1. **What is the difference between stack and queue ?**

You can refer to Stack as a linear form of data structure. In this, a user can delete and insert elements from only one side of a list. It is known as the ****top****. The stack data structure implements and follows the Last In, First Out (LIFO) principle. It implies that the element that is inserted last comes out first.

You can also refer to Queue as a linear form of data structure. In this, a user can insert elements from only one side of the list. It is known as the rear. Also, one can delete these elements from another side- known as the front. This type of data structure implements and follows the First In, First Out (FIFO) principle. It implies that the first element that inserts into a list will come out first.

## Difference Between Stack and Queue

|  |  |  |
| --- | --- | --- |
| ****Parameter**** | ****Stack Data Structure**** | ****Queue Data Structure**** |
| Basics | It is a linear data structure. The objects are removed or inserted at the same end. | It is also a linear data structure. The objects are removed and inserted from two different ends. |
| Working Principle | It follows the Last In, First Out (LIFO) principle. It means that the last inserted element gets deleted at first. | It follows the First In, First Out (FIFO) principle. It means that the first added element gets removed first from the list. |
| Pointers | It has only one pointer- the ****top****. This pointer indicates the address of the topmost element or the last inserted one of the stack. | It uses two pointers (in a simple queue) for reading and writing data from both the ends- the ****front**** and the ****rear****. The rear one indicates the address of the last inserted element, whereas the front pointer indicates the address of the first inserted element in a queue. |
| Operations | Stack uses ****push**** and ****pop**** as two of its operations. The pop operation functions to remove the element from the list, while the push operation functions to insert the element in a list. | Queue uses ****enqueue**** and ****dequeue**** as two of its operations. The dequeue operation deletes the elements from the queue, and the enqueue operation inserts the elements in a queue. |
| Structure | Insertion and deletion of elements take place from one end only. It is called the top. | It uses two ends- front and rear. Insertion uses the rear end, and deletion uses the front end. |
| Full Condition Examination | When top== max-1, it means that the stack is full. | When rear==max-1, it means that the queue is full. |
| Empty Condition Examination | When top==-1, it indicates that the stack is empty. | When front = rear+1 or front== -1, it indicates that the queue is empty. |
| Variants | A Stack data structure does not have any types. | A Queue data structure has three types- circular queue, priority queue, and double-ended queue. |
| Visualization | You can visualize the Stack as a vertical collection. | You can visualize a Queue as a horizontal collection. |
| Implementation | The implementation is simpler in a Stack. | The implementation is comparatively more complex in a Queue than a stack. |

1. **Application of stack**

A Stack is a widely used linear data structure in modern computers in which insertions and deletions of an element can occur only at one end, i.e., top of the Stack. It is used in all those applications in which data must be stored and retrieved in the last

### Following is the various Applications of Stack in Data Structure:

* Evaluation of Arithmetic Expressions
* Backtracking
* Delimiter Checking
* Reverse a Data
* Processing Function Calls

## 1. Evaluation of Arithmetic Expressions

A stack is a very effective [data structure](https://www.javatpoint.com/data-structure-tutorial) for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

### Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

* Infix Notation
* Prefix Notation
* Postfix Notation

## 2. Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem.

## 3. Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters.

## 4. Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

## 5. Processing Function Calls:

Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.

1. **What is stack ?**

A stack represents a sequence of objects or elements in a linear data structure format. The stack consists of a bounded bottom and all the operations are carried out on the top position. Whenever an element is added to the stack by the push operation, the top value is incremented by one, and when an element is popped out from the stack, the top value is decremented by one. A pointer to the top position of the stack is also known as the stack pointer.

A stack may be fixed in size or may have dynamic implementation where the size is allowed to change. In the case of bounded capacity stacks, trying to add an element to an already full stack causes a stack overflow exception. Similarly, a condition where a pop operation tries to remove an element from an already empty stack is known as underflow.

A stack is considered to be a restricted data structure as only a limited number of operations are allowed. Besides the push and pop operations, certain implementations may allow for advanced operations such as:

* Peek — View the topmost item in the stack.
* Duplicate — Copy the top item’s value into a variable and push it back into the stack.
* Swap — Swap the two topmost items in the stack.
* Rotate — Move the topmost elements in the stack as specified by a number or move in a rotating fashion.

Software implementations of the stack concept are done using arrays and linked lists where the top position is tracked using a variable or header pointer respectively. Many programming languages provide built-in features to support stack implementation.

Hardware stacks are implemented for the purpose of memory allocation and access using a fixed origin and size. Stack registers are used to store the value of the stack pointer.

