Lab Report 07

Stack Using LL and Queue

Submitted By

Muhammad Awais / 2883

Submitted To

Ma’am Zanaib Malik

Department of Computer Science and Engineering

National University of modern language, Islamabad

List of Task (Stack using linked list)

1. Node Class
2. Function Push
3. Pop Function
4. IsEmpty
5. Top Value
6. Infix to postfix
7. Infix to prefix
8. Check Precedency

1. Description of linkedStack.h

Linked Stack, which implements a stack data structure using a singly linked list.

The class template has a single data member, which is a pointer to the head node of the linked list, named top.

The class template provides four member functions:

push: This function takes a single argument of type t and adds it to the head of the linked list, effectively pushing it onto the stack.
pop: This function removes the element at the head of the linked list and returns its value, effectively popping it from the stack.
isEmpty: This function returns a boolean value indicating whether the stack is empty (i.e., whether top is NULL).
topValue: This function returns the value of the element at the head of the linked list without removing it.
Note that the implementation of the push and pop functions depends on the implementation of the Node class, which is defined in a separate header file included at the top of the class template file (Node.h). The Node class presumably implements a basic singly linked list node with a data member of type t and a next pointer to the next node in the list.

Source Code

```
#include<iostream>
#include "E:\BSCS 3rd Semester\DSA\My_code\Singly linked list\Node.h"
using namespace std;
template<class t>
class linkedStack
{
```

```
private:
    Node<t> *top;

    public:
        linkedStack()
        {
          top=0;
        }

        void push(t element);// add to head
        t pop();// remove from head
        bool isEmpty();
        t topValue();


}; // End of Class
```

2. Description of Push Function

Source Code

```
template<class t>
void linkedStack<t>::push(t element)
{
    Node<t> *n= new Node<t>( element,NULL);
    if(top==0)
    {
        top=n;


    }
    else
    {
        n->setNext(top);
        top=n;
    }
}
```

3. Description of Void Pop

This is the implementation of the pop member function of the linkedStack class template.

The function first checks if the top pointer is NULL, which means that the linked list is currently empty. In this case, the function prints an error message and does not remove any node from the list. Alternatively, it could return some default value or throw an exception to indicate the error.

If the top pointer is not NULL, then the function proceeds to remove the head node from the list and return its data value. To do so, the function first creates a temporary pointer temp pointing to the head node. It then retrieves the data value of the head node and updates the top pointer to point to the next node in the list (i.e., the node pointed to by the next member of the head node). Finally, the function deletes the original head node pointed to by temp to free its memory.

Thus, the pop function effectively removes and returns the top element from the stack by removing the head node from the linked list and returning its data value.

Source Code

```
template<class t>
t linkedStack<t>::pop()
{
     if(top==0 ) // empty
    {
        cerr<<"List is empty Nothing will remove"<<endl;
        //return NULL;
    }
      else
       {
        Node<t>* temp = top;
        t element=top->getData();
        top = top->getNext();
        delete temp;
        return element;
      }
      }
```

Output

```
  4    /*Declaraed a function string reversal that will take a string
  5    string stringReversal(string text);
  6    using namespace std;
  7    int main(int argc, char** argv) {
  8        Stack<char> s1(5);
  9        s1.push('c');
 10        s1.push('d');
 11        s1.push('f');
 12        s1.push('g');
 13        s1.push('z');
 14        //cout<<"Top value is -> "<<s1.topValue();
 15       while(!s1.isEmpty())// s1.isEmpty==false
 16        {
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS E:\BSCS 3rd Semester\DSA\lab code\Stack\Stack using Array> cd "e:\BSC
++ main.cpp -o main } ; if ($?) { .\main }
z
g
f
d
c
PS E:\BSCS 3rd Semester\DSA\lab code\Stack\Stack using Array>
```

4. Description of IsEmpty

This is the implementation of the isEmpty member function of the linkedStack class template.

