UNIT - I

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks-Operations, array and linked representations of stacks, stack applications, Queues-operations, array and linked representations.

Data Structure

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned.

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process
in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

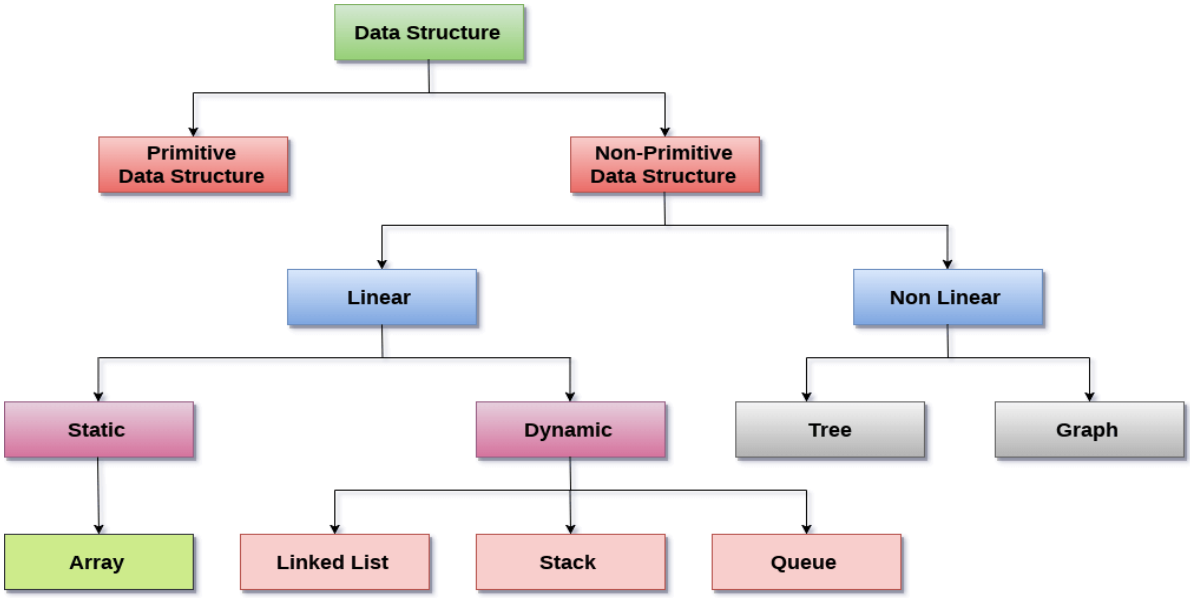
Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Linear Data Structures

If a data structure organizes the data in sequential order, then that data structure is called a Linear DataStructure.

Example

1. Arrays
2. List (Linked List)
3. Stack
4. Queue

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.
A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.
It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures:
This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Non - Linear Data Structures

If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure.

Example

1. Tree
2. Graph
3. Dictionaries
4. Heaps
5. Tries, Etc.,

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.
Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.
If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.
If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

Abstract Data Type:

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called information hiding. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

Stack

A Stack is linear data structure. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the **top of the stack**. Stack principle is **LIFO (last in, first out)**. Which element inserted last on to the stack that element deleted first from the stack.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

Real life examples of stacks are:

Operations on stack:

The two basic operations associated with stacks are:

1. Push
2. Pop

While performing push and pop operations the following test must be conducted on the stack.

a) Stack is empty or not b) stack is full or not

1. **Push:** Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

2. **Pop:** Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack (or) Implementation of stack:

The stack should be represented in two ways:

1. Stack using array
2. Stack using linked list

1. Stack using array:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a ***stack overflow*** condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a ***stack underflow*** condition.

1. **push():** When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().

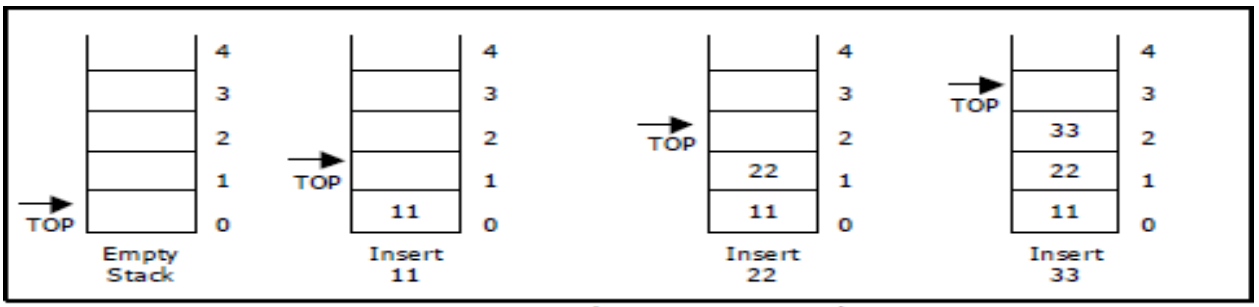


Figure . Push operations on stack

Initially **top=-1**, we can insert an element in to the stack, increment the top value i.e **top=top+1**. We can insert an element in to the stack first check the condition is stack is full or not. i.e **top>=size-1**. Otherwise add the element in to the stack.

Algorithm: Procedure for push():

Step 1: START
Step 2: if top>=size-1 then
Write " Stack is Overflow"
Step 3: Otherwise
3.1: read data value 'x'
3.2: top=top+1;
3.3: stack[top]=x;
Step 4: END

2.Pop(): When an element is taken off from the stack, the operation is performed by pop(). Below

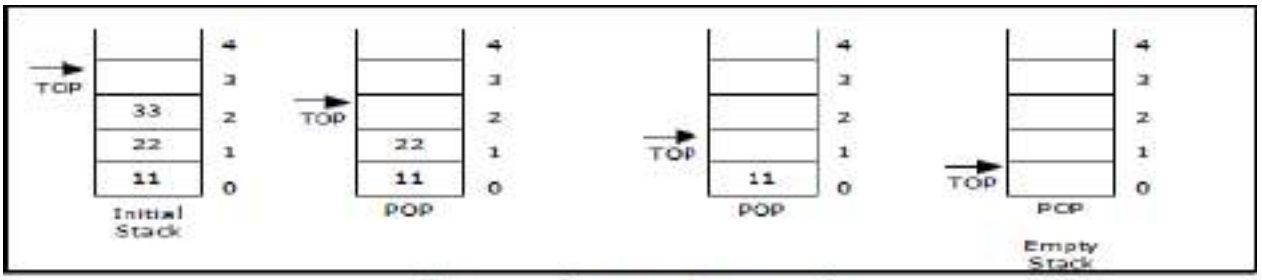


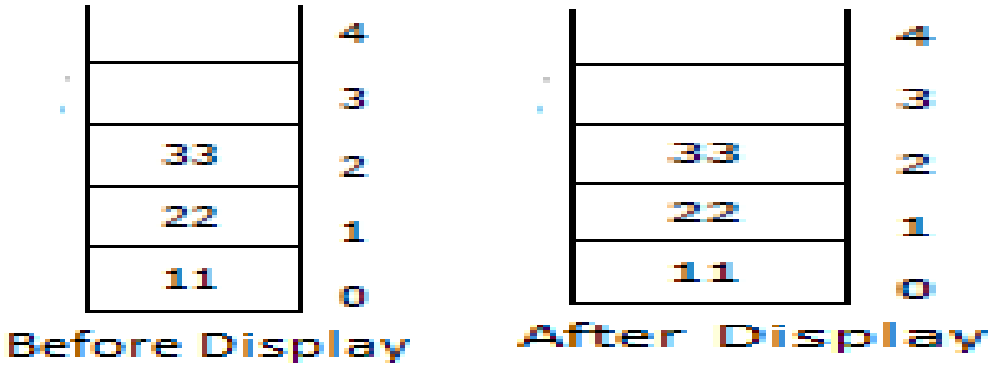
Figure Pop operations on stack

figure shows a stack initially with three elements and shows the deletion of elements using pop().

We can insert an element from the stack, decrement the top value i.e **top=top-1**.
We can delete an element from the stack first check the condition is stack is empty or not.
i.e **top==-1**. Otherwise remove the element from the stack.

Algorithm: procedure pop():
Step 1: START
Step 2: if top==-1 then
Write "Stack is Underflow"
Step 3: otherwise
3.1: print "deleted element"
3.2: top=top-1;
Step 4: END

3.display(): This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.



Algorithm: procedure pop():
Step 1: START
Step 2: if top==-1 then
Write "Stack is Underflow"
Step 3: otherwise
3.1: print "Display elements are"
3.2: for top to 0
Print 'stack[i]'
Step 4: END

Stack Implementation Using Arrays

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void display();
void main ()
{

   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   printf("*********Stack operations using array\n*********");
   while(choice != 4)
   {
      printf("Chose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("Exiting....");
            break;
         }
         default:
         {
            printf("Please Enter valid choice ");
         }
      };
   }
```

```
}
  void push ()
{
    int val;
    if (top == n )
    printf("\n Stack Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}
  void pop ()
{
    if(top == -1)
    printf("Stack Underflow");
    else
    top = top -1;
}
void display()
{
   if(top == -1)
    {
        printf("Stack is empty");
    }
    printf("stack elements are\n ")
    for (i=top;i>=0;i--)
    {
        printf(" %d ",stack[i]);
    }

}
```

OUTPUT:

```
Enter the number of elements in the stack 5
*********Stack operations using array*********
----------------------------------------------
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
5
```

```
Please Enter valid choice Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
stack elements are
 12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
stack elements are
 12
12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
4
Exiting....
```

Applications of STACK:

Application of Stack :

- Recursive Function.
- Expression Evaluation.
- Expression Conversion.
  - Infix to postfix
  - Infix to prefix
  - Postfix to infix
  - Postfix to prefix
  - Prefix to infix
  - Prefix to postfix
- Reverse a Data
- Processing Function Calls

Expressions:

- An expression is a collection of operators and operands that represents a specific value.
- Operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,
- Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location

Expression types:

Based on the operator position, expressions are divided into THREE types. They are as follows.

- **Infix Expression**
  - In infix expression, operator is used in between operands.
  - Syntax : operand1 operator operand2
  - Example

- **Postfix Expression**

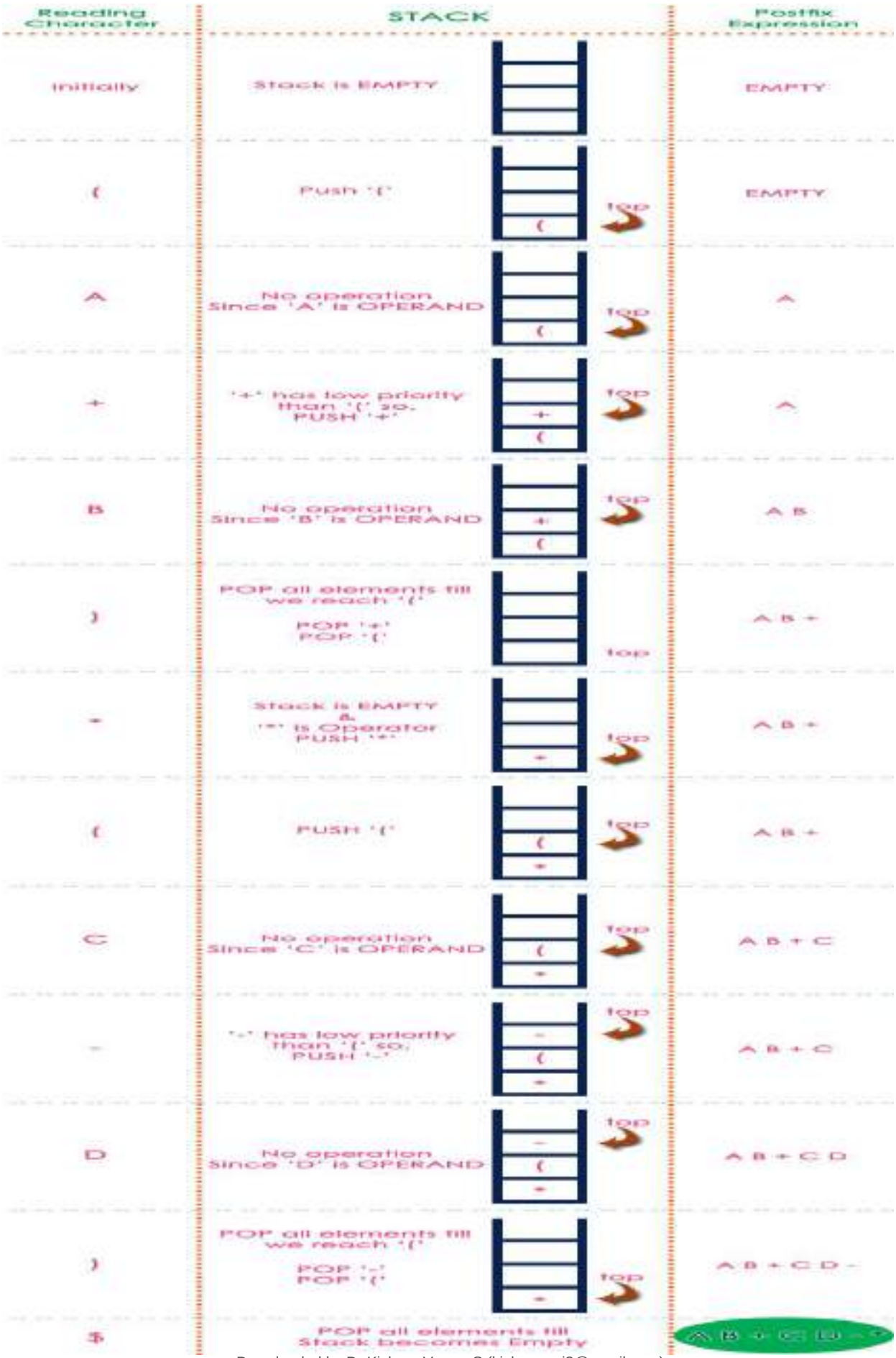
- In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".
- Syntax : operand1 operand2 operator
- Example:

- **Prefix Expression**
  - In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".
  - Syntax : operator operand1 operand2
  - Example:

Infix to postfix conversion using stack:

- Procedure to convert from infix expression to postfix expression is as follows:
- Scan the infix expression from left to right.
- If the scanned symbol is left parenthesis, push it onto the stack.
- If the scanned symbol is an operand, then place directly in the postfix expression (output).
- If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
- If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example-1


Reading Character
STACK
Postfix Expression
Initially
Stack is EMPTY
EMPTY
(
Push '('
top
EMPTY
A
No operation
Since 'A' is OPERAND
top
A
+
'+' has low priority
than '(' so,
PUSH '+'
top
A
B
No operation
Since 'B' is OPERAND
top
A B
)
POP all elements till
we reach '('
POP '+'
POP '('
top
A B +
*
Stack is EMPTY
&
'*' is Operator
PUSH '*'
top
A B +
(
PUSH '('
top
A B +
C
No operation
Since 'C' is OPERAND
top
A B + C
-
'-' has low priority
than '(' so,
PUSH '-'
top
A B + C
D
No operation
Since 'D' is OPERAND
top
A B + C D
)
POP all elements till
we reach '('
POP '-'
POP '('
top
A B + C D -
$
POP all elements till
Stack becomes Empty
A B + C D - *

Example2:
Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example3

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	
*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	

(	a b c * +	+ (	
d	a b c * + d	+ (	
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A+(B *C–(D/E↑F)*G)*H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Evaluation of postfix expression:

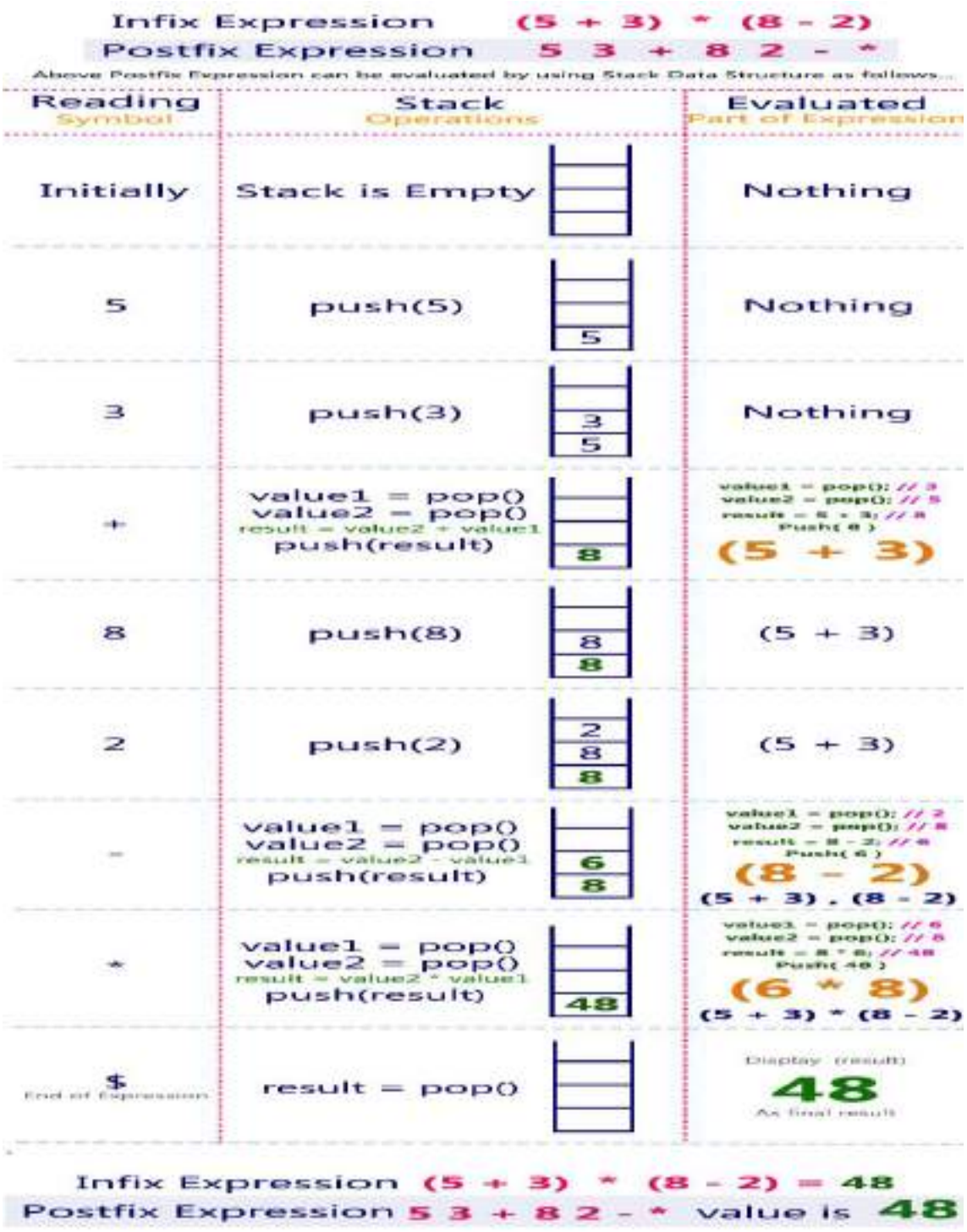
- The postfix expression is evaluated easily by the use of a stack.
- When a number is seen, it is pushed onto the stack;
- when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
- When an expression is given in postfix notation, there is no need to know any precedence rules.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example2


Infix Expression (5 + 3) * (8 - 2)
Postfix Expression 5 3 + 8 2 - *
Above Postfix Expression can be evaluated by using Stack Data Structure as follows...
Reading Symbol
Stack Operations
Evaluated Part of Expression
Initially
Stack is Empty
Nothing
5
push(5)
Nothing
3
push(3)
Nothing
+
value1 = pop()
value2 = pop()
result = value2 + value1
push(result)
value1 = pop(); // 3
value2 = pop(); // 5
result = 5 + 3; // 8
Push(8)
(5 + 3)
8
push(8)
(5 + 3)
2
push(2)
(5 + 3)
-
value1 = pop()
value2 = pop()
result = value2 - value1
push(result)
value1 = pop(); // 2
value2 = pop(); // 8
result = 8 - 2; // 6
Push(6)
(8 - 2)
(5 + 3) , (8 - 2)
*
value1 = pop()
value2 = pop()
result = value2 * value1
push(result)
value1 = pop(); // 6
value2 = pop(); // 8
result = 8 * 6; // 48
Push(48)
(6 * 8)
(5 + 3) * (8 - 2)
$
End of Expression
result = pop()
Display (result)
48
As final result
Infix Expression (5 + 3) * (8 - 2) = 48
Postfix Expression 5 3 + 8 2 - * value is 48

Example 3:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

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.

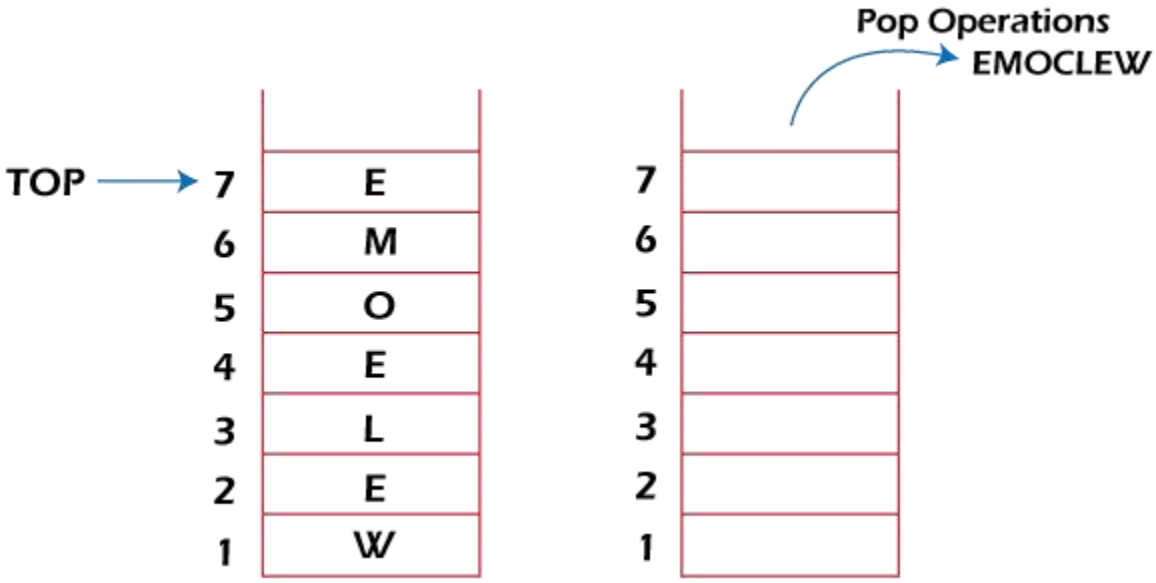
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

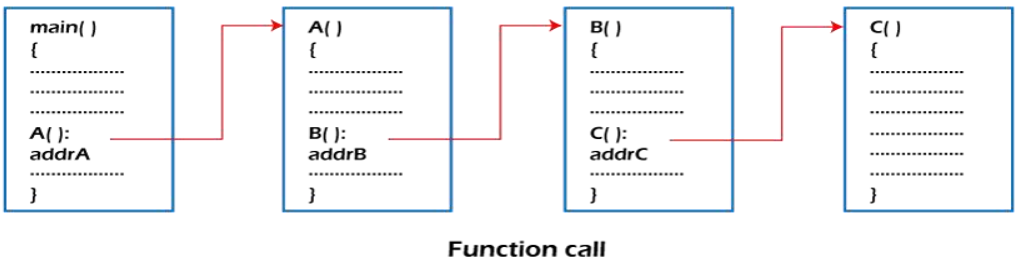
Reverse a String

A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



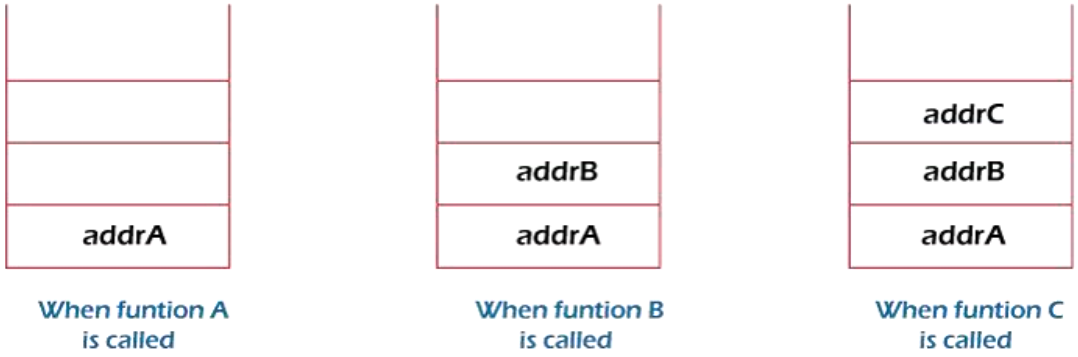
Processing Function Calls:

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.



When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.



Different states of stack

The above figure shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack. Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

QUEUE

A queue is linear data structure and collection of elements. A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. The principle of queue is a "FIFO" or "First-in-first-out".

Queue is an abstract data structure. A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

More real-world examples can be seen as queues at the ticket windows and bus-stops and our college library.

The operations for a queue are analogues to those for a stack; the difference is that the insertions go at the end of the list, rather than the beginning.

Operations on QUEUE:

A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

- **Enqueue or insertion**: which inserts an element at the end of the queue.
- **Dequeue or deletion**: which deletes an element at the start of the queue.

Representation of Queue (or) Implementation of Queue:

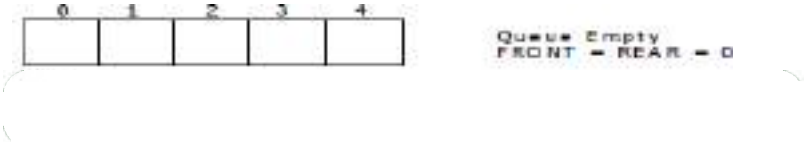
The queue can be represented in two ways:

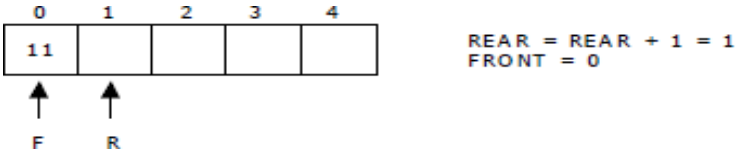
1. Queue using Array
2. Queue using Linked List

1.Queue using Array:

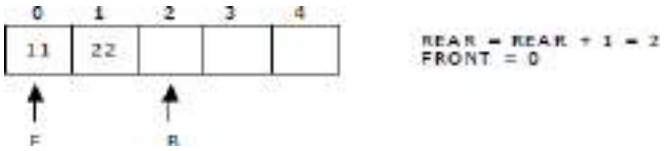
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.

Now, insert 11 to the queue. Then queue status will be:

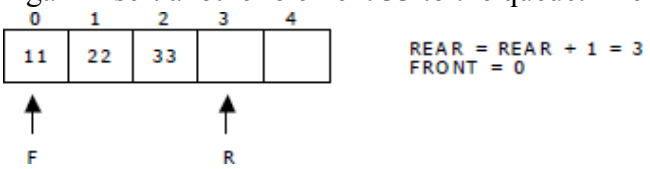




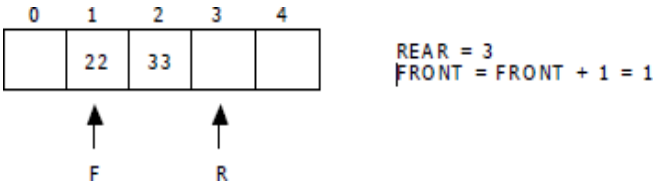
Next, insert 22 to the queue. Then the queue status is:



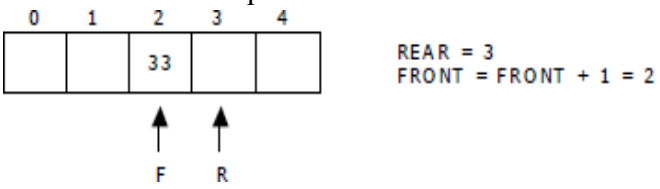
Again insert another element 33 to the queue. The status of the queue is:



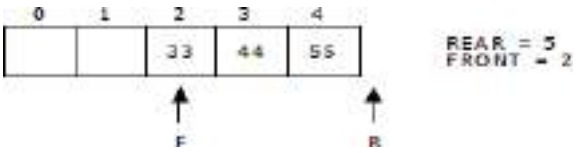
Now, delete an element. The element deleted is the element at the front of the queue.So the status of the queue is:



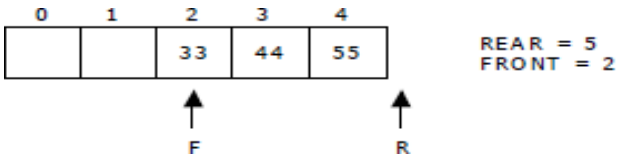
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



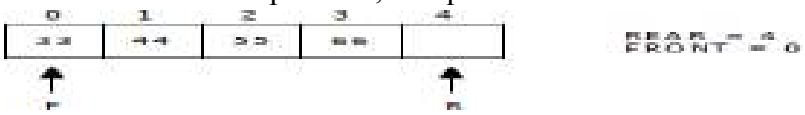
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time**.**

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
- **Step 2:** EXIT

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- **Step 1 -** Check whether **queue** is **EMPTY**.
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

Queue Implementation using Arrays

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
   int choice;
   while(choice != 4)
   {
```

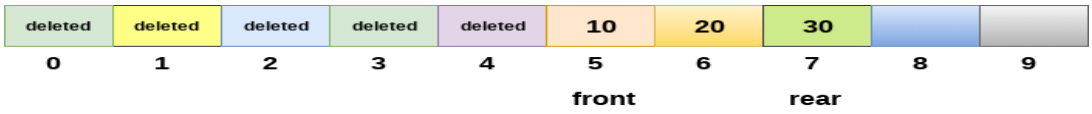
```
        printf("\n*************************Main
Menu*****************************\n");

printf("\n================================================================
===\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
```

```
    queue[rear] = item;
    printf("\nValue inserted ");

}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;

    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
            front = front + 1;
        }
        printf("\nvalue deleted ");
    }


}

void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        for(i=front;i<=rear;i++)
        {
            printf("\n%d\n",queue[i]);
        }
    }
}
```

Drawback of array implementation of Queue

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

One of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Types of Queues

There are four types of Queues**:**

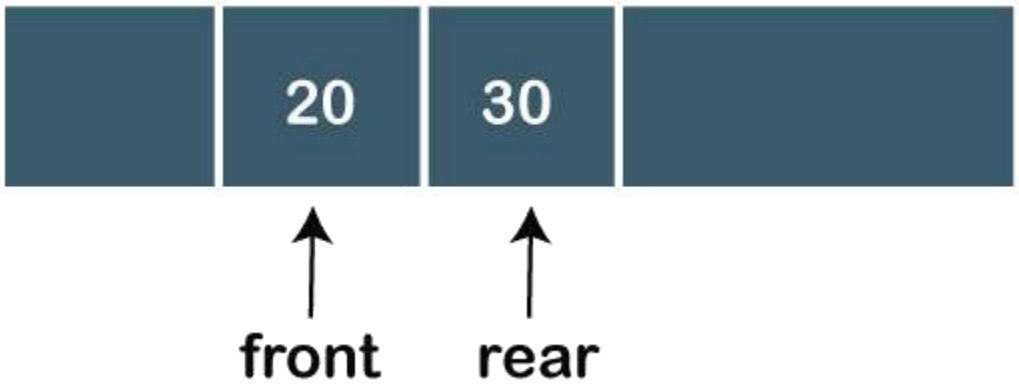
1. Linear Queue
2. Circular Queue
3. Priority Queue
4. Deque

1. **Linear Queue**

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule. The linear Queue can be represented, as shown in the below

figure:

The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:

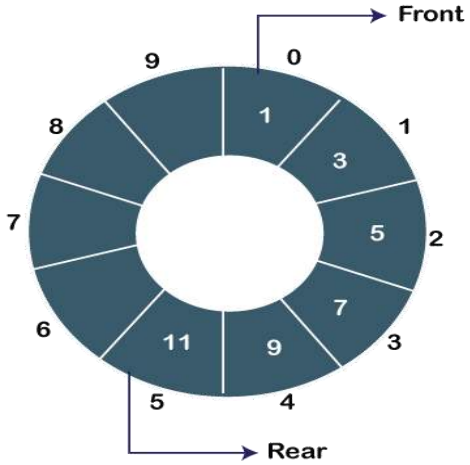


In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

2. Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer as all the ends are connected to another end. The circular queue can be represented as:



he drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

3. Priority Queue

A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:

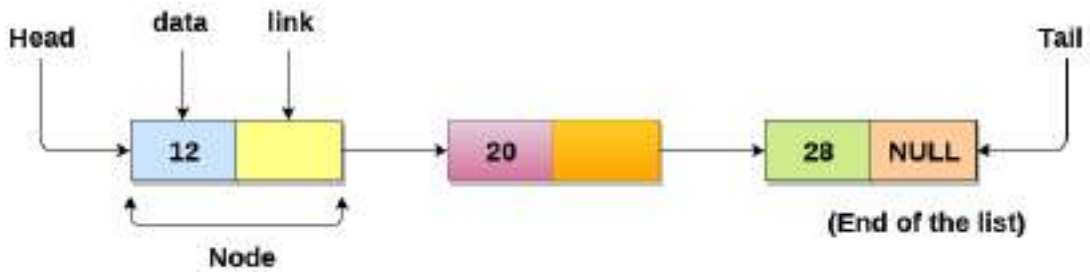
The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

4. Deque

Both the Linear Queue and Deque are different as the linear queue follows the FIFO principle whereas, deque does not follow the FIFO principle. In Deque, the insertion and deletion can occur from both ends.

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can't be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Differences between the array and linked list in a tabular form.

ARRAYS	LINKED LISTS
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address
Array elements store in a contiguous memory location	Linked list elements can be stored anywhere in the memory or randomly stored
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time	In the case of a linked list, memory is allocated at run time
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement

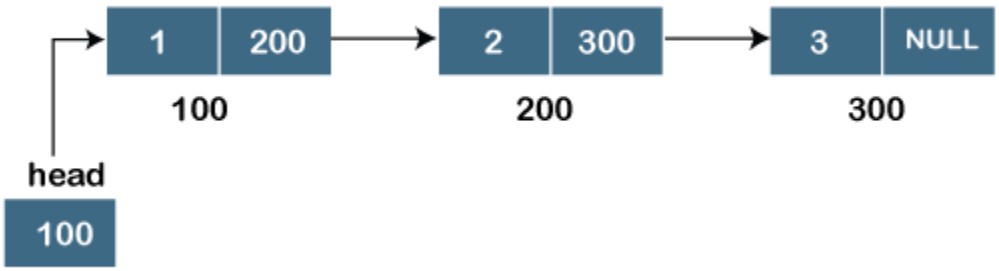
Types of Linked List

The following are the types of linked list:

1. **Singly linked list**
2. **Doubly linked list**
3. **Circular linked list**

Singly Linked list

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a pointer.
Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



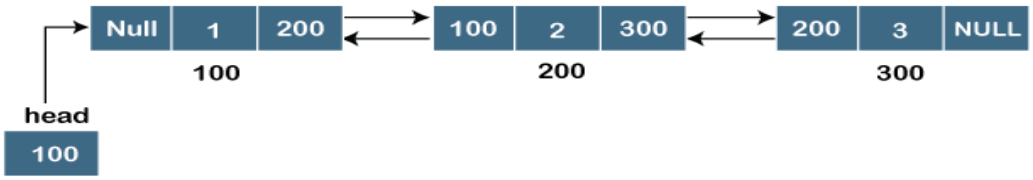
We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a ***head pointer***.
The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.
Representation of the node in a singly linked list
struct node

```
{
  int data;
  struct node *next;
}
```

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Doubly linked list

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.
Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

```
struct node
{
 int data;
 struct node *next;
 struct node *prev;
}
```

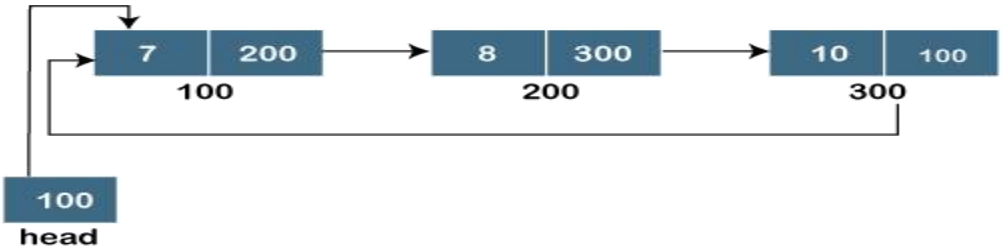
In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next and prev** is **struct node** as both the pointers are storing the address of the node of the ***struct node*** type.

Circular linked list

A circular linked list is a variation of a singly linked list. The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

```
struct node
        {
          int data;
          struct node *next;
        }
```

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:

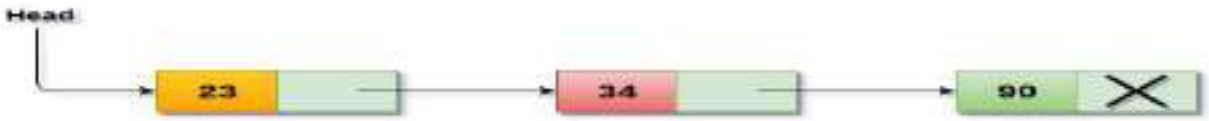


Singly linked list

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

1. Inserting at Beginning
2. Inserting at the End of the LIst
3. Inserting after specified node

Insertion in singly linked list at beginning

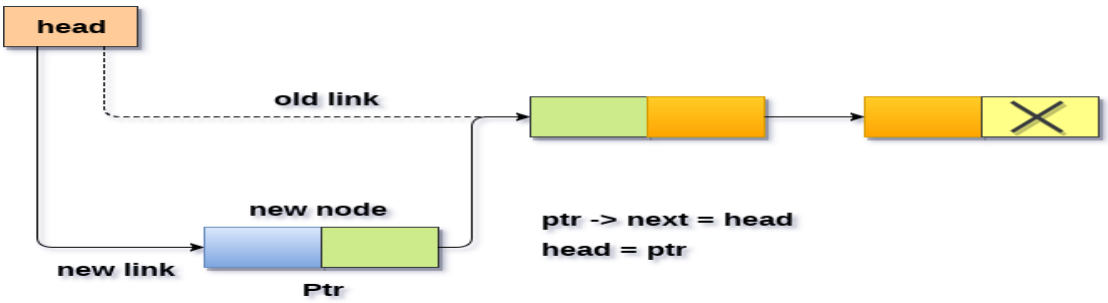
Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need to be followed in order to inser a new node in the list at beginning.

1. Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.
   ptr = (struct node *) malloc(sizeof(struct node *));
   ptr → data = item
2. Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.

   ptr->next = head

3. At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

head = ptr;



Algorithm

- **Step 1:** IF PTR = NULL
  Write OVERFLOW
  Go to Step 7
  [END OF IF]
- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR → NEXT
- **Step 4:** SET NEW_NODE → DATA = VAL
- **Step 5:** SET NEW_NODE → NEXT = HEAD
- **Step 6:** SET HEAD = NEW_NODE
- **Step 7:** EXIT

Function for inserting element at beginning of the list

```
void beginsert()
{
   struct node *ptr;
   int item;
   ptr = (struct node *) malloc(sizeof(struct node *));
   if(ptr == NULL)
   {
      printf("\n memory insufficient to allocate");
   }
   else
   {
      printf("\nEnter value\n");
      scanf("%d",&item);
      ptr->data = item;
      ptr->next = head;          head = ptr;
```

```
        printf("\nNode inserted");
      }
    }
```

2.Inserting at the End of the List

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list(CASE 1)
2. The node is being added to the end of the linked list(CASE2)

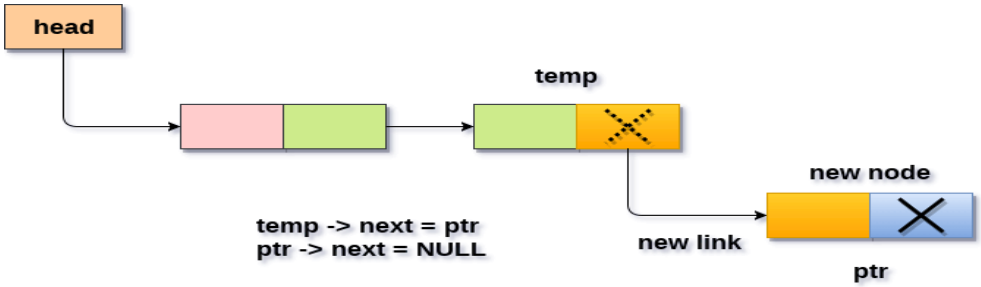
in the first case,(CASE1)

- The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node by using malloc statement in C. Data and the link part of the node are set up by using the following statements.
  ptr->data = item;
  ptr -> next = NULL;
- Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list. This will be done by using the following Statements.
  Head = ptr

In the second case: CASE(2):

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.
  Temp = head
- Then, traverse through the entire linked list using the statements:
  while (temp→ next != NULL)
  temp = temp → next;
- At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**. We need to make the next part o
- If the temp node (which is currently the last node of the list) point to the new node (ptr)
  .temp = head;

```
    while (temp -> next != NULL)
    {
      temp = temp -> next;
    }
    temp->next = ptr;
    ptr->next = NULL;
```



Inserting node at the last into a non-empty list

Algorithm

Step 1: IF PTR = NULL Write OVERFLOW
Go to Step 1
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR - > NEXT
Step 4: SET NEW_NODE - > DATA = VAL
Step 5: SET NEW_NODE - > NEXT = NULL
Step 6: SET PTR = HEAD
Step 7: Repeat Step 8 while PTR - > NEXT != NULL
Step 8: SET PTR = PTR - > NEXT
[END OF LOOP]
Step 9: SET PTR - > NEXT = NEW_NODE
Step 10: EXIT

Function for inserting element at the end of the list

```
void lastinsert()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
     printf("\nOVERFLOW");
   }
   else
   {
     printf("\nEnter value?\n");          scanf("%d",&item);
      ptr->data = item;
      if(head == NULL)
```

```
        {
            ptr -> next = NULL;
            head = ptr;
            printf("\nNode inserted");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");

        }
    }
}
```

Insertion in singly linked list after specified Node

- In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted. This will be done by using the following statements.

```
emp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                return;
            }
        }
```

- Allocate the space for the new node and add the item to the data part of it. This will be done by using the following statements.

```
        ptr = (struct node *) malloc (sizeof(struct node));
        ptr->data = item;
```

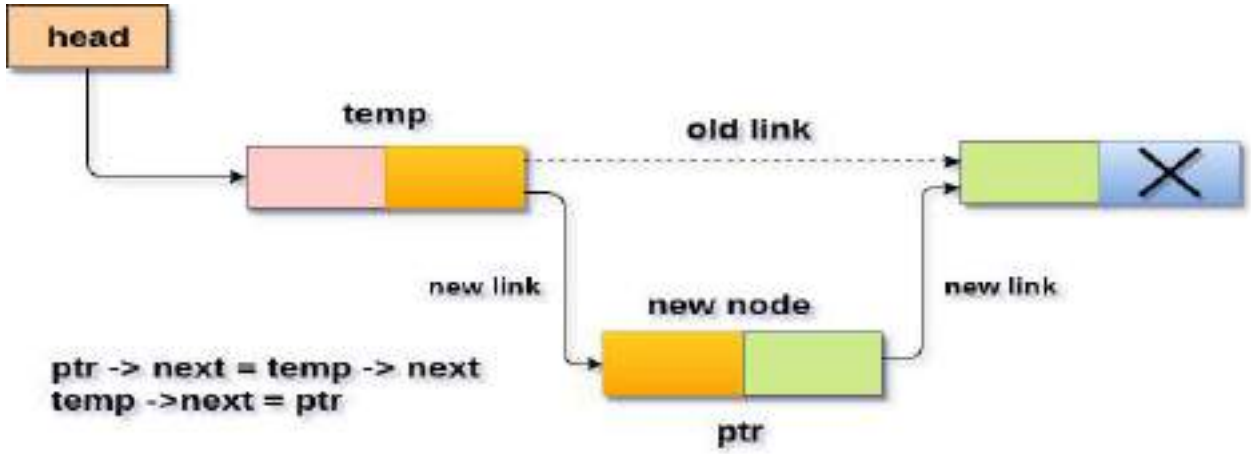
- Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be

in between temp and the next of the temp). This will be done by using the following statements.

ptr→ next = temp → next

now, we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

temp ->next = ptr;



Algorithm

- **STEP 1:** IF PTR = NULL
  WRITE OVERFLOW
  GOTO STEP 12
  END OF IF
- **STEP 2:** SET NEW_NODE = PTR
- **STEP 3:** NEW_NODE → DATA = VAL
- **STEP 4:** SET TEMP = HEAD
- **STEP 5:** SET I = 0
- **STEP 6:** REPEAT STEP 5 AND 6 UNTIL I<loc< li=""></loc<>
- **STEP 7:** TEMP = TEMP → NEXT
- **STEP 8:** IF TEMP = NULL
  WRITE "DESIRED NODE NOT PRESENT"
  GOTO STEP 12
  END OF IF
  END OF LOOP
- **STEP 9:** PTR → NEXT = TEMP → NEXT
- **STEP 10:** TEMP → NEXT = PTR
- **STEP 11:** SET PTR = NEW_NODE
- **STEP 12:** EXIT

C Function

```
void randominsert()
{
   int i,loc,item;
```

```
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=1;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
```

Deletion

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

1. Deleting at Beginning
2. Deleting at the End of the List
3. Deleting after specified node

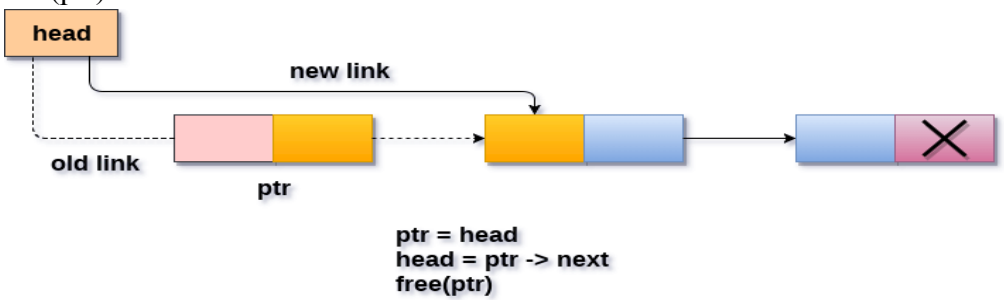
Deletion in singly linked list at beginning

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head. This will be done by using the following statements

ptr = head;
head = ptr->next;

Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.

free(ptr)



Deleting a node from the beginning

Algorithm

- **Step 1:** IF HEAD = NULL
  Write UNDERFLOW
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = HEAD
- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** EXIT

C function

```
void begdelete()
   {
      struct node *ptr;
      if(head == NULL)
      {
         printf("\nList is empty");
      }
      else
      {
         ptr = head;
         head = ptr->next;
         free(ptr);
         printf("\n Node deleted from the begining ...");
      }
```

}

Deletion in singly linked list at the end

Here are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

the condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null. This will be done by using the following statements.

```
ptr = head
head = NULL
free(ptr)
```

In the second scenario,

The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

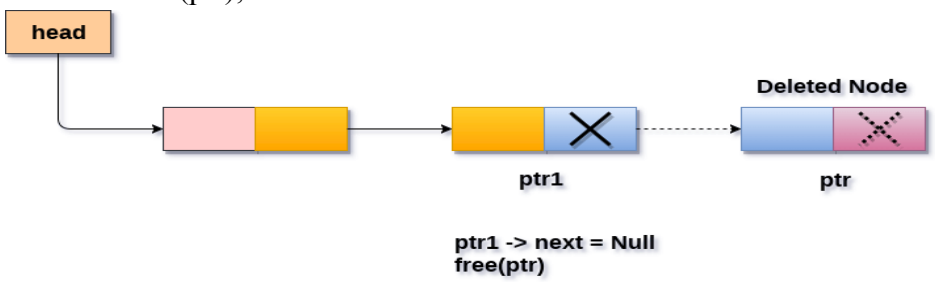
For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers ptr and ptr1 will be used where ptr will point to the last node and ptr1 will point to the second last node of the list.

this all will be done by using the following statements.

```
ptr = head;
    while(ptr->next != NULL)
    {
       ptr1 = ptr;
       ptr = ptr ->next;
    }
```

Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by ptr will become free. It will be done by using the following statements.

```
ptr1->next = NULL;
   free(ptr);
```



Deleting a node from the last

Algorithm

- **Step 1:** IF HEAD = NULL
  Write UNDERFLOW
  Go to Step 8
  [END OF IF]
- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Steps 4 and 5 while PTR -> NEXT!= NULL
- **Step 4:** SET PREPTR = PTR
- **Step 5:** SET PTR = PTR -> NEXT
  [END OF LOOP]
- **Step 6:** SET PREPTR -> NEXT = NULL
- **Step 7:** FREE PTR
- **Step 8:** EXIT

C Function

```
void end_delete()
   {
      struct node *ptr,*ptr1;
     if(head == NULL)
      {
         printf("\nlist is empty");
      }
      else if(head -> next == NULL)
      {
         head = NULL;
         free(head);
         printf("\nOnly node of the list deleted ...");
      }
      else
      {
         ptr = head;
         while(ptr->next != NULL)
            {
               ptr1 = ptr;
               ptr = ptr ->next;
            }
           ptr1->next = NULL;
           free(ptr);
           printf("\n Deleted Node from the last ...");
```

```
        }
    }
}
```

Deletion in singly linked list after the specified node

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.

Use the following statements to do so.

```
ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nThere are less than %d elements in the list..",loc);
            return;
        }
    }
```

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

This will be done by using the following statements.

Deletion a node from specified position

Algorithm

- **STEP 1:** IF HEAD = NULL

  WRITE UNDERFLOW
  GOTO STEP 10
  END OF IF

- **STEP 2:** SET TEMP = HEAD
- **STEP 3:** SET I = 0
- **STEP 4:** REPEAT STEP 5 TO 8 UNTIL I<loc< li=""></loc<>
- **STEP 5:** TEMP1 = TEMP
- **STEP 6:** TEMP = TEMP → NEXT
- **STEP 7:** IF TEMP = NULL

  WRITE "DESIRED NODE NOT PRESENT"
  GOTO STEP 12
  END OF IF
- **STEP 8:** I = I+1

  END OF LOOP
- **STEP 9:** TEMP1 → NEXT = TEMP → NEXT
- **STEP 10:** FREE TEMP
- **STEP 11:** EXIT

Searching in singly linked list

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

- **Step 1:** SET PTR = HEAD
- **Step 2:** Set I = 0
- **STEP 3:** IF PTR = NULL

  WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF
- **STEP 4:** REPEAT STEP 5 TO 7 UNTIL PTR != NULL
- **STEP 5:** if ptr → data = item

  write i+1
  End of IF
- **STEP 6:** I = I + 1
- **STEP 7:** PTR = PTR → NEXT

  [END OF LOOP]

- **STEP 8:** EXIT

C Function

```
void search()
{
   struct node *ptr;
   int item,i=0,flag;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      while (ptr!=NULL)
      {
         if(ptr->data == item)
         {
            printf("item found at location %d ",i+1);
            flag=0;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      if(flag==1)
      {
         printf("Item not found\n");
      }
   }
```

Traversing in singly linked list

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
ptr = head;
```

```
while (ptr!=NULL)
   {
      ptr = ptr -> next;
   }
```

Algorithm

- **STEP 1:** SET PTR = HEAD
- **STEP 2:** IF PTR = NULL
  WRITE "EMPTY LIST"
  GOTO STEP 7
  END OF IF
- **STEP 4:** REPEAT STEP 5 AND 6 UNTIL PTR != NULL
- **STEP 5:** PRINT PTR→ DATA
- **STEP 6:** PTR = PTR → NEXT
  [END OF LOOP]
- **STEP 7:** EXIT

SINGLY LINKED LIST ADVANTAGE

1) Insertions and Deletions can be done easily.
2) It does not need movement of elements for insertion and deletion.
3) It space is not wasted as we can get space according to our requirements.
4) Its size is not fixed.
5) It can be extended or reduced according to requirements.
6) Elements may or may not be stored in consecutive memory available
7) It is less expensive.

DISADVANTAGE

1) It requires more space as pointers are also stored with information.
2) Different amount of time is required to access each element.
3) If we have to go to a particular element then we have to go through all those elements that come before that element.
4) we can not traverse it from last & only from the beginning.
5) It is not easy to sort the elements stored in the linear linked list.

Applications of Linked Lists

Graphs, queues, and stacks can be implemented by using Linked List.

DOUBLE LINKED LIST

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.

Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
}
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

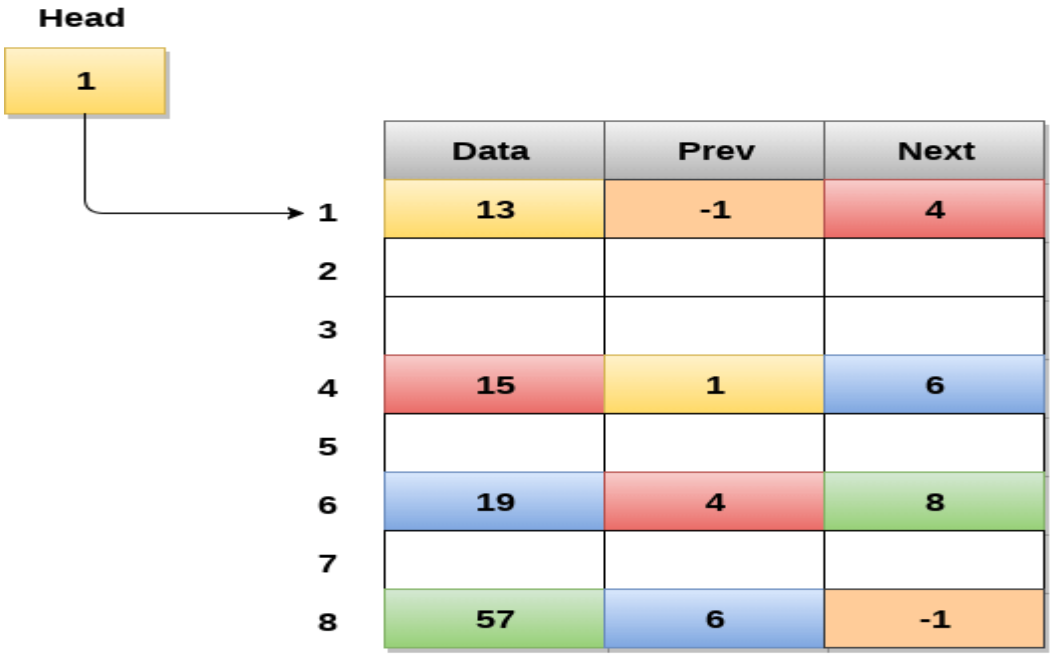
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly

linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.
We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

The following operations are performed on double linked list

1) Insertion

- Insertion at beginning
- Insertion at End
- Insertion at specified position

1) Deletion
    - Deletion from the Beginning
    - Deletion from the End
    - Deletion of the node having specified data
2) Searching
3) Traversing

Node Creation

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
};
struct node *head;
```

INSERTION

Insertion in doubly linked list at beginning

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory. This will be done by using the following statement.

  ```
  ptr = (struct node *)malloc(sizeof(struct node));
  ```

- Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

  ```
  ptr->next = NULL;
  ptr->prev=NULL;
  ptr->data=item;
  head=ptr;
  ```

- In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer

of the node. The prev pointer of the existing head will point to the new node being inserted.

- This will be done by using the following statements.

  ptr->next = head;

  head→prev=ptr;

  Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

  ptr→prev =NULL

  head = ptr

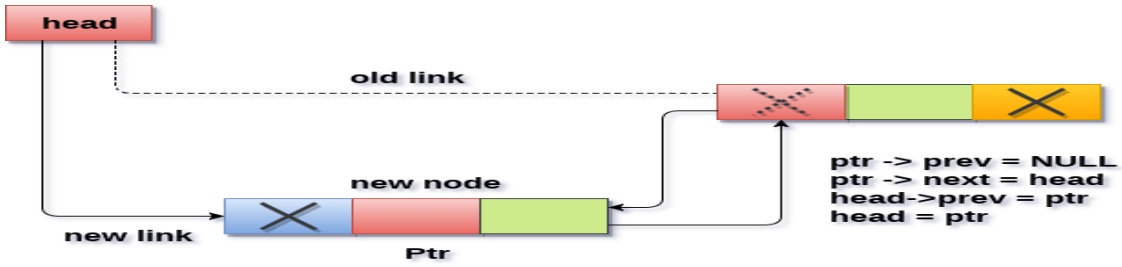
Algorithm :

- **Step 1:** IF ptr = NULL

  Write OVERFLOW

  Go to Step 9

  [END OF IF]
- **Step 2:** SET NEW_NODE = ptr
- **Step 3:** SET ptr = ptr -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> PREV = NULL
- **Step 6:** SET NEW_NODE -> NEXT = START
- **Step 7:** SET head -> PREV = NEW_NODE
- **Step 8:** SET head = NEW_NODE
- **Step 9:** EXIT



Insertion into doubly linked list at beginning

C Function

```
void insertbeginning( )
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  int item;
  printf("enter the value");
```

```
  scanf("%d",&item);
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
        if(head==NULL)
        {
                ptr->next = NULL;
                ptr->prev=NULL;
                ptr->data=item;
                head=ptr;
        }
        else
        {
                ptr->data=item;
                ptr->prev=NULL;
                ptr->next = head;
                head->prev=ptr;
                head=ptr;
         }
}
```

Insertion in doubly linked list at the end

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

- Allocate the memory for the new node. Make the pointer **ptr** point to the new node being inserted.

  ```
  ptr = (struct node *) malloc(sizeof(struct node));
  ```

- Check whether the list is empty or not. The list is empty if the condition **head == NULL** holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

  ```
  ptr->next = NULL;
   ptr->prev=NULL;
   ptr->data=item;
    head=ptr;
  ```

- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in

order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

```
Temp = head;
while (temp != NULL)
{
        temp = temp → next;
 }
```

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.

```
temp→next =ptr;
```

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

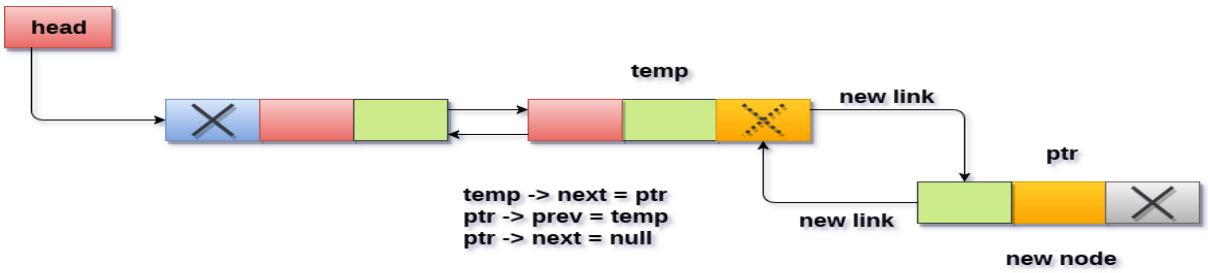
```
ptr → prev = temp;
```

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

```
ptr → next = NULL
```

Algorithm

- **Step 1:** IF PTR = NULL
  Write OVERFLOW
  Go to Step 11
  [END OF IF]
- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> NEXT = NULL
- **Step 6:** SET TEMP = START
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != NULL
- **Step 8:** SET TEMP = TEMP -> NEXT
  [END OF LOOP]
- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10C:** SET NEW_NODE -> PREV = TEMP
- **Step 11:** EXIT



Insertion into doubly linked list at the end

C Program

```
void insertlast()
{
  struct node *ptr = (struct node *) malloc(sizeof(struct node));
  int item;
  printf(“enter the value”);
  scanf(“%d”,&item);
  struct node *temp;
  if(ptr == NULL)
  {
    printf("\nOVERFLOW");
  }
  else
  {
     ptr->data=item;
    if(head == NULL)
    {
       ptr->next = NULL;
       ptr->prev = NULL;
       head = ptr;
    }
    else
    {
      temp = head;
      while(temp->next!=NULL)
      {
         temp = temp->next;
      }
      temp->next = ptr;
```

```
        ptr ->prev=temp;
        ptr->next = NULL;
      }
printf("\nNode Inserted\n");
  }
}
```

Insertion in doubly linked list after Specified node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required. Use the following steps for this purpose.

- Allocate the memory for the new node. Use the following statements for this.

```
ptr = (struct node *)malloc(sizeof(struct node));
```

- Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.

```
temp=head;
for(i=0;i<loc;i++)
{
  temp = temp->next;
  if(temp == NULL) // the temp will be //null if the list doesn't last long //up to mentio
    ned location
  {
    return;
  }
}
```

- The temp would point to the specified node at the end of the **for** loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

```
ptr → next = temp → next;
```

make the **prev** of the new node ptr point to temp.

```
ptr → prev = temp;
```

make the **next** pointer of temp point to the new node ptr.

```
temp → next = ptr;
```

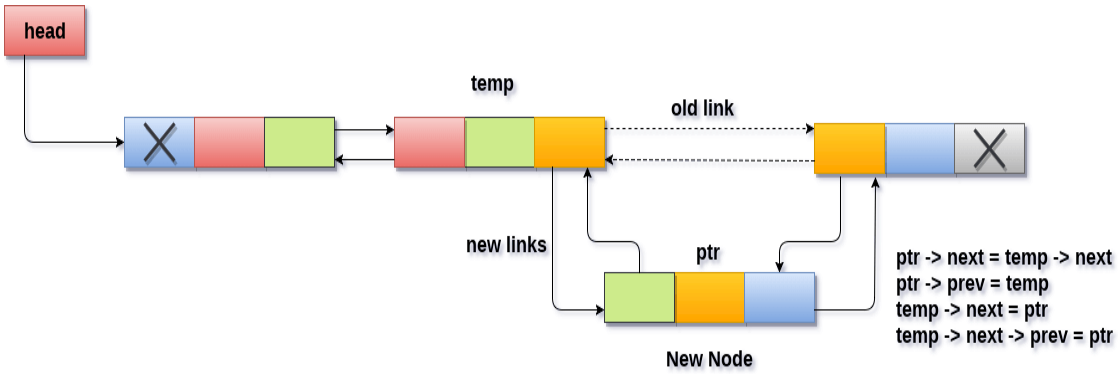
make the **previous** pointer of the next node of temp point to the new node.

```
temp → next → prev = ptr;
```

Algorithm

- **Step 1:** IF PTR = NULL
  Write OVERFLOW
  Go to Step 15
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = START
- **Step 6:** SET I = 0
- **Step 7:** REPEAT 8 to 10 until I<="" li="">
- **Step 8:** SET TEMP = TEMP -> NEXT
- **STEP 9:** IF TEMP = NULL
- **STEP 10:** WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
  GOTO STEP 15
  [END OF IF]
  [END OF LOOP]
- **Step 11:** SET NEW_NODE -> NEXT = TEMP -> NEXT
- **Step 12:** SET NEW_NODE -> PREV = TEMP
- **Step 13 :** SET TEMP -> NEXT = NEW_NODE
- **Step 14:** SET TEMP -> NEXT -> PREV = NEW_NODE
- **Step 15:** EXIT



Insertion into doubly linked list after specified node

C Function

```
void insert_specified(int item)
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  struct node *temp;
  int i, loc;
  if(ptr == NULL)
  {
     printf("\n OVERFLOW");
```

```
    }
    else
    {
        printf("\nEnter the location\n");
        scanf("%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
        ptr->data = item;
        ptr->next = temp->next;
        ptr -> prev = temp;
        temp->next = ptr;
        temp->next->prev=ptr;
        printf("Node Inserted\n");
    }
}
```

DELETION OPERATION

Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

Ptr = head;

head = head → next;

now make the prev of this new head node point to NULL. This will be done by using the following statements.

head → prev = NULL

Now free the pointer ptr by using the **free** function.

free(ptr)

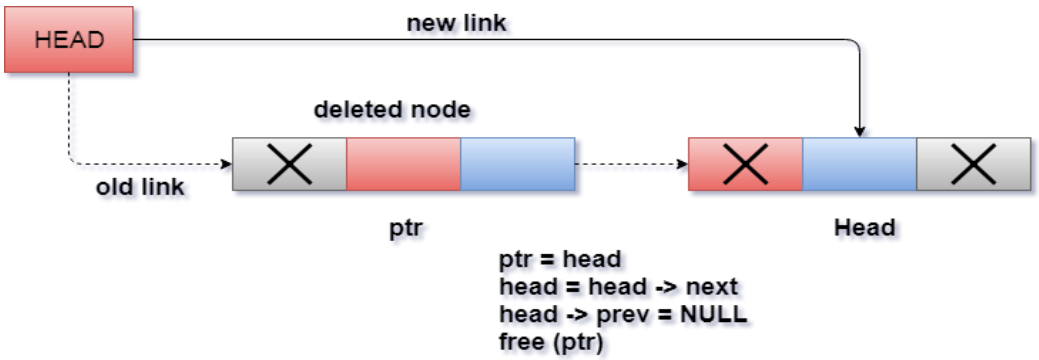
Algorithm

- **STEP 1:** IF HEAD = NULL

  WRITE UNDERFLOW

  GOTO STEP 6

- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

C FUNCTION

```
void beginning_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW\n");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```

Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
if(ptr->next != NULL)
{
        ptr = ptr -> next;
}
```

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

  ptr → prev → next = NULL

  free the pointer as this the node which is to be deleted.

```
free(ptr)
```

ALGORITHM

- **Step 1:** IF HEAD = NULL

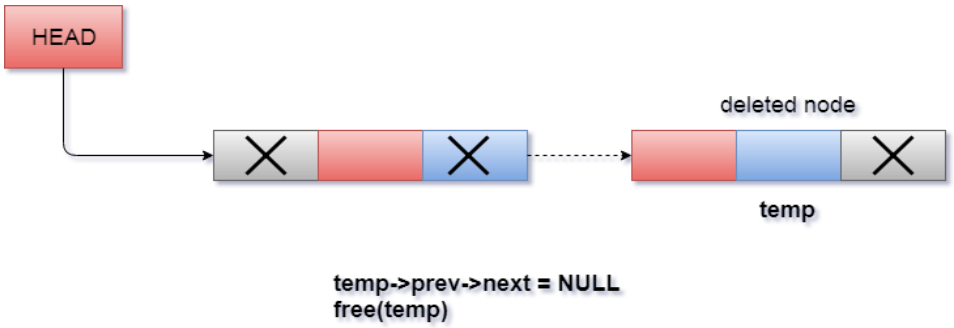
  Write UNDERFLOW
  Go to Step 7
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL
- **Step 4:** SET TEMP = TEMP->NEXT

  [END OF LOOP]

- **Step 5:** SET TEMP ->PREV-> NEXT = NULL

- **Step 6:** FREE TEMP
- **Step 7:** EXIT



Deletion in doubly linked list at the end

C PROGRAM

```
void last_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW\n");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      if(ptr->next != NULL)
      {
         ptr = ptr -> next;
      }
      ptr -> prev -> next = NULL;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```

Deletion in doubly linked list after the specified node

In order to delete the node after the specified data, we need to perform the following steps.

- Copy the head pointer into a temporary pointer temp.

```
temp = head
```

- Traverse the list until we find the desired data value.

```
while(temp -> data != val)
temp = temp -> next;
```

- Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
{
        return;
}
```

- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
{
        temp ->next = NULL;
}
```

- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

```
ptr = temp -> next;
temp -> next = ptr -> next;
ptr -> next -> prev = temp;
free(ptr);
```

Algorithm

- **Step 1:** IF HEAD = NULL

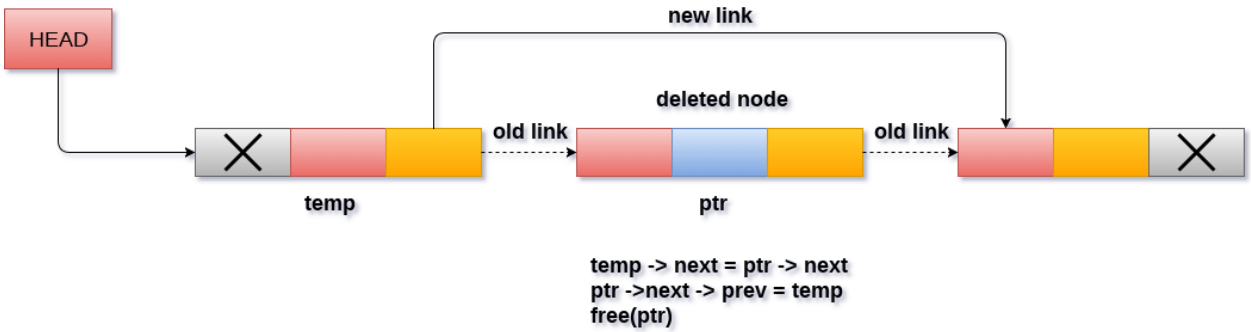
Write UNDERFLOW
Go to Step 9
[END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** Repeat Step 4 while TEMP -> DATA != ITEM
- **Step 4:** SET TEMP = TEMP -> NEXT

[END OF LOOP]

- **Step 5:** SET PTR = TEMP -> NEXT
- **Step 6:** SET TEMP -> NEXT = PTR -> NEXT

- **Step 7:** SET PTR -> NEXT -> PREV = TEMP
- **Step 8:** FREE PTR
- **Step 9:** EXIT



Deletion of a specified node in doubly linked list

C FUNCTION

```
void delete_specified( )
{
   struct node *ptr, *temp;
   int val;
   printf("Enter the value");
   scanf("%d",&val);
   temp = head;
   while(temp -> data != val)
   temp = temp -> next;
   if(temp -> next == NULL)
   {
      printf("\nCan't delete\n");
   }
   else if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = temp -> next;
      temp -> next = ptr -> next;
      ptr -> next -> prev = temp;
      free(ptr);
      printf("\nNode Deleted\n");
```

```
    }
}
```

Searching for a specific node in Doubly Linked List

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.
  ptr = head
- declare a local variable I and assign it to 0.
  i=0
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm

- **Step 1:** IF HEAD == NULL
  WRITE "UNDERFLOW"
  GOTO STEP 8
  [END OF IF]
- **Step 2:** Set PTR = HEAD
- **Step 3:** Set i = 0
- **Step 4:** Repeat step 5 to 7 while PTR != NULL
- **Step 5:** IF PTR → data = item
  return i
  [END OF IF]
- **Step 6:** i = i + 1
- **Step 7:** PTR = PTR → next
- **Step 8:** Exit

C FUNCTION

```
void search()
{
   struct node *ptr;
   int item,i=0,flag;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
```

```
}
else
{
   printf("\nEnter item which you want to search?\n");
   scanf("%d",&item);
   while (ptr!=NULL)
   {
      if(ptr->data == item)
      {
         printf("\nitem found at location %d ",i+1);
         flag=0;
         break;
      }
      else
      {
         flag=1;
      }
      i++;
      ptr = ptr -> next;
   }
   if(flag==1)
   {
      printf("\nItem not found\n");
   }
}
```

Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

```
Ptr = head
```

then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

```
while(ptr != NULL)
{
        printf("%d\n",ptr->data);
        ptr=ptr->next;
}
```

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

- **Step 1:** IF HEAD == NULL

  WRITE "UNDERFLOW"
  GOTO STEP 6
  [END OF IF]

- **Step 2:** Set PTR = HEAD
- **Step 3:** Repeat step 4 and 5 while PTR != NULL
- **Step 4:** Write PTR → data
- **Step 5:** PTR = PTR → next
- **Step 6:** Exit

C Function

```
int traverse()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      ptr = head;
      while(ptr != NULL)
      {
         printf("%d\n",ptr->data);
         ptr=ptr->next;
      }
   }
 }
```

Differences between Singly linked list and Doubly linked list

Singly linked list (SLL)	**Doubly linked list (DLL)**
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.

Singly linked list (SLL)	Doubly linked list (DLL)
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.

3.Write a program that uses functions to perform the following operations on circular linked list:

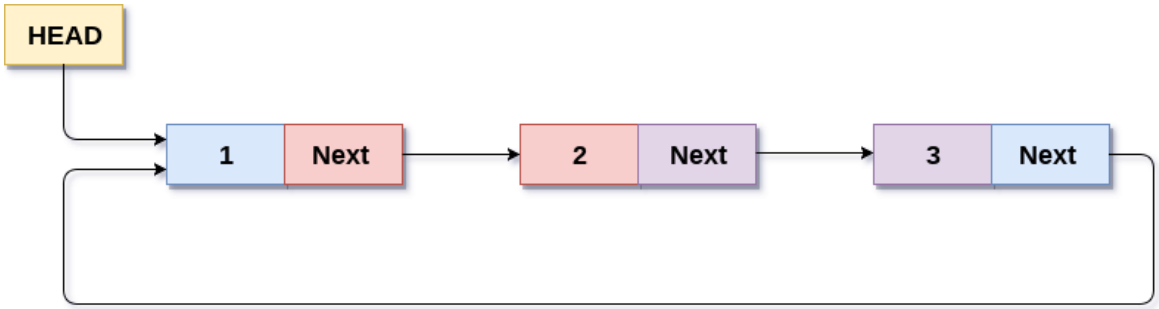
i) Creation ii) Insertion iii) Deletion iv) Traversal

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Write a program that uses functions to perform the following operations on circular linked list:

i) Creation ii) Insertion iii) Deletion iv) Traversal

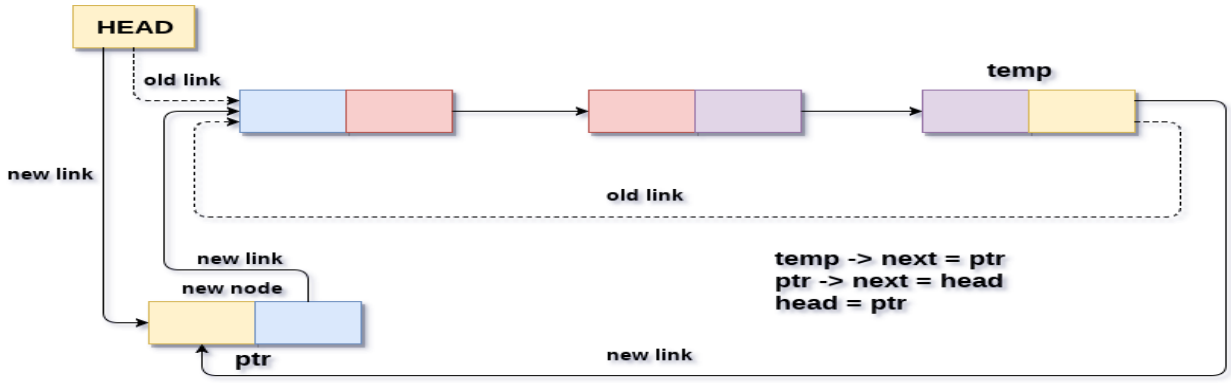
i)Creation

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("1.Append List\n2.Exit\n3.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }

   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      ptr -> data = item;
```

```
        if(head == NULL)
        {
           head = ptr;
           ptr -> next = head;
        }
        else
        {
           temp = head;
           while(temp->next != head)
              temp = temp->next;
           ptr->next = head;
           temp -> next = ptr;
           head = ptr;
        }
    printf("\nNode Inserted\n");
    }
}
```

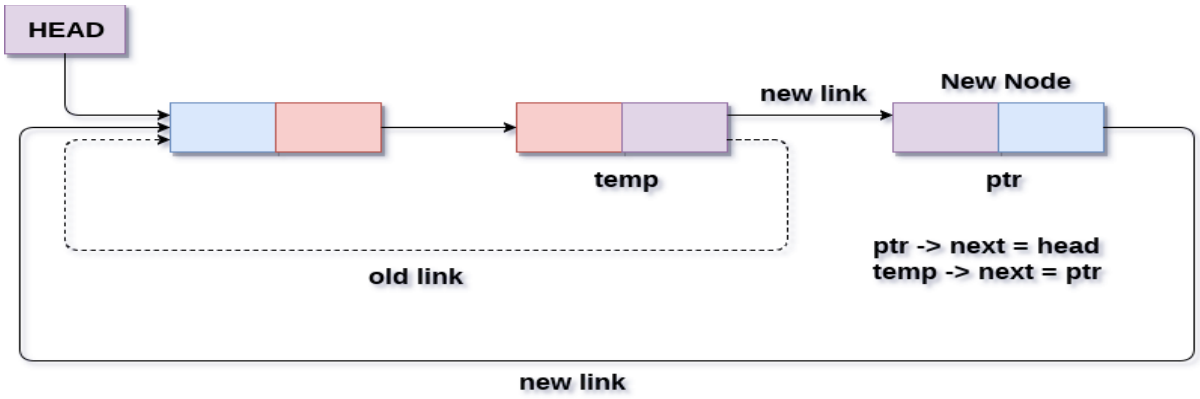
ii)Insertion

Insertion into circular singly linked list at beginning



Insertion into circular singly linked list at beginning

Insertion into circular singly linked list at the end



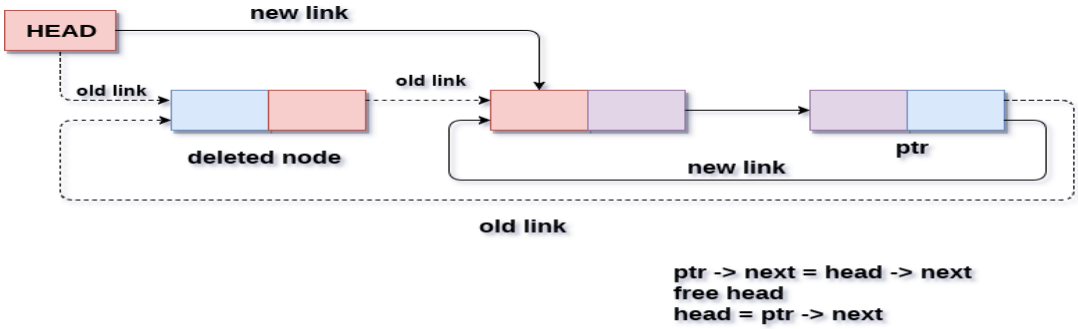
Insertion into circular singly linked list at end

