C++ Program to Implement Queue using Array

The following program demonstrates how we can implement a queue using array in C++:

```
// C++ Program to implement a queue using array
#include <iostream>
using namespace std;

// defining the max size of the queue
#define MAX_SIZE 100

// Implement the queue data structure
class Queue {
public:
    int front;
    int rear;
    int arr[MAX_SIZE];

    // initializing pointers in the constructor
    Queue(): front(-1), rear(-1) {}

    // Function to check if the queue is empty or not
    bool isEmpty() { return front == -1 || front > rear; }

    // Function to check if the queue is full or not
    bool isFull() { return rear == MAX_SIZE - 1; }

    // Function to get the front element of the queue
    int getFront()
    {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return -1;
        }
        return arr[front];
    }

    // Function to get the rear element of the queue
    int getRear()
    {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return -1;
        }
        return arr[rear];
    }

    // Function to enqueue elements from the queue
```

```
    void enqueue(int val)
    {
        // Check overflow condition
        if (isFull()) {
            cout << "Queue is full" << endl;
            return;
        }
        // if queue is empty, set front to 0
        if (isEmpty())
            front = 0;

        rear++;
        arr[rear] = val;
    }

    // Function to dequeue elements from the queue
    int dequeue()
    {
        // Check underflow condition
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return -1;
        }
        int ans = arr[front];
        front++;

        // if queue becomes empty, reset both pointers
        if (isEmpty())
            front = rear = -1;

        return ans;
    }

    // Display function to print the queue
    void display()
    {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return;
        }
        cout << "Queue:  ";
        for (int i = front; i <= rear; i++) {
            cout << arr[i] << " ";
        }

        cout << endl;
    }
};
```

```cpp
int main()
{
    // Created Queue of size 5
    Queue q;

    // Enqueueing elements
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    // Displaying status of the queue after enqueuing
    cout << "\nAfter Enqueueing:" << endl;

    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;

    q.display();

    // Enqueueing more elements
    q.enqueue(4);
    q.enqueue(5);

    // Displaying the updated queue
    q.display();

    // Enqueueing one more element to demonstrate overflow
    // condition
    q.enqueue(6);

    // Dequeueing elements
    cout << "\nDequeueing elements:" << endl;
    cout << "Dequeued element: " << q.dequeue() << endl;
    cout << "Dequeued element: " << q.dequeue() << endl;

    // Displaying status of the queue after dequeueing
    cout << "\nAfter Dequeueing:" << endl;

    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;

    q.display();

    return 0;
}
```

Output

```
After Enqueueing:
Front element: 1
Rear element: 3
Queue:  1 2 3
Queue:  1 2 3 4 5

Dequeueing elements:
Dequeued element: 1
Dequeued element: 2

After Dequeueing:
Front element: 3
Rear element: 6
Queue:  3 4 5 6
```

C++ Program to Implement Circular Queue Using Array

C++

```cpp
// C++ program to implement the circular queue using array
#include <bits/stdc++.h>

// defining the max size of the queue
#define MAX_SIZE 5

using namespace std;

// class that represents queue
class Queue {
public:
    // index pointers and data array
    int front, rear;
    int arr[MAX_SIZE];

    // constructor to initialize the index pointers
    Queue() { front = rear = 0; }

    // checking if queue is empty
    bool isEmpty()
    {
        if (front == rear)
            return true;
        return false;
    }

    // checking if the queue is full
    bool isFull()
```

```
    {
        if ((rear + 1) % MAX_SIZE == front)
            return true;
        return false;
    }

    // enqueue operation
    void enqueue(int val)
    {
        if (this->isFull()) {
            printf("Queue Overflow!\n");
            return;
        }
        rear = (rear + 1) % MAX_SIZE;
        arr[rear] = val;
    }

    // dequeue operation
    void dequeue()
    {
        if (this->isEmpty()) {
            printf("Queue Underflow!\n");
            return;
        }
        front = (front + 1) % MAX_SIZE;
    }

    // peek function
    int peek()
    {
        if (this->isEmpty()) {
            printf("Queue is Empty!\n");
            return -1;
        }
        return arr[(front + 1) % MAX_SIZE];
    }

    // utility to print queue
    void print()
    {
        if (this->isEmpty())
            return;
        for (int i = (front + 1) % MAX_SIZE; i < rear;
             i = (i + 1) % MAX_SIZE) {

            printf("%d ", arr[i]);
        }
        cout << arr[rear];
```

```
    }
};

// driver code
int main()
{
    Queue q;

    q.enqueue(11);
    q.enqueue(11);
    q.enqueue(11);
    q.enqueue(11);
    q.enqueue(11);
    q.enqueue(11);

    q.dequeue();

    q.dequeue();
    q.enqueue(123);

    q.print();

    return 0;
}
```