1. **What is queue ?**

 queue consists of a number of packets. These packets are bound to be routed over the network, lined up in a sequential way with a changing header and trailer and taken out of the queue for transmission by a network device using some defined packet processing algorithm like first in first out (FIFO), last in last out (LIFO), etc. The queue dequeues, or takes out a data packet from the head, when it needs to transfer and trailer by adding new data packets to the queue, which is known as enqueuing.  
  
A queue works almost on the same methodology used at banks or supermarkets, where the customer is treated according to its arrival. An example would be FIFO or some other priority if they are a privileged customer. Similarly, a network queue processes data packets based on their arrival, priority, smallest task first and multitasking, FIFO, LIFO, emption and pre-emption.

1. **What is linked list ?**

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

**Link** − Each link of a linked list can store a data called an element.

**Next** − Each link of a linked list contains a link to the next link called Next.

**LinkedList** − A Linked List contains the connection link to the first link called First.

## Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

Linked List contains a link element called first.

Each link carries a data field(s) and a link field called next.

Each link is linked with its next link using its next link.

Last link carries a link as null to mark the end of the list.

## Types of Linked List

Following are the various types of linked list.

**Simple Linked List** − Item navigation is forward only.

**Doubly Linked List** − Items can be navigated forward and backward.

**Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

## Basic Operations

Following are the basic operations supported by a list.

**Insertion** − Adds an element at the beginning of the list.

**Deletion** − Deletes an element at the beginning of the list.

**Display** − Displays the complete list.

**Search** − Searches an element using the given key.

**Delete** − Deletes an element using the given key.

## Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

NewNode.next −> RightNode;

It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;



This will put the new node in the middle of the two. The new list should look like this −



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

## Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

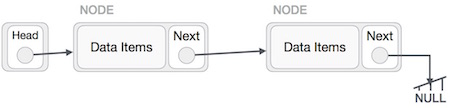


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

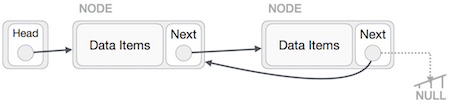


## Reverse Operation

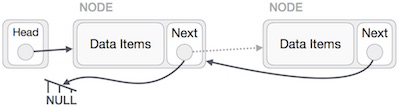
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



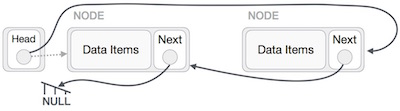
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



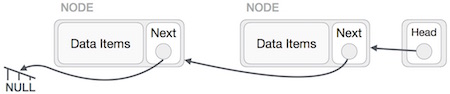
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



1. **What is array ?**

An array is a collection of similar data elements stored at contiguous memory locations. It is the simplest data structure where each data element can be accessed directly by only using its index number.

For instance, if we want to store the marks scored by a student in 5 subjects, then there’s no need to define individual variables for each subject. Rather, we can define an array which will store the data elements at contiguous memory locations.

· array ****marks[5]**** defines the marks scored by a student in 5 different subjects where each subject marks are located at a particular location in the array i.e. ****marks[0]**** denotes the marks scored in first subject, ****marks[1]**** denotes the marks scored in 2nd subject and so on.

## ****Need of using Array –****

In programming, most of the cases need to store a large amount of data of similar type. To store such a huge amount of data, we need to define numerous variables. It would be very tough to memorize all variable names while writing the programs. Instead, it is better to define an array and store all the elements into it.

## ****Advantages of Array –****

* Arrays represent multiple data elements of the same type using a single name.
* In an array, accessing or searching an element is easy by using the index number.
* An array can be traversed easily just by incrementing the index by 1.
* Arrays allocate memory in contiguous memory locations for all its data elements.

### ****Types of indexing in Array-****

· 0 (Zero Based Indexing) –  The first element of array refers to index 0.

· 1 (One Based Indexing) –  The first element of array refers to index 1.

· n (n Based Indexing) – The base index of an array can be chosen as per requirement.

1. **Recursive function of stack ?**

**Ye likh k bhejo samajh nai aa raha hai**

1. **Define hashing ?**

* Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
* Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
* It is a technique to convert a range of key values into a range of indexes of an array.
* It is used to facilitate the next level searching method when compared with the linear or binary search.
* Hashing allows to update and retrieve any data entry in a constant time O(1).
* Constant time O(1) means the operation does not depend on the size of the data.
* Hashing is used with a database to enable items to be retrieved more quickly.
* It is used in the encryption and decryption of digital signatures.

## What is Hash Function?

* A fixed process converts a key to a hash key is known as a **Hash Function.**
* This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
* Hash value represents the original string of characters, but it is normally smaller than the original.
* It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
* If the hash values are same, the message is transmitted without errors.