The function checks if the top pointer is NULL, which means that the linked list is currently empty. If the top pointer is NULL, the function returns true, indicating that the stack is empty. Otherwise, the function returns false, indicating that the stack is not empty.

Thus, the isEmpty function effectively determines whether the stack is empty by checking whether the linked list is empty or not

Source Code

```
template<class t>
bool linkedStack<t>::isEmpty()
{
    if(top==0)

        return true;
}
```

Output

```
3   /* run this program using the console pauser or add your own getch,
4   /*Declaraed a function string reversal that will take a string as a
5   string stringReversal(string text);
6   using namespace std;
7   int main(int argc, char** argv) {
8       Stack<char> s1(5);
9       s1.push('c');
10      s1.push('d');
11      s1.push('f');
12      s1.push('g');
13      s1.push('z');
14      cout<<s1.isEmpty();
15  //    while(!s1.isEmpty())// s1.isEmpty==false
16  //  {
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS E:\BSCS 3rd Semester\DSA\lab code\Stack\Stack using Array> cd "e:\BSCS 3rd
++ main.cpp -o main } ; if ($?) { .\main }
0
PS E:\BSCS 3rd Semester\DSA\lab code\Stack\Stack using Array>
```

5. Description Top Value

This is the implementation of the topValue member function of the linkedStack class template.

The function checks if the top pointer is NULL, which means that the linked list is currently empty. If the top pointer is NULL, the function prints an error message to the console using cerr. Otherwise, the function returns the data stored in the node pointed to by top.

Thus, the topValue function effectively retrieves the value at the top of the stack by returning the data stored in the node pointed to by top. However, if the stack is empty, the function does not return any value and prints an error message to the console instead.

Source Code

```
template<class t>
t linkedStack<t>::topValue()
{
     if(top==0)
    {
        cerr<<" Stack is underflow \n";
    }
    else
    {
        return top->getData();
    }
```

```
}
```

6. Description of infix to postfix

The infixToPostfix() function takes an infix expression as a string and converts it to a postfix expression using a stack. It starts by initializing an empty stack of characters s1 and an empty string postfix to hold the final postfix expression. It then loops through each character in the input infix string.

If the character is an operand (letter), it is appended to the postfix string.

If the character is an opening parenthesis '(', it is pushed onto the stack.

If the character is an operator ('+', '-', '/', '*'), the function checks if the stack is empty or not. If the stack is empty, the operator is pushed onto the stack. If the stack is not empty, the function checks the precedence of the operator with the operator at the top of the stack. If the precedence of the incoming operator is lower than or equal to the precedence of the top of the stack, the top of the stack is popped and added to the postfix string until the stack is empty, the top of the stack is '(', or the precedence of the top of the stack is lower than the incoming operator. Then the incoming operator is pushed onto the stack.

If the character is a closing parenthesis ')', the function pops and adds operators from the stack to the postfix string until it finds the corresponding opening parenthesis '(' at the top of the stack. Then the opening parenthesis is popped and discarded.

After the loop, the function checks if the stack is not empty, it pops and adds any remaining operators to the postfix string.

Finally, the function returns the postfix string, which is the postfix expression equivalent of the input infix expression.

