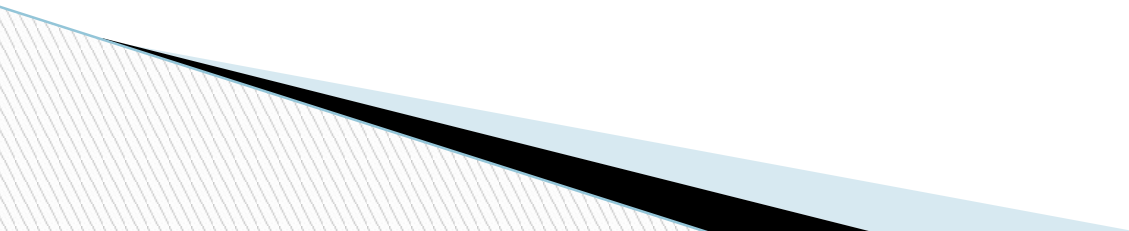


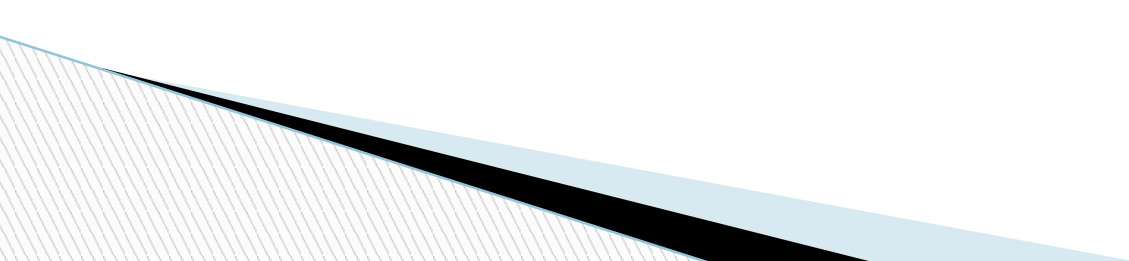
# **Fundamentals of Algorithm**

**SYBSC (Computer Science)**

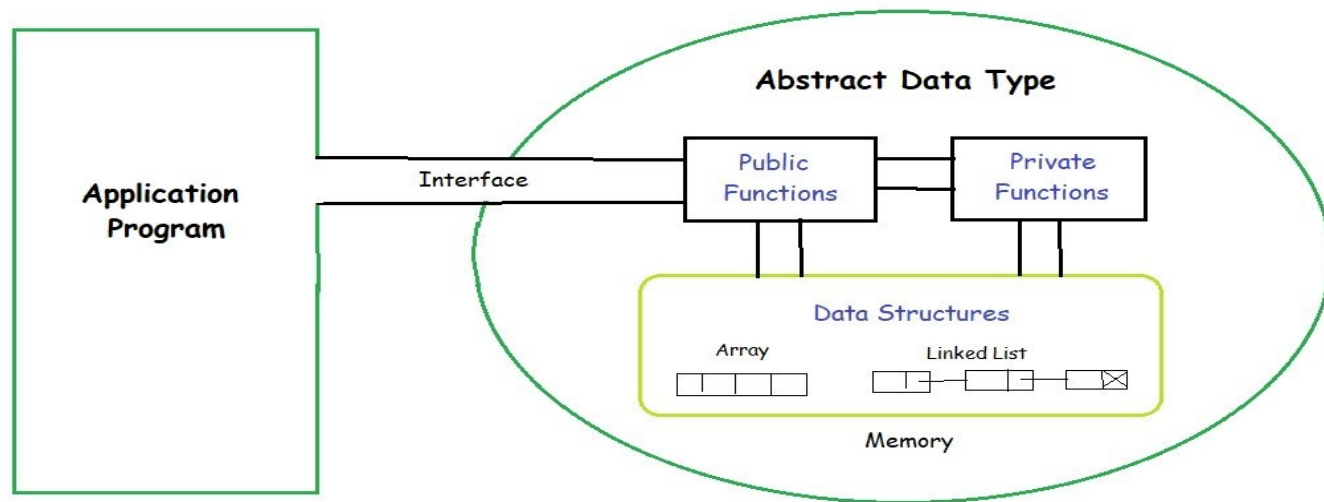


## **Abstract Data Type (ADT)**

The abstract data type is special kind of data type, whose behaviour is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive data types, but operation logics are hidden.



