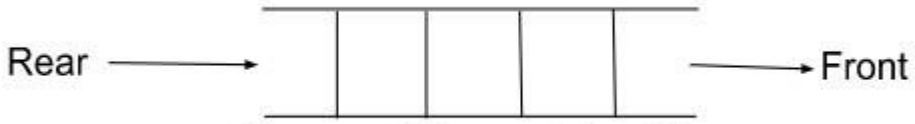
1. Queue

1.1 Definition and Queue Operations

A Queue is a linear data structure that resembles a queue in real life. You all have been part of some queue in school, the billing counter, or any other place, where the one entered first will exit first in the queue. Similarly, a queue in data structure also follows the FIFO principle, which defines First In First Out. The element inserted first in the queue will terminate first, compared to the remaining.

A queue has two endpoints and is open to both ends.

- **Front** − It is the end from where elements move out of the queue.
- **Rear** − It is the end from where elements are inserted in the queue.



Diagrammatic Representation of Queue

It can be implemented using a one-dimensional array, pointer, structures, and linked list. C++ library contains various built-in functions that help to manage the queue, its operations take place only at the front and rear ends.

PRIMITIVE OPERATIONS ON QUEUE

Basic Operations

Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue.

Queue uses two pointers − **front** and **rear**. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

Insertion operation: enqueue()

The *enqueue()* is a data manipulation operation that is used to insert elements into the stack. The following algorithm describes the enqueue() operation in a simpler way.

Algorithm

1 − START

2 – Check if the queue is full.
3 − If the queue is full, produce overflow error and exit.
4 − If the queue is not full, increment rear pointer to point the next empty space.
5 − Add data element to the queue location, where the rear is pointing.
6 − return success.
7 – END

Deletion Operation: dequeue()

The *dequeue()* is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.

Algorithm
1 – START
2 − Check if the queue is empty.
3 − If the queue is empty, produce underflow error and exit.
4 − If the queue is not empty, access the data where front is pointing.
5 − Increment front pointer to point to the next available data element.
6 − Return success.
7 – END

The peek() Operation

The peek() is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.

Algorithm
1 – START
2 – Return the element at the front of the queue
3 – END

The isFull() Operation

The isFull() operation verifies whether the stack is full.

Algorithm
1 – START
2 – If the count of queue elements equals the queue size, return true
3 – Otherwise, return false
4 – END

The isEmpty() operation

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1 – START
2 – If the count of queue elements equals zero, return true
3 – Otherwise, return false
4 – END

Applications

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:
1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served

1.2 Queue ADT and its Array Implementation

```
template <class KeyType>
class Queue
{
        //objects: a finite ordered list with zero or more elements.
        public:
        Queue(int MaxQueueSize=DefaultSize);
        //create an empty queue whose maximum size is MaxQueueSize
        Boolean IsFullQ();
        //if (number of elements in queue== MaxQueueSize) return TRUE
        //else return FALSE
        void Add(const Keytype& item);
        //if (IsFullQ()) then QueueFull()
        //else insert item at rear of queue
        Boolean IsEmptyQ();
        //if number of elements in the queue is equal to 0 then return TRUE
        //else return FALSE
        Keytype* Delete(KeyType&);
        //if (IsEmpty()) then QueueEmpty() and return 0
        //else remove the item at front of queue and return a pointer to it.
        ~Queue() {}

};
```

IMPLEMENTATION OF QUEUE DATA STRUCTURE

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.
Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

```
/* Below program is written in C++ language */
#include <iostream>

using namespace std;

const int MAX_SIZE = 100;

class Queue {
private:
   int arr[MAX_SIZE];
   int front;
   int rear;

public:
   Queue() {
      front = -1;
      rear = -1;
   }

   void enqueue(int item) {
      if (rear == MAX_SIZE - 1) {
         cout << "Queue is full. Unable to enqueue " << item << endl;
         return;
      }

      if (isEmpty())
         front = 0;

      arr[++rear] = item;
      cout << item << " enqueued to the queue." << endl;
   }

   void dequeue() {
      if (isEmpty()) {
         cout << "Queue is empty. Unable to dequeue." << endl;
         return;
      }
```

```
        int item = arr[front++];
        cout << item << " dequeued from the queue." << endl;

        if (front > rear)
            front = rear = -1;
    }

    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty. No element to peek." << endl;
            return -1;
        }

        return arr[front];
    }

    bool isEmpty() {
        return front == -1 && rear == -1;
    }

    bool isFull() {
        return rear == MAX_SIZE - 1;
    }

    int size() {
        if (isEmpty())
            return 0;
        else
            return rear - front + 1;
    }
};

int main() {
    Queue myQueue;

    myQueue.enqueue(10);
    myQueue.enqueue(20);
    myQueue.enqueue(30);
    myQueue.enqueue(40);
    myQueue.enqueue(50);
    myQueue.enqueue(60);  // Queue is full. Unable to enqueue 60

    cout << "Front element: " << myQueue.peek() << endl;  // Front element: 10
    cout << "Queue size: " << myQueue.size() << endl;     // Queue size: 5
```