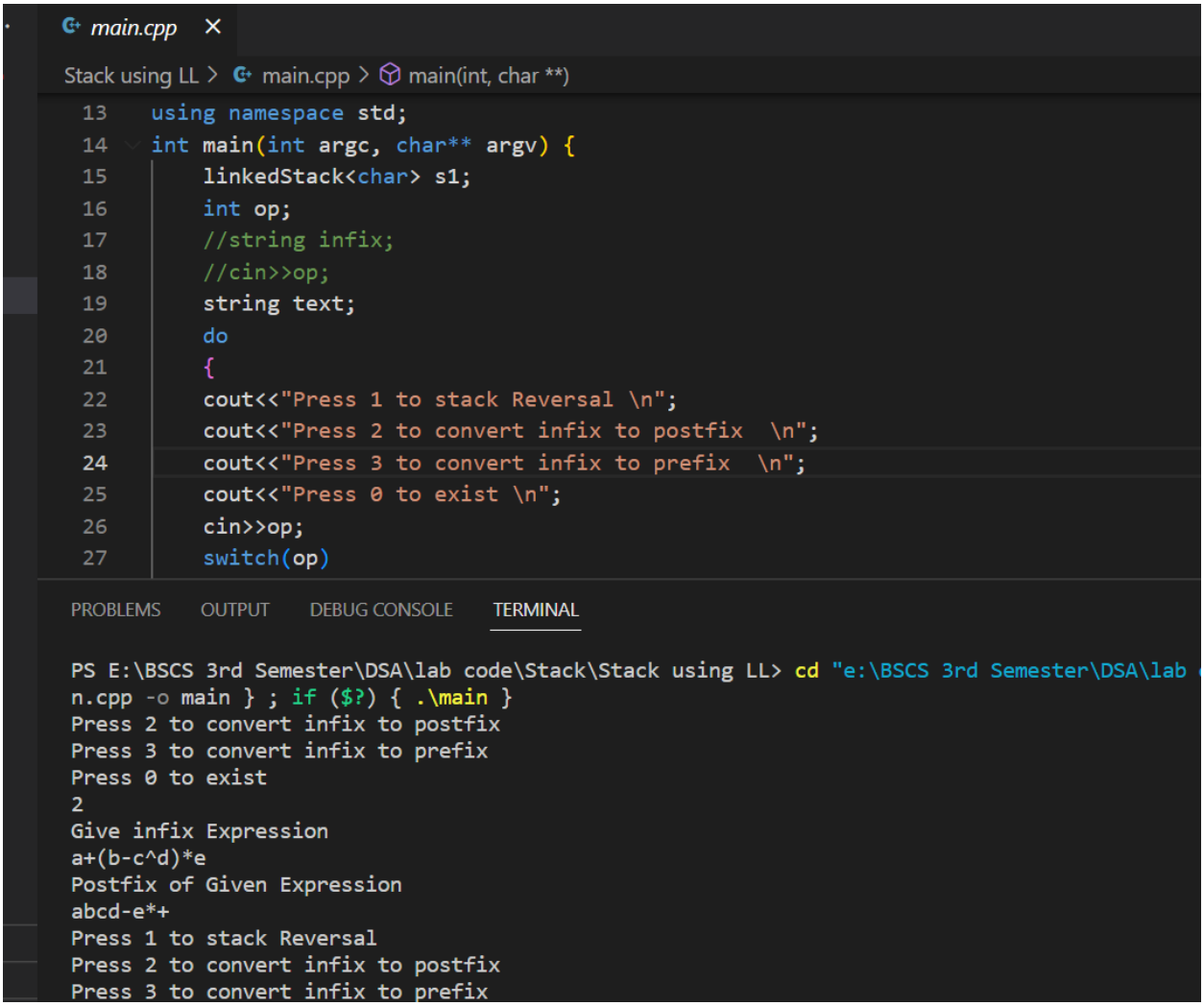
Source Code

```
//// Function infix to posfix
string infixToPostfix(string infix)
{
    linkedStack<char> s1;// point 01
    string postfix=""; // output
    for (int i=0; i<infix.length(); i++)// point 02
    {
      if(isalpha(infix[i]))// 2.1 operend
         {
```

```
            postfix+=infix[i]; //   postfix=postfix+infix[i]
        }// end 2.1

      else if(infix[i]=='(')// 2.2 opening
      {
        s1.push(infix[i]);
        }// end 2.2

      else if(infix[i]=='+' || infix[i]=='-'|| infix[i]=='/' || infix[i]=='*') //
2.3 operator
      {
         if(s1.isEmpty())// 2.3.1 empty
         {
            s1.push(infix[i]);
          }// Emd of 2.3.1

        else // 2.3.2 not empty
        {
            while(!s1.isEmpty() && s1.topValue()!='(' && prec(s1.topValue()) >=
prec(infix[i]) )
            {
              postfix+=s1.pop();

            }// End of while
            s1.push(infix[i]);
        } // End of  2.3.2


        }// end 2.3

    else if(infix[i]==')') // 2.4 closing
    {
        while(s1.topValue()!='(')
        {
         postfix+=s1.pop();

        }// End
       s1.pop();
    }   // end 2.4

}// End of point 02

  if(!s1.isEmpty()) // we can remove if
  {
    while(!s1.isEmpty())
```

```
        {
         postfix+=s1.pop();

        }// End
  }

  return postfix;
}
// End of string infix To Postfix
```