The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.



**Date( month, day, year ):** Creates a new Date instance initialized to the given Gregorian date which must be valid. Year 1 BC and earlier are indicated by negative year components.

**day():** Returns the Gregorian day number of this date.

**month():** Returns the Gregorian month number of this date.

**year():** Returns the Gregorian year of this date.

**monthName():** Returns the Gregorian month name of this date.

**dayOfWeek():** Returns the day of the week as a number between 0 and 6 with 0 representing Monday and 6 representing Sunday.

**numDays( otherDate ):** Returns the number of days as a positive integer between this date and the otherDate.

**isLeapYear():** Determines if this date falls in a leap year and returns the appropriate

## Bag ADT

**Bag():** Creates a bag that is initially empty.

**length ():** Returns the number of items stored in the bag. Accessed using the len() function.

**contains ( item ):** Determines if the given target item is stored in the bag and returns the appropriate boolean value. Accessed using the in operator.

**add( item ):** Adds the given item to the bag.

**remove( item ):** Removes and returns an occurrence of item from the bag. An exception is raised if the element is not in the bag.

**iterator ():** Creates and returns an iterator that can be used to iterate over the collection of items

A set is a container that stores a collection of unique values over a given comparable domain in which the stored values have no particular ordering.

**Set():** Creates a new set initialized to the empty set.

**length ():** Returns the number of elements in the set, also known as the cardinality. Accessed using the `len()` function.

**contains ( element ):** Determines if the given value is an element of the set and returns the appropriate boolean value. Accessed using the `in` operator.

**add( element ):** the set by adding the given value or element to the set if the element is not already a member. If the element is not unique, no action is taken and the operation is skipped.

**remove( element ):** Removes the given value from the set if the value is contained in the set and raises an exception otherwise.

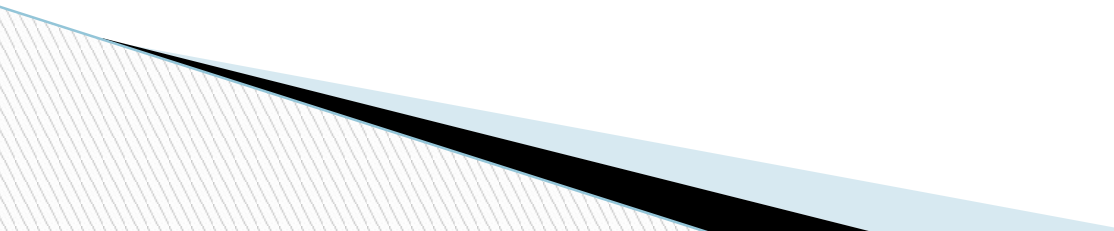
**equals ( setB ):** Determines if the set is equal to another set and returns a boolean value. For two sets, A and B, to be equal, both A and B must contain the same number of elements and all elements in A must also be elements in B. If both sets are empty, the sets are equal. Access with `==` or `!=`.

**isSubsetOf( setB ):** Determines if the set is a subset of another set and returns a boolean value. For set A to be a subset of B, all elements in A must also be elements in B.

**union( setB ):** Creates and returns a new set that is the union of this set and setB. The new set created from the union of two sets, A and B, contains all elements in A plus those elements in B that are not in A. Neither set A nor set B is modified by this operation.

**intersect( setB ):** Creates and returns a new set that is the intersection of this set and setB. The intersection of sets A and B contains only those elements that are in both A and B. Neither set A nor set B is modified by this operation.

