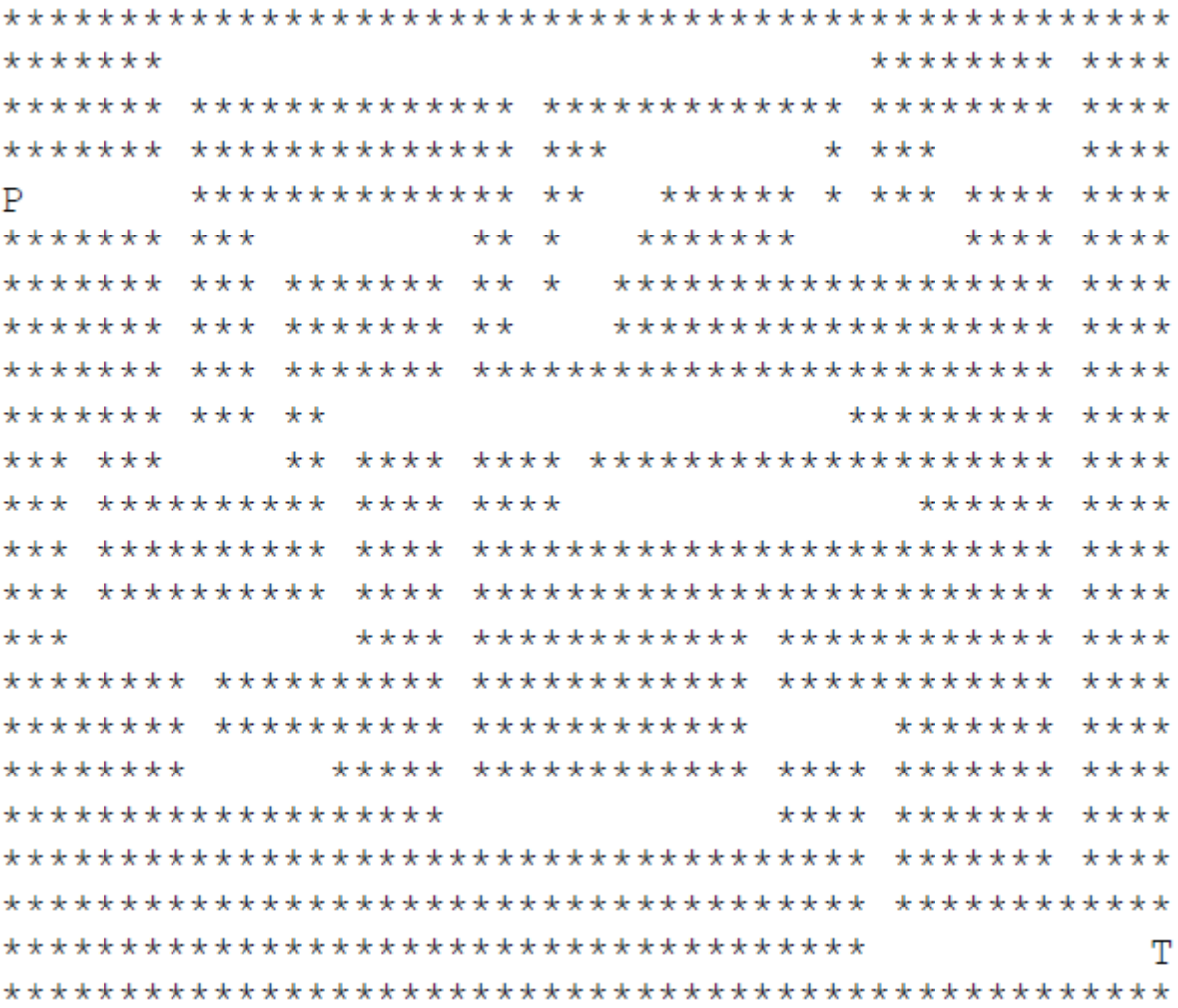
Lab 04
Data Structures
BS DS Fall 2024 Morning/Afternoon

Task 1 **20 marks**

Write a function that solves a maze problem. In this particular version of the problem, a hiker must find a path to the top of a mountain. Assume that the hiker leaves a parking lot, marked **P**, and explores the maze until he reaches the top of a mountain, marked **T**.



The maze is represented by a 2D array of characters, the character * marks a barrier, and P and T mark the parking lot and mountain top, respectively. A blank space marks a step along a path. Your program then attempts to find a path through the maze and returns "solved" or "unsolved" depending on the outcome and if the maze is solved it should print the complete path.

