap the new node tries to rearrange itself inside the heap as per the primary condition of the heap.  
  After doing so successfully, we say the heap is heapified.
* While deleting nodes from a heap we can only delete the root node. The remaining nodes rearrange itself inside the heap as per the primary condition of the heap. After doing so successfully, we say the heap is heapified.

**PRIORITY QUEUE**

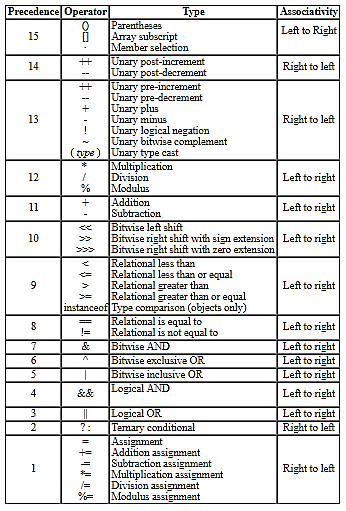
* Priority Queues is an abstract data type that allows arbitrary element insertion and allows the removal of the element that has first priority
* When an element is added to a priority queue, the user designates its priority by providing a key.
* Methods:  
  1) insert(k,v)  
  2) min()  
  3) removeMin()  
  4) size()  
  5) isEmpty()

**ALGEBRAIC EXPRESSION (INFIX, PREFIX, POSTFIX)**

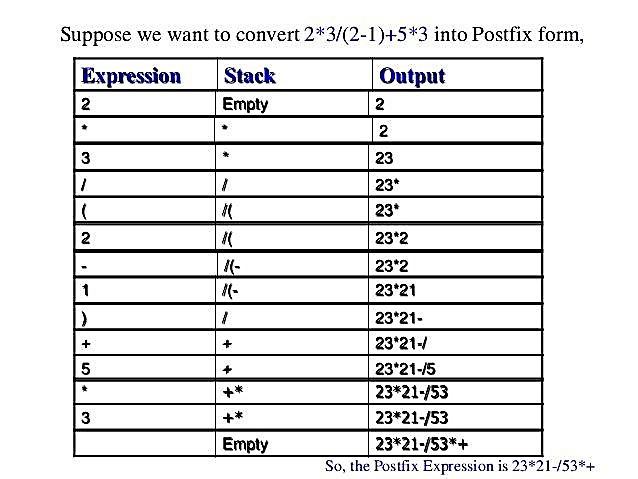
* An algebraic expression is a legal combination of operands and operators
* Operand is a quantity on which the mathematical operation is performed.
* Normal algebraic expression: - Infix expression
* Operators should be pushed to stack.
* Operands should be moved to output.
* Postfix: Operators will be displayed after the operands in the output as per their priority order.
* Prefix: Operators will be displayed before the operands in the output as per their priority order.

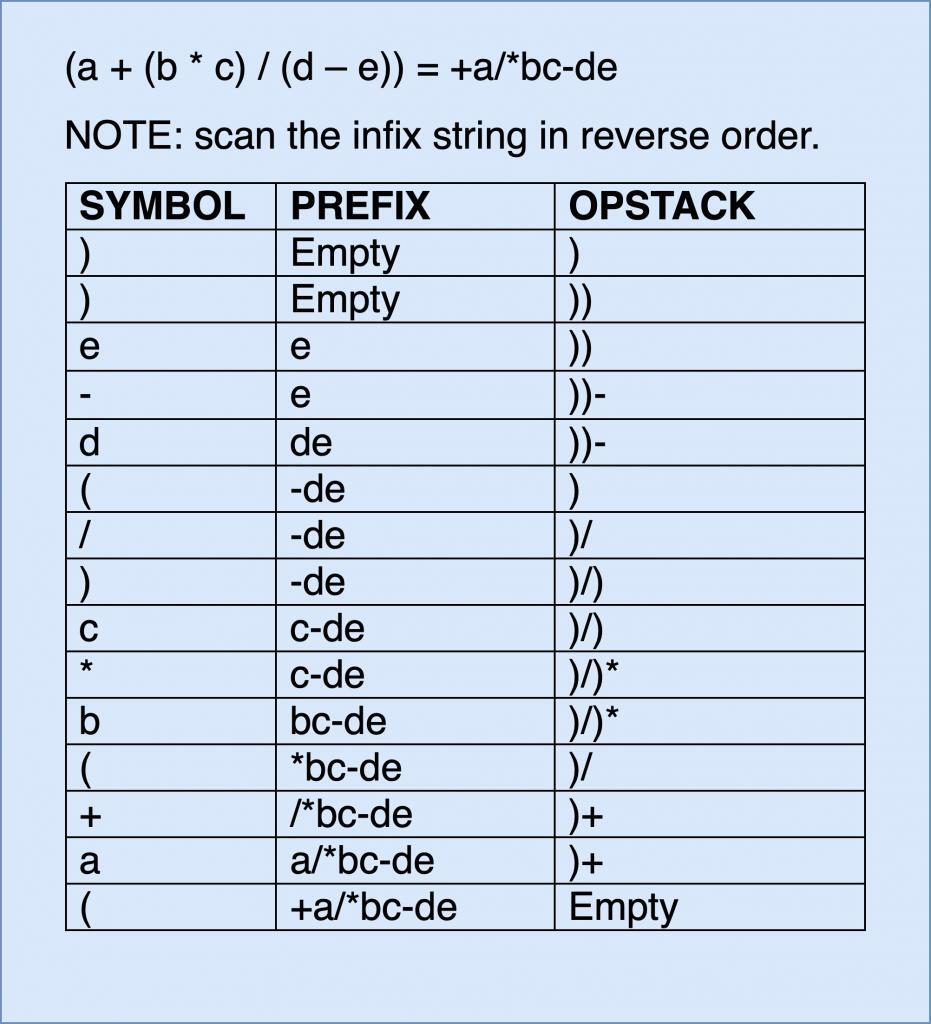
**OPERATOR PRIORITIES**

Examples of infix to prefix and postfix.

Priority: (),^,/,\*,+,-  
  


|  |  |  |
| --- | --- | --- |
| **Infix** | **PostFix** | **PreFix** |
| a+b | ab+ | +ab |
| (a+b)\*(c+d) | ab+cd+\* | \*+ab+cd |
| a-b/(c\*d^e) | abcde^\*/- | -a/b\*c^de |

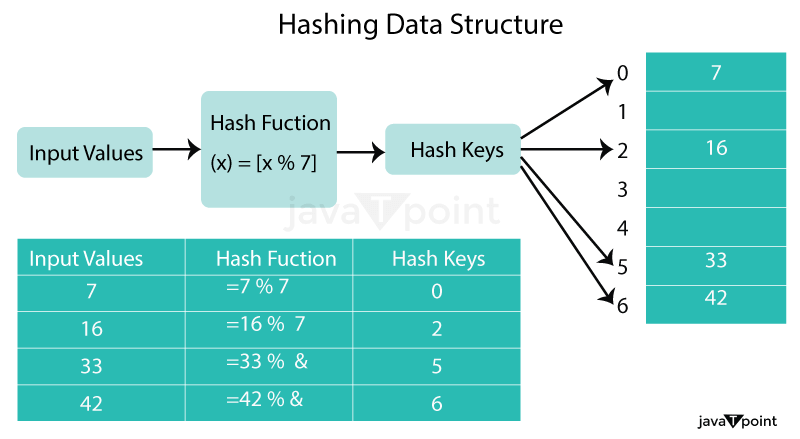




Ex questions: Convert the following expressions:

1) a+(b/c)  
2) ((a-(b+c))\*d)/(e+f)

**HASHING**



The process of hashing can be broken down into three steps:

* Input: The data to be hashed is input into the hashing algorithm.
* Hash Function: The hashing algorithm takes the input data and applies a mathematical function to generate a fixed-size hash value. The hash function should be designed so that different input values produce different hash values, and small changes in the input produce large changes in the output.
* Output: The hash value is returned, which is used as an index to store or retrieve data in a data structure.

Hash Function: A hash function is a type of mathematical operation that takes an input (or key) and outputs a fixed-size result known as a hash code or hash value. The hash function must always yield the same hash code for the same input in order to be deterministic. Additionally, the hash function should produce a unique hash code for each input, which is known as the hash property.

There are different types of hash functions, including:

Division method:

This method involves dividing the key by the table size and taking the remainder as the hash value. For example, if the table size is 10 and the key is 23, the hash value would be 3 (23 % 10 = 3).[H(k) = k%m (OR) H(k) = (k%m)+1 where m=table size, k=Hash key]

Multiplication method:

This method involves multiplying the key by a constant and taking the fractional part of the product as the hash value. For example, if the key is 23 and the constant is 0.618, the hash value would be 2 (floor(10\*(0.61823 - floor(0.61823))) = floor(2.236) = 2).

Universal hashing:

This method involves using a random hash function from a family of hash functions. This ensures that the hash function is not biased towards any particular input and is resistant to attacks.

Mid Square Method:

A good hash function for numerical values is the mid-square method. The mid-square method squares the key value, and then takes the middle r bits of the result, giving a value in the range 0 to 2r-1. This works well because most or all bits of the key value contribute to the result.

**Collision Resolution Strategies:**

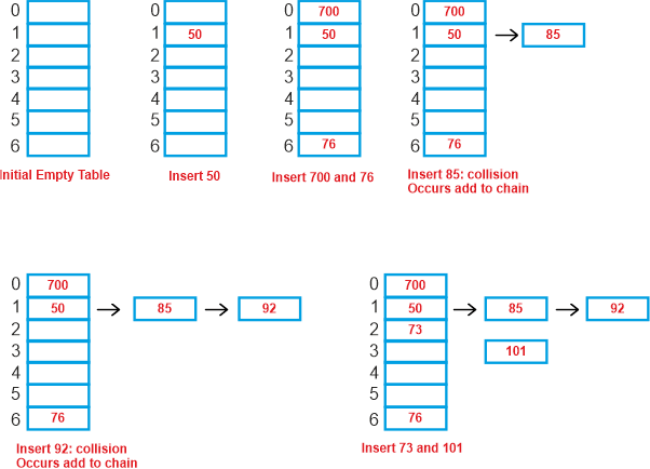
If two keys map on the same hash table index then we have a collision. As the number of elements in the tables increases the likelihood of a collision increases, so make the table as large as practical. Collisions may still happen, so we need a collision resolution strategy.