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void beginsert ();
void lastinsert ();
void display();
void main ()
{
   int choice =0;
   while(choice != 4)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.display\n4.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         beginsert();
         break;
         case 2:
         lastinsert();
         break;
         case 3:
         display();
         case 4:
```

```
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data?");
```

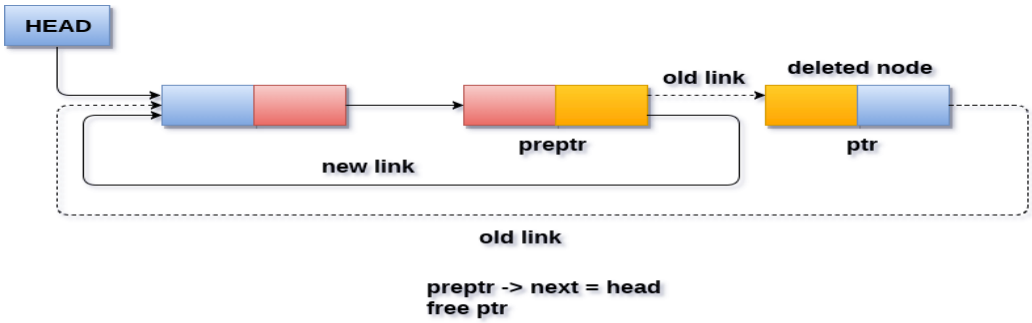
```
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
           head = ptr;
           ptr -> next = head;
        }
        else
        {
           temp = head;
           while(temp -> next != head)
           {
              temp = temp -> next;
           }
           temp -> next = ptr;
           ptr -> next = head;
        }

        printf("\nnode inserted\n");
     }

}

void display()
{
   struct node *ptr;
   ptr=head;
   if(head == NULL)
   {
      printf("\nnothing to print");
   }
   else
   {
      printf("\n printing values ... \n");

      while(ptr -> next != head)
      {

         printf("%d\n", ptr -> data);
         ptr = ptr -> next;
      }
      printf("%d\n", ptr -> data);
   }
   }
```

Deletion in circular singly linked list at beginning



Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end



Deletion in circular singly linked list at end

iii) Deletion

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void create();
void begin_delete();
void last_delete();
void display();
void main ()
{
   int choice =0;
   while(choice != 5)
```

```
    {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.create\n2.Delete from Beginning\n3.Delete from last\n4.Show\n5.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {

            case 1:
            create();
            break;
            case 2:
            begin_delete();
            break;
            case 3:
            last_delete();
            break;
            case 4:
            display();
            break;
            case 5:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}

void create()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data?");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
```

```
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }

        printf("\nnode inserted\n");
    }

}

void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nUNDERFLOW");
    }
    else if(head->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }

    else
    {   ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
        printf("\nnode deleted\n");

    }
}
void last_delete()
{
    struct node *ptr, *preptr;
    if(head==NULL)
    {
        printf("\nUNDERFLOW");
    }
    else if (head ->next == head)
    {
        head = NULL;
        free(head);
```

```
        printf("\nnode deleted\n");

    }
    else
    {
        ptr = head;
        while(ptr ->next != head)
        {
            preptr=ptr;
            ptr = ptr->next;
        }
        preptr->next = ptr -> next;
        free(ptr);
        printf("\nnode deleted\n");

    }
}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }

}
```

iv) Traversal

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void traverse();
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         traverse();
         break;
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }

   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      ptr -> data = item;
```

```
        if(head == NULL)
        {
           head = ptr;
           ptr -> next = head;
        }
        else
        {
           temp = head;
           while(temp->next != head)
              temp = temp->next;
           ptr->next = head;
           temp -> next = ptr;
           head = ptr;
        }
     printf("\nNode Inserted\n");
     }

}
void traverse()
{
   struct node *ptr;
   ptr=head;
   if(head == NULL)
   {
      printf("\nnothing to print");
   }
   else
   {
      printf("\n printing values ... \n");

      while(ptr -> next != head)
      {

         printf("%d\n", ptr -> data);
         ptr = ptr -> next;
      }
      printf("%d\n", ptr -> data);
   }

}
```

Dictionaries:- linear list representation, skip list representation, operations insertion, deletion and searching, hash table representation, hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.
Basic operations that can be performed on dictionary are:
1. Insertion of value in the dictionary
2. Deletion of particular value from dictionary
3. Searching of a specific value with the help of key

Linear List Representation

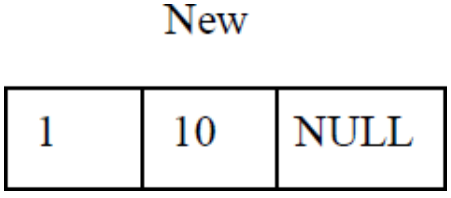
The dictionary can be represented as a linear list. The linear list is a collection of pair and value.
There are two method of representing linear list.
1. Sorted Array- An array data structure is used to implement the dictionary.
2. Sorted Chain- A linked list data structure is used to implement the dictionary

Structure of linear list for dictionary:

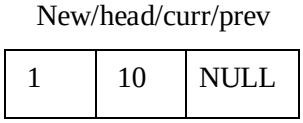
```
class dictionary
{

private:
        int k,data;
        struct node
        {
        public: int key;
        int value;
        struct node *next;
        } *head;

public:
        dictionary();
        void insert_d( );
        void delete_d( );
        void display_d( );
        void length();
};
```

Insertion of new node in the dictionary:

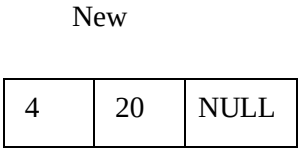
Consider that initially dictionary is empty then
head = NULL
We will create a new node with some key and value contained in it.



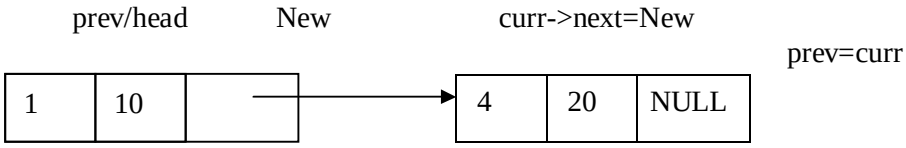
Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be 'curr' and 'prev' as well. The 'cuur' node will always point to current visiting node and 'prev' will always point to the node previous to 'curr' node. As now there is only one node in the list mark as 'curr' node as 'prev' node.



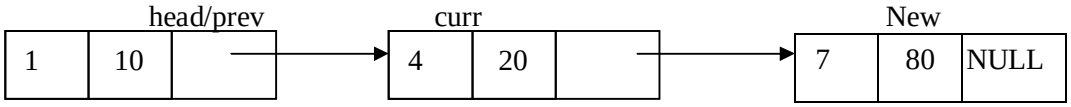
Insert a record, key=4 and value=20,



Compare the key value of 'curr' and 'New' node. If New->key > Curr->key then attach New node to 'curr' node.

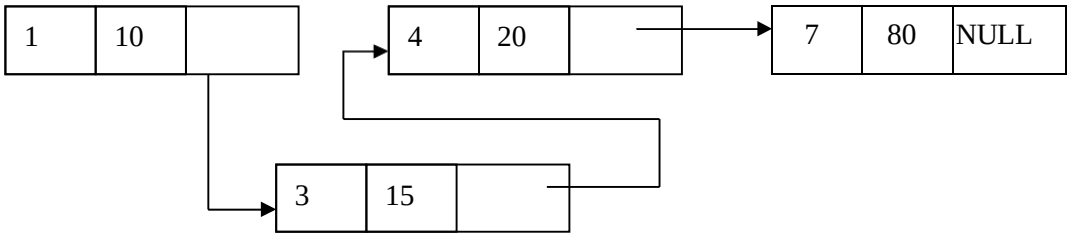


Add a new node <7,80> then



If we insert <3,15> then we have to search for it proper position by comparing key value.

(curr->key < New->key) is false. Hence else part will get executed.

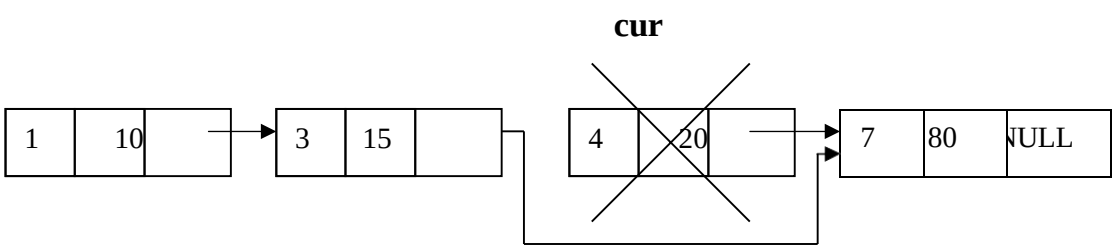



```
void dictionary::insert_d( )
{
node *p,*curr,*prev;
            cout<<"Enter an key and value to be inserted:";
            cin>>k;
           cin>>data;
```

```
        p=new node;
        p->key=k;
       p->value=data;
       p->next=NULL;
        if(head==NULL)
                head=p;
       else
       {
              curr=head;
               while((curr->key<p->key)&&(curr->next!=NULL))
              {
                     prev=curr;
                     curr=curr->next;
                }
               if(curr->next==NULL)
              {
                     if(curr->key<p->key)
                      {
                            curr->next=p;
                             prev=curr;
                     }
                     else
                     {
                           p->next=prev->next;
                            prev->next=p;
                     }
                }
                else
              {
                     p->next=prev->next;
                     prev->next=p;
              }
              cout<<"\nInserted  into dictionary Sucesfully.... \n";
       }
}
```

The delete operation:

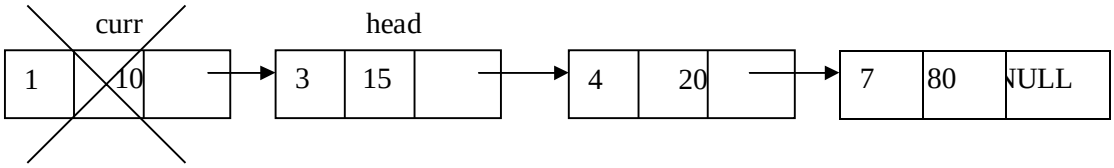
Case 1: Initially assign ‘head’ node as ‘curr’ node.Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node’s key value. We will get node which is to be deleted in variable ‘curr’. The node given by variable ‘prev’ keeps track of previous node of ‘cuu’ node. For eg, delete node with key value 4 then



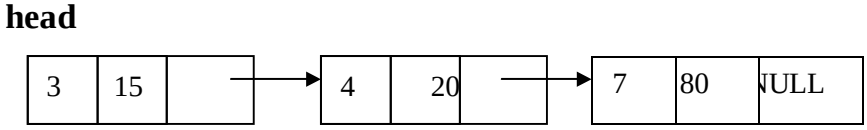
Case 2:

If the node to be deleted is head node
i.e.. if(curr==head)

Then, simply make ‘head’ node as next node and delete ‘curr’



Hence the list becomes




```
void dictionary::delete_d( )
{
node*curr,*prev;
        cout<<"Enter key value that you want to delete...";
        cin>>k;
        if(head==NULL)
            cout<<"\ndictionary is Underflow";
        else
          {   curr=head;
              while(curr!=NULL)
              {
              if(curr->key==k)
                      break;
              prev=curr;
              curr=curr->next;
              }
          }
        if(curr==NULL)
            cout<<"Node not found...";
        else
          {
             if(curr==head)
```

```
                    head=curr->next;
                else
                    prev->next=curr->next;
            delete curr;
            cout<<"Item deleted from dictionary...";
            }
  }
```

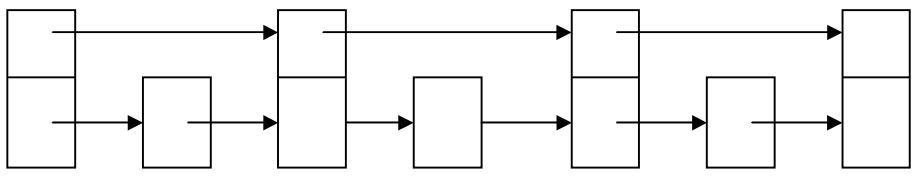
The length operation:

```
int dictionary::length()
{
struct node *curr;
int count;
        count=0;
        curr=head;
        if(curr==NULL)
        {
                cout<<"The list is empty";
                return 0;
        }
        while(curr!=NULL)
        {
                count++;
                cur=curr->next;
        }
return count;
 }
```

SKIP LIST REPRESENTATION

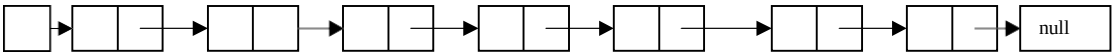
Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level.

There are two special nodes in the skip list one is head node which is the starting node of the list and tail node is the last node of the list

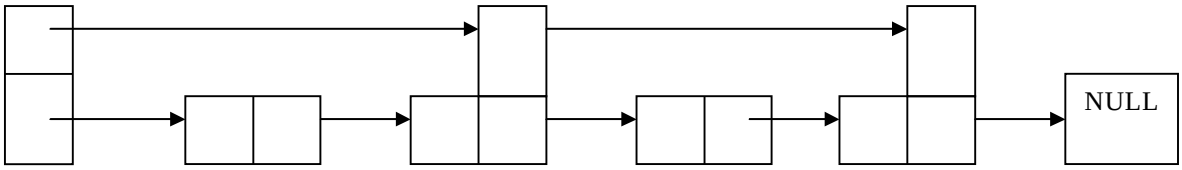


The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time.

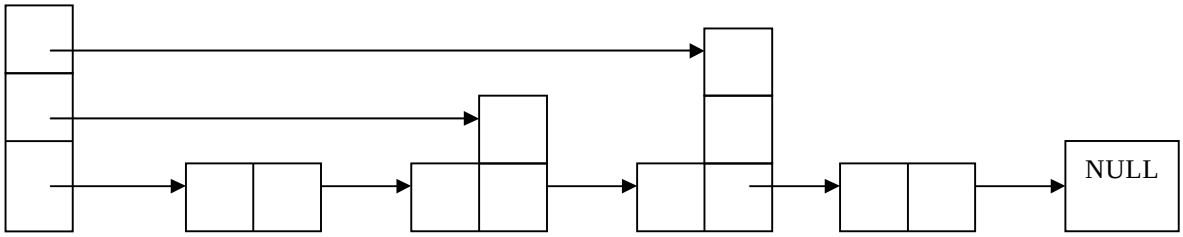
Eg:



Now to search any node from above given sorted chain we have to search the sorted chain from head node by visiting each node. But this searching time can be reduced if we add one level in every alternate node. This extra level contains the forward pointer of some node. That means in sorted chain come nodes can holds pointers to more than one node.



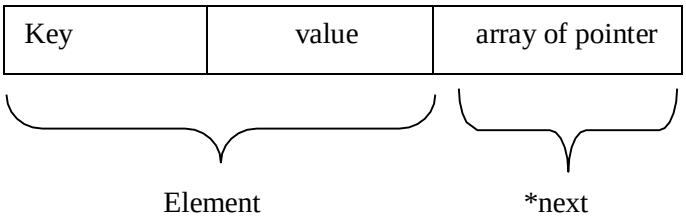
If we want to search node 40 from above chain there we will require comparatively less time. This search again can be made efficient if we add few more pointers forward references.



Node structure of skip list:

```
template <class K, class E>
struct skipnode
{
        typedef pair<const K,E> pair_type;
        pair_type element;
        skipnode<K,E> **next;
        skipnode(const pair_type &New_pair, int MAX):element(New_pair)
        {
                next=new skipnode<K,E>*[MAX];
        }
};
```

The individual node looks like this:



Searching:

The desired node is searched with the help of a key value.

```
template<class K, class E>
skipnode<K,E>* skipLst<K,E>::search(K& Key_val)
{
skipnode<K,E>* Forward_Node = header;
for(int i=level;i>=0;i--)
{
while (Forward_Node->next[i]->element.key < key_val)
Forward_Node = Forward_Node->next[i];
last[i] = Forward_Node;
}
return Forward_Node->next[0];
}
```

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the key_val. If the node key is less than the key_val, the search continues moving forward at the same level. If o the other hand, the node key is equal to or greater than the key_val, the search drops one level and continues forward. This process continues until the desired key_val has been found if it is present in the skip list. If it is not, the search will either continue at the end of the list or until the first key with a value greater than the search key is found.

Insertion:

There are two tasks that should be done before insertion operation:

1. Before insertion of any node the place for this new node in the skip list is searched. Hence before any insertion to take place the search routine executes. The last[] array in the search routine is used to keep track of the references to the nodes where the search, drops down one level.
2. The level for the new node is retrieved by the routine randomelevel()

```
template<class K,class E>
void skipLst<K,E>::insert(pair<K,E>& New_pair)
{
if(New_pair.key >= tailkey)
{
cout<<"Key is too large";
}

skipNode<K,E>* temp = search(New_pair.key);
if(temp->element.key == New_pair.key)
```

```
{
temp->element.value=New_pair.value;
return;
}

if*New_Level > levels)
{
New_Level = ++levels;
last[New_Level] = header;
}

skipNode<K,E> *newNode = new skipNode<K,E>(New_pair, New_Level+1);

for(int i=0;i<=New_Level;i++)
{
newNode->next[i] = last[i]->next[i];
last[i]->next[i] = newNode;
}
len++;
return;
}
```

Determining the level of each node:

```
template <class K, class E>
int skipLst<K,E>::randomlevel()
{
int lvl=0;
while(rand() <= Lvl_No)
lvl=lvl+1;
if(lvl<=MaxLvl)
return lvl;
else
return MaxLvl;
}
```

Deletion:

First of all, the deletion makes use of search algorithm and searches the node that is to be deleted. If the key to be deleted is found, the node containing the key is removed.

```
template<class K, class E>
void skipLst<K,E>::delet(K& Key_val)
{
if(key_val>=tailKey)
return;
skipNode<K,E>* temp = search(Key_val);
if(temp->elemnt.key != Key_val)
return;

for(int i=0;i<=levels;i++)
```

```
{
if(last[i]->next[i] == temp)
last[i]=>next[i] = temp->next[i];
}

while(level>0 && header->next[level] == tail)
levels--;
delete temp;
len--;
}
```

HASH TABLE REPRESENTATION

- Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.
- Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table.
- The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

HASH FUNCTION

- Hash function is a function which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table.
- The integer returned by the hash function is called hash key.

For example: Consider that we want place some employee records in the hash table The record of employee is placed with the help of key: employee ID. The employee ID is a 7 digit number for placing the record in the hash table. To place the record 7 digit number is converted into 3 digits by taking only last three digits of the key.

If the key is 496700 it can be stored at 0^{th} position. The second key 8421002, the record of those key is placed at 2^{nd} position in the array.
Hence the hash function will be- H(key) = key%1000
Where key%1000 is a hash function and key obtained by hash function is called hash key.

- **Bucket and Home bucket:** The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

The function H(key) is home bucket for the dictionary with pair whose value is key.

TYPES OF HASH FUNCTION
There are various types of hash functions that are used to place the record in the hash table-

1. Division Method: The hash function depends upon the remainder of division.

Typically the divisor is table length.
For eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then

h(key) = record % table size

54%10=4
72%10=2
89%10=9
37%10=7

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

2. Mid Square:

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number.
Consider that if we want to place a record 3111 then

$3111^2 = 9678321$
for the hash table of size 1000
H(3111) = 783 (the middle 3 digits)

3. Multiplicative hash function:

The given record is multiplied by some constant value. The formula for computing the hash key is-

H(key) = floor(p *(fractional part of key*A)) where p is integer constant and A is constant real number.

Donald Knuth suggested to use constant A = 0.61803398987

If key 107 and p=50 then

H(key) = floor(50*(107*0.61803398987))
= floor(3306.4818458045)
= 3306
At 3306 location in the hash table the record 107 will be placed.

4. Digit Folding:

The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.
For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together

H(key) = 123+654+12
= 789
The record will be placed at location 789

5. Digit Analysis:

The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix, r. Then examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

COLLISION

the hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This function need to be designed very carefully and it should not return the same hash key address for two different records. This is an undesirable situation in hashing.

Definition: The situation in which the hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different records is called synonym.

Similarly when there is no room for a new pair in the hash table then such a situation is called overflow. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions.

For example,

Consider a hash function.

H(key) = recordkey%10 having the hash table size of 10

The record keys to be placed are

131, 44, 43, 78, 19, 36, 57 and 77
131%10=1
44%10=4
43%10=3
78%10=8
19%10=9
36%10=6
57%10=7
77%10=7

0	
1	131
2	
3	43
4	44
5	
6	36
7	57
8	78
9	19

Now if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called **collision**. From the index 7 if we look for next vacant position at subsequent indices 8.9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

COLLISION RESOLUTION TECHNIQUES

If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.

1. Chaining
2. Open addressing (linear probing)
3. Quadratic probing
4. Double hashing
5. Double hashing
6. Rehashing

CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

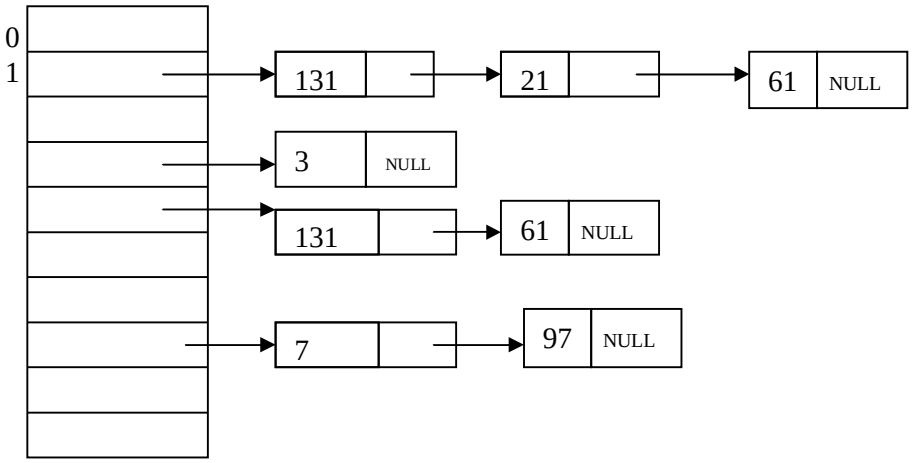
For eg;

Consider the keys to be placed in their home buckets are
131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as H(key) = key % D

Where D is the size of table. The hash table will be-

Here D = 10



A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example:

Consider that following keys are to be inserted in the hash table

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.
We will use Division hash function. That means the keys are placed using the formula

H(key) = key % tablesize
H(key) = key % 10

For instance the element 131 can be placed at

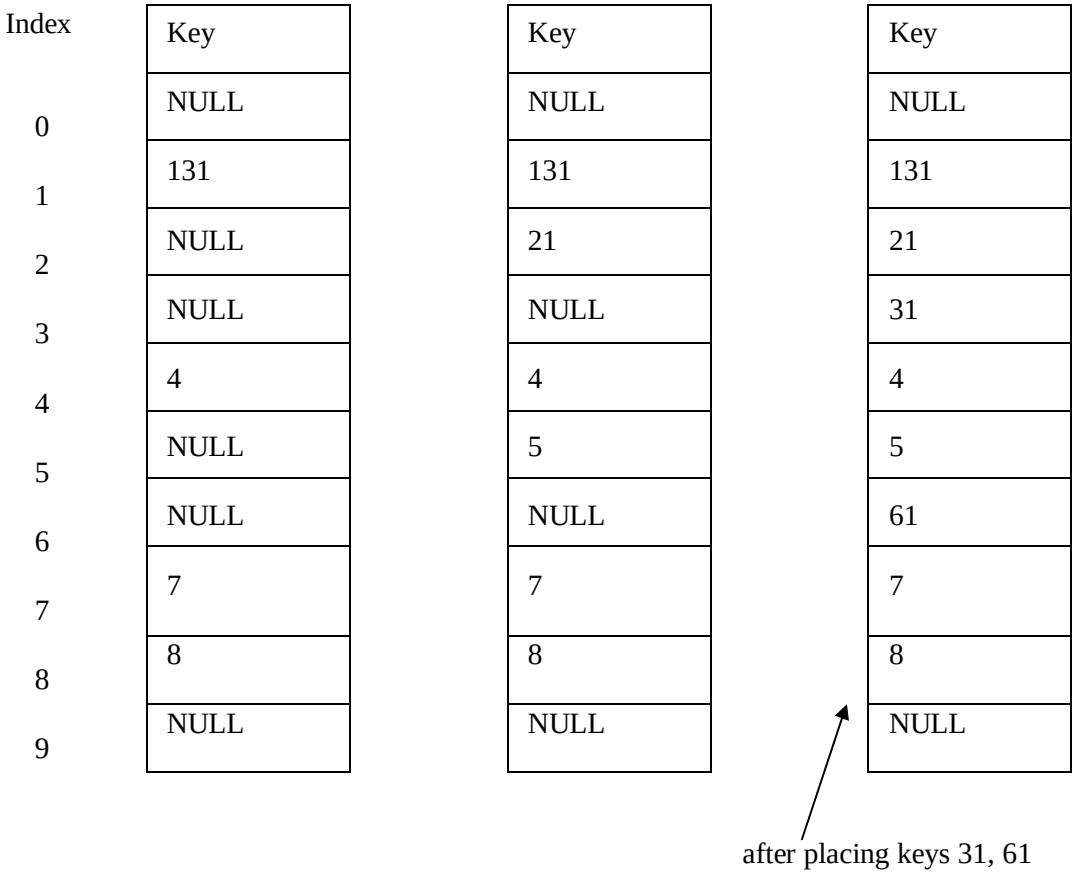
H(key) = 131 % 10
= 1

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4, 8, 7.

Now the next key to be inserted is 21. According to the hash function

H(key)=21%10
H(key) = 1

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.



The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0^{th} index.

Problem with linear probing:

One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

19%10 = 9
18%10 = 8
39%10 = 9
29%10 = 9
8%10 = 8

cluster is formed

rest of the table is empty

this cluster problem can be solved by quadratic probing.

Key
39
29
8
18
19

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(key) = (Hash(key) + i^2) \% m)$$

where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

37 % 10 = 7
90 % 10 = 0
55 % 10 = 5
22 % 10 = 2
11 % 10 = 1

Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

$H_i (key) = (Hash(key) + i^2) \% m$

Consider i = 0 then
$(17 + 0^2) \% 10 = 7$

$(17 + 1^2)$ % 10 = 8, when i =1

The bucket 8 is empty hence we will place the element at index 8.
Then comes 49 which will be placed at index 9.

49 % 10 = 9

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	

Now to place 87 we will use quadratic probing.

(87 + 0) % 10 = 7
(87 + 1) % 10 = 8… but already occupied
$(87 + 2^2)$ % 10 = 1.. already occupied
$(87 + 3^2)$ % 10 = 6

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	

DOUBLE HASHING

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert.
There are two important rules to be followed for the second function:

- it must never evaluate to zero.
- must make sure that all cells can be probed.

The formula to be used for double hashing is

$H_1(key) = key \bmod tablesize$

$H_2(key) = M - (key \bmod M)$

where M is a prime number smaller than the size of the table.

Consider the following elements to be placed in the hash table of size 10
37, 90, 45, 22, 17, 49, 55
Initially insert the elements using the formula for $H_1(key)$.
Insert 37, 90, 45, 22

$H_1(37) = 37 \% 10 = 7$
$H_1(90) = 90 \% 10 = 0$
$H_1(45) = 45 \% 10 = 5$
$H_1(22) = 22 \% 10 = 2$
$H_1(49) = 49 \% 10 = 9$

Key
90
22
45
37
49

Now if 17 to be inserted then

$H_1(17) = 17 \% 10 = 7$
$H_2(key) = M - (key \% M)$

Here M is prime number smaller than the size of the table. Prime number smaller than table size 10 is 7

Hence M = 7

$H_2(17) = 7-(17 \% 7)$
$= 7 - 3 = 4$

That means we have to insert the element 17 at 4 places from 37. In short we ha v
jumps. Therefore the 17 will be placed at index 1.

Key
90
17
22
45
37
49

Now to insert number 55

$H_1(55) = 55 \% 10 = 5 \longrightarrow$ Collision

$H_2(55) = 7-(55 \% 7)$
$= 7 - 6 = 1$

That means we have to take one jump from index 5 to place 55.
Finally the hash table will be -

Key
90
17
22
45
55
37
49

Comparison of Quadratic Probing & Double Hashing

The double hashing requires another hash function whose probing efficiency is same as some another hash function required when handling random collision.

The double hashing is more complex to implement than quadratic probing. The quadratic probing is fast technique than double hashing.

REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.

- When table is completely full
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,

H(key) = key mod tablesize

37 % 10 = 7
90 % 10= 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7 Collision solved by linear probing
49 % 10 = 9

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be

H(key) key mod 23

37 % 23 = 14
90 % 23 = 21
55 % 23 = 9
22 % 23 = 22
17 % 23 = 17
49 % 23 = 3
87 % 23 = 18

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	

Now the hash table is sufficiently large to accommodate new insertions.

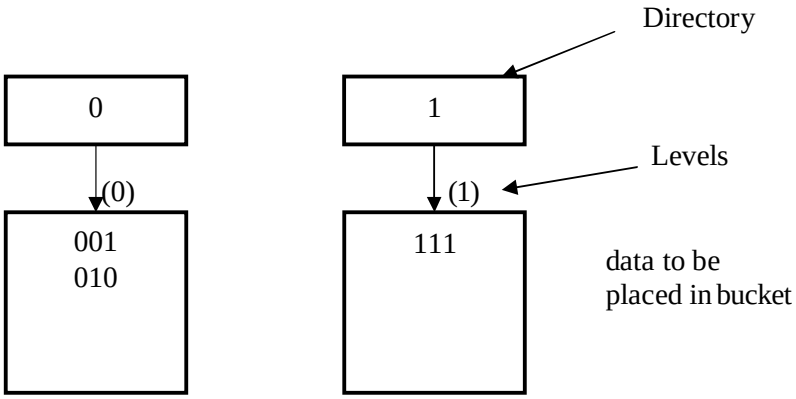
Advantages:

1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

EXTENSIBLE HASHING

- Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.
- Extensible hashing grow and shrink similar to B-trees.
- In extensible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in parenthesis.

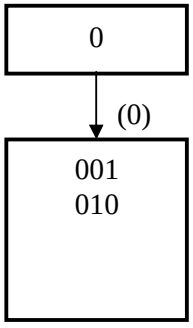
For eg:



- The bucket can hold the data of its global depth. If data in bucket is more than global depth then, split the bucket and double the directory.

Consider we have to insert 1, 4, 5, 7, 8, 10. Assume each page can hold 2 data entries (2 is the depth).

Step 1: Insert 1, 4

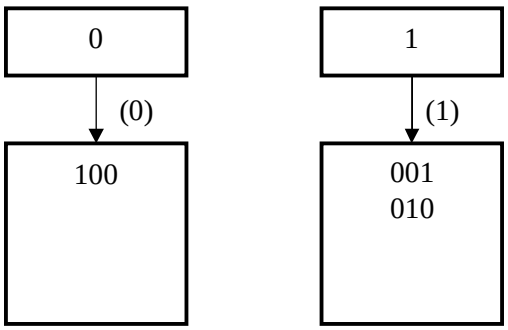


1 = 001

4 = 100

We will examine last bit of data and insert the data in bucket.

Insert 5. The bucket is full. Hence double the directory.



1 = 001

4 = 100

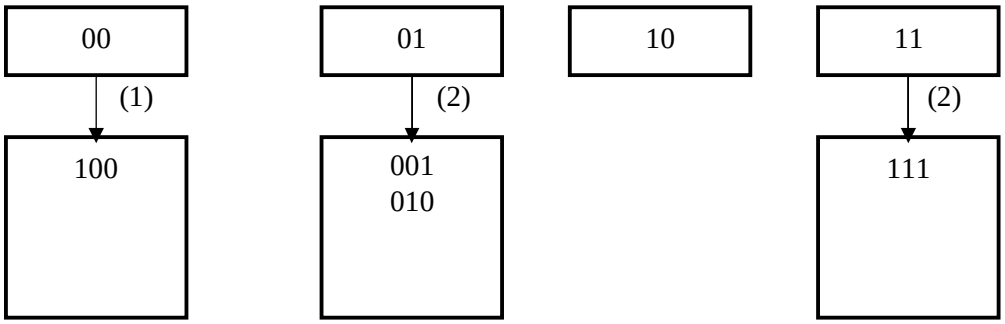
5 = 101

Based on last bit the data is inserted.

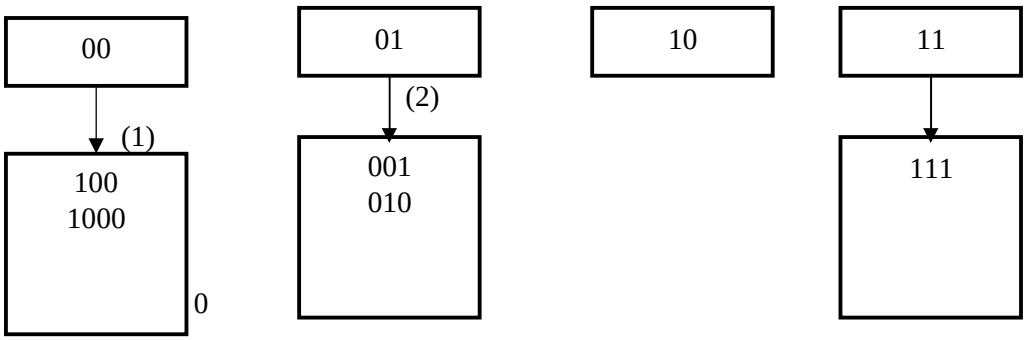
Step 2: Insert 7

7 = 111

But as depth is full we can not insert 7 here. Then double the directory and split the bucket. After insertion of 7. Now consider last two bits.



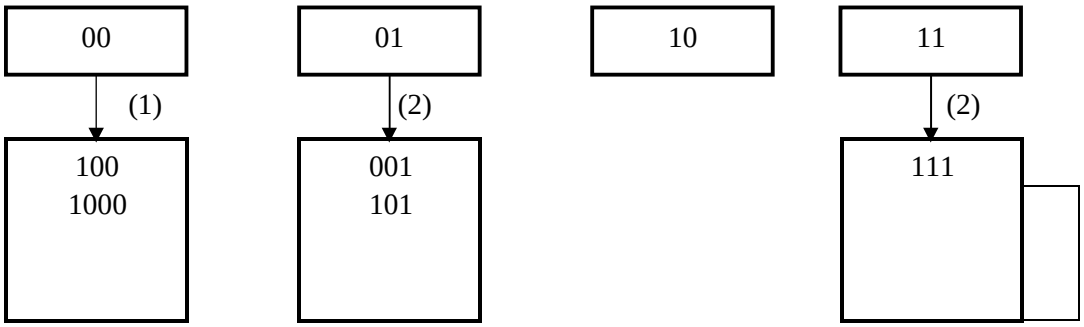
Step 3: Insert 8 i.e. 1000



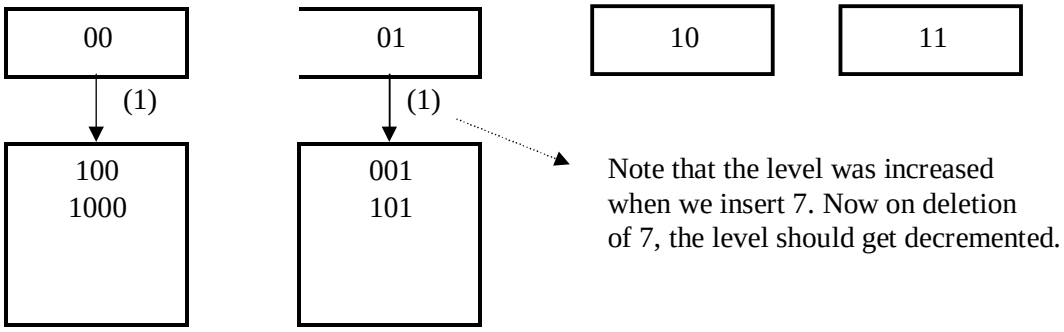
Thus the data is inserted using extensible hashing.

Deletion Operation:

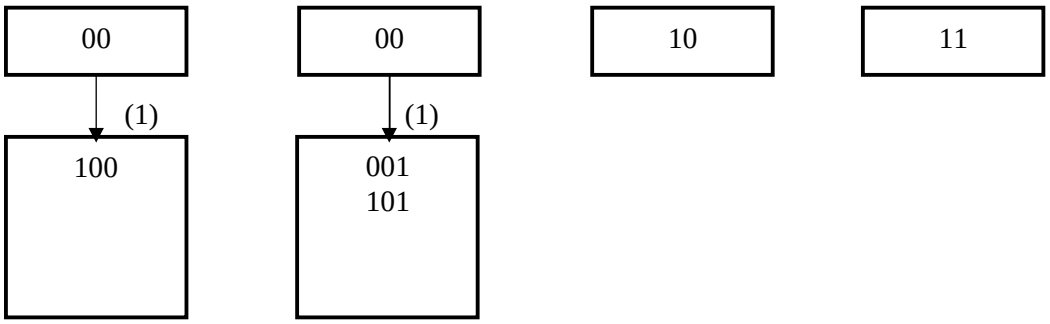
If we wan tot delete 10 then, simply make the bucket of 10 empty.



Delete 7.



Delete 8. Remove entry from directory 00.



Applications of hashing:

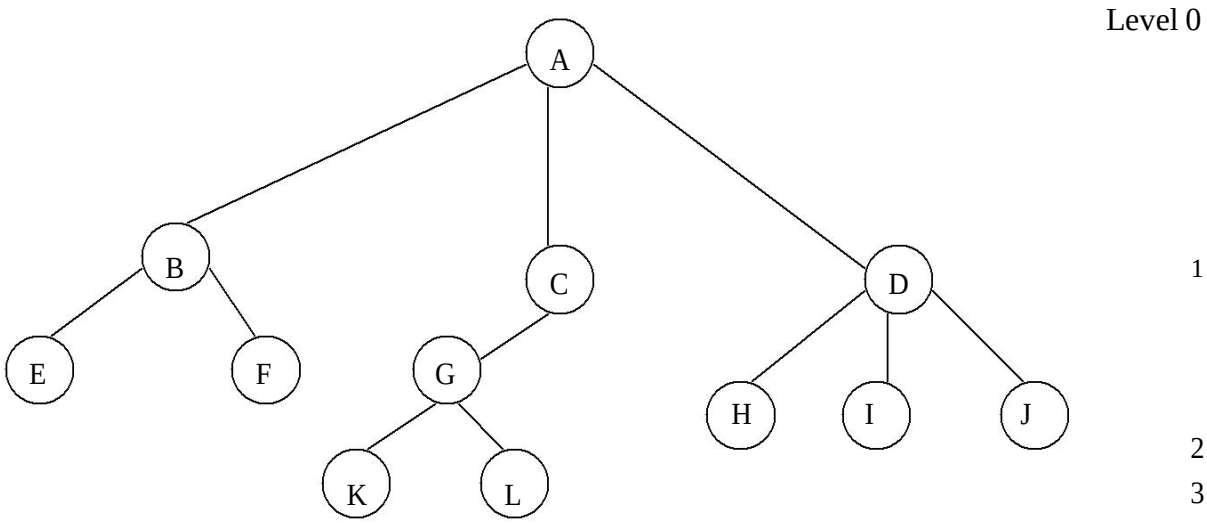
1. In compilers to keep track of declared variables.
2. For online spelling checking the hashing functions are used.
3. Hashing helps in Game playing programs to store the moves made.
4. For browser program while caching the web pages, hashing is used.
5. Construct a *message authentication code* (MAC)
6. Digital signature.
7. Time stamping
8. Key updating: key is hashed at specific intervals resulting in new key

UNIT - III

Search Trees: Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Red –Black, Splay Trees.

TREES

A Tree is a data structure in which each element is attached to one or more elements directly beneath it.



Terminology

- The connections between elements are called **branches**.
- A tree has a single root, called **root** node, which is shown at the top of the tree. i.e. root is always at the highest level 0.
- Each node has exactly one node above it, called **parent.** Eg: A is the parent of B,C and D.
- The nodes just below a node are called its **children**. ie. child nodes are one level lower than the parent node.
- A node which does not have any child called **leaf or terminal node**. Eg: E, F, K, L, H, I and M are leav
- Nodes with at least one child are called **non terminal or internal nodes**.
- The child nodes of same parent are said to be **siblings**.
- A **path** in a tree is a list of distinct nodes in which successive nodes are connected by branches in the tree.
- The **length** of a particular path is the number of branches in that path. The **degree** of a node of a tree is the number of children of that node.
- The maximum number of children a node can have is often referred to as the **order** of a tree. The **height or depth** of a tree is the length of the longest path from root to any leaf.

1. **Root:** This is the unique node in the tree to which further sub trees are attached. Eg: A
2. **Degree of the node:** The total number of sub-trees attached to the node is called the degree of the node. Eg: For node A degree is 3. For node K degree is 0
3. **Leaves:** These are the terminal nodes of the tree. The nodes with degree 0 are always the leaf nodes. Eg: E, F, K, L,H, I, J

4. **Internal nodes:** The nodes other than the root node and the leaves are called the internal nodes. Eg:B, C, D, G
5. **Parent nodes:** The node which is having further sub-trees(branches) is called the parent node ofthose sub-trees. Eg: B is the parent node of E and F.
6. **Predecessor:** While displaying the tree, if some particular node occurs previous to some other nodethen that node is called the predecessor of the other node. Eg: E is the predecessor of the node B.
7. **Successor:** The node which occurs next to some other node is a successor node. Eg: B is thesuccessor of E and F.
8. **Level of the tree**: The root node is always considered at level 0, then its adjacent children are supposed to be at level 1 and so on. Eg: A is at level 0, B,C,D are at level 1, E,F,G,H,I,J are at level 2,K,L are at level 3.
9. **Height of the tree:** The maximum level is the height of the tree. Here height of the tree is 3. Theheight if the tree is also called depth of the tree.
10. **Degree of tree:** The maximum degree of the node is called the degree of the tree.

BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

A binary tree is either empty or consists of
a) a node called the root
b) left and right sub trees are themselves binary trees.

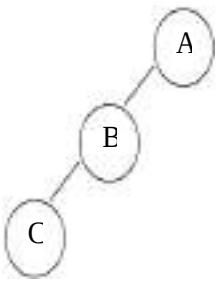
A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub-tree.

In binary tree each node will have one data field and two pointer fields for representing thesub-branches. The degree of each node in the binary tree will be at the most two.

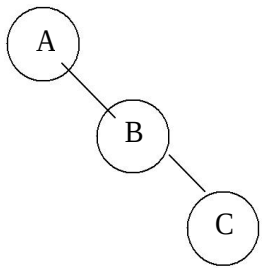
Types Of Binary Trees:

There are 3 types of binary trees:

1. **Left skewed binary tree:** If the right sub-tree is missing in every node of a tree we call it as left skewed tree.

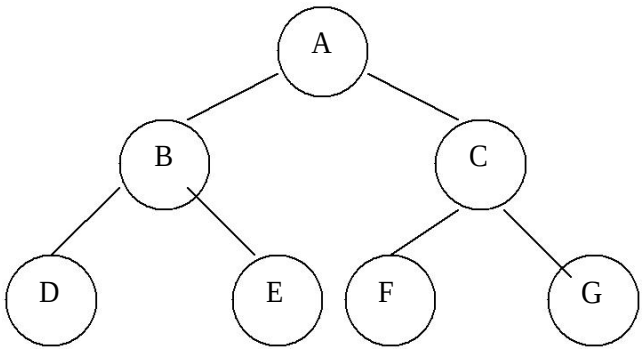


2. **Right skewed binary tree:** If the left sub-tree is missing in every node of a tree we call it is right sub-tree.



3. **Complete binary tree:**

The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth d will contain exactly 2^l nodes at each level *l*, where *l* is from 0 to d.



Note:

1. A binary tree of depth n will have maximum 2^n-1 nodes.
2. A complete binary tree of level *l* will have maximum 2*l* nodes at each level, where *l* starts from 0.
3. Any binary tree with n nodes will have at the most n+1 null branches.
4. The total number of edges in a complete binary tree with n terminal nodes are 2(n-1).

Binary Tree Representation

A binary tree can be represented mainly in 2 ways:

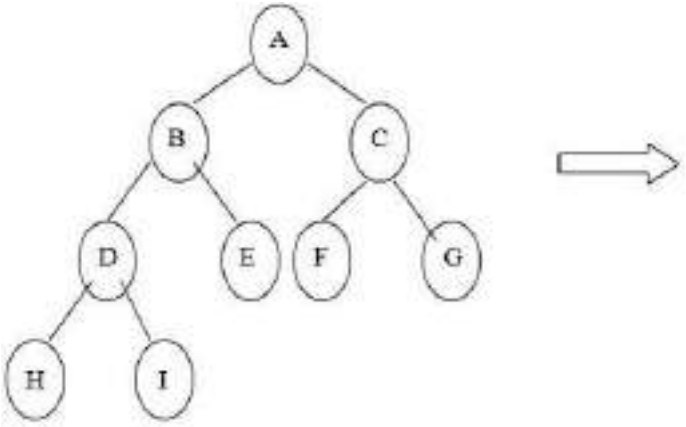
a) Sequential Representation
b) Linked Representation

a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.

1) The root of binary tree is stored in the 1 st location of array
2) If a node is in the j th location of array, then its left child is in the location 2J+1 and its right child in the location 2J+2

The maximum size that is required for an array to store a tree is 2^{d+1}-1, where d is the depth of the tree.



POSITION	ARRAY
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
.	.
.	.
.	.
.	.
.	.

Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular nodeis fast because of the random access.

Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.
2. In this type of representation the maximum depth of the tree has to be fixed. Because we have decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we choose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary tree can be preserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

b) Linked Representation

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer.

In linked list each node will look like this:

Left Child	Data	Right Child

Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.

2. Insertions and deletions which are the most common operations can be done without moving the nodes.

Disadvantages of linked representation:

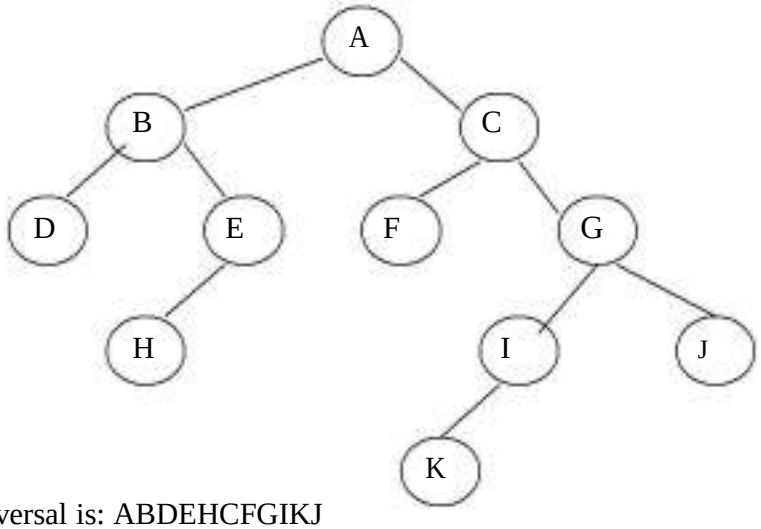
1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

TRAVERSING A BINARY TREE

Traversing a tree means that processing it so that each node is visited exactly once. A binary tree can be
traversed a number of ways.The most common tree traversals are

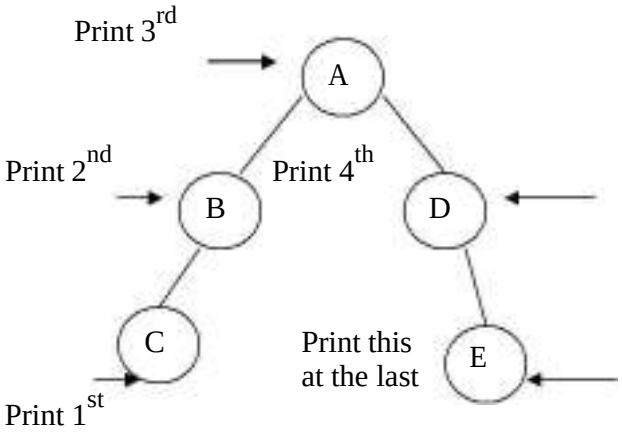
- In-order
- Pre-order and
- Post-order

Pre-order	1.Visit the root 2.Traverse the left sub tree in pre-order 3.Traverse the right sub tree in pre-order.	Root | Left | Right
In-order	1.Traverse the left sub tree in in-order 2.Visit the root 3.Traverse the right sub tree in in-order.	Left | Root | Right
Post-order	1.Traverse the left sub tree in post-order 2.Traverse the right sub tree in post-order. 3.Visit the root	Left | Right | Root



The pre-order traversal is: ABDEHCFGIKJ
The in-order traversal is : DBHEAFCKIGJ
The post-order traversal is:DHEBFKIJGCA

Inorder Traversal:

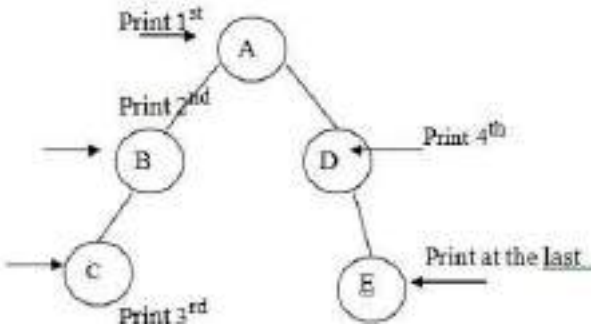


C-B-A-D-E is the inorder traversal i.e. first we go towards the leftmost node. i.e. C so print that node C. Then go back to the node B and print B. Then root node A then move towards the right sub-tree print D and finally E. Thus we are following the tracing sequence of Left|Root|Right. This type of traversal is called inorder traversal. The basic principle is to traverse left sub-tree then root and then the right sub-tree.

Pseudo Code:

```
template <class T>
void inorder(bintree<T> *temp)
{
        if(temp!=NULL)
        {
                inorder(temp->left);
                cout<<"temp->data";
                inorder(temp->right);
        }
}
```

Preorder Traversal:

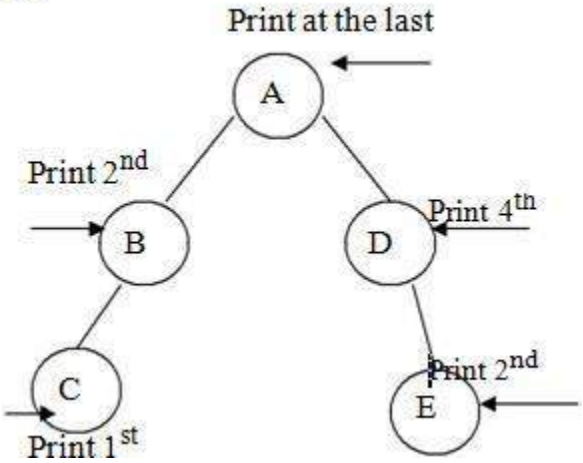


is the preorder traversal of the above fig. We are following Root|Left|Right path i.e. data at the root node will be printed first then we move on the left sub-tree and go on printing the data till we reach to the left most node. Print the data at that node and then move to the right sub- tree. Follow the same principle at each sub-tree and go on printing the data accordingly.

```
template <class T>
void preorder(bintree<T> *temp)
```

```
{
        if(temp!=NULL)
        {
                cout<<"temp->data";          preorder(temp->left);
                preorder(temp->right);
        }
}
```

Postorder Traversal:

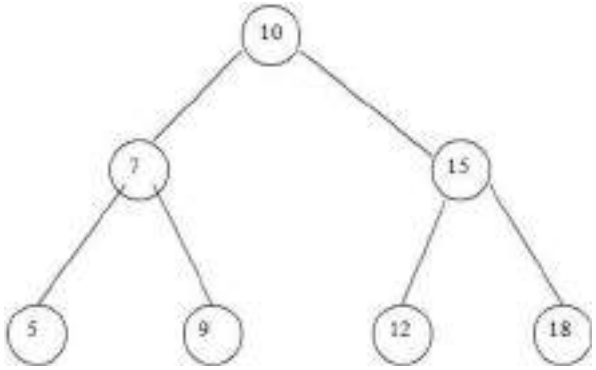


From figure the postorder traversal is C-D-B-E-A. In the postorder traversal we are following the Left|Right|Root principle i.e. move to the leftmost node, if right sub-tree is there or not if not then print the leftmost node, if right sub-tree is there move towards the right most node. The key idea here is that at each sub-tree we are following the Left|Right|Root principle and print the data accordingly.
Pseudo Code:

```
template <class T>
void postorder(bintree<T> *temp)
{
        if(temp!=NULL)
        {
                postorder(temp->left);
                postorder(temp->right);
                cout<<"temp->data";
        }
}
```

BINARY SEARCH TREE

In the simple binary tree the nodes are arranged in any fashion. Depending on user's desire the new nodes can be attached as a left or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. And thus the searching time complexity will get increased unnecessarily. So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at **left sub-tree < root node value < right sub-tree values.**



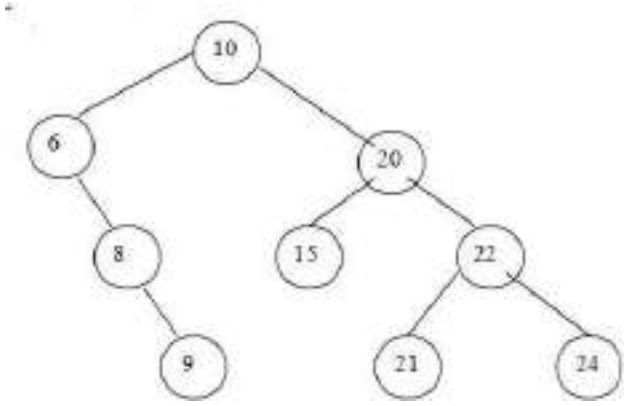
Operations On Binary Search Tree:

The basic operations which can be performed on binary search tree are.

1. Insertion of a node in binary search tree.
2. Deletion of a node from binary search tree.
3. Searching for a particular node in binary search tree.

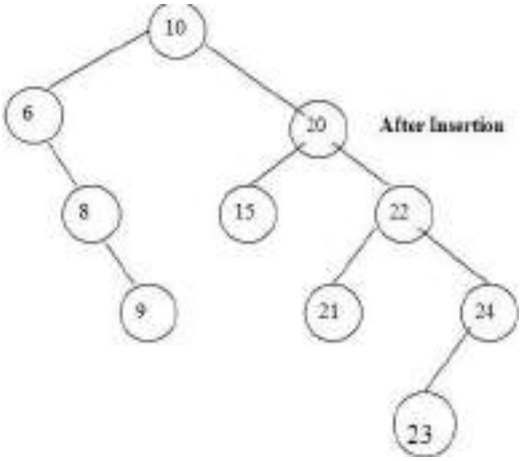
Insertion of a node in binary search tree.

While inserting any node in binary search tree, look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the value of the current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.



Before Insertion

In the above fig, if we wan to insert 23. Then we will start comparing 23 with value of root node i.e. 10. As 23 is greater than 10, we will move on right sub-tree. Now we will compare 23 with 20 and move right, compare 23 with 22 and move right. Now compare 23 with 24 but it is less than 24. We will move on left branch of 24. But as there is node as left child of 24, we can attach 23 as left child of 24.



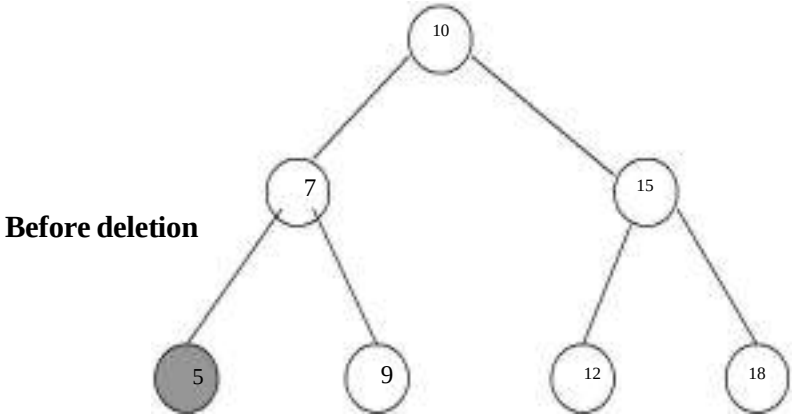
Deletion of a node from binary search tree.

For deletion of any node from binary search tree there are three which are possible.

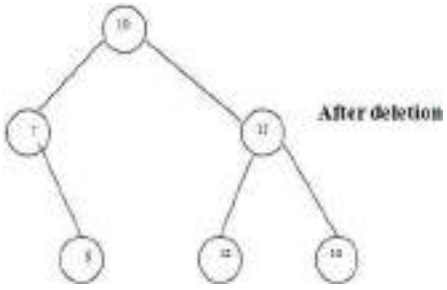
i. Deletion of leaf node.
ii. Deletion of a node having one child.
iii. Deletion of a node having two children.

Deletion of leaf node.

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.

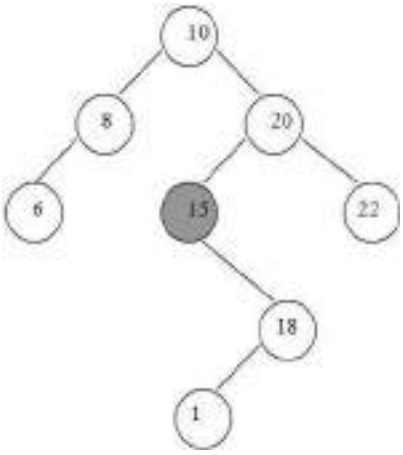


From the above fig, we want to delete the node having value 5 then we will set left pointer of its parent node as NULL. That is left pointer of node having value 7 is set to NULL.

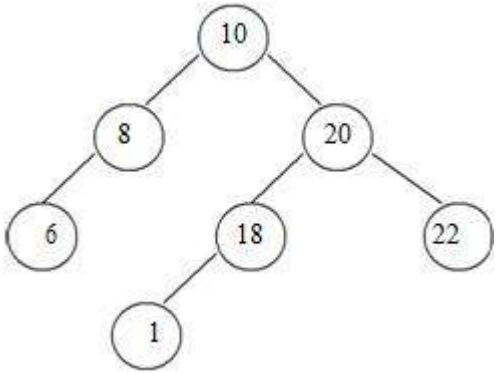


Deletion of a node having one child.

To explain this kind of deletion, consider a tree as given below.

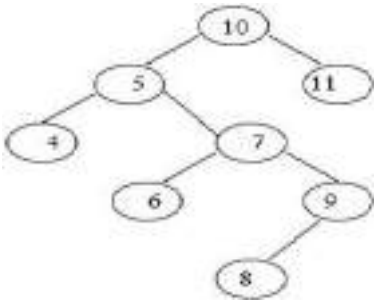


If we want to delete the node 15, then we will simply copy node 18 at place of 16 and then set the node free



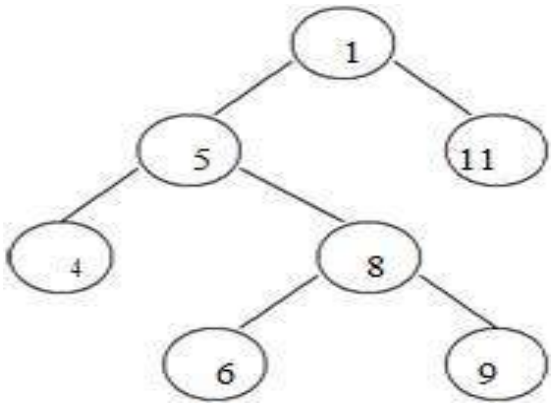
Deletion of a node having two children.

Consider a tree as given below.



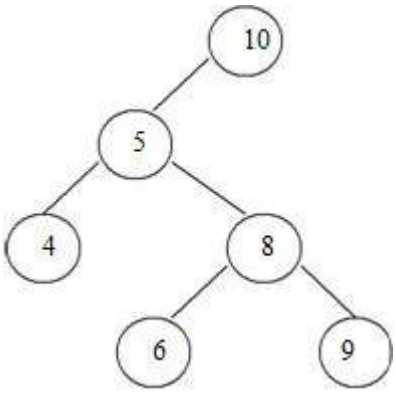
Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. We will then find out the inorder successor of node 7. The inorder successor will be simply copied at location of node 7.

That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.



Searching for a node in binary search tree.

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare "node is not present in the tree".



In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub tree. Now compare 9 with 8 but 9 is greater than 8 we will move on right sub branch. As the node we will get holds the value 9. Thus the desired node can be searched.

AVL TREES

Adelsion Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to height of sub trees. The tree can be made balanced and because of this retrieval of any node can be done in O(log n) times, where n is total number of nodes. From the name of these scientists the tree is called AVL tree.

Definition:

An empty tree is height balanced if T is a non empty binary tree with T_L and T_R as its left and right sub trees. The T is height balanced if and only if

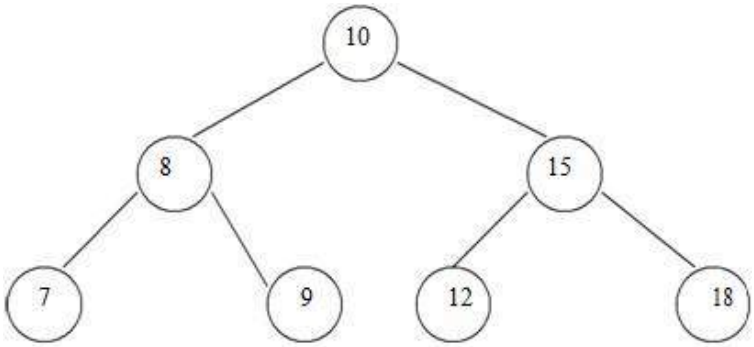
i. T_L and T_R are height balanced.
ii. h_L-h_R <= 1 where h_L and h_R are heights of T_L and T_R.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

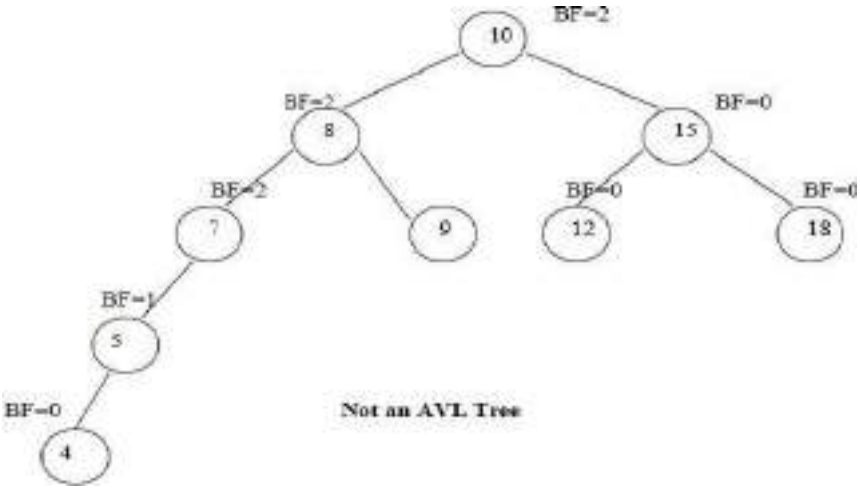
Definition of Balance Factor:

The balance factor BF(T) of a node in binary tree is defined to be h_L-h_R where h_L and h_R are heights of left and right sub trees of T.

For any node in AVL tree the balance factor i.e. BF(T) **is -1, 0 or +1**.



AVL Tree



Not an AVL Tree

Height of AVL Tree:

Theorem: The height of AVL tree with n elements (nodes) is O(log n).

Proof: Let an AVL tree with n nodes in it. N_h be the minimum number of nodes in an AVL tree of height h.

In worst case, one sub tree may have height h-1 and other sub tree may have height h-2. And both these sub trees are AVL trees. Since for every node in AVL tree the height of left and right sub trees differ by at most 1.

Hence

$N_h = N_{h-1}+N_{h-2}+1$

Where N_h denotes the minimum number of nodes in an AVL tree of height h.

$N_0=0$ $N_1=2$

We can also write it as
$N > N_h = N_{h-1}+N_{h-2}+1$

$> 2N_{h-2}$

$> 4N_{h-4}$

.
.
> 2iNh-2i

If value of h is even, let i = h/2-1

Then equation becomes

N > 2h/2-1N2

= N > 2(h-1)/2x4 (N2 = 4)

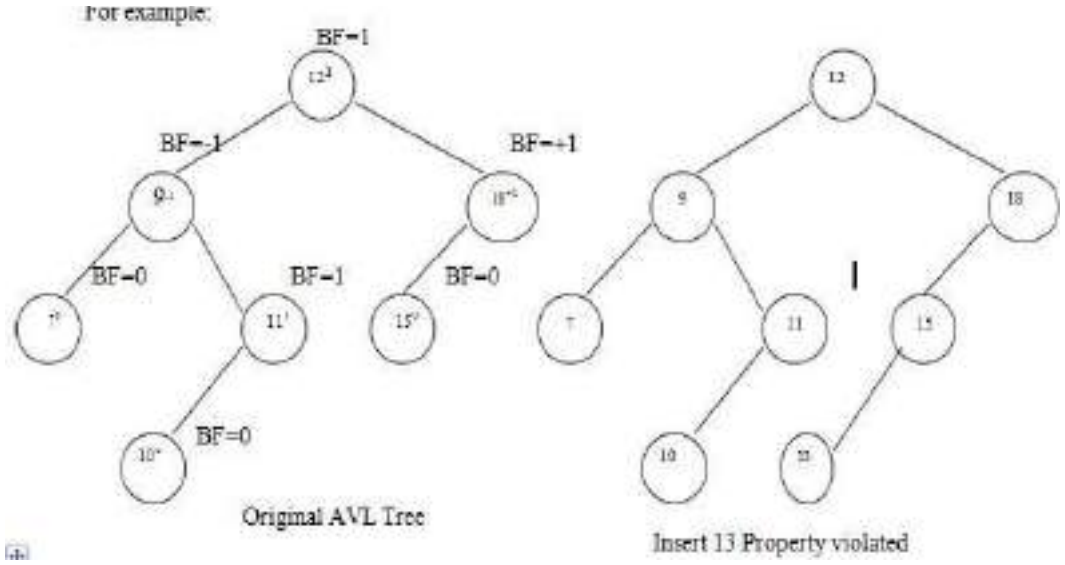
= O(log N)

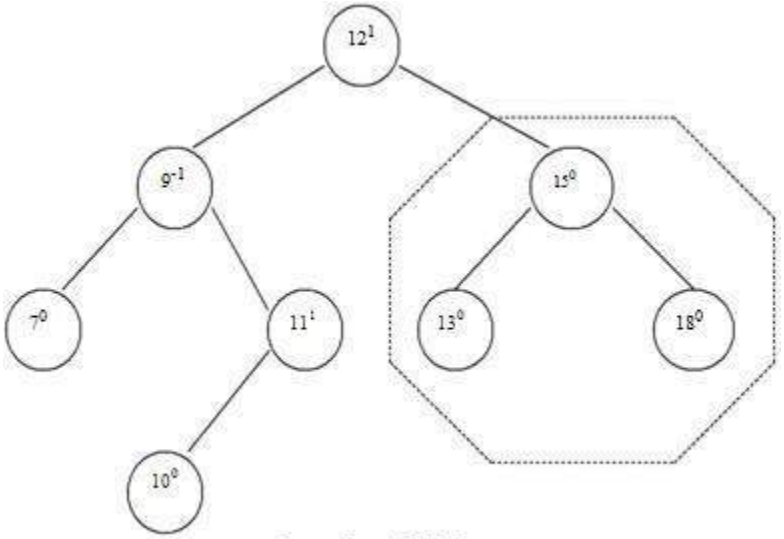
If value of h is odd, let I = (h-1)/2 then equation becomes
N > 2(h-1)/2 N1
N > 2(h-1)/2 x 1
H = O(log N)

This proves that height of AVL tree is always O(log N). Hence search, insertion and deletion can be carried out in logarithmic time.

Representation of AVL Tree

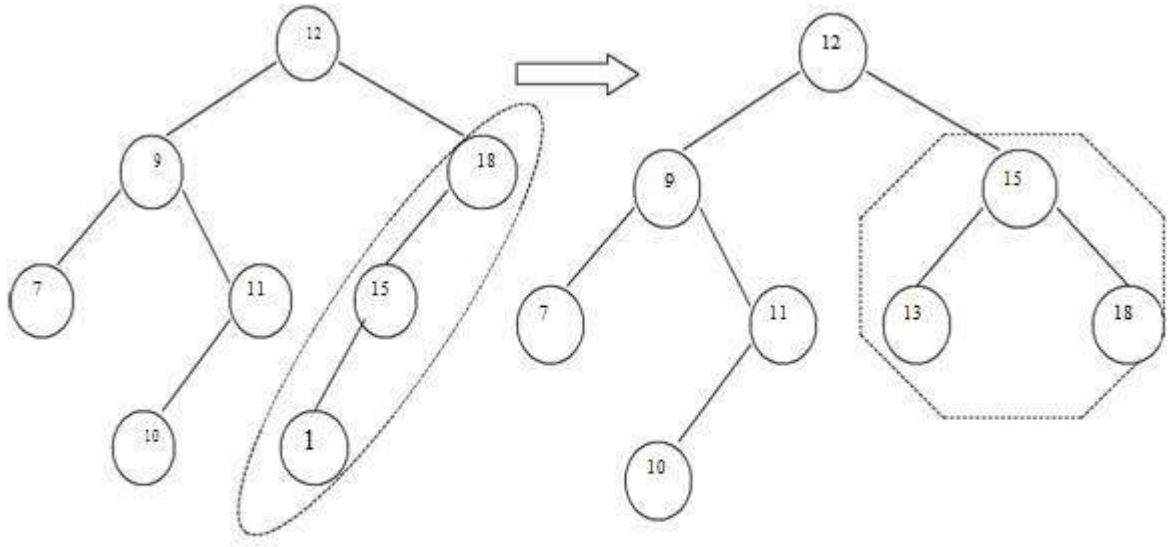
- The AVL tree follows the property of binary search tree. In fact AVL trees are basically binary search trees with balance factors as -1, 0, or +1.
- After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0, or +1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition. The balance factor is denoted at right top corner inside the node.





- After insertion of a new node if balance condition gets destroyed, then the nodes on that path(new node insertion point to root) needs to be readjusted. That means only the affected sub tree is to be rebalanced.
- The rebalancing should be such that entire tree should satisfy AVL property.

In above given example-



Nodes 18, 15, 13 are to be adjusted

By adjusting 15 the entire Tree satisfies AVL property

Insertion of a node.

There are four different cases when rebalancing is required after insertion of new node.

1. An insertion of new node into left sub tree of left child. (LL).
2. An insertion of new node into right sub tree of left child. (LR).
3. An insertion of new node into left sub tree of right child. (RL).
4. An insertion of new node into right sub tree of right child.(RR).

Some modifications done on AVL tree in order to rebalance it is called rotations of AVL tree

There are two types of rotations:

Single rotation
- Left-Left(LL rotation)
- Right-Right(RR rotation)

Double rotation
- Left-Right(LR rotation)
- Right-Left(RL rotation)

Insertion Algorithm:

1. Insert a new node as new leaf just as an ordinary binary search tree.
2. Now trace the path from insertion point(new node inserted as leaf) towards root. For each node 'n' encountered, check if heights of left (n) and right (n) differ by at most 1.

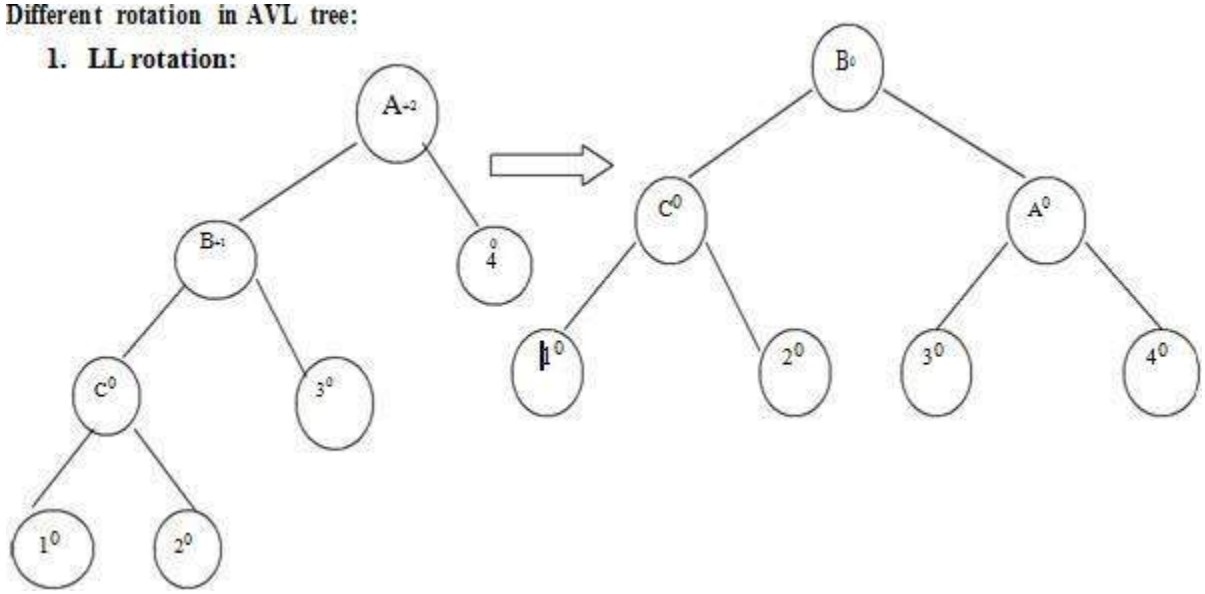
a) If yes, move towards parent (n).

b) Otherwise restructure by doing either a single rotation or a double rotation.

Thus once we perform a rotation at node 'n' we do not require to perform any rotation at any ancestor on 'n'.

Different rotation in AVL tree:

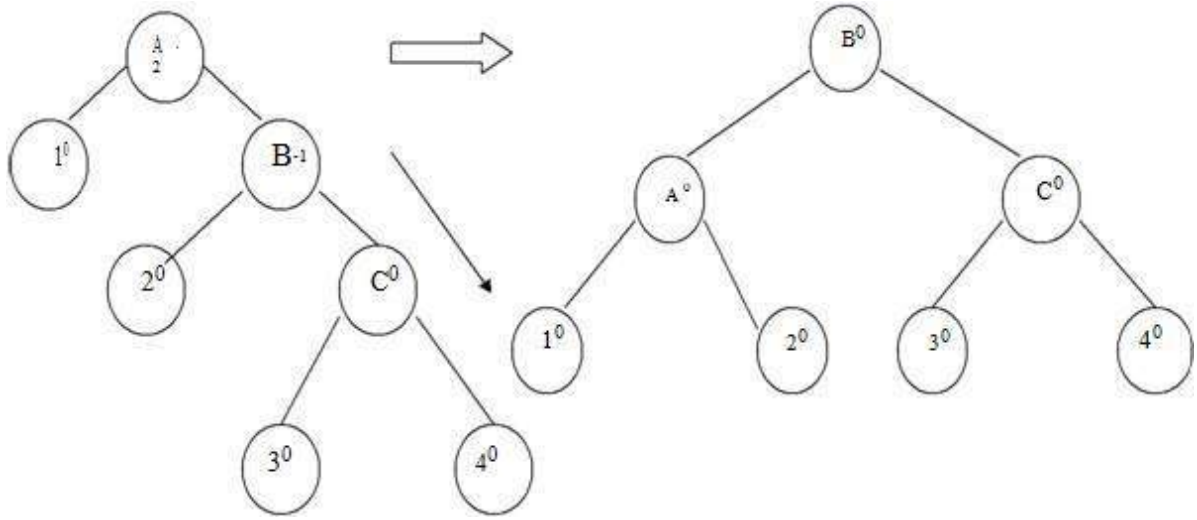
1. LL rotation:



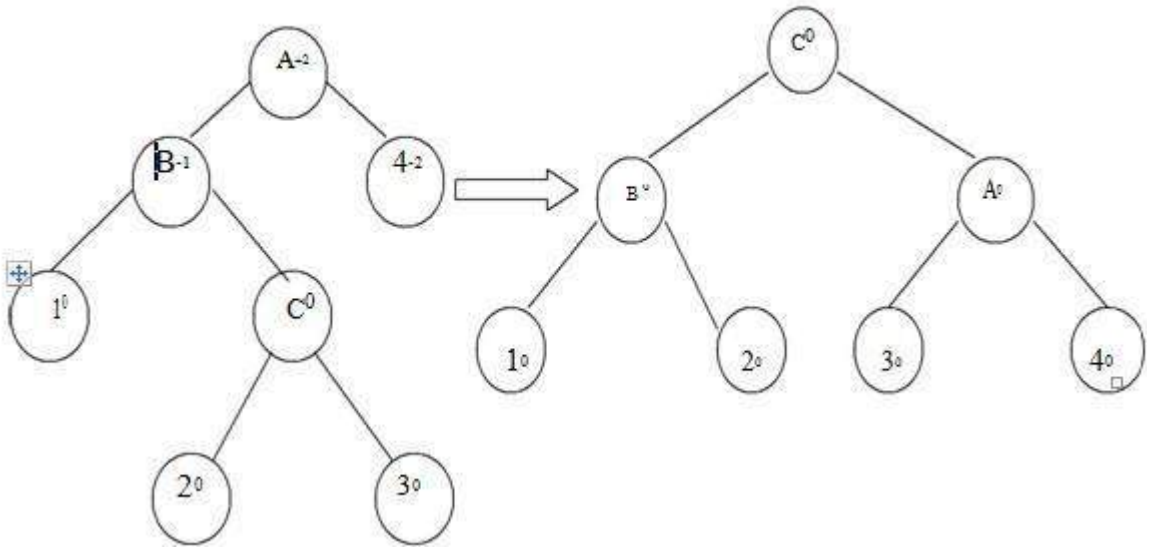
When node '1' gets inserted as a left child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2.
The LL rotation has to be applied to rebalance the nodes.

2. RR rotation:

When node '4' gets attached as right child of node 'C' then node 'A' gets unbalanced. The rotation which needs to be applied is RR rotation as shown in fig.

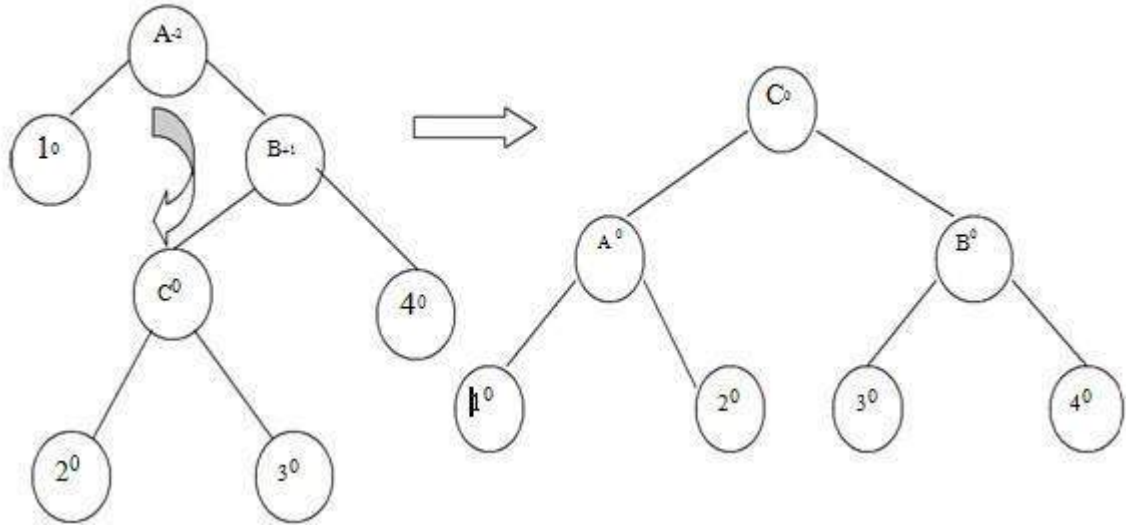


3. LR rotation:



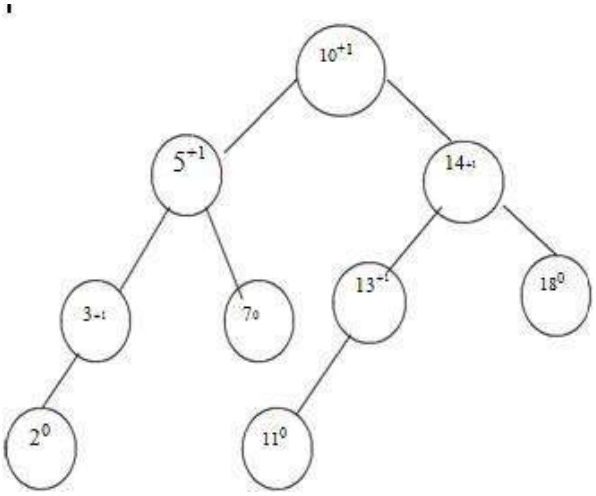
When node '3' is attached as a right child of node 'C' then unbalancing occurs because of LR. Hence LR rotation needs to be applied.

4. RL rotation



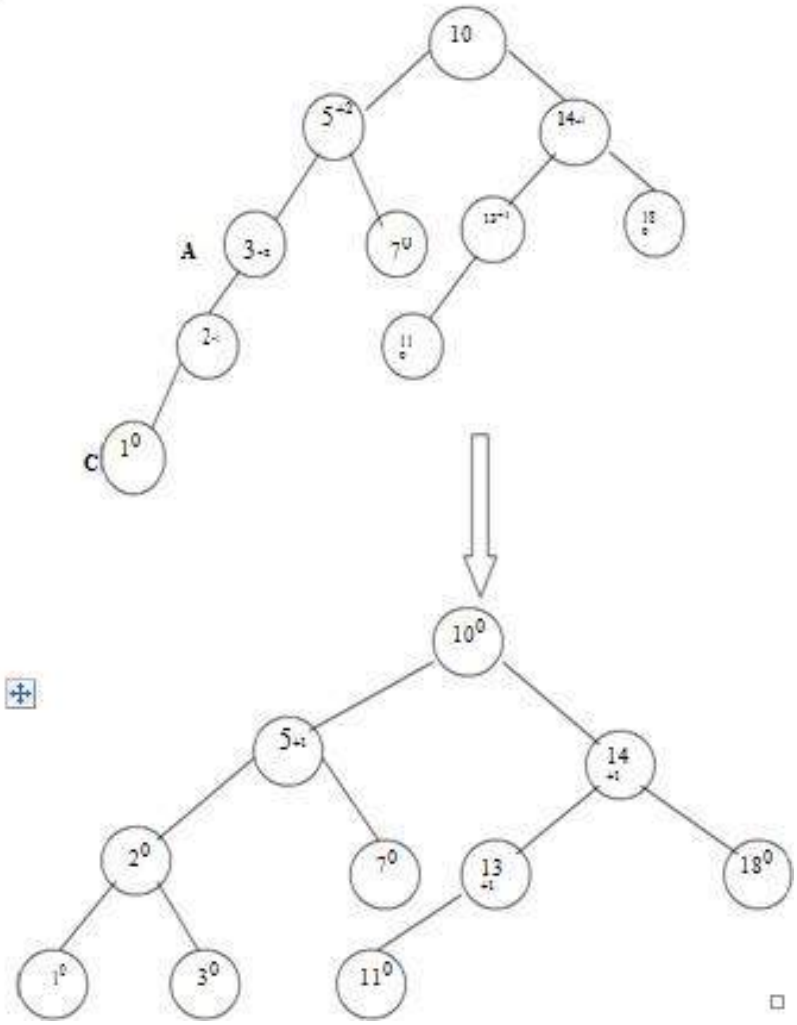
When node '2' is attached as a left child of node 'C' then node 'A' gets unbalanced as its balance factor becomes -2. Then RL rotation needs to be applied to rebalance the AVL tree.
Example:

Insert 1, 25, 28, 12 in the following AVL tree.



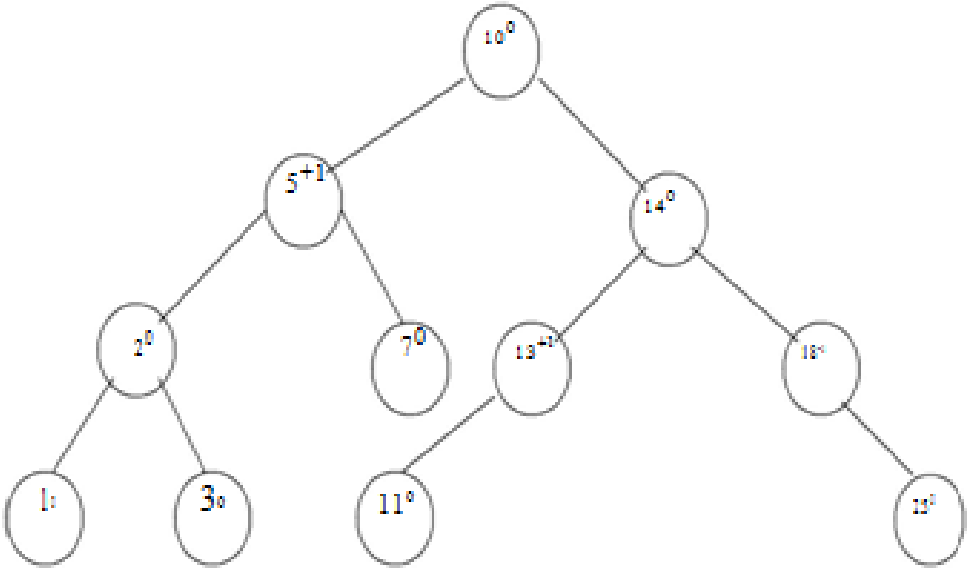
Insert 1

To insert node ‘1’ we have to attach it as a left child of ‘2’. This will unbalance the tree as follows. We will apply LL rotation to preserve AVL property of it.



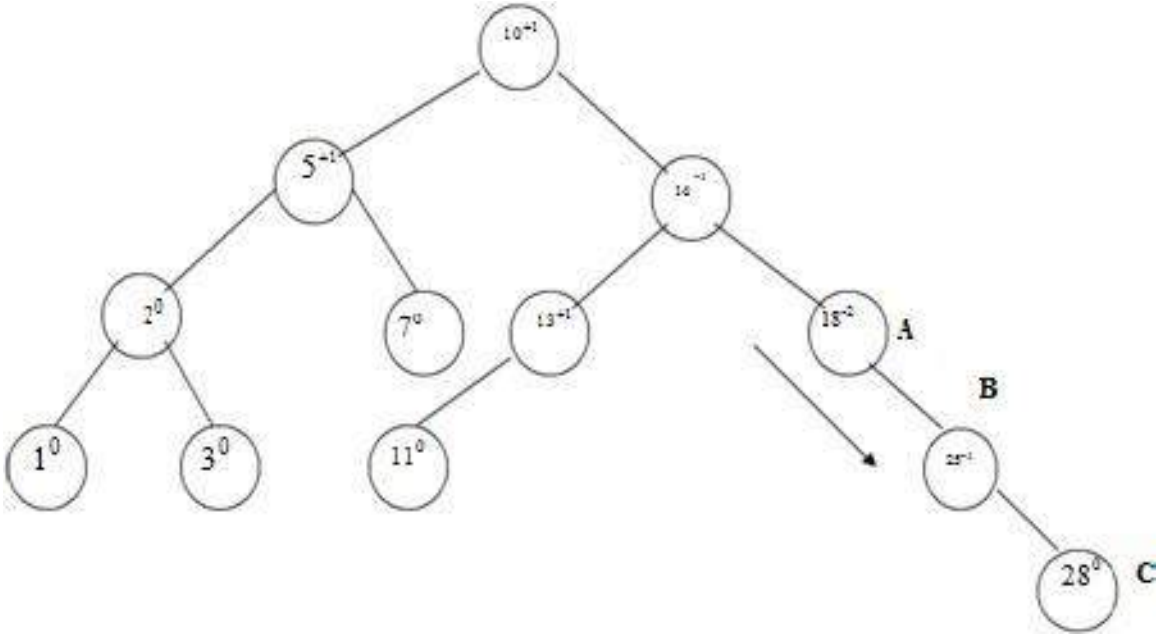
Insert 25

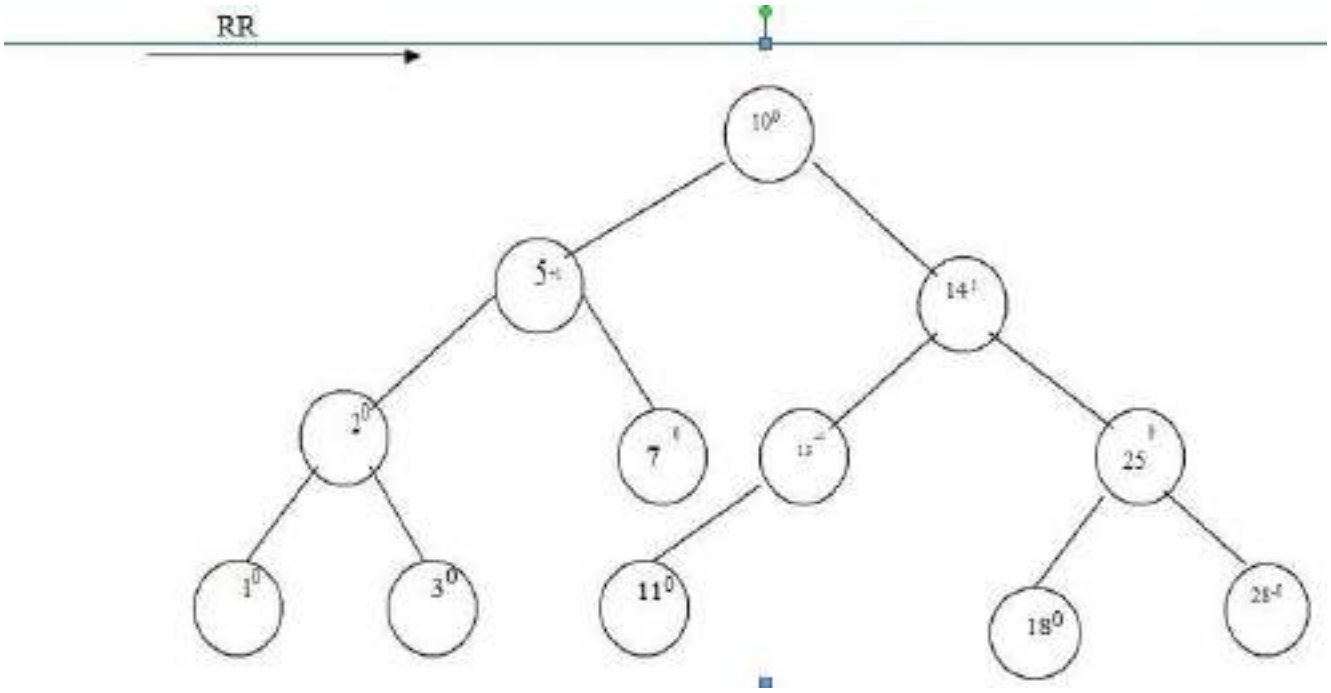
We will attach 25 as a right child of 18. No balancing is required as entire tree preserves the AVL property



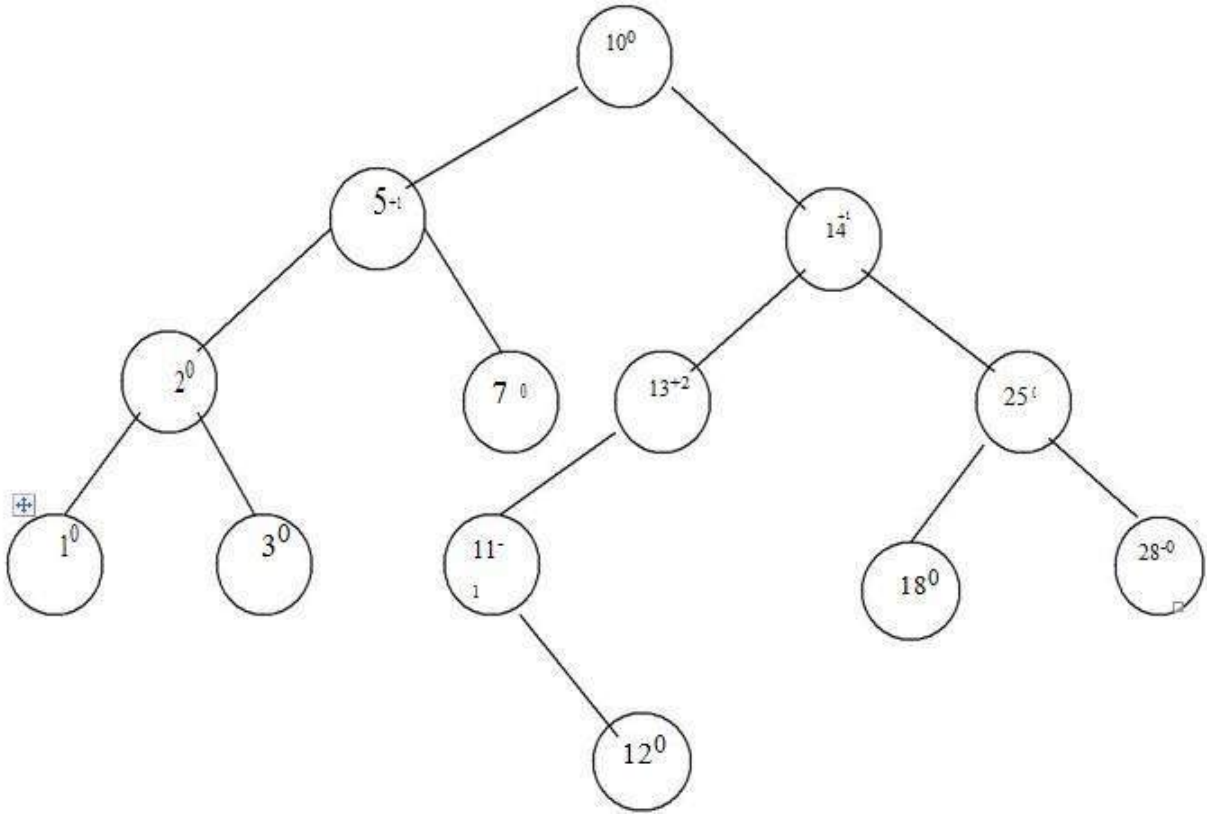
Insert 28

The node '28' is attached as a right child of 25. RR rotation is required to rebalance.

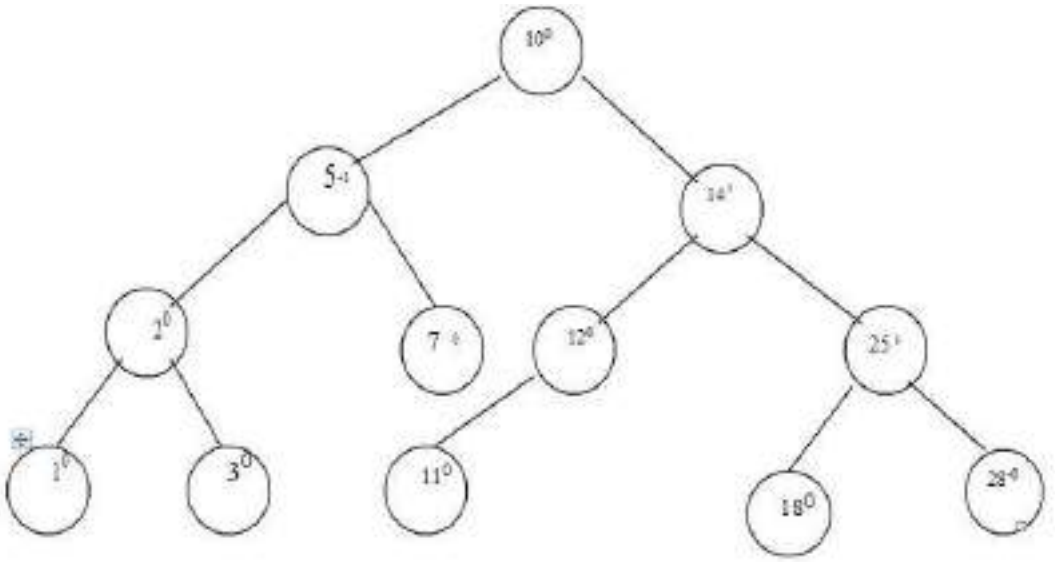




Insert 12



To rebalance the tree we have to apply LR rotation.



Thus by applying various rotations depending upon direction of insertion of new node the AVL tree can be restructured.

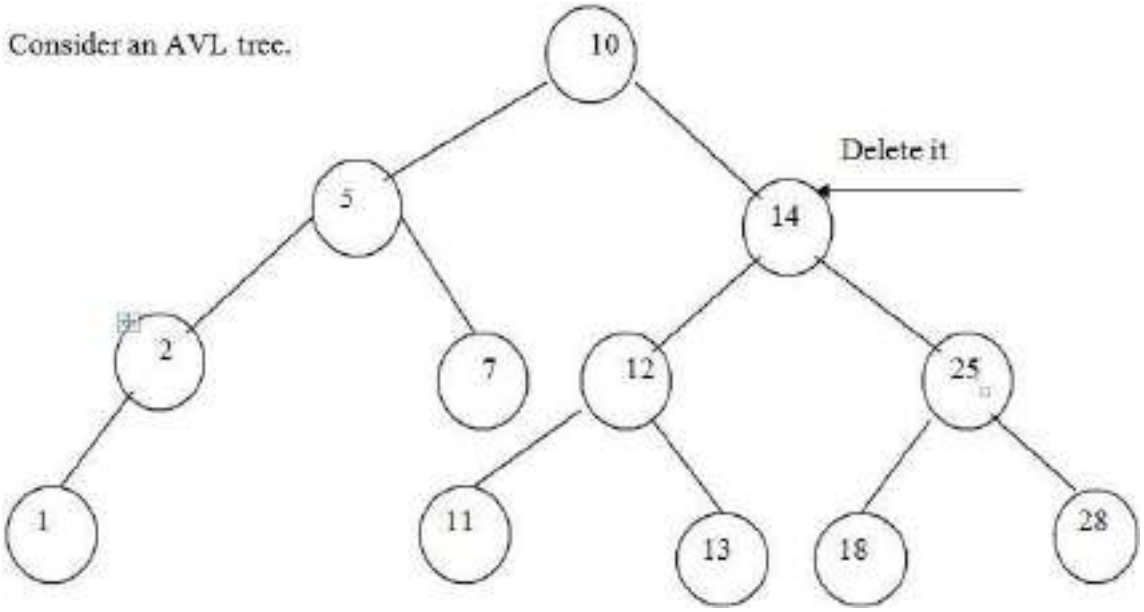
Deletion of a node:

Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And thereby various rotations need to be applied.

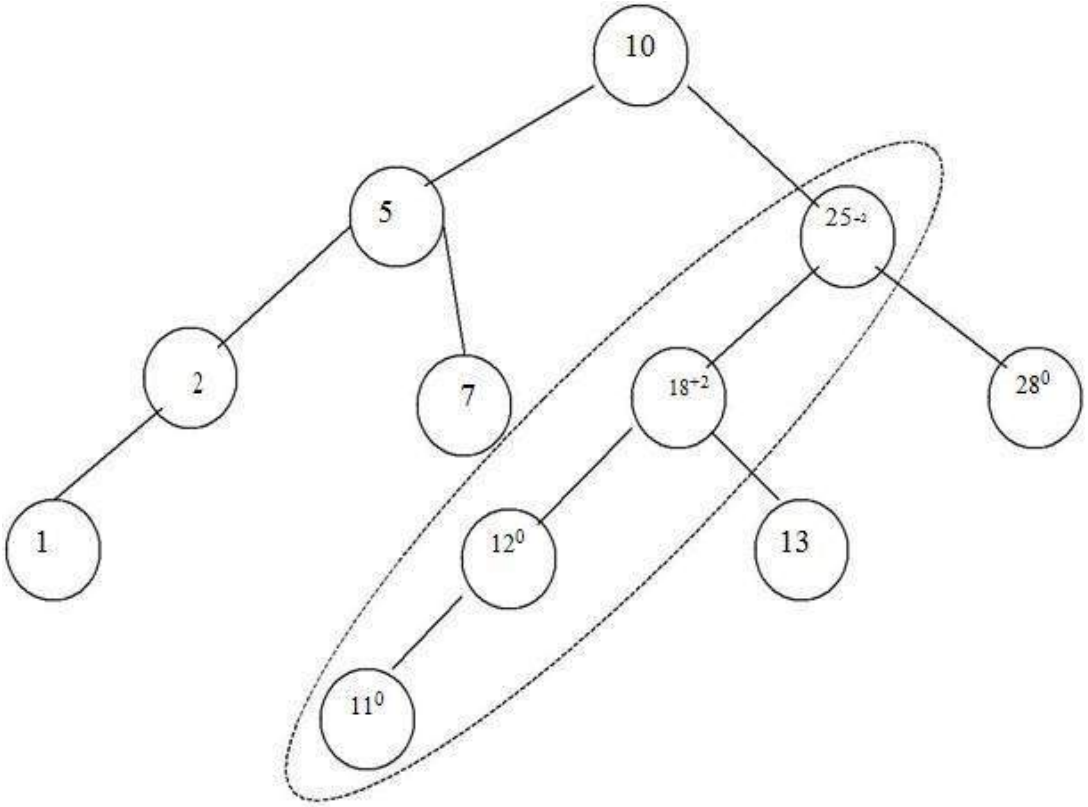
Algorithm for deletion:

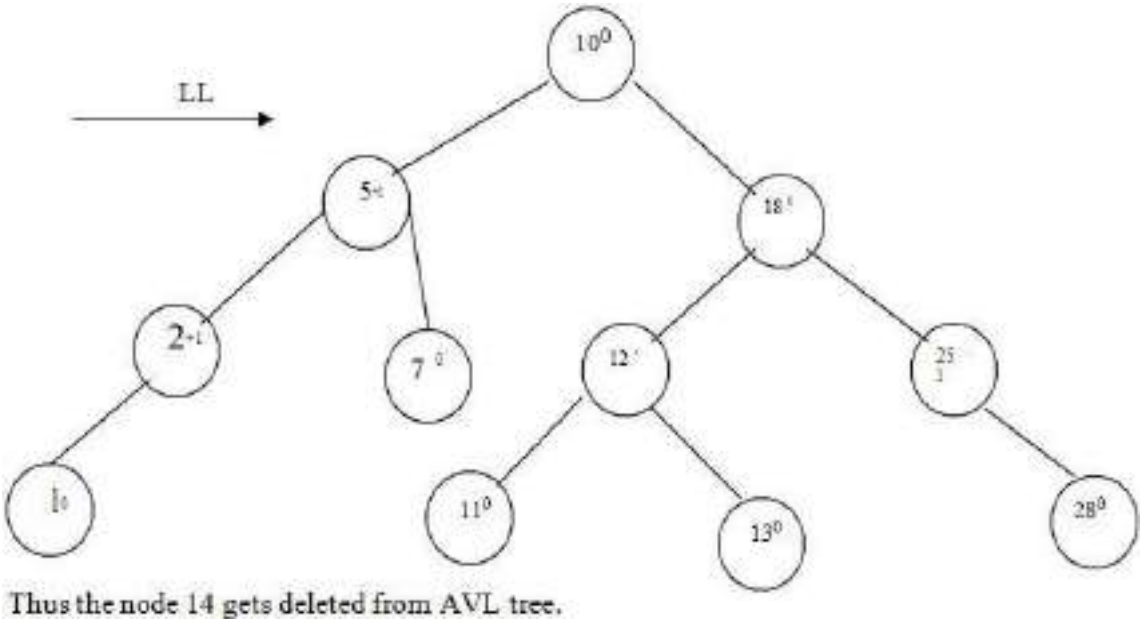
The deletion algorithm is more complex than insertion algorithm.

1. Search the node which is to be deleted.
2. a) If the node to be deleted is a leaf node then simply make it NULL to remove.
   b) If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove this node.
3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some sub tree then balance that sub tree using appropriate single or double rotations. The deletion algorithm takes O(log n) time to delete any node.



The tree becomes





Thus the node 14 gets deleted from AVL tree.

Searching:

The searching of a node in an AVL tree is very simple. As AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is the same one is used to search a node from AVL tree.

The searching of a node from AVL tree takes O(log n) time.

RED - BLACK TREE

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

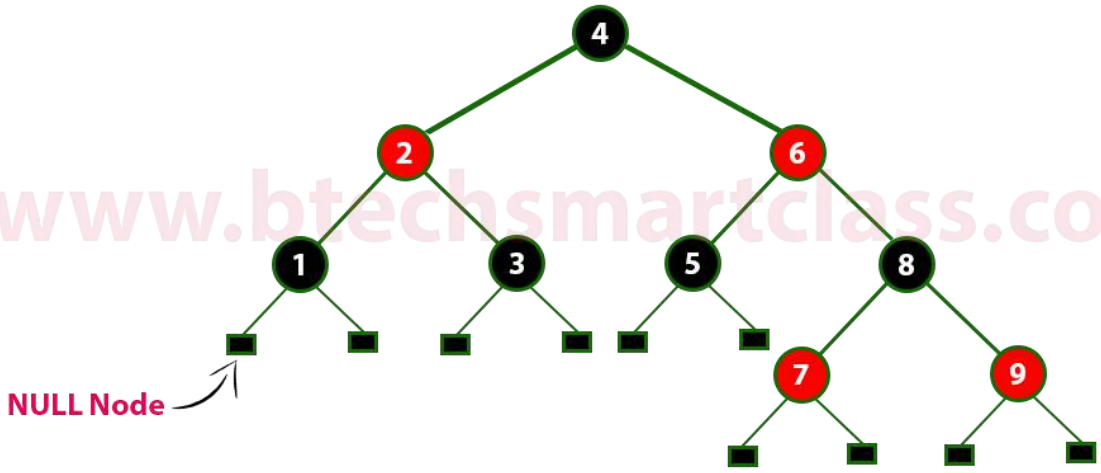
In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

- **Property #1:** Red - Black Tree must be a Binary Search Tree.
- **Property #2:** The ROOT node must be colored BLACK.
- **Property #3:** The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
- **Property #4:** In all the paths of the tree, there should be same number of BLACK colored nodes.
- **Property #5:** Every new node must be inserted with RED color.
- **Property #6:** Every leaf (e.i. NULL node) must be colored BLACK.

Example

Following is a Red-Black Tree which is created by inserting numbers from 1 to 9.



The above tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

- **1. Recolor**
- **2. Rotation**
- **3. Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node with color Red.
- **Step 4 -** If the parent of newNode is Black then exit from the operation.
- **Step 5 -** If the parent of newNode is Red then check the color of parentnode's sibling of newNode.
- **Step 6 -** If it is colored Black or NULL then make suitable Rotation and Recolor it.
- **Step 7 -** If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Example

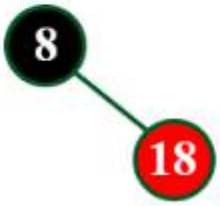
Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

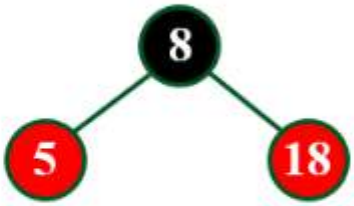
insert (18)

Tree is not Empty. So insert newNode with red color.



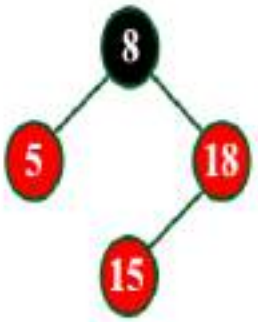
insert (5)

Tree is not Empty. So insert newNode with red color.



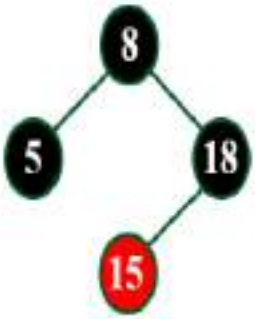
insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

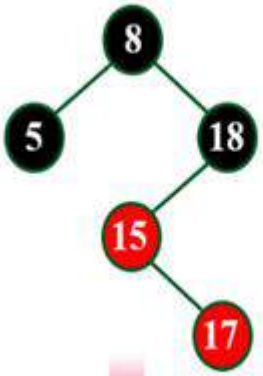
After RECOLOR



After Recolor operation, the tree is satisfying all Red Black Tree properties.

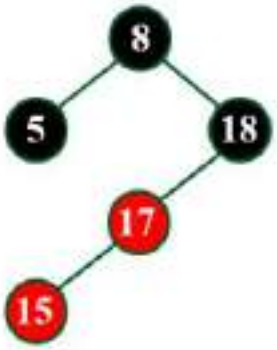
insert (17)

Tree is not Empty. So insert newNode with red color.

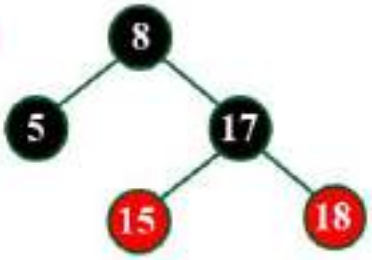


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

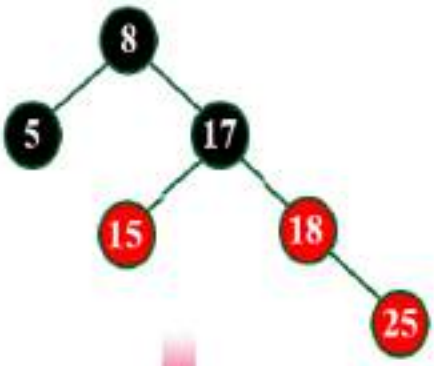


After Right Rotation & Recolor

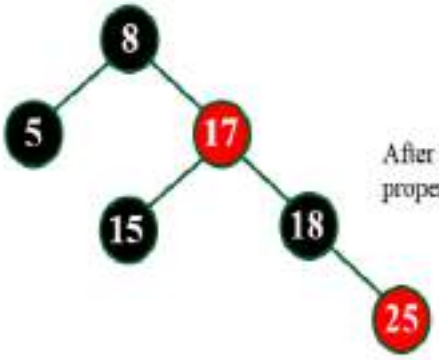


insert (25)

Tree is not Empty. So insert newNode with red color.



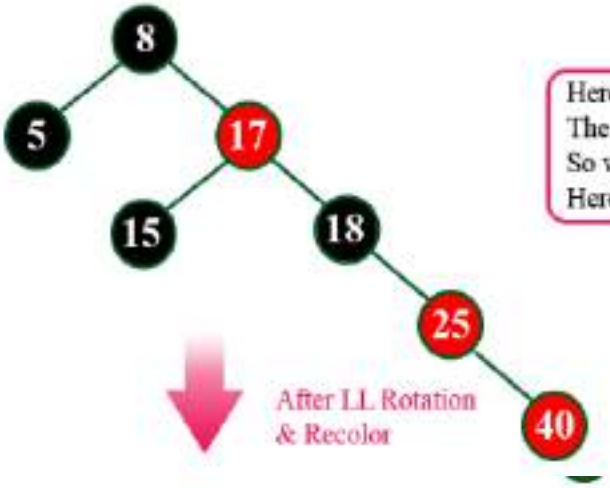
Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.



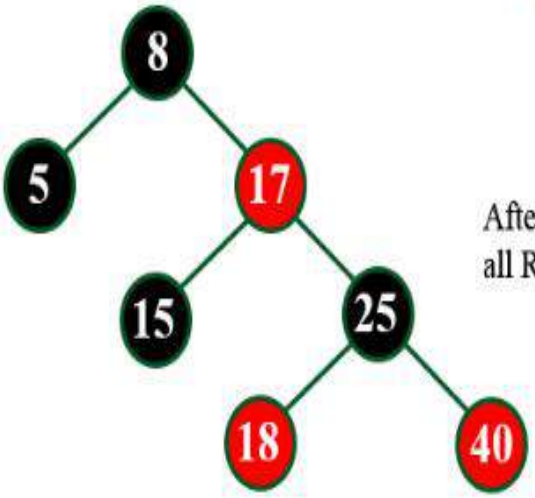
After Recolor operation, the tree is satisfying all Red Black Tree properties.

Insert (40)

Tree is not Empty. So insert newNode with red color.



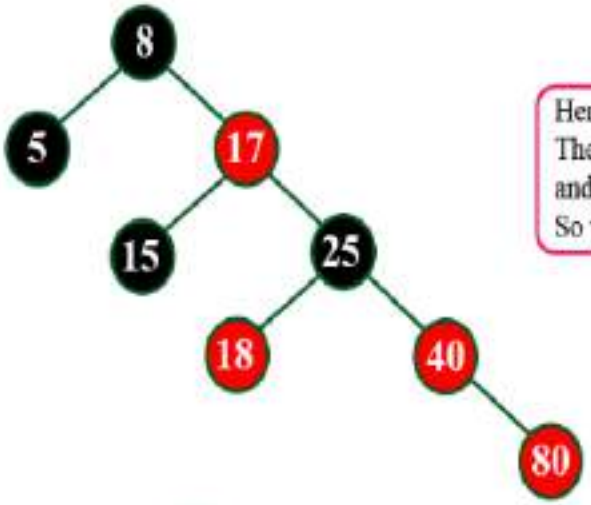
Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.



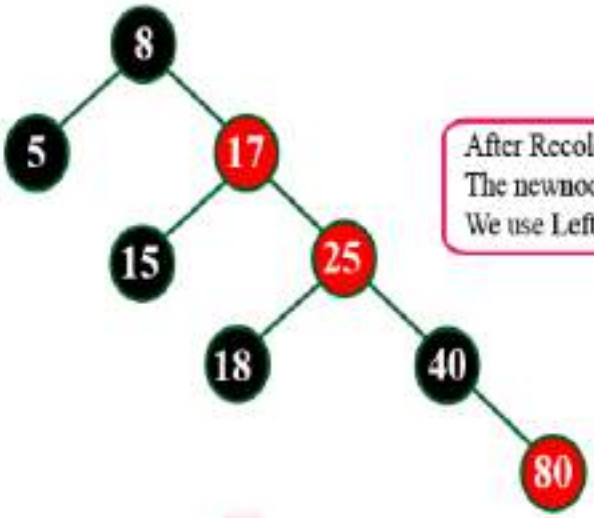
After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (80)

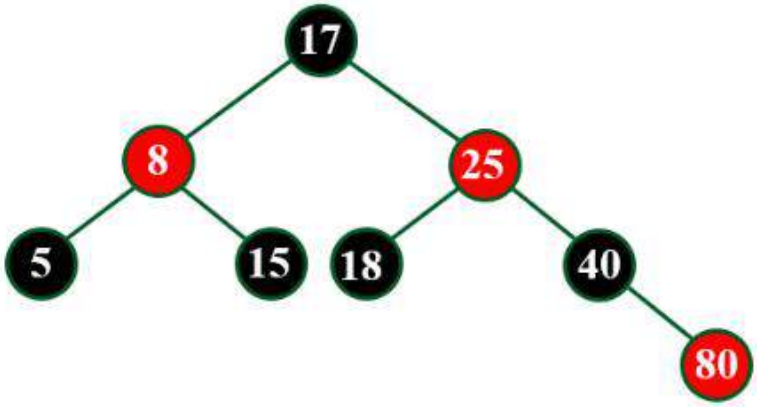
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.



After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.



Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red-Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

Splay Tree

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

In splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

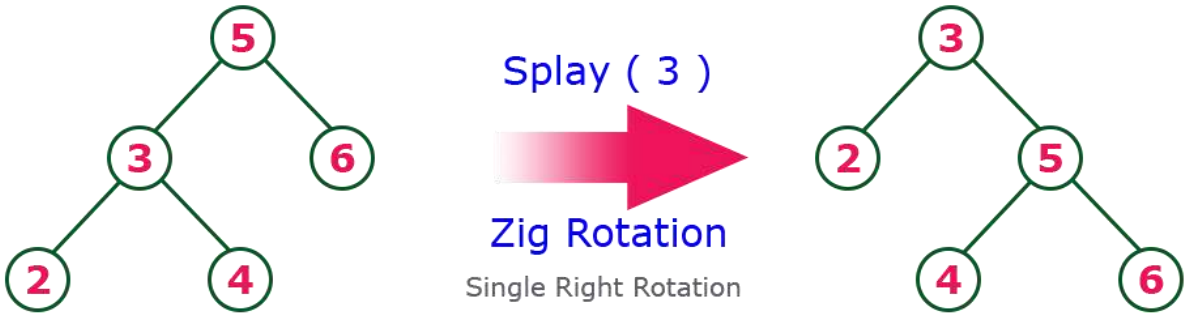
- **1. Zig Rotation**
- **2. Zag Rotation**
- **3. Zig - Zig Rotation**
- **4. Zag - Zag Rotation**
- **5. Zig - Zag Rotation**
- **6. Zag - Zig Rotation**

Example

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations.

In zig rotation, every node moves one position to the right from its current position. Consider the following example...

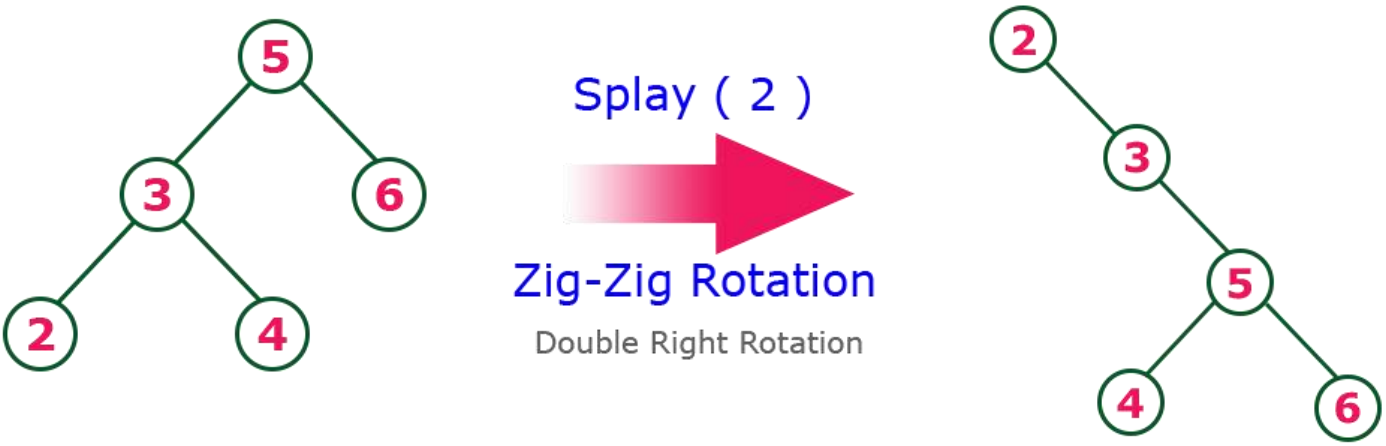


Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...

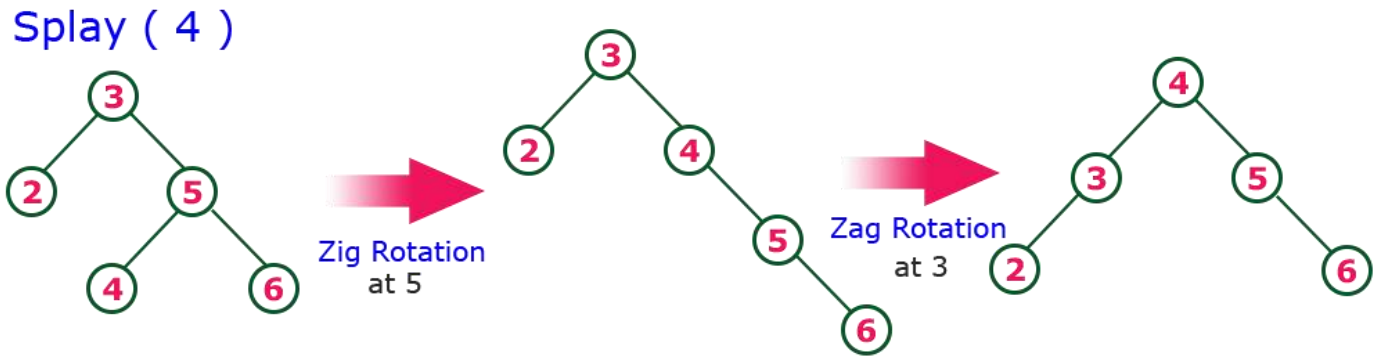


Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...

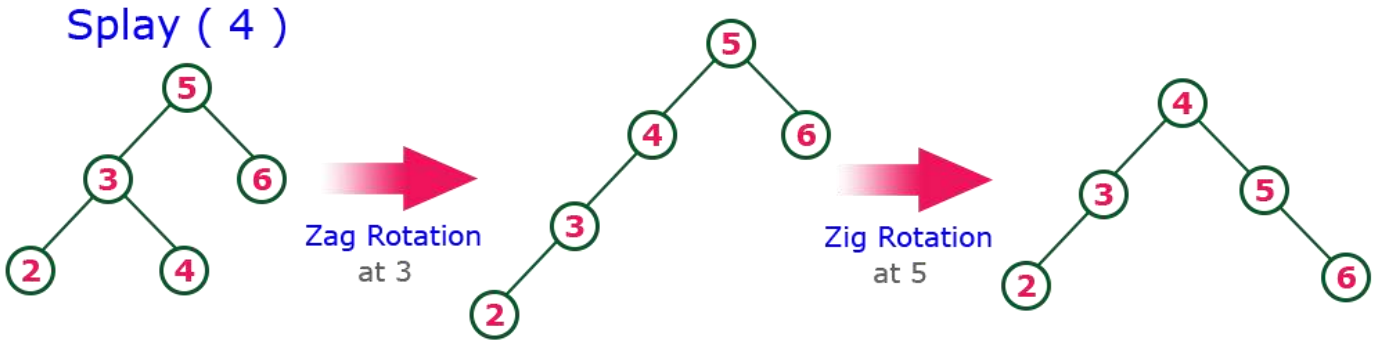
Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- **Step 4 -** After insertion, **Splay** the **newNode**

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

UNIT – IV

Graphs: Graph Implementation Methods. Graph Traversal Methods. Sorting: Heap Sort, External Sorting- Model for external sorting, Merge Sort.

Introduction to Graphs

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

Graph is a collection of vertices and arcs in which vertices are connected with arcs (or)

Graph is a collection of nodes and edges in which nodes are connected with edges

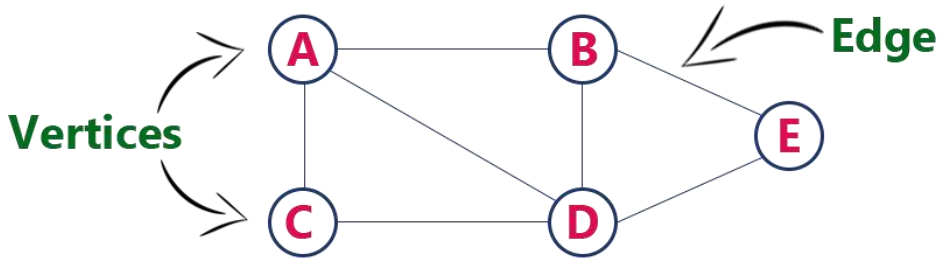
Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 7 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

We use the following terms in graph data structure...

Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge -** An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted edge is a edge with value (cost) on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Graph Representations/ Implementation Methods

Graph data structure is represented using following representations...

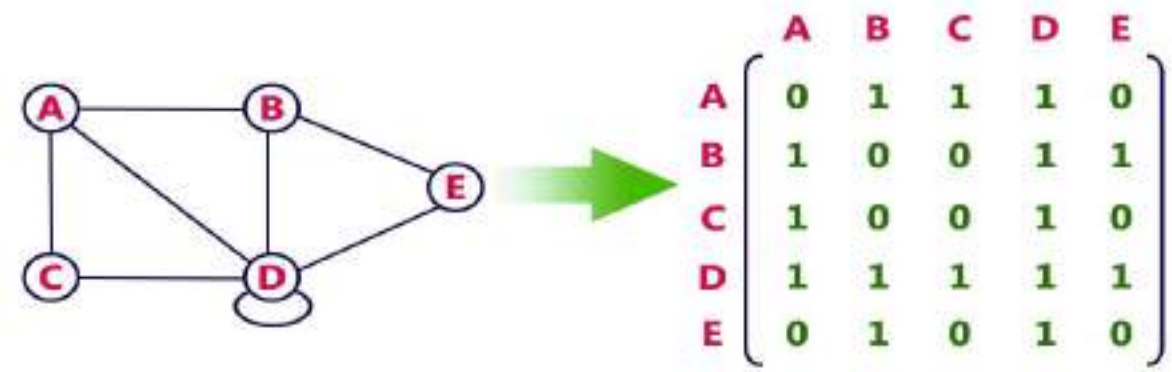
1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

Adjacency Matrix

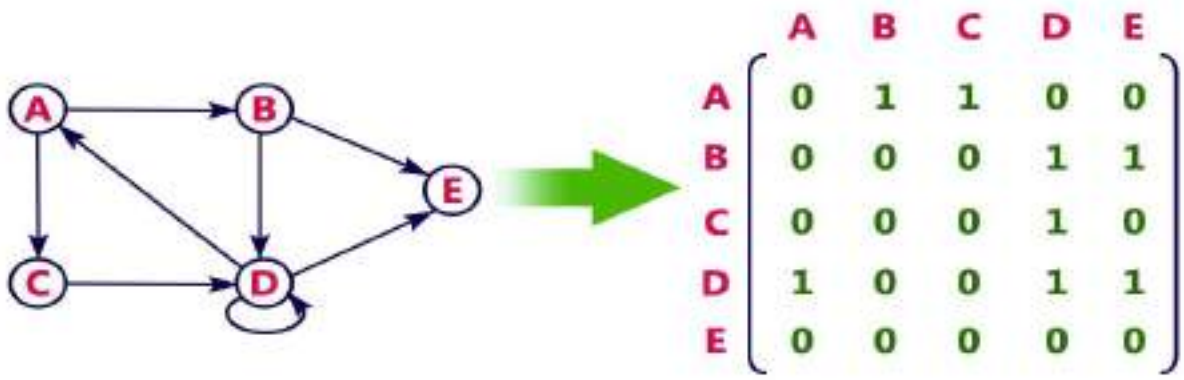
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0.

Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



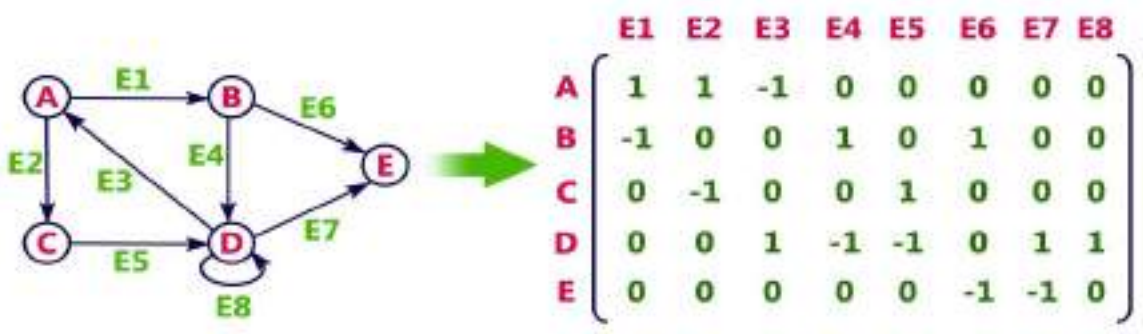
Directed graph representation...



Incidence Matrix

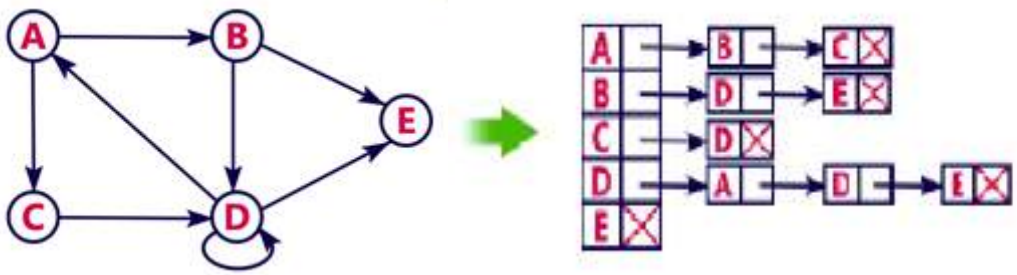
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 7 edges is represented using a matrix of size 4X7. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with **0 or 1 or -1**. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation...

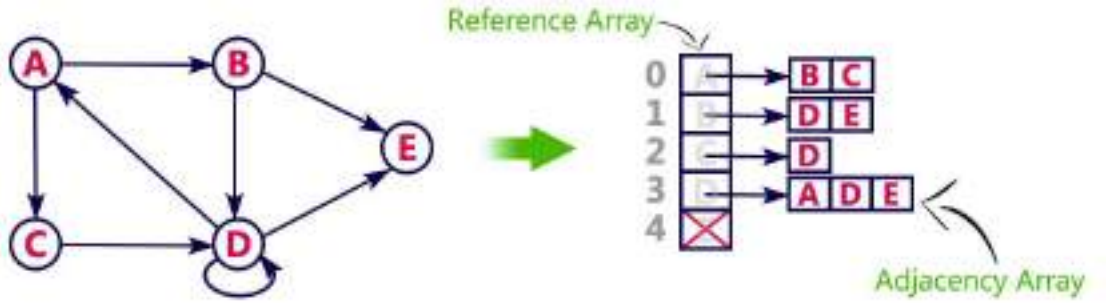


Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Graph Traversal Methods

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

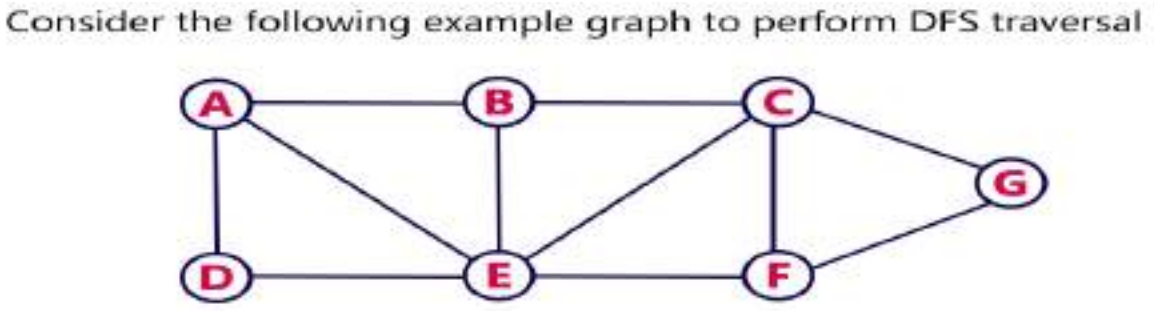
1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal. We use the following steps to implement DFS traversal...

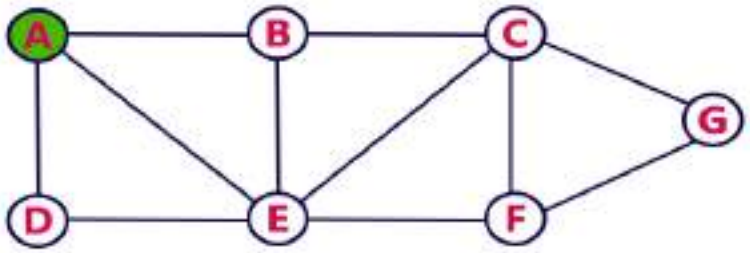
- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Example



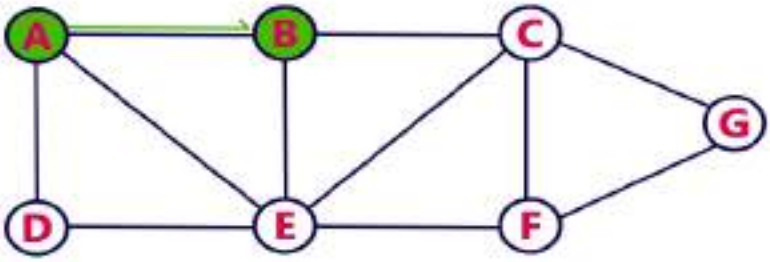
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



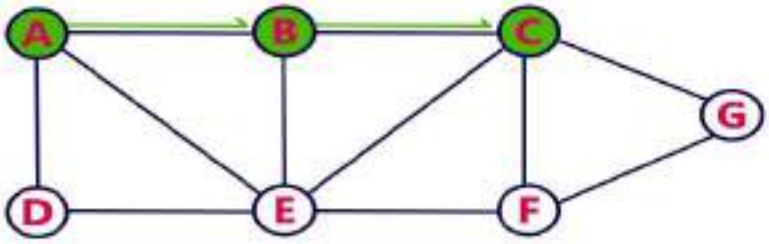
Step 2:

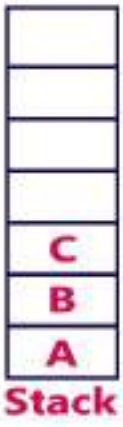
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

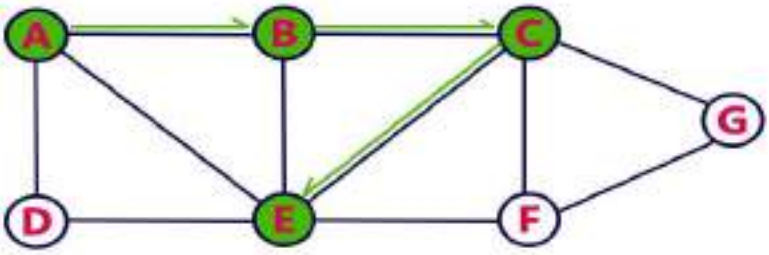
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





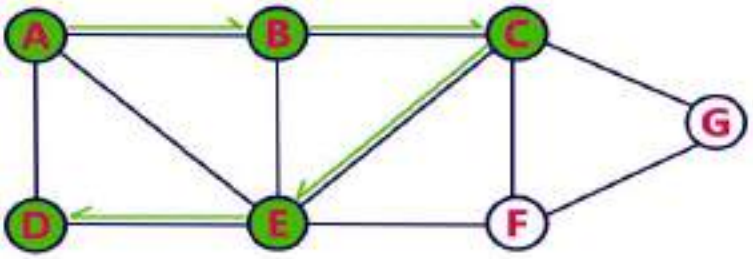
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



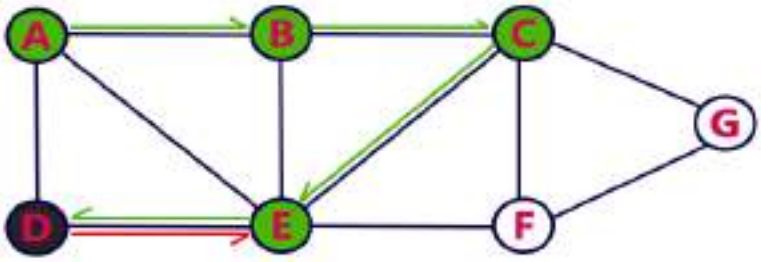
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



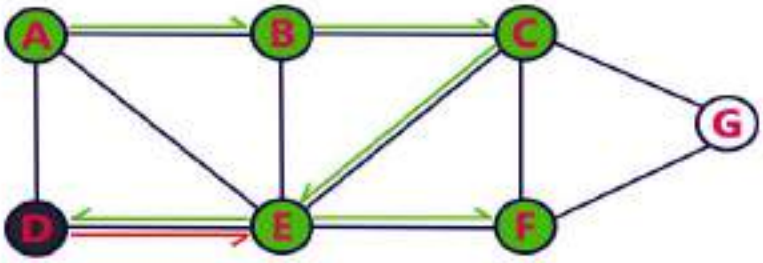
Step 6:

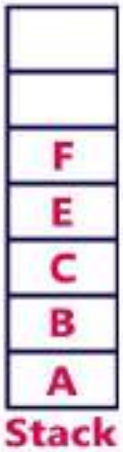
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:

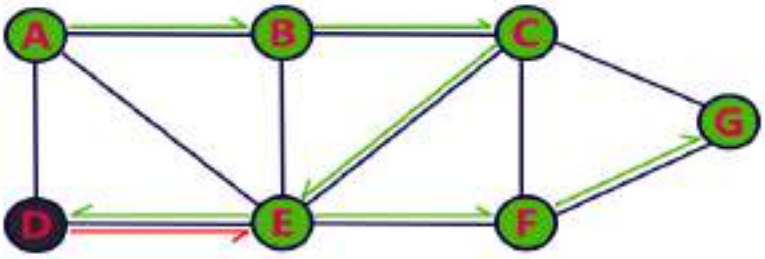
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.





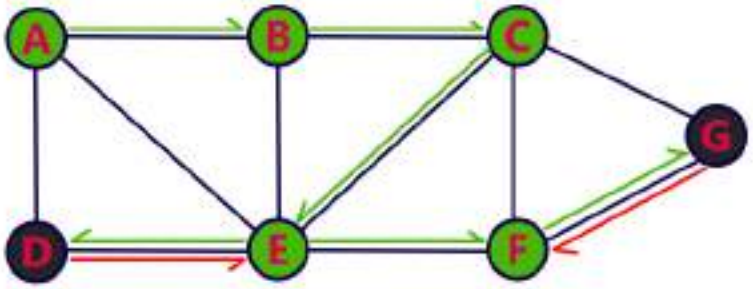
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



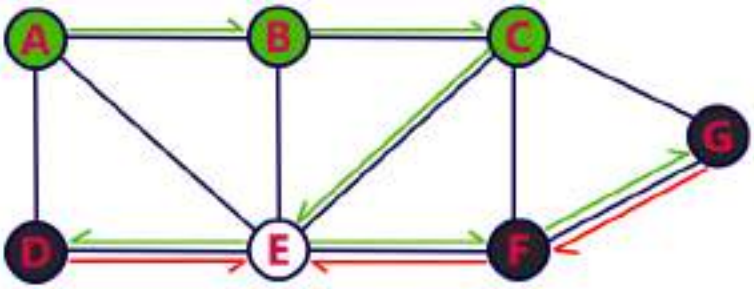
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



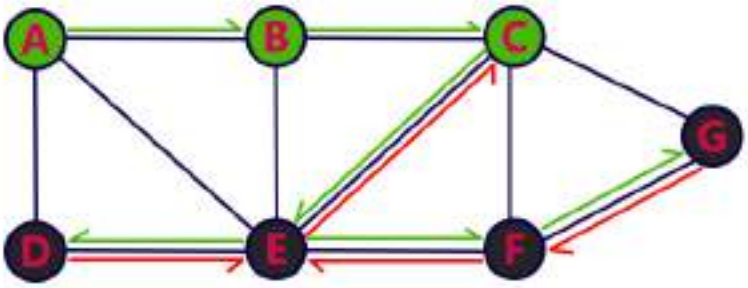
Step 10:

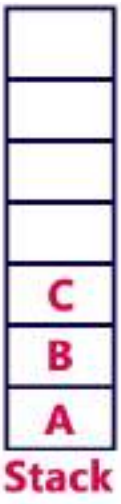
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:

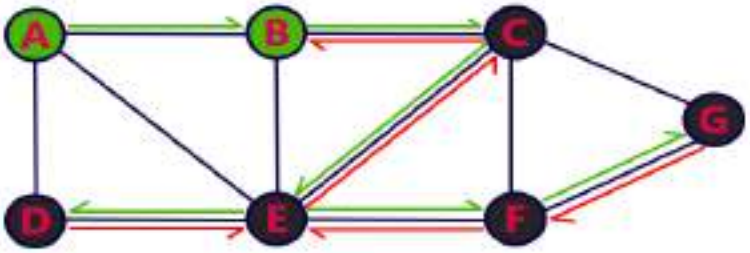
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





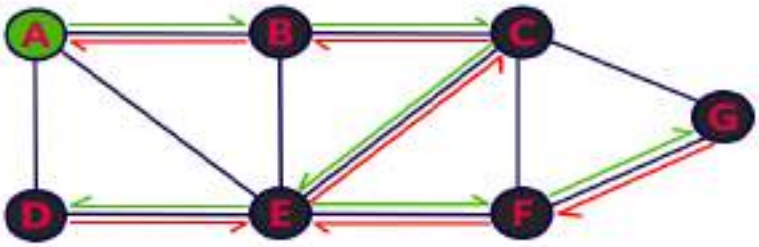
Step 12:

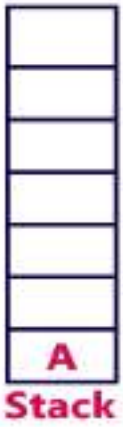
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



Step 13:

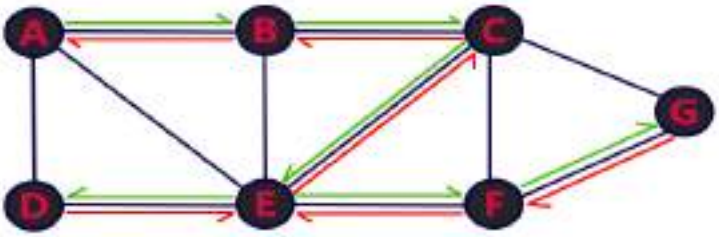
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

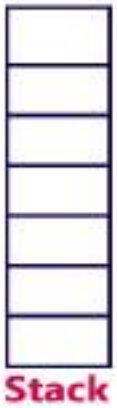




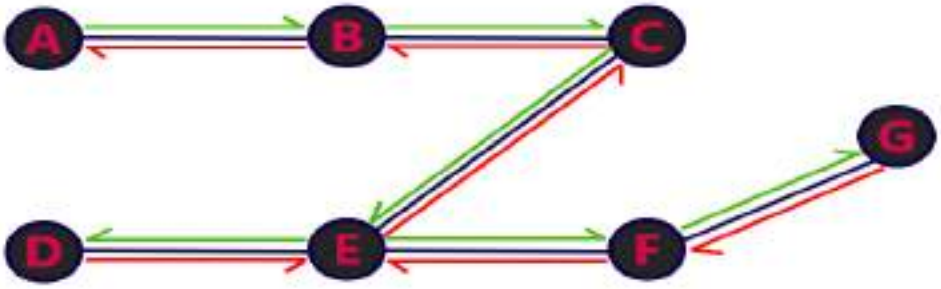
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Example for DFS:

```
#include<stdio.h>
#include<conio.h>
void dfs(int);

int a[10][10],visited[10],n,s;

void main()
{
    int i,j;
    clrscr();
    printf("\n Enter the number of Vertices :");
    scanf("%d",&n);

    printf("\n Enter graph data in adjacent matrix form :");
    for(i=1;i<=n;i++)
       for(j=1;j<=n;j++)
       scanf("%d",&a[i][j]);
    for(i=1;i<=n;i++)
    visited[i]=0;
    printf("\n Enter the Source Vertex :");
    scanf("%d",&s);
    dfs(s);
    getch();
}

void dfs(int i)
{
   int j;
   printf("%d---",i);
   visited[i]=1;
   for(j=1;j<=n;j++)
    if((!visited[j])&&(a[i][j]==1))
    dfs(j);
}
```

Output:

```
Enter the number of Vertices :5

 Enter graph data in adjacent matrix form :
0       1       1       1       0
1       0       0       1       1
1       0       0       1       0
1       1       1       1       1
0       1       0       1       0

 Enter the Source Vertex :1
1---2---4---3---5---
```

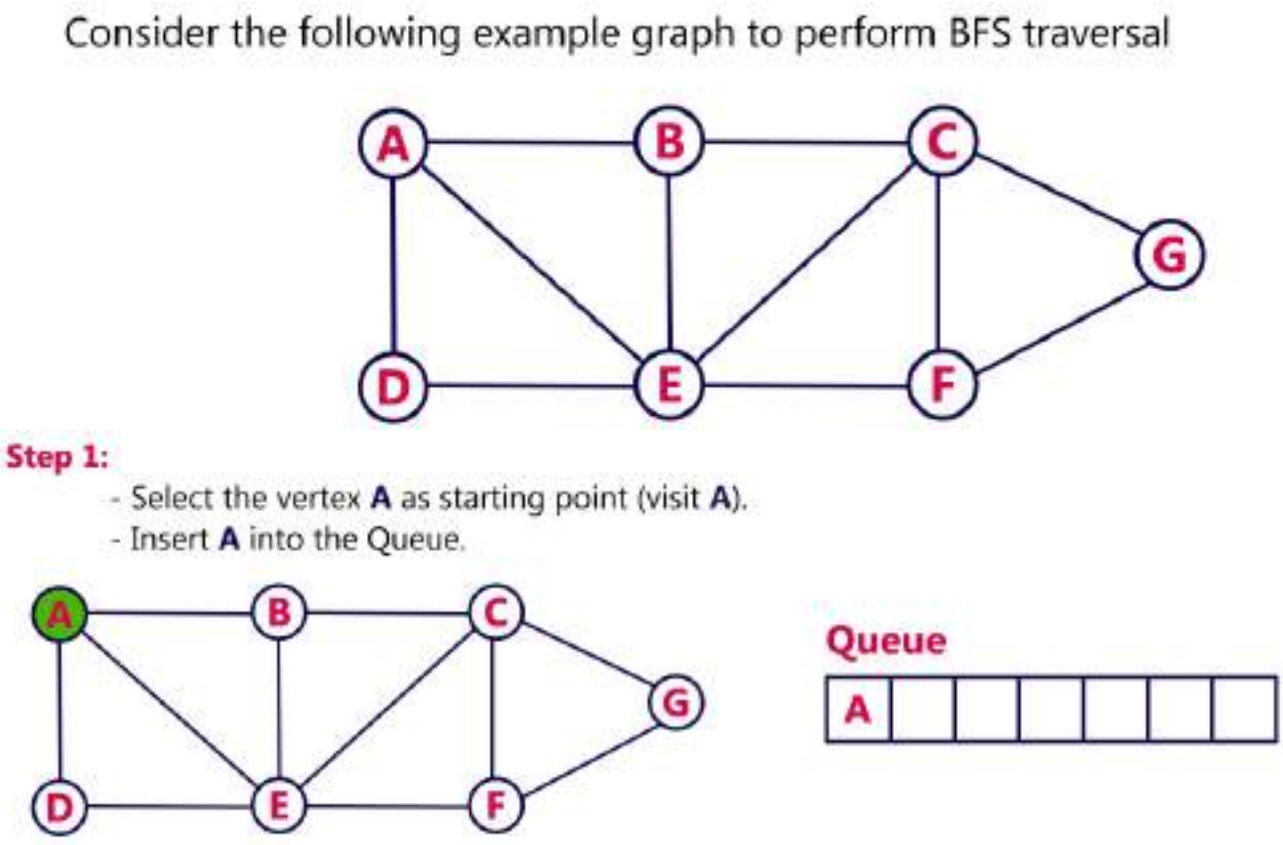
BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

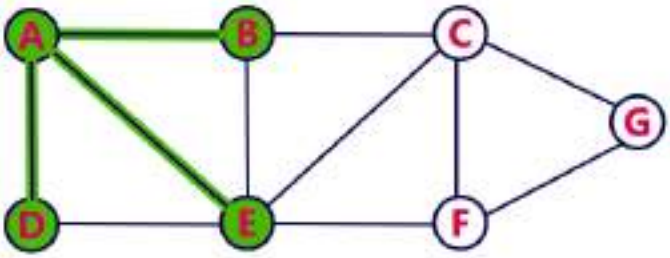
- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Example



Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

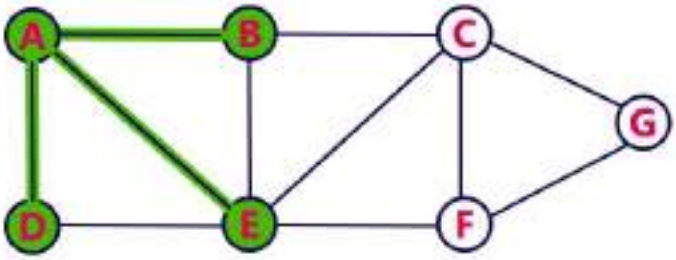


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

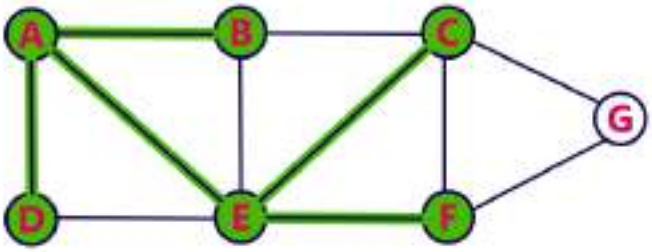


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

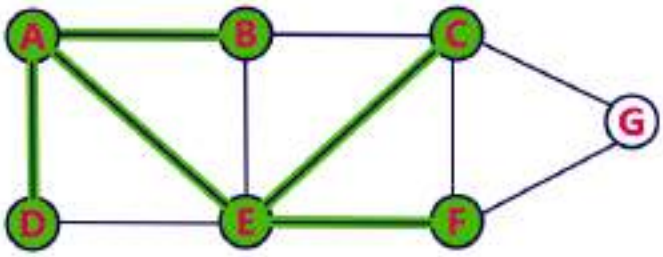


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

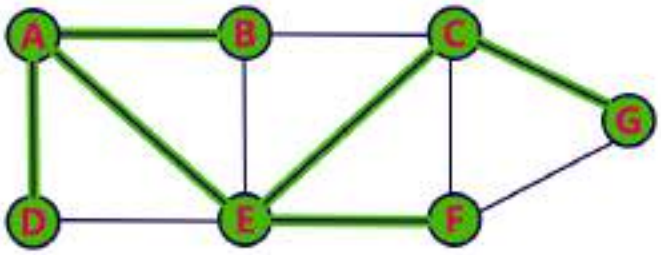


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

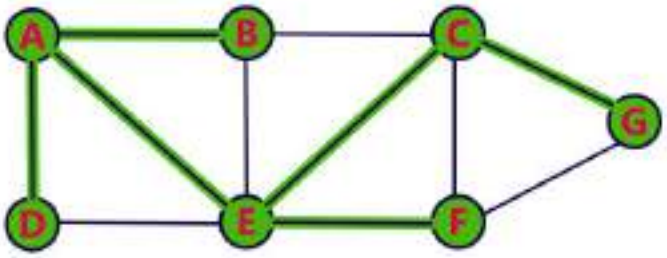


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

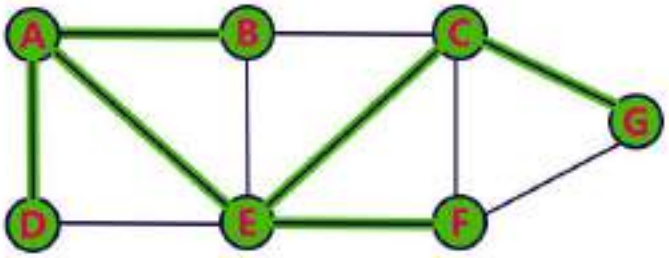


Queue

						G

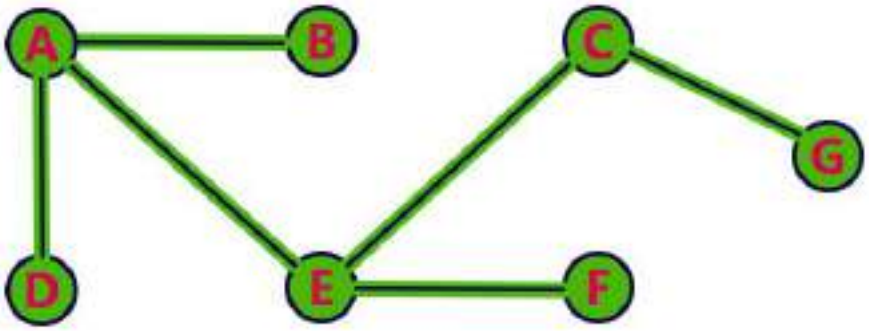
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Example for BFS:

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20];
int n,i,j,f=0,r=-1;
void bfs(int v)
{
   for(i=1;i<=n;i++)

     if(a[v][i]&&!visited[i])
     {
      q[++r]=i;
     }
     if(f<=r)
     {
     visited[q[f]]=1;
     printf("%d---",q[f]);
     bfs(q[f++]);
     }
}

void main()
{
    int v;
    clrscr();
    printf("\n Enter the number of Vertices :");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
       q[i]=0;
       visited[i]=0;
    }
    printf("\n Enter graph data in adjacent matrix form :");
    for(i=1;i<=n;i++)
       for(j=1;j<=n;j++)
       scanf("%d",&a[i][j]);
    printf("\n Enter the Source Vertex :");
    scanf("%d",&v);
    visited[v]=1;
    printf("\n BFS Traversal of vertices :\n");
    printf("\n%d--",v);
    bfs(v);
    getch();
}
```

Output:

```
 Enter the number of Vertices :5

 Enter graph data in adjacent matrix form :
0       1       1       1       0
1       0       0       1       1
1       0       0       1       0
1       1       1       1       1
0       1       0       1       0

 Enter the Source Vertex :1

 BFS Traversal of vertices :

1--2---3---4---4---5---4---5---5---
```

Differences between BFS and DFS

The following are the differences between the BFS and DFS:

	BFS	DFS
Full form	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Technique	It a vertex-based technique to find the shortest path in a graph.	It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node.
Definition	BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level.	DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes.
Data Structure	Queue data structure is used for the BFS traversal.	Stack data structure is used for the BFS traversal.
Backtracking	BFS does not use the backtracking concept.	DFS uses backtracking to traverse all the unvisited nodes.

