**Saransh Kalra**

**R11528524**

**Q1 header file**

#ifndef lab\_05\_hpp

#define lab\_05\_hpp

#include <stdio.h>

*//creating the node class for the node of the linked list*

**class** Node

{

**public**:

*//variable info for storing node info*

**int** info;

*//pointer of class node, pointing to the next node of the linked list*

Node\* next;

};

*//class for linked list*

**class** LinkedList

{

**public**:

*//pointers to point of the head and tail of the list*

Node \*head, \*tail;

LinkedList(); *//constructor to initialize head and tail to 0 or NULL*

~LinkedList(); *//deconstructor to free the memory borrowed from heap*

*//member functions to insert a node at the head and the tail of the linked list*

**void** insertAtHead(**int** value);

**void** insertAtTail(**int** value);

*//Print the info value before deletion of the node and then deleting the node from the head and tail*

*//of the linked list*

**void** deleteAtHead();

**void** deleteAtTail();

*//function for deleting the smallest info out of the whole linked list*

**void** deleteSmallest();

*//function to remove the duplicates of the number in the linked list*

**void** removeDuplicates(**int** number);

*//search the given info found at which node of a linked list*

**void** search(**int** value);

*//member function to display the list*

**void** displayList();

};

#endif */\* lab\_05\_hpp \*/*

**Q1 cpp file**

#include "lab\_05.hpp"

#include <stdio.h>

#include <iostream>

**using** **namespace** std;

*//Initialising the head and tail pointers to NULL*

LinkedList::LinkedList(){

**this**->head=**this**->tail=**NULL**;

}

*//deconstructor to free the memory occupied by linked list*

LinkedList::~LinkedList(){

*//while the head doesn't reach till node, traverse through the list and free the memory occupied by each node*

**while**(**this**->head!=**NULL**)

{

Node \*ptr;

ptr = **this**->head;

**this**->head = **this**->head->next;

**delete** ptr;

}

}

*//member function to insert a node at the head of the linked list*

**void** LinkedList::insertAtHead(**int** value){

*//if head is NULL, means the list is empty, put the info in the first node and make the tail as head too as single node is both the head and tail*

**if**(**this**->head==**NULL**){

*//initialise a pointer to store the new info to be added*

Node \*ptr = **new** Node();

ptr->info=value;

*//the new pointer's next node is the old head*

ptr->next=**this**->head;

*//the updated head points to the new pointer now*

**this**->head=ptr;

*//making the tail to be the head*

**this**->tail=**this**->head;

}

**else**{

*//same as above just dont make the head the tail this time as this is not the first node*

Node \*ptr = **new** Node();

ptr->info=value;

ptr->next=**this**->head;

**this**->head=ptr;

}

}

*//member function to insert a node at the tail of the linked list*

**void** LinkedList::insertAtTail(**int** value){

*//if tail is NULL, means the list is empty, put the info in the first node and make the head as tail too as single node is both the head and tail*

**if**(**this**->tail==**NULL**){

*//initialise a pointer to store the new info to be added*

Node \*ptr = **new** Node();

ptr->info=value;

*//the old tail's pointer to next node is the new pointer*

**this**->tail->next=ptr;

*//the updated tail points to the new pointer now*

**this**->tail=ptr;

*//making the head to be the tail*

**this**->head=**this**->tail;

}

**else**{

*//same as above just dont make the tail the head this time as this is not the first node*

Node \*ptr = **new** Node();

ptr->info=value;

ptr->next=**NULL**;

**this**->tail->next=ptr;

**this**->tail=ptr;

}

}

*//Print the info value before deletion of the node and then deleting the node at the head of the linked list*

**void** LinkedList::deleteAtHead(){

*//if the head points to NULL, means the list is empty, therefore error message printed*

**if**(**this**->head==**NULL**){

cout<<"there is nothing to be deleted, the list is empty"<<endl;

}

*//if the head's next points to NULL, means there is just one record so print deleted info and make both the head and tail to be NULL, making the list to be empty*