Two approaches:

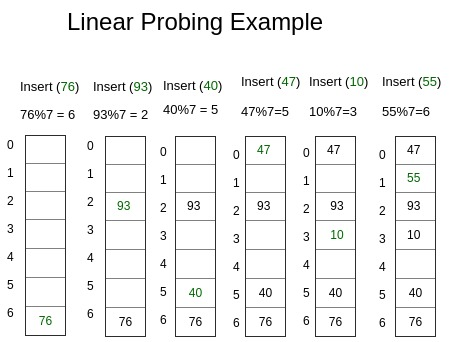
1) Separate chaining [using Linked-List]: Chain together several keys/entries in each position

2) Open addressing:(Probing, Double Hashing)

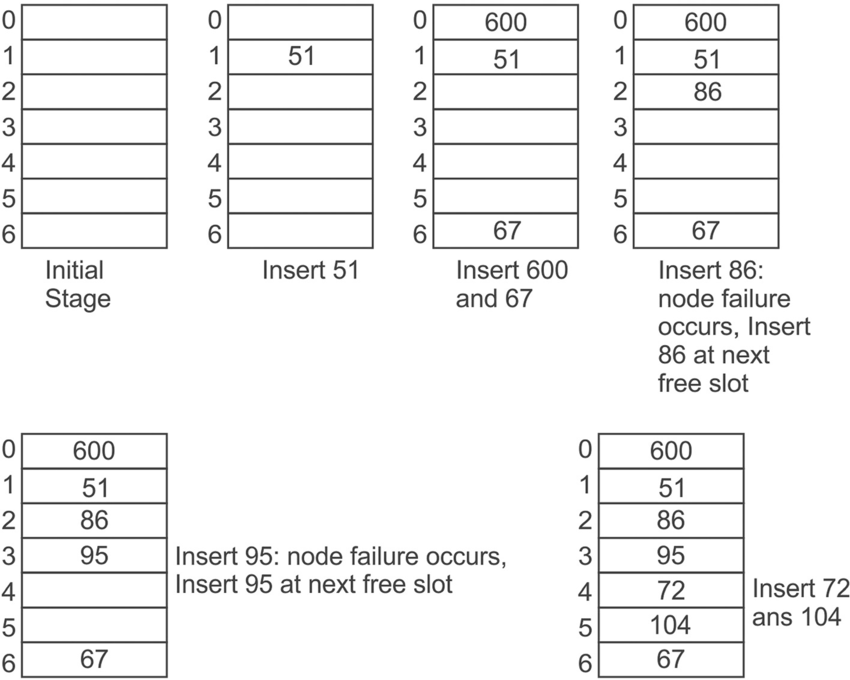
* **Separate Chaining**
* **Linear Probing** [H(k) = k%size{Normal} => H’(k) = (k%size) + 1{After Collision}]{Primary Clustering}
* **Quadratic Probing** [H(k) = k%size{Normal} => H’(k) = (k+(index)²)%size ;kϵ(1,size-1){After Collision}]{Secondary Clustering}
* **Double Hashing** [H(k) = k%size{Normal} => H’(k) = L-(k%L)=> the H’(k) value must be added to the H(k) value to find the position of the key to be inserted{After Collision} where L<size and L is a prime number that is closest to the value of size]
* **ReHashing** [H(k) = k%SIZE where SIZE = a prime number that is greater than and closest to the value of size\*2]

****

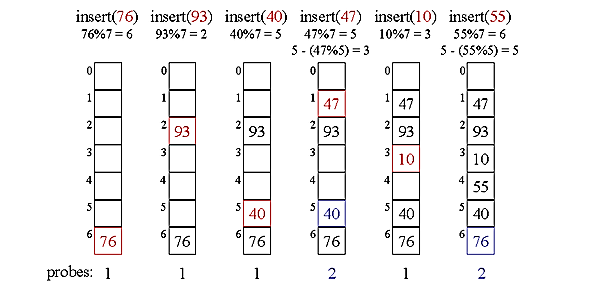
**Separate Chaining**

****

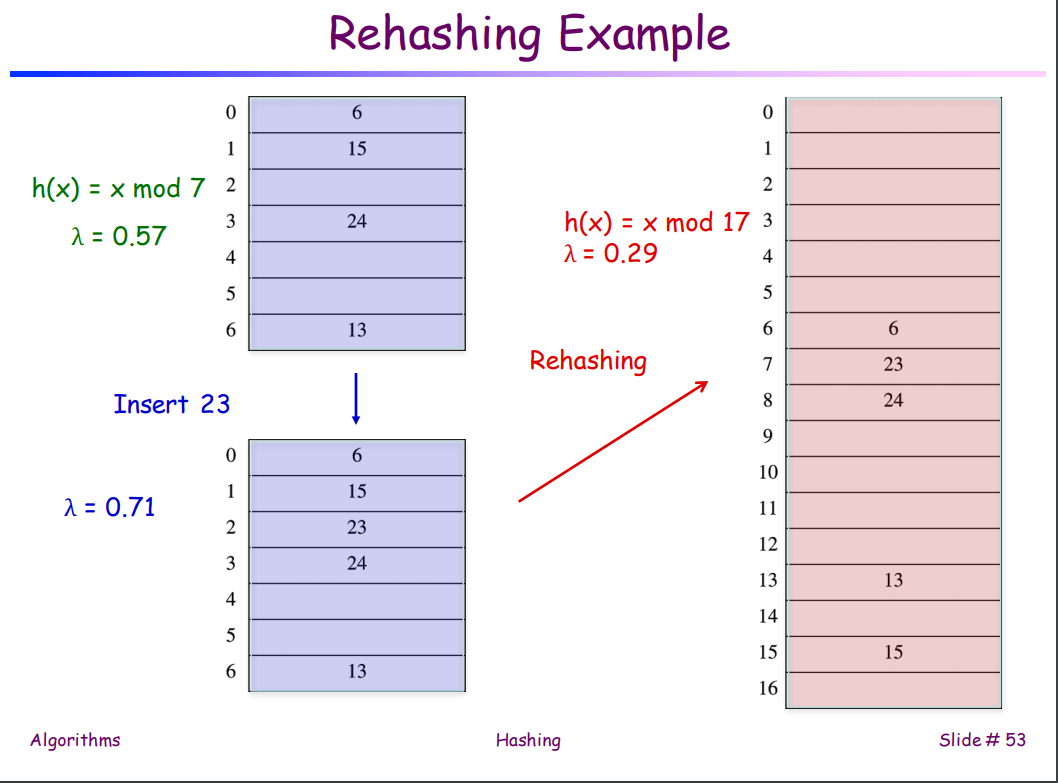
**Linear Probing**

****

**Quadratic Probing**

****

**Double Hashing**

****

**ReHashing**

**Applications of Hashing**

* Compilers use hash tables to keep track of declared variables
* A hash table can be used for online spelling checkers
* Games

**TRIES**

**Preprocessing Strings**

• Preprocessing the pattern speeds up pattern matching queries– After preprocessing the pattern, KMP’s algorithm performs pattern matching in time proportional to the text size

• If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern

• A trie is a compact data structure for representing a set of strings, such as all the words in a text– A tries supports pattern matching queries in time proportional to the pattern size  
  
**Standard Tries**

• The standard trie for a set of strings S is an ordered tree such that:

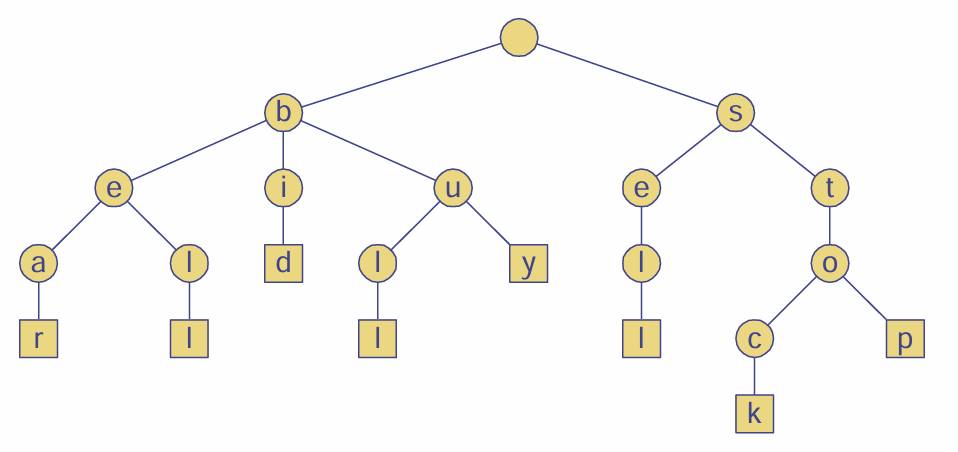
–Each node but the root is labeled with a character

–The children of a node are alphabetically ordered

–The paths from the external nodes to the root yield the strings of S

• Example: standard trie for the set of strings

S = { bear, bell, bid, bull, buy, sell, stock, stop }



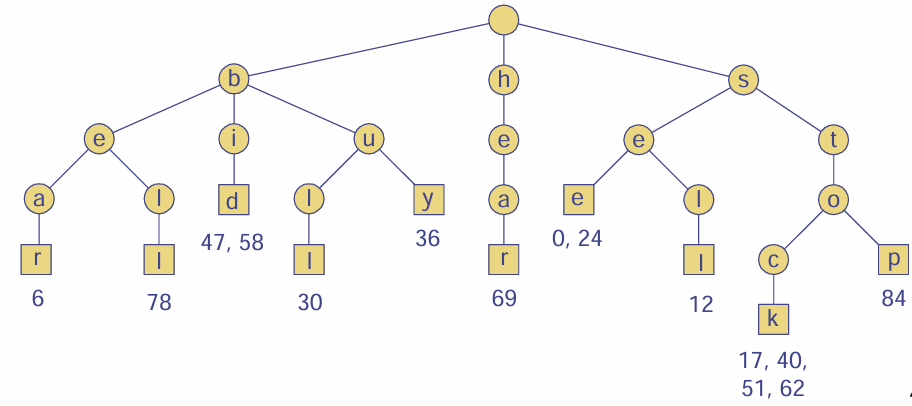
**Analysis of Standard Tries**

• A standard trie uses O(n) space and supports searches, insertions and deletions in time O(dm), where:

* n total size of the strings in S
* m size of the string parameter of the operation
* d size of the alphabet

**Word Matching with a Trie**

We insert the words of the text into a trie. Each leaf stores the occurrences of the associated word in the text.



**Binary Search Trees (BST)**

* It is a type of binary tree where each node can have atmost two child nodes
* In this type of tree the values of the nodes in the left subtree is lesser than the root value and the values of the nodes of the right subtree are greater than the root value.

