Data Structures
Lab Manual 5

Topic: Stack and Queue
(Implementation using array list & linked list)

Session: Spring 2023

Faculty of Information Technology

UCP Lahore Pakistan

Objectives:
The objectives of implementing a stack and queue using an array list and linked list are as follows:

Stack using Array List: The objective of implementing a stack using an array list is to achieve constant-time complexity for basic operations such as push and pop. The implementation should ensure that the stack is dynamic in size and can handle resizing when it reaches its capacity. Additionally, the implementation should provide methods to check if the stack is empty or full and to access the top element of the stack without removing it.

Stack using Linked List: The objective of implementing a stack using a linked list is to have a dynamic and memory-efficient data structure that can handle push and pop operations in constant time. The implementation should ensure that the head of the linked list always points to the top of the stack. Additionally, the implementation should provide methods to check if the stack is empty or to access the top element of the stack without removing it.

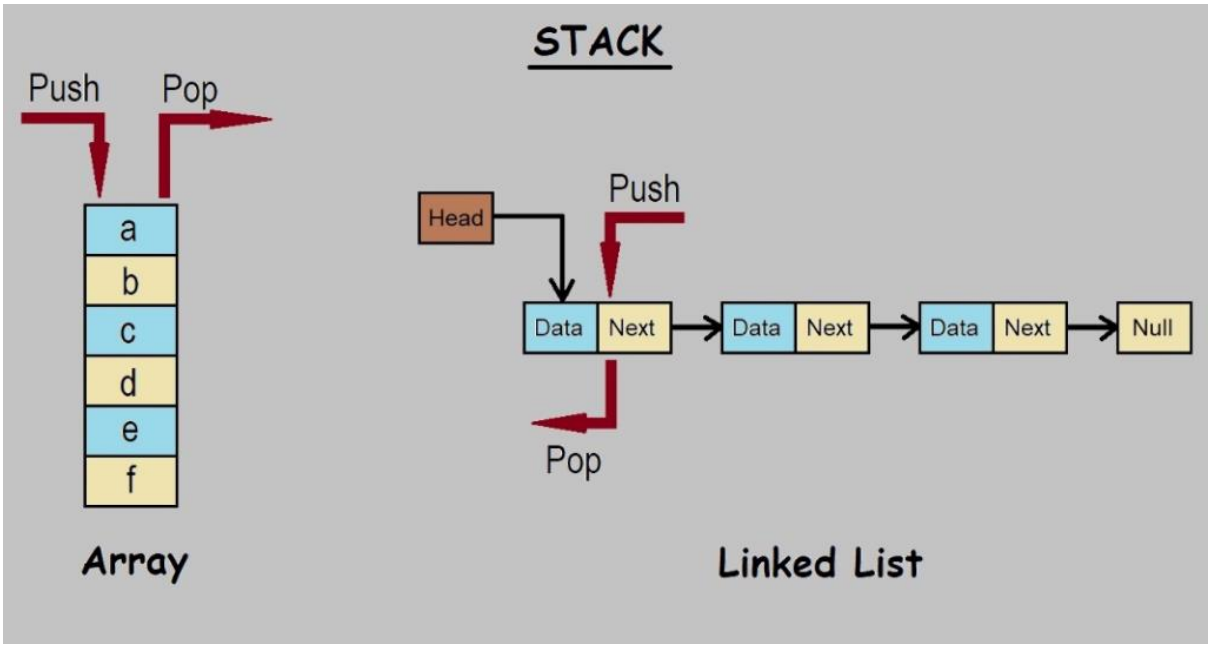
Queue using Array List: The objective of implementing a queue using an array list is to achieve constant-time complexity for basic operations such as enqueue and dequeue. The implementation should ensure that the queue is dynamic in size and can handle resizing when it reaches its capacity. Additionally, the implementation should provide methods to check if the queue is empty or full and to access the front and rear elements of the queue without removing them.

