**1. Problem Discussion :-**

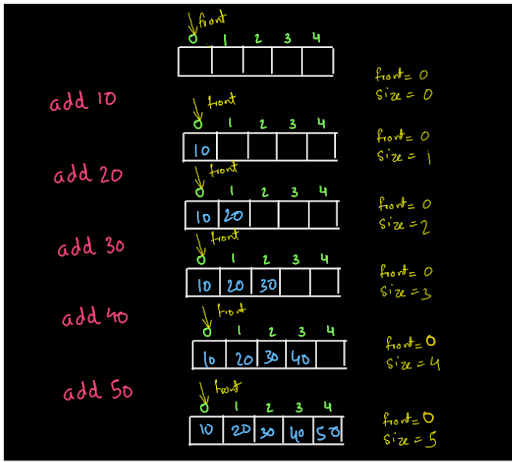
1. You are required to complete the code of our CustomQueue class. The class should mimic the behaviour of a Queue and implement FIFO semantic.

2. Here is the list of functions that you are supposed to complete 2.1. add -> Should accept new data if there is space available in the underlying array or print "Queue overflow" otherwise. 2.2. remove -> Should remove and return value according to FIFO, if available or print "Queue underflow" otherwise and return -1. 2.3. peek -> Should return value according to FIFO, if available or print "Queue underflow" otherwise and return -1. 2.4. size -> Should return the number of elements available in the queue. 2.5. display -> Should print the elements of queue in FIFO manner (space- separated) ending with a line-break.

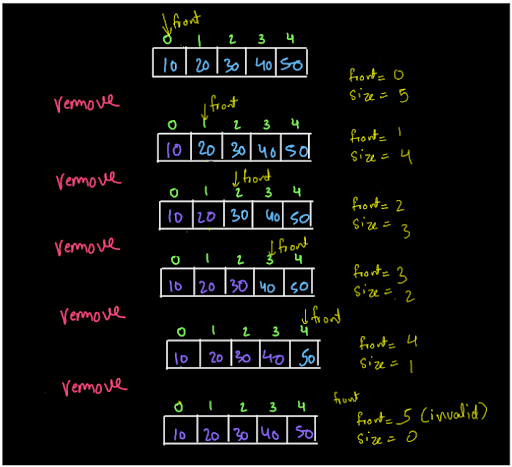
**2. Approach :-**

Building a Normal Queue: You may skip this section of the article if you already know how to build a normal queue. If you want to revise it or even study it, you are most welcome. Let us start with a very basic idea of building a linear queue.

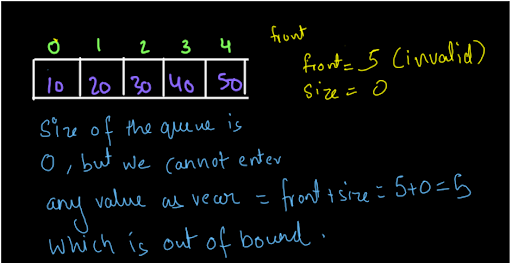
Building a Linear Queue: As we know the queue data structure works in a FIFO (first-in-first-out) fashion. So, we add the data at the end of the queue every time and we remove the data from its front. Have a look at the figure shown below:

****

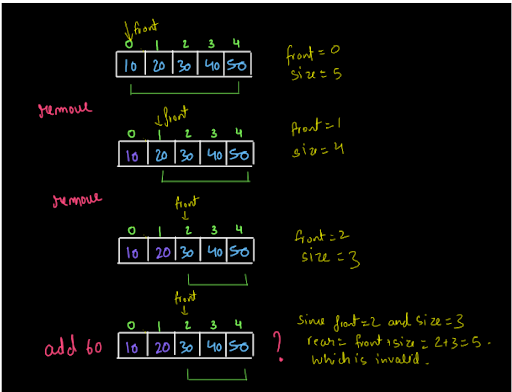
We keep on adding the data in the queue and data gets added at the last of the queue. Now, if we try to add any more elements to the queue we will not be able to add those elements as the queue is full. The condition for the queue to be full is when this array "data" becomes full. So, the condition is size=data.length. So, we are adding the values into the queue always at the rear end. Let us call the index at which value is inserted as rear. So, rear=front +size. For instance, initially front is at 0 and the size is also 0. So, rear=front+size=0+0=0, and we add the value 10 at data[rear]. Then after adding a value, the size becomes 1. So, the next value is added at rear=front+size=0+1=1. So, data[rear]=data[1]=20. We can continue inserting like this. Similarly, the removal of elements from the queue is shown below:

****

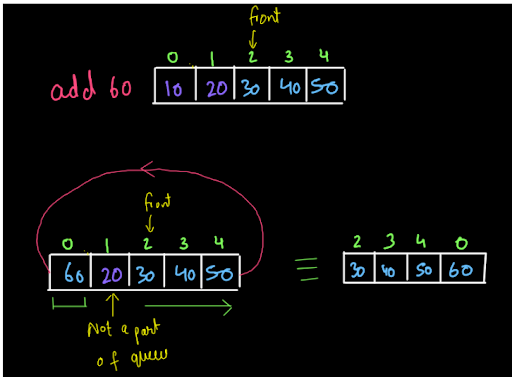
We always remove the first element from the queue. The elements shown in purple indicate that they are not a part of the queue anymore. For instance, we remove 10 from the queue, not by shifting the elements, just by moving front to the next index (and decrementing the size value by 1) indicating that the queue now starts from index 1 and any element before index=1is not a part of the queue. When size=0, there is no element left in the queue and we cannot dequeue (remove) any element from the queue. So, if we try to remove any element after this, we will print "queue underflow". Did you notice anything strange in this queue? Hint: look at (fig-3) the last array, when the queue is empty. Need for a Circular Queue: Let us take the last stage of the queue.

****

The size of the queue is 0 as we have removed all the elements from the queue. Still, we can not enter any value into the queue as the front became 5 when we removed the last value and also rear=front+size=5+0=5, which is also an invalid position to enter a value into the queue. Similarly, look at the diagram shown below:

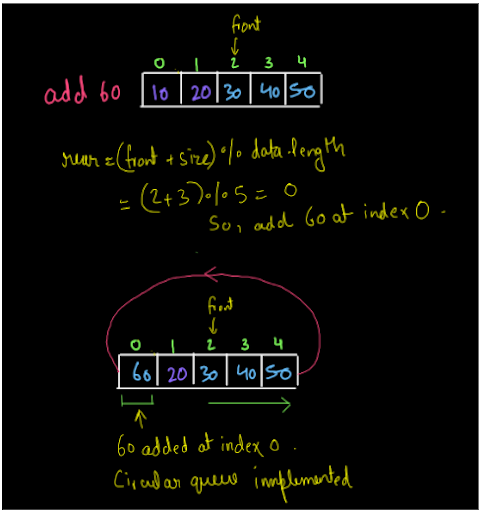
****

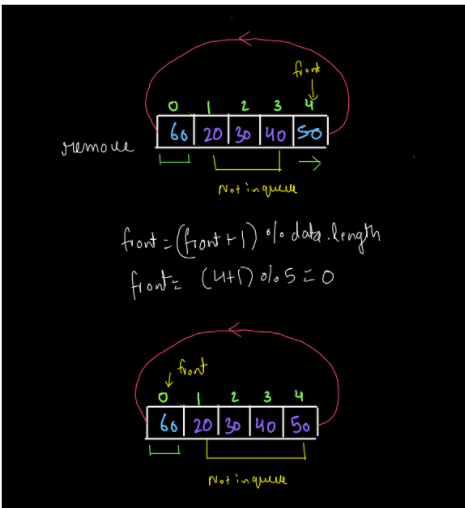
We removed the first two elements from the queue. So, only three elements inside the array are now a part of the queue. Now, if we want to insert an element into the queue, it can be inserted at the rear position only. Now, rear=front+size=2+3=5, which is out of the bound for the array. So, even if 2 spaces were vacant in that array i.e. 2 spaces were available which were not a part of the queue, still, we could not insert our elements there. So, you saw, in the first case, when we had the entire array vacant then also, we were not able to enter any values into it and we were not able to enter the values into the queue even if certain places (not the entire array) were vacant. So, what is the solution to this? Have a look at the diagram given below:

****

We want our queue to show a circular behavior like this to utilize the space available. So, this can be implemented easily, using a modulus operator.

Building Circular Queue: Enqueue (Add): While adding the values into the queue, we were calculating the rear as rear=front+size. Now, we will calculate rear as rear=(front+size)%data.length. This will help us achieve a circular implementation of the queue. (See fig-7) Dequeue (Remove): While removing the values, we will not just move front forward by doing front++. Rather, we will do: front=(front+1)%data.length. This will help achieve the circular implementation (See fig-8). Display: While displaying the queue also, we will display it from the front. So, we will again increment the front as shown above i.e. front=(front+1)%data. length

****

****

**3. CODE -:**

ConsoleCpp

#include <iostream>

#include <exception>

using namespace std;

class CustomQueue {

public:

int\* arr;

int sizeQ;

int front;

int arrSize;

CustomQueue(int cap) {

sizeQ = 0;

front = 0;

arr = new int[cap];

arrSize = cap;

}

int size() {

return sizeQ;

}

void add(int data) {

if (sizeQ == arrSize) {

cout << "Queue overflow" << endl;

return;

}

int idx = (front + sizeQ) % (arrSize);

arr[idx] = data;

sizeQ++;

}

int peek() {

if (sizeQ == 0) {

cout << "Queue underflow" << endl;

return -1;

}

else {

int val = arr[front];

return val;

}

}

int remove() {

if (sizeQ == 0) {

cout << "Queue underflow" << endl;

return -1;

}

else {

int rem = arr[front];

front = (front + 1) % arrSize;

sizeQ--;

return rem;

}

}

void display() {

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

int idx = (front + i) % arrSize;

cout << arr[idx] << " ";

}

cout << endl;

}

};

int main() {

int n;

cin >> n;

CustomQueue q(n);

string str;

cin >> str;

while (str != "quit") {

if (str == "add") {

int val;

cin >> val;

q.add(val);

}

else if (str == "remove") {

int val = q.remove();

if (val != -1) {

cout << val << endl;

}

}

else if (str == "peek") {

int val = q.peek();

if (val != -1) {

cout << val << endl;

}

}

else if (str == "size") {

cout << q.size() << endl;

}

else if (str == "display") {

q.display();

}

cin >> str;

}

return 0;

}

**4. CODE DISCUSSION -:**

We will modify the add(value) function that we have created in our previous problem. Whenever we get our queue full i.e. when size=data.length, we will create another array that will be double the size of the queue. We will copy our elements from the queue to the array in such a way that the elements of the queue are placed linearly starting from index 0 in the array as shown in fig-1. For this we will do arr[i]=data[(front+i)%data.length] for n iterations, where n is the current size of the queue

After this, we will copy the array back to the original queue and we will add the element at the end as we used to do for a normal queue.

**5. Analysis -:**

Time Complexity : O(1)

The time complexity is O(1).

Space Complexity : O(n)

This space complexity is O(n).

That was easy. Wasn't it? Our desire to make you learn will remain unsatisfactory if you still have doubts. We strongly recommend you to watch our video lecture on Infix Conversions for clearing any type of doubts. Suggestions and feedback are always welcomed. You can contact us via our website. All the best for a bright future! Happy Coding!