**Tree Traversals:**

* **Inorder Traversal:** (Left->Root->Right) [Ascending Order] {1,3,6,8,10,14}
* **Preorder Traversal:** (Root->Left->Right) {8,3,1,6,10,14} [Useful for creating a copy of the tree]
* **Postorder Traversal:** (Left->Right->Root) {1,6,3,10,14,8} [Useful for deleting a tree]

**Deletion**

Case-1: Deleting a leaf node

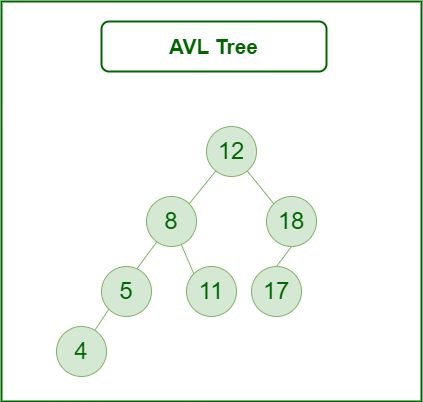
Case-2: Deleting a node having single child node

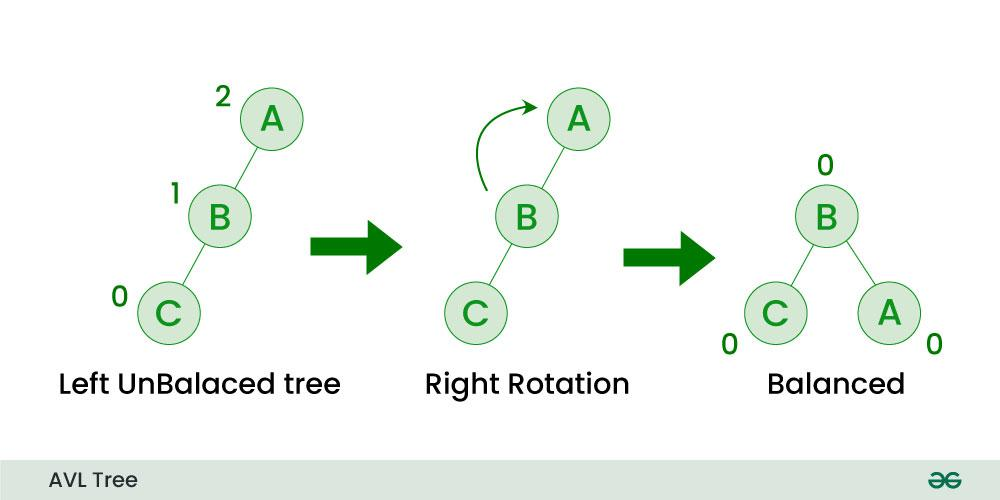
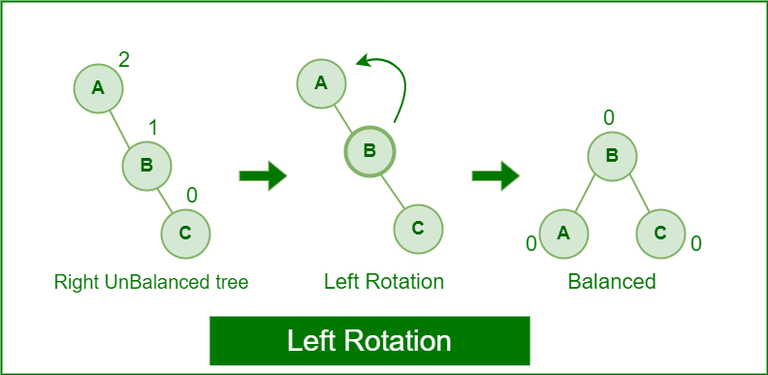
Case-3: Deleting a node having two child nodes  
 (i) If the node being deleted is an intermediary node

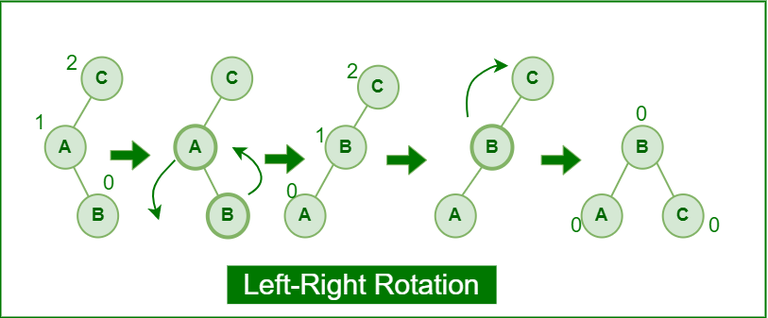
(ii) If the node being deleted is the root node (Replace the root as the highest value node from the left subtree OR replace the root as the lowest value node from the right subtree)

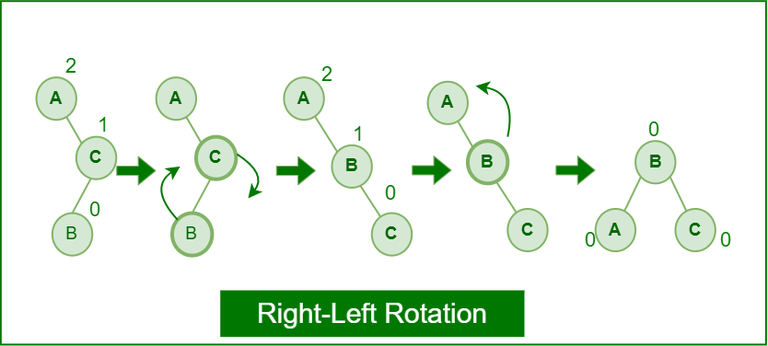
**AVL Tree**

* It is a type of binary search tree with the concept of weight
* Weight of a node (Balance Factor[N]) = Height of left subtree(N) – Height of right subtree(N)
* A node in a tree is height-balanced if the height of its subtrees differ by no more than 1 i.e., if the subtrees have heights h1 and h2 then |h1-h2|<=1.
* Weight of a node must be among the following three values in order for the tree to be balanced: \*\*{-1,0,1}\*\*
* **1)** **Left Rotation(LL Rotation):** When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2  
  **2)** **Right Rotation(RR Rotation):** When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2.  
  **3)** **Left-Right Rotation -> Left-Left:** Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.  
  **4)** **Right-Left Rotation -> Right-Right**: As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.









**Red Black Tree**

Red Black Tree is a self-balancing binary tree in which all the nodes have the following properties:

**Property-1:** Every node has a colour either red or black  
**Property-2:** Root node of the tree is always black  
**Property-3:** There are no two adjacent red nodes. In other words, a red node cannot have a red parent or red child.  
**Property-4:** All paths from the root node to a leaf node has equal number of black nodes.

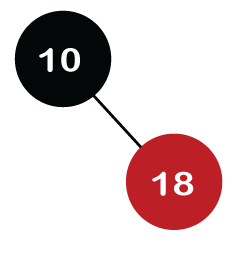
* If tree is empty then create a new node as root node with its colour as black.
* If tree is not empty create a new node as leaf node with its colour as red.
* If parent of new node is black, then exit.
* If parent of new node is red then check the colour of parent’s sibling of new node.  
  (a) If colour is black or null node is present, then do suitable rotation and re-colour.  
  (b) If colour is red, then re-colour and also check if parent’s parent of new node is not root node, then re-colour it and recheck.  
    
  (OR)
* It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
* This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
* In the Red-Black tree, the root node is always black in color.
* In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
* If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
* Every path from a node to any of its descendant's NIL node should have same number of black nodes.

**How does a Red-Black Tree ensure balance:** Basic principle behind the balancing in Red-Black Tree

**Let's understand the insertion in the Red-Black tree.**

**10, 18, 7, 15, 16, 30, 25, 40, 60  
  
Step 1:** Initially, the tree is empty, so we create a new node having value 10. This is the first node of the tree, so it would be the root node of the tree. As we already discussed, that root node must be black in color, which is shown below:  

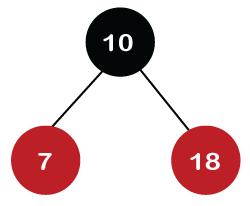

**Step 2:** The next node is 18. As 18 is greater than 10 so it will come at the right of 10 as shown below.



We know the second rule of the Red Black tree that if the tree is not empty then the newly created node will have the Red color. Therefore, node 18 has a Red color, as shown in the below figure:

Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. In the above figure, the parent of the node is black in color; therefore, it is a Red-Black tree.

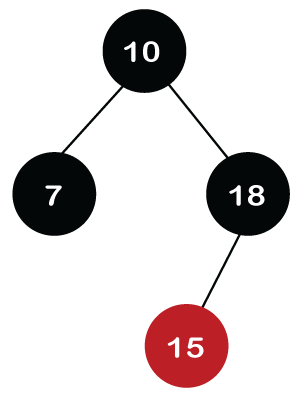
**Step 3:** Now, we create the new node having value 7 with Red color. As 7 is less than 10, so it will come at the left of 10 as shown below.



Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. As we can observe, the parent of the node 7 is black in color, and it obeys the Red-Black tree's properties.

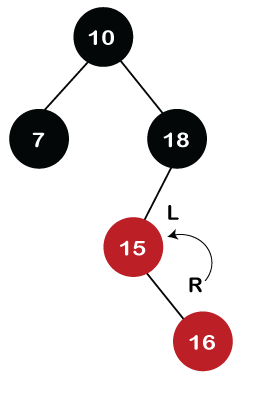
**Step 4:** The next element is 15, and 15 is greater than 10, but less than 18, so the new node will be created at the left of node 18. The node 15 would be Red in color as the tree is not empty.

The above tree violates the property of the Red-Black tree as it has Red-red parent-child relationship. Now we have to apply some rule to make a Red-Black tree. The rule 4 says that **if the new node's parent is Red, then we have to check the color of the parent's sibling of a new node.** The new node is node 15; the parent of the new node is node 18 and the sibling of the parent node is node 7. As the color of the parent's sibling is Red in color, so we apply the rule 4a. The rule 4a says that we have to recolor both the parent and parent's sibling node. So, both the nodes, i.e., 7 and 18, would be recolored as shown in the below figure.



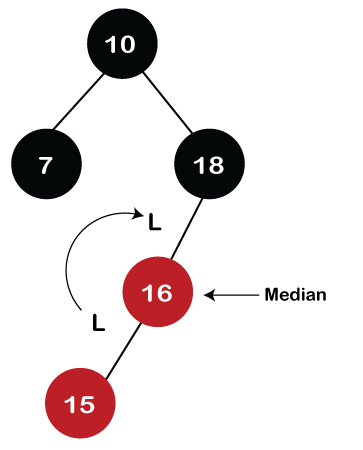
We also have to check whether the parent's parent of the new node is the root node or not. As we can observe in the above figure, the parent's parent of a new node is the root node, so we do not need to recolor it.