**else** **if**(**this**->head->next==**NULL**){

cout<<"The value being deleted is: "<<**this**->head->info<<endl;

Node \*temp = **this**->head;

**this**->head=**this**->tail=**NULL**;

**delete** temp;

}

*//else just make the head to be its next and print the old head info before that*

**else**{

cout<<"The value being deleted is: "<<**this**->head->info<<endl;

Node \*temp = **this**->head;

**this**->head=**this**->head->next;

**delete** temp;

}

}

*//Print the info value before deletion of the node and then deleting the node at the tail of the linked list*

**void** LinkedList::deleteAtTail(){

*//if the head points to NULL, means the list is empty, therefore error message printed*

**if**(**this**->head==**NULL**){

cout<<"there is nothing to be deleted, the list is empty"<<endl;

}

*//if the head's next points to NULL, means there is just one record so print deleted info and make both the head and tail to be NULL, making the list to be empty*

**else** **if**(**this**->head->next==**NULL**){

cout<<"The value being deleted is: "<<**this**->tail->info<<endl;

Node \*temp = **this**->tail;

**this**->head=**this**->tail=**NULL**;

**delete** temp;

}

*//else just make the tail's next to be NULL, and the old tail to be the previous record of tail*

**else**{

cout<<"The value being deleted is: "<<**this**->tail->info<<endl;

Node \*ptr = **this**->head;

*//do that by traversing the list till the pointer whose next is NULL, to find the record who's next is tail and then make it the tail of the linked list*

**while**(ptr->next!=**this**->tail){

ptr=ptr->next;

}

Node \*temp = **this**->tail;

**this**->tail=ptr;

**this**->tail->next=**NULL**;

**delete** temp;

}

}

*//function for deleting the smallest info out of the whole linked list*

**void** LinkedList::deleteSmallest(){

*//if the head points to NULL, means the list is empty, therefore error message printed*

**if**(**this**->head==**NULL**){

cout<<"there is nothing to be deleted, the list is empty"<<endl;

}

*//if head equals tail, there is only one element in the list, so delete that*

**else** **if**(**this**->head==**this**->tail) {

Node \*temp = **this**->head;

cout<<"The smallest value being deleted is: "<<temp->info<<endl;

**this**->head = **this**->tail = **NULL**;

**delete** temp;

}

*//else just find the minimum and delete that*

**else** {

*//a pointer to traverse the list starting from the next of the head*

Node \*ptr = **this**->head->next;

*//a pointer to keep record of the previous pointer starting at the head*

Node \*prev = **this**->head;

*//initial minimum set to the head, giving the reason being the transversal pointer is initialised to next of head*

Node \*minimum = **this**->head;

*//initial previous of minimum set to be NULL*

Node \*minimum\_prev = **NULL**;

*//traversing the list till the pointer comes to tail to find minimum*

**while**(ptr!=**NULL**){

*//if the traversing pointer has info less than minimum set minimum to that record and the previous to the minimum to the new previous pointer*

**if**(ptr->info < minimum->info) {

minimum = ptr;

minimum\_prev = prev;

}

*//move traversal pointer and previous pointer to the next*

prev = prev->next;

ptr = ptr->next;

}

*//mode pointer made to store the node that has to deleted*

Node \*temp = minimum;

*//if minimum is head then just make the head point to the next of head*

**if**(minimum==**this**->head){

**this**->head=**this**->head->next;

}

*//if minimum is the tail then make the new tail to be previous of the minimum and the next to be NULL*

**else** **if**(minimum==**this**->tail) {

**this**->tail=minimum\_prev;

minimum\_prev->next = **NULL**;

}

*//else just set the next of the previous of minimum to be the next of the minimum*

**else** {

minimum\_prev->next = minimum->next;

}

*//print what's being deleted and delete the record*

cout<<"The smallest value being deleted is: "<<temp->info<<endl;

