21/11/2023
Tuesday

QUEUE

Queue is a linear data structure that follows FIFO (First In First Out) principle.

Queue STL -

Header file = #include<queue>

Creation = ~~queq~~ queue<data_type>queue_name;

Eg- queue<int>q;
queue<char>q;
queue<string>q;
queue<bool>q;

Insertion = q.push(data);

Remove = q.pop();

Size = q.size();

Check if empty = q.empty();

Front = q.front();

Back or rear = q.back();

Code -

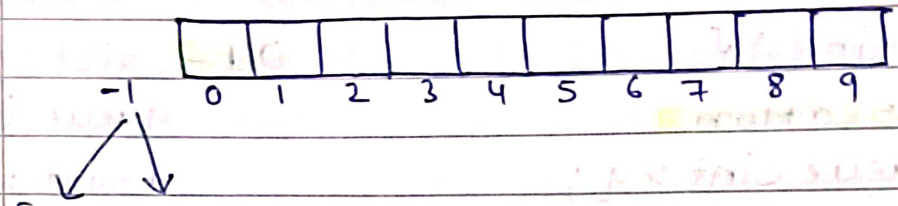
```
#include <iostream>
#include <queue>
using namespace std;

int main (){
  //creation
  queue <int> q;

  //insert
  q.push(10);
  q.push(20);
  q.push(30);

  //remove
  q.pop();

  //size
  cout <<"size:"<< q.size() <<endl;

  //empty
  if (q.empty()){
      cout << "queue is empty" <<endl;
    }
  else{
     cout << "queue is not empty"<<endl;
    }

  //front
  cout<<"front: "<< q.front() <<endl;

  // rear
  cout<<"rear: " << q.back() <<endl;
}
```

O/P -

```
size: 2
queue is not empty
front : 20
rear : 30
```

Note - In Queue, push rear mai hota hai.
Pop front se hota hai.

* **Queue implementation using array -**



= Initially, front and rear -1 pr hai.

= **Push in queue -**

- If q.empty() and first element insert krna hai. In that case -

```
front ++;
rear ++;
arr [rear] = val;
```

- **Normal case -**

```
rear ++;
arr [rear] = val;
```

- **Overflow ka case -**

```
rear == size - 1;
queue is full, overflow.
```

Pop from queue -

- **Queue is empty -**

(front == -1 && rear == -1) - condition for empty queue is empty, underflow.

- **Single element ka case -**

```
(front == rear)
 arr [front] = -1;
 front = -1 ;
 rear = -1;
```

- **Normal case -**

```
arr [front] = -1;
front ++;
```

Size of queue -

- **Queue is empty -**

```
(front == -1 && rear == -1)
  return 0;
```

- **Queue has element -**

```
return rear - front + 1;
```

= Front element in queue -

- Empty ka case -

(front == -1)
no element in queue, no front

- Non-empty queue -

return arr[front];

= Rear element in queue -

- Empty ka case -

(rear == -1)
no element in queue, no rear

- Non-empty queue -

return arr[rear];

= check if queue is empty -

if(front == -1 && rear == -1)
queue is empty, return true;
else return false;

Code -

```
class Queue {
    public:
    int* arr;
    int size;
    int front;
    int rear;

Queue (int size){
    arr = new int [size];
    this -> size = size;
    front = -1;
    rear = -1;
}

void push (int val){

    //queue full hai
    if (rear == size-1) {
      cout << "queue overflow " << endl;
      return;
    }
    //empty array me push krna hai
    else if (front == -1 && rear == -1) {
        front ++;
        rear ++;
        arr [rear] = val;
    }
    //normal case
    else {
        rear ++;
        arr [rear] = val;
    }
}
```

```
void pop(){

    // queue empty hai
    if (front == -1 && rear == -1) {
        cout << "queue underflow " << endl;
        return;
    }
    // single element hai
    else if (front == rear) {
        arr[front] = -1;
        front = -1;
        rear = -1;
    }
    // normal case
    else {
        arr[front] = -1;
        front++;
    }
}

int getSize( ){

    // jb empty hai
    else
    if (front == -1 && rear == -1) {
        return 0;
    }
    else {
        return rear - front + 1;
    }
}
```

```
int getfront () {

    //Jb empty hai
    if (front == -1) {
      cout<<"no element in queue, no front"<<endl;
return-1;}
    else{
      return arr[front];
    }
  }

int getRear () {

    //Jb empty hai
    if (rear == -1){
    cout<<"no element in queue, no rear"<< endl;
    return -1;
    }
    else{
      return arr[rear];
    }
  }

bool isEmpty () {
  if (front == -1 && rear == -1) {
      return true;
    }
  else {
      return false;
    }
} };
```

Time complexity of all functions is O(1).

Circular Queue -

Code-

```
class cqueue {
    public:
    int* arr;
    int size;
    int front;
    int rear;

cqueue (int size){
    arr = new int [size];
    this -> size = size;
    front = -1;
    rear = -1;
    }

void push (int val) {

//overflow ka case -> conditions bhul jate h log

if((front == 0 && rear == size -1) || front - rear ==1)
 {
    cout << "queue overflow" << endl;
    return;
 }

// empty ke case mei -> ye case bhul jate hai

else if (front == -1 && rear == -1) {
    front ++; rear ++;
    arr [front] = val;
}
```

```
//circular nature

else if (front != 0 && rear == size -1 ){
    rear = 0;
    arr [rear]= val;
  }

//normally

else{
    rear ++;
    arr [rear] = val;
  }
}

void pop () {

  //underflow ka case

if (front == -1 && rear == -1) {
    cout <<" queue overflow "<<endl;
  }

//single element wala case → ye case bhul jate hai

else if (front == rear){
    arr[front] == -1;
    front = -1;
    rear = -1;
  }
```

```
// circular nature

else if (front == size-1){
        arr[front] = -1;
        front = 0;
    }

// normally,

    else{
        arr[front] = -1;
        front++;
    }
  }
};
```

Limitation of normal queue -

For instance, there is a queue q -

10	20	30	40

Now, i have popped out one element. Now, the queue is -

↳ empty place

-1	20	30	40

Now, if i try to push any element, then - queue overflow aa jayega jbki queue mei ek place empty hai lekin normal queue is trh designed hai. So, normal queue mei memory

wastage hota hai. This limitation is overcom by circular queue.

Note- Jb circular nature ki baat ho, then - we can use modulus there.
kyuki agr rear % size krenge to -

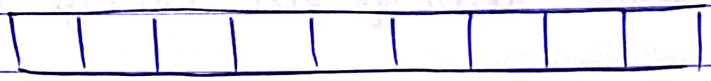
0 % 5 = 0
1 % 5 = 1
2 % 5 = 2
3 % 5 = 3
4 % 5 = 4
5 % 5 = 0
6 % 5 = 1
7 % 5 = 2

When rear reaches (size - 1)th position, then vo fir se 0th position pr chla jata hai and circular nature bnata h queue ka.

But as modulo operator is expensive i.e. it is ~~an easy~~ heavy operation. That's why avoid using it as much as you can as the solution written using it may give TLE.

* Doubly Ended Queue - (Deque)

Unlike normal queue, doubly Ended Queue mei dono side se push bhi kr skte hai and dono side se pop bhi kr skte hai.
Header file - #include <deque>



Push_front
Pop_front

Push_back
Pop_back

let a deque dq -

- dq.push_back(10);

  10

- dq.push_back(20);

  10 20

- dq.push_front(30);

  30 10 20

- dq.push_front(40);

  40 30 10 20

- dq.pop_back();

40 30 10

- dq.pop_front();

30 10

* **Circular doubly ended queue -**

```
class Deque {
    public:
    int * arr;
    int size;
    int front;
    int rear;

  Deque (int size){
      arr = new int [size];
      this -> size = size;
      front = -1;
      rear = -1;
    }
```

```
void pushBack (int val) {

  // empty wala case
  if (front == -1 && rear == -1) {
        front ++;
        rear ++;
        arr [rear] = val;
      }

  // overflow ka case
  else if((front == 0 && rear == size -1) ||
          (front - rear == 1)) {
     cout << "queue overflow" << endl;
   }

  // circular nature
  else if (rear == size -1 && front != 0) {
        rear = 0;
        arr [rear] = val;
      }

  // normal flow
  else {
      rear ++;
      arr [rear] = val;
    }
}
```

```
void pushfront (int val) {

    //empty wala case
  if (front == -1 && rear == -1) {
        front ++;
        rear ++;
        arr [front] = val;
      }

    //overflow ka case
else if((front == 0 && rear == size-1) ||
                (front-rear == 1)){
   cout << "queue overflow" << endl;
  }

    //circular nature
else if (front == 0 && rear != size-1) {
      front = size-1;
      arr [front] = val;
    }

    //normal flow
else {
      front --;
      arr [front] = val;
    }
}
```

```
void popfront(){

    //empty wala case
 if (front == -1 && rear == -1){
     cout << "queue underflow" << endl;

    // single element wala case
else if (front == rear){
        arr[front] = -1;
        front = -1;
        rear = -1;
    }

    // circular nature
else if (front == size -1){
        arr[front] = -1;
        front = 0;
    }

    // normal flow
else {
    arr[front] = -1;
    front ++;
    }
}
```

```
void popBack() {

    // empty wala case
    if (front == -1 && rear == -1) {
        cout << "queue underflow" << endl;
    }

    // single element ka case
    else if (front == rear) {
        arr[rear] = -1;
        front = -1;
        rear = -1;
    }

    // circular nature
    else if (rear == 0) {
        arr[rear] = -1;
        rear = size - 1;
    }

    // normal flow
    else {
        arr[rear] = -1;
        rear--;
    }
}
};
```