**Step 5:** The next element is 16. As 16 is greater than 10 but less than 18 and greater than 15, so node 16 will come at the right of node 15. The tree is not empty; node 16 would be Red in color, as shown in the below figure:

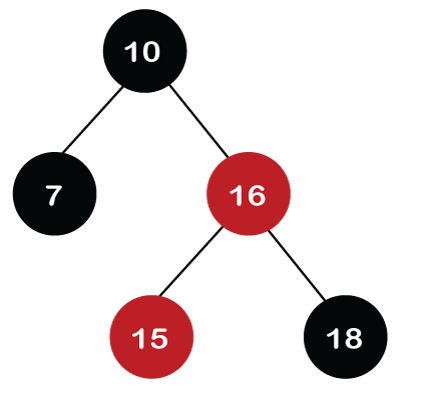


In the above figure, we can observe that it violates the property of the parent-child relationship as it has a red-red parent-child relationship. We have to apply some rules to make a Red-Black tree. Since the new node's parent is Red color, and the parent of the new node has no sibling, so rule 4a will be applied. The rule 4a says that some rotations and recoloring would be performed on the tree.

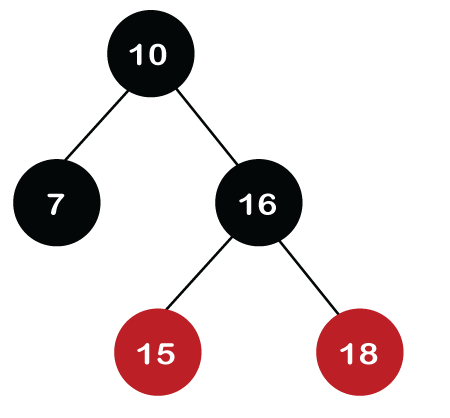
Since node 16 is right of node 15 and the parent of node 15 is node 18. Node 15 is the left of node 18. Here we have an LR relationship, so we require to perform two rotations. First, we will perform left, and then we will perform the right rotation. The left rotation would be performed on nodes 15 and 16, where node 16 will move upward, and node 15 will move downward. Once the left rotation is performed, the tree looks like as shown in the below figure:



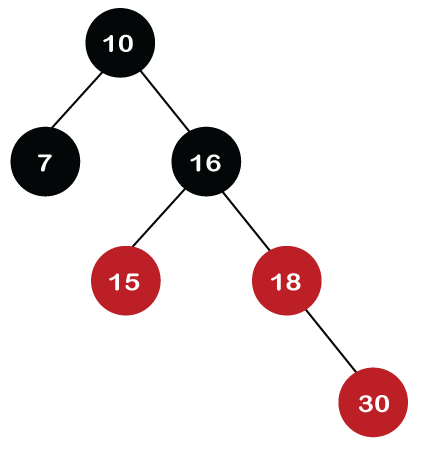
In the above figure, we can observe that there is an LL relationship. The above tree has a Red-red conflict, so we perform the right rotation. When we perform the right rotation, the median element would be the root node. Once the right rotation is performed, node 16 would become the root node, and nodes 15 and 18 would be the left child and right child, respectively, as shown in the below figure.



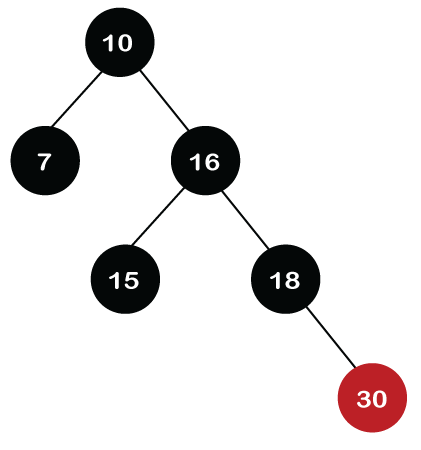
After rotation, node 16 and node 18 would be recolored; the color of node 16 is red, so it will change to black, and the color of node 18 is black, so it will change to a red color as shown in the below figure:



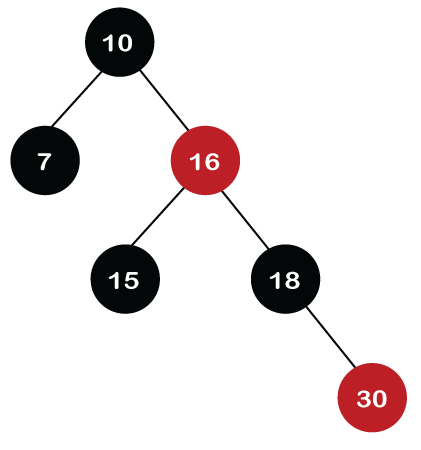
**Step 6:** The next element is 30. Node 30 is inserted at the right of node 18. As the tree is not empty, so the color of node 30 would be red.



The color of the parent and parent's sibling of a new node is Red, so rule 4b is applied. In rule 4b, we have to do only recoloring, i.e., no rotations are required. The color of both the parent (node 18) and parent's sibling (node 15) would become black, as shown in the below image.

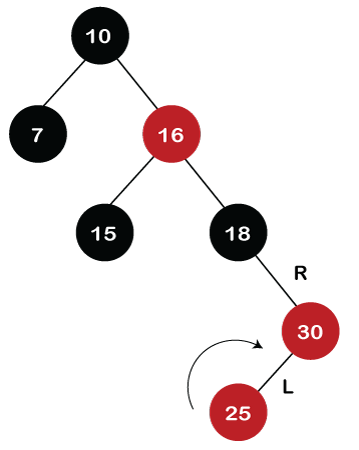


We also have to check the parent's parent of the new node, whether it is a root node or not. The parent's parent of the new node, i.e., node 30 is node 16 and node 16 is not a root node, so we will recolor the node 16 and changes to the Red color. The parent of node 16 is node 10, and it is not in Red color, so there is no Red-red conflict.

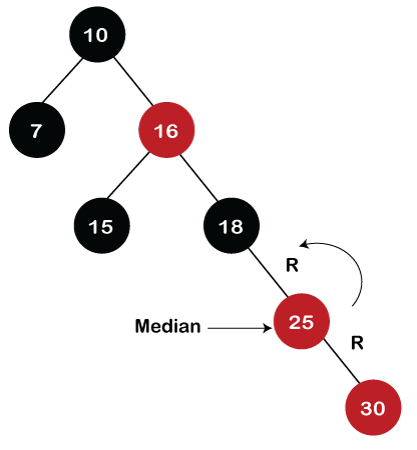


**Step 7:** The next element is 25, which we have to insert in a tree. Since 25 is greater than 10, 16, 18 but less than 30; so, it will come at the left of node 30. As the tree is not empty, node 25 would be in Red color. Here Red-red conflict occurs as the parent of the newly created is Red color.

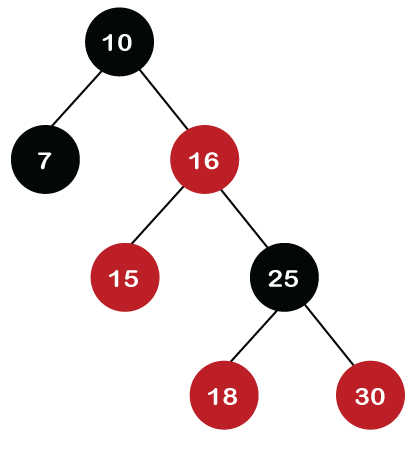
Since there is no parent's sibling, so rule 4a is applied in which rotation, as well as recoloring, are performed. First, we will perform rotations. As the newly created node is at the left of its parent and the parent node is at the right of its parent, so the RL relationship is formed. Firstly, the right rotation is performed in which node 25 goes upwards, whereas node 30 goes downwards, as shown in the below figure.



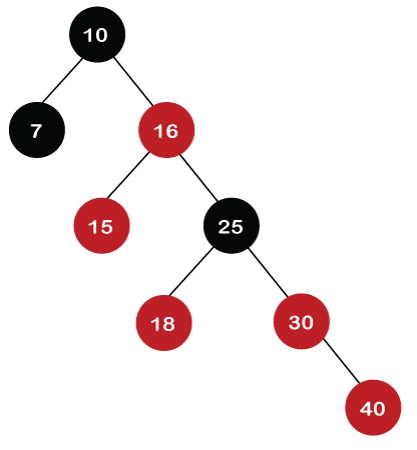
After the first rotation, there is an RR relationship, so left rotation is performed. After right rotation, the median element, i.e., 25 would be the root node; node 30 would be at the right of 25 and node 18 would be at the left of node 25.



Now recoloring would be performed on nodes 25 and 18; node 25 becomes black in color, and node 18 becomes red in color.



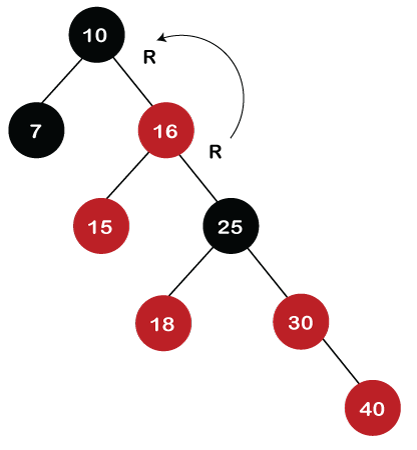
**Step 8:** The next element is 40. Since 40 is greater than 10, 16, 18, 25, and 30, so node 40 will come at the right of node 30. As the tree is not empty, node 40 would be Red in color. There is a Red-red conflict between nodes 40 and 30, so rule 4b will be applied.



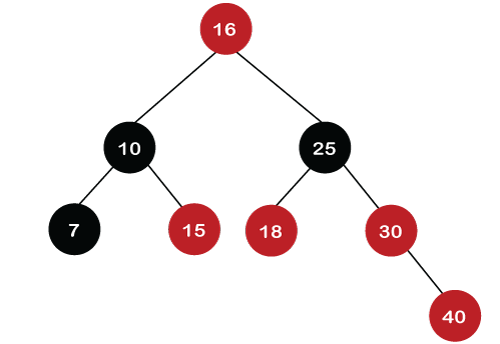
As the color of parent and parent's sibling node of a new node is Red so recoloring would be performed. The color of both the nodes would become black, as shown in the below image.

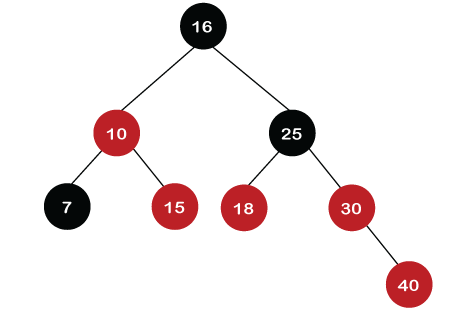
After recoloring, we also have to check the parent's parent of a new node, i.e., 25, which is not a root node, so recoloring would be performed, and the color of node 25 changes to Red.