Output

```
Queue Overflow!

Queue Overflow!

123
```

C++ Program to Implement Queue using Linked List

C++

```
#include <iostream>
using namespace std;

// Define Node structure
struct Node {
    // The data held by the node
    int data;
    // Pointer to the next node in the list
    Node* next;
};
```

```cpp
// Define Queue class
class Queue {
    // Pointer to the front node of the queue
    Node* front;
    // Pointer to the rear node of the queue
    Node* rear;

public:
    // Constructor initializes an empty queue
    Queue()
        : front(nullptr)
        , rear(nullptr)
    {
    }

    // Enqueue adds an element at the end of the queue
    void enqueue(int x)
    {
        // Create a new node with given data
        Node* newNode = new Node{ x, nullptr };
        // If the queue is empty
        if (rear == nullptr) {
            // Both front and rear point to the new node
            front = rear = newNode;
        }
        else {
            // Link the new node at the end of the queue
            rear->next = newNode;
            // Update rear to the new node
            rear = newNode;
        }
    }

    // Dequeue removes the element at the front of the queue
    void dequeue()
    {
        // If the queue is empty, do nothing
        if (front == nullptr)
            return;
        // Temporary pointer to the front node
        Node* temp = front;
        // Move front to the next node
        front = front->next;
        // If the queue is now empty
        if (front == nullptr)
            // Set rear to nullptr
            rear = nullptr;
        // Delete the old front node
```

```cpp
        delete temp;
    }

    // Peek returns the front element of the queue
    int peek()
    {
        if (!isEmpty())
            return front->data;
        else
            throw runtime_error("Queue is empty");
    }

    // isEmpty checks if the queue is empty
    bool isEmpty()
    {
        // Return true if front is nullptr
        return front == nullptr;
    }
};

// Main function
int main()
{
    // Create a queue
    Queue q;
    // Enqueue elements into the queue
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    // Output the front element
    cout << "Front element is: " << q.peek() << endl;
    // Dequeue the front element
    q.dequeue();
    // Output the new front element
    cout << "Front element is: " << q.peek() << endl;
    // Dequeue the remaining elements
    q.dequeue();
    q.dequeue();
    // Check if the queue is empty and output the result
    cout << "Queue is empty: " << q.isEmpty() << endl;
    return 0;
}
```

Output

```
Front element is: 10

Front element is: 20
```

```
Queue is empty: 1
```

Time and Space Complexity

- The **time complexity** of the implementation of queue using linked list is O(1).
- The **space complexity** of the implementation of queue using linked list is O(n), where n is number of the nodes in the linked list.

Application of the Linked List Queue

- It can applies in operating system uses the queues for job scheduling.
- It can be used in web servers for the handling incoming requests in the order.
- It can be used for the buffering data streams into the media players.
- It can applies on the inter-process communication for the facilitate asynchronous data exchange between the processes.

Conclusion

Implementing the queue using Linked list in the C++ offers the advantages of the dynamic memory usage. It can allowing the queue to the expand based on the needs of the application. This approach can particularly useful in the scenarios where the maximum size of the queue is not known in advance.

Queue Reversal

Given a Queue **Q** containing **N** elements. The task is to reverse the Queue. Your task is to complete the function **rev(),** that reverses the **N** elements of the queue.

```
Input:
6
4 3 1 10 2 6
Output:
6 2 10 1 3 4
Explanation:
After reversing the given elements of the queue , the
resultant queue will be 6 2 10 1 3 4.
```

```
class Solution
{
  public:
  queue<int> rev(queue<int> q)
  {
```

```
        // add code here.
        stack<int> s;
        while(!q.empty())
        {
          s.push(q.front());
          q.pop();
        }
        while(!s.empty())
        {
          q.push(s.top());
          s.pop();
        }
        return q;
    }
};
```

Reversing the first K elements of a Queue

Description - Given an integer k and a queue of integers, we need to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.
Only the following standard operations are allowed on the queue.

