Data Structures Lab
Session 6

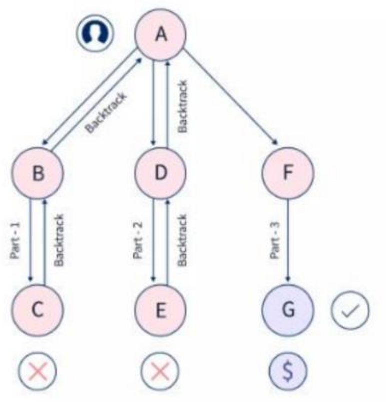
Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

Note:
- Lab manual cover following below elementary sorting algorithms: **{Backtracking, Applications of Stacks - Notations (Prefix, Infix, Postfix), Queue with Array and Linked, Circular Queue}**
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Get your lab checked at the end of the session.

Backtracking

A **backtracking** algorithm is a problem-solving algorithm that uses a **brute force approach** for finding the desired output. The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach. To understand this clearly, consider the given example. Suppose you are standing in **front of three roads**, one of which has a **bag of gold at its end**, but you don't know which one it is.

- Firstly, you will **go in Path 1**, if that is not the one, then come out of it, and **go into Path 2**, and again if that is not the one, come out of it and **go into Path 3**.
- So, let's say we are standing at 'A' and we **divided** our problem into three smaller sub-problems 'B', 'D' and 'F'.
- And using this sub-problem, we have **three possible paths** to get to our solution -- 'C', 'E', & 'G'. So, let us see how we can solve the problem using backtracking.



I. Choose the 1st path: A--> B--> C => Not found our feasible solution => C--> B--> A (backtrack and move back to A again)

II. After backtracking & choosing the 2nd path: A--> D--> E => Not found our feasible solution => E--> D--> A (backtrack and move back to A again)

III. Backtrack & choose the 3rd path: A--> F--> G => Got our solution

In backtracking we try to solve a **sub problem**, and if we don't reach the desired solution, then undo whatever we did for solving that sub problem, and try solving another sub problem, till we find the **best solution** for that problem (if any).

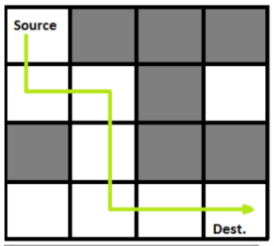
Rat in a Maze Problem:

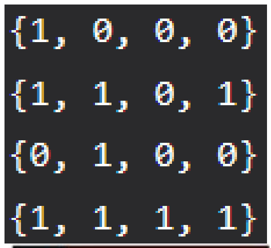
The rat in a maze problem is a **path finding puzzle** in which our objective is to find an o**ptimal path** from a starting point to an exit point. In this puzzle, there is a rat which is trapped inside a maze represented by a square matrix. The maze contains different cells through which that rat can travel in order to reach the exit of maze.

We use a backtracking algorithm to **explore all possible paths**. While exploring the paths we keep track of the directions we have moved so far and when we reach to the bottom right cell, we record the path in a vector of strings. A Maze is given as **N*N binary matrix** of blocks where source block is the upper left most block i.e., **maze[0][0]** and destination block is lower rightmost block i.e., **maze[N-1][N-1]**. A rat starts from its source and must reach its destination. The directions in which the rat can move are **'U'(up), 'D'(down), 'L' (left), 'R' (right)**.

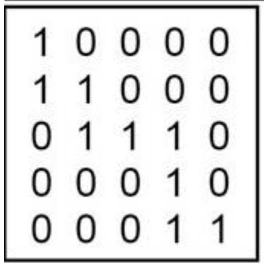
To solve the rat in a maze problem using the backtracking approach, follow the below steps –

- First, mark the starting cell as visited.
- Next, explore all directions to check if a valid cell exists or not.
- If there is a valid and unvisited cell is available, move to that cell and mark it as visited.
- If no valid cell is found, backtrack and check other cells until the exit point is reached.