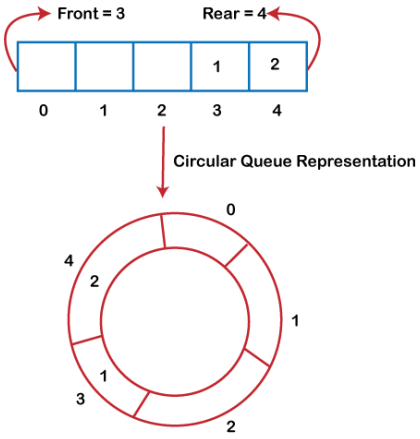
```
    myQueue.dequeue();
    myQueue.dequeue();
    myQueue.dequeue();

    cout << "Front element after dequeue: " << myQueue.peek() << endl;  // Front element after dequeue: 40
    cout << "Queue size after dequeue: " << myQueue.size() << endl;  // Queue size after dequeue: 2

    return 0;
}
```

To implement approach [A], you simply need to change the dequeue method, and include a for loop which will shift all the remaining elements by one position.

```
// function dequeue - to remove data from queue
int Queue :: dequeue()
{
        int i, item;
        // if queue is empty
         if(rear == -1)
        {
                cout<<"Empty queue";  return 0;
        }
        else
        {
                item = a[front];
                //shifting all other elements
        for(i=0; i<rear; i++)
        {
                a[i] = a[i+1];
        }
        rear--;  return item;
}
```

1.3 Circular Queue and its Array Implementation

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0^{th} position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of

a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- front== rear + 1;

Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE

SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

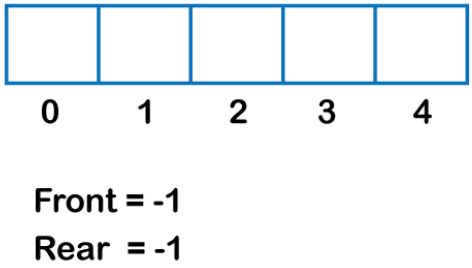
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

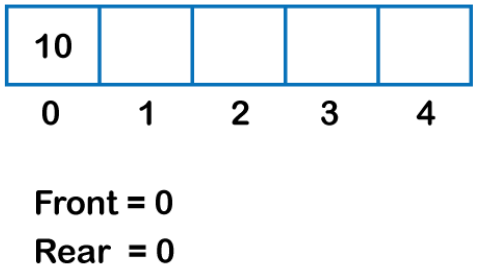
Step 2: SET VAL = QUEUE[FRONT]

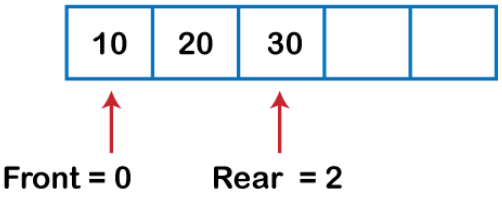
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

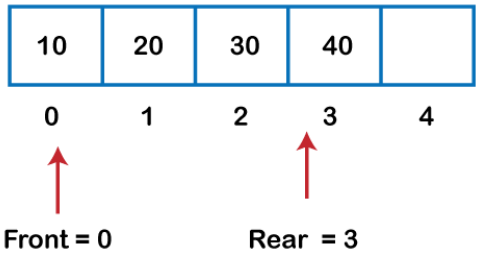
Step 4: EXIT

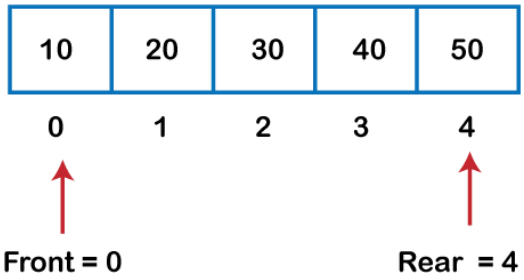
Let's understand the enqueue and dequeue operation through the diagrammatic representation.

