24) IMPLEMENT N QUEUE IN AN ARRAY

METHOD 1 (Divide the array in slots of size n/k)

A simple way to implement k queues is to divide the array in k slots of size n/k each, and fix the slots for different queues, i.e., use arr[0] to arr[n/k-1] for the first queue, and arr[n/k] to arr[2n/k-1] for queue2 where arr[] is the array to be used to implement two queues and size of array be n.

The problem with this method is an inefficient use of array space. An enqueue operation may result in overflow even if there is space available in arr[]. For example, consider k as 2 and array size n as 6. Let we enqueue 3 elements to first and do not enqueue anything to the second queue. When we enqueue the 4th element to the first queue, there will be overflow even if we have space for 3 more elements in the array.

METHOD 2 (A space efficient implementation)

The idea is similar to the stack post, here we need to use three extra arrays. In stack post, we needed two extra arrays, one more array is required because in queues, enqueue() and dequeue() operations are done at different ends.

Following are the three extra arrays are used:

1) front[]: This is of size k and stores indexes of front elements in all queues.
2) rear[]: This is of size k and stores indexes of rear elements in all queues.
2) next[]: This is of size n and stores indexes of next item for all items in array arr[].

Here arr[] is the actual array that stores k stacks.

Together with k queues, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in front[] are initialized as -1 to indicate that all queues are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to the next slot. Top of the free stack, 'free' is initialized as 0.

(HERE FOR GETTING THE POSITIONS WHICH ARE EMPTY AFTER POPPING, I HAVE USED ONE MORE ARRAY CALLED EMPTY_POSITION WHICH KEEPS THE TRACK.)

CODE OF METHOD 2:

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

class nqueue{
   private:
   int array[100];
   int empty_positions[100];
   int next_position[100];
   int *front;
   int *rear;
   int j=0;

   public:
   nqueue(int n){
      front=new int[n];
      rear=new int[n];
```

```
        for(int i=0;i<100;i++){
            empty_positions[i]=i;
            next_position[i]=-1;
        }
        for(int i=0;i<n;i++){
            front[i]=-1;
            rear[i]=-1;
        }
    }
    void push(int i,int data){
        if(front[i]==-1){
            front[i]={rear[i]=empty_positions[j]};
            array[empty_positions[j]]=data;
            j++;
        }
        else{
            array[empty_positions[j]]=data;
            next_position[rear[i]]=empty_positions[j];
            rear[i]=empty_positions[j];
            j++;
        }
    }
    void pop(int i){
        if(front[i]==rear[i]){
            j--;
            empty_positions[j]=front[i];
            next_position[front[i]]==-1;
            front[i]={rear[i]=-1};
        }
        else{
            j--;
            empty_positions[j]=front[i];
            int t=front[i];
            front[i]=next_position[t];
            next_position[t]=-1;
        }
    }
    void peek(int i){
        cout<<endl;
        int traverse=front[i];
        while(traverse!=-1){
            cout<<array[traverse]<<" ("<<traverse<<")  ";
            traverse=next_position[traverse];
        }
    }
};

int main(){
    nqueue nq(10);
    nq.push(0,10);
    nq.push(1,20);
    nq.push(2,30);
    nq.push(3,40);
    nq.push(0,50);
    nq.push(1,60);
    nq.push(2,70);
    nq.push(3,80);

    nq.pop(0);
    nq.push(3,10);

    nq.pop(2);
    nq.push(1,70);

    nq.pop(2);
    nq.push(0,70);

    nq.peek(0);
    nq.peek(1);
    nq.peek(2);
    nq.peek(3);
    return 0;
}
```