EXPERIMENT-6

**Aim:**

Develop programs to implement Stack applications such as balancing of parenthesis & Infix to postfix notation of expression.

**Description:**

**1) Algorithm for balancing of parenthesis**:

Step 1: Declare a stack (say stack).

Step 2: Now traverse the string input.

* If the current character is a starting bracket ( i.e. ( or { or [ ) then push it to stack.
* If the current character is a closing bracket ( i.e. ) or } or ]) then pop from stack.
  + If the popped character is the matching starting bracket then fine
  + else parenthesis are not balanced.
* If the current character is a closing bracket ( i.e. ) or } or ]) and the stack is empty, then parenthesis are not balanced.

Step 3: After complete traversal, if there is some starting bracket left in stack then the

string is not balanced else we have a balanced string.

**Example:**

• Input: ((()))()()

Output: Balanced

Explanation: All the brackets are well-formed.

• Input: ())((())

Output: Not Balanced

Explanation: The first closing bracket doesn’t have a corresponding opening

bracket

**Program:**

/\*A program to check the balance of the parenthesis\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

struct stack {

char stck[MAX];

int top;

}s;

void push(char item) {

if (s.top == (MAX - 1))

printf("Stack is Full\n");

else {

s.top = s.top + 1;

s.stck[s.top] = item;

}

}

void pop() {

if (s.top == -1)

printf("Stack is Empty\n");

else

s.top = s.top - 1;

}

int checkPair(char val1,char val2){

return (( val1=='(' && val2==')' )||( val1=='[' && val2==']' )||( val1=='{' && val2=='}' ));

}

int checkBalanced(char expr[], int len){

for (int i = 0; i < len; i++)

{

if (expr[i] == '(' || expr[i] == '[' || expr[i] == '{')

{

push(expr[i]);

}

else

{

if (s.top == -1)

return 0;

else if(checkPair(s.stck[s.top],expr[i]))

{

pop();

continue;

}

return 0;

}

}

// If there are still elements in stack then the expression is not balanced

if(s.top != -1)

return 0;

return 1;

}

int main() {

char exp[MAX];

printf("Enter the expression: ");

scanf("%s",exp);

int i = 0;

s.top = -1;

int len = strlen(exp);

checkBalanced(exp, len)?printf("Balanced"): printf("Not Balanced");

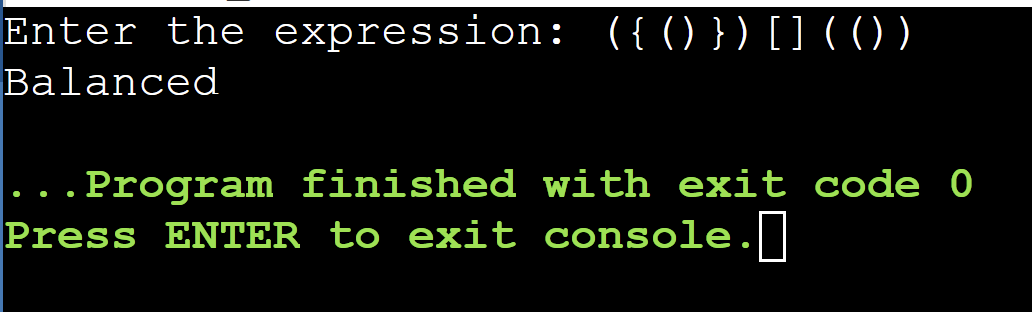
return 0;

}

**Output:**

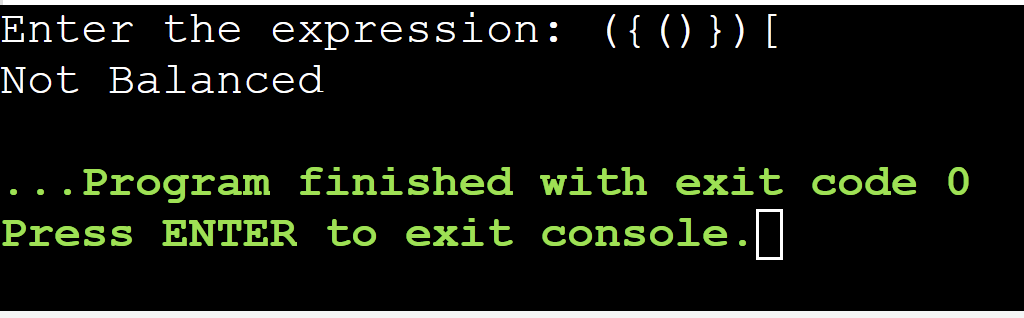
Enter the expression: ({()})[](())

Balanced



Enter the expression: ({()})[

Not Balanced



**2) Algorithm for Infix to postfix conversion**:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.).

**Example:**

Let’s take an example of converting infix expression to postfix:

Infix Expression: A+ (B\*C+D)/E

The corresponding expression in postfix form is: ABC\*D+E/+

**Program:**

/\*A program that converts infix expression to postfix\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

char stack[MAX];

char infix[MAX], postfix[MAX];

int top = -1;

void push(char);

char pop();

int isEmpty();

void inToPost();

void print();

int precedence(char);

int main()