After recoloring, red-red conflict occurs between nodes 25 and 16. Now node 25 would be considered as the new node. Since the parent of node 25 is red in color, and the parent's sibling is black in color, rule 4a would be applied. Since 25 is at the right of the node 16 and 16 is at the right of its parent, so there is an RR relationship. In the RR relationship, left rotation is performed. After left rotation, the median element 16 would be the root node, as shown in the below figure.



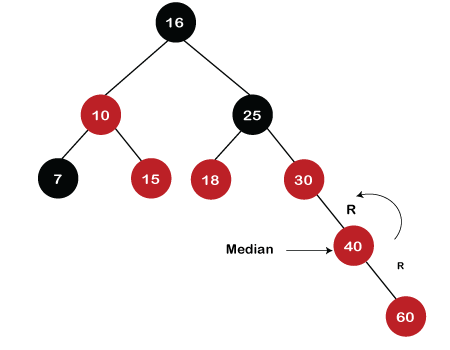
After rotation, recoloring is performed on nodes 16 and 10. The color of node 10 and node 16 changes to Red and Black, respectively as shown in the below figure.



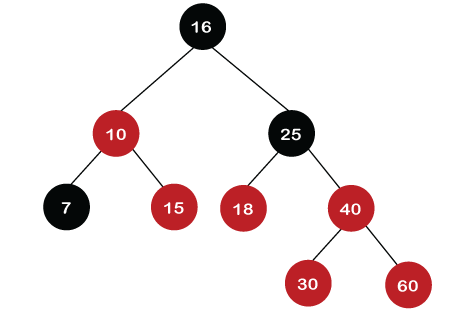


**Step 9:** The next element is 60. Since 60 is greater than 16, 25, 30, and 40, so node 60 will come at the right of node 40. As the tree is not empty, the color of node 60 would be Red.

As we can observe in the above tree that there is a Red-red conflict occurs. The parent node is Red in color, and there is no parent's sibling exists in the tree, so rule 4a would be applied. The first rotation would be performed. The RR relationship exists between the nodes, so left rotation would be performed.

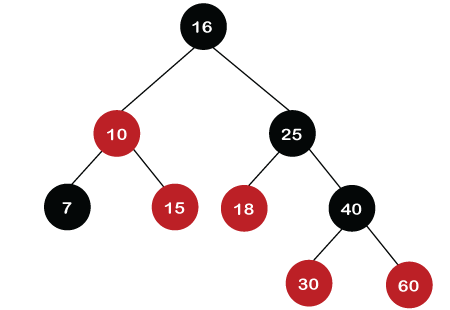


When left rotation is performed, node 40 will come upwards, and node 30 will come downwards, as shown in the below figure:



15

After rotation, the recoloring is performed on nodes 30 and 40. The color of node 30 would become Red, while the color of node 40 would become black.



15

The above tree is a Red-Black tree as it follows all the Red-Black tree properties.

**B-TREES**

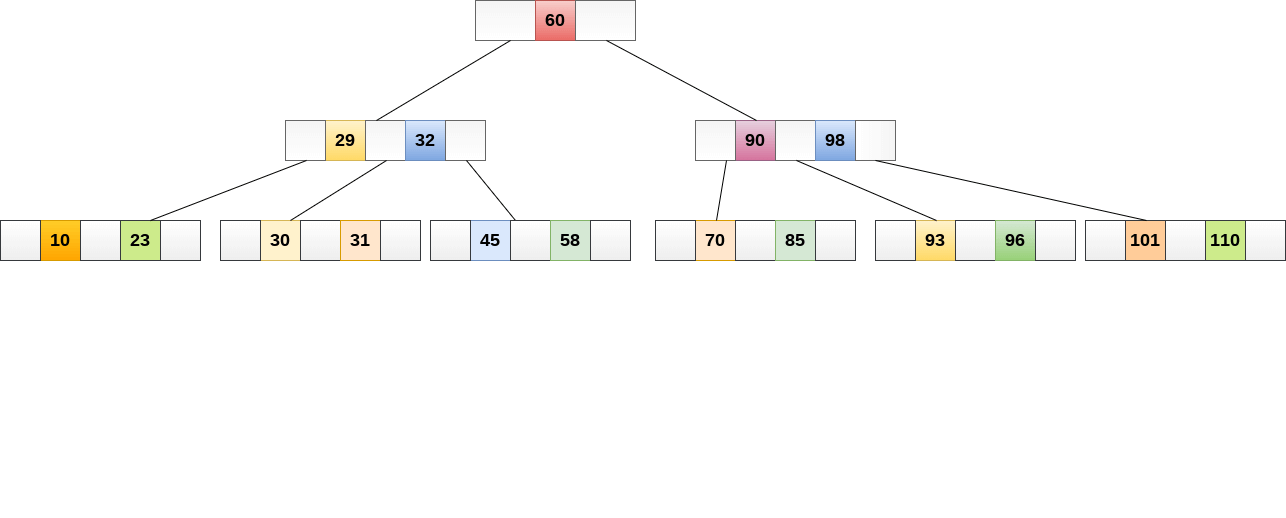
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

* Every node in a B-Tree contains at most m children.
* Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
* The root nodes must have at least 2 nodes.
* All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



**Searching:**

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

* Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
* Since, 40<49<56, traverse right sub-tree of 40.
* 49>45, move to right. Compare 49.
* match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.

**Inserting:**

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

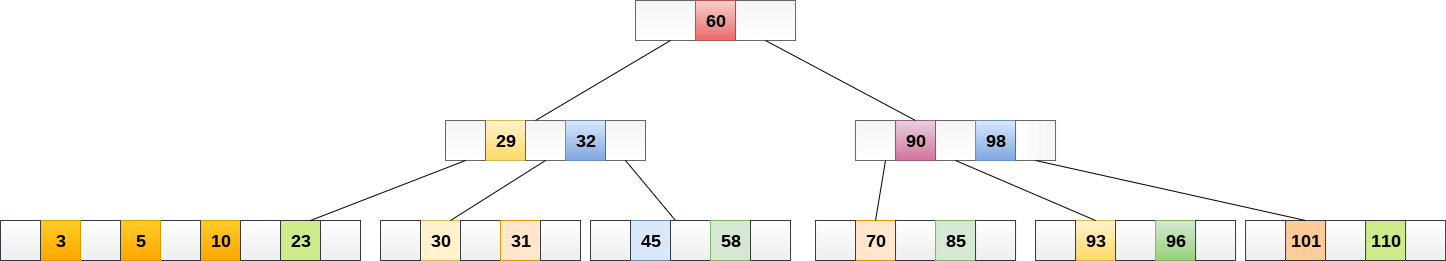
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.

* Split the node into the two nodes at the median.
* Push the median element upto its parent node.

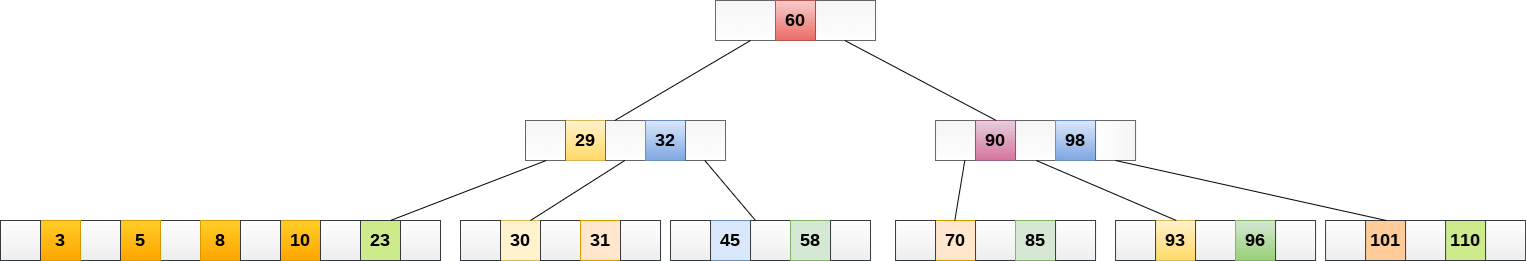
If the parent node also contain m-1 number of keys, then split it too by following the same steps.

**Example:**

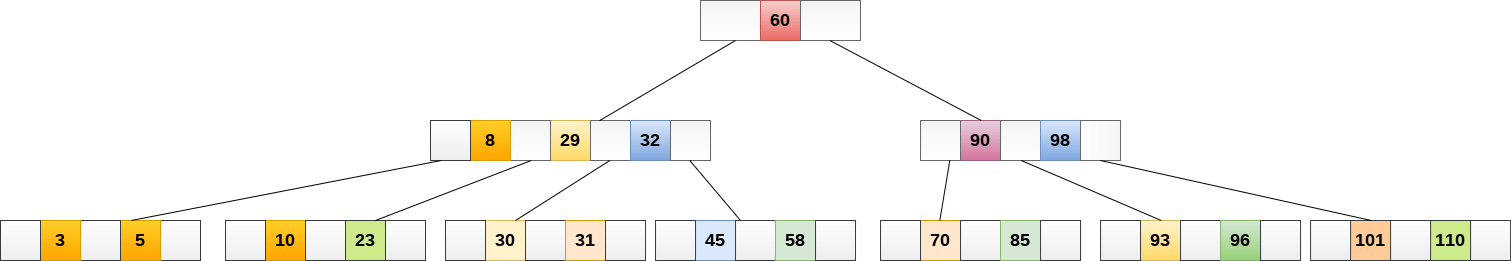
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4 ) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.