## What is Hash Table?

* Hash table or hash map is a data structure used to store key-value pairs.
* It is a collection of items stored to make it easy to find them later.
* It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
* It is an array of list where each list is known as bucket.
* It contains value based on the key.
* Hash table is used to implement the map interface and extends Dictionary class.
* Hash table is synchronized and contains only unique elements.

1. **What is self organizing list ?**

A self-organizing list serializes its elements to optimize mean access time using a self-organizing algorithm. A self-organizing list aims to make linear search more efficient by pushing frequently accessed items to the front of the list. In the best situation, a self-organizing list achieves near-constant time for element access.

A reorganizing method is used by a self-organizing list to adjust to different query distributions at runtime. While key-value ordering is the most frequent method for sorting lists, it is not the only one. Another way to organize lists to speed up searches is to sort them by expected access frequency. While the benefits may not be as substantial as when ordered by key value, organizing (at least roughly) by frequency of access can be much less expensive and speed up sequential search in some cases.

A sorted linked list's worst-case search time is O. (n). After one comparison with root, we can skip roughly half of the nodes with a Balanced Binary Search Tree. We have random access to a sorted array and can use Binary Search on arrays.

Self-organizing lists rearrange the records in the list according to the actual pattern of record access. For determining how to arrange the list, self-organizing lists employ a heuristic. These heuristics are comparable to buffer pool management guidelines. A buffer pool is, in reality, a type of self-organizing list. Because we normally must explore the contents of the buffers to determine if the needed information is already in the main memory, ordering the buffer pool by expected frequency of access is a suitable method.

When ordered by frequency of access, the buffer at the end of the list will be the most suited for reuse when a fresh page of information needs to be read. For managing self-organizing lists, there are three traditional heuristics. These three heuristics are listed below:

### 1. Frequency Count

One of the most obvious techniques to keep a list ordered by frequency is to keep track of how many times each record has been accessed and keep the records in that order at all times. Frequency count, or simply "count," will be the name of this procedure. The count is a buffer replacement approach that isn't utilized very often. If the number of accesses to a record exceeds the number of accesses to a record preceding it, the record will advance to the head of the list. As a result, the count will keep track of the records in the order in which they've happened so far.

The count does not respond well to the increasing frequency of access over time and requires space for access counts. A big number of people can access a record once it's been accessed. In addition to requiring space for access counts, the count does not respond well to increasing access frequency over time. Regardless of subsequent access history, once a record has been accessed a significant number of times under the frequency count system, it will remain near the top of the list.

### 2. Transpose

Any record found should be swapped with the record in the list that comes before it. Transpose is the name for this strategy. For list implementations that use linked lists or arrays, Transpose is a viable choice. Records that are used frequently will get to the top of the list over time. Records that were once frequently accessible but are no longer needed will gradually be pushed to the back of the file cabinet. As a result, it appears to have favorable features in terms of changing access frequency. Unfortunately, some access sequences are pathological and can cause transposition to fail.

Consider the situation in which the list's last record (called XX) is accessed. The next-to-last record (say YY) is then swapped out with this one. The next-to-last record (say YY) is then exchanged with this one, making YY the final record. If YY is now accessible, it will be replaced with XX. Because neither record will ever progress toward the front, a repeating series of accesses alternating between XX and YY will always search to the end of the list. In practice, however, such pathological situations are uncommon. Moving the accessed record forward in the list by a fixed number of steps is a variation on transposition.

### 3. Move to Front

It is moved to the top of the list when a record is located, pushing all other records back to one spot. The move-to-front strategy is equivalent to the most recently utilized buffer replacement strategy. If the records are stored in a linked list, this heuristic is simple to apply. When records are stored in an array, moving a record forward from near the end causes many records to (slightly) shift positions.

When at least n searches are conducted, the cost of move-to-front is finite because it requires at most twice the number of accesses required by the optimal static ordering for n records. In other words, if we had known the sequence of (at least n) searches ahead of time and stored the data in order of frequency to reduce the total cost of these accesses, the cost would have been at least half of what the move-to-front heuristic required. (Amortized analysis can be used to verify this).

Finally, move-to-front adapts to local variations in access frequency, so if a record is often accessed for a short time, it will appear near the front of the list during that time. When records are processed sequentially, move-to-front performs badly, especially if the sequential order is repeated numerous times.

## Advantages of Self-organizing List

A self-organizing list has the following advantages, such as:

* During the compilation or interpretation of program source code, language translators such as compilers and interpreters use self-organizing lists to maintain symbol tables.
* Embedded systems are currently undergoing research to incorporate the self-organizing list data format to reduce bus transition activity, which causes power consumption in those circuits.
* Artificial intelligence and neural networks, as well as self-adjusting systems, utilize these lists. As in the case of the LFU algorithm, the methods used in self-organizing lists are also used as caching algorithms.