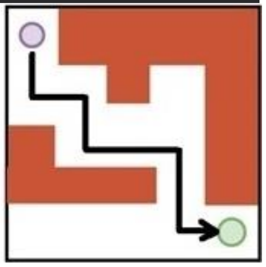




```
{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}
```



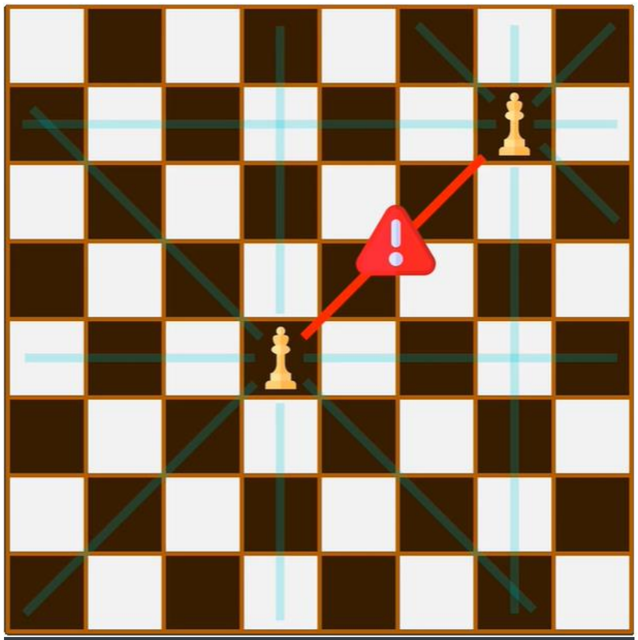
```
1 0 0 0 0
1 1 0 0 0
0 1 1 1 0
0 0 0 1 0
0 0 0 1 1
```

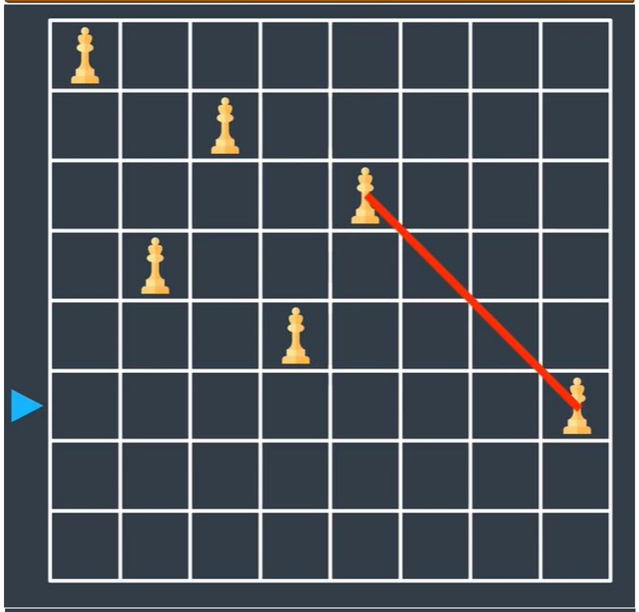


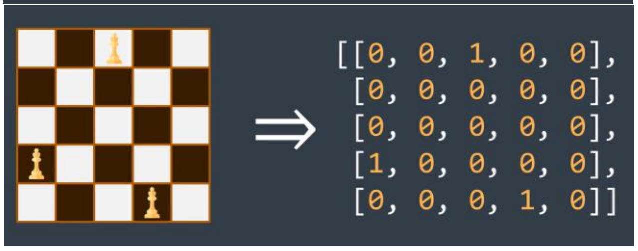
In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves. Following is the above-mentioned maze transformed into binary.

<u>N-Queen Problem:</u>

The N-Queens problem is a classic problem in computer science and combinatorial optimization. The goal is to place N queens on an N×N chessboard in such a way that **no two queens threaten each other**. In other words, no two queens can share **the same row, column, or diagonal**.