Output



7. Description of infix to prefix

infixToPrefix:

- Uses a stack to convert the infix expression to prefix notation.
- Reverses the infix expression and iterates through each character of the reversed infix expression.
- If the current character is an operand, it is added to the prefix string.
- If the current character is a closing parenthesis, it is pushed onto the stack.
- If the current character is an operator, it is pushed onto the stack if the stack is empty. If the stack is not empty, then it keeps popping elements from the stack and adding them to the prefix string until it reaches a closing parenthesis or an operator with lower precedence than the current operator. Then it pushes the current operator onto the stack.
- If the current character is an opening parenthesis, it keeps popping elements from the stack and adding them to the prefix string until it reaches a closing parenthesis. It also discards the closing parenthesis from the stack.
- After iterating through the reversed infix expression, it pops all remaining elements from the stack and adds them to the prefix string.
- In summary, infixToPostfix converts an infix expression to postfix notation by iterating through the expression and using a stack to keep track of the operators, while infixToPrefix converts an infix expression to prefix notation by first reversing the expression, iterating through the reversed expression, and then using a stack to keep track of the operators.

Source Code

```
///  Infix to prefix

string infixToPrefix(string infix)
{
    linkedStack<char> s1; // 01
    string prefix=""; // 02
    string r_infix=stringReversal(infix); // 03

    for (int i=0; i<r_infix.length(); i++)// point 04
    {
      if(isalpha(r_infix[i]))// 2.1 operend
         {
            prefix+=r_infix[i]; //  postfix=postfix+infix[i]
        }// end 4.1

      else if(r_infix[i]==')')// 2.2 opening
      {
        s1.push(r_infix[i]);
```

```
        }// end 2.2

      else if(r_infix[i]=='+' || r_infix[i]=='-'|| r_infix[i]=='/' ||
r_infix[i]=='*') // 2.3 operator
      {
         if(s1.isEmpty())// 4 empty
         {
            s1.push(r_infix[i]);
           }// Emd of 4

        else // 4 not empty
        {
            while(!s1.isEmpty() && s1.topValue()!=')' && prec(s1.topValue()) >
prec(infix[i]) )
            {
              prefix+=s1.pop();

            }// End of while
            s1.push(r_infix[i]);
        } // End of 4


        }// end 4


    else if(r_infix[i]=='(') // 2.4 closing
    {
        while(s1.topValue()!=')')
        {
         prefix+=s1.pop();

        }// End
       s1.pop();
    }   // end 4

}// End of point 04

 // 05 // we can remove if

    while(!s1.isEmpty())
        {
         prefix+=s1.pop();

        }// End 05
```

```
string r_prefix=stringReversal(prefix);
return r_prefix;
}// end of infix to prefix
```