You have to design a **solve_maze** function.

```
void solve_maze(char maze[6][6], Pair s, Pair d, int row, int col)
```

Test your function using the following driver program

```
int main() {
        char maze1[6][6] = {
         {' ', '*', ' ', '*', ' ', ' '},
         {' ', '*', ' ', '*', ' ', ' '},
         {'P', ' ', ' ', ' ', '*', ' '},
         {'*', ' ', '*', '*', '*', ' '},
         {' ', ' ', ' ', ' ', '*', 'T'},
         {'*', ' ', ' ', ' ', ' ', ' '}
    };

    Pair start = {2,0};
    Pair dest  = {4,5};

    solve_maze(maze1, start, dest, 6, 6);

    char maze2[6][6] = {
         {' ', '*', ' ', '*', ' ', ' '},
         {' ', '*', ' ', '*', ' ', ' '},
         {'P', ' ', ' ', ' ', '*', ' '},
         {'*', '*', '*', '*', '*', ' '},
         {' ', ' ', ' ', ' ', '*', 'T'},
         {'*', ' ', ' ', ' ', ' ', ' '}
            };

     solve_maze(maze2, start, dest, 6, 6);

        return 0;
}
```

The output of the following program should be

```
Solved
(2,0) (2,1) (3,1) (4,1) (4,2) (4,3) (5,3) (5,4) (5,5) (4,5)
UnSolved
```

Task 2 — 20 marks

String words Reverse

Your task is to implement a function that receives a string and reverses each word in it using stack. You can assume that the string only consists of alphabets and spaces. The order of the words should remain same but characters within each word should get reversed.

For example:
String: “Welcome to DSA”
Modified string: “emocleW ot ASD”

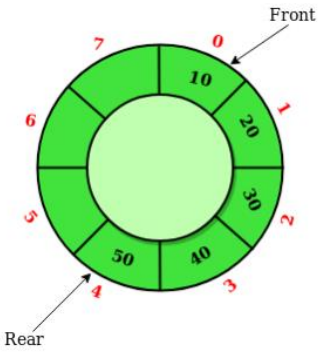
```
string reverseWords(const string &str)
```

Task 3 **10 marks**

Circular Queue

A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. It is also called **'Ring Buffer'**.



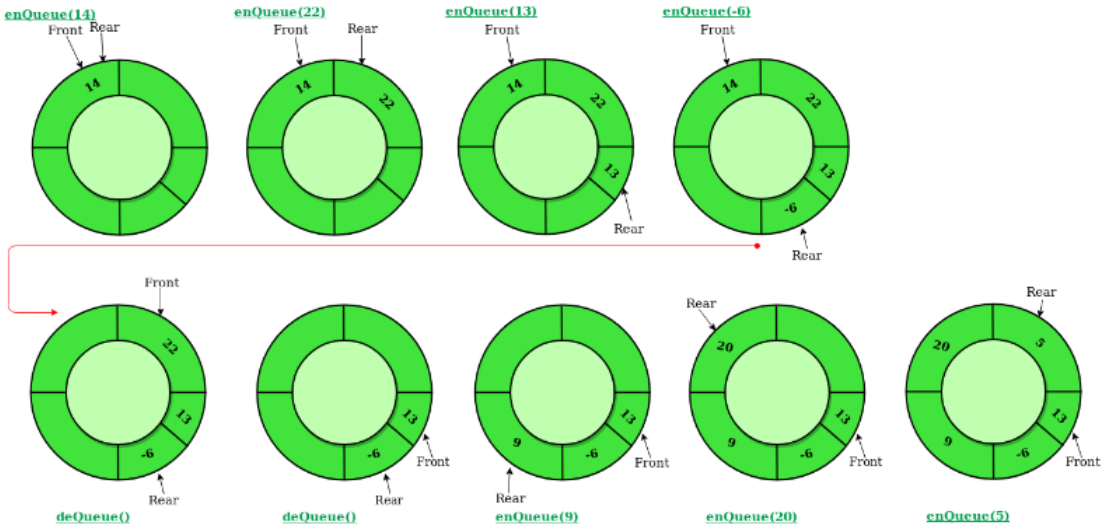
In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

Operations on Circular Queue:

- **Front:** Get the front item from the queue.
- **Rear:** Get the last item from the queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
    - Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
    - If it is full then display Queue is full.
    - If the queue is not full then, insert an element at the end of the queue.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
    - Check whether the queue is Empty.
    - If it is empty then display Queue is empty.
    - If the queue is not empty, then get the last element and remove it from the queue.

Illustration of Circular Queue Operations:

Follow the below image for a better understanding of the enqueue and dequeue operations.