{

printf("Enter the infix expression: ");

gets(infix);

inToPost();

print();

return 0;

}

void inToPost()

{

int i, j = 0;

char symbol, next;

for (i = 0; i < strlen(infix); i++)

{

symbol = infix[i];

switch (symbol)

{

case '(':

push(symbol);

break;

case ')':

while ((next = pop()) != '(')

postfix[j++] = next;

break;

case '+':

case '-':

case '\*':

case '/':

case '^':

while (!isEmpty() && precedence(stack[top]) >= precedence(symbol))

postfix[j++] = pop();

push(symbol);

break;

default:

postfix[j++] = symbol;

}

}

while (!isEmpty())

postfix[j++] = pop();

postfix[j] = '\0';

}

int precedence(char symbol)

{

switch (symbol)

{

// Higher value means higher precedence

case '^':

return 3;

case '/':

case '\*':

return 2;

case '+':

case '-':

return 1;

default:

return 0;

}

}

void print()

{

int i = 0;

printf("The equivalent postfix expression is: ");

while (postfix[i])

{

printf("%c", postfix[i++]);

}

printf("\n");

}

void push(char c)

{

if (top == MAX - 1)

{

printf("Stack Overflow\n");

return;

}

top++;

stack[top] = c;

}

char pop()

{

char c;

if (top == -1)

{

printf("Stack Underflow\n");

exit(1);

}

c = stack[top];

top = top - 1;

return c;

}

int isEmpty()

{

if (top == -1)

return 1;

else

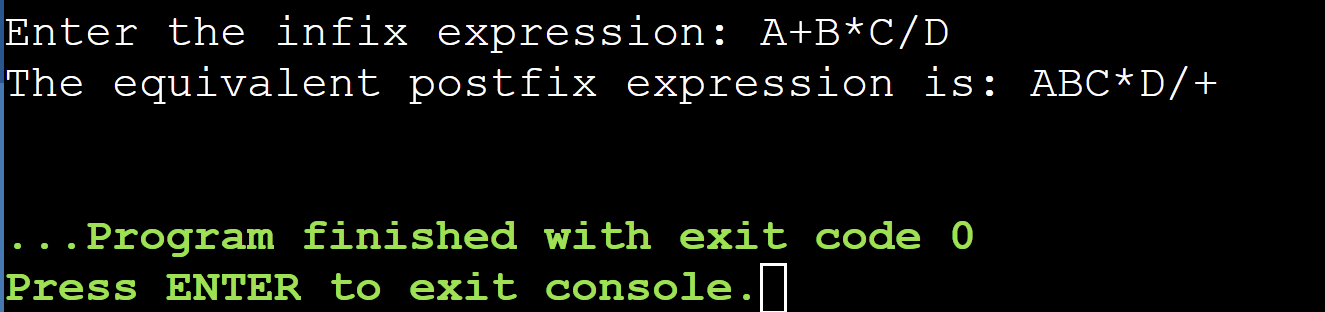
return 0;

}

**Output:**

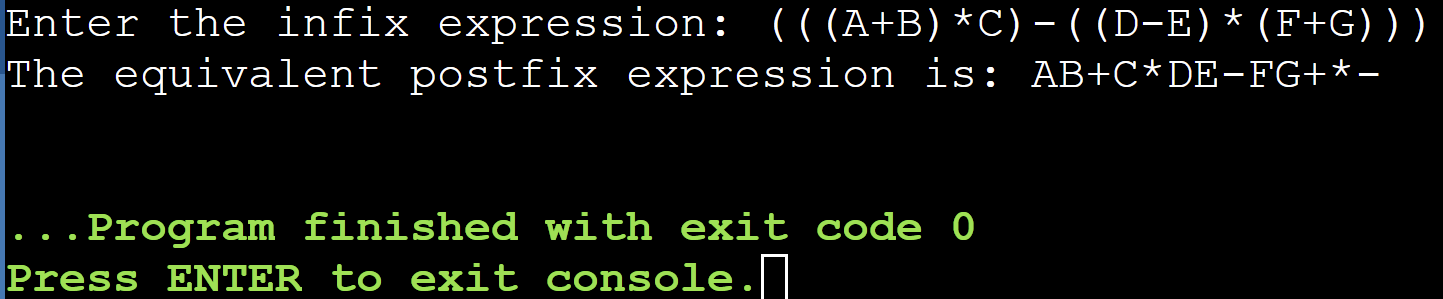
Enter the infix expression: A+B\*C/D

The equivalent postfix expression is: ABC\*D/+

****

Enter the infix expression: (((A+B)\*C)-((D-E)\*(F+G)))

The equivalent postfix expression is: AB+C\*DE-FG+\*-



**3) Algorithm for postfix evaluation**:

* Create a stack that holds integer type data to store the operands of the given postfix expression. Let it be st.
* Iterate over the string from left to right and do the following -
  + If the current element is an operand, push it into the stack.
  + Otherwise, if the current element is an operator do the following -
    - Pop an element from st, let it be op1.
    - Pop another element from st, let it be op2.
    - Computer the result of op2 OPERATOR op1, and push it into the stack. Note the order *i*.*e*. op2 OPERATOR op1 should not be changed otherwise it will affect the final result in some cases.