**difference( setB ):** Creates and returns a new set that is the difference of this set and setB. The set difference,  $A \setminus B$ , contains only those elements that are in A but not in B. Neither set A nor set B is modified by this operation.



```
smith = Set()
smith.add( "CSCI-112" )
smith.add( "MATH-121" )
smith.add( "HIST-340" )
smith.add( "ECON-101" )
roberts = Set()
roberts.add( "POL-101" )
roberts.add( "ANTH-230" )
roberts.add( "CSCI-112" )
roberts.add( "ECON-101" )
```

```
uniqueCourses = smith.difference( roberts )
for course in sameCourses :
    print( course )
```

```
if smith == roberts :
    print( "Smith and Roberts are taking the same
courses." )
else :
    sameCourses = smith.intersection( roberts )
    if sameCourses.isEmpty() :
        print( "Smith and Roberts are not taking any of
^
+ \"the same courses.\" )
    else :
        print( "Smith and Roberts are taking some of
the ^
+ \"same courses:\"" )
        for course in sameCourses :
            print( course )
```





**Map():** Creates a new empty map.

**length ():** Returns the number of key/value pairs in the map.

**contains ( key ):** Determines if the given key is in the map and returns True if the key is found and False otherwise.

**add( key, value ):** Adds a new key/value pair to the map if the key is not already in the map or replaces the data associated with the key if the key is in the map. Returns True if this is a new key and False if the data associated with the existing key is replaced.

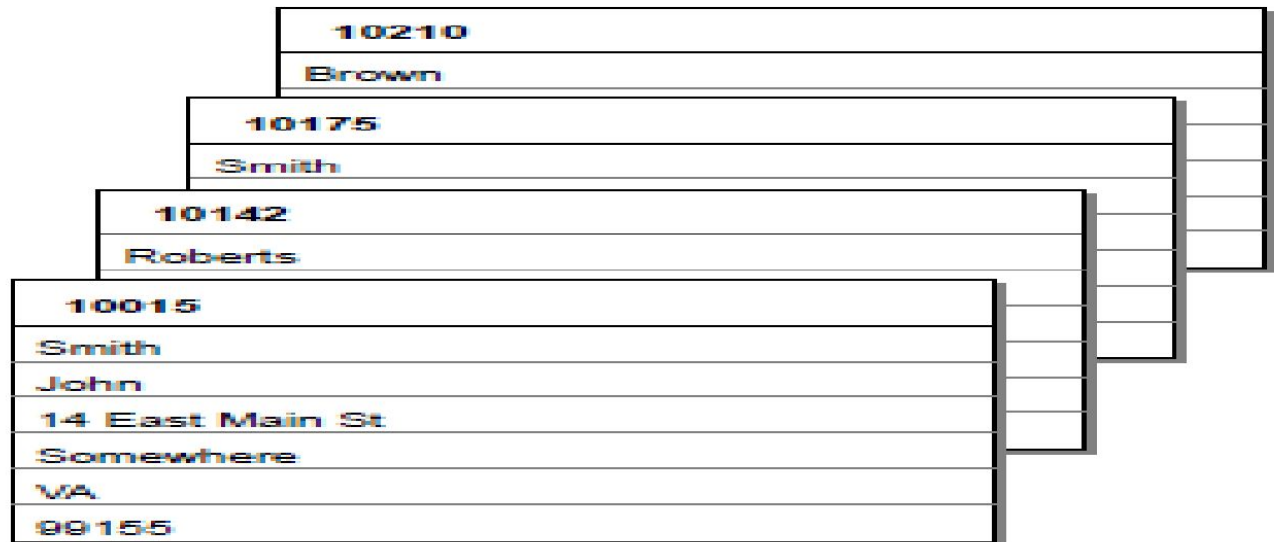
**remove( key ):** Removes the key/value pair for the given key if it is in the map and raises an exception otherwise.

**valueOf( key ):** Returns the data record associated with the given key. The key must exist in the map or an exception is raised.

**iterator ():** Creates and returns an iterator that can be used to iterate over the keys in the map

## Maps

Searching for data items based on unique key values is a very common application in computer science. An abstract data type that provides this type of search capability is often referred to as a map or dictionary since it maps a key to a corresponding value.



