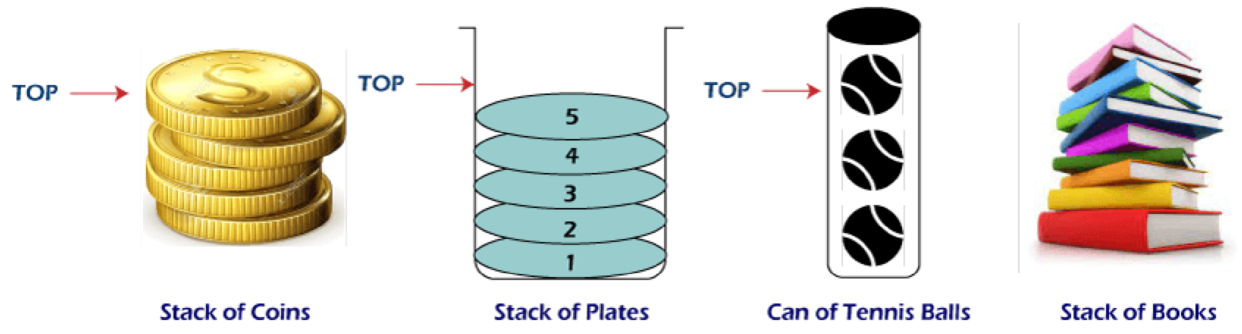
Data Structures Lab 7

Course: Data Structures (CL2001) **Semester:** Spring 2024
Instructor: Bushra Sattar **T.A:** M Asad

Note:

- Lab manual cover following below Stack topics
  {Stack with Array and Linked list, Applications of Stack, Queue with Array and Linked list}
  Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

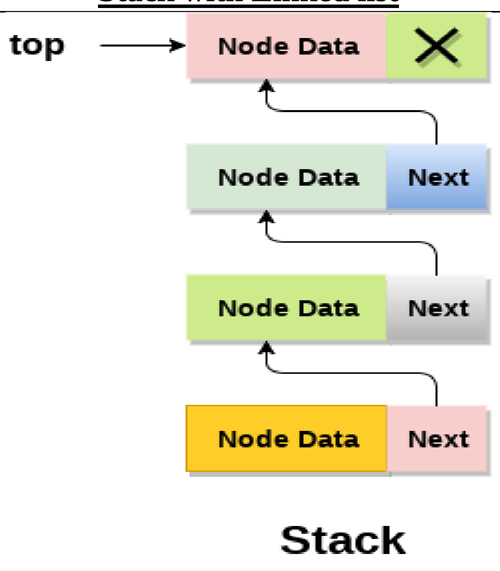
Stack with Array



Sample Code of Stack in Array

```
// Stack implementation in Java

class Stack {
  private int arr[];
  private int top;
  private int capacity;

  // Creating a stack
  Stack(int size) {
    arr = new int[size];
    capacity = size;
    top = -1;
  }

  // Add elements into stack
  public void push(int x) {
    if (isFull()) {
      System.out.println("OverFlow\nProgram Terminated\n");
      System.exit(1);
    }
    System.out.println("Inserting " + x);
    arr[++top] = x;
  }

  // Remove element from stack
  public int pop() {
    if (isEmpty()) {
      System.out.println("STACK EMPTY");
      System.exit(1);
    }
    return arr[top--];
  }

  // Utility function to return the size of the stack
  public int size() {
    return top + 1;
  }

  // Check if the stack is empty
  public Boolean isEmpty() {
    return top == -1;
  }

  // Check if the stack is full
  public Boolean isFull() {
    return top == capacity - 1;
  }

  public void printStack() {
    for (int i = 0; i <= top; i++) {
      System.out.println(arr[i]);
    }
  }

  public static void main(String[] args) {
    Stack stack = new Stack(5);

    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);

    stack.pop();
    System.out.println("\nAfter popping out");

    stack.printStack();

  }
}
```