**delete** temp;

}

}

*//function to remove the duplicates of the number in the linked list*

**void** LinkedList::removeDuplicates(**int** number){

*//flag variable to indicate whether value was found or not*

**int** flag=0;

*//if head points to NULL, means the list is empty*

**if**(**this**->head==**NULL**)

cout<<"The whole list is empty, so the record was not found"<<endl;

*//else traverse the whole linked list to find if some node's info matches with the given info*

**else**{

*//pointers to point to previous and the pointer for traversal*

Node \*ptr = **this**->head;

Node \*prev = **this**->head;

*//loop to traverse through the list*

**while**(ptr!=**NULL**){

*//if we find the number in the linked list in the traversal*

**if**(ptr->info==number){

*//store the pointer to be deleted and previous pointer to it*

Node\* del = ptr;

Node\* del\_prev = prev;

*//if the head is being deleted, update the head*

**if**(del==**this**->head) {

**this**->head=**this**->head->next;

*//keep the previous at the new head*

prev = **this**->head;

}

*//else if the tail is being deleted, update the tail*

**else** **if**(del==**this**->tail) {

**this**->tail = del\_prev;

**this**->tail->next = **NULL**;

}

*//else just connect the previous's next to the delted's next*

**else** {

del\_prev->next = del->next;

}

*//update the flag to 1 to indicate that the record was found*

flag=1;

*//update the previous pointer only when the head is not the number to be deleted*

**if**(del!=**this**->head) {

ptr=ptr->next;

}

*//print the deleted node's info and delete that node*

cout<<"The value being deleted is: "<<del->info<<endl;

**delete** del;

}

*//else just update the prev and traversal pointer*

**else** {

**if**(ptr!=**this**->head) {

prev=prev->next;

}

ptr=ptr->next;

}

}

*//error message if info was not found*

**if**(flag==0){

cout<<"The given value "<<number<<" was not found in the list!"<<endl;

}

}

}

*//member function to search the given info found at which node of a linked list*

**void** LinkedList::search(**int** value){

*//counter variable to count at which position given info was found*

**int** count=0;

*//flag variable to indicate whether value was found or not*

**int** flag=0;

*//if head points to NULL, means the list is empty*

**if**(**this**->head==**NULL**)

cout<<"The whole list is empty, so the record was not found"<<endl;

*//else traverse the whole linked list to find if some node's info matches with the given info*

**else**{

Node \*ptr = **this**->head;

**while**(ptr->info!=value && ptr!=**this**->tail){

ptr=ptr->next;

count++;

**if**(ptr->info==value){

flag=1;

}

}

*//error message if info was not found*

**if**(flag==0){

cout<<"The given value "<<value<<" was not found in the list!"<<endl;

}

**else**{

cout<<"The value "<<value<<" was found and its the no. "<<count+1<<" record."<<endl;

}

}

}

*//member function to display the linked list*

**void** LinkedList::displayList(){

*//if head points to NULL, the list is empty*

**if**(**this**->head==**NULL**){

cout<<"The whole list is empty, nothing to display."<<endl;

}

*//else traverse through the list displaying each node's info followed by an arrow*

**else**{

Node \*ptr;

ptr = **this**->head;

**while**(ptr!=**NULL**){

cout<<ptr->info<<"->";

ptr = ptr->next;

}

cout<<endl;

}

}

**Q1 main file**

#include <iostream>

#include "lab\_05.hpp"

**using** **namespace** std;

**int** main() {

*//initialising instance variable of linked list*

LinkedList LinkedList1;

*//call to member function to delete at tail*

LinkedList1.deleteAtTail();

*//call to member function to delete at head*

LinkedList1.deleteAtHead();

*//call to insert 1 at head*

LinkedList1.insertAtHead(1);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 2 at tail*

LinkedList1.insertAtTail(2);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 3 at tail*

LinkedList1.insertAtTail(3);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 4 at tail*

LinkedList1.insertAtTail(4);

*//display the list function called*

LinkedList1.displayList();

*//delete the smallest in the list*

LinkedList1.deleteSmallest();

*//delete the smallest in the list*

LinkedList1.deleteSmallest();