10210	Brown
10175	Smith
10142	Roberts
10015	Smith
	John
	14 East Main St
	Somewhere
	VA
99155	

# Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

**Linear Search ( Array A, Value x)**

**Step 1: Set i to 1**

**Step 2: if  $i > n$  then go to step 7**

**Step 3: if  $A[i] = x$  then go to step 6**

**Step 4: Set i to  $i + 1$**

**Step 5: Go to Step 2**

**Step 6: Print Element x Found at index i and go to step 8**

**Step 7: Print element not found**

**Step 8: Exit**

Linear Search



# BINARY SEARCH ALGORITHM

Find - '75'

1	3	12	14	23	34	55	65	75	78
---	---	----	----	----	----	----	----	----	----

0 1 2 3 4 5 6 7 8 9

Iteration 1 -

$arr[mid] < 75$

Iteration 2 -

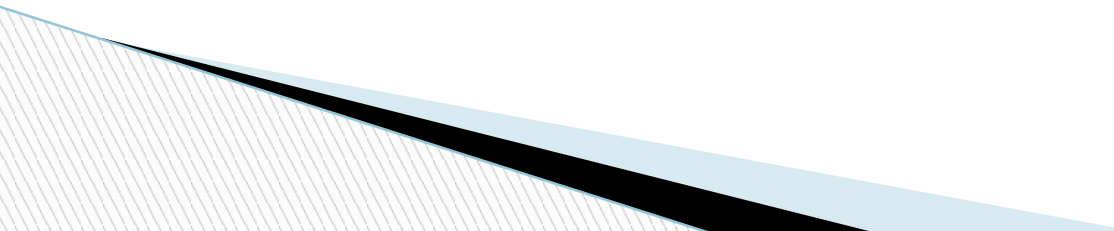
$arr[mid] < 75$

Iteration 3 -

$arr[mid] == x$

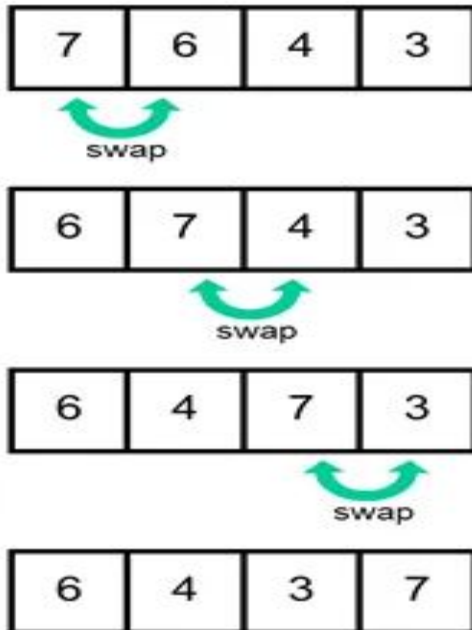
# Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where  $n$  is the number of items.

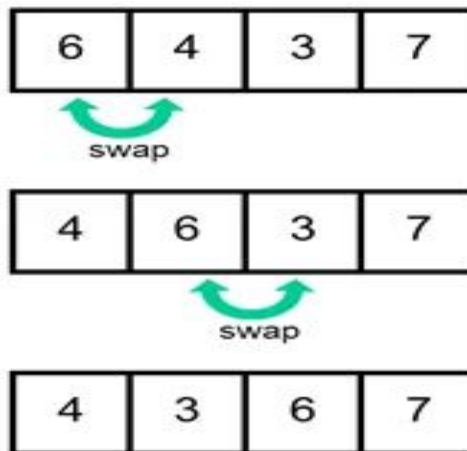


## How Bubble Sort Works?

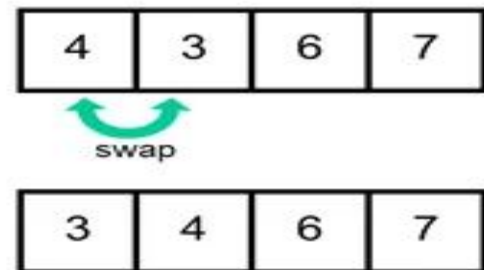
First pass



Second pass



Third pass



## **selection sort**

In the **selection sort** algorithm, an array is sorted by finding the minimum element from the unsorted part and inserting it at the beginning. Worst case time complexity is  $O(n^2)$





