Data Structures Lab 06

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Sameer Faisal **T.A:** N/A

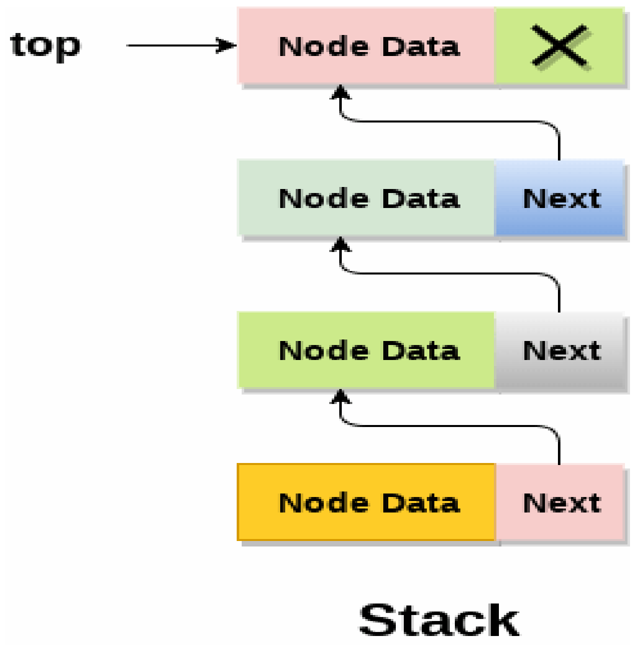
Note:

- Lab manual cover following below elementary sorting algorithms **{Stack with array and linked list, Prefix and Postfix Conversions, Linear Queue with array and its drawback, Circular Queue with array, queue with linked list. }**
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Get your lab checked at the end of the session.

Sample Code of Stack in Array

```
class Stack {
    int top;

public:
    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
}
```

<u>Stack with Linked list</u>



```
struct Node
{
   int data;
   struct Node* link;
};

struct Node* top;

// Utility function to add an element
// data in the stack insert at the beginning
void push(int data)
{

   // Create new node temp and allocate memory
   struct Node* temp;
   temp = new Node();

   // Check if stack (heap) is full.
   // Then inserting an element would
   // lead to stack overflow
   if (!temp)
   {
      cout << "\nHeap Overflow";
      exit(1);
   }

   // Initialize data into temp data field
   temp->data = data;

   // Put top pointer reference into temp link
   temp->link = top;

   // Make temp as top of Stack
   top = temp;
}
```

Application of Stack (convert infix expression to postfix)

The ^ operator has the highest priority, followed by * and / which have equal priority and are evaluated from left to right, and finally + and - which also have equal priority and are evaluated from left to right.

Rules to pop and push operators in a stack:

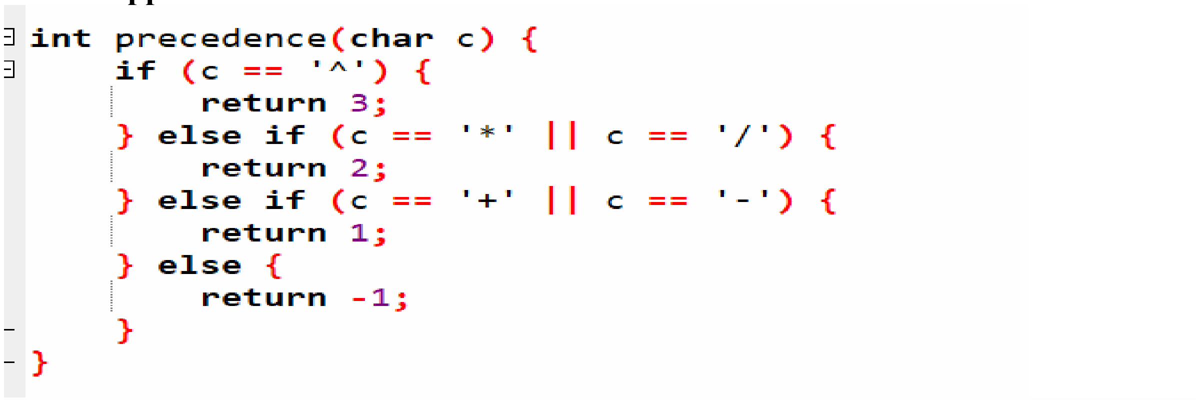
When an operand is encountered in the input, it is added to the output queue.

- When an operator is encountered in the input, the following rules are applied:
- If the stack is empty or the top of the stack contains a left parenthesis, the operator is pushed onto the stack.
- If the operator has higher precedence than the top of the stack, it is pushed onto the stack.
- If the operator has lower or equal precedence than the top of the stack, the top of the stack is popped and added to the output queue. This continues until the operator has higher precedence than the top of the stack, or the stack is empty or contains a left parenthesis.
- If the incoming operator is a right parenthesis, operators are popped from the stack and added to the output queue until a left parenthesis is encountered. The left parenthesis is popped from the stack and discarded.
- After all the tokens have been processed, any remaining operators in the stack are popped and added to the output queue.

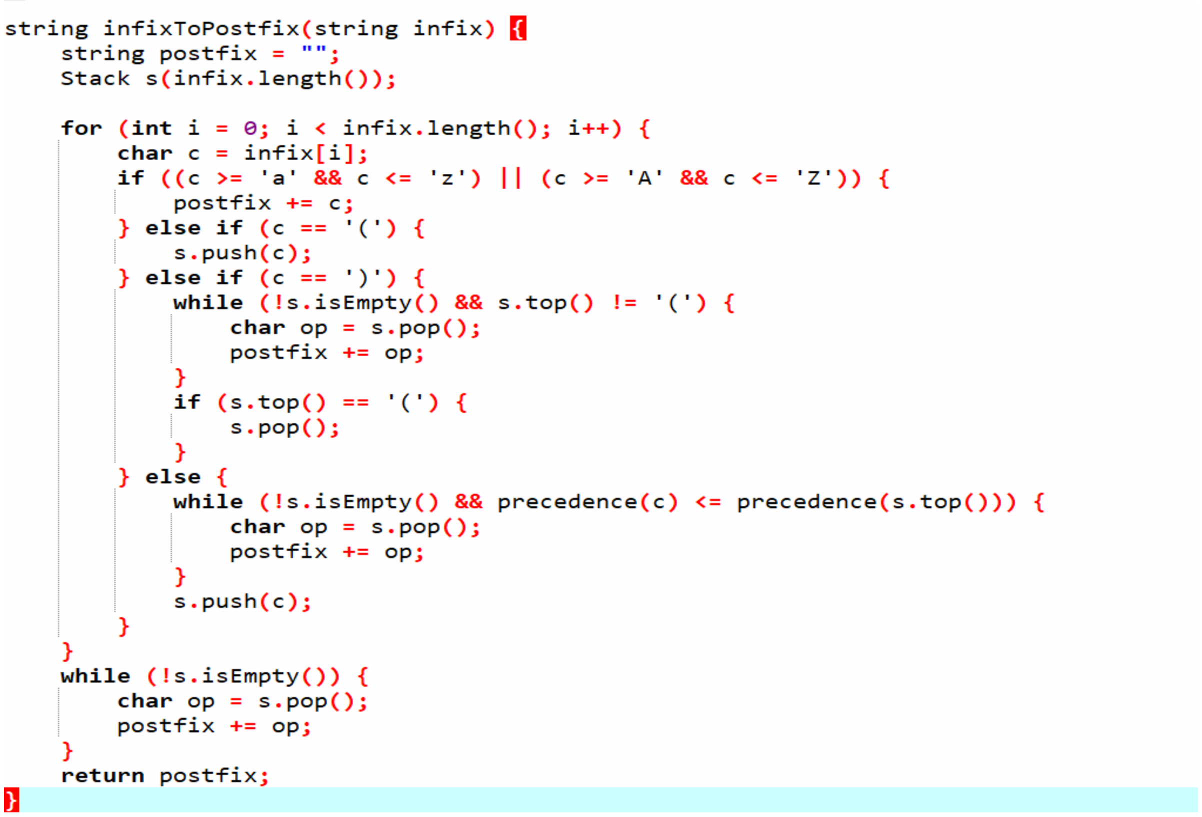
Code Explanation: The algorithm reads each character of the input string and performs the following actions based on the type of the character:

- If the character is an operand, append it to the output string.
- If the character is a left parenthesis, push it onto the stack.
- If the character is a right parenthesis, pop operators from the stack and append them to the output string until a left parenthesis is encountered. Pop the left parenthesis from the stack and discard it.
- If the character is an operator, pop operators from the stack and append them to the output string until an operator of lower precedence is encountered, or the stack is empty, or a left parenthesis is encountered. Push the operator onto the stack.
- After all the characters have been processed, pop any remaining operators from the stack and append them to the output string.

Code Snippet



```
int precedence(char c) {
    if (c == '^') {
        return 3;
    } else if (c == '*' || c == '/') {
        return 2;
    } else if (c == '+' || c == '-') {
        return 1;
    } else {
        return -1;
    }
}
```



```
string infixToPostfix(string infix) {
    string postfix = "";
    Stack s(infix.length());

    for (int i = 0; i < infix.length(); i++) {
        char c = infix[i];
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
            postfix += c;
        } else if (c == '(') {
            s.push(c);
        } else if (c == ')') {
            while (!s.isEmpty() && s.top() != '(') {
                char op = s.pop();
                postfix += op;
            }
            if (s.top() == '(') {
                s.pop();
            }
        } else {
            while (!s.isEmpty() && precedence(c) <= precedence(s.top())) {
                char op = s.pop();
                postfix += op;
            }
            s.push(c);
        }
    }
    while (!s.isEmpty()) {
        char op = s.pop();
        postfix += op;
    }
    return postfix;
}
```

a) Analyze (A+B/C*(D+C)-F)

<u>Infix to Prefix Conversion:</u>

```
string infixToPrefix(string infix) {
   string prefix = "";
   Stack st(infix.length());

   for (int i = infix.length() - 1; i >= 0; i--) {
      char c = infix[i];

      if (isalnum(c)) {
         prefix = c + prefix;
      } else if (c == ')') {
         st.push(c);
      } else if (c == '(') {
         while (!st.isEmpty() && st.pop() != ')') {
            prefix = st.pop() + prefix;
         }
      } else if (isOperator(c)) {
         while (!st.isEmpty() && getPrecedence(st.peek()) >= getPrecedence(c)) {
            prefix = st.pop() + prefix;
         }
         st.push(c);
      }
   }

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

   return prefix;
}
```

Explanations:

- The infixToPrefix function takes an infix expression as input and returns the corresponding prefix expression. The function first reverses the input string using the reverse function, so that it can be processed from right to left.
- The function initializes a stack operators to hold the operators encountered during processing, and a string prefix to hold the prefix expression.
- The function iterates through the reversed infix expression, one character at a time.
- If the current character is an operand (i.e., not an operator or parenthesis), it is added to the prefix string.
- If the current character is an operator, it is pushed onto the operators stack if it has higher precedence than the top operator in the stack, or if the stack is empty. If the current operator has lower precedence than the top operator in the stack, the top operator is popped from the stack and added to the prefix

string, and the process is repeated until the top operator has lower precedence or the stack is empty. Finally, the current operator is pushed onto the stack.

- If the current character is a left parenthesis ((), it is pushed onto the operators stack.
- If the current character is a right parenthesis ()), the top operator is popped from the stack and added to the prefix string until a left parenthesis is encountered. The left parenthesis is then discarded, and the process continues with the next character.
- After all characters have been processed, any remaining operators on the stack are popped and added to the prefix string.
- The prefix string is then reversed back to its original order using the reverse function, and returned as the result of the function.

Linear Queue:

A linear queue is a type of queue where the elements are stored in a contiguous memory location, similar to an array. In a linear queue, the insertion of elements is done at one end (rear) and the deletion of elements is done from the other end (front).

```
#include<iostream>
using namespace std;
int front = -1; int rear = -1; int N = 5; int arr[5];

bool isFull() {
   if(rear == N - 1)  // Rear has reached the last position in the array
      return true;
   else
      return false;
}

bool isEmpty() {
   if(front == -1 || front > rear)  // Front has moved beyond rear
      return true;
   else
      return false;
}

void enqueue(int value) {
   if(isFull()) {
      cout << "Queue is Full!" << endl;
      return;
   }
   else if(isEmpty()) {
      front = rear = 0;
   }
   else {
      rear++;
   }
   arr[rear] = value;
```

```
    }

    void dequeue() {
        if(isEmpty()) {
            cout << "Queue is Empty!" << endl;
            return;
        }
        else if(front == rear) {  // Only one element left
            front = rear = -1;
        }
        else {
            front++;
        }
    }

    void display() {
        if(isEmpty()) {
            cout << "Queue is Empty!" << endl;
            return;
        }

        cout << "\nElements of the Queue: \n" << endl;
        for(int i = front; i <= rear; i++) {
            cout << "arr[" << i << "] = " << arr[i] << endl;
        }
    }

    int main() {
        enqueue(10);
        enqueue(20);
        enqueue(30);
        enqueue(40);
        enqueue(50);

        display();

        dequeue();  // Remove first element (10)
        dequeue();  // Remove next element (20)

        display();

         enqueue(60);   // Try adding an element (this will fail because the queue is full in a linear
    queue)

        return 0;
}
```