*//delete the smallest in the list*

LinkedList1.deleteSmallest();

*//display the list function called*

LinkedList1.displayList();

*//call to insert 4 at head*

LinkedList1.insertAtHead(4);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 2 at tail*

LinkedList1.insertAtTail(2);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 3 at tail*

LinkedList1.insertAtTail(3);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 4 at tail*

LinkedList1.insertAtTail(4);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 4 at tail*

LinkedList1.insertAtTail(4);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 4 at tail*

LinkedList1.insertAtTail(4);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 4 at tail*

LinkedList1.insertAtTail(4);

*//display the list function called*

LinkedList1.displayList();

*//remove the duplicates function called*

LinkedList1.removeDuplicates(4);

*//display the list function called*

LinkedList1.displayList();

**return** 0;

}

**Q1 output**

**there is nothing to be deleted, the list is empty**

**there is nothing to be deleted, the list is empty**

**1->**

**1->2->**

**1->2->3->**

**1->2->3->4->**

**The smallest value being deleted is: 1**

**The smallest value being deleted is: 2**

**The smallest value being deleted is: 3**

**4->**

**4->4->**

**4->4->2->**

**4->4->2->3->**

**4->4->2->3->4->**

**4->4->2->3->4->4->**

**4->4->2->3->4->4->4->**

**4->4->2->3->4->4->4->4->**

**The value being deleted is: 4**

**The value being deleted is: 4**

**The value being deleted is: 4**

**The value being deleted is: 4**

**The value being deleted is: 4**

**The value being deleted is: 4**

**2->3->**

**Program ended with exit code: 0**

**Q2 header file**

#ifndef lab5\_part\_2\_hpp

#define lab5\_part\_2\_hpp

#include <stdio.h>

*//creating the node class for the node of the stack*

**class** Node{

**private**:

*//variable info for storing node info*

**int** info;

*//pointer of class node, pointing to the next node of the stack*

Node \*next;

**public**:

*//constructor and deconstructor for node*

Node(**int**);

~Node();

*//getters and setters for info*

**int** get\_info();

**void** set\_info(**int**);

*//getters and setters for next*

Node\* get\_next();

**void** set\_next(Node\*);

};

**class** Queue{

*//pointer to point of the head and tail of the queue*

Node \*head, \*tail;

**public**:

*//constructor and deconstructor for the queue*

Queue();

~Queue();

**int** size;

*//function to add on head of the queue*

**void** queue\_enqueue(**int**);

*//function to remove from the tail of the queue*

**int** queue\_dequeue();

*//function to peek onto the head of the queue*

**int** queue\_peek();

*//function to print the queue*

**void** queue\_print();

*//function to move the nth record to the front*

**void** moveNthFront(**int** n);

*//function to check if the queue is empty*

**void** queue\_isEmpty();

};

#endif */\* lab5\_part\_2\_hpp \*/*

**Q2 Cpp file**

#include "lab5\_part\_2.hpp"

#include <iostream>

**using** **namespace** std;

*//constructor for node class*

Node::Node(**int** info){

*//set info to the info of the node*

**this**->info = info;*//next of the node is set to null*

**this**->next = 0;

}

*//no need of deconstructor as queue deconstructor handles these*

Node::~Node(){

}

*//getter for the info of the class node*

**int** Node::get\_info(){

**return** **this**->info;

}

*//setter for the info of the class node*

**void** Node::set\_info(**int** info){

**this**->info = info;

}

*//getter for the next of the node class*

Node\* Node::get\_next(){

**return** **this**->next;

}

*//setter for the next of the node class*

**void** Node::set\_next(Node \*next){

**this**->next = next;

}

*//constructor for the queue class*

Queue::Queue(){

**this**->head=0;

**this**->tail=0;

**this**->size=0;

}

*//deconstructor for the queue class*

Queue::~Queue(){

Node \*temp = **this**->head;

**while**(temp!=**NULL**){

temp = temp->get\_next();