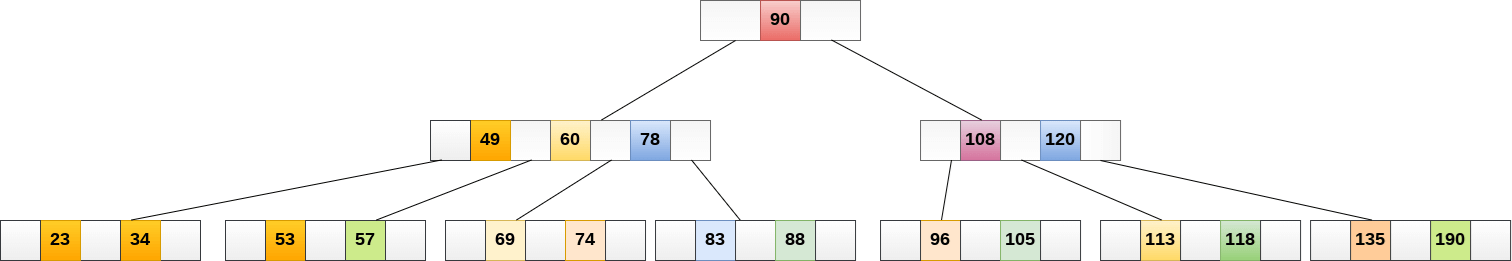
* If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
* If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

1. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
2. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

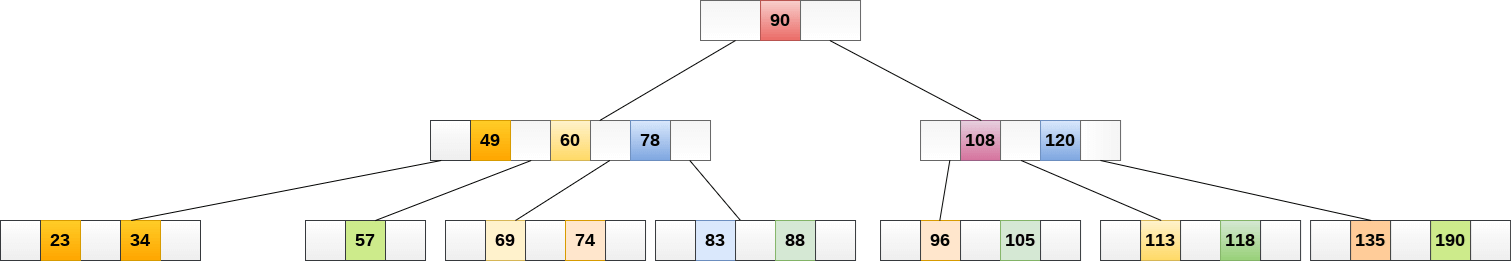
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

**Example:**

Delete the node 53 from the B Tree of order 5 shown in the following figure.

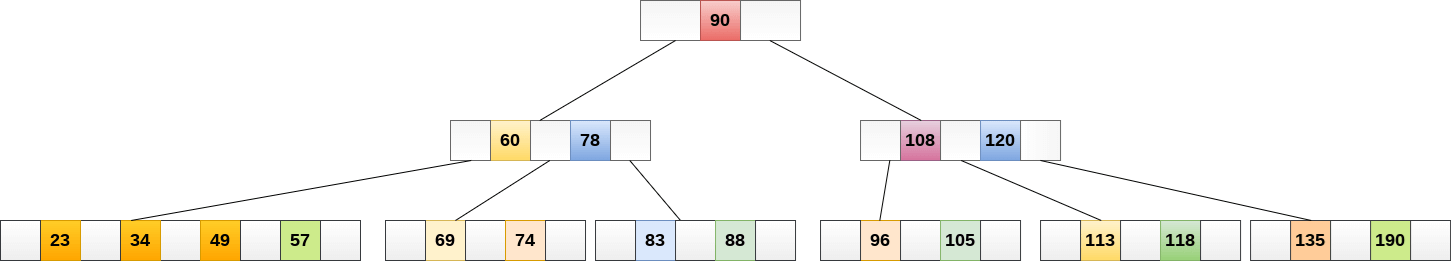


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



**GRAPHS: CONCEPTS AND ALGORITHMS**

* It is a type of data structure that has a series of vertices and edges (V,E) connected to each other.
* We can represent a graph in the following ways:  
  -Adjacency Matrix  
  -Adjacency List  
  -Map  
  -Edge List
* A graph is a non-linear data structure consisting of a set of vertices and a set of edges that connect pairs of vertices. Graphs are used to represent connections or relationships between objects
* Types of graphs:  
  1) Undirected Graph  
  2) Directed Graph  
  3) Complete Graph: A graph that has maximum number of edges  
  4) Cyclic Graph  
  5) Acyclic Graph
* Graph ADT:   
  -AddVertex(vertex)  
  -AddEdge(vertex1, vertex2)  
  -RemoveVertex(vertex)  
  -RemoveEdge(vertex1,vertex2)  
  -GetNeighbors(vertex)  
  -GetVertices()
* The adjacency matrix is a commonly used way to represent graphs, especially for dense graphs where the number of eds=ges is close to the maximum possible number of edges. In this representation, a 2D matrix is used to store whether there is an edge between each pair of vertices.
* The adjacency list representation is a popular way to represent graphs. This representation is particularly efficient for sparse graphs where the number of edges is much smaller than maximum possible number of edges.

**BFS (Breadth First Search):**

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

**Breadth First Search (BFS) for a Graph Algorithm:**

1. Initialization: Enqueue the starting node into a queue and mark it as visited.
2. Exploration: While the queue is not empty:

* Dequeue a node from the queue and visit it (e.g., print its value).
* For each unvisited neighbour of the dequeued node:
  + Enqueue the neighbour into the queue.
  + Mark the neighbour as visited.

1. Termination: Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

**DFS (Depth-First Search):**

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step-by-step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.

3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.

1. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
2. If no vertex is left, go back and pop a vertex from the stack.
3. Repeat steps 2, 3, and 4 until the stack is empty.

**Algorithm:**

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

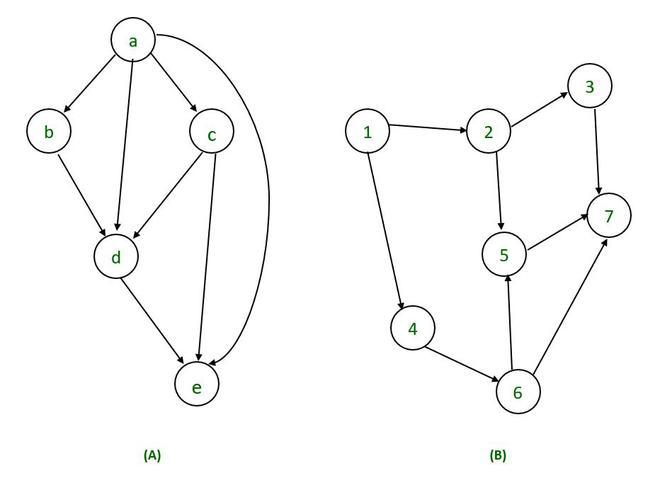
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

**Directed Acyclic Graph (DAG):**

A Directed Acyclic Graph, often abbreviated as DAG, is a fundamental concept in graph theory. DAGs are used to show how things are related or depend on each other in a clear and organized way. In this article, we are going to learn about Directed Acyclic Graph, its properties, and application in real life.



Directed Acyclic Graph has two important features:

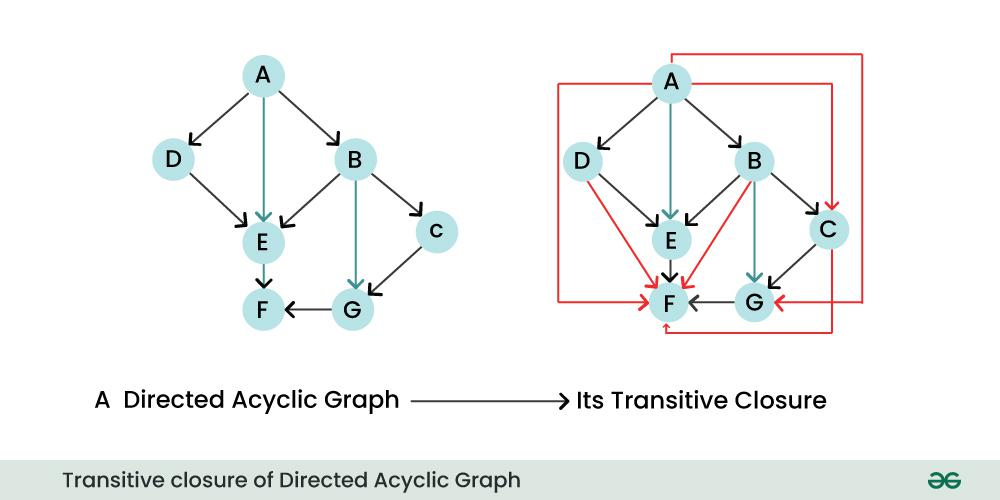
* **Directed Edges:** In Directed Acyclic Graph, each edge has a direction, meaning it goes from one vertex (node) to another. This direction signifies a one-way relationship or dependency between nodes.
* **Acyclic:** The term “acyclic” indicates that there are no cycles or closed loops within the graph. In other words, you cannot traverse a sequence of directed edges and return to the same node, following the edge directions. Formation of cycles is prohibited in DAG. Hence this characteristic is essential.

**Properties of Directed Acyclic Graph DAG:**

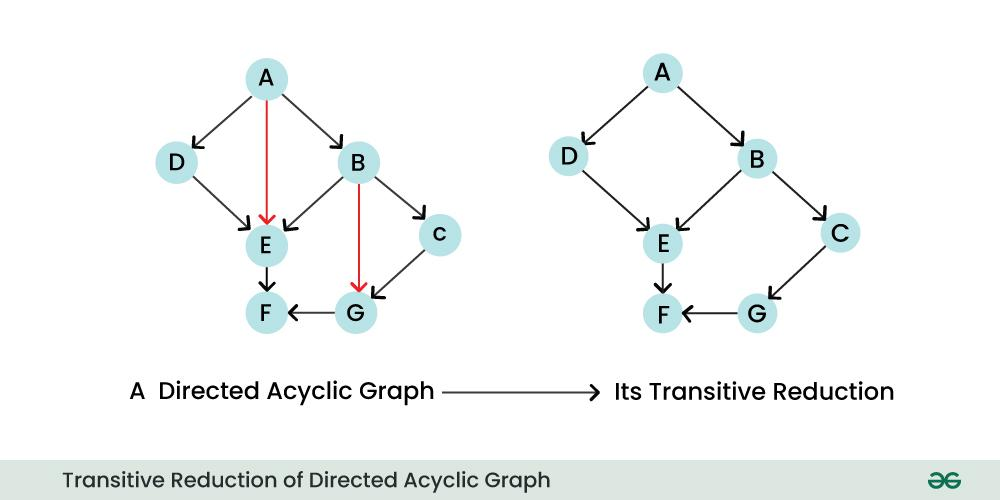
Directed Acyclic Graph (DAG) has different properties that make them usable in graph problems.