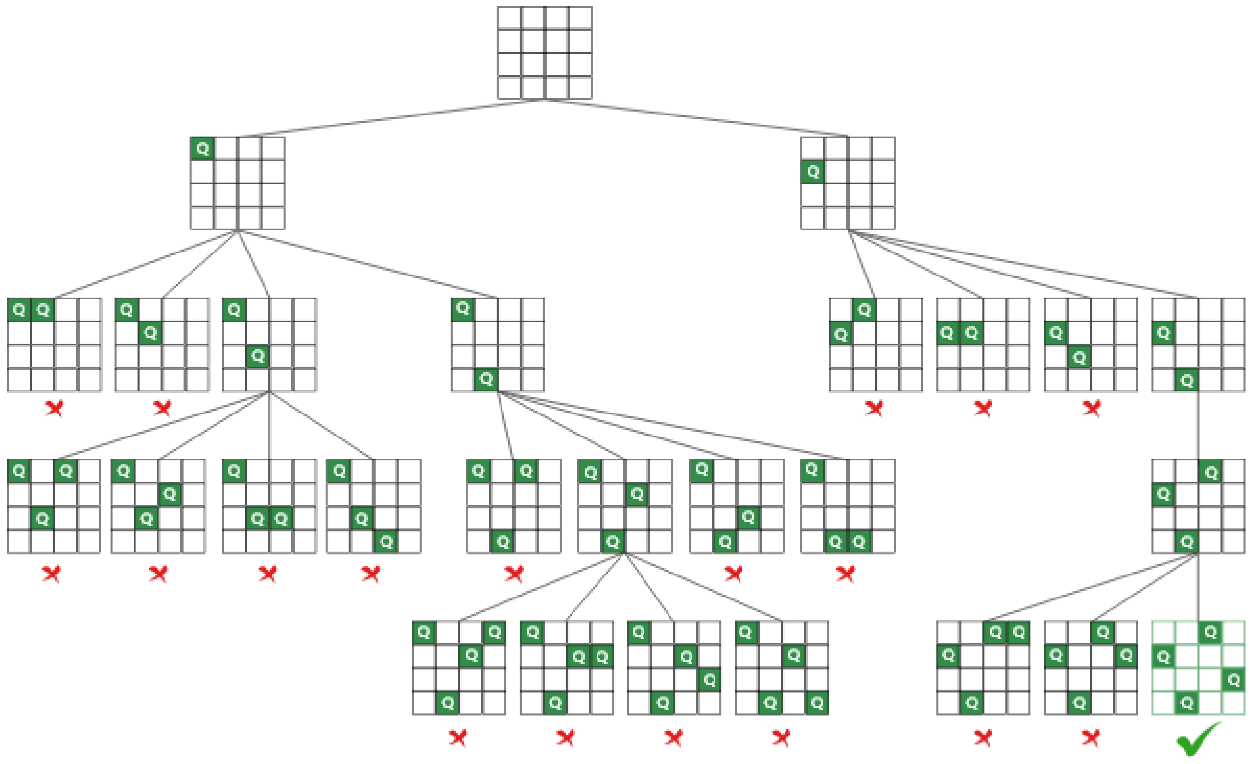






```
[[0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0],
 [0, 0, 0, 1, 0]]
```

Working of Algorithm:



Difference between Recursion and Backtracking:

There is no major difference between these two in terms of their approach, except for what they do:

- Backtracking uses **recursion to solve** its problems. It does so by exploring **all the possibilities** of any problem, unless it finds the best and most feasible solution to it.

- Recursion occurs when a function calls itself **repeatedly** to split a problem into smaller sub-problems, until it reaches the **base case**.

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.

Infix Notation:

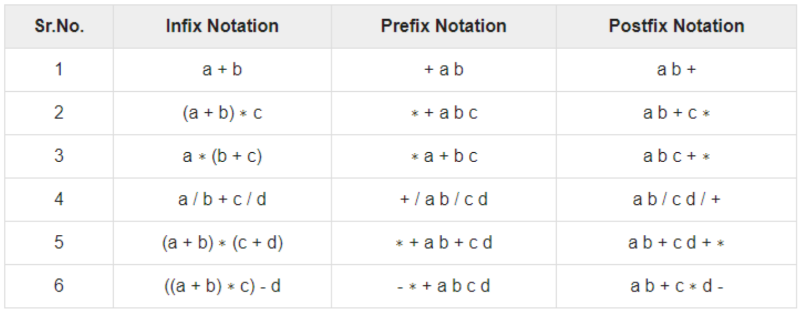
We write expression in **infix** notation, e.g. **a - b + c**, where operators are used **in**-between operands. It is **easy for us humans to read, write, and speak in infix notation** but the same does not go well with computing devices. An algorithm to process infix notation could be **difficult and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

Prefix Notation:

In this notation, operator is **prefixed** 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**.