Output

```
38          cin>>text;
39          cout<<"Postfix of Given Expression \n";
40          cout<<infixToPostfix(text);
41          cout<<endl;
42          break;
43        case 3:
44         cout<<"Give infix Expression  \n";
45          cin>>text;
46          cout<<"Prefix of Given Expression \n";
47          cout<<infixToPrefix(text);
48          cout<<endl;
49          break;
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Press 1 to stack Reversal
Press 2 to convert infix to postfix
Press 3 to convert infix to prefix
Press 0 to exist
3
Give infix Expression
a(b-c)
Prefix of Given Expression
a-bc
Press 1 to stack Reversal
Press 2 to convert infix to postfix
Press 3 to convert infix to prefix
Press 0 to exist
```

8. Description of Check Precedency

The prec function is a simple implementation of operator precedence, which returns an integer value based on the input operator.

The function takes a single character op as its input, which represents the operator whose precedence needs to be determined.

The function uses a series of if-else statements to check the operator and assign the appropriate precedence value. The ^ operator has the highest precedence, followed by * and /, and then + and -.

The function returns the integer value representing the precedence of the operator.

Source Code

```
int prec(char op)
{
    if(op=='^')
     return 3;

     else if(op=='*' || op=='/')
     return 2;
```

```
    else if(op=='+' || op=='-')
    return 1;
}// end
```

List of Task (Simple Queue Using Array)

1. Queue.h
2. Enqeue
3. Dequeue
4. Is Empty
5. Is Full
6. Front value
7. Rear Value

1. Description of Queue.h

Queue class with five member functions: Enqueue, Dequeue, isEmpty, isFull, frontValue, and rearValue. The class contains private member variables front, rear, size, and a pointer arr which holds the actual elements of the queue.

The constructor takes an optional integer parameter size which defaults to 10, and initializes front and rear to -1. The arr pointer is then dynamically allocated an array of t elements of size this->size.

The Enqueue function takes an argument element of type t and adds it to the rear of the queue if there is space. The Dequeue function removes and returns the front element of the queue if the queue is not empty. The isEmpty function returns true if the queue is empty, and false otherwise. The isFull function returns true if the queue is full, and false otherwise.

The frontValue function returns the value of the front element of the queue without removing it. The rearValue function returns the value of the rear element of the queue without removing it.

Note that the implementation of some member functions is missing and should be completed elsewhere in the code

Source Code

```
template<class t>
class Queue
{
 private:
```

```
    int front;
    int rear;
    int size;
    t *arr;

    public:
        Queue(int size=10) // Defualt as well as parametrized
        {
         rear=-1;
          front=-1;
          this->size=size;
          arr=new t(this->size);
         }

        void Enqueue(t element);
        t Dequeue();
        bool isEmpty();
        bool isFull();
        t frontValue();
        t rearValue();

}; // End of Class
```

2. Description of Enqueue

Function: Enqueue

This function is used to insert an element at the rear end of the Queue.
Input:

The input parameter 'element' is of type t and represents the element to be inserted.
Output:

This function does not return any value.
Function Flow:

Check if the rear index of the Queue is equal to (size-1), which means that the Queue is already full and the insertion of the element is not possible. If so, then print an error message and exit.
If the rear index is -1 (initial condition) and the front index is also -1, which means that the Queue is empty, then insert the first element at the front as well as the rear position of the Queue, and increment both front and rear indices by 1.

If the rear index is not -1, which means that there are already some elements in the Queue, then insert the element at the rear position of the Queue and increment the rear index by 1.
End of Function.

Source Code

```
void Queue<t>::Enqueue(t element)
{
    if(rear==size-1)// error
    {
        cerr<<"Queue is full \n";
    }

    else if(rear==-1 && front-1)// first element
    {
        rear++;
        front++;
        arr[rear]=element;

    }

    else  // in between element
    {
        rear++;
        arr[rear]=element;

    }
}// End
```

3. Description of Dequeue

This is a templated implementation of a Queue data structure in C++. Here's a brief description of the class and its methods:

- Class: Queue<t> (template)
    - Members:
        - **front**: an integer representing the index of the front element in the queue
        - **rear**: an integer representing the index of the rear element in the queue
        - **size**: an integer representing the maximum size of the queue
        - **arr**: a dynamic array of type **t** to store the elements of the queue

- Methods:
  - **Queue(int size=10)**: a constructor that initializes the queue with a default or user-defined size
  - **Enqueue(t element)**: a method to insert an element at the rear of the queue
  - **Dequeue()**: a method to remove and return the front element from the queue
  - **isEmpty()**: a method that returns true if the queue is empty, false otherwise
  - **isFull()**: a method that returns true if the queue is full, false otherwise
  - **frontValue()**: a method that returns the value of the front element without removing it
  - **rearValue()**: a method that returns the value of the rear element without removing it

The method **Enqueue()** inserts an element at the rear of the queue. If the queue is full, it prints an error message. If the queue is empty, it initializes both **front** and **rear** to 0 and inserts the first element at the first position. Otherwise, it increments **rear** and inserts the element at the new position.

The method **Dequeue()** removes and returns the front element from the queue. If the queue is empty, it prints an error message. If there is only one element in the queue, it returns that element and sets both **front** and **rear** to -1 to indicate that the queue is now empty. Otherwise, it returns the element at the front position and increments **front**.

The other methods (**isEmpty()**, **isFull()**, **frontValue()**, and **rearValue()**) are self-explanatory and return the respective values

Source Code

```
template<class t>
t Queue<t>::Dequeue()
{
    if(rear==-1 && front==-1)// error
    {
        cerr<<"Queue is empty \n";
    }

    else if(rear==front)// element on first
    {

        t element=arr[front];
        rear=-1;
        front=-1;
        return element;
    }

    else //  more than one element
    {
```

```
        t element=arr[front];
         front++;
        return element;
    }
}// end of Dequeue
```

4. Description of isEmpty

```
1. template<class t>
2. bool Queue<t>::isEmpty()
3. {
4.     if(rear==-1 && front==-1)
5.     return  true;
6. }// End
7.
8.
```

5. Description of isFull

```
1. template<class t>
2. bool Queue<t>::isFull()
3. {
4.     if(rear==size-1)
5.     return true;
6. }// End
```

6. Description of Front Value

```
7. template<class t>
8. t Queue<t>::frontValue()
9. {
10.     if(front==-1)// error
11.     {
12.         cerr<<"Queue is empty \n";
13.     }
14.
15.
16.     else
17.     {
18.         return arr[front];
19.     }
```

```
20.
21.}// End frontValue
```

7. Description of Rear Value

```
template<class t>
t Queue<t>::rearValue()
{
    if(rear==-1)// error
    {
        cerr<<"Queue is empty \n";
    }


    else
    {
        return arr[rear];
    }

}// End frontValue
```

Some up Output

```
8       q1.Enqueue('a');// f=0 r=1 |a|
9       q1.Enqueue('b');
10      q1.Enqueue('c');
11      q1.Enqueue('d');
12      q1.Enqueue('e');
13      cout<<"Front value -> "<<q1.frontValue()<<endl;
14      cout<<"Rear value -> "<<q1.rearValue()<<endl;
15  //  cout<<q1.isFull();
16  //  cout<<q1.Dequeue();
17  //  cout<<q1.Dequeue();
18  //  cout<<q1.Dequeue();
19  //  q1.Enqueue('h'); // error
20
21
22      while(!q1.isEmpty())
23      {
24          cout<<q1.Dequeue()<<endl;
25      }
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\BSCS 3rd Semester\DSA\lab code\Queue\Using Array> cd "e:\BSCS 3rd Semester\DSA\la
-o main } ; if ($?) { .\main }
Front value -> a
Rear value -> e
a
b
c
d
e
PS E:\BSCS 3rd Semester\DSA\lab code\Queue\Using Array>
```

List of Task (Circular Queue)