Queue using Linked List: The objective of implementing a queue using a linked list is to have a dynamic and memory-efficient data structure that can handle enqueue and dequeue operations in constant time. The implementation should ensure that the head of the linked list always points to the front of the queue and the tail of the linked list always points to the rear of the queue. Additionally, the implementation should provide methods to check if the queue is empty or to access the front and rear elements of the queue without removing them.

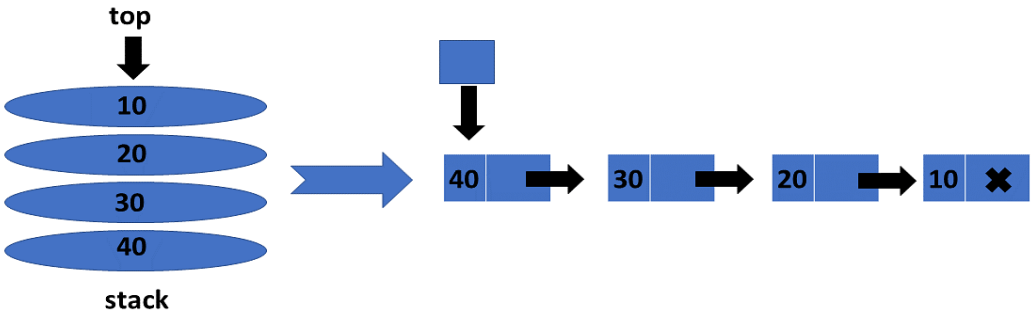
Stack:
A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It is an abstract data type that can be thought of as a stack of plates. The last plate that is added to the stack is the first one that can be removed.
A stack has two main operations: push and pop. The push operation adds an element to the top of the stack, while the pop operation removes the top element from the stack. Additionally, a stack may have other operations such as peek, which returns the top element without removing it, and is_empty, which checks if the stack is empty. A stack can be implemented using various data structures, including an array, a linked list, or a dynamic array (array list). The implementation of a stack may vary depending on the chosen data structure, but the basic operations and principles remain the same. In addition to its common use cases, such as function call stack, expression evaluation, and undo/redo operations, a stack can be used to

solve various algorithmic problems. For example, a stack can be used to check if a given string of parentheses is balanced or to evaluate a postfix expression.

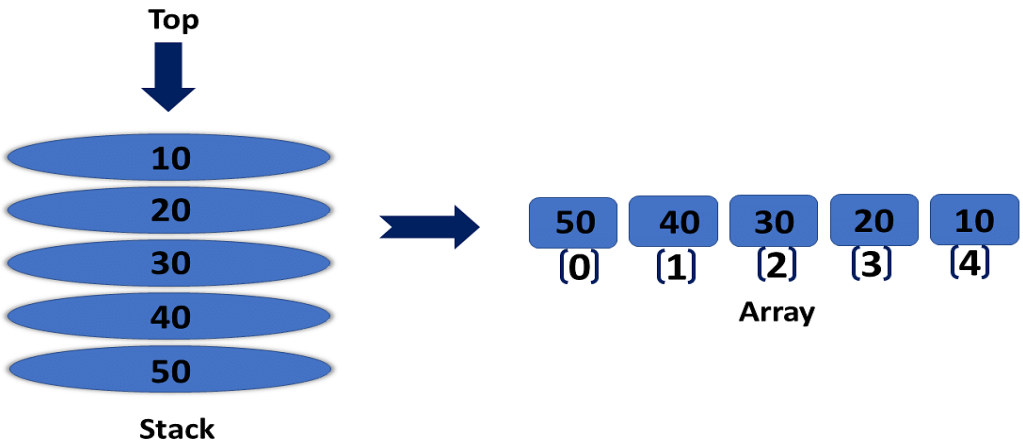
Overall, a stack is a simple yet powerful data structure that has many practical and theoretical applications.



Implement a stack using single linked list: Implementing a stack using a singly linked list is a popular approach to creating a stack data structure in computer science. In this implementation, we can use a linked list to represent the stack, where the head of the linked list represents the top of the stack. Each node in the linked list stores the value of an element in the stack and a reference to the next node in the list.