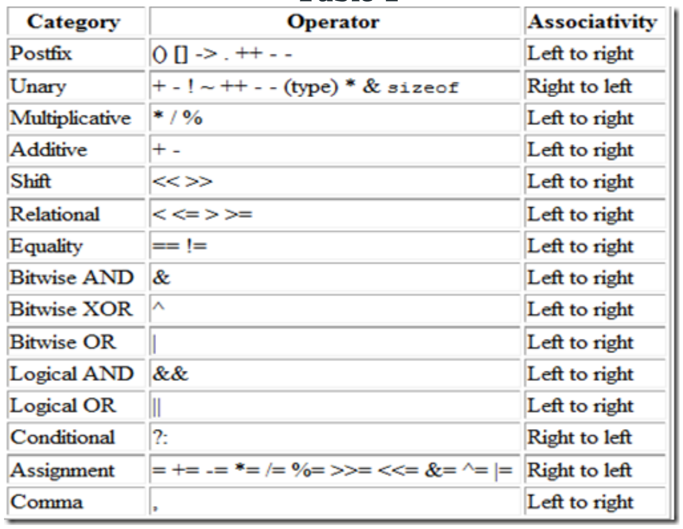
Postfix Notation:

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** 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 -

Table-1



Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Precedence and Associativity Table – 2

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 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.

Working of Algorithm:

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.

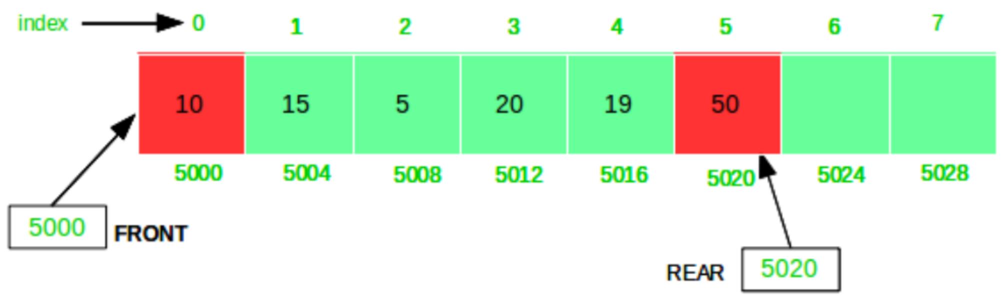
Reference: https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/

Linear Queue with Array

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.

A linear queue is a type of queue where the elements are stored in a **contiguous memory location**, like 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)**. Mainly the following three basic operations are performed in the queue:

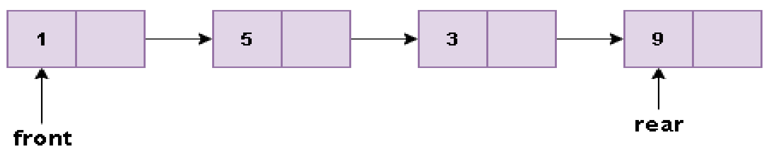
- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.

