DADABHOY INSTITUTE OF HIGHER EDUCATION
DIHE

Course Title: **Data Structure & Algorithm**

Mid Term Examination
Spring Semester- 2022

Name: **Muhammad Muzammil**

Program: **Computer science**

Id: **BSCS/3-17/M03033**

Semester: **7th, Evening**

Q1: (a) Difference between Linear and Non-Linear Data Structures?

Ans) Linear Data Structures

A Linear data structure have data elements arranged in sequential manner and each member element is connected to its previous and next element. This connection helps to traverse a linear data structure in a single level and in single run. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

Non-linear Data Structures

A non-linear data structure has no set sequence of connecting all its elements and each element can have multiple paths to connect to other elements. Such data structures support multi-level storage and often cannot be traversed in single run. Such data structures are not easy to implement but are more efficient in utilizing computer memory. Examples of non-linear data structures are Tree, BST, Graphs etc.

Following are the important differences between Linear Data Structures and Non-linear Data Structures.

Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
1	Data Element Arrangement	In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
2	Levels	In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
3	Implementation complexity	Linear data structures are easier to implement.	Non-linear data structures are difficult to understand and implement as compared to linear data structures.
4	Traversal	Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
5	Memory utilization	Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.

Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
6	Time Complexity	Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
7	Examples	Array, List, Queue, Stack.	Graph, Map, Tree.

(b) What are the basic operations supported by a list? Describe any two in detail

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

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

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

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

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

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.

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

Basic Operations

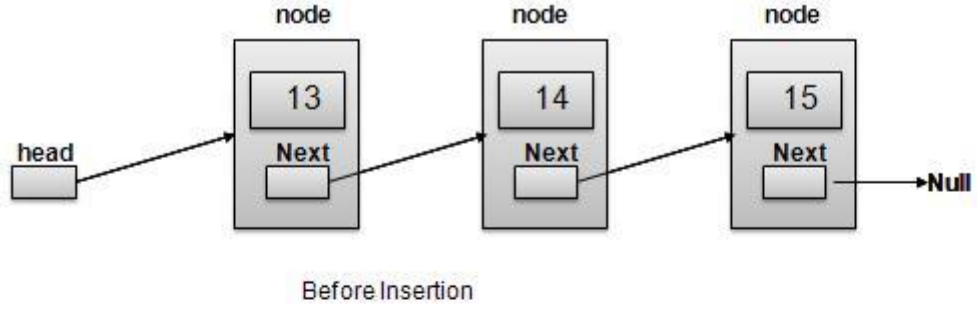
Following are the basic operations supported by a list.

- **Insertion** − add an element at the beginning of the list.
- **Deletion** − delete an element at the beginning of the list.
- **Display** − displaying complete list.
- **Search** − search an element using given key.
- **Delete** − delete an element using given key.