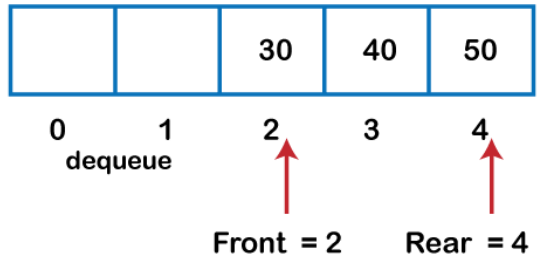


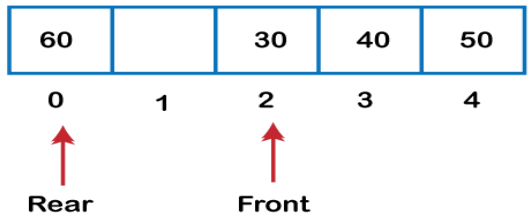


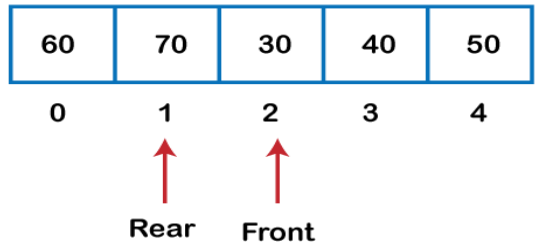












C++ program to implement a Circular Queue using an Array

```
#include <iostream>

const int MAX_SIZE = 5; // Adjust the maximum size as needed

class CircularQueue {
private:
    int arr[MAX_SIZE];
    int front;
    int rear;

public:
    CircularQueue() {
        front = -1;
        rear = -1;
    }

    bool isEmpty() {
        return (front == -1 && rear == -1);
    }

    bool isFull() {
        return (front == (rear + 1) % MAX_SIZE);
    }

    void enqueue(int item) {
        if (isFull()) {
            std::cout << "Queue is full. Unable to enqueue " << item << std::endl;
            return;
        }

        if (isEmpty())
            front = rear = 0;
        else
            rear = (rear + 1) % MAX_SIZE;

        arr[rear] = item;
        std::cout << item << " enqueued to the queue." << std::endl;
    }

    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty. Unable to dequeue." << std::endl;
            return;
        }
```

```
        int item = arr[front];
        if (front == rear)
            front = rear = -1;
        else
            front = (front + 1) % MAX_SIZE;

        std::cout << item << " dequeued from the queue." << std::endl;
    }

    int peek() {
        if (isEmpty()) {
            std::cout << "Queue is empty. No element to peek." << std::endl;
            return -1;
        }

        return arr[front];
    }

    int size() {
        if (isEmpty())
            return 0;
        else if (rear >= front)
            return (rear - front + 1);
        else
            return (MAX_SIZE - front + rear + 1);
    }
};

int main() {
    CircularQueue myQueue;

    myQueue.enqueue(10);
    myQueue.enqueue(20);
    myQueue.enqueue(30);
    myQueue.enqueue(40);
    myQueue.enqueue(50);
    myQueue.enqueue(60);  // Queue is full. Unable to enqueue 60

    std::cout << "Front element: " << myQueue.peek() << std::endl;  // Front element: 10
    std::cout << "Queue size: " << myQueue.size() << std::endl;     // Queue size: 5

    myQueue.dequeue();
    myQueue.dequeue();
    myQueue.dequeue();
```

```
    std::cout << "Front element after dequeue: " << myQueue.peek() << std::endl;  // Front element after dequeue: 40
    std::cout << "Queue size after dequeue: " << myQueue.size() << std::endl;     // Queue size after dequeue: 2

    return 0;
}
```

In this implementation, the circular queue uses an array to store the elements. The front and rear variables keep track of the positions of the front and rear of the queue, respectively. The modulo operator % is used to ensure circular wrapping around the array.

The program demonstrates enqueuing, dequeuing, peeking at the front element, and checking the size of the circular queue.

1.4 Double Ended Queue and Priority Queue

Priority Queue is an extension of queue with following properties.
1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.
Generally, the value of the element itself is considered for assigning the priority. For example: The element with the highest value is considered as the highest priority element. However, in other case, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priority according to our need.

Implementation of Priority Queue
Priority queue can be implemented using an array, a linked list, a heap data structure or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Priority Queue Operations
Basic operations of a priority queue are inserting, removing and peeking elements.

2. Linked List

2.1 List - Definition and List Operations

A **list** is a collection of homogeneous set of elements or objects. If the size of list is not fixed at compile time and grow or shrink at runtime then it is called **static list** however if the size of the list is fixed at compile time and grow or shrink at runtime then it is called **dynamic list.**

The beginning of the list is called head and the end of the list is called the tail.

Example: Stack and queue are special type of list which can be implemented using array or linked list.

List Operations

- **Add or Insert Operation**: In the Add or Insert operation, a new item (of any data type) is added in the List Data Structure or Sequence object.
- **Replace or reassign Operation**: In the Replace or reassign operation, the already existing value in the List object is changed or modified. In other words, a new value is added at that particular index of the already existing value.
- **Delete or remove Operation**: In the Delete or remove operation, the already present element is deleted or removed from the Dictionary or associative array object.
- **Find or Lookup or Search Operation**: In the Find or Lookup operation, the element stored in that List Data Structure or Sequence object is fetched.