Working of Circular queue operations

You have to complete the following ADT

```
template<class T>
class Queue
{
private:
        int capacity;
        int rear;
        int front;
        int noOfElement;
        T *data;
public:
        Queue(int cap);
        Queue(Queue<T> &ref);
        void enQueue(T val);
        T deQueue();
        int getNoOfElement();
        int isFull();
        int isEmpty();
        T getFront();
        T getRear();
        Queue<T> operator=(Queue<T> &ref);
        ~Queue();
        void display()
        {
                cout<<"Elements in the circular queue are: ";
                for(int i=0;i<noOfElement;i++)
                        cout<<data[(i+front)%capacity]<<" ";
                cout<<endl;
        }
};
```

Test your function using the following driver program

```
int main()
{
        Queue<int> q(5);

        q.enQueue(14);
        q.enQueue(22);
        q.enQueue(13);
        q.enQueue(-6);
        q.display();
        cout<<"Deleted Value = "<<q.deQueue()<<endl;
        cout<<"Deleted Value = "<<q.deQueue()<<endl;
        q.display();

        q.enQueue(9);
        q.enQueue(20);
        q.enQueue(5);
        q.display();
        return 0;
}
```

The output of the following program should be

```
Elements in the circular queue are: 14 22 13 -6
Deleted value =  14
Deleted value =  22
Elements in the circular queue are: 13 -6
Elements in Circular Queue are: 13 -6 9 20 5
```

Task 4 — 30 marks

There are *n* people in a line queuing to buy tickets, where the 0^{th} person is at the front of the line and the $(n - 1)^{th}$ person is at the back of the line.

You are given a ***0-indexed integer array*** tickets of length *n* where the number of tickets that the i^{th} person would like to buy is ***tickets[i].***

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets. If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person at position *k* (0-indexed) to finish buying tickets.

Example 1:
Input: tickets = [2,3,2], k = 2
Output: 6

Example 2:
Input: tickets = [5,1,1,1], k = 0
Output: 8

Bonus Task — 50 marks

In this Task you are required to simulate ***Schedule***r. Scheduler is something which is responsible for assigning the CPU time to the processes.

The ***Scheduler*** should take in ***Processes*** and add them to a Queue. o Once all the ***Processes*** are read, your program would execute each.

But before executing any process, the program should print the name and other information about the process.
Now it should start executing the processes in the Queue. For each process, execution just means that process stays at the head of the Queue until time equal to its ***execution time*** has passed. The process is deleted from the list after that. At this moment your program should print that such and such process has finished execution. E.g.
Messenger.exe, 6 completed execution
You must remember that this is a simulator. This means that you'll have to define your own timer, one which increments in unit steps.
After the passage of every 10 time units, you program must stop the processing of current process and start the next process, the current process will be taken from the front of the queue to the end of the queue and the time remaining to complete process should be decremented.

Hints:
Design a class "Process" which will have following data members,

processId
processName
executionTime
And a method to display its state.

Create a circular Queue and add the process in it.
Process the "Process" at front of Queue and then remove it from front and add at the end and continue the processing.
Remove the "Process" completely from queue if it has "0" "execution time" remaining.
Repeat this process until the Queue is not empty.

*You have to create the following files and will have to code all the classes listed in these files.

Process.h

```
class Process {
private:
    int processId;
    string processName;
    int executionTime;
public:

    Process() {
        processId = 0;
        processName = "";
        executionTime = 0;
    }

    Process(int id, string name, int time);

    int getId();
    string getName();
    int getExecutionTime();
    void setExecutionTime(int t);
};
```

Scheduler.h

```
class Scheduler {
private:
    Queue<Process> q;
    int timeQuantum;
public:
    Scheduler(Process arr[], int len, int tq);
    void assignProcessor()
};
```

Queue // use the task 3 queue

main.cpp

```
int main()
{
      Process arr[] = {
         Process(1, "notepad.exe", 20),
         Process(13, "mp3player.exe", 5),
         Process(4, "bcc.exe", 30),
         Process(11, "explorer.exe", 2)
    };


    Scheduler s(arr, 4, 5);  // time quantum = 5
    s.assignProcessor();
      return 0;
}
```

The output of this program should be:

```
Executing process notepad for 5 units
Executing process mp3player for 5 units
Executing process bcc for 5 units
Executing process explorer for 2 units
Executing process notepad for 5 units
Executing process bcc for 5 units
Executing process notepad for 5 units
Executing process bcc for 5 units
Executing process notepad for 5 units
Executing process bcc for 5 units
Executing process bcc for 5 units
Executing process bcc for 5 units
```