In real-world collections, the basic Move to Front and transposition methods can be used to organize a spice drawer by moving used items to the front of the drawer or to transpose a cleaning item with its front-most neighbor when it is used.

1. **What is skip list ?**

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

## Skip list structure

It is built in two layers: The lowest layer and Top layer.

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

## Skip List Basic Operations

There are the following types of operations in the skip list.

* ****Insertion operation:**** It is used to add a new node to a particular location in a specific situation.
* ****Deletion operation:**** It is used to delete a node in a specific situation.
* ****Search Operation:**** The search operation is used to search a particular node in a skip list.

## Advantages of the Skip list

1. If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
2. The skip list is simple to implement as compared to the hash table and the binary search tree.
3. It is very simple to find a node in the list because it stores the nodes in sorted form.
4. The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
5. The skip list is a robust and reliable list.

## Disadvantages of the Skip list

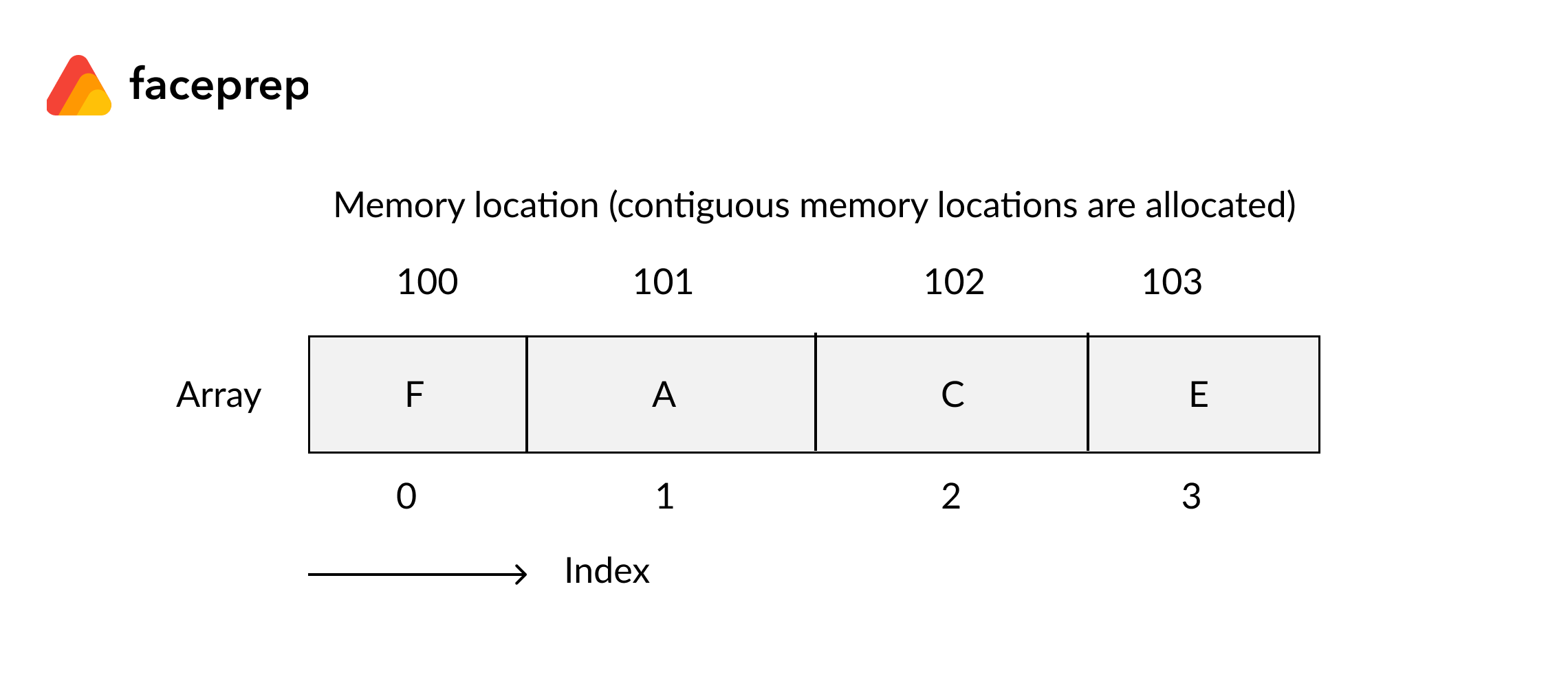
1. It requires more memory than the balanced tree.
2. Reverse searching is not allowed.
3. The skip list searches the node much slower than the linked list.

## Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. It is also used with the QMap template class.
4. The indexing of the skip list is used in running median problems.
5. The skip list is used for the delta-encoding posting in the Lucene search.
6. **Advantage and disadvantage of array and linked list --**

## ****Advantages of Arrays****

* Arrays represent multiple data items of the same type using a single name.
* In arrays, the elements can be accessed randomly by using the index number.
* Arrays allocate memory in contiguous memory locations for all its elements. Hence there is no chance of extra memory being allocated in case of arrays. This avoids memory overflow or shortage of memory in arrays.



* Using arrays, other data structures like linked lists, stacks, queues, trees, graphs etc can be implemented.
* Two-dimensional arrays are used to represent matrices.

## 

## ****Disadvantages of Arrays****

* The number of elements to be stored in an array should be known in advance.
* An array is a static structure (which means the array is of fixed size). Once declared the size of the array cannot be modified. The memory which is allocated to it cannot be increased or decreased.
* Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.
* Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.

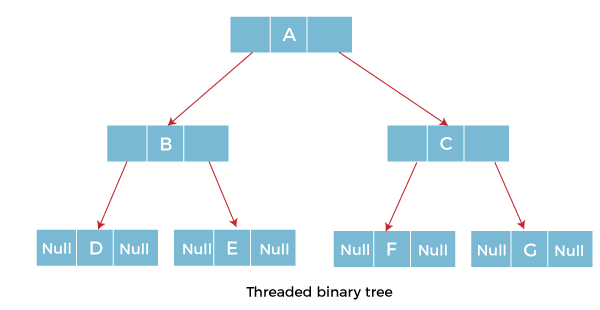
### **Advantages of Linked List**

1. The linked list is a dynamic data structure.
2. You can also decrease and increase the linked list at run-time. That is, you can allocate and deallocate memory at run-time itself.
3. In this, you can easily do insertion and deletion functions. That is, you can easily insert and delete the node.
4. Memory is well utilized in the linked list. Because in it, we do not have to allocate memory in advance.
5. Its access time is very fast, and it can be accessed at a certain time without memory overhead.
6. You can easily implement linear data structures using the linked list like a stack, queue.

### ****Disadvantages of Linked List****

1. The linked list requires more memory to store the elements than an array, because each node of the linked list points a pointer, due to which it requires more memory.
2. It is very difficult to traverse the nodes in a linked list. In this, we cannot access randomly to any one node. (As we do in the array by index.) For example: - If we want to traverse a node in an n position, then we have to traverse all the nodes that come before n, which will spoil a lot of our time.
3. Reverse traversing in a linked list is very difficult, because it requires more memory for the pointer.
4. **What is threaded binary tree ?**

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of ****n**** nodes then ****n+1**** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as ****threaded binary trees****. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

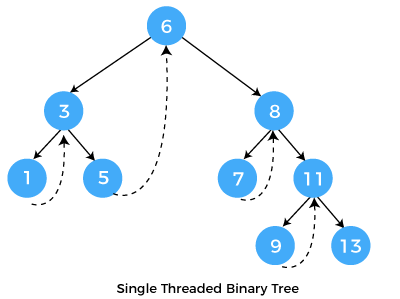


### Types of Threaded Binary Tree

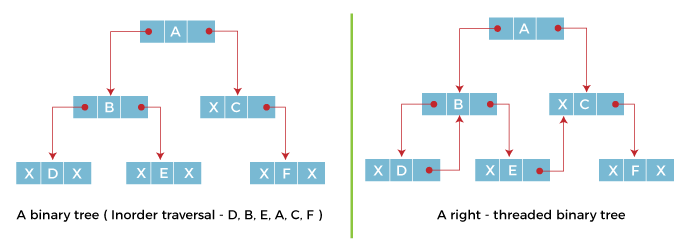
There are two types of threaded Binary Tree:

* One-way threaded Binary Tree
* Two-way threaded Binary Tree

****One-way threaded Binary trees:****

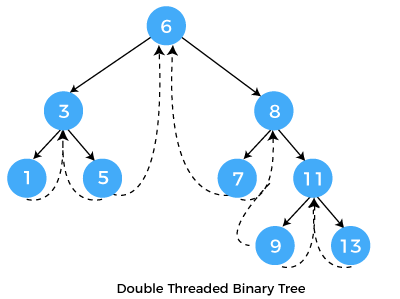


In one-way threaded binary trees, a thread will appear either in the right or left link field of a node. If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called ****Right threaded binary trees****. If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called ****Left threaded binary trees.**** Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees. In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.