Stack with Linked list



Sample Code of Stack in Linked List

```
class Node {
public:
    int data;
    Node* link;

    // Constructor
    Node(int n)
    {
        this->data = n;
        this->link = NULL;
    }
};

class Stack {
    Node* top;

public:
    Stack() { top = NULL; }

    void push(int data)
    {

        // Create new node temp and allocate memory in heap
        Node* temp = new Node(data);

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

Stack with Array and Linked list

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

  Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false

Application of Stack

- An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Infix Notation

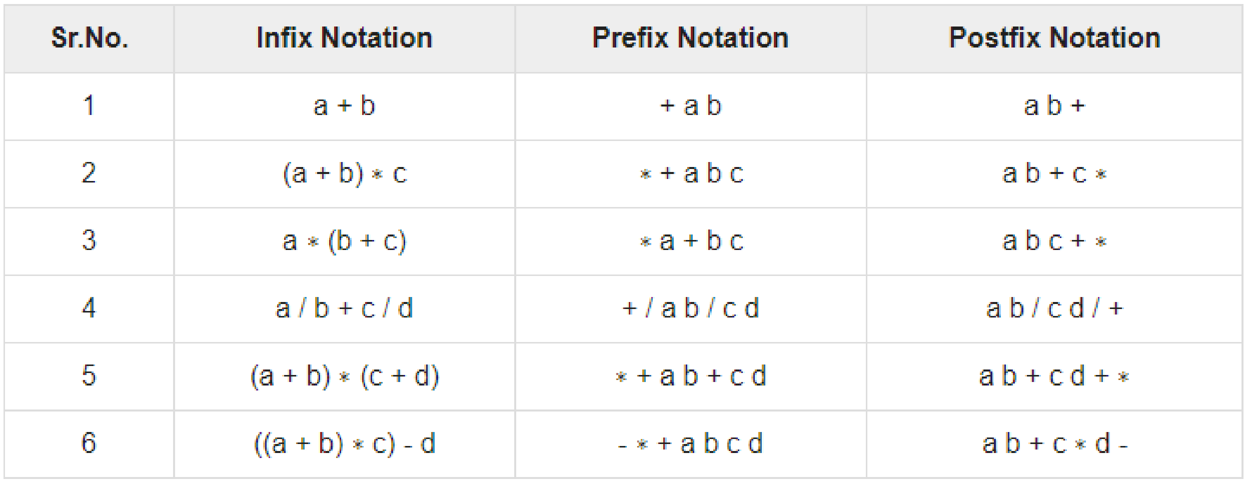
We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.



Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

<u>Queue with Array and Linked list</u>

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Following are the Operations on Queue
1.Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
2.Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
3.Front: Get the front item from queue.
4.Rear: Get the last item from queue.

Applications of Queue

A priority queue in c++ is a type of container adapter, which processes only the highest priority element, i.e. the first element will be the maximum of all elements in the queue, and elements are in decreasing order.