1. Node Class
2. Enqueue
3. Dequeue
4. Isfull
5. isEmpty, rear value & front value is same as one-way Queue
6. Some up Output

1. Description of Node class

```
template<class t>
class CircularQueue
{
 private:
    int front;
    int rear;
    int size;
    t *arr;

    public:
        CircularQueue(int size=10) // Defualt as well as parametrized
        {
         rear=-1;
          front=-1;
          this->size=size;
          arr=new t(this->size);
         }

        void Enqueue(t element);
        t Dequeue();
        bool isEmpty();
        bool isFull();
        t frontValue();
        t rearValue();

}; // End of Class
```

2. Description of Enqueue

In the Enqueue function, there are three cases to consider:

If the queue is full (i.e., front == 0 and rear == size - 1 or rear + 1 == front), an error message is printed and the element is not added to the queue.
If the queue is empty (i.e., rear == -1 and front - 1), the front and rear pointers are initialized to 0, and the element is added to the queue.
If the queue is not full or empty, the rear pointer is incremented, and the element is added to the rear of the queue

Source Code

```
void CircularQueue<t>::Enqueue(t element)
{
    if((front==0 && rear==size-1) || rear+1==front)// error
    {
        cerr<<"Queue is full \n";
        return;
    }

    else if(rear==-1 && front-1)// first element
    {
        rear++;
        front++;
    //  arr[rear]=element;

    }

    else if(front>0 && rear==size-1) // refill we can remove front>0 (why)
    {
        rear=0;
        //arr[rear]=element;
    }
    else  // in between element
    {
        rear++;
    //  arr[rear]=element;

    }
    arr[rear]=element;
```

```
}// End Enqueue
```

3. Description of Dequeue

In the Dequeue function, there are three cases to consider:

If the queue is empty (i.e., rear == -1 and front == -1), an error message is printed and a default value is returned.
If there is only one element in the queue (i.e., rear == front), the front and rear pointers are reset to -1, and the element is returned.
If there is more than one element in the queue, the element at the front of the queue is removed and returned, and the front pointer is incremented. If the front pointer reaches the end of the queue and there are still elements in the queue, it is wrapped around to the beginning of the queue (i.e., front > rear && front == size - 1)

Source Code

```
template<class t>

t CircularQueue<t>::Dequeue()
{
    if(rear==-1 && front==-1)// error
    {
        cerr<<"Queue is empty \n";
    }

    else if(rear==front)// element on first
    {

        t element=arr[front];
        rear=-1;
        front=-1;
        return element;
    }

    else if(front>rear && front==size-1) // refill happened
    {
      t element = arr[front];
        front=0;
        return element;
    }

    else //  more than one element
```

```
    {

        t element=arr[front];
         front++;
        return element;
    }
}// end of Dequeue
```