There are following properties of Directed Acyclic Graph (DAG):

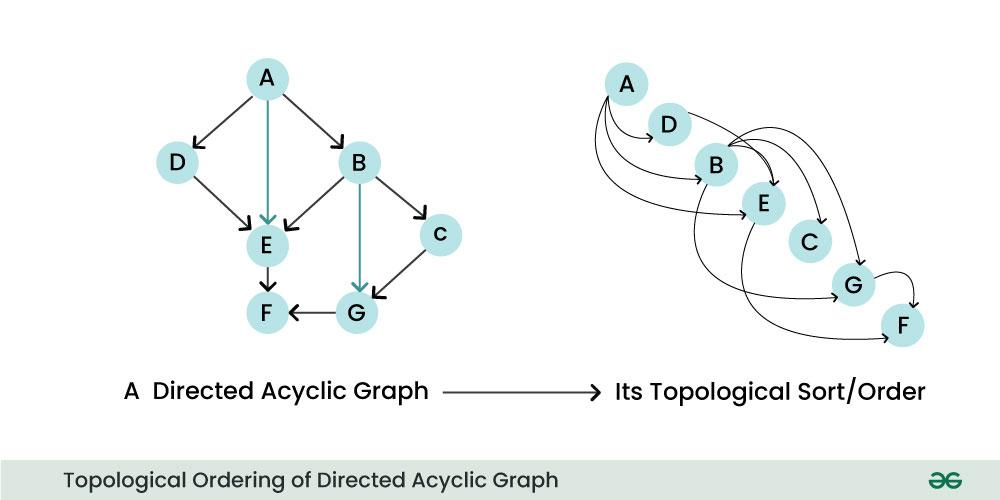
* **Reachability Relation:** In DAG, we can determine if there is a reachability relation between two nodes. Node A is said to be reachable from node B if there exists a directed path that starts at node B and ends at node A. This implies that you can follow the direction of edges in the graph to get from B to A.
* **Transitive Closure:** The transitive closure of a directed graph is a new graph that represents all the direct and indirect relationships or connections between nodes in the original graph. In other words, it tells you which nodes can be reached from other nodes by following one or more directed edges.



* **Transitive Reduction:** The transitive reduction of a directed graph is a new graph that retains only the essential, direct relationships between nodes, while removing any unnecessary indirect edges. In essence, it simplifies the graph by eliminating edges that can be inferred from the remaining edges.



* **Topological Ordering:** A DAG can be topologically sorted, which means you can linearly order its nodes in such a way that for all the edges, start node of the edge occurs earlier in the sequence. This property is useful for tasks like scheduling and dependency resolution.



A topological sort, or topological ordering, of a directed graph is a linear ordering of its vertices where every directed edge (u, v) from vertex u to vertex v comes before v in the ordering. For example, in the DAG ABCDEFGH, some possible topological sorts are: ABCDEFGH, ACBEGDFH, and BEGADCFH.

The topological sort algorithm takes a directed graph and returns an array of nodes where each node appears before all the nodes it points to. The algorithm works by:

1. Identifying a node with no incoming edges
2. Adding that node to the ordering
3. Removing it from the graph
4. Repeating until there aren't any more nodes with indegree zero

You can verify that an order is topological by deleting one and one vertex, and never deleting a vertex with an in-edge.

Topological sorting for a graph is not possible if the graph is not a directed acyclic graph (DAG). A DAG is a type of graph whose nodes are directionally related to each other and don't form a directional closed loop. In the practice of analytics engineering, DAGs are often used to visually represent the relationships between your data models.

**Single Source Shortest Path Algorithm (Djikstra’s Algorithm):**

Dijkstra’s algorithm is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

**Can Dijkstra’s Algorithm work on both Directed and Undirected graphs?**

Yes, Dijkstra’s algorithm can work on both directed graphs and undirected graphs as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.

* In a directed graph, each edge has a direction, indicating the direction of travel between the vertices connected by the edge. In this case, the algorithm follows the direction of the edges when searching for the shortest path.
* In an undirected graph, the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.

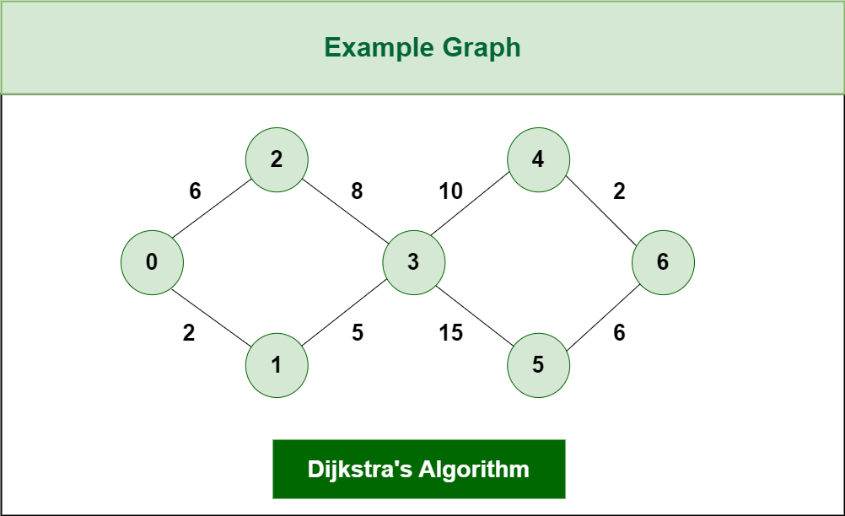
**Algorithm for Dijkstra’s Algorithm:**

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

Let’s see how Dijkstra’s Algorithm works with an example given below:

Dijkstra’s Algorithm will generate the shortest path from Node 0 to all other Nodes in the graph.

Consider the below graph:



The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

For this graph, we will assume that the weight of the edges represents the distance between two nodes.

As, we can see we have the shortest path from,

Node 0 to Node 1, from

Node 0 to Node 2, from

Node 0 to Node 3, from

Node 0 to Node 4, from

Node 0 to Node 6.

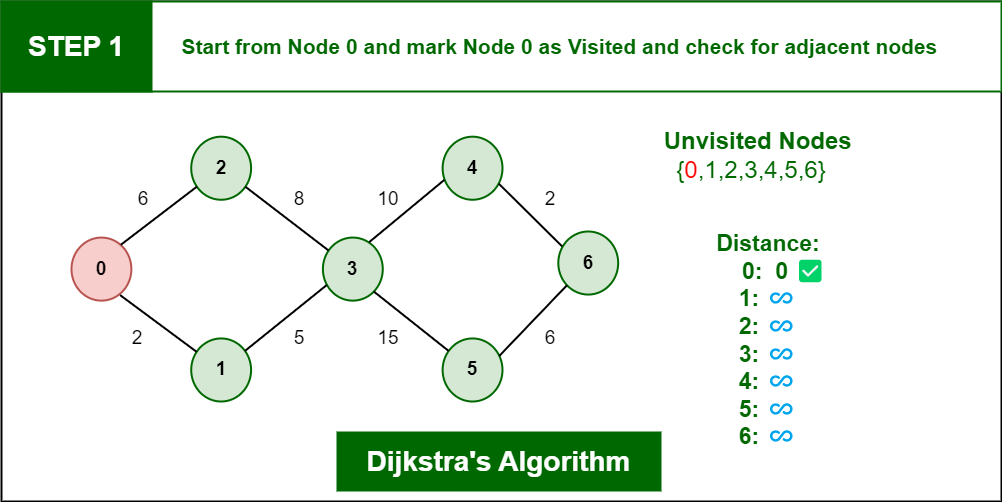
Initially we have a set of resources given below :

* The Distance from the source node to itself is 0. In this example the source node is 0.
* The distance from the source node to all other node is unknown so we mark all of them as infinity.

Example: 0 -> 0, 1-> ∞,2-> ∞,3-> ∞,4-> ∞,5-> ∞,6-> ∞.

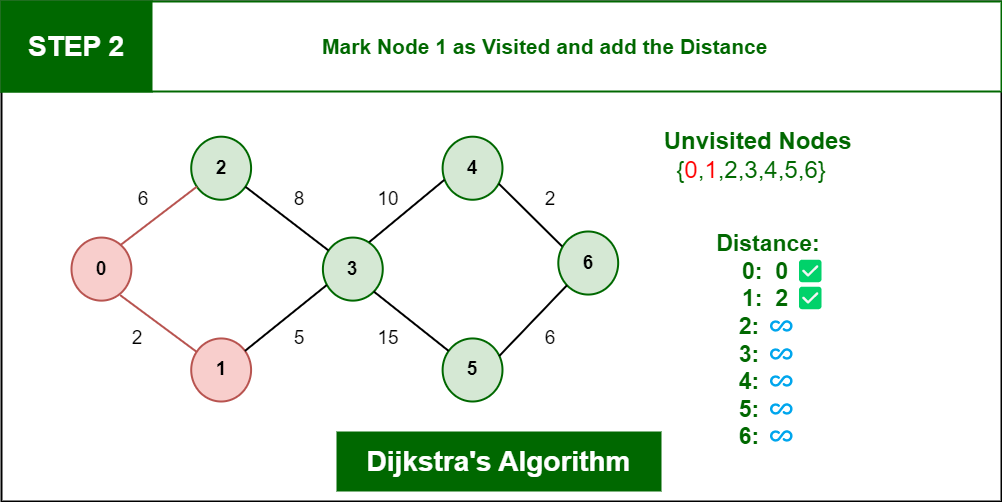
* we’ll also have an array of unvisited elements that will keep track of unvisited or unmarked Nodes.
* Algorithm will complete when all the nodes marked as visited and the distance between them added to the path. Unvisited Nodes:- 0 1 2 3 4 5 6.

**Step 1:** Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.



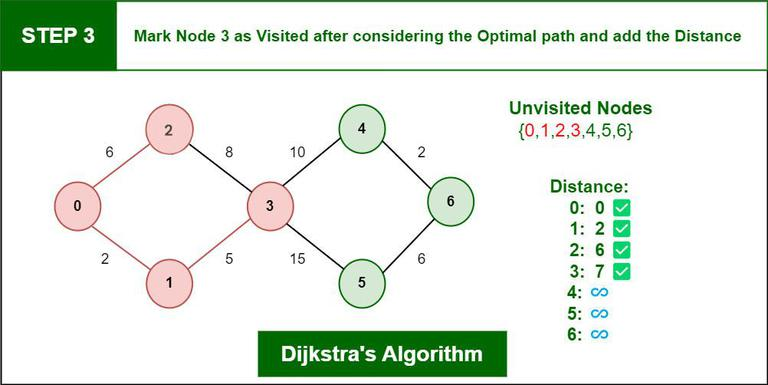
**Step 2:** Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6 ) and choose Node with minimum distance. In this step Node 1 is Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: Node 0 -> Node 1 = 2



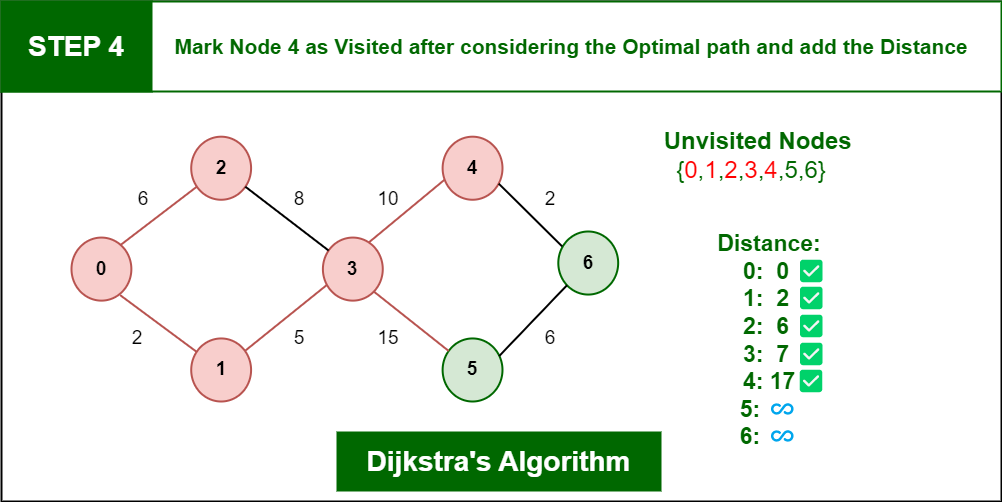
**Step 3:** Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:

Distance: Node 0 -> Node 1 -> Node 3 = 2 + 5 = 7



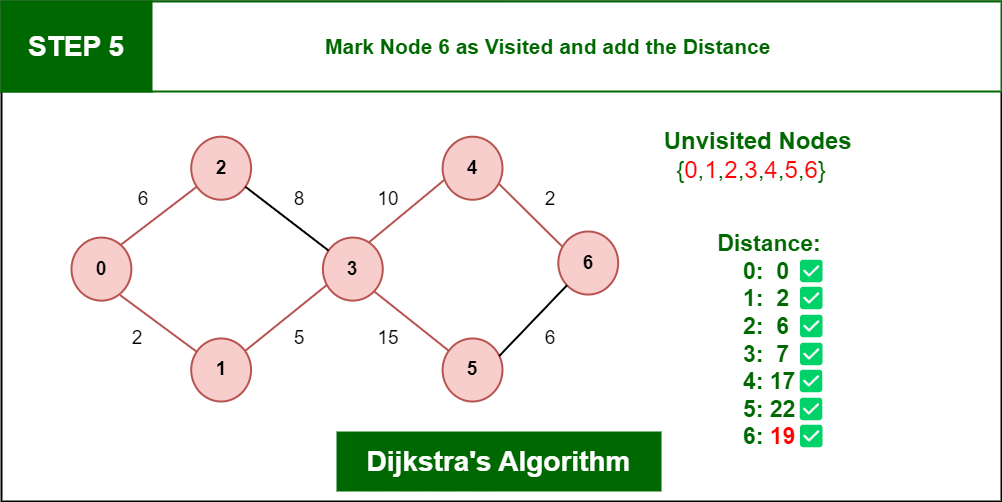
**Step 4:** Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step Node 4 is Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17



**Step 5:** Again, Move Forward and check for adjacent Node which is Node 6, so marked it as visited and add up the distance, Now the distance will be:

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19



So, the Shortest Distance from the Source Vertex is 19 which is optimal one.

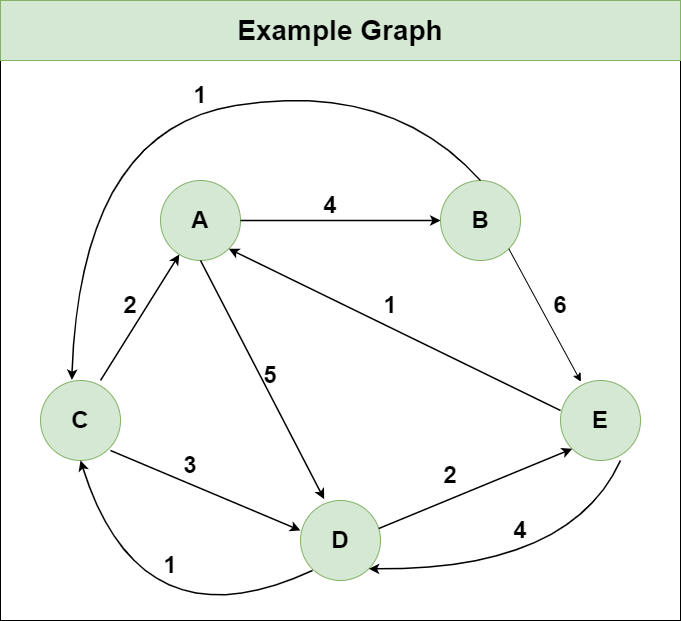
**All Pairs Shortest Path Algorithm (Floyd Warshall’s Algorithm):**

The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

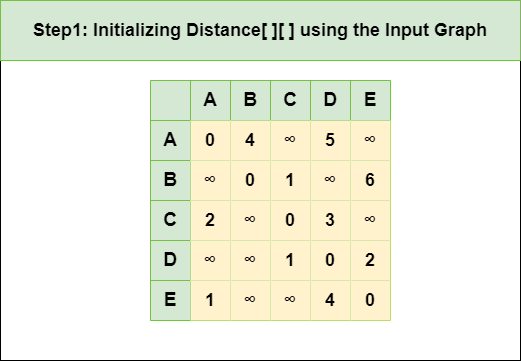
**Floyd Warshall Algorithm:**

* Initialize the solution matrix same as the input graph matrix as a first step.
* Then update the solution matrix by considering all vertices as an intermediate vertex.
* The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
* When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
  + For every pair (i, j) of the source and destination vertices respectively,
  + k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
  + k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j], if dist[i][j] > dist[i][k] + dist[k][j]

Suppose we have a graph as shown in the image:



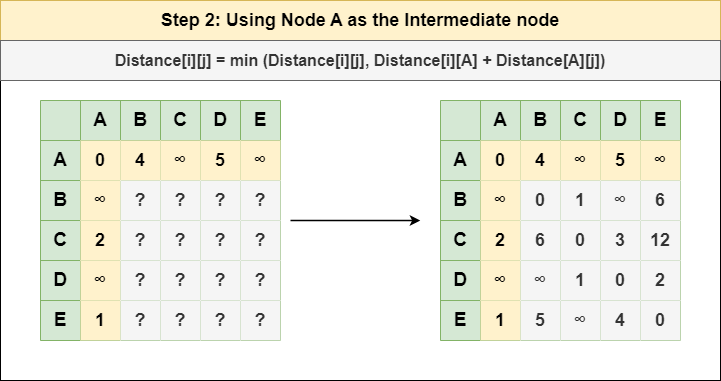
**Step 1**: Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from i to j, also Distance[i][j] = Infinity if there is no edge from i to j.



**Step 2:** Treat node A as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to A) + (Distance from A to j ))

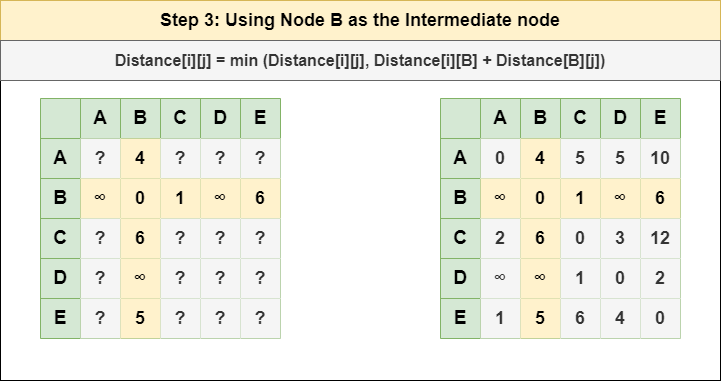
= Distance[i][j] = minimum (Distance[i][j], Distance[i][A] + Distance[A][j])



**Step 3**: Treat node B as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to B) + (Distance from B to j ))

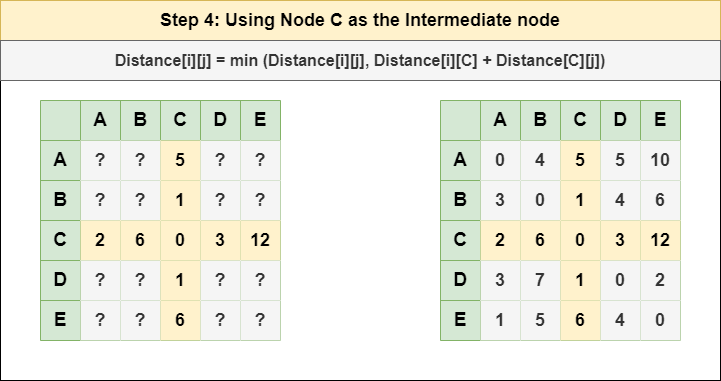
= Distance[i][j] = minimum (Distance[i][j], Distance[i][B] + Distance[B][j])



**Step 4:** Treat node C as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to C) + (Distance from C to j ))

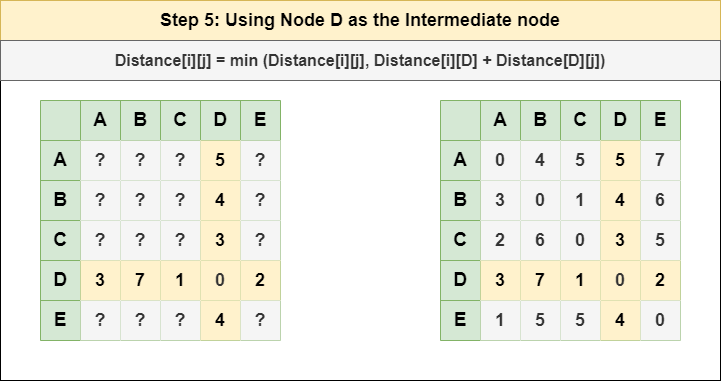
= Distance[i][j] = minimum (Distance[i][j], Distance[i][C] + Distance[C][j])



**Step 5:** Treat node D as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to D) + (Distance from D to j ))

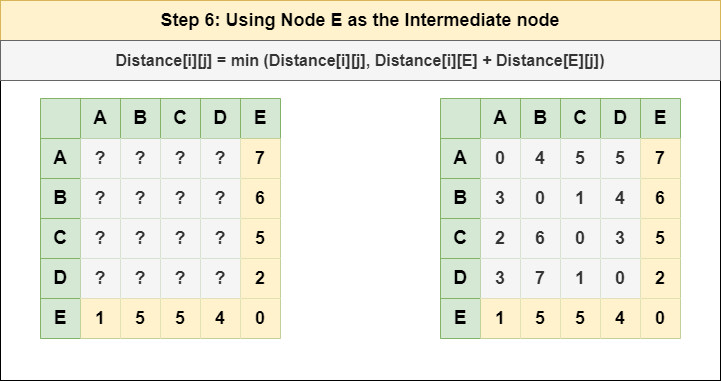
= Distance[i][j] = minimum (Distance[i][j], Distance[i][D] + Distance[D][j])



**Step 6:** Treat node E as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to E) + (Distance from E to j ))

= Distance[i][j] = minimum (Distance[i][j], Distance[i][E] + Distance[E][j])



**Step 7:** Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.