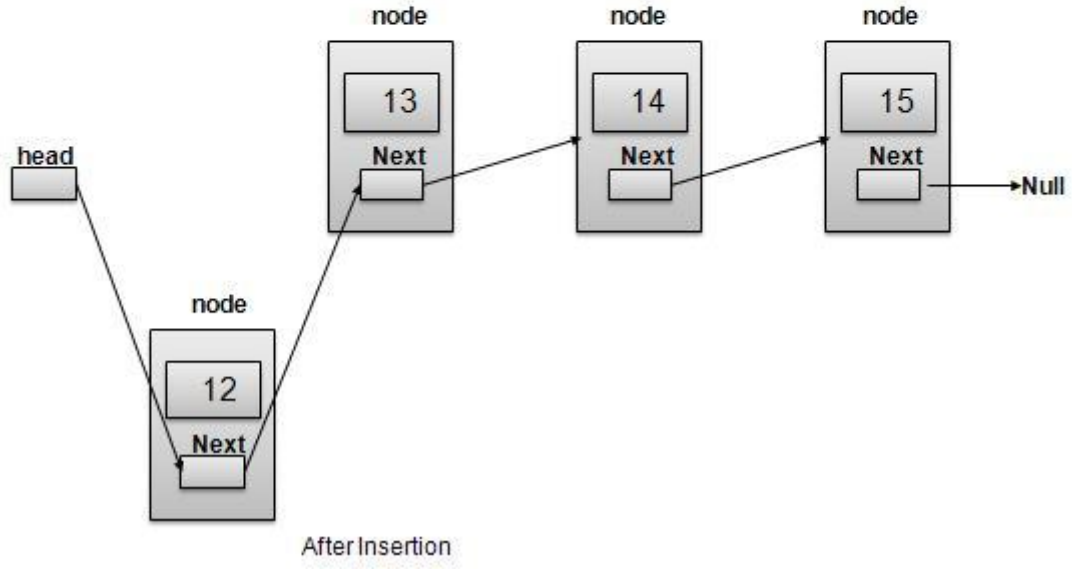
Insertion Operation

Insertion is a three-step process −

- Create a new Link with provided data.
- Point New Link to old First Link.
- Point First Link to this New Link.



Before Insertion



After Insertion

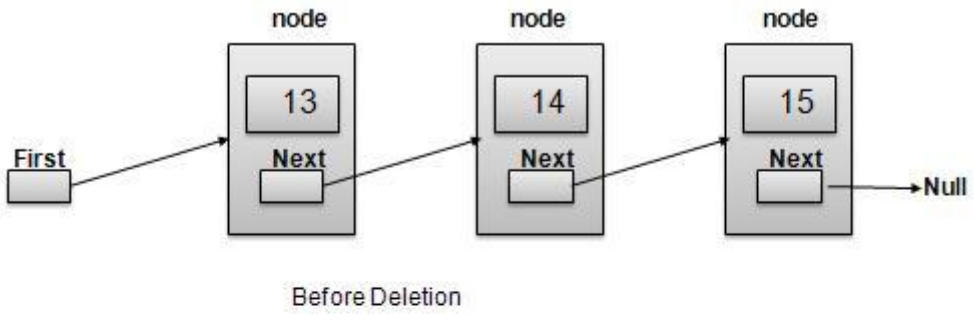
//insert link at the first location

```
void insertFirst(int key, int data){
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

   //point it to old first node
   link->next = head;

   //point first to new first node
   head = link;
}
```

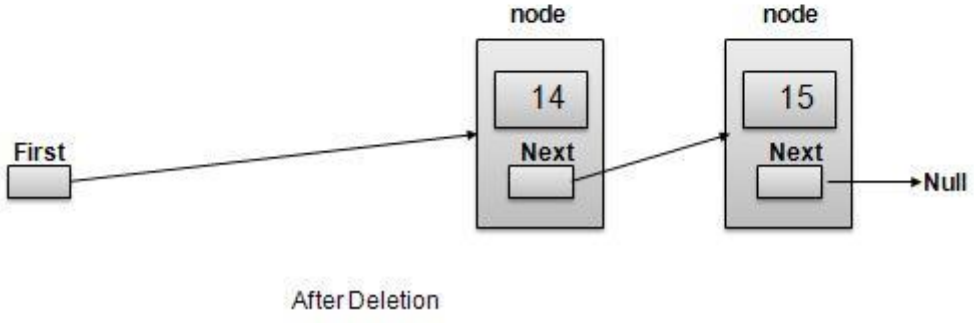
Deletion Operation

Deletion is a twostep process −

- Get the Link pointed by First Link as Temp Link.
- Point First Link to Temp Link's Next Link.