2.2 List ADT and its Array Implementation

Let L be a list and p indicates the position of list L, then the following basic operations can be performed on the list L.

- Create (L): Create a list L.
- Insert(X,P,L): Insert X at position P in the list L.
- Find(X,L): Return the position of first occurrence of element X in List L.
- Retrieve(P,L): Return the element at position P in List L.
- Delete(P,L): Delete the element at position P of list L.
- Next(P,L): Return the element at position P+1 in list L.
- Previous(P,L): Return the element at position P-1 in list L.
- MakeEmpty(L): Make the List L an empty list.
- First(L): Return the first position on list L.
- Print(L): Print the elements of list L.

Implementation of List

There are two ways to implement the list

1. Contiguous List-Array implementation of list (Static)
2. Linked list-Pointer implementation of list (Dynamic)

Array Implementation of List

An **array** is a collection of elements of same type placed in contiguous memory location and can be accessed individually using index to a unique given name. It means array can contain one type of data only, either all integer, all characters or all floating points numbers. An array can be single dimensional or multi-dimensional.

In array implementation of list, we keep the elements of list in an array and use and integer to count the number of elements in the list. The counter integer can also be used to locate the end of the list within that array. Elements of the list can be accessed with an index.

Advantage of array implementation

1. Random access is possible.
2. Array implementation is easier as compared to linked list implementation.

Disadvantage of array implementation

1. Elements in an array are always stored in contiguous memory so inserting and deleting an element in an array may require all elements to be shifted.
2. The size of array is fixed, so it does not provide good memory utilization.

The following C++ program shows how list can be implemented using an array.

```
#include <iostream>

const int MAX_SIZE = 100; // Maximum size of the static list

using namespace std;

class StaticList {
private:
   int arr[MAX_SIZE];
   int length;

public:
   StaticList() {
      length = 0; // Initialize the length to zero
   }
```

```
    void insert(int value) {
        if (length >= MAX_SIZE) {
            cout << "List is full. Cannot insert more elements.\n";
            return;
        }
        arr[length] = value;
        length++;
    }

    void remove(int value) {
        int index = search(value);
        if (index == -1) {
            cout << "Element not found in the list.\n";
            return;
        }
        for (int i = index; i < length - 1; i++) {
            arr[i] = arr[i + 1];
        }
        length--;
    }

    int search(int value) {
        for (int i = 0; i < length; i++) {
            if (arr[i] == value) {
                return i; // Return the index if the element is found
            }
        }
        return -1; // Return -1 if the element is not found
    }

    void display() {
        if (length == 0) {
            cout << "List is empty.\n";
            return;
        }
        cout << "List elements: ";
        for (int i = 0; i < length; i++) {
            cout << arr[i] << " ";
        }
        cout << "\n";
    }
};
```

```
int main() {
    StaticList myList;

    myList.display(); // Output: List is empty.

    myList.insert(10);
    myList.insert(20);
    myList.insert(30);
    myList.insert(40);

    myList.display(); // Output: List elements: 10 20 30 40

    myList.remove(20);

    myList.display(); // Output: List elements: 10 30 40

    int searchResult = myList.search(30);
    if (searchResult != -1) {
        cout << "Element found at index " << searchResult << ".\n";
    } else {
        cout << "Element not found in the list.\n";
    }

    return 0;
}
```

The following C program shows how list can be implemented using an array.

```
#include<stdio.h>
 #include<conio.h> void main()
  {
  int a[5]={1,2,3,4,5};
  int b[5]= {2,4,6,8,10} ;
  int c[5];

  for( int i=0;i<=4;i++)
  {
          c[i]=a[i]+b[i];
  }

  for(int j=0;j<=4;j-H-)
  {
          printf("%d\t",c[j]);
  }
  getch();
```

i. Algorithm to insert a new element in a contiguous list

Consider we have an array A[max] of size max whose upper bound is represented by ub. It is assumed that ub is -1 for an empty array.

1. If ub = max-1
   a. Display "full"
   b. Exit
2. Read position from user
3. If position > ub+l
   a. Display "out of range"
   b. Exit
4. Read the data item to be inserted.
5. Shift element

```
right T=ub;
While (T>=position)
A[T+1]
=A[T];
T=T-1;
}
```

6. ub=ub +1;
7. A[pos] =item;
8. Exit

ii. Algorithm for deletion of an item in **a contiguous list**

1. If ub = -1
   a. Display "array is empty"
   b. Exit
2. Read position to be deleted.
3. If(position >=0 && position <=ub)
   a. Shift element left

```
T=position;
While (T< ub)
 {
      A[T]
      =A[T+1];
      T=T+1
}
```

   b. ub = ub -1;
      Else
   a. Display "index out of range"
4. Exit

Differences between the array and linked list:

Array	Linked list
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 takes less time while performing any operation like insertion, deletion, etc.
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.