Difference between a queue and priority queue:

- Priority Queue container processes the element with the highest priority, whereas no priority exists in a queue.

- Queue follows First-in-First-out (FIFO) rule, but in the priority queue highest priority element will be deleted first.
- If more than one element exists with the same priority, then, in this case, the order of queue will be taken.

Basic Operations of Queue
A queue is an object (an abstract data structure - ADT) that allows the following operations:

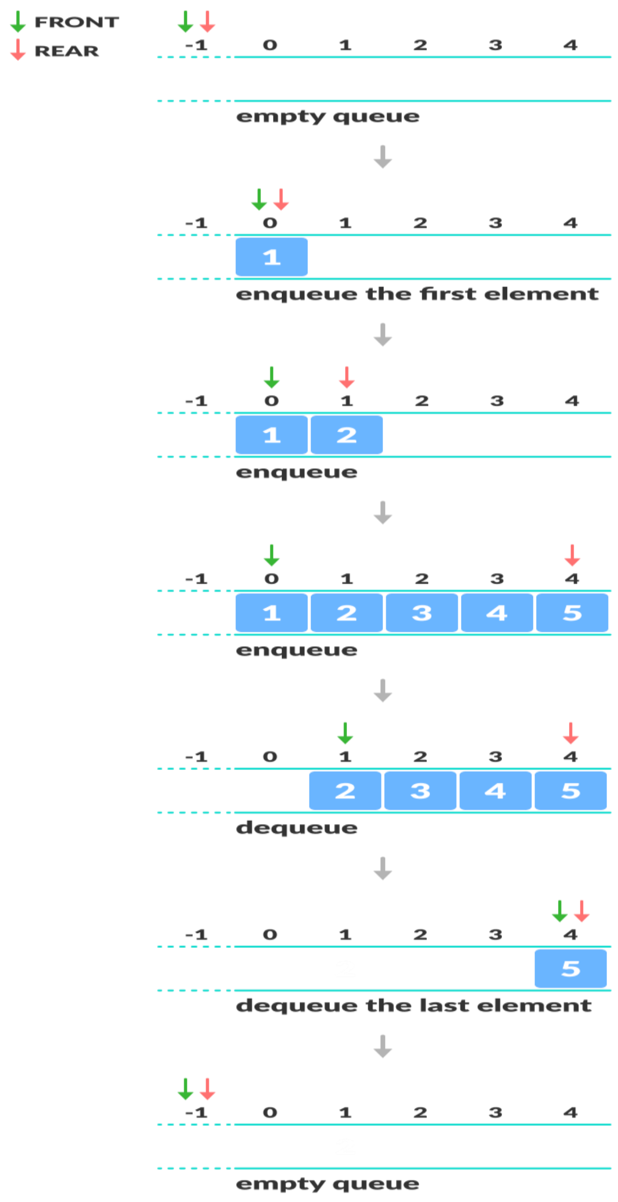
Enqueue: Add an element to the end of the queue
Dequeue: Remove an element from the front of the queue
IsEmpty: Check if the queue is empty
IsFull: Check if the queue is full
Peek: Get the value of the front of the queue without removing it

Queue operations work as follows:

- Two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

Enqueue Operation
- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

Dequeue Operation
- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1



Algorithm: Queue Implementation

- items[]: An array to store elements in the queue.
- front: An integer representing the front index of the queue.
- rear: An integer representing the rear index of the queue.

1. Initialization:

- Set front = -1 and rear = -1 to indicate an empty queue.

2. Check if Queue is Full (isFull Function):

```
   - If front is 0 and rear is SIZE - 1, return true (queue is full).
   - Otherwise, return false (queue is not full).
```

3. Check if Queue is Empty (isEmpty Function):

```
   - If front is -1, return true (queue is empty).
   - Otherwise, return false (queue is not empty).
```

4. Enqueue (enQueue Function):

```
   - If the queue is full, print "Queue is full."
   - If the queue is empty, set front = 0.
   - Increment rear.
   - Set items[rear] to the new element.
   - Print "Inserted [element]."
```

5. Dequeue (deQueue Function):

```
   - If the queue is empty, print "Queue is empty" and return -1.
   - Store items[front] in a variable (element).
   - If front is greater than or equal to rear, reset front and rear to -1 (empty queue).
   - Otherwise, increment front.
   - Print "Deleted -> [element]" and return the deleted element.
```

6. Display (display Function):

```
   - If the queue is empty, print "Empty Queue."
   - Otherwise:
     - Print "Front index -> [front]."
     - Print "Items -> [elements in the queue]."
     - Print "Rear index -> [rear]."
```

Linear vs Circular Queue

What is a Linear Queue?
A linear queue is a linear data structure that serves the request first, which has been arrived first. It consists of data elements which are connected in a linear fashion. It has two pointers, i.e., front and rear, where the insertion takes place from the front end, and deletion occurs from the front end.

Operations on Linear Queue
There are two operations that can be performed on a linear queue:

Enqueue: The enqueue operation inserts the new element from the rear end.
Dequeue: The dequeue operation is used to delete the existing element from the front end of the queue.

What is a Circular Queue?
As we know that in a queue, the front pointer points to the first element while the rear pointer points to the last element of the queue. The problem that arises with the linear queue is that if some empty cells occur at the beginning of the queue then we cannot insert new element at the empty space as the rear cannot be further incremented.

A circular queue is also a linear data structure like a normal queue that follows the FIFO principle but it does not end the queue; it connects the last position of the queue to the first position of the queue. If we want to insert new elements at the beginning of the queue, we can insert it using the circular queue data structure.

In the circular queue, when the rear reaches the end of the queue, then rear is reset to zero. It helps in refilling all the free spaces. The problem of managing the circular queue is overcome if the first position of the queue comes after the last position of the queue.

Conditions for the queue to be a circular queue

Front ==0 and rear=n-1
Front=rear+1
If either of the above conditions is satisfied means that the queue is a circular queue.

Operations on Circular Queue
The following are the two operations that can be performed on a circular queue are:

Enqueue: It inserts an element in a queue. The given below are the scenarios that can be considered while inserting an element:
If the queue is empty, then the front and rear are set to 0 to insert a new element.
If queue is not empty, then the value of the rear gets incremented.
If queue is not empty and rear is equal to n-1, then rear is set to 0.
Dequeue: It performs a deletion operation in the Queue. The following are the points or cases that can be considered while deleting an element:
If there is only one element in a queue, after the dequeue operation is performed on the queue, the queue will become empty. In this case, the front and rear values are set to -1.
If the value of the front is equal to n-1, after the dequeue operation is performed, the value of the front variable is set to 0.
If either of the above conditions is not fulfilled, then the front value is incremented.

Steps for Enqueue Operation in Circular Queue

Check if the queue is full.
If the queue is full then, the print queue is full.
If not, check the condition(rear == size -1 && front != 0)
If it is true then set the rear = 0 and insert the new element.
Algorithm to Insert an Element in a Circular Queue in Data Structure

```
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
```

Steps for Dequeue Operations in Circular Queue:

Firstly, check whether the queue is empty or not.
If the queue is empty then display the queue is empty.
If not, check the condition(rear == front)
If the condition is true then set front = rear = -1.
Else, front == size-1 and return the element.
Algorithm to Delete an Element from the Circular Queue in Data Structure

```
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
```

Lab Task

Task 1:

(A)

- Implement stack with array and insert 10 strings in stack
- Delete 4 elements in stack
- And now swap last element to first element in a stack
- Print all steps

(B) Create a stack using a linked list in C++

Task-2:

Implement a C++ class named Queue with enqueue, dequeue, front, and isEmpty functions, allowing users to add elements to the end, remove elements from the front, retrieve the front element without removal, and check if the queue is empty, respectively. Write utility function to pop top element from the stack

Task 3:

Insert at the bottom in the stack

Task 4:

Develop a program utilizing a queue to manage order processing, with functionalities including adding new orders in the queue, processing the next order, skipping cancelled orders, and displaying details of the current order.

Task 5:

Suppose P is an arithmetic expression written in postfix notation.

The following algorithm that uses a stack to hold operands, evaluate the VALUE of P.

1. Add the right parenthesis ")" at the end of P /* This acts as sentinel*/
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel ")" is encountered
3. If an operand is encountered, push it onto stack
4. If an operand OP is encountered, then:
   (a) remove the two top element of the stack (let, T be the top element and NT be the next-to-top element.)
   (b) evaluate NT OP T
   (c) place the result of (b) back on stack
5. Set value equal to the top element of stack
6. Exit

Task 7:

Develop a program to manage a system where customers join a virtual queue for customer support. Implement a C++ program using the queue data structure to handle customer requests, ensuring that they are served in the order they joined the queue. Write the enqueue and dequeue functions for this customer support system.