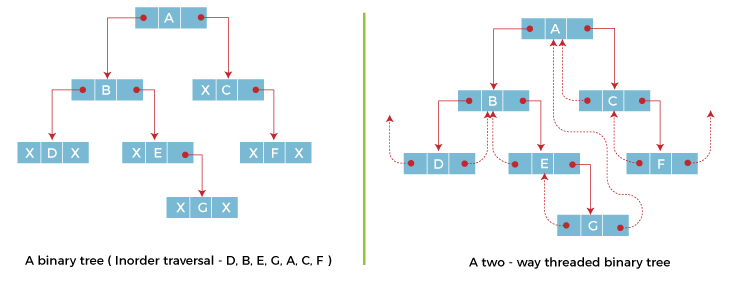


The above figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the inorder successor of a node D. In the same way other nodes containing values in the right link field will contain NULL value.

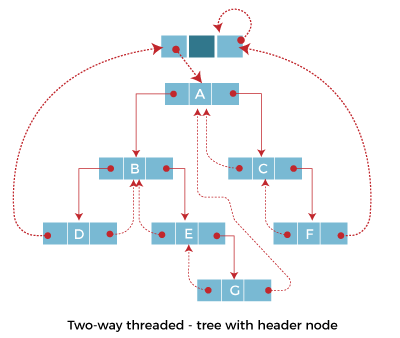
****Two-way threaded Binary Trees:****



In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.



The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F. If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B. Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor. In the same way, other nodes containing NULL values in their link fields are filled with threads.



In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node. This is because they don't have any inorder predecessor and successor respectively. This is indicated by threads pointing nowhere. So in order to maintain the uniformity of threads, we maintain a special node called the ****header node****. The header node does not contain any data part and its left link field points to the root node and its right link field points to itself. If this header node is included in the two-way threaded Binary tree then this node becomes the inorder predecessor of the first node and inorder successor of the last node. Now threads of left link fields of the first node and right link fields of the last node will point to the header node.

****Advantages of Threaded Binary Tree:****

* In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
* It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links. It almost behaves like a circular linked list.

****Disadvantages of Threaded Binary Tree:****

* When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
* Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

1. **Balancing avl tree**

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

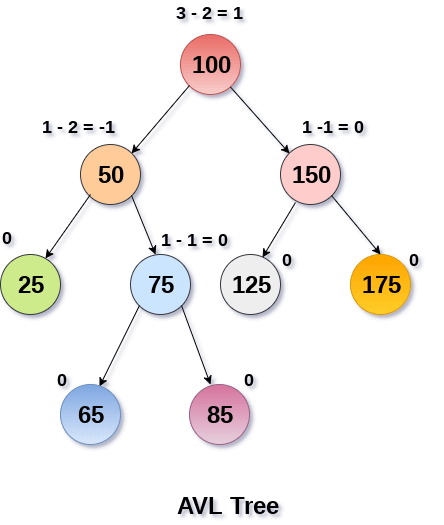
## Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



## Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion](https://www.javatpoint.com/insertion-in-avl-tree) | Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations. |
| 2 | [Deletion](https://www.javatpoint.com/deletion-in-avl-tree) | Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree. |

## Why AVL Tree?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is ****O(h)****. However, it can be extended to ****O(n)**** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be ****O(log n)**** where n is the number of nodes.

## AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than ****-1, 0, and 1****. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

### 1. 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](https://www.javatpoint.com/rr-rotation-in-avl-tree) is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

### 2. 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](https://www.javatpoint.com/ll-rotation-in-avl-tree) is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

### 3. LR Rotation

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.

****Let us understand each and every step very clearly:****

|  |  |
| --- | --- |
| **State** | **Action** |
| IMG_258 | A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C |
| IMG_259 | As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node ****A****, has become the left subtree of ****B****. |
| IMG_260 | After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of ****C**** |
| IMG_261 | Now we perform LL clockwise rotation on full tree, i.e. on node C. node ****C**** has now become the right subtree of node B, A is left subtree of B |
| IMG_262 | Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now. |

### 4. RL Rotation

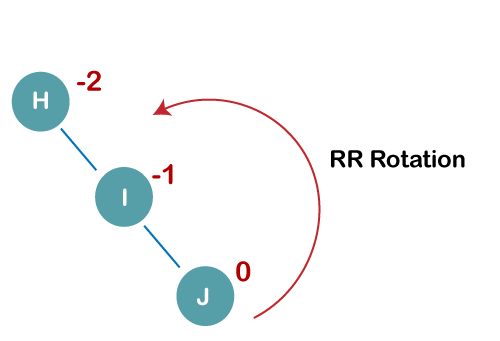
As already discussed, that double rotations are bit tougher than single rotation which has already explained above. [R L rotation](https://www.javatpoint.com/rl-rotation-in-avl-tree) = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full 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.