2.3 Linked List - Definition and its Operations

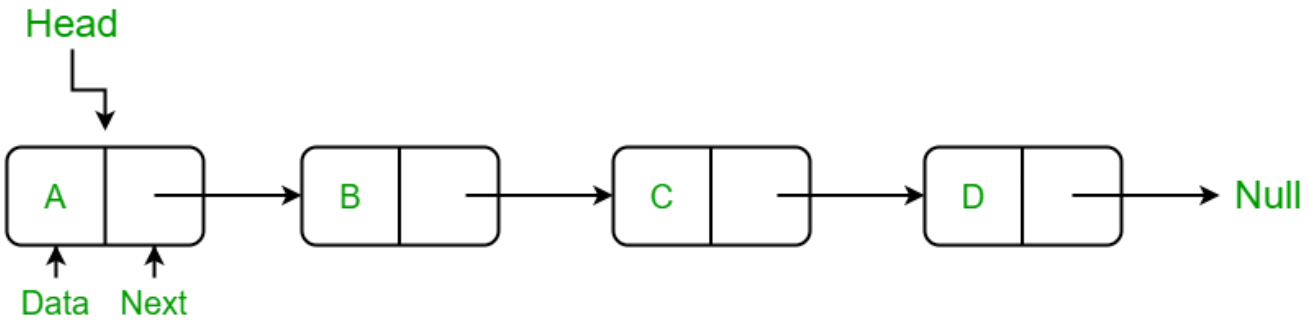
A linked list is a collection of "nodes" connected together via links. These nodes consist of the data to be stored and a pointer to the address of the next node within the linked list. In the case of arrays, the size is limited to the definition, but in linked lists, there is no defined size. Any amount of data can be stored in it and can be deleted from it.

There are three types of linked lists −

- **Singly Linked List** − The nodes only point to the address of the next node in the list.
- **Doubly Linked List** − The nodes point to the addresses of both previous and next nodes.
- **Circular Linked List** − The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked.

2.4 Singly Linked List

A **singly linked list** is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.



Singly Linked List

Basic Operations

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below −

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Singly Linked List ADT

```
ADT SinglyLinkedList {
  // Operations:

  void initialize()    // Create an empty linked list

  bool isEmpty()    // Check if the linked list is empty

  void insertAtBeginning(value)    // Insert a new node at the beginning of the list

  void insertAtEnd(value)    // Insert a new node at the end of the list

  void insertAtPosition(value, position)    // Insert a new node at a specific position in the list

  void deleteFromBeginning()    // Delete the first node of the list
```

void deleteFromEnd() // Delete the last node of the list

void deleteFromPosition(position) // Delete a node from a specific position in the list

bool search(value) // Search for a specific value in the list and return true if found, false otherwise

value getValueAtPosition(position) // Get the value of the node at a specific position in the list

int length() // Get the length (number of nodes) of the list

void display() // Print the elements of the list

}

This ADT provides the basic operations to create, manipulate, and traverse a singly linked list. Implementations of these operations will involve handling the internal structure of the linked list, such as maintaining the head pointer, updating next pointers, and managing memory allocation and deallocation.

Implementation of Singly Linked List in C++

```
#include <iostream>

using namespace std;

// Node structure
struct Node {
    int data;
    Node* next;
};

// Singly Linked List class
class SinglyLinkedList {
private:
    Node* head;

public:
    // Constructor
    SinglyLinkedList() {
        head = nullptr;
    }

    // Destructor
    ~SinglyLinkedList() {
        // Delete all nodes to free memory
        Node* current = head;
```