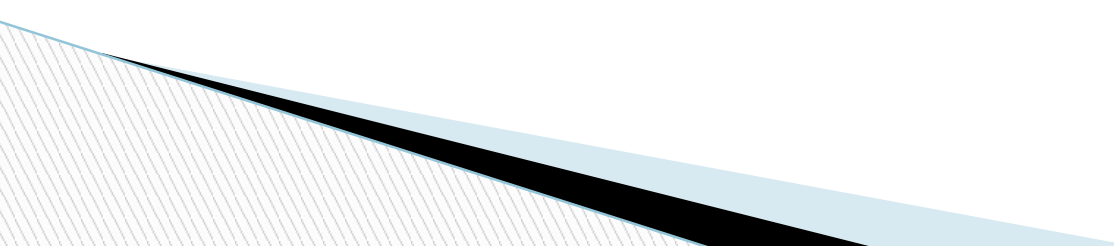
## Insertion Sort

The Insertion sort is a straightforward and more efficient algorithm than the previous bubble sort algorithm. The insertion sort algorithm concept is based on the deck of the card where we sort the playing card according to a particular card.

The array spilled virtually in the two parts in the insertion sort - An **unsorted part** and **sorted part**.

The sorted part contains the first element of the array and other unsorted subpart contains the rest of the array. The first element in the unsorted array is compared to the sorted array so that we can place it into a proper sub-array.

It focuses on inserting the elements by moving all elements if the right-side value is smaller than the left side.



## Merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being  $O(n \log n)$ , it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.



```

procedure mergesort( var a as array )
  if ( n == 1 ) return a

```

```

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]

```

```

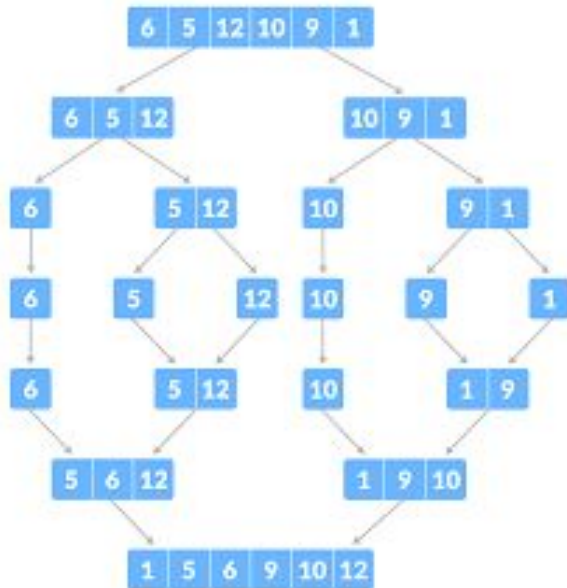
  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

```

```

  return merge( l1, l2 )
end procedure

```



```

procedure merge( var a as array, var b as array )

```

```

  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

```

```

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

```

```

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while
  return c

```

# Stack

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

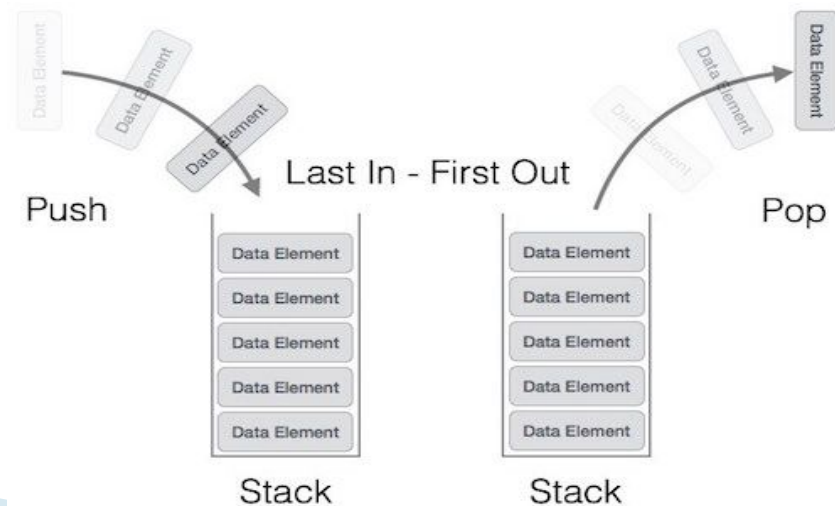
Mainly the following three basic operations are performed in the stack:

**Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

**Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

**Peek or Top:** Returns top element of stack.

**isEmpty:** Returns true if stack is empty





## Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

**push()** – Pushing (storing) an element on the stack.

**pop()** – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

**peek()** – get the top data element of the stack, without removing it.

**isFull()** – check if stack is full.

**isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

# peek()

## Algorithm of peek()

### function –

begin procedure peek

    return stack[top]

end procedure

```
int peek()
```

```
{ return stack[top]; }
```

```
begin procedure isfull
if top equals to MAXSIZE
    return true
else
return false endif
end procedure
```

```
bool isfull()
{
if(top == MAXSIZE)
return true;
Else
    return false;
}
```

```
begin procedure
isempty
if top less than 1
    return true
Else
    return false
endif
end procedure
```

```
bool isempty()
{ if(top == -1)
    return true;
else
return false; }
```



# Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

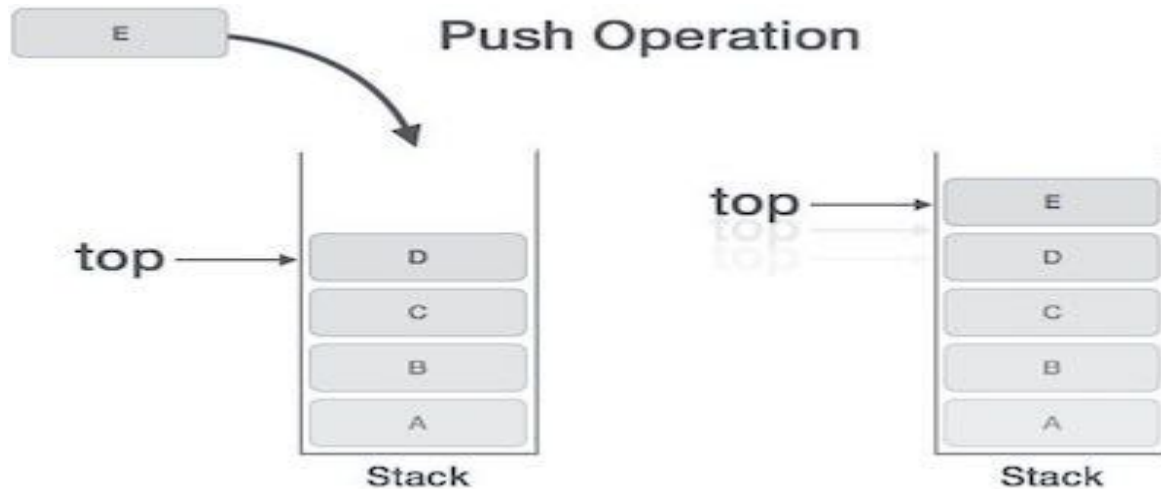
**Step 1** – Checks if the stack is full.

**Step 2** – If the stack is full, produces an error and exit.

**Step 3** – If the stack is not full, increments **top** to point next empty space.

**Step 4** – Adds data element to the stack location, where **top** is pointing.

**Step 5** – Returns success.



```
begin procedure push: stack, data
if stack is full
return null
endif top ← top + 1
stack[top] ← data
```

```
void push(int data)
{ if(!isFull())
{ top = top + 1;
  stack[top] = data;
}
else
{ printf("Could not insert data, Stack is full.\n"); }
}
```

# Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

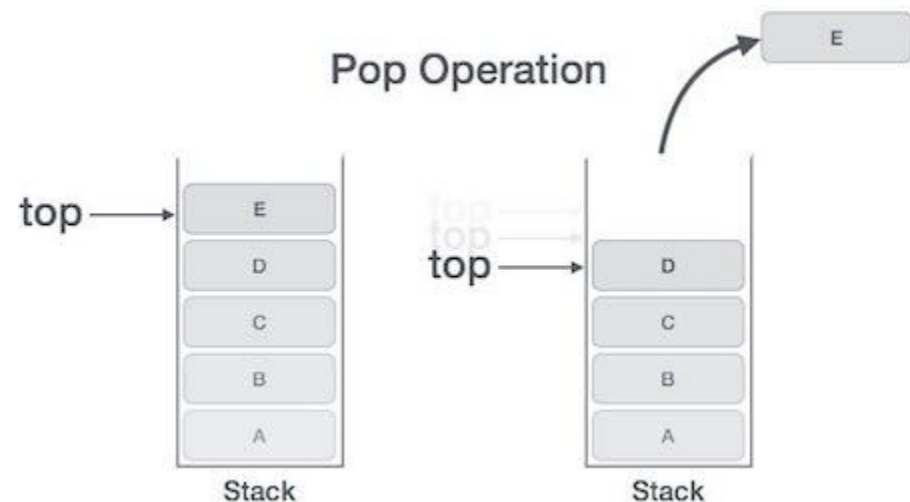
**Step 1** – Checks if the stack is empty.

**Step 2** – If the stack is empty, produces an error and exit.

**Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.

**Step 4** – Decreases the value of top by 1.

**Step 5** – Returns success.



```
begin procedure pop: stack
  if stack is empty
return null
  endif data ← stack[top] top ← top - 1
  return data
end procedure
```

```
int pop(int data)
{ if(!isempty())
  { data = stack[top];
    top = top - 1;
    return data;
  }
else
{ printf("Could not retrieve data, Stack is empty.\n");
}
}
```

# Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



## Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

## Basic Operations

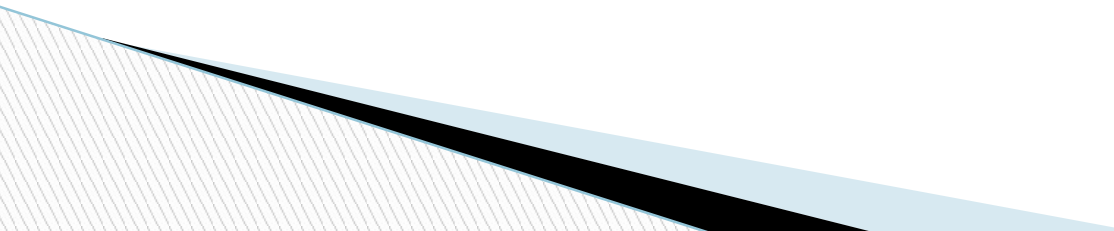
Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.



## Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

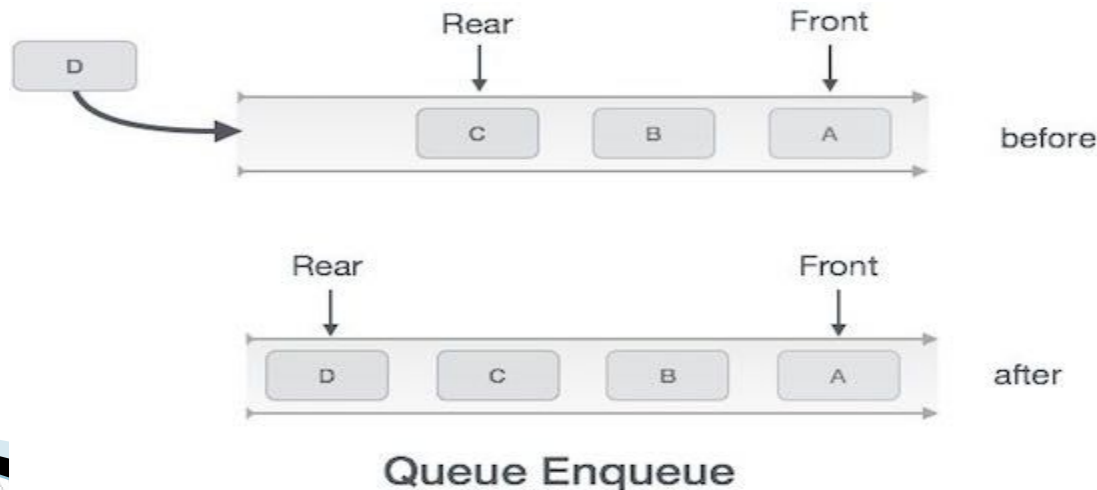
**Step 1** – Check if the queue is full.