Number of edges	BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex.	In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex.
Optimality	BFS traversal is optimal for those vertices which are to be searched closer to the source vertex.	DFS traversal is optimal for those graphs in which solutions are away from the source vertex.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Suitability for decision tree	It is not suitable for the decision tree because it requires exploring all the neighboring nodes first.	It is suitable for the decision tree. Based on the decision, it explores all the paths. When the goal is found, it stops its traversal.
Memory efficient	It is not memory efficient as it requires more memory than DFS.	It is memory efficient as it requires less memory than BFS.

Sorting

Sorting is the act of arranging data in ascending or descending order based on linear relationships among data items.

It is possible to sort by name, number, and record. Sorting reduces the difficulty of finding information: for example, looking up a friend's phone number from a telephone dictionary is relatively straightforward because the names in the phone book are alphabetical.

Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.

The records are either sorted either numerically or alphanumerically. The records are then arranged in ascending or descending order depending on the numerical value of the key. Here is an example, where the sorting of a lists of marks obtained by a student in any particular subject of a class.

Categories of Sorting
The techniques of sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

Internal Sorting: If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

External Sorting: When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

The Efficiency of Sorting Techniques

To get the amount of time required to sort an array of 'n' elements by a particular method, the normal approach is to analyze the method to find the number of comparisons (or exchanges) required by it. Most of the sorting techniques are data sensitive, and so the metrics for them depends on the order in which they appear in an input array.

Various sorting techniques are analyzed in various cases and named these cases as follows:

1. **Best Time Complexity:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.

2. **Average Time Complexity:** In the average case take all random inputs and calculate the computation time for all input and then we divide it by the total number of inputs.

3. **Worst Time Complexity:** Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is present at the last location of large data then the worst case occurs.

Hence, the result of these cases is often a formula giving the average time required for a particular sort of size 'n.' Most of the sort methods have time requirements that range from O(nlog n) to $O(n^2)$.

Types of Sorting Techniques

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Bucket Sort	Ω(n +k)	θ(n +k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n + k)
Count Sort	Ω(n +k)	θ(n +k)	O(n +k)	O(k)

Bubble Sort:

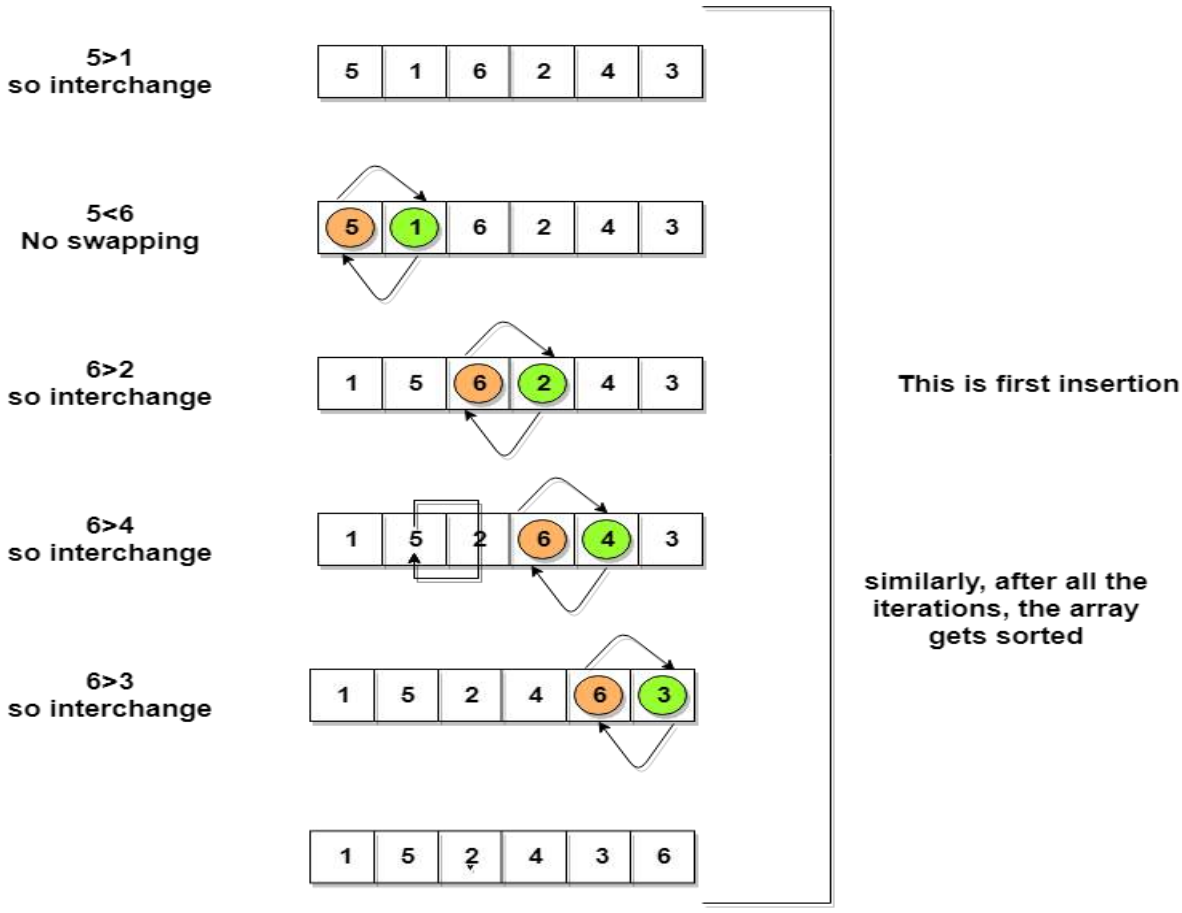
Bubble Sort Algorithm is used to arrange **N elements** in ascending order, and for that, you have to begin with 0^{th} element and compare it with the first element. If the 0^{th} element is found greater than the 1^{st} element, then the swapping operation will be performed, i.e., the two values will get interchanged. In this way, all the elements of the array get compared.

Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values `{5, 1, 6, 2, 4, 3}`



So as we can see in the representation above, after the first iteration, `6` is placed at the last index, which is the correct position for it.

Similarly after the second iteration, `5` will be at the second last index, and so on.

Example:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>

void bubblesort(int *,int);
void main()
{
    int *a,n,i;
    clrscr();
    printf("\n Enter the size of the array : ");
    scanf("%d",&n);
    a=(int *)calloc(n,sizeof(int));
    printf("\n Enter the Elements :");
    for(i=0;i<n;i++)
          scanf("%d",&a[i]);
          printf("\n Elements before sorting :\n\n");
          for(i=0;i<n;i++)
            printf("\t %d",a[i]);
            bubblesort(a,n);
            printf("\n\n Elements after sorting :\n\n");
            for(i=0;i<n;i++)
            printf("\t %d",a[i]);
   getch();
}

void bubblesort(int *a,int n)
{
   int i,j,t;
   for(i=0;i<n;i++)
   {
          for(j=0;j<n-i-1;j++)
          {
            if(a[j]>a[j+1])
            {
              t=a[j];
              a[j]=a[j+1];
              a[j+1]=t;
            }      }      }     }
```

Output:

```
        Enter the size of the array : 5
        Enter the Elements :1 5 3 4 2
        Elements before sorting :

         1      5      3      4      2

        Elements after sorting :

         1      2      3      4      5
```

Selection Sort in Data Structure

Selection sort is another sorting technique in which we find the minimum element in every iteration and place it in the array beginning from the first index. Thus, a selection sort also gets divided into a sorted and unsorted subarray.

What is Selection Sort?

Selection sort has the following characteristics:

- Comparison-based sorting algorithm.
- In place sorting technique.
- An unstable sorting algorithm i.e. does not preserve the order of duplicate elements.
- Time complexity is $O(n^2)$.
- Selection sort is the best algorithm when swapping is a costly operation.

In every iteration, the selection sort algorithm selects the smallest element from the whole array and swaps it with the leftmost element of the unsorted sub-array.

Steps involved in Selection Sort

1. Find the smallest element in the array and swap it with the first element of the array i.e. a[0].
2. The elements left for sorting are n-1 so far. Find the smallest element in the array from index 1 to n-1 i.e. a[1] to a[n-1] and swap it with a[1].
3. Continue this process for all the elements in the array until we get a sorted list.

Working of Selection Sort

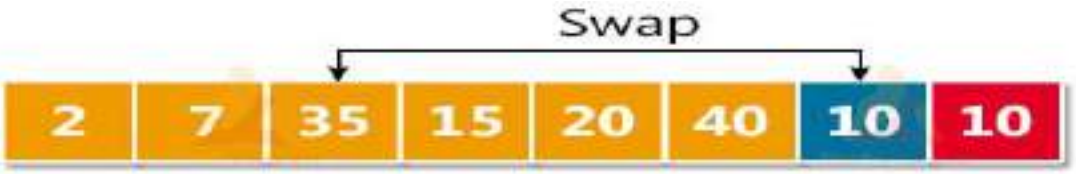
Let us understand the working of the selection sort algorithm with the help of the following example:

Consider an array having elements: 40, 10, 35, 15, 20, 2, 10, 7 as shown:

Iteration 1:

Iteration 2:

Iteration 3:



Iteration 4:

Iteration 5:

Iteration 6:

Iteration 7:

Iteration 8:

Thus, the final array after selection sort is:

In this array, we can clearly see that the order of duplicate elements has changed. Thus, selection sort is an unstable algorithm.

Example:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>

void main()
{
    int a[20],n,i,j,t;
    clrscr();
    printf("\n Enter the size of the array : ");
    scanf("%d",&n);
    printf("\n Enter the Elements :");
    for(i=0;i<n;i++)
          scanf("%d",&a[i]);
          printf("\n\t\t Selection Sort");
          printf("\n\t\t **************\n");
          printf("\n Elements before sorting :\n\n");
          for(i=0;i<n;i++)
            printf("\t %d",a[i]);

            for(i=0;i<=n-2;i++)
            {
              for(j=i+1;j<n;j++)
          {
            if(a[i]>a[j])
            {
              t=a[i];
              a[i]=a[j];
              a[j]=t;
            }
          }
     }

   printf("\n\n Elements after sorting:\n\n");
   for(i=0;i<n;i++)
   printf("\t%d",a[i]);
   getch();
}
```

Output:

```
        Enter the size of the array : 6
        Enter the Elements :   4    8   10   6   2   0

                  Selection Sort
                  **************
         Elements before sorting :

         4       8       10      6       2       0

         Elements after sorting:

        0       2       4       6       8       10
```

Insertion Sort in Data Structure

Insertion sort is a comparison-based sorting algorithm mostly suited for small datasets. Insertion sort has wide practical implementations. For example, sorting of play cards is an implementation of insertion sort. Arranging randomly arranged answer sheets in the ascending order of roll numbers is also an example of insertion sort.

What is Insertion Sort?

In insertion sort, there are two lists in the array at every point: A partially sorted list and an unsorted list. Thus, we virtually split the array into sorted and unsorted parts.

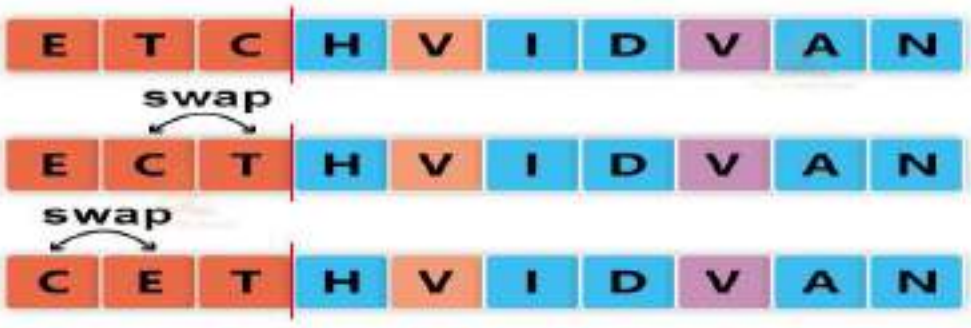
Every time we pick up an element from the unsorted array and place it in the partially sorted array. Thus, we insert the elements from an unsorted array at their correct position in the sorted array. That is why the name **Insertion.** It is not suitable for very large datasets.
Insertion sort is a stable element. The order of duplicate elements is preserved after the sorting is performed.

How Insertion Sort Works?

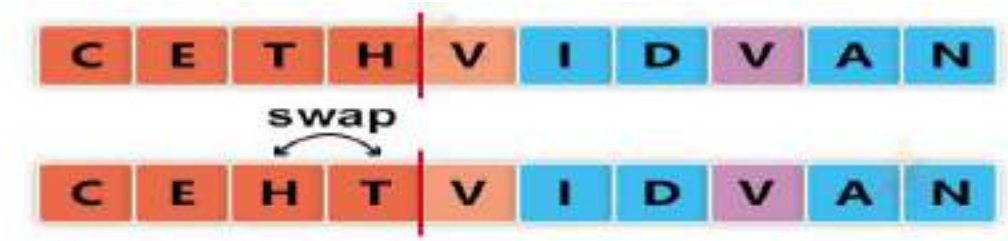
In each iteration, we will bring one element from the unsorted list to the sorted list and place it at its correct position.

Iteration 1:

Iteration 2:

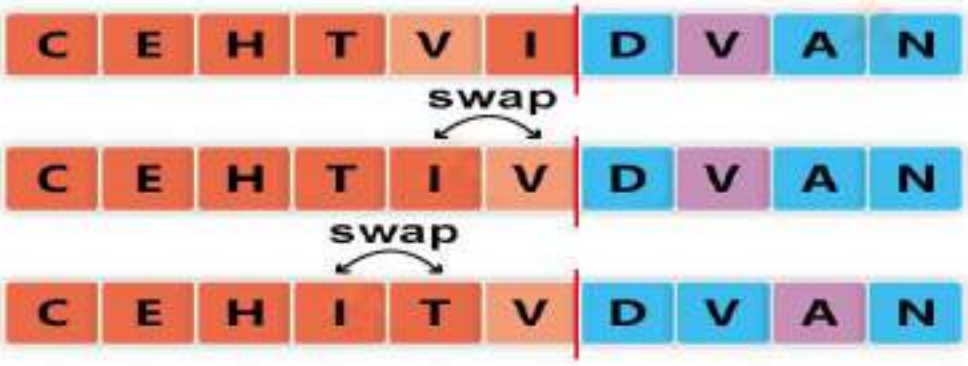


Iteration 3:



Iteration 4:

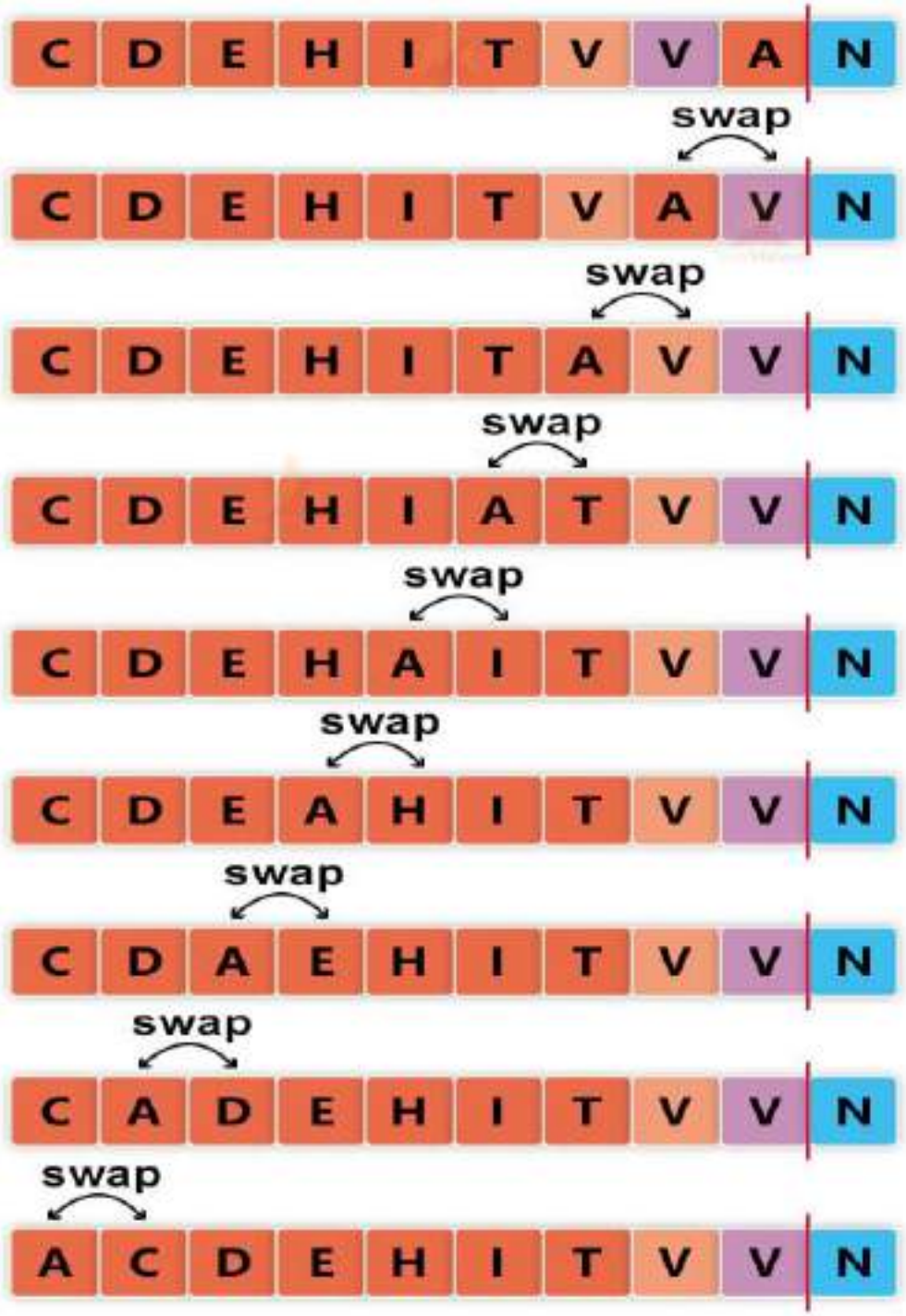
Iteration 5:



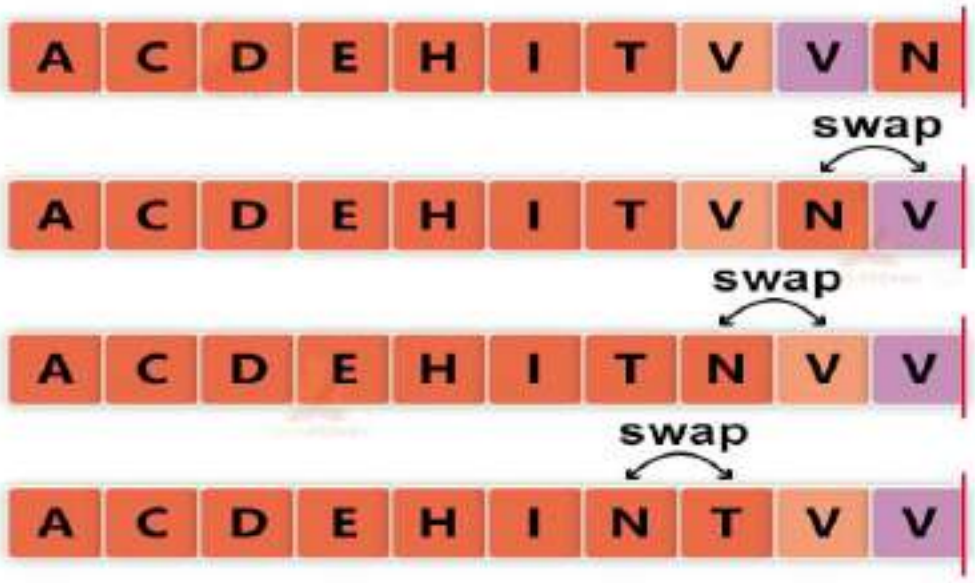
Iteration 6:

Iteration 7:

Iteration 8:



Iteration 9:



Finally, after the 9th iteration, we have got the sorted array.

We need to note here that the insertion sort is a stable sorting algorithm. Thus, the order of duplicates is preserved. Here, we have two duplicate elements- 'V'. thus, the order of both the 'V's will remain the same before and after sorting.

Example:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>

void main()
{
    int a[20],n,i,j,k,t;
    clrscr();
    printf("\n Enter the size of the array : ");
    scanf("%d",&n);
    printf("\n Enter the Elements :");
    for(i=0;i<n;i++)
          scanf("%d",&a[i]);
          printf("\n\t\t Insertion Sort");
          printf("\n\t\t **************\n");
          printf("\n Elements before sorting :\n\n");
          for(i=0;i<n;i++)
            printf("\t %d",a[i]);

            for(i=1;i<n;i++)
            {
              for(j=0;j<i;j++)
          {
            if(a[j]>a[i])
```

```
            {
              t=a[j];
              a[j]=a[i];
              for(k=i;k>j;k--)
              a[k]=a[k-1];
              a[k+1]=t;
            }
          }
    }

   printf("\n\n Elements after sorting:\n\n");
   for(i=0;i<n;i++)
   printf("\t%d",a[i]);
   getch();   }
```

Output:

Enter the size of the array :
8

Enter the Elements :
1 4 7 9 2 3 5 8

Insertion Sort
****

Elements before sorting :

1 4 7 9 2 3 5 8

Elements after sorting:

1 2 3 4 5 7 8 9

Heap Sort

In this article, we will discuss the Heapsort Algorithm. Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

Before knowing more about the heap sort, let's first see a brief description of **Heap.**

What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

What is heap sort?

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

Now, let's see the algorithm of heap sort.

Algorithm

1. HeapSort(arr)
2. BuildMaxHeap(arr)
3. for i = length(arr) to 2
4. swap arr[1] with arr[i]
5. heap_size[arr] = heap_size[arr] ? 1
6. MaxHeapify(arr,1)
7. End

BuildMaxHeap(arr)

1. BuildMaxHeap(arr)
2. heap_size(arr) = length(arr)
3. for i = length(arr)/2 to 1
4. MaxHeapify(arr,i)
5. End

MaxHeapify(arr,i)

1. MaxHeapify(arr,i)
2. L = left(i)
3. R = right(i)
4. if L ? heap_size[arr] and arr[L] > arr[i]
5. largest = L
6. else
7. largest = i
8. if R ? heap_size[arr] and arr[R] > arr[largest]
9. largest = R

10. if largest != i
11. swap arr[i] with arr[largest]
12. MaxHeapify(arr,largest)
13. End

Working of Heap sort Algorithm

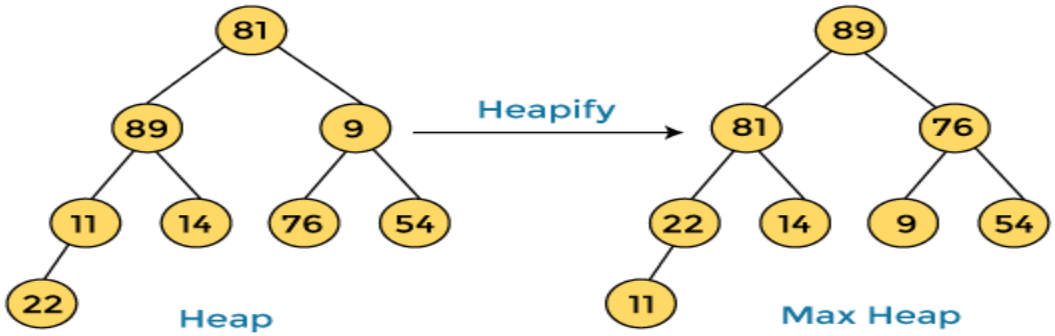
Now, let's see the working of the Heapsort Algorithm.

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

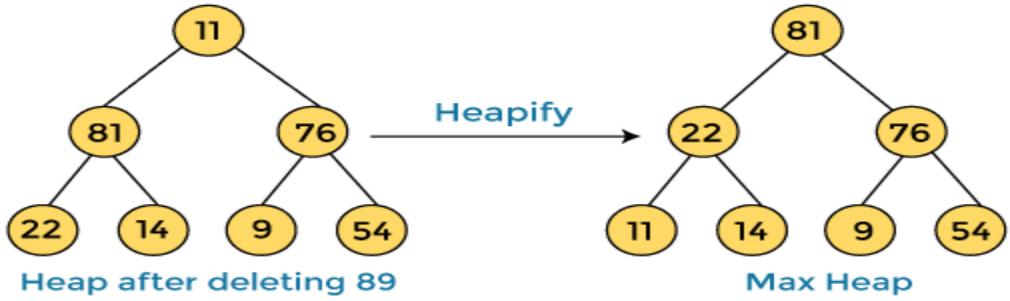
Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

First, we have to construct a heap from the given array and convert it into max heap.



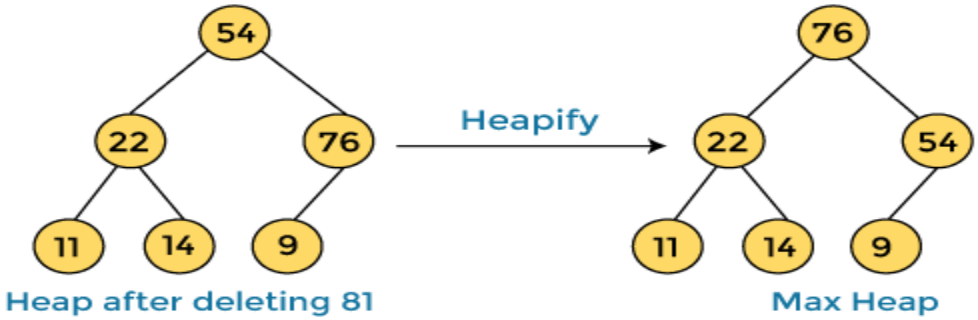
After converting the given heap into max heap, the array elements are -

Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



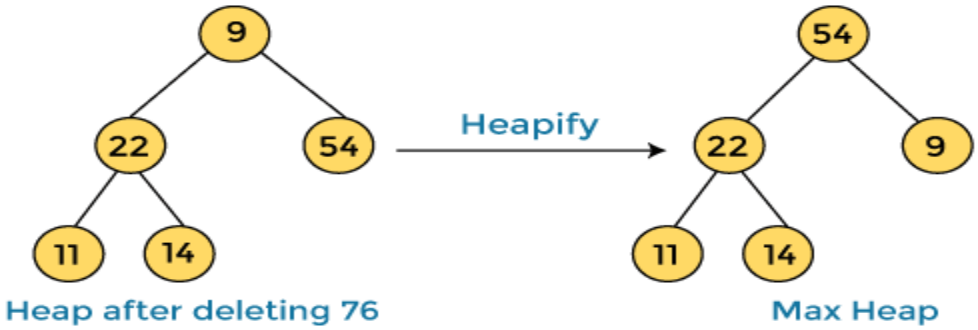
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



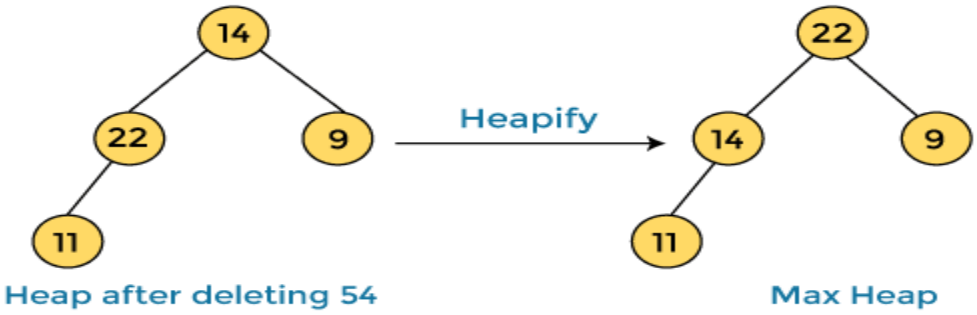
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.

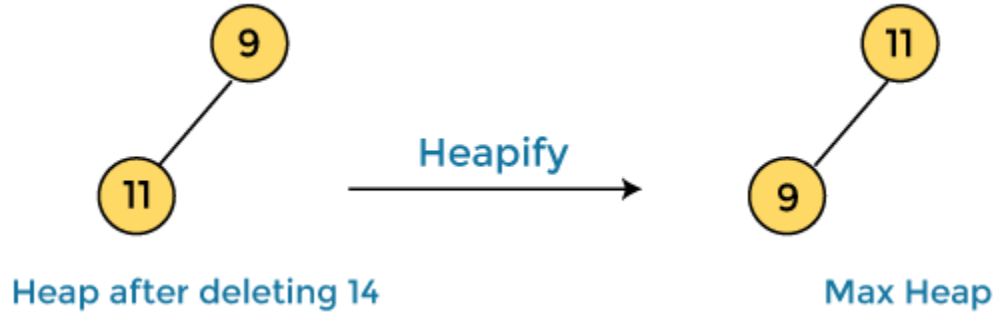


After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

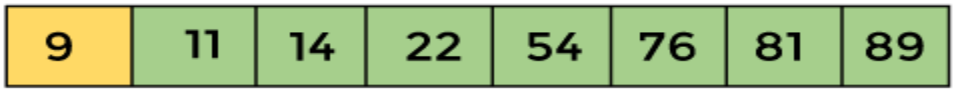
In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

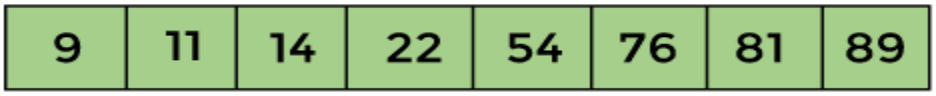
In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity

Space Complexity	O(1)
Stable	N0

- The space complexity of Heap sort is O(1).

Implementation of Heapsort

Now, let's see the programs of Heap sort in different programming languages.

Program: Write a program to implement heap sort in C language.

```
1. #include <stdio.h>
2. /* function to heapify a subtree. Here 'i' is the
3. index of root node in array a[], and 'n' is the size of heap. */
4. void heapify(int a[], int n, int i)
5. {
6.     int largest = i; // Initialize largest as root
7.     int left = 2 * i + 1; // left child
8.     int right = 2 * i + 2; // right child
9.     // If left child is larger than root
```

```
10.     if (left < n && a[left] > a[largest])
11.         largest = left;
12.     // If right child is larger than root
13.     if (right < n && a[right] > a[largest])
14.         largest = right;
15.     // If root is not largest
16.     if (largest != i) {
17.         // swap a[i] with a[largest]
18.         int temp = a[i];
19.         a[i] = a[largest];
20.         a[largest] = temp;
21.
22.         heapify(a, n, largest);
23.     }
24. }
25. /*Function to implement the heap sort*/
26. void heapSort(int a[], int n)
27. {
28.     for (int i = n / 2 - 1; i >= 0; i--)
29.         heapify(a, n, i);
30.     // One by one extract an element from heap
31.     for (int i = n - 1; i >= 0; i--) {
32.         /* Move current root element to end*/
33.         // swap a[0] with a[i]
34.         int temp = a[0];
35.         a[0] = a[i];
36.         a[i] = temp;
37.
38.         heapify(a, i, 0);
39.     }
40. }
41. /* function to print the array elements */
42. void printArr(int arr[], int n)
43. {
44.     for (int i = 0; i < n; ++i)
45.     {
46.         printf("%d", arr[i]);
47.         printf(" ");
```

```
48.     }
49.
50. }
51. int main()
52. {
53.     int a[] = {48, 10, 23, 43, 28, 26, 1};
54.     int n = sizeof(a) / sizeof(a[0]);
55.     printf("Before sorting array elements are - \n");
56.     printArr(a, n);
57.     heapSort(a, n);
58.     printf("\nAfter sorting array elements are - \n");
59.     printArr(a, n);
60.     return 0;
61. }
```

Output

```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

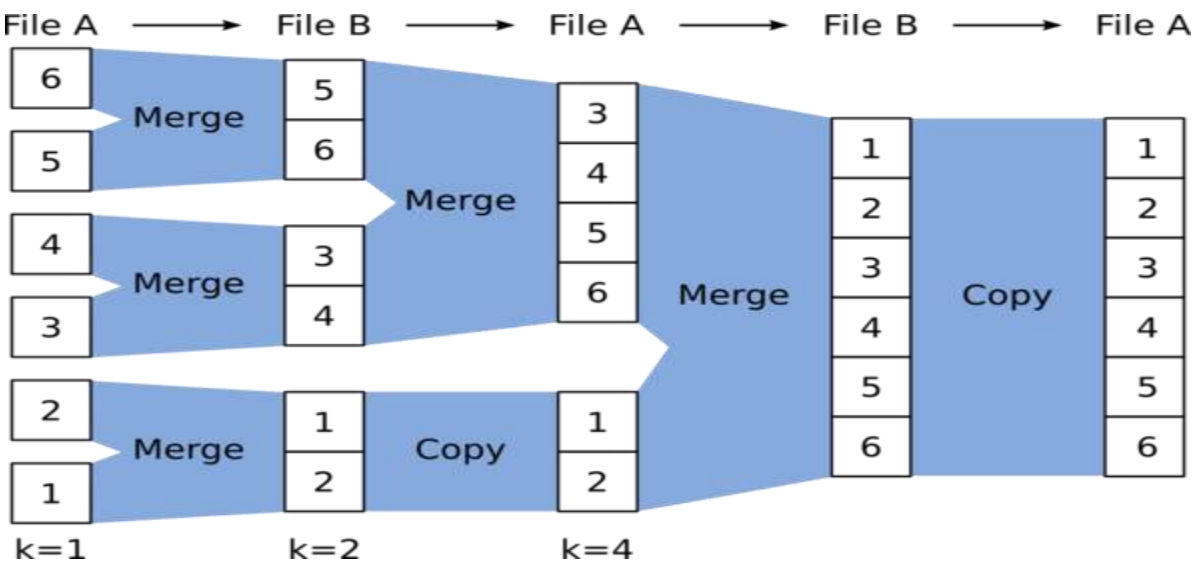
External sorting

An external sorting algorithm is an algorithm that can handle massive amounts of information. Users utilize it when the data that needs sorting doesn't fit into a computer's primary memory (usually the random access memory [RAM]). In such a case, you must place the information in an external memory device (usually a hard disk drive [HDD]).

Think of it this way. We should drink at least eight glasses of water a day to stay healthy. Let's say you use a 1-liter tumbler while at work, and your goal is to drink eight glasses in eight hours. You know that 1 liter is equal to four glasses. To reach your goal, you'll need to refill your tumbler once while in the office to reach your goal. In this scenario, the tumbler represents the computer's RAM, which can only "process" four glasses of water (or data) per batch, and the HDD is the water source, which can provide massive amounts of "data," depending on your goal.

How Does an External Sorting Algorithm Work?

The following diagram shows how a simple external sorting algorithm works:



You can sort a vast mass of data made up of six blocks into three smaller chunks, that is if the computer can only handle two blocks at a time. Each time the two-block chunks are processed, they get stored in a temporary file. Once all of the smaller chunks are processed, they are merged back for storing as an entire massive data block in an external storage device like an HDD.

Examples:

1. Merge sort
2. Tape sort
3. Polyphase sort
4. External radix
5. External merge

Merge Sort

Merge sort is another sorting technique and has an algorithm that has a reasonably proficient space-time complexity - O(n log n) and is quite trivial to apply. This algorithm is based on splitting a list, into two comparable sized lists, i.e., left and right and then sorting each list and then merging the two sorted lists back together as one.

Merge sort can be done in two types both having similar logic and way of implementation. These are:

- Top down implementation
- Bottom up implementation

Below given figure shows how Merge Sort works:

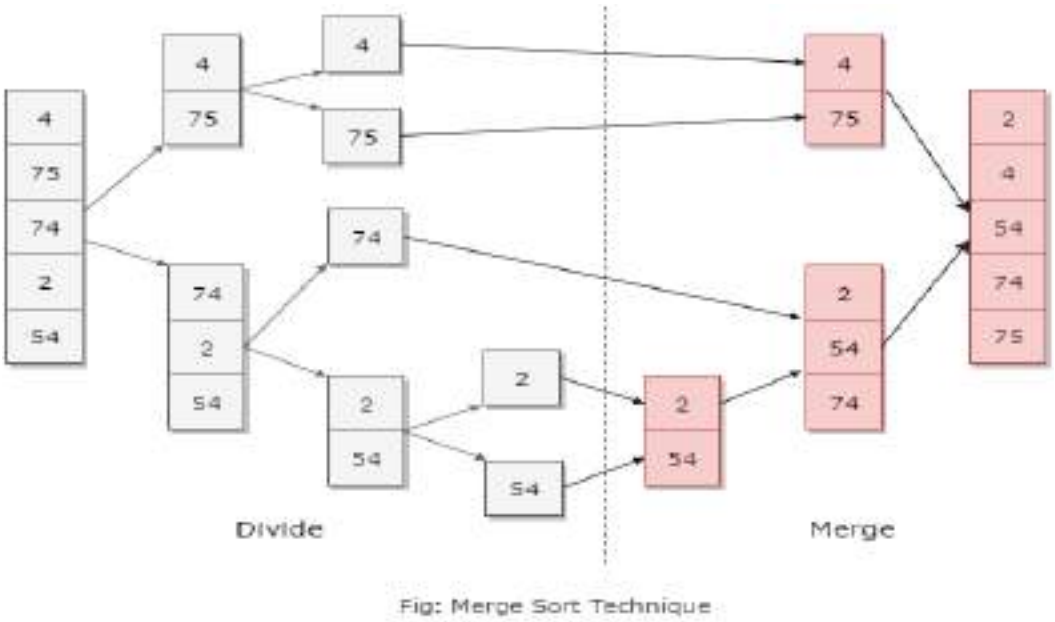


Fig: Merge Sort Technique

Algorithm for Merge Sort

1. algorithm Merge_Sort(list)
2. Pre: list 6= fi
3. Post: list has been sorted into values of ascending order
4. if list.Count = 1 // when already sorted
5. return list
6. end if
7. m <- list.Count = 2
8. left <- list(m)
9. right <- list(list.Count - m)
10. for i <- 0 to left.Count - 1
11. left[i] <- list[i]
12. end for
13. for i <- 0 to right.Count -1
14. right[i] <- list[i]
15. end for
16. left <- Merg_Sort(left)
17. right <- Merge_Sort(right)
18. return MergeOrdered(left, right)
19. end Merge_Sort

It is to be noted that, Merge sort uses the concept of divide and conquer technique that was invented by John von Neumann in the year 1945.