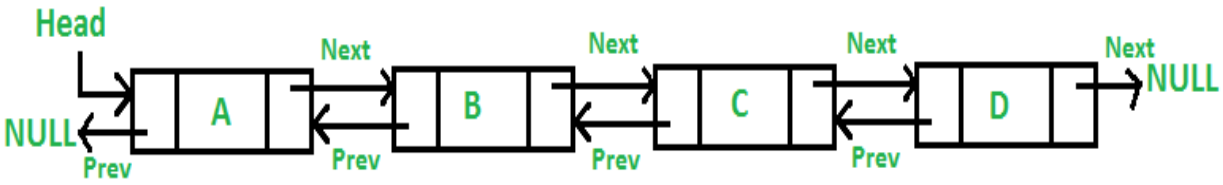
```
    while (current != nullptr) {
        Node* temp = current;
        current = current->next;
        delete temp;
    }
}

// Insert a new node at the beginning of the list
void insertAtBeginning(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}

// Insert a new node at the end of the list
void insertAtEnd(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* current = head;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }
}

// Delete the first node of the list
void deleteFromBeginning() {
    if (head == nullptr) {
        return;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
}

// Delete the Last node of the list
void deleteFromEnd() {
    if (head == nullptr) {
        return;
```

```
        }

        // If the list has only one node
        if (head->next == nullptr) {
            delete head;
            head = nullptr;
            return;
        }

        Node* current = head;
        Node* prev = nullptr;

        // Traverse to the last node
        while (current->next != nullptr) {
            prev = current;
            current = current->next;
        }

        // Update the previous node's next pointer and delete the last node
        prev->next = nullptr;
        delete current;
    }

    // Print the elements of the list
    void display() {
        Node* current = head;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

int main() {
    SinglyLinkedList myList;

    myList.insertAtBeginning(5); // Insert 5 at the beginning
    myList.insertAtBeginning(4); // Insert 4 at the beginning
    myList.insertAtBeginning(3); // Insert 3 at the beginning
    myList.insertAtBeginning(2); // Insert 2 at the beginning
    myList.insertAtEnd(6);       // Insert 6 at the end

    myList.display(); // Output: 2 3 4 5 6

    myList.deleteFromEnd(); // Delete the last node
```

```
    myList.deleteFromBeginning(); // Delete the first node

    myList.display(); // Output: 3 4 5

    return 0;
}
```

2.5 Doubly Linked List and Circular Linked List

A **doubly linked list** (DLL) is a special type of linked list in which each node contains a pointer to the previous node as well as the next node of the linked list.



Advantages of Doubly Linked List over the singly linked list:

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

Disadvantages of Doubly Linked List over the singly linked list:

- Every node of DLL Requires extra space for a previous pointer.
- All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

IMPLEMENTATION OF DOUBLY LINKED LIST

```
#include <iostream>

using namespace std;

// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;
};

// Doubly Linked List class
class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    // Constructor
    DoublyLinkedList() {
        head = nullptr;
        tail = nullptr;
    }

    // Destructor
    ~DoublyLinkedList() {
        // Delete all nodes to free memory
        Node* current = head;
        while (current != nullptr) {
            Node* temp = current;
            current = current->next;
            delete temp;
        }
    }
    // Insert a new node at the beginning of the list
    void insertAtBeginning(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->prev = nullptr;
        newNode->next = head;

        if (head != nullptr) {
            head->prev = newNode;
```

```
    } else {
        tail = newNode;
    }

    head = newNode;
}

// Insert a new node at the end of the list
void insertAtEnd(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->prev = tail;
    newNode->next = nullptr;

    if (tail != nullptr) {
        tail->next = newNode;
    } else {
        head = newNode;
    }

    tail = newNode;
}

// Delete the first node of the list
void deleteFromBeginning() {
    if (head == nullptr) {
        return;
    }

    Node* temp = head;
    head = head->next;

    if (head != nullptr) {
        head->prev = nullptr;
    } else {
        tail = nullptr;
    }

    delete temp;
}

// Delete the last node of the list
void deleteFromEnd() {
    if (tail == nullptr) {
        return;
    }
```

```
        Node* temp = tail;
        tail = tail->prev;

        if (tail != nullptr) {
            tail->next = nullptr;
        } else {
            head = nullptr;
        }

        delete temp;
    }

    // Print the elements of the list
    void display() {
        Node* current = head;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

int main() {
    DoublyLinkedList myList;

    myList.insertAtBeginning(3); // Insert 3 at the beginning
    myList.insertAtBeginning(2); // Insert 2 at the beginning
    myList.insertAtEnd(4);       // Insert 4 at the end

    myList.display(); // Output: 2 3 4

    myList.deleteFromBeginning(); // Delete the first node

    myList.display(); // Output: 3 4

    myList.deleteFromEnd(); // Delete the last node

    myList.display(); // Output: 3

    return 0;
}
```

IMPLEMENTATION OF CIRCULAR LINKED LIST

In a circular linked list, the last node's `next` pointer would not be `nullptr` but instead point back to the first node (the `head`). This creates a circular structure where traversal can loop indefinitely.

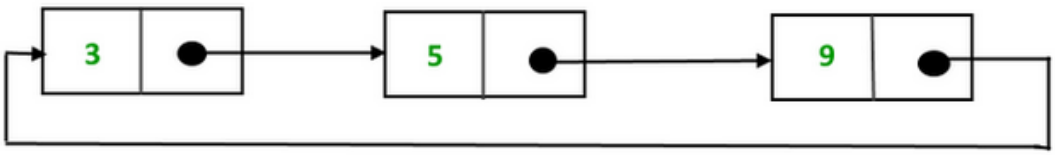
Advantages of Circular Linked Lists:

- 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 a queue.