**delete** temp;

}

}

*//enqueue function for the queue class*

**void** Queue::queue\_enqueue(**int** info){

*//new node to be added with info as the argument*

Node \*newNode = **new** Node(info);

*//we are pushing it on the head, so make the next of new node to be head*

newNode->set\_next(**this**->head);

*//if head is null and new node is the first node then the new node is also the tail*

**if**(!**this**->head) {

**this**->tail = newNode;

}

*//the new head is the new node*

**this**->head = newNode;

**this**->size++;

}

**int** Queue::queue\_dequeue(){

*//if tail is null, queue is empty give error*

**if**(!**this**->tail) {

cout<<"Queue is empty! no. of items avilable to dequeue: "<<endl;

**return** 0;

}

**else**{

**this**->size--;

*//if the next of head is null, there is only one element which will be deleted*

**if**(**this**->head->get\_next()==**NULL**){

*//temp for storing the value that is being dequeued*

**int** temp = **this**->tail->get\_info();

*//deleting the node at tail*

**delete** **this**->tail;

cout<<"The value being popped is: "<<endl;

*//as this was the last element make the head and tail point to null making teh list empty*

**this**->head=**this**->tail=**NULL**;

**return** temp;

}

**else**{

cout<<"The value being popped is: "<<endl;

*//else make a tranversal pointer with starts as head*

Node \*ptr = **this**->head;

*//traverse till it doesn't reach the node previous to tail*

**while**(ptr->get\_next()!=**this**->tail){

ptr=ptr->get\_next();

}

*//temp for storing the value that is being dequeued*

**int** temp = **this**->tail->get\_info();

*//delete the node at tail*

**delete** **this**->tail;

*//setting the traversed pointer as teh new tail*

**this**->tail=ptr;

*//setting its next to be null as its the end of teh queue*

**this**->tail->set\_next(**NULL**);

*//returning the removed value*

**return** temp;

}

}

}

*//peek function for queue*

**int** Queue::queue\_peek(){

*//if tail is not null, return the tail as the last value on the queue*

**if**(**this**->tail){

cout<<"Last Item in queue: ";

**return** **this**->head->get\_info();

}

*//else error message that stack is empty*

**else** {

cout<<"queue is empty! Items in Queue: ";

**return** 0;

}

}

*//function to move the nth record to the front*

**void** Queue::moveNthFront(**int** n){

*//if index is out of bounds check*

**if**(n<=0 || n>**this**->size) {

*//empty queue check*

**if**(!**this**->tail){

cout<<"Queue is empty nothing to move! "<<endl;

}

**else** {

cout<<"index out of bounds!"<<endl;

}

}

**else** {

*//counter for keeping record number*

**int** count = 0;

*//pointer from traversal initialised to next of head and prev initialised from head*

Node \*ptr = **this**->head->get\_next();

Node \*prev = **this**->head;

*//if n is 1 means move the item at first of queue, so no need just give an error saying the value is already at front*

**if**(n==1) {

cout<<"the value is already at the front"<<endl;

}

*//else if its the last on the list move the head to its next and set the tail i.e. front of the queue to the old head*

**else** **if**(n==**this**->size) {

Node\* temp = **this**->head;

**this**->head = **this**->head->get\_next();

**this**->tail->set\_next(temp);

**this**->tail = temp;

**this**->tail->set\_next(**NULL**);

}

*//else just start from head traverse till size-n as it takes entries from head and takes off from head so the entry number the user wants is from counted from the tail*

**else** {

**while**(ptr) {

count++;

**if**(count == (**this**->size-n)) {

prev->set\_next(ptr->get\_next());

**this**->tail->set\_next(ptr);

**this**->tail = ptr;

**this**->tail->set\_next(**NULL**);

}

ptr = ptr->get\_next();

prev = prev->get\_next();

}

}

}

}

*//function for printing the queue*

**void** Queue::queue\_print(){

*//if head is NULL, error queue is empty*

**if**(!**this**->head){

cout<<"queue is empty!"<<endl;