- enqueue(x) : Add an item x to rear of queue
- dequeue() : Remove an item from front of queue
- size() : Returns number of elements in queue.
- front() : Finds front item.

```
Input : Q = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
        k = 5
Output : Q = [50, 40, 30, 20, 10, 60, 70, 80, 90, 100]
```

Solution - The idea is to use an auxiliary stack and follow these steps to solve the problem -

1. Create an empty stack.
2. Push the first K elements into a Stack
3. Enqueue the contents of stack at the back of the queue.
4. Remove the remaining elements from queue and enqueue them at the end of the Queue

```
void reverseQueueFirstKElements(k, Queue)
{
    if (Queue.empty() == true || k > Queue.size())
        return
    if (k <= 0)
```

```
        return
    stack  Stack

/* Push the first K elements into a Stack*/
    for ( i = 1 to k) {
        Stack.push(Queue.front())
        Queue.pop()
    }
/* Enqueue the contents of stack at the back of the queue*/
    while (!Stack.empty()) {
        Queue.push(Stack.top())
        Stack.pop()
    }
/* Remove the remaining elements and enqueue them at the end of
the Queue*/
    for (int i = 0 to i < Queue.size() - k) {
        Queue.push(Queue.front())
        Queue.pop()
    }
}
```

Time Complexity : O(n) , n : size of queue **Auxiliary Space :** O(k)

USE DEQUE:

```
class Solution
{
  public:

  // Function to reverse first k elements of a queue.
  queue<int> modifyQueue(queue<int> q, int k)
  {
    deque<int> d;
    if (q.empty() == true || k > q.size())
      return q;
    if (k <= 0)
      return q;

    // Dequeue the first k elements of the queue and push them onto a deque
    for (int i = 0; i < k; i++) {
      d.push_front(q.front());
      q.pop();
    }

    // Pop the elements from the deque and enqueue them back into the queue
    while (!d.empty()) {
      q.push(d.front());
      d.pop_front();
    }
```

```
        // Dequeue the remaining elements from the queue and enqueue them back into the queue
        for (int i = 0; i < q.size() - k; i++) {
            q.push(q.front());
            q.pop();
        }
        return q;
    }
};
```

Expected Time Complexity : O(N)
Expected Auxiliary Space : O(K)

Rotate Deque By K

Given a Deque **deq** of size **N** containing non-negative integers and a positive integer **K**, the task is to rotate elements of the Deque by **K** places in a circular fashion. There will be two rotations which you have to implement depending on the type rotating query. Below is the description of rotating type and value of **K** for which you have to rotate in circular way:
Type-1. right_Rotate_Deq_ByK(): this function should rotate deque by K places in **a clockwise** fashion.
Type-2. left_Rotate_Deq_ByK(): this function should rotate deque by K places in **an anti-clockwise** fashion.

Example 1:

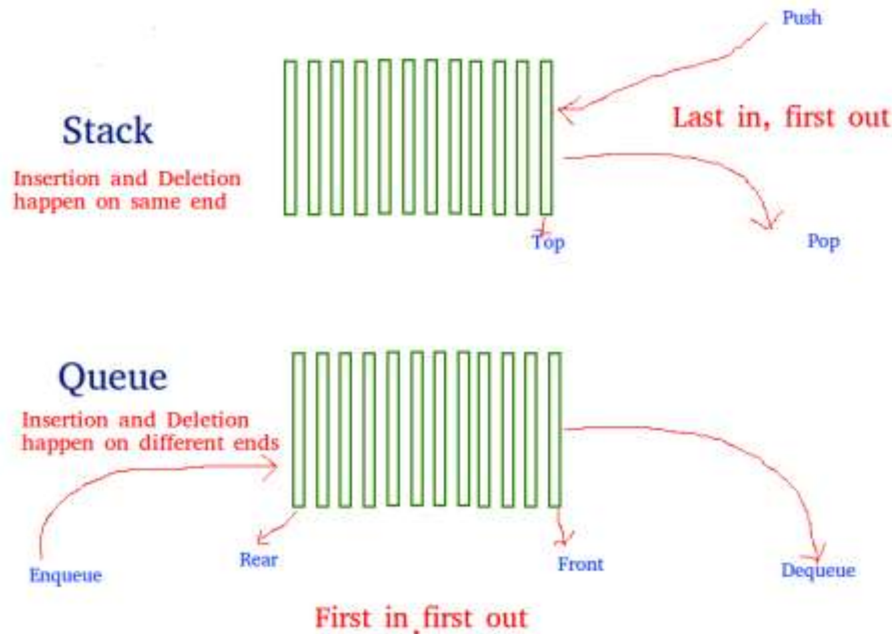
```
Input:
6
1 2 3 4 5 6
1 2

Output:
5 6 1 2 3 4

Explanation:
The dequeue is 1 2 3 4 5 6.
```

```
The query type is 1 and k is 2. So, we
need to right rotate dequeue by 2 times.
In 1 right rotation we get 6 1 2 3 4 5.
In another we get 5 6 1 2 3 4.
```

```
//Function to rotate deque by k places in anti-clockwise direction.
void left_Rotate_Deq_ByK(deque<int> &deq, int n, int k)
{
   // your code here
   for (int i = 0; i < k; i++) {
      int val = deq.front();
      deq.pop_front();
      deq.push_back(val);
   }
}

//Function to rotate deque by k places in clockwise direction.
void right_Rotate_Deq_ByK(deque<int> &deq, int n, int k)
{
   // your code here
   for (int i = 0; i < k; i++) {
      int val = deq.back();
      deq.pop_back();
      deq.push_front(val);
   }
}
```