Disadvantages of circular linked list:

- Compared to singly linked lists, circular lists are more complex.
- Reversing a circular list is more complicated than singly or doubly reversing a circular list.
- It is possible for the code to go into an infinite loop if it is not handled carefully.

To modify the code to implement a circular linked list from Singly Linked List, you would need to make the following changes:



In the Node struct, change the next pointer type from Node* to Node* and initialize it to nullptr in the constructor.

```
struct Node {
   int data;
   Node* next;

   Node(int value) {
      data = value;
      next = nullptr;
   }
};
```

In the SinglyLinkedList class, modify the insertAtEnd function to update the next pointer of the last node (tail) to point back to the head, thus creating a circular link.

```
void insertAtEnd(int value) {
```

```
    Node* newNode = new Node(value);

    if (head == nullptr) {
        head = newNode;
        newNode->next = head; // Make the new node point to itself to form a circular link
    } else {
        Node* current = head;
        while (current->next != head) {
            current = current->next;
        }
        current->next = newNode;
        newNode->next = head; // Make the new node point back to the head to form a circular link
    }
}
```

Update the deleteFromEnd function to handle circular linked lists properly.

```
void deleteFromEnd() {
    if (head == nullptr) {
        return;
    }

    // If the list has only one node
    if (head->next == head) {
        delete head;
        head = nullptr;
        return;
    }

    Node* current = head;
    Node* prev = nullptr;

    // Traverse to the last node
    while (current->next != head) {
        prev = current;
        current = current->next;
    }

    // Update the previous node's next pointer to the head and delete the last node
    prev->next = head;
    delete current;
}
```

By making these changes, the singly linked list implementation will be transformed into a circular linked list, where the next pointer of the last node points back to the head node, forming a circular structure.

To modify the code to implement a circular linked list from Doubly Linked List, you would need to make the following changes:

In the `Node` struct, change the `next` pointer type from `Node*` to `Node*` and initialize it to `nullptr` in the constructor.

```
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int value) {
        data = value;
        prev = nullptr;
        next = nullptr;
    }
};
```

In the `DoublyLinkedList` class, modify the `insertAtEnd` and `deleteFromEnd` functions to update the `next` pointer of the last node (`tail`) to point to the `head` instead of `nullptr`, creating the circular link.

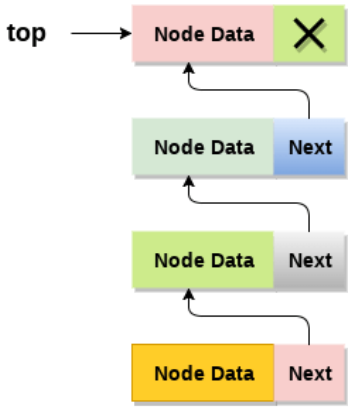
```
void insertAtEnd(int value) {
    Node* newNode = new Node(value);
    newNode->prev = tail;
    newNode->next = head;

    if (tail != nullptr) {
        tail->next = newNode;
    } else {
        head = newNode;
    }

    head->prev = newNode; // Update the previous pointer of the head to the new node
    tail = newNode; // Update the tail to the new node
}

void deleteFromEnd() {
    if (tail == nullptr) {
```

```
        return;
    }

    Node* temp = tail;
    tail = tail->prev;
    tail->next = head; // Update the next pointer of the new tail to the head

    if (head != nullptr) {
        head->prev = tail;
    }

    delete temp;
}
```

By making these changes, you would transform the doubly linked list implementation into a circular doubly linked list.

2.6 Linked Implementation of Stack and Queue

Implementing Stack using Linked List



Stack

Stack is a collection of items in which addition of new items and deletion of existing items are done from only one end known as top of stack. Since, linked list is also a list of items, concepts of stack can be implemented using linked lists instead of an array.

1. The two operations that we apply on a stack are push (inserting item on the top) and pop (deleting item at the top).
2. The push operation is similar to adding a new node at the beginning of the list (insertHead()).
3. A stack items can be accessed only through its top, and a list items can be accessed from the pointer to the first node.
4. Similarly, removing the first node from a linked list (deleteHead()) is similar to popping an item from a stack. In both cases, the only immediately accessible item of a collection is removed from that collection, and the next item becomes immediately accessible one.
5. The first node of the list is the top of the stack

Linked Implementation of Stack in C++

```
#include <iostream>

// Node struct to represent individual elements of the stack
struct Node {
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

// Stack class
class Stack {
private:
    Node* top; // Pointer to the top element of the stack

public:
    Stack() {
        top = nullptr;
    }

    // Function to check if the stack is empty
    bool isEmpty() {
        return top == nullptr;
    }

    // Function to push an element onto the stack
    void push(int value) {
        Node* newNode = new Node(value);
        newNode->next = top;
        top = newNode;
        std::cout << value << " pushed to the stack." << std::endl;
    }

    // Function to pop the top element from the stack
    void pop() {
        if (isEmpty()) {
            std::cout << "Stack is empty. Cannot pop an element." << std::endl;
            return;
        }
        Node* temp = top;
```