}

**else**{

*//create a temp node for traversing starting for head*

Node \*temp = **this**->head;

cout<<"{ ";

*//while temp is not null*

**while**(temp){

cout<<temp->get\_info()<<"->";

temp = temp->get\_next();

}

cout<<" }"<<endl;

cout<<"size: "<<**this**->size<<endl;

}

}

*//function to check if the queue is empty*

**void** Queue::queue\_isEmpty(){

*//if head is not null its not empty*

**if**(**this**->head){

cout<<"no! queue isn't empty. "<<endl;

}

**else**{

cout<<"yes! the queue is empty. "<<endl;

}

}

**Q2 main file**

#include <iostream>

#include "lab5\_part\_2.hpp"

**using** **namespace** std;

**int** main() {

*//queue for testing*

Queue q1;

*//initialising a choice variable*

**char** choice = '1';

*//while the choice is not the exit choice display the menu again and again*

**while**(choice!='7'){

cout<<"QUEUE MENU"<<endl;

cout<<"----------"<<endl;

cout<<"1. Push a number on end of queue"<<endl;

cout<<"2. Pop a number from the start of the queue"<<endl;

cout<<"3. Check if the queue is empty"<<endl;

cout<<"4. Take a peek in the queue"<<endl;

cout<<"5. Print the queue"<<endl;

cout<<"6. Move nth value to the front"<<endl;

cout<<"7. Exit"<<endl;

cout<<"Enter your choice: "<<endl;

cin>>choice;

*//switch case for menu*

**switch** (choice) {

*//menu option 1 for enqueue into queue*

**case** '1':

**int** info;

*//user input for the number they want to put in*

cout<<"enter the number you want to push: "<<endl;

cin>>info;

q1.queue\_enqueue(info);

**break**;

*//menu option 2 for dequeue from the queue*

**case** '2':

cout<<q1.queue\_dequeue()<<endl;

**break**;

*//menu option 3 to check if the queue is empty*

**case** '3':

q1.queue\_isEmpty();

**break**;

*//menu option 4 to peek at the tail of the queue*

**case** '4':

cout<<q1.queue\_peek()<<endl;

**break**;

*//menu option 5 to print the queue*

**case** '5':

q1.queue\_print();

**break**;

*//menu option 6 to move nth to the front*

**case** '6':

**int** info1;

cout<<"enter the record no. you want to move to the front: "<<endl;

cin>>info1;

q1.moveNthFront(info1);

**break**;

*//menu option 7 to exit*

**case** '7':

**break**;

*//default being an invalid choice*

**default**:

cout<<"invalid choice"<<endl;

**break**;

}

}

**return** 0;

}

**Q2 output**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

**Program ended with exit code: 0**6

**enter the record no. you want to move to the front:**

1

**Queue is empty nothing to move!**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

1

**enter the number you want to push:**

1

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

1

**enter the number you want to push:**

2

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

1

**enter the number you want to push:**

3

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

5

**{ 3->2->1-> }**

**size: 3**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

6

**enter the record no. you want to move to the front:**

3

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

5

**{ 2->1->3-> }**

**size: 3**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

6

**enter the record no. you want to move to the front:**

1

**the value is already at the front**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

6

**enter the record no. you want to move to the front:**

2

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

5

**{ 2->3->1-> }**

**size: 3**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

2

**The value being popped is:**

**1**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

2

**The value being popped is:**

**3**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

6

**enter the record no. you want to move to the front:**

1

**the value is already at the front**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

6

**enter the record no. you want to move to the front:**

2

**index out of bounds!**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

2

**The value being popped is:**

**2**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

6

**enter the record no. you want to move to the front:**

1

**Queue is empty nothing to move!**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

5

**queue is empty!**

**QUEUE MENU**

**----------**

**1. Push a number on end of queue**

**2. Pop a number from the start of the queue**

**3. Check if the queue is empty**

**4. Take a peek in the queue**

**5. Print the queue**

**6. Move nth value to the front**

**7. Exit**

**Enter your choice:**

7