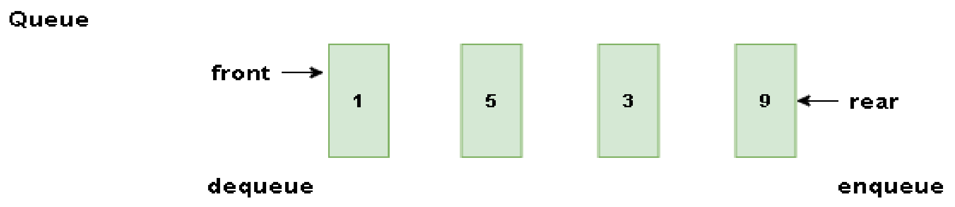


Sample Code of Queue in Array:

```
const int MAX = 100

class Queue {
  public:
    int front, rear, size
    int arr[MAX]

    Queue() {
        front = 0
        rear = -1
        size = 0
    }
    bool enqueue(int value) {
        if (isFull()) {
            print "Queue Overflow"
            return false
        } else {
            rear = rear + 1
            arr[rear] = value
            size = size + 1
            print value
            return true
        }
    }
    int dequeue() {
        if (isEmpty()) {
            print "Queue Underflow"
            return -1
        } else {
            int value = arr[front]
            front = front + 1
            size = size - 1
            print value
            return value
        }
    }

    bool isEmpty() {
        return (size == 0 || front > rear)
    }

    bool isFull() {
        return (size == MAX || rear == MAX - 1)
    }
    int frontElement() {
        if (isEmpty()) {
            print "Queue is empty"
            return -1
        } else {
            return arr[front]
        }
    }

    void display() {
        if (isEmpty()) {
            print "Queue is empty"
            return
        } else {
            print "Queue elements: "
            for i = 0 to size-1
                print arr[front + i]
            }
        }
    }
};
```

Linear Queue with linked list





Sample Code of Queue in Linked List:

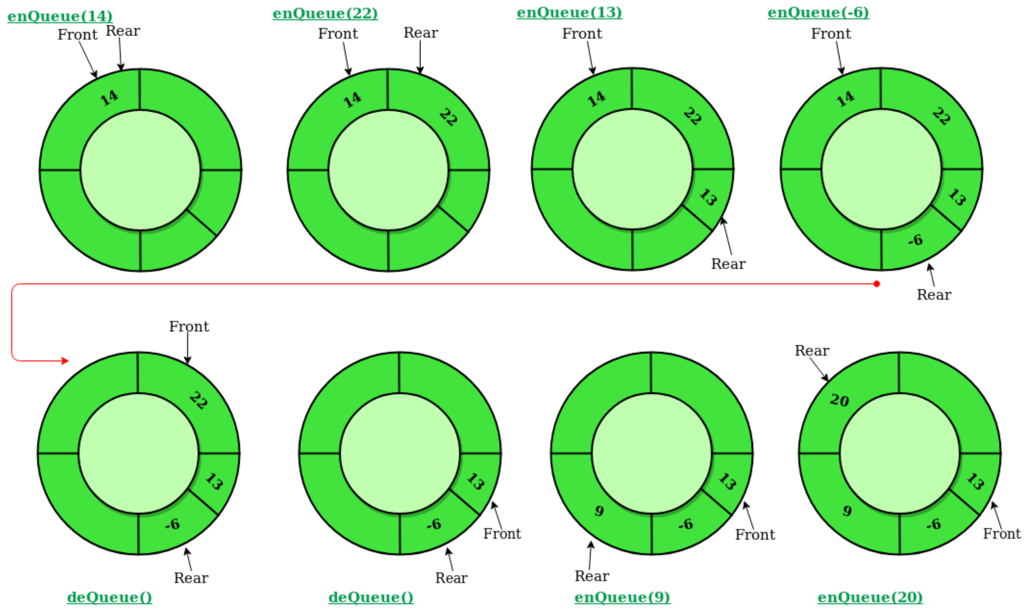
```
class Node {
    int data
    Node* next

    Node(int value) {
        data = value
        next = nullptr
    }
}
class Queue {
    Node* front
    Node* rear
    int size
    Queue() {
        front = nullptr
        rear = nullptr
        size = 0
    }
    bool enqueue(int value) {
        Node* newNode = new Node(value)
        if (isEmpty()) {
            front = rear = newNode
        } else {
            rear->next = newNode
            rear = newNode
        }
        size = size + 1
        print value
        return true
    }
    int dequeue() {
        if (isEmpty()) {
            print "Queue Underflow"
            return -1
        } else {
            int value = front->data
            Node* temp = front
            front = front->next
            delete temp
            size = size - 1
            print value
            return value
        }
    }
    bool isEmpty() {
        return (front == nullptr)
    }
    int frontElement() {
        if (isEmpty()) {
            print "Queue is empty"
            return -1
        } else {
            return front->data
        }  }
    void display() {
        if (isEmpty()) {
            print "Queue is empty"
            return
        } else {
            print "Queue elements: "
            Node* current = front
            while (current != nullptr) {
                print current->data
                current = current->next

                if (current != nullptr) {
                    print " -> "     elements
                }
            }
            print newline
        }
    }};
```