Before Deletion



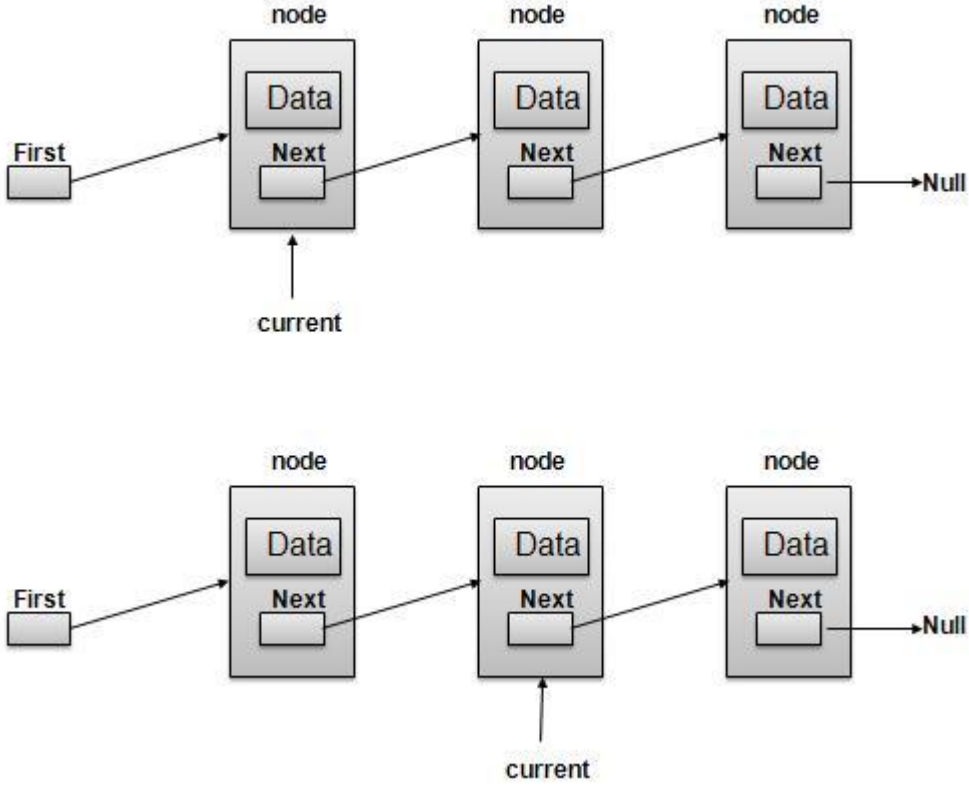
After Deletion

```
//delete first item
struct node* deleteFirst(){
   //save reference to first link
   struct node *tempLink = head;

   //mark next to first link as first
   head = head->next;

   //return the deleted link
   return tempLink;
}
```

Navigation Operation

Navigation is a recursive step process and is basis of many operations like search, delete etc. −

- Get the Link pointed by First Link as Current Link.
- Check if Current Link is not null and display it.
- Point Current Link to Next Link of Current Link and move to above step.



Q2: (a) Define pop operation in stack with the help of algorithm and coding?

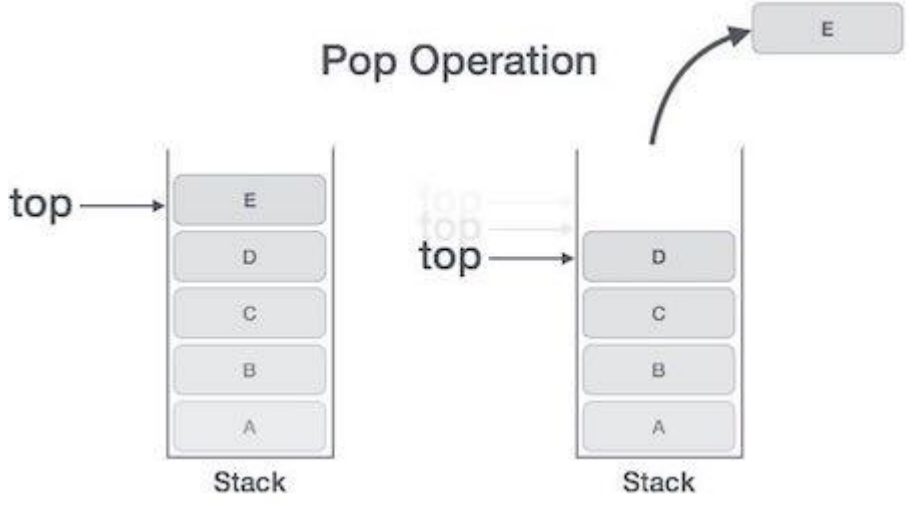
ANS) Pop Operation

POP operation is performed on the stack to remove items from the stack. We can perform the Pop operation only at the top of the stack. In an array implementation of pop () operation, the data element is not actually removed, instead the top is decremented to a lower position in the stack to point to the next value.

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.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

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

(b) What are the advantages of Circular Linked List?

