**Bitwise information**

**1)How to set a bit in the number 'num' :** If we want to set a bit at nth position in number 'num' ,it can be done using 'OR' operator( | ).

* First we left shift '1' to n position via (1 << n)
* Then, use 'OR' operator to set bit at that position.'OR' operator is used because it will set the bit even if the bit is unset previously in binary representation of number 'num'.

**2)How to unset/clear a bit at n'th position in the number 'num' :**

Suppose we want to unset a bit at nth position in number 'num' then we have to do this with the help of 'AND' (&) operator.

* First we left shift '1' to n position via (1 << n) than we use bitwise NOT operator '~' to unset this shifted '1'.
* Now after clearing this left shifted '1' i.e making it to '0' we will 'AND'(&) with the number 'num' that will unset bit at nth position position.

**3)Toggling a bit at nth position :** Toggling means to turn bit 'on'(1) if it was 'off'(0) and to turn 'off'(0) if it was 'on'(1) previously.We will be using 'XOR' operator here which is this '^'. The reason behind 'XOR' operator is because of its properties.

* Properties of 'XOR' operator.  
  + 1^1 = 0
  + 0^0 = 0
  + 1^0 = 1
  + 0^1 = 1
* If two bits are different then 'XOR' operator returns a set bit(1) else it returns an unset bit(0).

**4)Checking if bit at nth position is set or unset:**

It is quite easily doable using 'AND' operator.

* + Left shift '1' to given position and then 'AND'('&').

If the result of the AND operation is 1 then the bit at nth position is set otherwise it is unset.

**5)Divide by 2**:

x = x >> 1;

**Logic**: When we do arithmetic right shift, every bit is shifted to right and blank position is substituted with sign bit of number, 0 in case of positive and 1 in case of negative number. Since every bit is a power of 2, with each shift we are reducing the value of each bit by factor of 2 which is equivalent to division of x by 2.  
**Example**:

x = 18(00010010)  
x >> 1 = 9 (00001001)

**6)Multiplying by 2**:

x = x << 1;

**Logic**: When we do arithmetic left shift, every bit is shifted to left and blank position is substituted with 0 . Since every bit is a power of 2, with each shift we are increasing the value of each bit by a factor of 2 which is equivalent to multiplication of x by 2.  
**Example**:

x = 18(00010010)  
x << 1 = 36 (00100100)

**7)Find log base 2 of a 32 bit integer**:

int log2(int x)

{

int res = 0;

while (x >>= 1)

res++;

return res;

}

**Logic**: We right shift x repeatedly until it becomes 0, meanwhile we keep count on the shift operation. This count value is the log2(x).

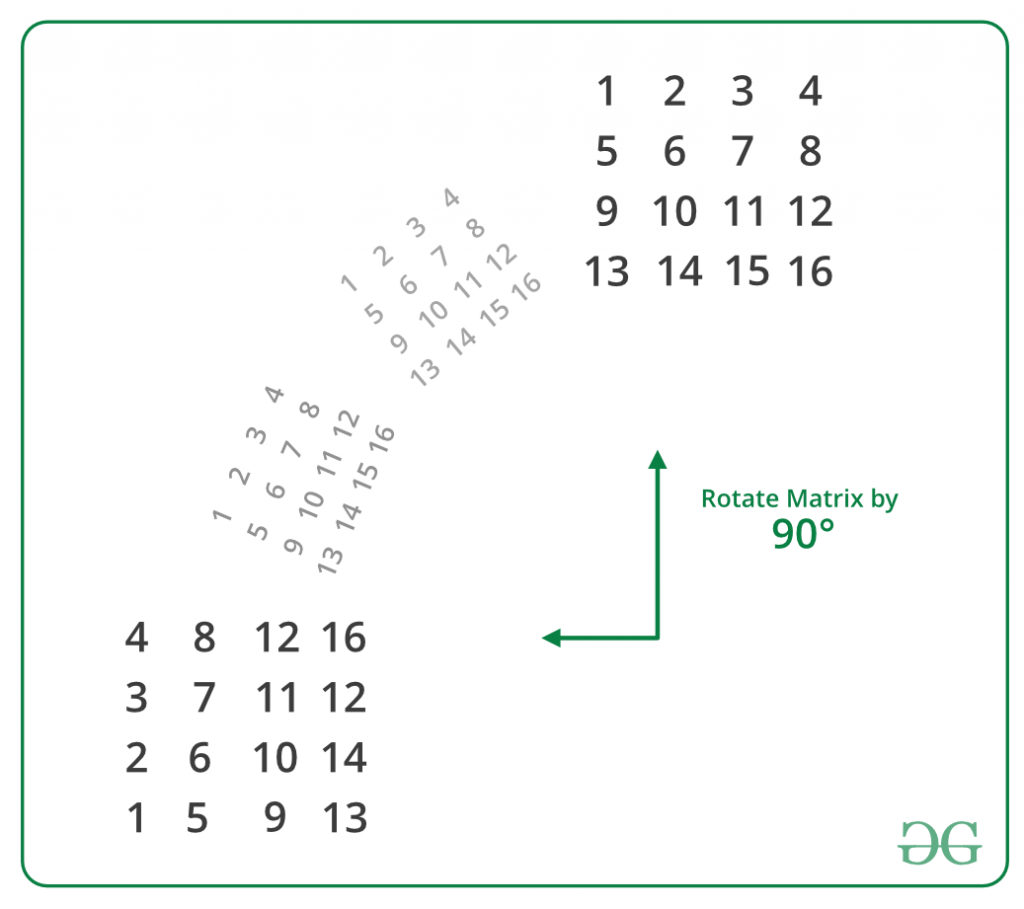
**8)Flipping the bits of a number**: It can be done by a simple way, just simply subtract the number from the value obtained when all the bits are equal to 1 .  
**For example**:

Number : Given Number  
Value : A number with all bits set in given number.  
Flipped number = Value – Number.  
  