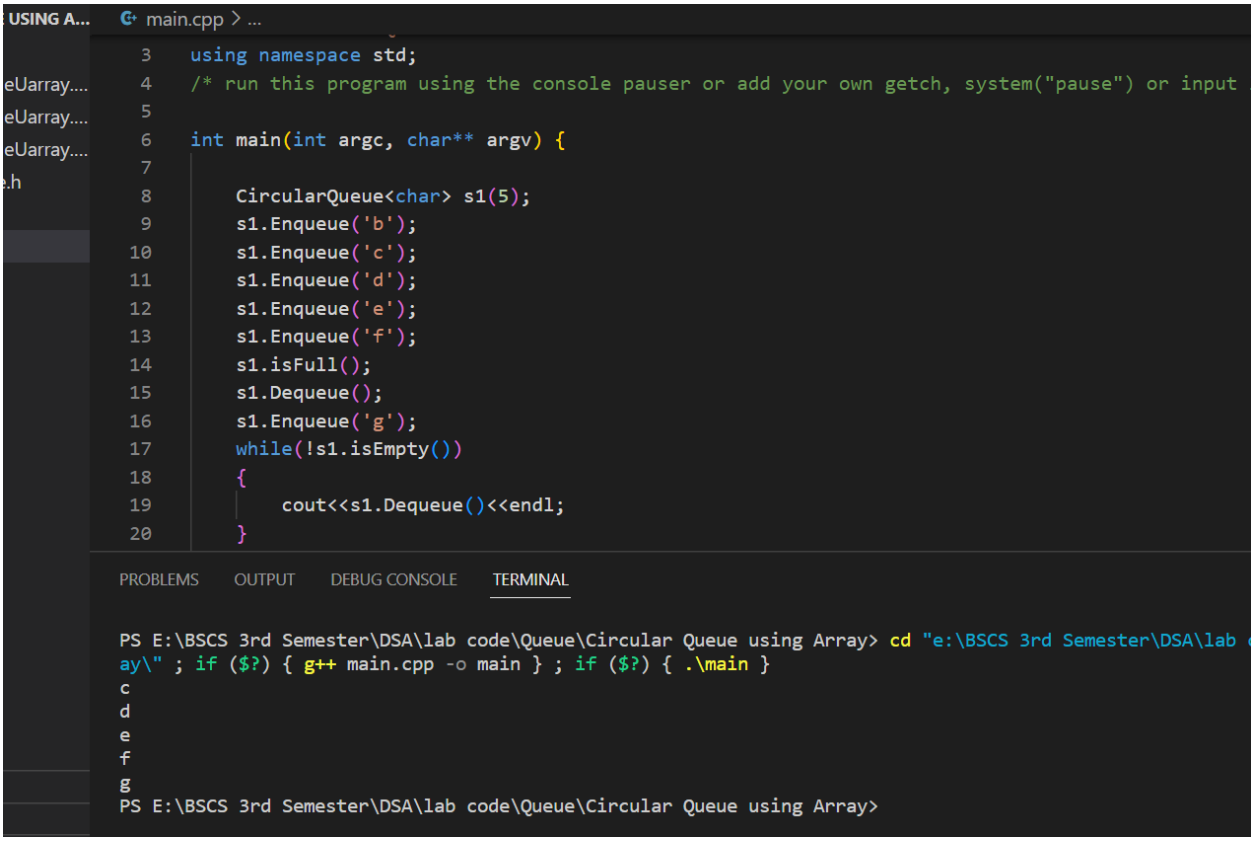
4. Description of isFull

- This is a member function of the CircularQueue class template.
- It returns a boolean value indicating whether the circular queue is full or not.
- It first checks if the front of the queue is at the starting index (0) and the rear is at the last index (size-1), in which case the queue is full.
- If the above condition is not true, it checks if the rear index is one less than the front index (circularly), in which case the queue is also full.
- If either of the above conditions is true, the function returns true, indicating that the queue is full.
- If both the above conditions are false, the function returns false, indicating that the queue is not full.

Source Code

```
template<class t>
bool CircularQueue<t>::isFull()
{
    if((front==0 &&rear==size-1 )|| rear+1==front)
    return true;
}// End
```

Some up output



```
using namespace std;
/* run this program using the console pauser or add your own getch, system("pause") or input

int main(int argc, char** argv) {

    CircularQueue<char> s1(5);
    s1.Enqueue('b');
    s1.Enqueue('c');
    s1.Enqueue('d');
    s1.Enqueue('e');
    s1.Enqueue('f');
    s1.isFull();
    s1.Dequeue();
    s1.Enqueue('g');
    while(!s1.isEmpty())
    {
        cout<<s1.Dequeue()<<endl;
    }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\BSCS 3rd Semester\DSA\lab code\Queue\Circular Queue using Array> cd "e:\BSCS 3rd Semester\DSA\lab
ay\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
c
d
e
f
g
PS E:\BSCS 3rd Semester\DSA\lab code\Queue\Circular Queue using Array>
```

End of Lab 07