ANS) **Advantages of Circular Linked Lists:**

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- Circular Doubly Linked Lists are used for implementation of advanced data structures like

- We can go to any node from any node in the Circular linked list which was not possible in the singly linked list if we reached the last node.

- Easily we can go to head from the last node

- In a circular list, any node can be starting point means we can traverse each node from any point.
- Some problems are circular and a circular data structure would be more natural when used to represent it
- The entire list can be traversed starting from any node (traverse means visit every node just once)
- fewer special cases when coding (all nodes have a node before and after it)

Q3: (a) What are the three ways use to traverse a tree? Apply In-order Traversal to visit the given below tree

ANS) Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

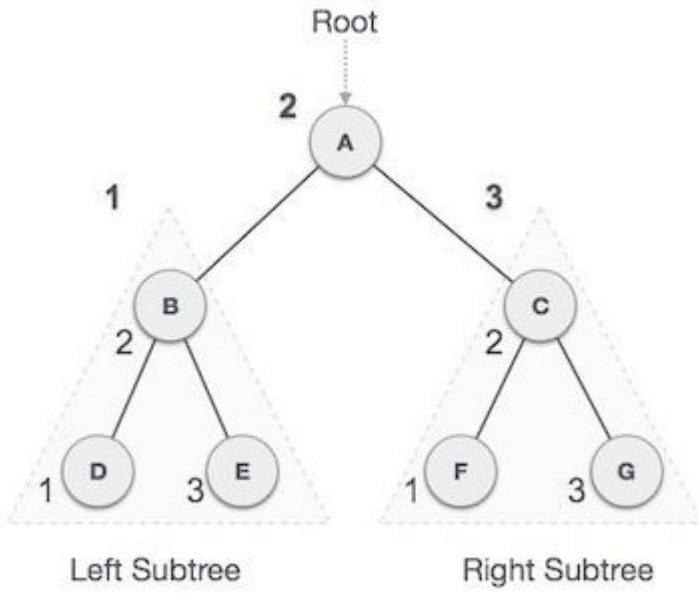
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in order traversal of this tree will be −

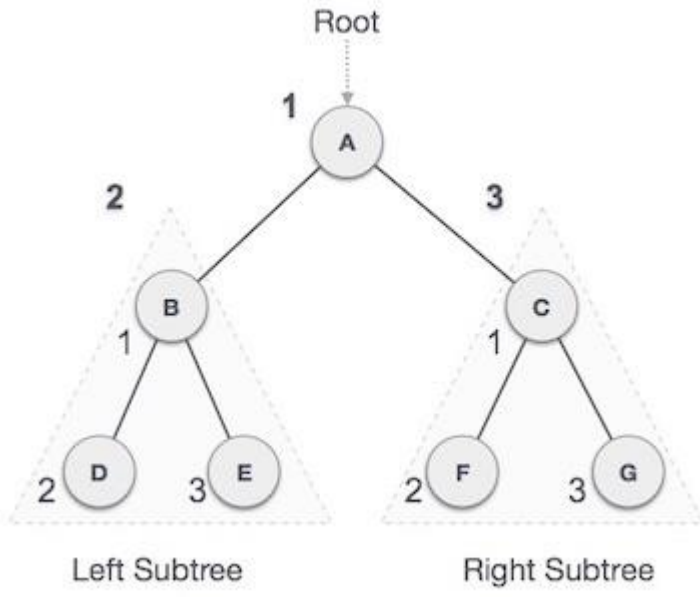
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

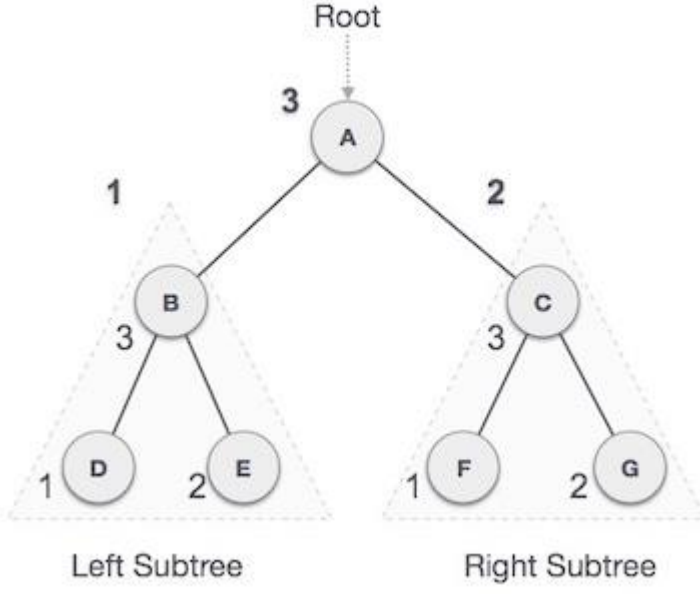
$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

