**LAB SESSION 3: QUEUE DATA STRUCTURE**

**AIM:** To implement

1. Efficient Circular Queue
2. Task Scheduler using Queue Data Structure

**PROBLEM DEFINITION:**

1. Develop a C program to implement Efficient Circular Queue. Include the following functions:
   * 1. Insert
     2. Delete
     3. Peek
     4. Display

    2. Create a Structure TASK with the following data fields:

          TaskId(int)

          TaskTitle(char [])

          TaskDuration (int) to be interpreted in seconds

          Status: Idle/ Queued/Completed

Create an array of TASK comprising of 10 tasks. (Preferably take input from file or initialise at compile time).

Implement a Queue data structure to schedule task from the above array. Max size of Queue is 5. Your options should include the following:

1. Enter the task id of the tasks to be scheduled (insert these in a queue). If the status is queued or completed ten do no allow scheduling. If the queue is full, ask the user to wait for a certain amount of time (Print the minimum time of waiting and the max waiting time to the user)
2. Run the Tasks: delete the task at start of the queue and run it for the given taskDuration. (Use delay() function to suspend the running of the program for the indicated time). Once the task is completed prompt the user the menu and continue based on his choice.
3. Display the details of tasks that are queued up.

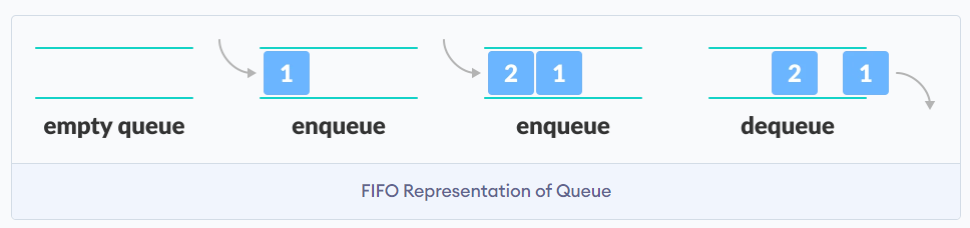
**THEORY:**

Queue is a data structure that follows a particular order in which the operations are performed for storing data. The order is First In First Out **(FIFO)**. In this data structure we have a Head and a tail pointer which refers to the element at start and end of the queue. To retrieve or delete data from the stack we have only one position that is the TOP position.

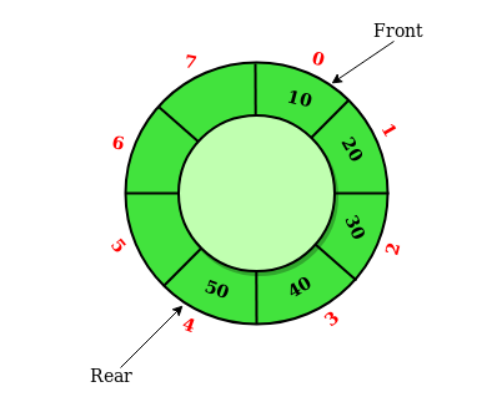
**Basic operations of Queue:**

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



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 cannot insert the next element even if there is a space in front of queue.



**Basic operations of 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.

**ALGORITHMS**

**Efficient Circular Queue**

1) Inserting

Function: insert(int item)

Input: item - the item to be inserted

1. Check if the circular list is full:

- Call isFull function.

- If isFull returns true:

- Print "Circular list full!".

- Return.

2. If front is -1 (indicating an empty list):

- Set front to 0.

3. Calculate the new position for rear using circular indexing:

- Increment rear by 1.

- Apply modulo operation (% MAX) to ensure it stays within bounds.

4. Insert the item into the circular list:

- Assign the value of 'item' to CQ\_arr[rear].

**End of Function**

2) Deleting

1. Declare a variable 'item' of type int.

2. Check if the circular queue is empty:

- Call the 'isEmpty' function.

- If 'isEmpty' returns true:

- Print "Circular Queue Underflow!".

- Return NULL.

3. Retrieve the item from the front of the circular queue:

- Assign 'CQ\_arr[front]' to 'item'.

4. Check if front and rear are equal:

- Set front to -1.

- Set rear to -1.

5. Update front to the next position:

- Increment front by 1.

- Apply modulo operation (% MAX) to ensure it stays within bounds.

6. Return the deleted item.

3) Peeking

1. Check if the circular queue is empty:

- Call the 'isEmpty' function.

- If 'isEmpty' returns true:

- Print "stack underflow!".

- Exit the program with status code 1.

2. Return the element at the front of the circular queue:

- Return 'CQ\_arr[front]'.

4) Displaying

1. Declare a variable 'i' of type int.

2. Check if the circular queue is empty:

- Call the 'isEmpty' function.

- If 'isEmpty' returns true:

- Print "Queue underflow!".

- Return.

3. Set 'i' to the value of 'front'.

4. Check if front is less than or equal to rear:

- In this case, iterate over the elements from 'i' to 'rear':

- Print 'CQ\_arr[i]'.

- Increment 'i'.

5. If front is greater than rear (indicating wrap-around):

- Iterate over the elements from 'i' to 'MAX - 1':

- Print 'CQ\_arr[i]'.

- Increment 'i'.

- Reset 'i' to 0.

- Iterate over the elements from 'i' to 'rear':

- Print 'CQ\_arr[i]'.

- Increment 'i'.

**Task Schedule using Queue Data Structure**

1) Schedule a Task

1. Declare a variable 'ID' of type int.

2. Print the prompt "Enter Task ID: ".

3. Read the input 'ID' from the user.

4. Iterate over 'i' from 0 to 9 (inclusive):

a. If (task + i)->taskID is equal to 'ID':

i. Check if the task is in Queued or Completed status:

- If yes, print "Task cannot be scheduled" and return.

ii. Check if the circular queue is full:

- If yes, print "Task scheduler queue is Full" and return.

iii. If the front of the queue is -1 (indicating an empty queue):

- Set queue.front to 0.

iv. Calculate the new position for 'rear' using circular indexing:

- Increment queue.rear by 1.

- Apply modulo operation (% MAX) to ensure it stays within bounds.

v. Assign (task + i) to queue.task[queue.rear].

vi. Update the status of the task in the queue to Queued.

vii. Print "Task with ID <ID> is scheduled".

viii. Return.

5. If no task with ID 'ID' is found in the loop, print "Task with ID <ID> Not found!".

2) Run a Task/Deque

1. Declare a pointer variable 'item' of type Task.

2. Check if the task scheduler queue is empty:

a. Call the 'isEmpty' function.

b. If 'isEmpty' returns true:

- Print "Task Scheduler Queue Underflow!".

- Return.

3. Assign the task at the front of the queue to 'item':

- Set 'item' as 'queue.task[queue.front]'.

4. Check if the front and rear of the queue are equal (indicating the last item in the queue):

a. If yes, set front and rear to -1 to indicate an empty queue.

5. Update front to the next position using circular indexing:

- Increment 'queue.front' by 1.

- Apply modulo operation (% MAX) to ensure it stays within bounds.

6. Print information about the running task:

- Print the task's title and the time required for execution.

- Use 'item->taskTitle' and 'item->delay\_t'.

7. Simulate a delay for the time required for execution:

- Call 'delay' function with 'item->delay\_t' as an argument.

8. Update the status of the task to Completed:

- Set 'item->status' to Completed.

9. Print a message indicating successful completion of the task:

- Print "Task with ID <item->taskID> was completed successfully".

3) Display all Tasks

Function: void display(Task \*item)

Input: item - a pointer to a Task structure

1. Print task information:

- Print "Task Title: <item->taskTitle>".

- Print "Task ID : <item->taskID>".

End of Function

Function: void Display\_Queue()

1. Declare a variable 'i' of type int.

2. Check if the task scheduler queue is empty:

a. Call the 'isEmpty' function.

b. If 'isEmpty' returns true:

- Print "Task Scheduler Queue underflow!".

- Return.

3. Set 'i' to the value of 'queue.front'.

4. Check if front is less than or equal to rear (indicating no wrap-around):

- In this case, iterate over the elements from 'i' to 'queue.rear':

- Call 'display' with 'queue.task[i]' as argument.

- Increment 'i'.

5. If front is greater than rear (indicating wrap-around):

- Iterate over the elements from 'i' to 'MAX - 1':

- Call 'display' with 'queue.task[i]' as argument.

- Increment 'i'.

- Reset 'i' to 0.

- Iterate over the elements from 'i' to 'queue.rear':

- Call 'display' with 'queue.task[i]' as argument.

- Increment 'i'.

**PROGRAM AND OUTPUT:**

#include<stdio.h>

#include<string.h>

#include<time.h>

#include<stdlib.h>

#define MAX 5

enum Status{Idle, Queued, Completed};

typedef struct TASK{

int taskID;

char taskTitle[20];

int delay\_t;

enum Status status;

} Task;

Task task[10];

typedef struct {

Task \*(task)[5];

int front;

int rear;

}Queue;

Queue queue;

int isFull(){

if((queue.front == 0 && queue.rear == MAX -1) || queue.front == queue.rear + 1 )

return 1;

else return 0;

}

int isEmpty(){

if(queue.rear == -1)

return 1;

else

return 0;

}

void delay(int time){

long pause;

clock\_t time1,time2;

pause = time ;

time2 = time1 = clock();

while( (time1-time2) < pause ){

time1 = clock();

}

}

void init\_tasks(){

strcpy(task[0].taskTitle, "something1");

task[0].taskID = 1001;

task[0].delay\_t = 5;

strcpy(task[1].taskTitle,"something2");

task[1].taskID = 1002;

task[1].delay\_t = 6;

strcpy(task[2].taskTitle,"something3");

task[2].taskID = 1003;

task[2].delay\_t = 7;

strcpy(task[3].taskTitle,"something4");

task[3].taskID = 1004;

task[3].delay\_t = 8;

strcpy(task[4].taskTitle,"something5");

task[4].taskID = 1005;

task[4].delay\_t = 2;

strcpy(task[5].taskTitle,"something6");

task[5].taskID = 1006;

task[5].delay\_t = 7;

strcpy(task[6].taskTitle,"something7");

task[6].taskID = 1007;

task[6].delay\_t = 10;

strcpy(task[7].taskTitle,"something8");

task[7].taskID = 1008;

task[7].delay\_t = 11;

strcpy(task[8].taskTitle,"something9");

task[8].taskID = 1009;

task[8].delay\_t = 4;

strcpy(task[9].taskTitle,"something10");

task[9].taskID = 1010;

task[9].delay\_t = 9;

queue.front = -1;

queue.rear = -1;

}

void Task\_Scheduler(){

int ID;

printf("Enter Task ID: ");

scanf("%d",&ID);

for(int i = 0; i<10; i++){

if((task+i)->taskID == ID){

if((task+i)->status == Queued || (task+i)->status == Completed){

printf("\nTask cannot be scheduled\n");

return;

}

else if(isFull()){

printf("\nTask scheduler queue is Full\n");

return;

}

else{

if(queue.front == -1)

queue.front = 0;

queue.rear = (queue.rear + 1) % MAX;

queue.task[queue.rear] = (task+i);

queue.task[queue.rear]->status = Queued;

printf("Task with ID %d is scheduled\n",ID);

return;

}

}

}

printf("\nTask with ID %d Not found !\n",ID);

}

void Task\_Run(){

Task \*item;

if(isEmpty()){

printf("\nTask Scheduler Queue Underflow!\n");

return ;

}

item = queue.task[queue.front];

if(queue.front == queue.rear){

queue.front = -1;

queue.rear = -1;

}

queue.front = (queue.front + 1) % MAX ;

printf("%s is Running...\nTime required : %d\n",item->taskTitle,item->delay\_t);

delay(item->delay\_t);

item->status = Completed;

printf("Task with ID %d was completed successfully\n",item->taskID);

}

void display(Task \*item){

printf("\nTask Title: %s\n",item->taskTitle);

printf("Task ID : %d\n",item->taskID);

}

void Display\_Queue(){

int i;

if(isEmpty()){

printf("Task Scheduler Queue underflow!\n");

return;

}

i = queue.front;

if(queue.front <= queue.rear){

while(i <= queue.rear){

display(queue.task[i]);

i++;

}

}

else{

while(i <= MAX-1){

display(queue.task[i]);

i++;

}

i = 0;

while(i <= queue.rear){

display(queue.task[i]);

i++;

}

}

}

int main()

{

init\_tasks();

int op;

while(1)

{

printf("\n\*\*\*\*\*MENU\*\*\*\*\*\n");

printf("1. Schedule Task\n2. Run Task\n3. Display Queue\n4.Exit\n");

scanf("%d",&op);

switch(op)

{

case 1:

Task\_Scheduler();

break;

case 2:

Task\_Run();

break;

case 3:

Display\_Queue();

break;

case 4: return 0;

}

}

}

**CONCLUSION:** In this lab, we successfully implemented an efficient Circular Queue and utilized it to create a Task Scheduler. The Queue Data Structure proved essential for managing tasks effectively, demonstrating the power of well-structured data organization in real-world applications.