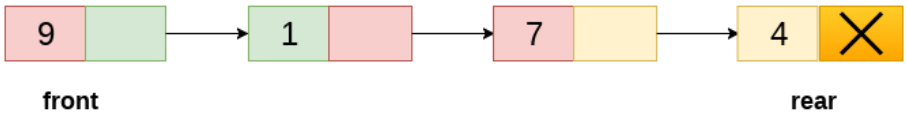
```
        top = top->next;
        int poppedValue = temp->data;
        delete temp;
        std::cout << poppedValue << " popped from the stack." << std::endl;
    }

    // Function to get the top element of the stack without removing it
    int peek() {
        if (isEmpty()) {
            std::cout << "Stack is empty. No element to peek." << std::endl;
            return -1; // Assuming -1 represents an empty stack
        }
        return top->data;
    }
};

// Example usage
int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Top element: " << stack.peek() << std::endl;

    stack.pop();
    stack.pop();
    stack.pop();

    if (stack.isEmpty()) {
        std::cout << "Stack is empty." << std::endl;
    } else {
        std::cout << "Stack is not empty." << std::endl;
    }

    return 0;
}
```

In this implementation, the Node struct is used to represent individual elements of the stack. Each node contains a data value and a pointer to the next node.

The Stack class manages the stack operations, including isEmpty(), push(value), pop(), and peek().

The main() function demonstrates example usage, where elements are pushed onto the stack, popped, and the top element is accessed using peek().

Implementing Queue using Linked List



Linked Queue

A queue is a special type of list that allows insertion at one end, called the front of queue, and deletion at other end, called the rear of queue.
i. To implement a queue in a linked list, we make the first node of the list as the front and the last node as the rear. So, we will use two pointer variables – front and rear. 'front' will point to the beginning of the list (like start) while 'rear' will point to the last node of the list .
ii. Initially, when the queue is empty, both rear and front will be NULL.
iii. To enqueue a new item, we insert it after the rear node and update rear to point to new node. This function will work similarly as insertTail(), because both are the process to add new item after the last node. In the case of Queue, we insert a new item after the node pointed by 'rear'.
iv. To dequeue an item, we remove the front node and update front to point to the next node. This function will work similarly as deleteHead() of general list or pop() operation of stack.

Linked Implementation of Queue in C++

```
#include <iostream>

// Node struct to represent individual elements of the queue
struct Node {
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

// Queue class
class Queue {
private:
    Node* front; // Pointer to the front of the queue
    Node* rear;  // Pointer to the rear of the queue

public:
    Queue() {
```

```
        front = nullptr;
        rear = nullptr;
    }

    // Function to check if the queue is empty
    bool isEmpty() {
        return front == nullptr;
    }

    // Function to enqueue (add) an element to the rear of the queue
    void enqueue(int value) {
        Node* newNode = new Node(value);
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        std::cout << value << " enqueued to the queue." << std::endl;
    }

    // Function to dequeue (remove) an element from the front of the queue
    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty. Cannot dequeue an element." << std::endl;
            return;
        }
        Node* temp = front;
        int dequeuedValue = temp->data;
        front = front->next;
        if (front == nullptr) {
            rear = nullptr;
        }
        delete temp;
        std::cout << dequeuedValue << " dequeued from the queue." << std::endl;
    }

    // Function to get the front element of the queue without removing it
    int peek() {
        if (isEmpty()) {
            std::cout << "Queue is empty. No element to peek." << std::endl;
            return -1; // Assuming -1 represents an empty queue
        }
        return front->data;
    }
};
```

```
// Example usage
int main() {
    Queue queue;

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);

    std::cout << "Front element: " << queue.peek() << std::endl;

    queue.dequeue();
    queue.dequeue();
    queue.dequeue();

    if (queue.isEmpty()) {
        std::cout << "Queue is empty." << std::endl;
    } else {
        std::cout << "Queue is not empty." << std::endl;
    }

    return 0;
}
```

The Queue class manages the queue operations and maintains two pointers: front and rear. The front pointer points to the front of the queue, and the rear pointer points to the rear of the queue.

The queue operations implemented are:

isEmpty(): Checks if the queue is empty.
enqueue(value): Adds an element with the given value to the rear of the queue.
dequeue(): Removes and returns the element from the front of the queue.
peek(): Returns the value of the front element without removing it.

In the main() function, you can see an example usage of the queue, where elements are enqueued, dequeued, and the front element is accessed using peek().