|  |  |
| --- | --- |
| **State** | **Action** |
| IMG_263 | A node ****B**** has been inserted into the left subtree of ****C**** the right subtree of ****A****, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A |
| IMG_264 | As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at ****C**** is performed first. By doing RR rotation, node ****C**** has become the right subtree of ****B****. |
| IMG_265 | After performing LL rotation, node ****A**** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A. |
| IMG_266 | Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node ****C**** has now become the right subtree of node B, and node A has become the left subtree of B. |
| IMG_267 | Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now. |

### Q: Construct an AVL tree having the following elements

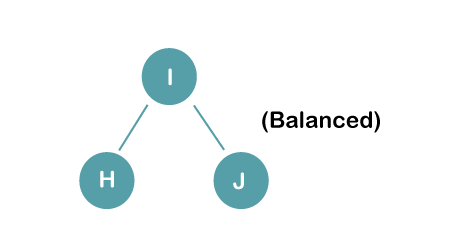
****H, I, J, B, A, E, C, F, D, G, K, L****

****1. Insert H, I, J****

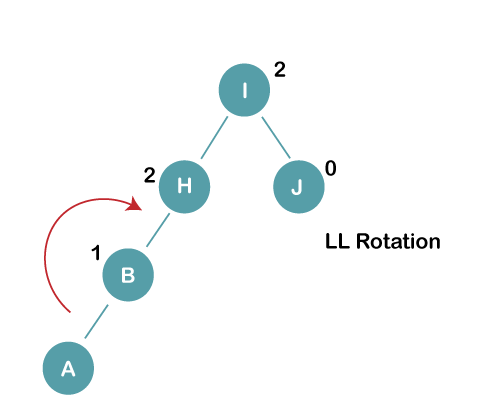


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

****The resultant balance tree is:****

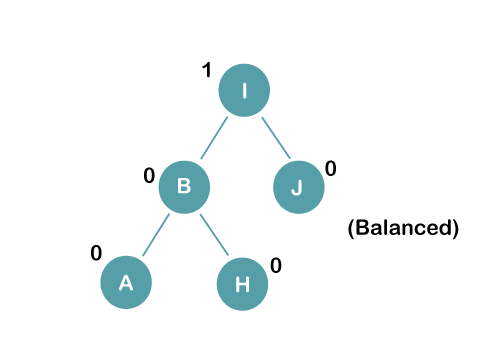


****2. Insert B, A****

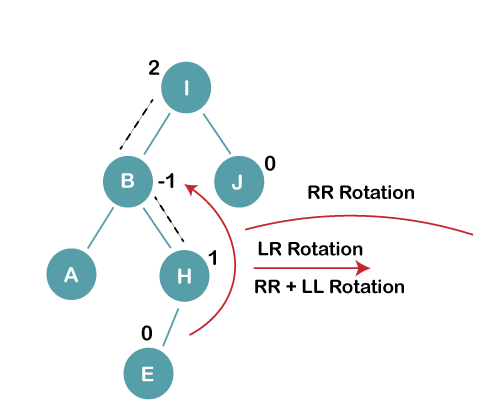


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

****The resultant balance tree is:****



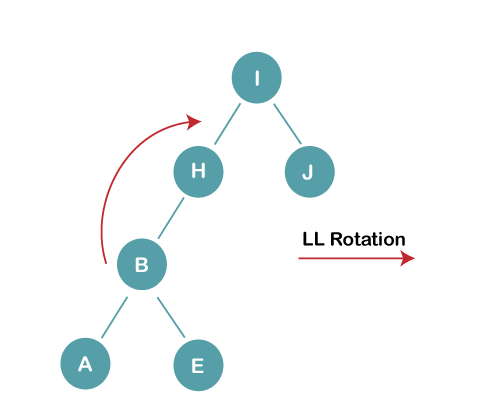
****3. Insert E****



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

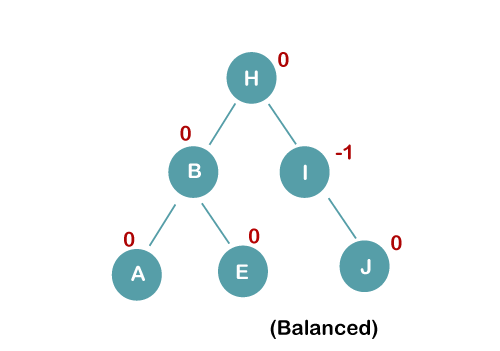
****3 a) We first perform RR rotation on node B****