(b) Describe Enqueue & Dequeue Operations in Queue with the help of algorithm?

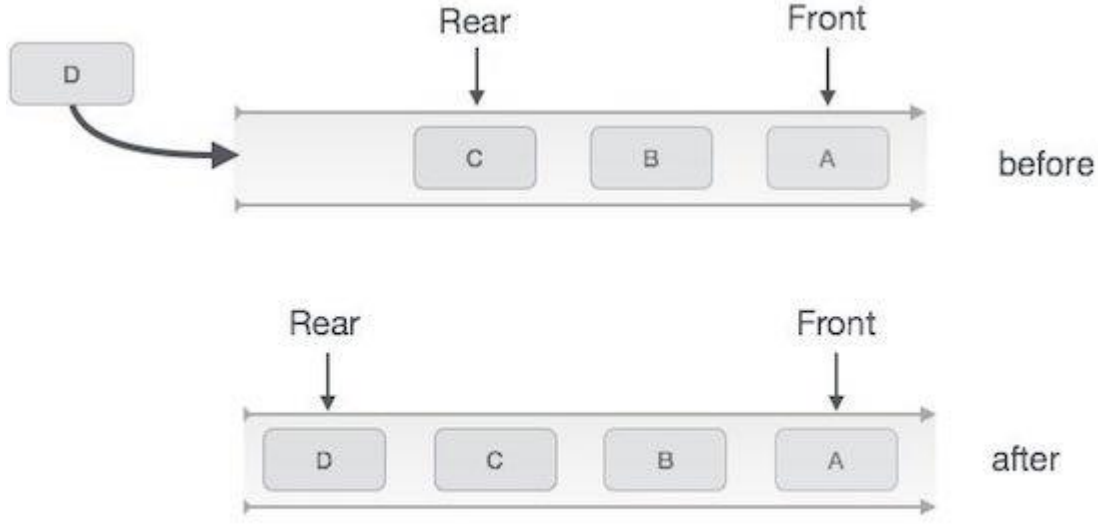
ANS) Enqueue Operation

Enqueue means inserting an element in the queue. In a normal queue at a ticket counter, where does a new person go and stand to become a part of the queue? The person goes and stands in the back. Similarly, a new element in a queue is inserted at the back of the queue.

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.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

procedure enqueue(data)

```
  if queue is full
    return overflow
  endif

  rear ← rear + 1
  queue[rear] ← data
  return true

end procedure
```

Implementation of enqueue() in C programming language −

Example

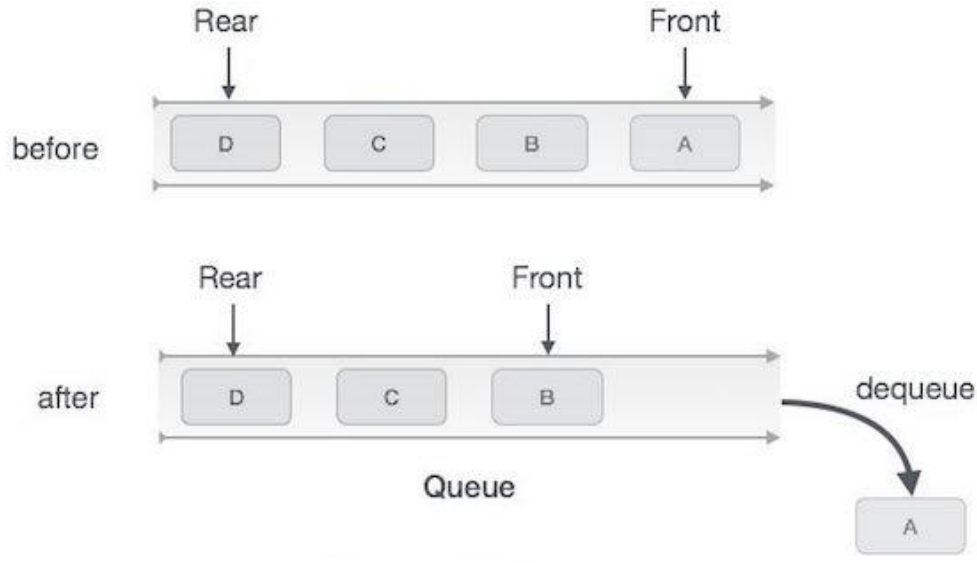
```
int enqueue(int data)
  if(isfull())
    return 0;

  rear = rear + 1;
  queue[rear] = data;

  return 1;
end procedure
```