Example :   
Number = 23,  
Binary form: 10111;  
After flipping digits number will be: 01000;  
Value: 11111 = 31;

**9)Swapping Two Numbers**: We can easily swap two numbers say **a** and **b** by using the XOR(^) operator by applying below operations:

a ^= b;  
b ^= a;   
a ^= b;

Matrix



**void rotateMatrix(int mat[][N])**

**{**

**// Consider all squares one by one**

**for (int i = 0; i < N / 2; i++)**

**{**

**// Consider elements in group of 4 in**

**// current square**

**for (int j = i; j < N-i-1; j++)**

**{**

**// store current cell in temp variable**

**int temp = mat[i][j];**

**// move values from right to top**

**mat[i][j] = mat[j][N-1-i];**

**// move values from bottom to right**

**mat[j][N-1-i] = mat[N-1-i][N-1-j];**

**// move values from left to bottom**

**mat[N-1-i][N-1-j] = mat[N-1-j][i];**

**// assign temp to left**

**mat[N-1-j][i] = temp;**

**}**

**}**

**}**

**Hashing information**

**Hashing** is a method of storing and retrieving data from a database efficiently.  
  
Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.  
  
With**balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.  
  
Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.  
This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need O(m \* 10n) space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.  
  
Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

[**Hash Function**](http://en.wikipedia.org/wiki/Hash_function)**:** A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.  
A good hash function should have following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.  
  
[**Hash Table**](http://en.wikipedia.org/wiki/Hash_table)**:** An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.  
  
**Collision Handling**: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

* **Chaining:**The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
* **Open Addressing:**In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

**Open Addressing**: Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).  
  
**Important Operations**:

* Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
* Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
* Delete(k): ***Delete operation is interesting***. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.  
  
**Open Addressing is done in following ways**:

1. ***Linear Probing:*** In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.  
   let **hash(x)** be the slot index computed using hash function and **S** be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S  
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S  
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S   
..................................................  
..................................................

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.  
  
[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/08/openAddressing1.png)  
**Clustering:** The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

1. ***Quadratic Probing*** We look for i2'th slot in i'th iteration.

let hash(x) be the slot index computed using hash function.  
If slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S  
If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S  
If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S  
..................................................  
..................................................

1. [**Double Hashing**](https://www.cdn.geeksforgeeks.org/double-hashing/) We use another hash function hash2(x) and look for i\*hash2(x) slot in i'th rotation.

let hash(x) be the slot index computed using hash function.  
If slot hash(x) % S is full, then we try (hash(x) + 1\*hash2(x)) % S  
If (hash(x) + 1\*hash2(x)) % S is also full, then we try (hash(x) + 2\*hash2(x)) % S  
If (hash(x) + 2\*hash2(x)) % S is also full, then we try (hash(x) + 3\*hash2(x)) % S  
..................................................  
..................................................

See [this](https://www.cse.cuhk.edu.hk/irwin.king/_media/teaching/csc2100b/tu6.pdf)for step by step diagrams.

**Comparison of above three:**

* Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
* Quadratic probing lies between the two in terms of cache performance and clustering.
* Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

| S.No. | **Seperate Chaining** | **Open Addressing** |
| --- | --- | --- |
| 1. | Chaining is Simpler to implement. | Open Addressing requires more computation. |
| 2. | In chaining, Hash table never fills up, we can always add more elements to chain. | In open addressing, table may become full. |
| 3. | Chaining is Less sensitive to the hash function or load factors. | Open addressing requires extra care for to avoid clustering and load factor. |
| 4. | Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted. | Open addressing is used when the frequency and number of keys is known. |
| 5. | Cache performance of chaining is not good as keys are stored using linked list. | Open addressing provides better cache performance as everything is stored in the same table. |
| 6. | Wastage of Space (Some Parts of hash table in chaining are never used). | In Open addressing, a slot can be used even if an input doesn't map to it. |
| 7. | Chaining uses extra space for links. | No links in Open addressing |

**Performance of Open Addressing:** Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table  
 n = Number of keys to be inserted in the hash table  
   
 Load factor α = n/m ( < 1 )  
  
 Expected time to search/insert/delete < 1/(1 - α)   
  
 So Search, Insert and Delete take (1/(1 - α)) time

**STRING**

**1**)Check two Strings anagram or not .

Step1:-initialize int array of 255 characters

Step2:-Take first String and increment in 255 initialized int array.

Step3:-Decrement Count in int array using String 2 characters.

Step4:-In int array every index should zero or else it is not anagram.

2)Given a String ,find the leftmost character that repeats.

Step1:-Initialize int array with 256.

Step2:-Traverse from right.

Step2:-For every character increase the count by 1 for the first appearance.

Step3:-If second time appearance is found take index of that character

3)Given a string, find first non repeating character.

Step1:-First initialize array with 256 character.

Step2:-For the first appearance update the index value to 1 or if again character repeat update to 2.

Step 3 :;Last return min index with 0 value.

4)Given a String ,find lexicographic rank of it

Step:- this can be solved using permutation and combination.

EX: STRING

**5)Give a test String and pattern String,find if a permutation of the pattern exists in the text.**

**Step1:Use window search**

**Step2:Count array for 1st window and for second window decrease the count of first character in 1st window.**

**6)Given String if two Strings is rotation of each other**

**Step1:-**Concate s1+s1 then do indexOf check

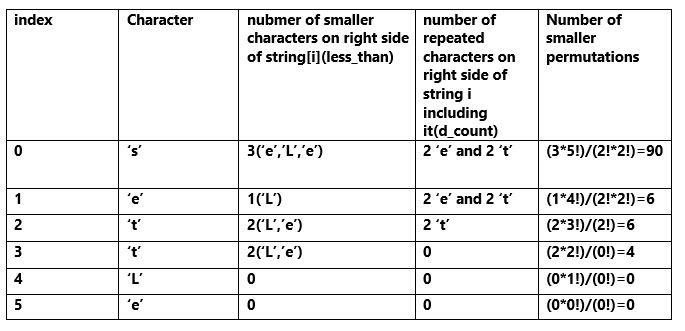
7)Pattren search algorithem

Step1:=Naïve algorithem

Rabin karp alogorithem,KMP(famous),Suffux tree

Lexicographic rank of a string with duplicate characters

**Method:** The method here is little bit different from the without repetition version. Here we have to take care of the duplicate characters also. Let’s look at the string “settLe”. It has repetition(2 ‘e’ and 2 ‘t’) as well as upper case letter(‘L’). Total 6 characters and total number of permutations are 6!/(2!\*2!).  
Now there are 3 characters(2 ‘e’ and 1 ‘L’) on the right side of ‘s’ which come before ‘s’ lexicographically. If there were no repetition then there would be 3\*5! smaller strings which have the first character less than ‘s’. But starting from position 0, till end there are 2 ‘s’ and 2 ‘t'(i.e. repetations). Hence number of possible smaller permutations with first letter smaller than ‘s’ are (3\*5!)/(2!\*2!).  
Similarly if we fix ‘s’ and look at the letters from index 1 to end then there is 1 character(‘L’) lexicographically less than ‘e’. And starting from position 1 there are 2 repeated characters(2 ‘e’ and 2 ‘t’). Hence number of possible smaller permutations with first letter ‘s’ and second letter smaller than ‘e’ are (1\*4!)/(2!\*2!).

Similarly we can form the following table:  


**WorkFlow:**

1. Initialize t\_count(total count) variable

to 1(as rank starts from 1).

2. Run a loop for every character of the string, string[i]:

(i) using a loop count less\_than(number of smaller

characters on the right side of string[i]).

(ii) take one array d\_count of size 52 and using a

loop count the frequency of characters starting

from string[i].

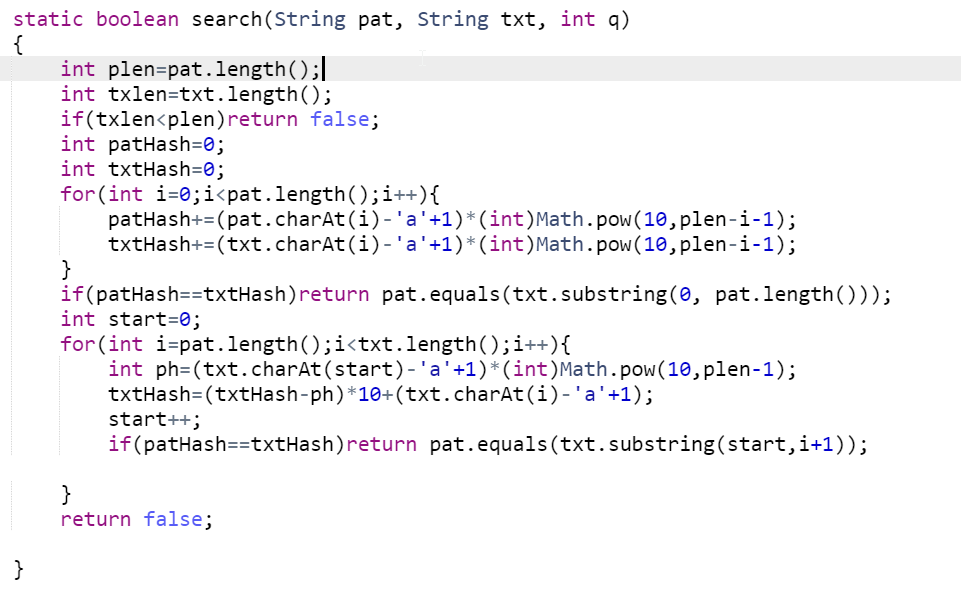
(iii) compute the product, d\_fac(the product of

factorials of each element of d\_count).

(iv) compute (less\_than\*fac(n-i-1))/(d\_fac).

Add it to t\_count.

3. return t\_count

’

**Linked List**

* It does not save data in contiguous location as in array.
* **Program to find the middle of the Linked list**

1)Use count variable and increment mid reference when count is odd

2)Use Fast pointer and slow pointer approach

* **Find the nth node from the end of a linked List**

Solution 1:

1)Traverse the list and find the length of the list.

2)Next traverse upto totallength-givenNthNode+1.

* **Reverse a linked list using recursive and also using while loop.**
* **Detect loop in a linked list.**

Solution1:Add one Boolean variable mark it has visited mean 1 while traversing and check.

Solution2:Point every next nodeto dummy node if next is already pointing dummy node.

Solution 3:using Hashset traverse and put in hashset and check.

Solution 4:Floyd Cycle detection.

* **Detect and Remove loop in linked list.**

Solution1:First detect loop using Floyd Cycle detection next move slow pointer from head one position and fast one position from meeting point

* **Delete node with onle pointer given to it**

Solution1:Copy next node data and copy current node next to next next node

* **Segregate all even and odd values in a linked list.**

Solution: Maintain two pointers even and odd pointer

* **Intersection of two linked list.**

Solution1:Use two Hashset

Solution2

* **Pairwise Swap node of a linked list.**

Solution1:

* **Clone Linked list using random pointer.**

Solution1

**Linked Lists are linear or sequential data structures in which elements are stored at non-contiguous memory location and are linked to each other using pointers.**

Like arrays, linked lists are also linear data structures but in linked lists elements are not stored at contiguous memory locations. They can be stored anywhere in the memory but for sequential access, the nodes are linked to each other using pointers.  
  
Each element in a linked list contains of two parts:

* **Data**: This part stores the data value of the node. That is the information to be stored at the current node.
* **Next Pointer**: This is the pointer variable or any other variable which stores the address of the next node in the memory.



**Advantages of Linked Lists over Arrays**: Arrays can be used to store linear data of similar types, but arrays have the following limitations:

1. The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage. On the other hand, linked lists are dynamic and the size of the linked list can be incremented or decremented during runtime.
2. Inserting a new element in an array of elements is expensive, because a room has to be created for the new elements and to create room, existing elements have to shift.  
     
   For example, in a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.  
  
On the other hand, nodes in linked lists can be inserted or deleted without any shift operation and is efficient than that of arrays.

**Disadvantages of Linked Lists**:

1. Random access is not allowed in Linked Lists. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists efficiently with its default implementation. Therefore, lookup or search operation is costly in linked lists in comparison to arrays.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache friendly. Since array elements are present at contiguous locations, there is a locality of reference which is not there in case of linked lists.

**Advantages of doubly linked lists over singly linked list**:

1. A DLL can be traversed in both forward and backward direction.
2. The delete operation in DLL is more efficient if the pointer to the node to be deleted is given.
3. We can quickly insert a new node before a given node.
4. In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

**Disadvantages of doubly linked lists over singly linked list**:

1. Every node of DLL Require extra space for a previous pointer.
2. All operations require an extra pointer previous to be maintained. For example, an insertion, we need to modify previous pointers together with next pointers.

**XOR Linked Lists** are Memory Efficient implementation of Doubly Linked Lists. An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.

**STACK**

The ***Stack*** is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

* The LIFO order says that the element which is inserted at the last in the Stack will be the first one to be removed. In LIFO order insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.
* The FILO order says that the element which is inserted at the first in the Stack will be the last one to be removed. In FILO order insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.

Mainly the following three basic operations are performed in the stack:

* **Push:**Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
* **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
* **Peek or Top:** Returns top element of stack.

* **isEmpty:**Returns true if stack is empty, else false.



**How to understand a stack practically?**

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.  
  
**Time Complexities of operations on stack:** The operations push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.  
  
  
**Implementation:** There are two ways to implement a stack.

* Using array
* Using linked list

**Pros:** Easy to implement. Memory is saved as pointers are not involved.  
**Cons:** It is not dynamic. It doesn’t grow and shrink depending on needs at runtime**os:** The linked list implementation of stack can grow and shrink according to the needs at runtime.  
**Cons:** Requires extra memory due to involvement of pointers.

**Applications of stack:**

* Stacks can be used to check for the balancing of paranthesis in an expression.
* Infix to Postfix/Prefix conversion.
* Redo-undo features at many places like editors, photoshop.
* Forward and backward feature in web browsers

**Infix expression:** The expression of the form *a op b*. When an operator is in-between every pair of operands.  
  
**Postfix expression:** The expression of the form *a b op*. When an operator is followed for every pair of operands.  
  
**Why postfix representation of the expression?** The compiler scans the expression either from left to right or from right to left.  
  
Consider the below expression:

a op1 b op2 c op3 d  
  
If op1 = +, op2 = \*, op3 = +

The compiler first scans the expression to evaluate the expression b \* c, then again scan the expression to add a to it. The result is then added to d after another scan.  
  
The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.  
  
The corresponding expression in postfix form is:**abc\*+d+**. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.  
  
**Algorithm to Convert an Infix expression to Postfix:**

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,  
   * If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '(' ), push it.
   * Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output.
8. Pop and output from the stack until it is not empty.
9. For conversion we use precedence and associativity(left-right,right-left) rules.

**1)Program to implement k stacks in an array.**

**// Java implementation to convert infix expression**

**// to equivalent postfix expression**

**// Note that here we have used the Stack class**

**// for all Stack operations**

**import java.util.Stack;**

**class Test**

**{**

**// A utility function to return precedence**

**// of a given operator**

**// Higher the returned value means**

**// higher the precedence**

**static int Prec(char ch)**

**{**

**switch (ch)**

**{**

**case '+':**

**case '-':**

**return 1;**

**case '\*':**

**case '/':**

**return 2;**

**case '^':**

**return 3;**

**}**

**return -1;**

**}**

**// The main method that converts given**

**// infix expression to postfix expression.**

**static String infixToPostfix(String exp)**

**{**

**// initializing empty String for result**

**String result = new String("");**

**// initializing empty stack**

**Stack<Character> stack = new Stack<>();**

**for (int i = 0; i<exp.length(); ++i)**

**{**

**char c = exp.charAt(i);**

**// If the scanned character is an operand,**

**// add it to output.**

**if (Character.isLetterOrDigit(c))**

**result += c;**

**// If the scanned character is an '(',**

**// push it to the stack.**

**else if (c == '(')**

**stack.push(c);**

**// If the scanned character is an ')',**

**// pop and output from the stack**

**// until an '(' is encountered.**

**else if (c == ')')**

**{**

**while (!stack.isEmpty() && stack.peek() != '(')**

**result += stack.pop();**

**if (!stack.isEmpty() && stack.peek() != '(')**

**return "Invalid Expression"; // invalid expression**

**else**

**stack.pop();**

**}**

**else // an operator is encountered**

**{**

**while (!stack.isEmpty() && Prec(c) <= Prec(stack.peek()))**

**result += stack.pop();**

**stack.push(c);**

**}**

**}**

**// pop all the operators from the stack**

**while (!stack.isEmpty())**

**result += stack.pop();**

**return result;**

**}**

**// Driver method**

**public static void main(String[] args)**

**{**

**String exp = "a+b\*(c^d-e)^(f+g\*h)-i";**

**System.out.println(infixToPostfix(exp));**

**}**

**}**

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have already discussed the conversion of infix to postfix expressions. In this post, the next step after that, that is evaluating a postfix expression is discussed.  
  
  
Following is the algorithm for evaluation of postfix expressions:

1. Create a stack to store operands (or values).
2. Scan the given expression and do following for every scanned element.  
   * If the element is a number, push it into the stack.
   * If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
3. When the expression is ended, the number in the stack is the final answer.

**Example:** Let the given expression be "***2 3 1 \* + 9 -***". We will first scan all elements one by one.

1. Scan '2', it's a number, so push it to stack. Stack contains '2'
2. Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)
3. Scan '1', again a number, push it to stack, stack now contains '2 3 1'
4. Scan '\*', it's an operator, pop two operands from the stack, apply the \* operator on operands, we get 3\*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.
5. Scan '+', it's an operator, pop two operands from the stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes '5'.
6. Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9'.
7. Scan '-', it's an operator, pop two operands from the stack, apply the - operator on operands, we get 5 - 9 which results in -4. We push the result '-4' to stack. Stack now becomes '-4'.
8. There are no more elements to scan, we return the top element from the stack (which is the only element left in the stack).

**// Java proram to evaluate value of a postfix expression**

**import java.util.Stack;**

**public class Test**

**{**

**// Method to evaluate value of a postfix expression**

**static int evaluatePostfix(String exp)**

**{**

**// create a stack**

**Stack<Integer> stack=new Stack<>();**

**// Scan all characters one by one**

**for(int i=0;i<exp.length();i++)**

**{**

**char c=exp.charAt(i);**

**// If the scanned character is an operand (number here),**

**// push it to the stack.**

**if(Character.isDigit(c))**

**stack.push(c - '0');**

**// If the scanned character is an operator, pop two**

**// elements from stack apply the operator**

**else**

**{**

**int val1 = stack.pop();**

**int val2 = stack.pop();**

**switch(c)**

**{**

**case '+':**

**stack.push(val2+val1);**

**break;**

**case '-':**

**stack.push(val2- val1);**

**break;**

**case '/':**

**stack.push(val2/val1);**

**break;**

**case '\*':**

**stack.push(val2\*val1);**

**break;**

**}**

**}**

**}**

**return stack.pop();**

**}**

**// Driver program to test above functions**

**public static void main(String[] args)**

**{**

**String exp="231\*+9-";**

**System.out.println("postfix evaluation: "+evaluatePostfix(exp));**

**}**

**}**

**In case of Evaluation of prefix expression we will traverse from right to left**

**Queue**

Like *Stack*data structure, ***Queue***is also a linear data structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO) which means that the element which is inserted first in the queue will be the first one to be removed from the queue. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.  
  
The difference between stacks and queues is in removing. In a stack we remove the most recently added item; in a queue, we remove the least recently added item.  
  
**Operations on Queue:** Mainly the following four basic operations are performed on queue:

* **Enqueue:**Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
* **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
* **Front:**Get the front item from queue.
* **Rear:** Get the last item from queue.

**Tree**

**1)Given binary tree find the maximum height of the tree.**

**2)Given a binary tree, check if it follows children sum property.**

**3)Given a binary tree is balanced or not**

**4)print all nodes at the level k**

**5)check if given tree is binary search tree or not**

**6)Given a binary tree convert it into doubly linked list**

**7)LCA of two nodes**

**8)Maximum sum from root to leaf path**

**9)Construct binary tree from in-order and post-order traversal**

**10)Threaded binary tree**

**BLANCED BINARY TREE**

**Binary Search Tree** is a node-based binary tree data structure which has the following properties:

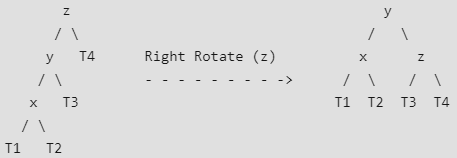
* The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
* The right subtree of a node contains only nodes with keys greater than the node's key.
* The left and right subtree each must also be a binary search tree.  
  There must be no duplicate nodes.

**AVL Tree**

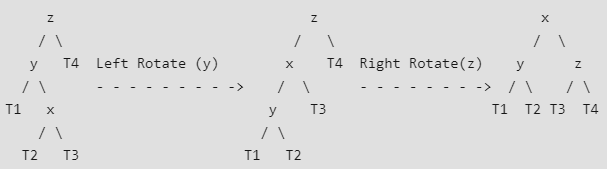
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

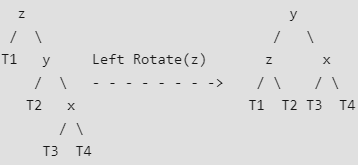
**a) Left Left Case**

T1, T2, T3 and T4 are subtrees.  


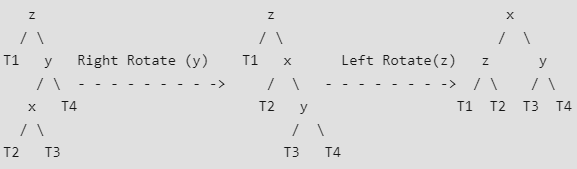
**b) Left Right Case**

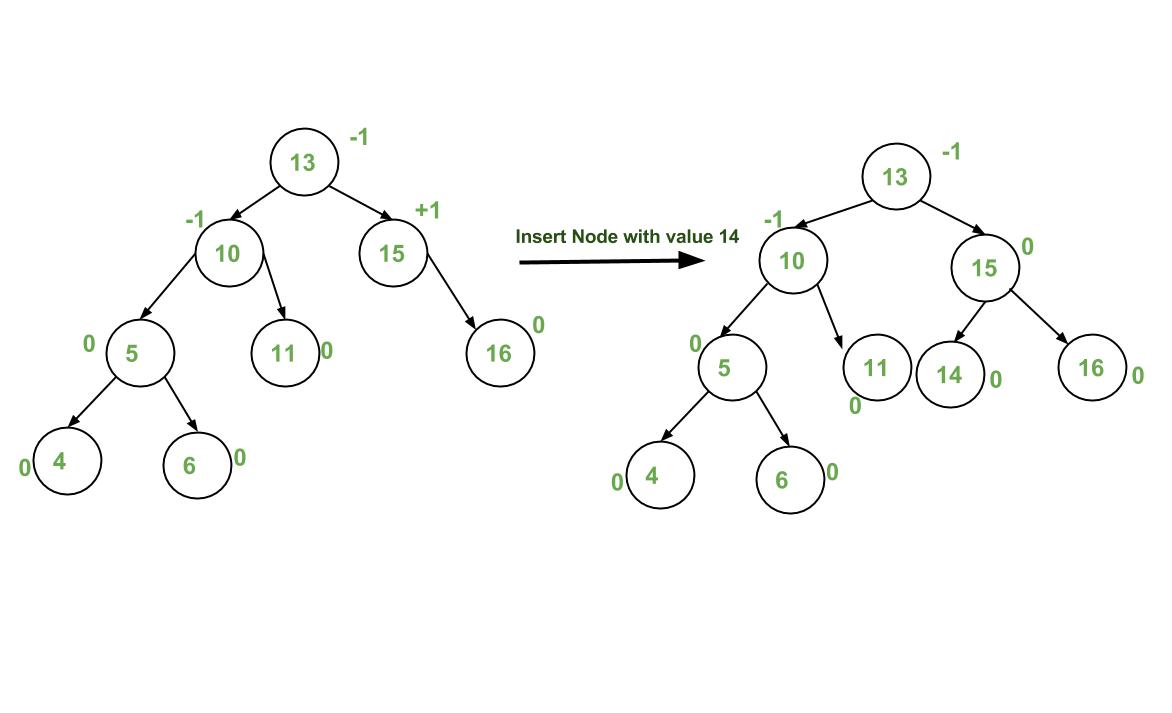
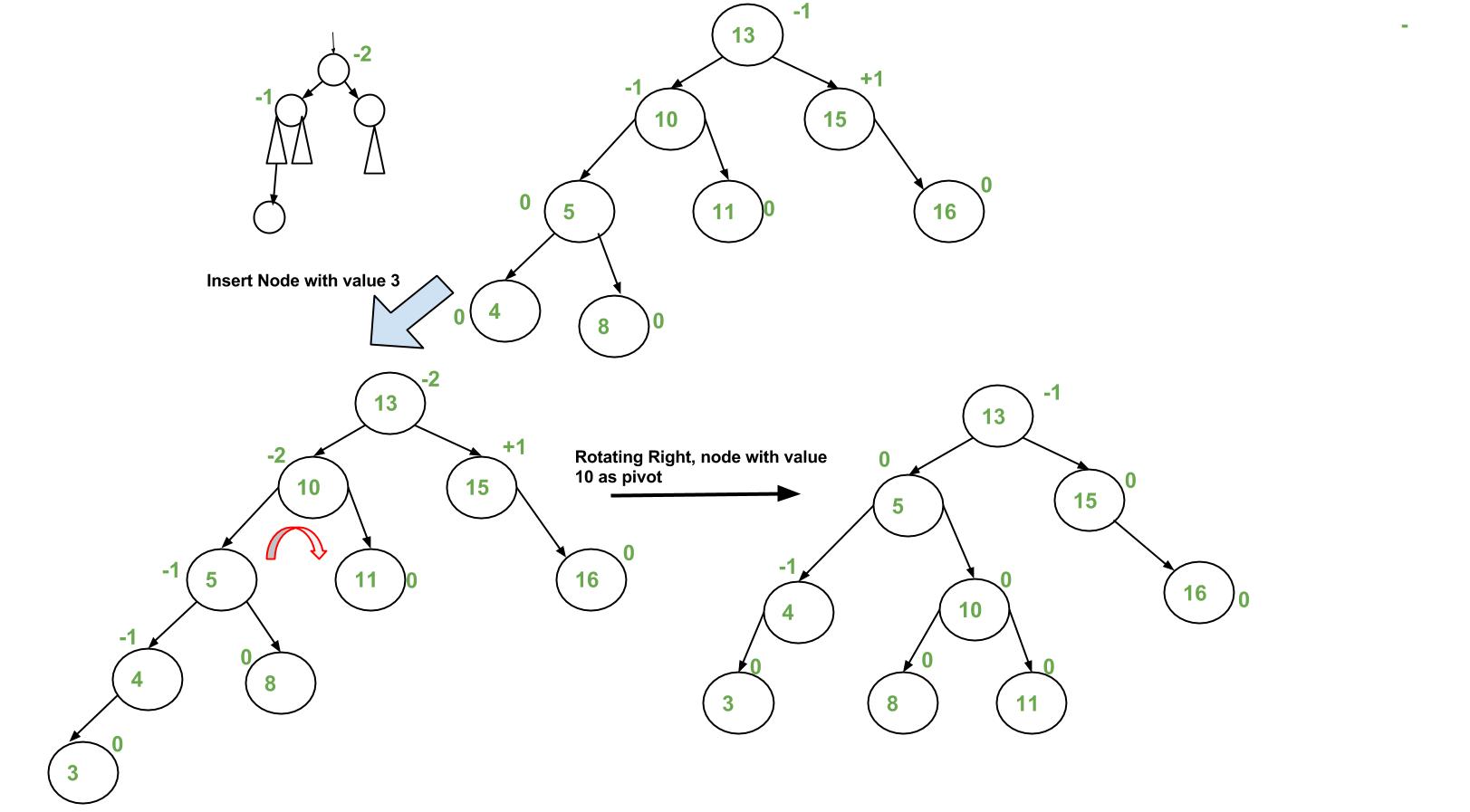
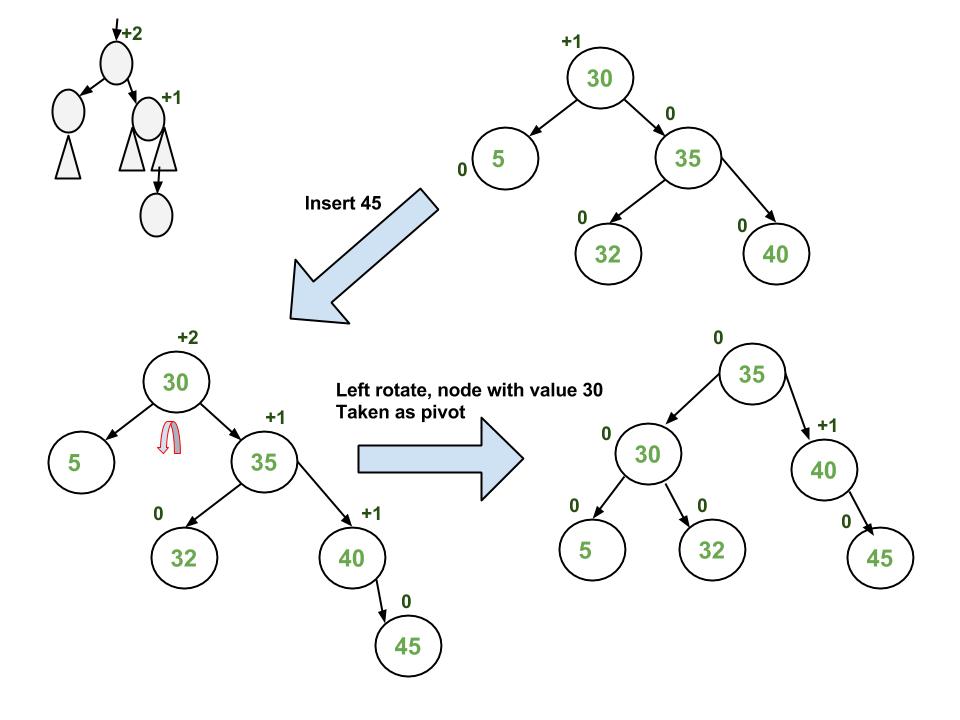
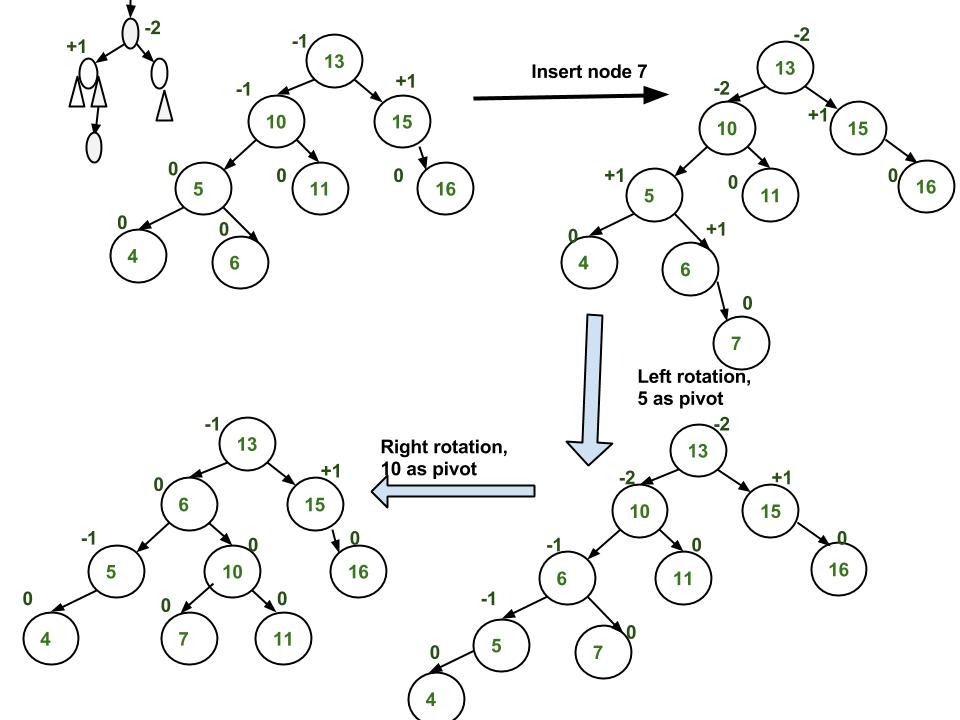
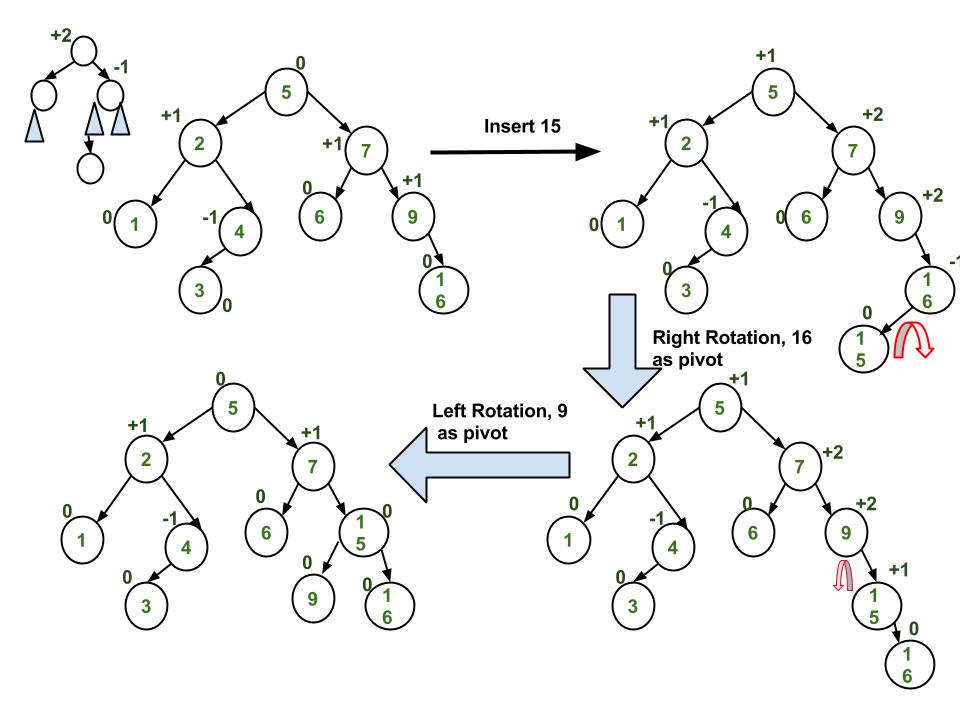


**c) Right Right Case**



**d) Right Left Case**



**Insertion Examples:**[](https://media.geeksforgeeks.org/wp-content/uploads/AVL-Insertion-1.jpg)  
[](https://media.geeksforgeeks.org/wp-content/uploads/AVL-Insertion1-1.jpg)  
[](https://media.geeksforgeeks.org/wp-content/uploads/AVL_INSERTION2-1.jpg)  
[](https://media.geeksforgeeks.org/wp-content/uploads/AVL_Insertion_3-1.jpg)  
[](https://media.geeksforgeeks.org/wp-content/uploads/AVL_Tree_4-1.jpg)  
**Time Complexity:** The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since the AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

**RED BLACK Tree**

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.  
[](https://www.geeksforgeeks.org/wp-content/uploads/RedBlackTree.png)  
**1)**Every node has a color either red or black.

**2)**Root of tree is always black.

**3)**There are no two adjacent red nodes (A red node cannot have a red parent or red child).

**4)**Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.