****The resultant tree after RR rotation is:****

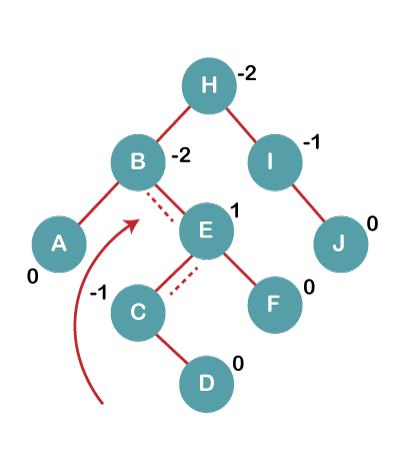


****3b) We first perform LL rotation on the node I****

****The resultant balanced tree after LL rotation is:****



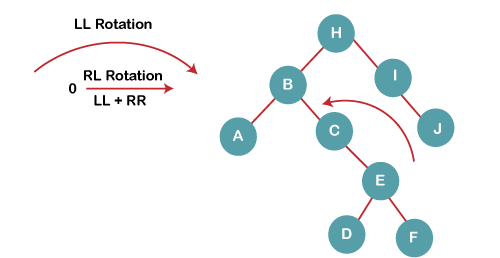
****4. Insert C, F, D****



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

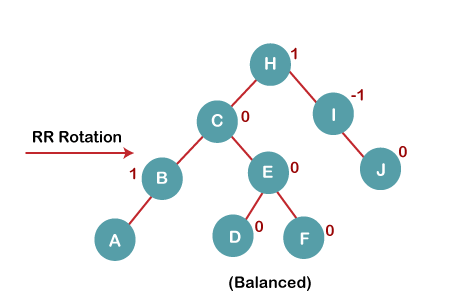
****4a) We first perform LL rotation on node E****

****The resultant tree after LL rotation is:****

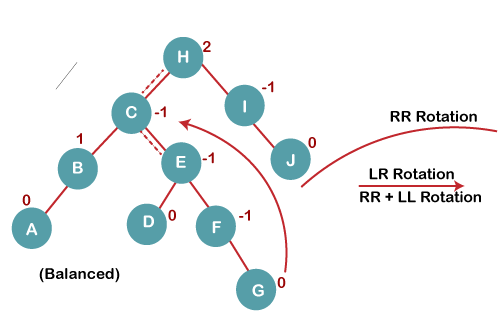


****4b) We then perform RR rotation on node B****

****The resultant balanced tree after RR rotation is:****



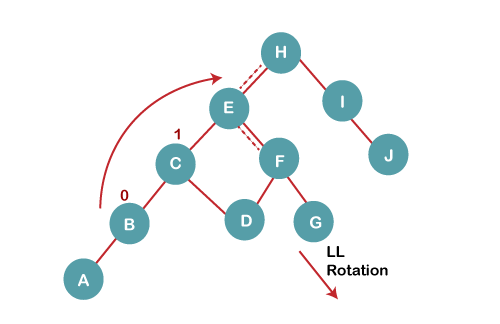
****5. Insert G****



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

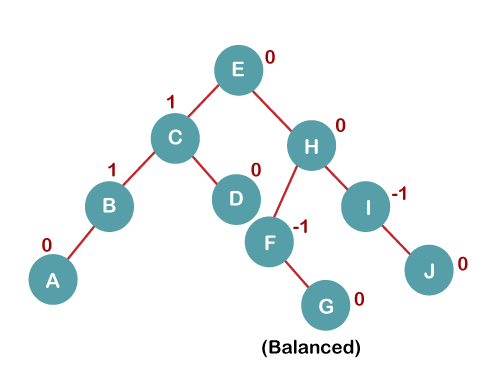
****5 a) We first perform RR rotation on node C****

****The resultant tree after RR rotation is:****

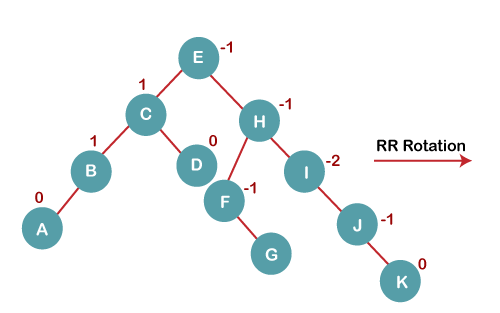


****5 b) We then perform LL rotation on node H****

****The resultant balanced tree after LL rotation is:****

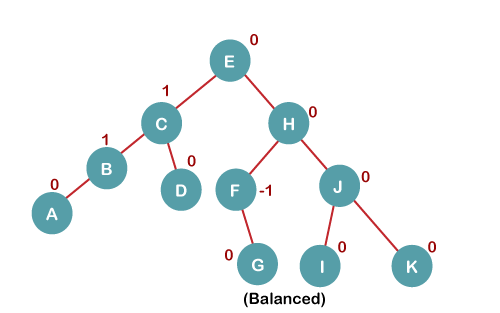


****6. Insert K****



On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

****The resultant balanced tree after RR rotation is:****



****7. Insert L****

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree