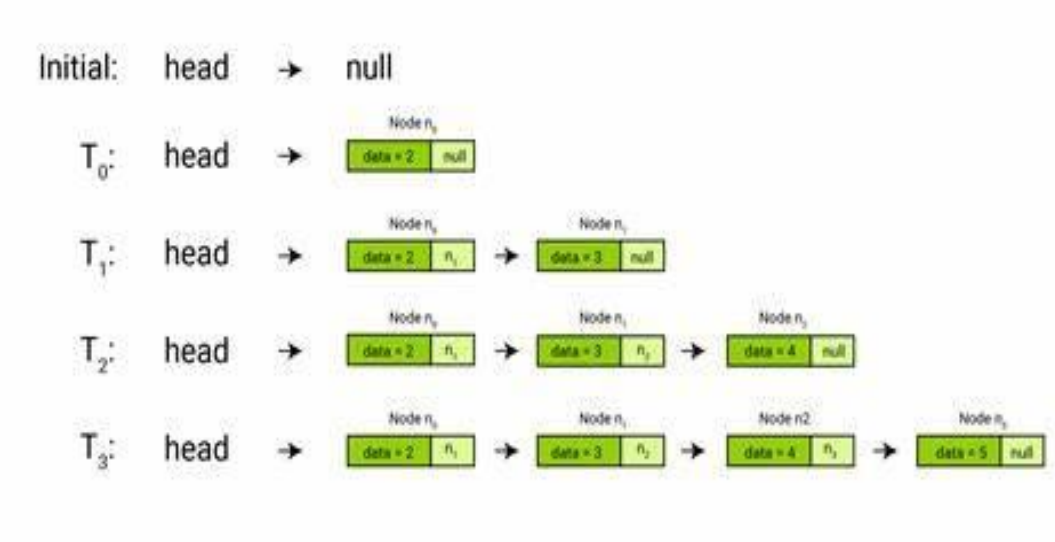
Implement a stack using array list: An array list implementation of a stack involves using an array to store the stack elements. The array is initially allocated with a fixed size, and when the stack reaches its capacity, the array is resized to accommodate more elements.



List of basic operation that can perform on stack:

1. Push: Adds an element to the top of the stack.
2. Pop: Removes the top element from the stack and returns it.
3. Peek/Top: Returns the top element of the stack without removing it.
4. Size: Returns the current number of elements in the stack.
5. Is Empty: Checks if the stack is empty.
6. Clear: Removes all elements from the stack.
7. Search: Finds the position of an element in the stack and returns its index, relative to the top of the stack.
8. Print: Displays the elements of the stack, usually for debugging purposes.

Inserting node in link list based Stack



Sample Task: A program that uses a linked list-based stack with getter and setter methods for its nodes to create a stack of integers and display it on the console.

Pseudocode:

1. Define a class called Node, which contains two private member variables: an integer data value and a pointer to the next node. It also contains four public member functions: a constructor, two getter and setter functions for the data value, and two getter and setter functions for the next pointer.
2. Next, define a Stack class, which contains one private member variable: a pointer to the top node of the stack. It also contains four public member functions: a constructor, a push() function to add an element to the top of the stack, a pop() function to remove the top element from the stack and return its value, an isEmpty() function to check if the stack is empty, and a display() function to print the contents of the stack.
3. The main function creates an instance of the Stack class called s.
4. Four integers are pushed onto the stack using the push() function: 10, 20, 30, and 40.
5. The display() function is called to print the contents of the stack.
6. The pop() function is called to remove the top element from the stack.
7. The display() function is called again to print the contents of the stack after the pop operation.
8. End of the program.

Implementation:

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

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

    Node(int val = 0) {
        data = val;
        next = NULL;
    }

    int getData() { return data; }
    void setData(int val) { data = val; }
    Node* getNext() { return next; }
    void setNext(Node* ptr) { next = ptr; }
};

class Stack {
private:
```

```
    Node* top;
public:

    Stack() { top = NULL; }

    void push(int val) {
        Node* newNode = new Node(val);
        newNode->setNext(top);
        top = newNode;
    }

    int pop() {
        if (isEmpty()) {
            cout << "Stack underflow!" << endl;
            return -1;
        }
        int val = top->getData();
        Node* temp = top;
        top = top->getNext();
        delete temp;
        return val;
    }

    bool isEmpty() {
        return (top == NULL);
    }

    void display() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return;
        }
        cout << "Stack: ";
        Node* temp = top;
        while (temp != NULL) {
            cout << temp->getData() << " ";
            temp = temp->getNext();
        }
        cout << endl;
    }
};

int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    s.push(40);
    s.display();
    s.pop();
    s.display();
    return 0;
}
```

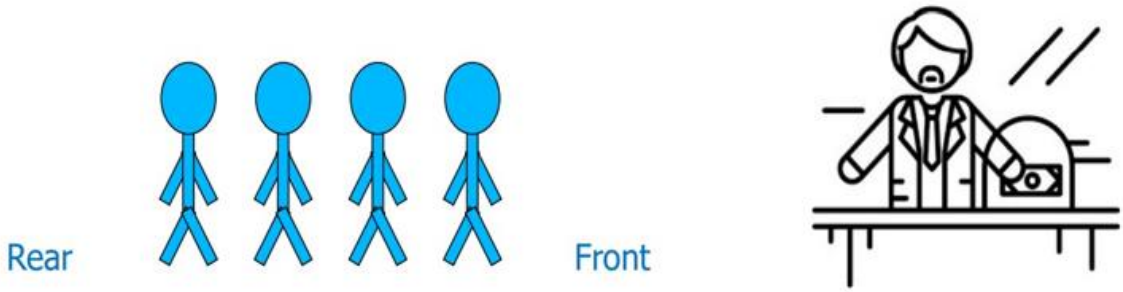
Queue:

A Queue is a linear data structure in computer science that follows the First-In-First-Out (FIFO) principle. It operates on a collection of elements and has two primary operations: Enqueue (adding an element to the back of the queue) and Dequeue (removing an element from the front of the queue). A queue can be visualized as a line of people waiting for a service, where the first person who arrived is served first. Hence, the name FIFO or First-In-First-Out. In programming, a queue is usually implemented using an array or a linked list. A queue has two primary pointers, the front and the rear, which point to the first and last elements of the queue, respectively. The Enqueue operation adds a new element to the rear end of the queue, while the Dequeue operation removes an element from the front end of the queue.

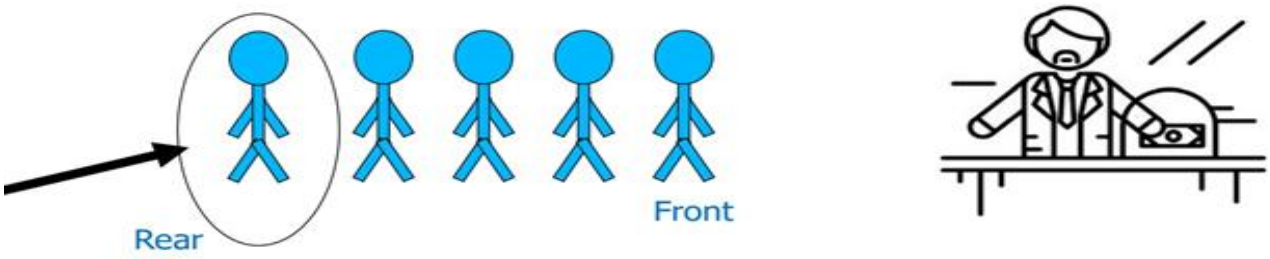
Queues are used in many applications, including operating systems, networking, and simulations, where they are used to manage resources and schedule tasks.

Array-based queues are used when we have to access elements randomly and with less memory. List-based queues are used when there are a greater number of enqueue and dequeue operations and the size of the queue is not known in advance.

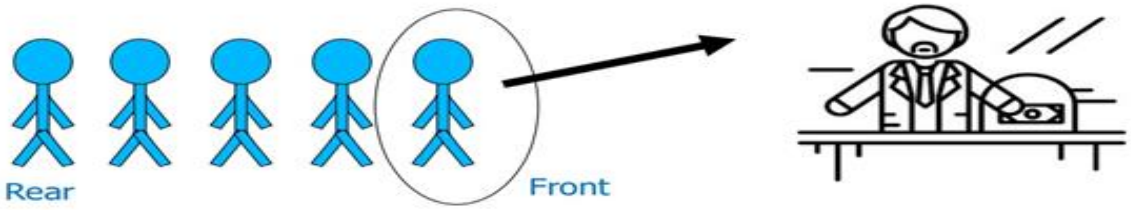
A Queue is like a line of people waiting for a bank teller:



New people must enter the queue at the Rear:



An item is always taken from the front of the queue:

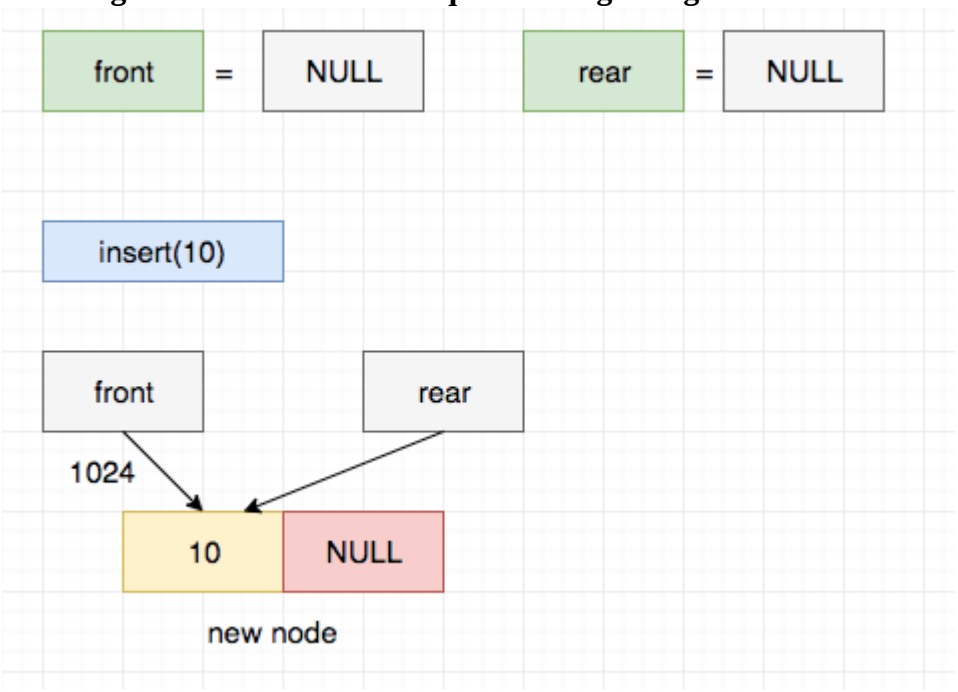


List of basic operation that can perform on Queue:

1. Enqueue: This operation adds an element to the rear end of the queue.
2. Dequeue: This operation removes an element from the front end of the queue.
3. Front: This operation returns the element at the front end of the queue without removing it.
4. Rear: This operation returns the element at the rear end of the queue without removing it.
5. isEmpty: This operation checks if the queue is empty or not.
6. isFull: This operation checks if the queue is full or not (in case of a fixed-size queue).

These operations allow us to manage a queue and perform the required actions based on the specific use-case or application.

Inserting node in link list based queue at beginning



Sample Task: A program that uses link list-based queue with getter and setter methods for its nodes to create a queue of integers and display it on the console.

Pseudocode:

1. Define a class named QueueNode , which represents a single node in the queue. It has two private data members: an integer data and a pointer next to the next node in the queue. It also has a constructor that takes an integer value (which defaults to 0) and sets the data member to that value and the next pointer to NULL.

2. The QueueNode class also has four public methods: getData, which returns the value of the data member; setData, which sets the value of the data member to a given value; getNext, which returns the next pointer; and setNext, which sets the next pointer to a given pointer.

3. Define another class, named Queue. This class represents the queue data structure and has two private data members: front, a pointer to the front node of the queue, and rear, a pointer to the rear node of the queue. It also has a constructor that sets both pointers to NULL.

4. The Queue class has three public methods: enqueue, which adds a new node to the rear of the queue; dequeue, which removes the front node of the queue and returns its value; and isEmpty, which returns true if the queue is empty and false otherwise.

5. The enqueue method creates a new QueueNode object with the given value and adds it to the rear of the queue. If the queue was previously empty, it sets both the front and rear pointers to the new node. If the queue was not empty, it sets the next pointer of the previous rear node to the new node, and updates the rear pointer to point to the new node.

6. The dequeue method removes the front node of the queue, deletes it, and returns its value. If the queue is empty, it prints an error message and returns -1. If the queue becomes empty as a result of the dequeue operation, both the front and rear pointers are set to NULL.

7. The isEmpty method returns true if the front pointer is NULL, indicating an empty queue, and false otherwise.

8. The display method prints out the contents of the queue. If the queue is empty, it prints an error message. Otherwise, it iterates through the queue using a temporary node pointer, printing out each value in turn.

9. In the main function, a Queue object is created. The user is prompted to enter integers to be added to the queue, which are read in one by one and added to the queue using the enqueue method. Once the user enters -1, the contents of the queue are displayed using the display method.

10. End of the program.

Implementation:

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

class QueueNode {
private:
    int data;
    QueueNode* next;
public:
    QueueNode(int val = 0) {
        data = val;
        next = NULL;
    }
    int getData() { return data; }
    void setData(int val) { data = val; }
    QueueNode* getNext() { return next; }
    void setNext(QueueNode* ptr) { next = ptr; }
};

class Queue {
private:
    QueueNode* front;
    QueueNode* rear;
public:

    Queue() {
        front = NULL;
        rear = NULL;
    }
    void enqueue(int val) {
        QueueNode* newNode = new QueueNode(val);
        if (isEmpty()) {
            front = newNode;
        }
        else {
            rear->setNext(newNode);
        }
        rear = newNode;
    }
    int dequeue() {
        if (isEmpty()) {
            cout << "Queue underflow!" << endl;
            return -1;
        }
        int val = front->getData();
        QueueNode* temp = front;
        front = front->getNext();
        delete temp;
        if (front == NULL) {
            rear = NULL;
        }
        return val;
    }

    bool isEmpty() {
        return (front == NULL);
    }
    void display() {
        if (isEmpty()) {
```

```
            cout << "Queue is empty!" << endl;
            return;
        }
        cout << "Queue: ";
        QueueNode* temp = front;
        while (temp != NULL) {
            cout << temp->getData() << " ";
            temp = temp->getNext();
        }
        cout << endl;
    }
};

int main() {
    Queue q;
    int val;
    cout << "Enter integers to enqueue into the queue (enter -1 to stop):" <<
endl;
    cin >> val;
    while (val != -1) {
        q.enqueue(val);
        cin >> val;
    }
    q.display();
    return 0;
}
```

Task For Lab:

1. Implement following functions for both array based stack and link list based stack.
    1. Function **Push** to add element to stack
    2. Function **Pop** to delete element from stack
    3. Function **Top** returns the top element of the stack with out removing it.
    4. Function **isEmpty** returns true if stack is empty otherwise return false.
    5. Function **isFull** returns true if stack is empty otherwise return false.
    6. Function **Size** returns current size of stack
    7. Function **Display** to print all data of stack