Linear Queue Drawback

One drawback of the linear queue is that it doesn't reuse space, so after a few dequeue operations, the queue may become full even though there are free slots at the beginning. A circular queue overcomes this limitation by reusing space.

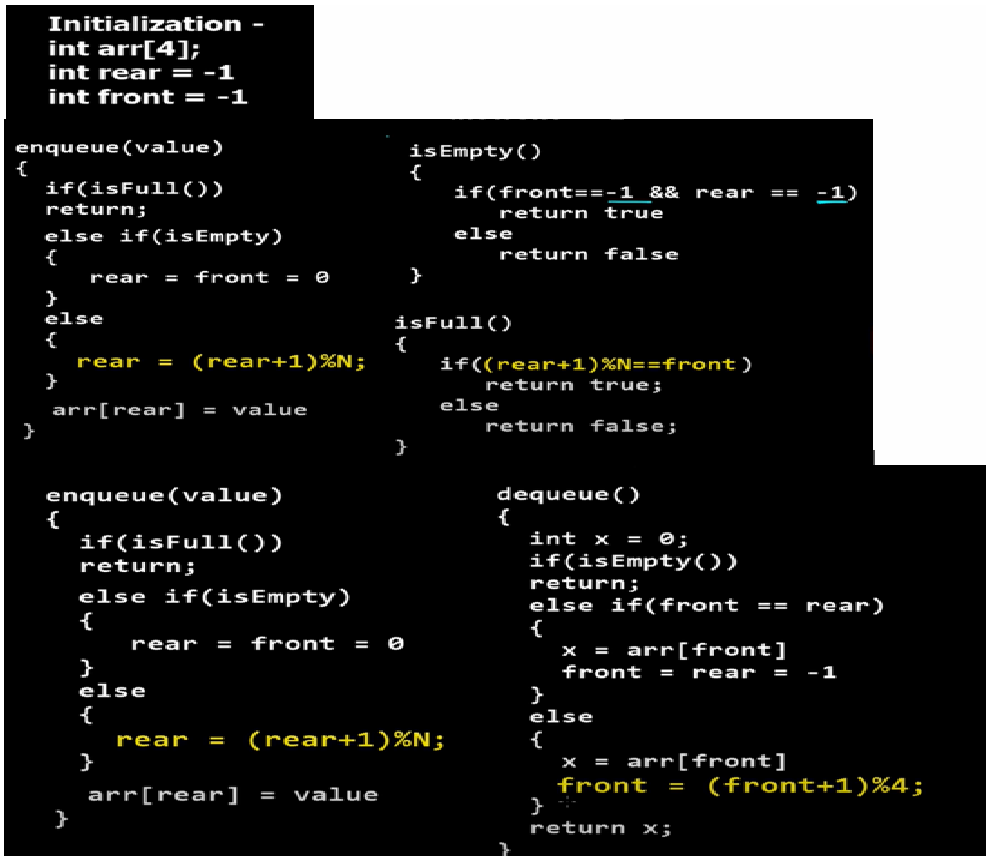
Circular Queue:

A circular queue is a type of data structure that allows you to implement a queue where the front and rear elements are linked together to form a circular chain. In a circular queue, the last element is connected to the first element, forming a circle. This means that when the queue becomes full, we can start adding new elements at the beginning of the queue, provided there is space available there.