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**. In a linear queue **implemented using an array**, the space at the front cannot be reused once an element is dequeued, leading to inefficient memory usage. 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

Sample Code of Circular Queue:

```
const int MAX = 100
class CircularQueue {
    int front, rear, size
    int arr[MAX]
    CircularQueue() {
        front = -1
        rear = -1
        size = 0
    }
    bool enqueue(int value) {
        if (isFull()) {
            print "Queue Overflow"
            return false
        } else {
            if (isEmpty()) {
                front = 0
            }
            rear = (rear + 1) % MAX
            arr[rear] = value
            size = size + 1
            print value + " enqueued to the queue"
            return true
        }
    }
    int dequeue() {
        if (isEmpty()) {
            print "Queue Underflow"
            return -1
        } else {
            int value = arr[front]
            if (front == rear) { // Empty Queue
                front = -1
                rear = -1
            } else {
                front = (front + 1) % MAX
            }
            size = size - 1
            print value
            return value
        }
    }
    bool isEmpty() {
        return (size == 0 || front == -1)
    }
    bool isFull() {
        return (size == MAX || ((rear+1) % size == front))
    }
    int frontElement() {
        if (isEmpty()) {
            print "Queue is empty"
            return -1
        } else {
            return arr[front]
        }
    }
    void display() {
        if (isEmpty()) {
            print "Queue is empty"
            return
        } else {
            print "Queue elements: "
            for i = 0 to size-1 {
                print arr[(front + i) % MAX]  } }}};
```

<u>Applications of Queue</u>

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.

Lab Tasks

Q1: Implement Queue with Linked List:

- Insert 10 Integers values in the queue.
- Write a utility function to display all the inserted integer values in the linked list in forward and reverse direction both.
- Write utility function to dequeue front element from the queue.

Q2. 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).

Q3. 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 equal 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).

Q4. In Table 1 Given above you are given an infix notation of ((a + b) * c) - d and you are required to convert it into postfix notation. Attempt to solve the question using queues and linked lists. (Assume a Tree when solving this approach).

Q5: Imagine you are working on a scientific application that involves processing a multi-dimensional array. You need to implement a function to calculate the sum of all elements in the array, which may contain nested arrays (creating a multi-dimensional structure using a jagged array). Write a C++ function int recursiveArraySum(int* arr[], int sizes[], int dim) that calculates the sum of all elements in a multi-dimensional array represented by a jagged array. The function should work for arrays containing nested arrays, and sizes is an array that contains the sizes of each dimension, and dim is the current dimension being processed.

Q6. Imagine you are given a grid representing a maze, where '0's represent obstacles (red highlighted) or walls, and '1's represent open paths or empty cells. You have two arrays at your disposal: an empty Solution array and a maze represented by the following grid:

Your task is to guide a lion through this maze from the starting point at (0, 0) to reach a piece of meat located at the destination point (4, 4). The lion is also not allowed to move diagonally. The lion can only move through open paths ('1's) while avoiding obstacles ('0's).

	0		0	
0		0		
	0	0		
			0	

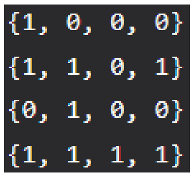
Please provide a C++ code that uses a backtracking algorithm to navigate the maze and move the lion to the meat. Additionally, display the contents of the Solution array after the lion has reached the meat.

Q7: Imagine a 4x4 grid where a person is on a mission to collect as many flags as possible. Each cell of the grid represents a potential location for placing a flag. However, there are constraints:

- Only one flag can be placed in each row or column.
- Additionally, no two flags can ever be placed on the same diagonal (i.e., no two flags can threaten each other diagonally).
- Your goal is to design a backtracking algorithm in C++ to help this person maximize the number of flags collected while adhering to these constraints. After implementing the algorithm, provide the code and report the maximum number of flags that can be placed on the 4x4 grid.

Q8. Solve the rat in a maze problem:

- Design the function with recursive approach to find the number of existing destination path in the below matrix:



```
{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}
```

- Design the function with recursive approach to find all-possible solutions in N-Queen Problem.