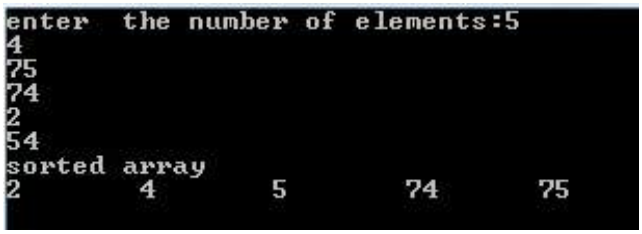
Example:

```
#include <iostream>
using namespace std;

void merge(int *,int, int , int );
void mergesort(int *a, int low, int high)
{
```

```
    int mid;
    if (low < high)
    {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,high,mid);
    }
    return;
}
// Merge sort concepts starts here
void merge(int *a, int low, int high, int mid)
{
    int i, j, k, c[50];
    i = low;
    k = low;
    j = mid + 1;
    while (i <= mid && j <= high)
    {
        if (a[i] < a[j])
        {
            c[k] = a[i];
            k++;
            i++;
        }
        else
        {
            c[k] = a[j];
            k++;
            j++;
        }
    }
    while (i <= mid)
    {
        c[k] = a[i];
        k++;
        i++;
    }
    while (j <= high)
    {
        c[k] = a[j];
        k++;
        j++;
    }
    for (i = low; i < k; i++)
    {
        a[i] = c[i];
    }
}
// from main mergesort function gets called
int main()
```

```
{
   int a[30], i, b[30];
   cout<<"enter  the number of elements:";
   for (i = 0; i <= 5; i++) { cin>>a[i];
   }
   mergesort(a, 0, 4);
   cout<<"sorted array\n";
   for (i = 0; i < 5; i++)
   {
      cout<<a[i]<<"\t";
   }
   cout<<"enter  the number of elements:";
   for (i = 0; i < 5; i++) { cin>>b[i];
   }
   mergesort(b, 0, 4);
   cout<<"sorted array:\n";
   for (i = 0; i < 5; i++)
   {
      cout<<b[i]<<"\t";
   }
   getch();
}
```

Output:



UNIT – V

Pattern Matching and Tries : Pattern Matching algorithms – Brute force, The Boyer-Moore algorithm, The Knuth-Morris-Pratt algorithm, Standard Tries, Compresses Tries, Suffix Tries.

What is pattern matching?

In computer science, pattern matching is the act of checking a given sequence of tokens for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact: **"either it will or will not be a match".**

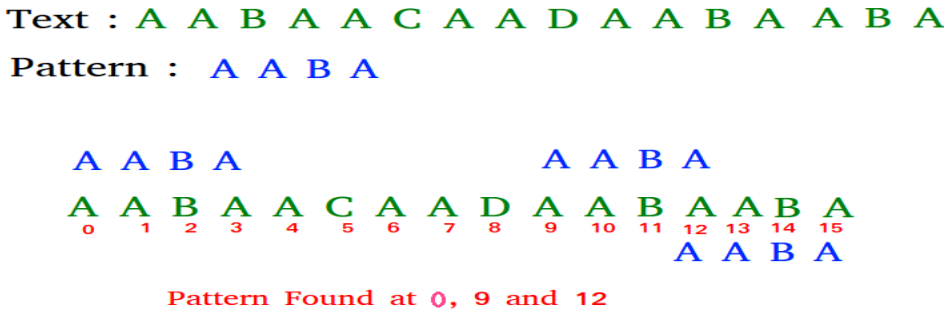
The patterns generally have the form of either **sequences or tree structures**. Uses of pattern matching include outputting the locations (if any) of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence **(i.e., search(find) and replace).**

Sequence patterns (e.g., a text string) are often described using regular expressions and matched using techniques such as **backtracking.**

Tree patterns are used in some programming languages as a general tool to process data based on its structure, e.g. C#, F#, Haskell, ML, Python, Ruby, Rust, Scala, Swift and the symbolic mathematics language Matematica have special syntax for expressing tree patterns and a language construct for conditional execution and value retrieval based on it.

Often it is possible to give alternative patterns that are tried one by one, which yields a powerful conditional programming construct. Pattern matching sometimes includes support for guards.

Pattern matching is used to determine whether source files of high-level languages are syntactically correct. It is also used to find and replace a matching pattern in a text or code with another text/code. Any application that supports search functionality uses pattern matching in one way or another.



What is pattern matching in regular expression?

A regular expression (sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. “find and replace”- like operations

Regular expressions allow you to select specific strings from a set of character strings. The use of regular expressions is generally associated with text processing.

Regular expressions can represent a wide variety of possible strings. While many regular expressions can be interpreted differently depending on the current locale, globalization features provide for contextual invariance across locales.

Pattern matching is used by the shell commands such as the ls command, whereas regular expressions are used to search for strings of text in a file by using commands, such as the grep command.

What is a pattern matching algorithm?

Pattern matching algorithms are the algorithms that are used to figure out whether a specific string pattern occurs in a string text. Two of the most widely used pattern matching algorithms are the Naive Algorithm for pattern matching and the pattern matching algorithm using finite automata.

However, these are not the only two pattern matching algorithms available. There are several solutions to the pattern matching problem that have been proposed. These pattern matching algorithms have been classified into online and offline solutions.

Online solutions happen to be dynamic and they do not need a priori knowledge of the pattern database, preprocessing could be carried out on the pattern. Online solutions generally have two phases: the **preprocessing phase** of the pattern, and the **search phase** of the pattern in the pattern database. While the preprocessing phase is going on, a data structure is built which tends to be proportional to the length of the pattern and details vary for various algorithms. The search phase makes use of the data structure and attempts to quickly figure out whether the pattern appears in the text. This phase is generally based on four separate approaches. These include the classical approach, suffix automata approach, bit-parallelism approach, and the hashing approach.

Offline solutions are based on preprocessing activities that are carried out on the patterns database to prepare for the matching process.

How does pattern matching work?

The Pattern Matching process looks for a specified pattern within a user-defined value. You can use Pattern Matching to recognize social security numbers, telephone numbers, ZIP codes, or any other information that follows a specific pattern. It is also useful for looking for information that follows leading text: for example, looking for "Name:" and then extracting the text that comes after it.

Another use is in reprocessing documents; for example, extracting a piece of information such as the date from a file name and entering it in a field. Pattern Matching works by "reading" through text strings to match patterns that are defined using **Pattern Matching Expressions**, also known as Regular Expressions. Pattern Matching can be used in Identification as well as in Pre-Classification Processing, Page Processing, or Storage Processing.

What is the importance of pattern matching?

Pattern matching has many useful applications, such as:

- **Natural Language Processing**: Applications like spelling and grammar checkers, spam detectors, translation, and sentiment analysis tools heavily depend on pattern recognition methods. Regular Expressions are helpful in identifying complex text patterns for natural language processing.
- **Image processing, segmentation, and analysis:** Pattern matching is used to give human recognition intelligence to machines which are required in image processing.
- **Computer vision**: Pattern matching is used to extract meaningful features from given image/video samples and is used in computer vision for various
- **Applications like biological and biomedical imaging:** Tumor identification is a classic example.

- **Seismic analysis**: The pattern matching approach is used for the discovery, imaging, and interpretation of temporal patterns in seismic array recordings. Statistical pattern recognition is implemented and used in different types of seismic analysis models.
- **Radar signal classification/analysis:** Pattern matching and signal processing methods are used in various applications of radar signal classifications like landmine detection and identification.
- **Speech recognition**: The greatest success in speech recognition has been obtained using pattern matching paradigms by treating each word as a unit.
- **Fingerprint identification**: Many fingerprint recognition methods are in use to perform fingerprint matching out of which pattern matching approaches is widely used.

String Searching (Pattern Matching)

- The object of string searching is to find the locationof a specific text pattern within a larger body of text(e.g., a sentence, a paragraph, a book, etc.).
- As with most algorithms, the main considerationsfor string searching are speed and efficiency.
- There are a number of string searching (Pattern Matching) algorithms inexistence today,
  - **Brute Force Algorithm**
  - **Rabin-Karp.**
  - **Boyer Moore Algorithm**
  - **Naïve Algorithm**
  - **Finite Automata**
  - **Knuth- Morris – Pratt Algorithm, etc….**

Brute Force Algorithm

- A pattern is compared to some text to check if the text contains the pattern or not. The Brute Force method compares the first character in the pattern to each character in the text.
- If a match is not made, the pattern is moved forwards one space along the text.
- If a match is found, then the pattern does not move and we compare the next character in the pattern to the next character in the text.
- The Brute Force algorithm compares the pattern tothe text, one character at a time, until unmatching characters are found:

Text : *T*WO ROADS DIVERGED *Pattern : R*OADS	IN	A	YELLOW	WOOD
T*W*O ROADS DIVERGED *R*OADS	IN	A	YELLOW	WOOD
TW*O* ROADS DIVERGED *R*OADS	IN	A	YELLOW	WOOD
TWO ROADS DIVERGED *R*OADS	IN	A	YELLOW	WOOD
TWO ***ROADS*** DIVERGED ***ROADS***	IN	A	YELLOW	WOOD

- Compared characters are italicized.
- Correct matches are in boldface type.

The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching theend of the text.

Brute Force Pseudo-Code

Here's the pseudo-code

```
do
    if (text letter == pattern letter) compare next letter of pattern to next
        letter of text

    else
        move pattern down text by one letter
while (entire pattern found or end of text)
```

```
tettththeheehhtehthethhthehehtht
the

tettththeheehhtehthethhthehehtht
 the

tettththeheehhtehthethhthehehtht
   the

tetththeheehhtehthethhthehehtht
     the

tetththeheehhtehthethhthehehtht
      the

tetththeheehhtehthethhthehehtht
       the
```

Brute Force-Complexity

Worst case: compares pattern to each substring oftext of length M.

For example, M=5.

- Given a pattern M characters in length, and a text Ncharacters in length...

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**

2) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**

3) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**

4) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**

5) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**

....

N) AAAAAAAAAAAAAAAAAAAAA*AAAAH*

AAAAH **5 comparisons made**

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)

Best case if pattern found: Finds pattern in first Mpositions of text.

For example, M=5.
Given a pattern M characters in length, and a text Ncharacters in length...

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAA **5 comparisons made**

- Total number of comparisons: M
- Best case time complexity: O(M)

Best case if pattern not found: Always mismatchon first character.

For example, M=5.

- Given a pattern M characters in length, and a text Ncharacters in length...

1) *A*AAAAAAAAAAAAAAAAAAAAAAAAAH*O*OOOH **1 comparison made**

2) A*A*AAAAAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

3) AA*A*AAAAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

4) AAA*A*AAAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

5) AAAA*A*AAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

...

N) AAAAAAAAAAAAAAAAAAAAAAA*A*AAAH

*O*OOOH **1 comparison made**

- Total number of comparisons: N
- Best case time complexity: O(N)

Boyer Moore Algorithm for Pattern Searching

Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file, browser, or database, pattern searching algorithms are used to show the search results.

Given a text txt[0..n-1] and a pattern pat[0..m-1] where n is the length of the text and m is the length of the pattern, write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.
Examples:

Input: txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output: Pattern found at index 10

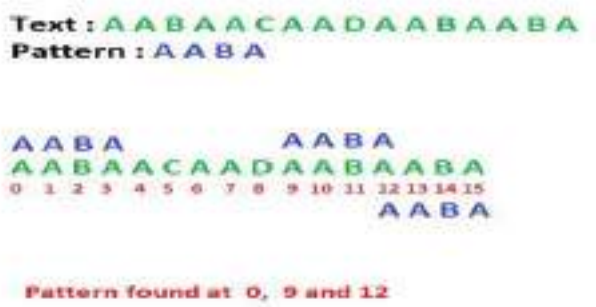
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



In this post, we will discuss the Boyer Moore pattern searching algorithm. Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern.
Boyer Moore is a combination of the following two approaches.

1. **Bad Character Heuristic**
2. **Good Suffix Heuristic**

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one.

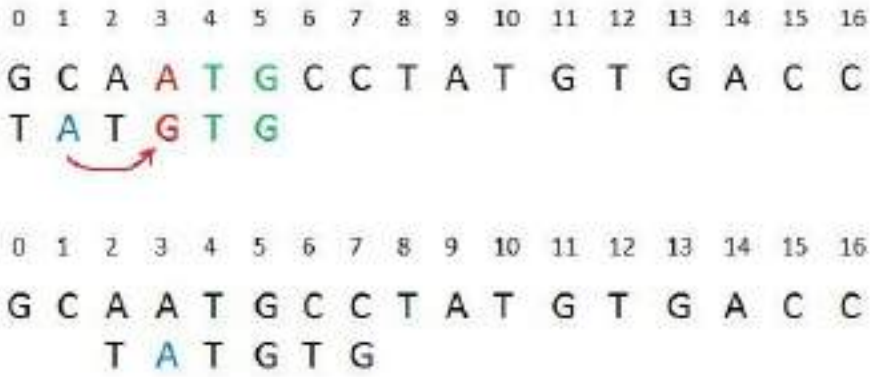
The Boyer Moore algorithm does preprocessing for the same reason. It processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. **So it uses greatest offset suggested by the two heuristics at every step**. Unlike the previous pattern searching algorithms, the **Boyer Moore algorithm starts matching from the last character of the pattern.**

Bad Character Heuristic

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –

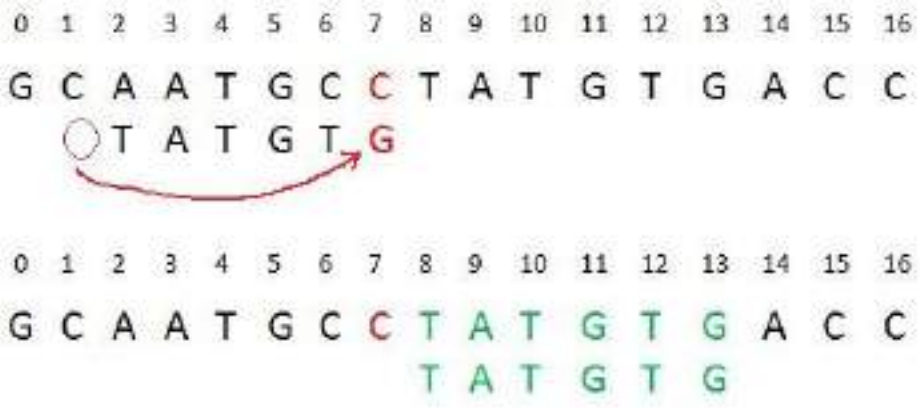
1. The mismatch becomes a match.
2. Pattern P moves past the mismatched character.

Case 1 – Mismatch become match
We will lookup the position of the last occurrence of the mismatched character in the pattern, and if the mismatched character exists in the pattern, then we'll shift the pattern such that it becomes aligned to the mismatched character in the text T.



Explanation: In the above example, we got a mismatch at position 3. Here our mismatching character is "A". Now we will search for last occurrence of "A" in pattern. We got "A" at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that "A" in pattern get aligned with "A" in text.

Case 2 – Pattern move past the mismatch character
We'll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.



Explanation:
Here we have a mismatch at position 7. The mismatching character "C" does not exist in pattern before position 7 so we'll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because "C" does not exist in the pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

In the following implementation, we preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes time in the best case.

Time Complexity : O(n x m)
Auxiliary Space: O(1)

The Bad Character Heuristic may take time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = "AAAAAAAAAAAAAAAAA" and pat[] = "AAAAA". The Bad Character Heuristic may take O(n/m) in the best case. The best case occurs when all the characters of the text and pattern are different.

Boyer Moore Algorithm | Good Suffix heuristic

We have already discussed Bad character heuristic variation of Boyer Moore algorithm. In this article we will discuss **Good Suffix** heuristic for pattern searching. Just like bad character heuristic, a preprocessing table is generated for good suffix heuristic.

Good Suffix Heuristic

Let **t** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern until :
1) Another occurrence of t in P matched with t in T.
2) A prefix of P, which matches with suffix of t
3) P moves past t

Case 1: Another occurrence of t in P matched with t in T

Pattern P might contain few more occurrences of **t**. In such case, we will try to shift the pattern to align that occurrence with t in text T. For example-

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

Explanation: In the above example, we have got a substring t of text T matched with pattern P (in green) before mismatch at index 2. Now we will search for occurrence of t ("AB") in P. We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times to align t in P with t in T. This is weak rule of original Boyer Moore and not much effective, we will discuss a **Strong Good Suffix rule** shortly.

Case 2: A prefix of P, which matches with suffix of t in T

It is not always likely that we will find the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes we can search for some **suffix of t** matching with some **prefix of P** and try to align them by shifting P. For example –

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Figure – Case 2

Explanation: In above example, we have got t ("BAB") matched with P (in green) at index 2-4 before mismatch . But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t. We have found prefix "AB" (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t "AB" starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.

Case 3: P moves past t

If the above two cases are not satisfied, we will shift the pattern past the t. For example –

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Figure – Case 3

Explanation: If above example, there exist no occurrence of t ("AB") in P and also there is no prefix in P which matches with the suffix of t. So, in that case, we can never find any perfect match before index 4, so we will shift the P past the t ie. to index 5.

Strong Good suffix Heuristic

Suppose substring **q = P[i to n]** got matched with **t** in T and **c = P[i-1]** is the mismatching character. Now unlike case 1 we will search for t in P which is not preceded by character **c**. The closest such occurrence is then aligned with t in T by shifting pattern P. For example –

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P	A	A	C	C	A	C	C	A	C									

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P							A	A	C	C	A	C	C	A	C			

Figure – strong suffix rule

Explanation: In above example, **q = P[7 to 8]** got matched with t in T. The mismatching character **c** is "C" at position P[6]. Now if we start searching t in P we will get the first occurrence of t starting at position 4. But this occurrence is preceded by "C" which is equal to c, so we will skip this and carry on searching. At position 1 we got another occurrence of t (in the yellow background). This occurrence is preceded by "A" (in blue) which is not equivalent to c. So we will shift pattern P 6 times to align this occurrence with t in T. We are doing this because we already know that character **c = "C"** causes the mismatch. So any occurrence of t preceded by c will again cause mismatch when aligned with t, so that's why it is better to skip this.

Knuth-Morris-Pratt Algorithm

KMP Algorithm is one of the most popular patterns matching algorithms. KMP stands for Knuth Morris Pratt. KMP algorithm was invented by **Donald Knuth** and **Vaughan Pratt** together and independently by **James H Morris** in the year 1970. In the year 1977, all the three jointly published KMP Algorithm. KMP algorithm was the first **linear time complexity** algorithm for string matching.

KMP algorithm is one of the string matching algorithms used to find a Pattern in a Text.

KMP algorithm is used to find a **"Pattern"** in a **"Text"**. This algorithm campares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called **"Prefix Table"** to skip characters comparison while matching. Some times prefix table is also known as **LPS Table**. Here LPS stands for **"Longest proper Prefix which is also Suffix"**.

Steps for Creating LPS Table (Prefix Table)

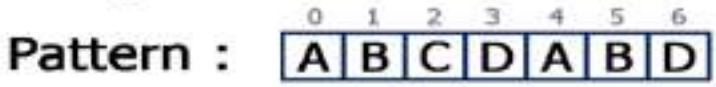
- **Step 1 -** Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
- **Step 2 -** Define variables **i & j**. Set i = 0, j = 1 and LPS[0] = 0.
- **Step 3 -** Compare the characters at **Pattern[i]** and **Pattern[j].**
- **Step 4 -** If both are matched then set **LPS[j] = i+1** and increment both i & j values by one.
  Goto to Step 3.

- **Step 5 -** If both are not matched then check the value of variable 'i'. If it is '0' then set **LPS[j] = 0** and increment 'j' value by one, if it is not '0' then set **i = LPS[i-1]**. Goto Step 3.
- **Step 6-** Repeat above steps until all the values of LPS[] are filled.

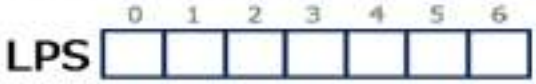
Let us use above steps to create prefix table for a pattern...

Example for creating KMP Algorithm's LPS Table (Prefix Table)

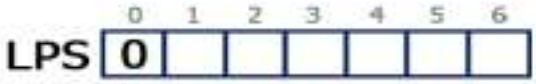
Consider the following Pattern



Let us define LPS[] table with size 7 which is equal to length of the Pattern

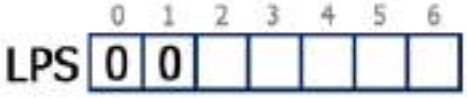


Step 1 - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.



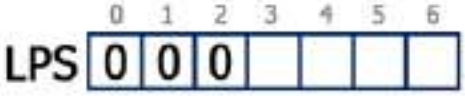
i = 0 and j = 1

Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.



i = 0 and j = 2

Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.



i = 0 and j = 3

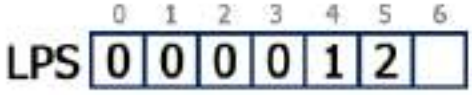
Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

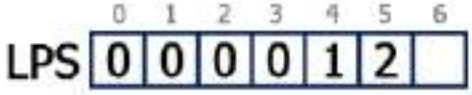
i = 1 and j = 5

Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.



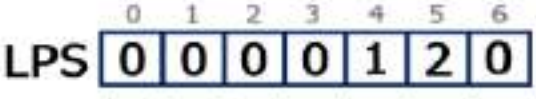
i = 2 and j = 6

Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.

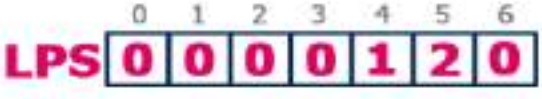


i = 0 and j = 6

Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.



Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...



How to use LPS Table

We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred.
When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern. If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

How the KMP Algorithm Works

Let us see a working example of KMP Algorithm to find a Pattern in a Text...

Consider the following Text and Pattern

Text : ABC ABCDAB ABCDABCDABDE
Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right

Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
	0	1	2	3	4	5	6																
Pattern	A	B	C	D	A	B	D																

Here mismatch occured at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

Step 2 - Start comparing first character of Pattern with next character of Text.

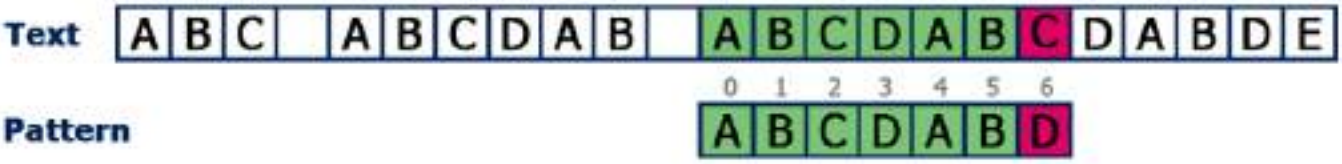
Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
					0	1	2	3	4	5	6												
Pattern					A	B	C	D	A	B	D												

Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values

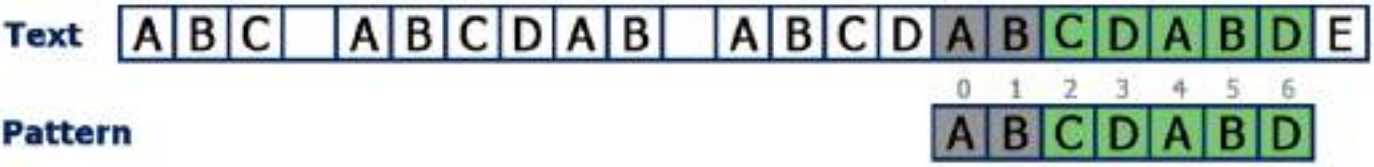
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

Trie Data Structure

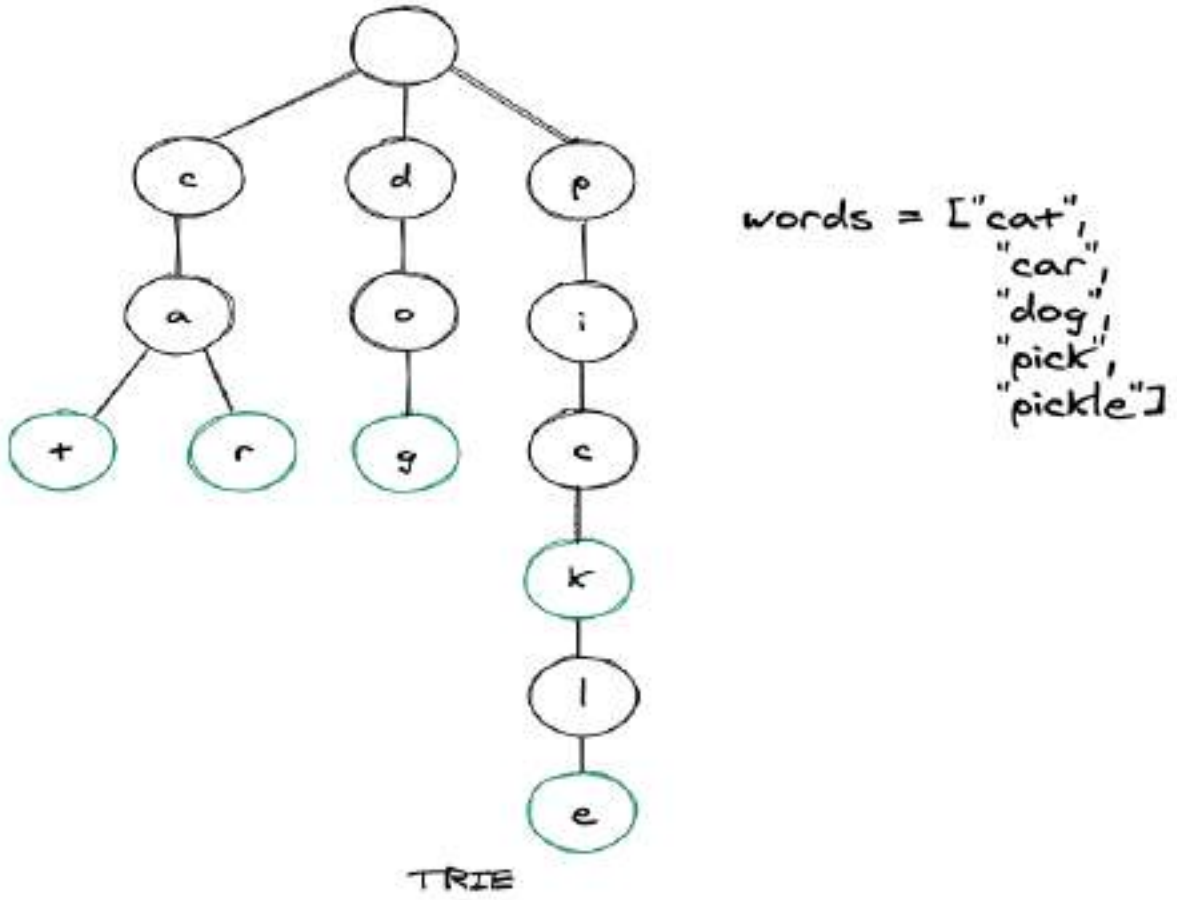
A **Trie is an advanced data structure** that is sometimes also known as **prefix tree or digital tree**. It is a tree that stores the data in an ordered and efficient way. We generally **use trie's to store strings**. Each node of a trie can have as many as 26 references (pointers).

Each node of a trie consists of two things:

- A character
- A boolean value is used to implement whether this character represents the end of the word.

Tries in general are used to store English characters, hence each character can have 26 references. Nodes in a trie do not store entire keys, instead, they store a part of the key(usually a character of the string). When we traverse down from the root node to the leaf node, we can build the key from these small parts of the key.

Let's build a trie by inserting some words in it. Below is a pictorial representation of the same, we have 5 words, and then we are inserting these words one by one in our trie.



As it can be seen in the image above, the key(words) can be formed as we traverse down from the root node to the leaf nodes. It can be noted that the green highlighted nodes, represents the end Of Word boolean value of a word which in turn means that this particular word is completed at this node. Also, the root node of a trie is empty so that it can refer to all the members of the alphabet the trie is using to store, and the children nodes of any node of a trie can have at most 26 references. Tries are not balanced in nature, unlike AVL trees.

Why use Trie Data Structure?

When we talk about the **fastest ways to retrieve values** from a data structure, **hash tables generally comes to our mind**. Though very efficient in nature but still very less talked about as when compared to hash tables, **trie's are much more efficient than hash tables** and also they possess several advantages over the same. Mainly:

- There won't be any collisions hence making the worst performance better than a hash table that is not implemented properly.
- No need for hash functions.
- Lookup time for a string in trie is O(k) where **k = length of the word**.
- It can take even less than O(k) time when the word is not there in a trie.

A trie is a tree-like information retrieval data structure whose nodes store the letters of an alphabet. It is also known as a digital tree or a radix tree or prefix tree.

Tries are classified into three categories:

1. Standard Trie
2. Compressed Trie
3. Suffix Trie

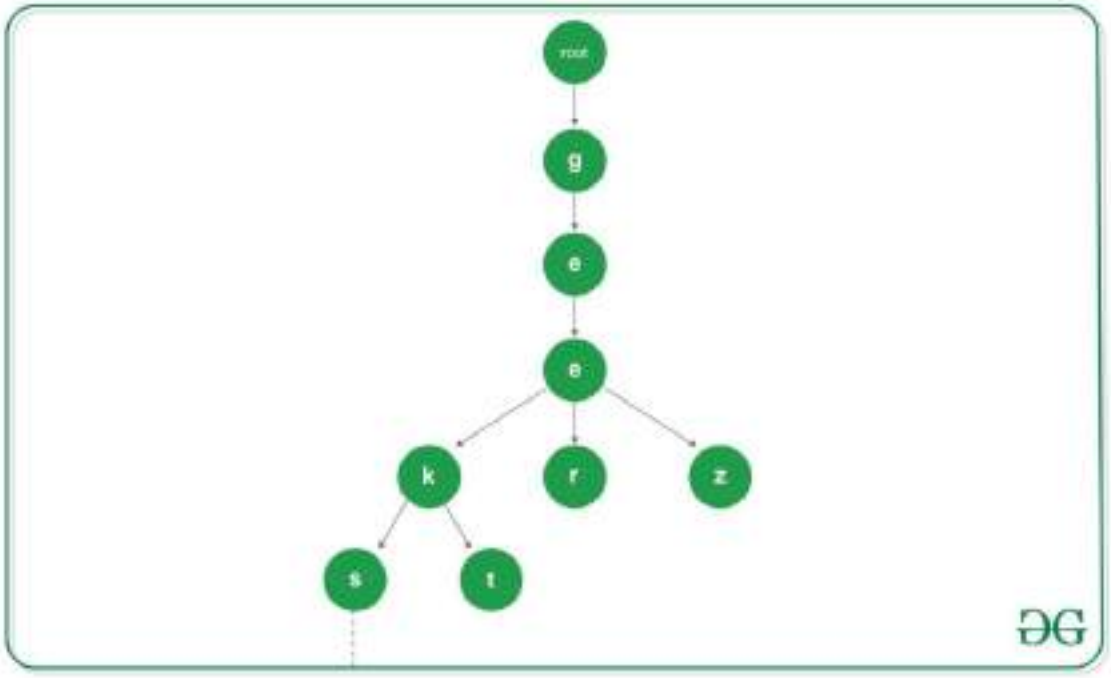
Standard Trie: A standard trie have the following properties:

A Standard Trie has the below structure:

```
class Node {
    // Array to store the nodes of a tree
    Node[] children = new Node[26];
    // To check for end of string
   boolean isWordEnd;
}
```

- It is an ordered tree like data structure.
- Each node(except the **root** node) in a standard trie is labeled with a character.
- The children of a node are in alphabetical order.
- Each node or branch represents a possible character of keys or words.
- Each node or branch may have multiple branches.
- The last node of every key or word is used to mark the end of word or node.

Below is the illustration of the Standard Trie:



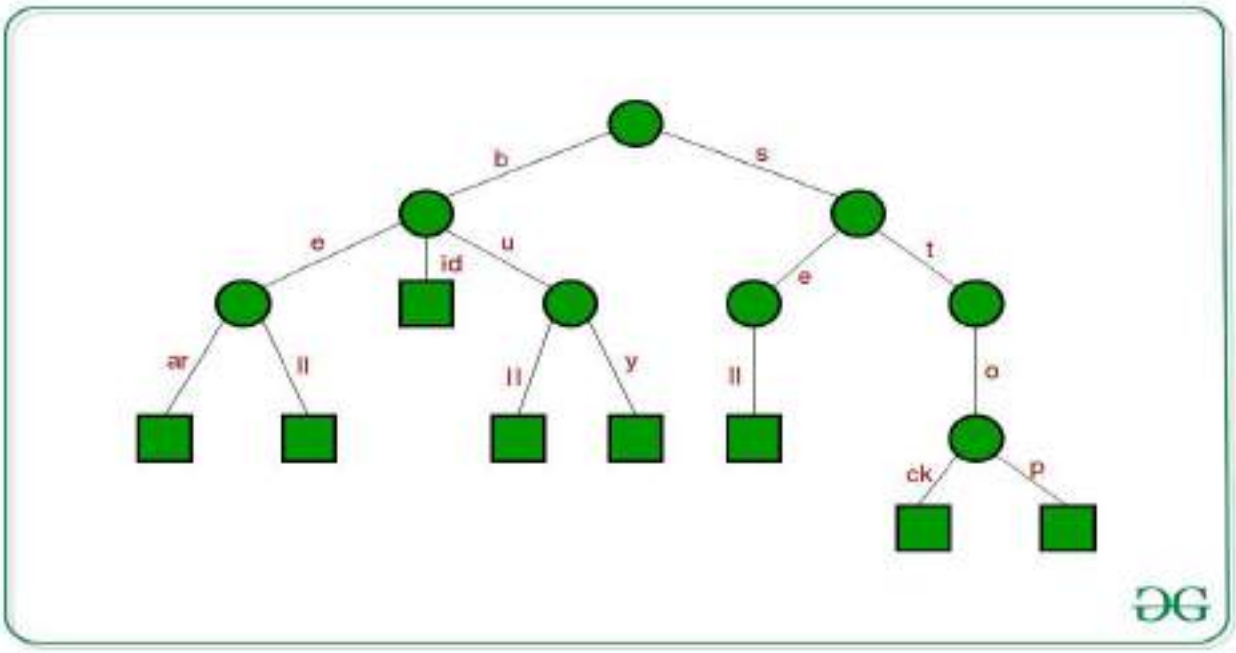
Compressed Trie A Compressed trie have the following properties:

A Compressed Trie has the below structure:

```
class Node {

// Array to store the nodes of tree
Node[] children = new Node[26];
// To store the edgeLabel
StringBuilder[] edgeLabel = new StringBuilder[26];
// To check for end of string
boolean isEnd;
}
```

- A Compressed Trie is an advanced version of the standard trie.
- Each nodes(except the **leaf** nodes) have atleast 2 children.
- It is used to achieve space optimization.
- To derive a Compressed Trie from a Standard Trie, compression of chains of redundant nodes is performed.
- It consists of grouping, re-grouping and un-grouping of keys of characters.
- While performing the insertion operation, it may be required to un-group the already grouped characters.
- While performing the deletion operation, it may be required to re-group the already grouped characters.
- A compressed trie T storing s strings(**keys**) has s external nodes and O(s) total number of nodes.

Below is the illustration of the Compressed Trie:



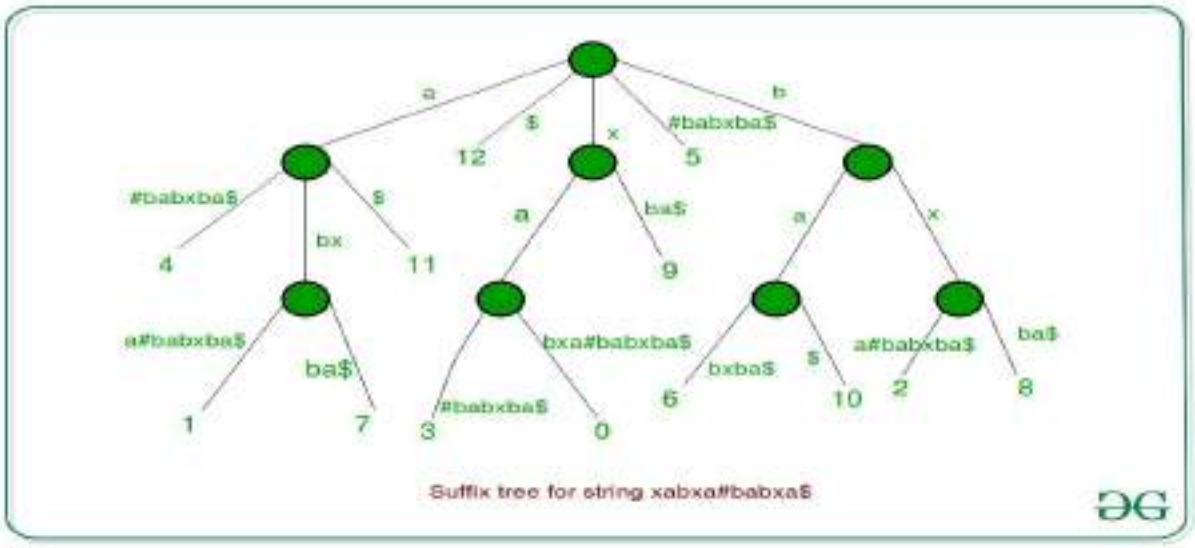
Suffix Trie A Suffix trie have the following properties:

A Compressed Trie has the below structure:

```
 struct SuffixTreeNode {
// Array to store the nodes
struct SuffixTreeNode *children[256];
  //pointer to other node via suffix link
struct SuffixTreeNode *suffixLink;
  // (start, end) interval specifies the edge,
// by which the node is connected to its
// parent node
int start;
int *end;
  // For leaf nodes, it stores the index of
// Suffix for the path  from root to leaf
int suffixIndex;
}
```

- A Suffix Trie is an advanced version of the compressed trie.
- The most common application of suffix trie is Pattern Matching.
- While performing the insertion operation, both the word and its suffixes are stored.
- A suffix trie is also used in word matching and prefix matching.
- To generate a suffix trie, all the suffixes of given string are considered as individual words.
- Using the suffixes, compressed trie is built.

Below is the illustration of the Suffix Trie:



Trie Applications

- The most common use of tries in real world is the **autocomplete feature** that we get on a search engine(now everywhere else too). After we type something in the search bar, the tree of the potential words that we might enter is greatly reduced, which in turn allows the program to enumerate what kinds of strings are possible for the words we have typed in.
- Trie also helps in the case where we want to store additional information of a word, say the popularity of the word, which makes it so powerful. You might have seen that when you type **"foot"** on the search bar, you get **"football"** before anything say **"footpath"**. It is because **"football"** is a much popular word.
- Trie also has helped in checking the correct spellings of a word, as the path is similar for a slightly misspelled word.
- **String matching** is another case where tries to excel a lot.

Key Points

- The time complexity of creating a trie is O(m*n) where m = number of words in a trie and n = average length of each word.
- Inserting a node in a trie has a time complexity of O(n) where n = length of the word we are trying to insert.
- Inserting a node in a trie has a space complexity of O(n) where n = length of the word we are trying to insert.
- Time complexity for searching a key(word) in a trie is O(n) where n = length of the word we are searching.
- Space complexity for searching a key(word) in a trie is O(1).
- Searching for a prefix of a key(word) also has a time complexity of O(n) and space complexity of O(1).