The circular queue has four primary operations:

- enqueue, which adds an element to the rear of the queue
- dequeue, which removes an element from the front of the queue
- isFull, which checks whether the queue is full
- isEmpty, which checks whether the queue is empty

Example:



```
Initialization -
int arr[4];
int rear = -1
int front = -1
```

```
enqueue(value)
{
  if(isFull())
  return;
  else if(isEmpty)
  {
     rear = front = 0
  }
  else
  {
    rear = (rear+1)%N;
  }
   arr[rear] = value
 }
```

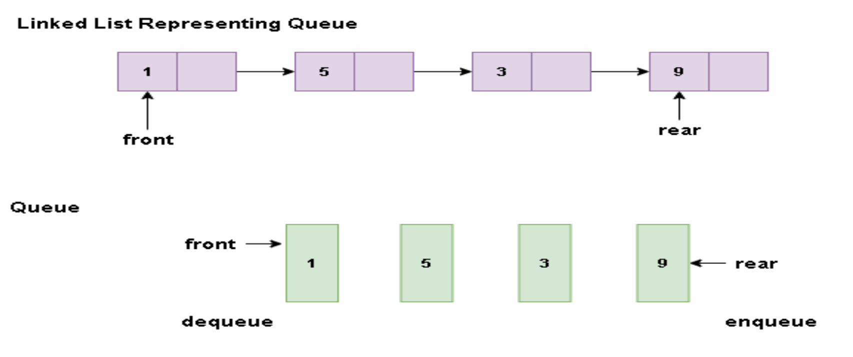
```
isEmpty()
{
   if(front==-1 && rear == -1)
      return true
   else
      return false
}
```

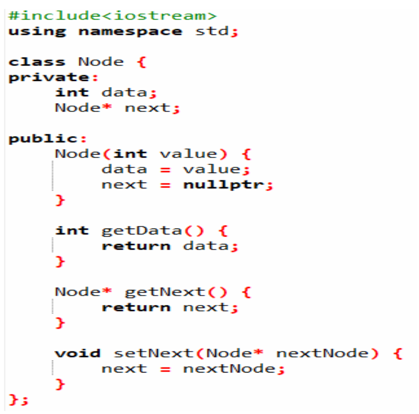
```
isFull()
{
   if((rear+1)%N==front)
      return true;
   else
      return false;
}
```

```
enqueue(value)
{
  if(isFull())
  return;
  else if(isEmpty)
  {
     rear = front = 0
  }
  else
  {
    rear = (rear+1)%N;
  }
   arr[rear] = value
 }
```

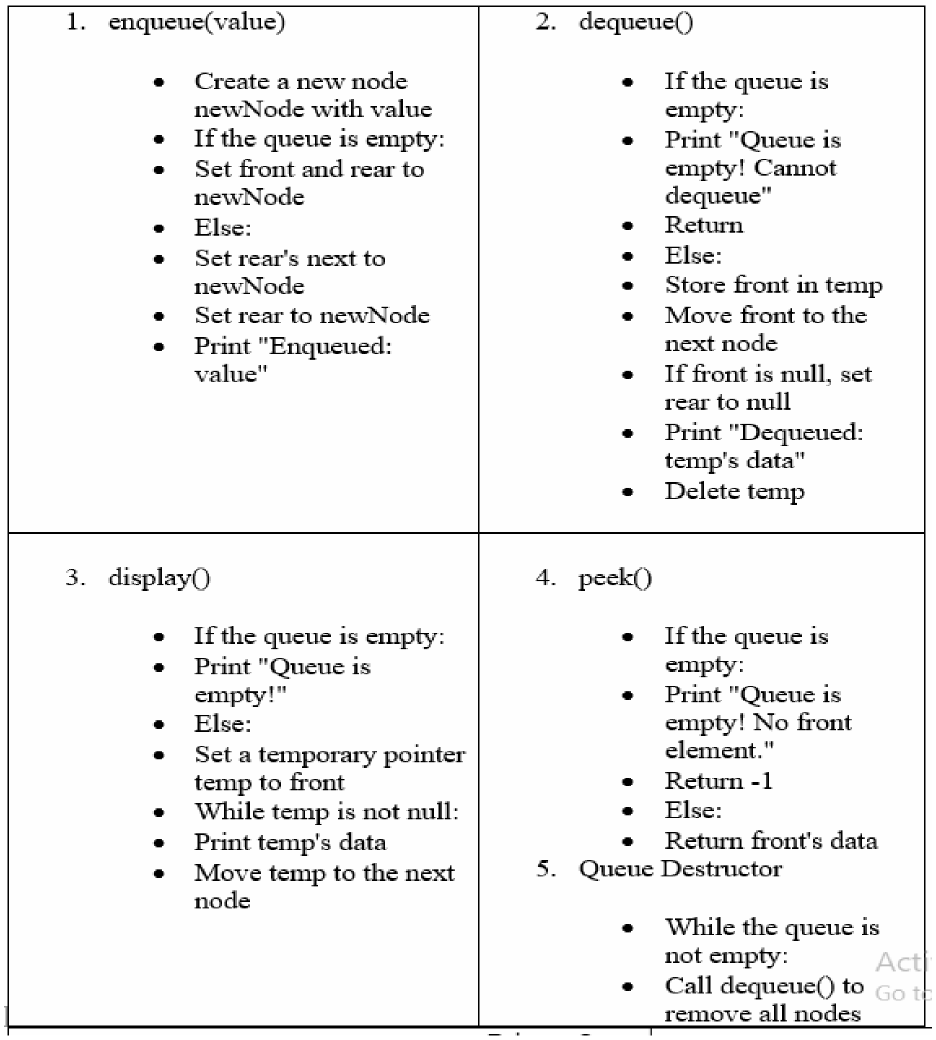
```
dequeue()
{
  int x = 0;
  if(isEmpty())
  return;
  else if(front == rear)
  {
    x = arr[front]
    front = rear = -1
  }
  else
  {
    x = arr[front]
   front = (front+1)%4;
  }
  return x;
}
```