Implementing Queue using Stack

Problem: Given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.



Solution: *A queue can be implemented using two stacks.* Let the queue to be implemented be **q** and stacks used to implement **q** are **stack1** and **stack2** respectively.

The queue ***q*** can be implemented in two ways: either enQueue operation costly or deQueue operation costly.

- By making enQueue operation costly:
  This method makes sure that oldest entered element(element inserted first) is always at the top of stack1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used. The idea is to while pushing an element, first move all elements from stack1 to stack2, insert the new element to stack1 and then again move all elements from stack2 to stack1. Below is the implementation of both enQueue() and deQueue() operations:

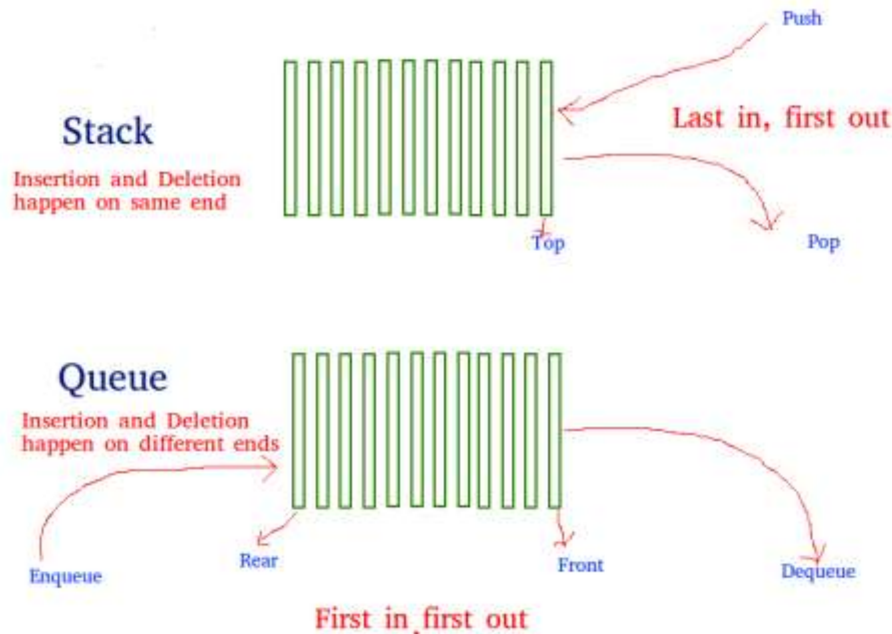
```
enQueue(q, x)
  1) While stack1 is not empty, push everything from stack1 to
stack2.
  2) Push x to stack1 (assuming size of stacks is unlimited).
  3) Push everything back to stack1.
Here the time complexity will be O(n)

deQueue(q)
```

```
1) If stack1 is empty then print an error
2) Pop an item from stack1 and return it
   Here time complexity will be O(1)
```

```
class StackQueue{
private:
   // These are STL stacks ( http://goo.gl/LxlRZQ )
   stack<int> s1;
   stack<int> s2;
public:
   void push(int);
   int pop();
}; */

//Function to push an element in queue by using 2 stacks.
void StackQueue :: push(int x)
{
       // Move all elements from s1 to s2
       while (!s1.empty()) {
          s2.push(s1.top());
          s1.pop();
       }

       // Push item into s1
       s1.push(x);

       // Push everything back to s1
       while (!s2.empty()) {
          s1.push(s2.top());
          s2.pop();
       }
}

//Function to pop an element from queue by using 2 stacks.
int StackQueue :: pop()
{
              // Your code here
              // Dequeue an item from the queue    {
       // if first stack is empty
       if (s1.empty()) {
          return -1;
       }

       // Return top of s1
       int x = s1.top();
       s1.pop();
       return x;
}
```

Implementing Stack using Queue

Problem: Given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.