**Step 2** – If the queue is full, produce overflow error and exit.

**Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.

**Step 4** – Add data element to the queue location, where the rear is pointing.

**Step 5** – return success.





## Deque Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

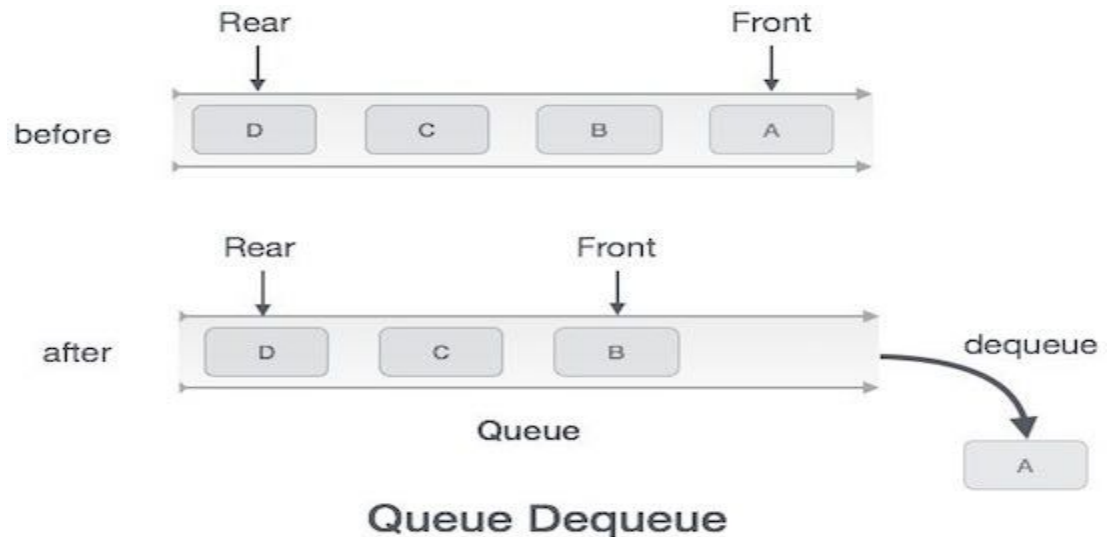
**Step 1** – Check if the queue is empty.

**Step 2** – If the queue is empty, produce underflow error and exit.

**Step 3** – If the queue is not empty, access the data where **front** is pointing.

**Step 4** – Increment **front** pointer to point to the next available data element.

**Step 5** – Return success.



## **Recursive function**

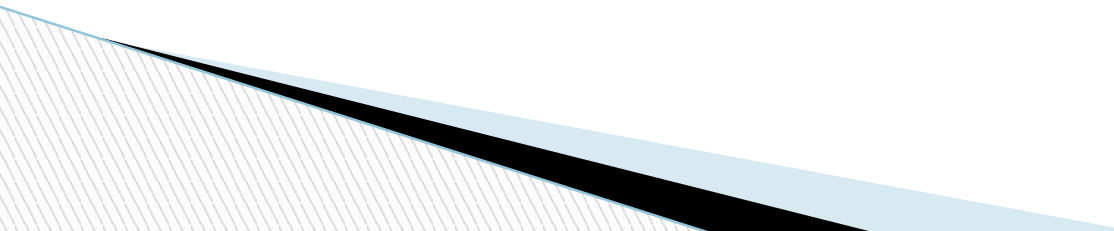
Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion.

### **Properties of recursive Function**

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –

**Base criteria** – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.

**Progressive approach** – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.



## **Analysis of Recursion**

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

### **Time Complexity**

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is  $O(1)$ , hence the  $(n)$  number of times a recursive call is made makes the recursive function  $O(n)$ .

### **Space Complexity**

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