Queue with linked list

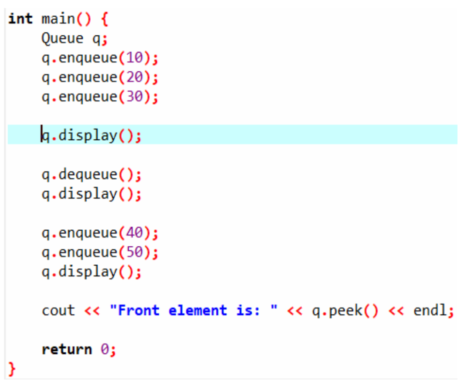




```
#include<iostream>
using namespace std;

class Node {
private:
    int data;
    Node* next;

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

    int getData() {
        return data;
    }

    Node* getNext() {
        return next;
    }

    void setNext(Node* nextNode) {
        next = nextNode;
    }
};
```



1. enqueue(value) • Create a new node newNode with value • If the queue is empty: • Set front and rear to newNode • Else: • Set rear's next to newNode • Set rear to newNode • Print "Enqueued: value"	2. dequeue() • If the queue is empty: • Print "Queue is empty! Cannot dequeue" • Return • Else: • Store front in temp • Move front to the next node • If front is null, set rear to null • Print "Dequeued: temp's data" • Delete temp
3. display() • If the queue is empty: • Print "Queue is empty!" • Else: • Set a temporary pointer temp to front • While temp is not null: • Print temp's data • Move temp to the next node	4. peek() • If the queue is empty: • Print "Queue is empty! No front element." • Return -1 • Else: • Return front's data 5. Queue Destructor • While the queue is not empty: • Call dequeue() to remove all nodes



```cpp
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    q.display();

    q.dequeue();
    q.display();

    q.enqueue(40);
    q.enqueue(50);
    q.display();

    cout << "Front element is: " << q.peek() << endl;

    return 0;
}
```

Lab Exercises

1. Implement and insert the values "BORROWROB" in the stack and identify if it's palindrome or not. Use the push and pop functions to accomplish this (Note: Use Arrays to accomplish this).

2. Implement a Queue based approach where assume you are the cashier in a supermarket and you need to make checkouts. Customer ID's Are 13,7,4,1,6,8,10. (Note: Use Arrays to accomplish this task with enqueue and dequeue).

3. Consider you have an expression x=12+13-5(0.5+0.5) +1 which results to 20. Implement a stack-based implementation to solve this question via linked lists (linked lists can be single or double) and the resulted output must be at the top of the stack. Note that the x and the equa

sign must be present in the stack and when inserting the top value (20 result) all the values must be present in the stack (You can pop and push them accordingly).