Dequeue Operation

Dequeue means removing an element from the queue. Since queue follows the FIFO principle, we need to remove the element of the queue which was inserted at first. Naturally, the element inserted first will be at the front of the queue so we will remove the front element and let the element behind it be the new front element.

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.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

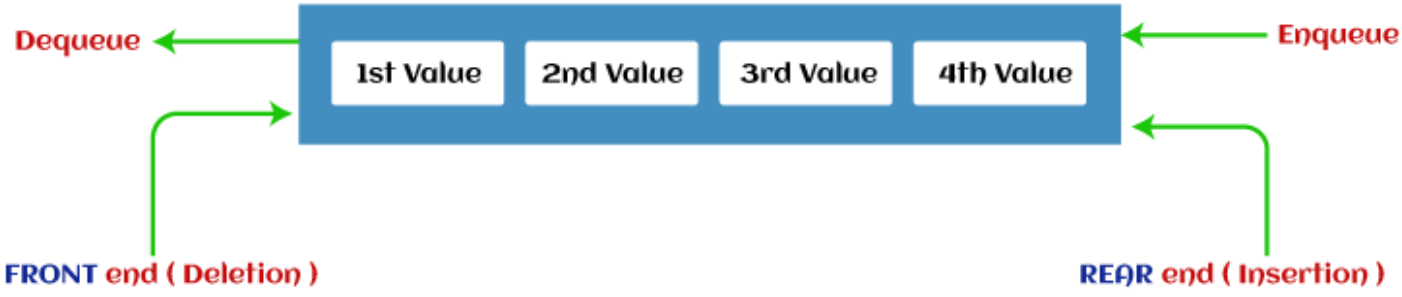
Implementation of dequeue () in C programming language −

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

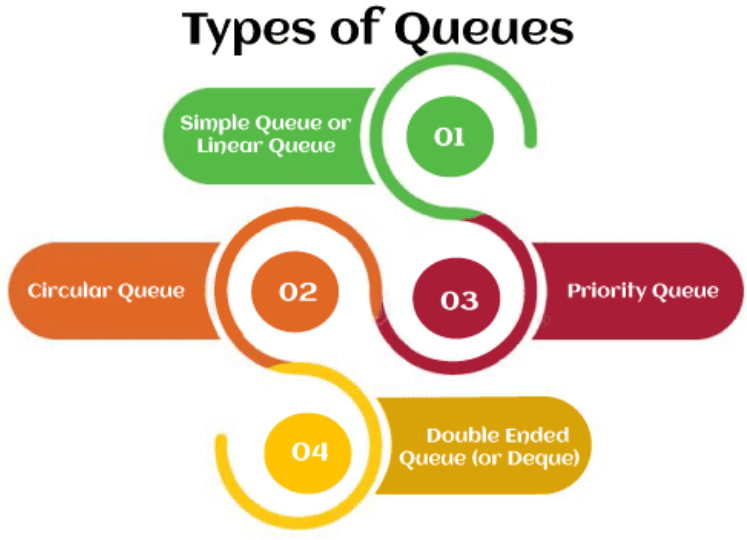
Queue

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the rear end or the tail of the queue, whereas the deletion is done from another end known as the front end or the head of the queue.



Types of Queues

There are four different types of queues that are listed as follows -



- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

Simple Queue or 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 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.