This problem is just the opposite of the problem described in the previous post of implementing a queue using stacks. Similar to the previous problem, a **stack** can also be implemented using two queues. Let stack to be implemented be **'s'** and queues used to implement be **'q1'** and **'q2'**.

Stack 's' can be implemented in two ways: either push or pop operation costly.

By making push operation costly: This method makes sure that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. The queue, 'q2' is used to put every new element at front of 'q1'.

```
push(s, x) // x is the element to be pushed and s is stack
  1) Enqueue x to q2
  2) One by one dequeue everything from q1 and enqueue to q2.
       3) One by one dequeue everything from q2 and enqueue to
        q1, Swap the names of q1 and q2

pop(s)
  1) Dequeue an item from q1 and return it.
```

```
class QueueStack{
private:
   queue<int> q1;
   queue<int> q2;
public:
   void push(int);
   int pop();
};
 */

//Function to push an element into stack using two queues.
void QueueStack :: push(int x)
{
        // Your Code
        while(q1.empty()==false)
        {
           q2.push(q1.front());
           q1.pop();
        }

        q1.push(x);

        while(q2.empty()==false)
        {
           q1.push(q2.front());
           q2.pop();
        }
}

//Function to pop an element from stack using two queues.
int QueueStack :: pop()
{
        // Your Code
        int x;
        if (q1.empty())
           return -1;
        else
        {
           x=q1.front();
           q1.pop();
        }
        return x;
```

}

An Interesting Method to Generate Binary Numbers from 1 to n

Given a number **N**, write a function that generates and prints all binary numbers with decimal values from 1 to N.
Examples:
Input: *n = 2*
Output: *1, 10*
Input: *n = 5*
Output: *1, 10, 11, 100, 101*

Naive Method: To solve the problem follow the below idea:
A simple method is to run a loop from 1 to n, and call decimal to binary inside the loop.

```
// C++ program to generate binary numbers from 1 to n
#include <bits/stdc++.h>
using namespace std;

void generatePrintBinary(int n)
{
   for(int i=1;i<=n;i++){
       string str="";
       int temp=i;
       while(temp){
           if(temp&1){str=to_string(1)+str;}
           else{str=to_string(0)+str;}

           temp=temp>>1;
       }
       cout<<str<<endl;
   }
}

// Driver code
int main()
{
    int n = 10;

    // Function call
    generatePrintBinary(n);
    return 0;
}
```

Output-

```
1
10
11
100
101
110
111
1000
1001
1010
```

Time Complexity: O(N*logN),N for traversing from 1 to N and logN for decimal to binary conversion

Auxiliary Space: O(1)

Generate Binary Numbers from 1 to n using the queue:

Follow the given steps to solve the problem:

- Create an empty queue of strings
- Enqueue the first binary number "1" to the queue.
- Now run a loop for generating and printing n binary numbers.
    - Dequeue and Print the front of queue.
    - Append "0" at the end of front item and enqueue it.
    - Append "1" at the end of front item and enqueue it.

```
// C++ program to generate binary numbers from 1 to n
#include <bits/stdc++.h>
using namespace std;

// This function uses queue data structure to print binary
// numbers
void generatePrintBinary(int n)
{
    // Create an empty queue of strings
    queue<string> q;

    // Enqueue the first binary number
    q.push("1");

    // This loops is like BFS of a tree with 1 as root
    // 0 as left child and 1 as right child and so on
    while (n--) {
        // print the front of queue
        string s1 = q.front();
        q.pop();
        cout << s1 << "\n";

        string s2 = s1; // Store s1 before changing it
```

```
        // Append "0" to s1 and enqueue it
        q.push(s1.append("0"));

        // Append "1" to s2 and enqueue it. Note that s2
        // contains the previous front
        q.push(s2.append("1"));
    }
}

// Driver code
int main()
{
    int n = 10;

    // Function call
    generatePrintBinary(n);
    return 0;
}
```

Output

```
1
10
11
100
101
110
111
1000
1001
1010
```

Time Complexity: O(N)
Auxiliary Space: O(N) as extra space is required in this method