**Example:**

Consider the following postfix expression:

4 1 2 \* + 3 -

The corresponding infix expression is 4 + (1 \* 2) - 3.

The evaluation will be 4 + 2 - 3 which results in 3.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <math.h>

#define SIZE 100

char stack[SIZE];

int top = -1;

void push(char item) {

if(top >= SIZE-1) {

printf("\nStack Overflow.");

} else {

stack[++top] = item;

}

}

char pop() {

char item;

if(top < 0) {

printf("stack under flow: invalid infix expression");

getchar();

exit(1);

} else {

item = stack[top--];

return(item);

}

}

void push\_int(int\* stack, int\* top, int item) {

if(\*top >= SIZE-1) {

printf("\nStack Overflow.");

} else {

stack[++(\*top)] = item;

}

}

int pop\_int(int\* stack, int\* top) {

int item;

if(\*top < 0) {

printf("stack under flow: invalid postfix expression");

getchar();

exit(1);

} else {

item = stack[\*top];

(\*top)--;

return(item);

}

}

int is\_operator(char symbol) {

if(symbol == '^' || symbol == '\*' || symbol == '/' || symbol == '+' || symbol =='-') {

return 1;

} else {

return 0;

}

}

int precedence(char symbol) {

if(symbol == '^') {

return(3);

} else if(symbol == '\*' || symbol == '/') {

return(2);

} else if(symbol == '+' || symbol == '-') {

return(1);

} else {

return(0);

}

}

void InfixToPostfix(char infix\_exp[], char postfix\_exp[]) {

int i, j;

char item;

char x;

push('(');

strcat(infix\_exp, ")");

i = 0;

j = 0;

item = infix\_exp[i];

while(item != '\0') {

if(item == '(') {

push(item);

} else if(isdigit(item) || isalpha(item)) {

postfix\_exp[j] = item;

j++;

} else if(is\_operator(item) == 1) {

x = pop();

while(is\_operator(x) == 1 && precedence(x) >= precedence(item)) {

postfix\_exp[j] = x;

j++;

x = pop();

}

push(x);

push(item);

} else if(item == ')') {

x = pop();

while(x != '(') {

postfix\_exp[j] = x;

j++;

x = pop();

}

} else {

printf("\nInvalid infix Expression.\n");

getchar();

exit(1);

}

i++;

item = infix\_exp[i];

}

if(top > 0) {

printf("\nInvalid infix Expression.\n");

getchar();

exit(1);

}

postfix\_exp[j] = '\0';

}

int evaluate\_postfix(char\* postfix\_expression) {

int stack[SIZE];

int top = -1;

char ch;

int i = 0;

int val;

int A, B;

while((ch = postfix\_expression[i++]) != '\0') {

if(isdigit(ch)) {

push\_int(stack, &top, ch - '0');

} else {

B = pop\_int(stack, &top);

A = pop\_int(stack, &top);

switch(ch) {

case '+': push\_int(stack, &top, A + B); break;

case '-': push\_int(stack, &top, A - B); break;

case '\*': push\_int(stack, &top, A \* B); break;

case '/': push\_int(stack, &top, A / B); break;

case '^': push\_int(stack, &top, pow(A, B)); break;

}

}

}

return stack[0];

}

int main() {

char infix\_expression[SIZE];

char postfix\_expression[SIZE];

printf("Enter an infix expression: ");

scanf("%s", infix\_expression);

InfixToPostfix(infix\_expression, postfix\_expression);

printf("Postfix expression: %s\n", postfix\_expression);

int result = evaluate\_postfix(postfix\_expression);

printf("Result: %d\n", result);

return 0;

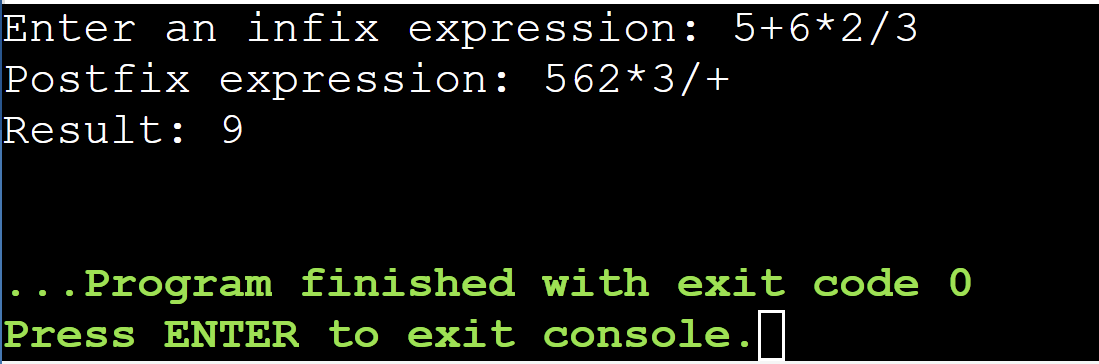
}

**Output:**

Enter an infix expression: 5+6\*2/3

Postfix expression: 562\*3/+

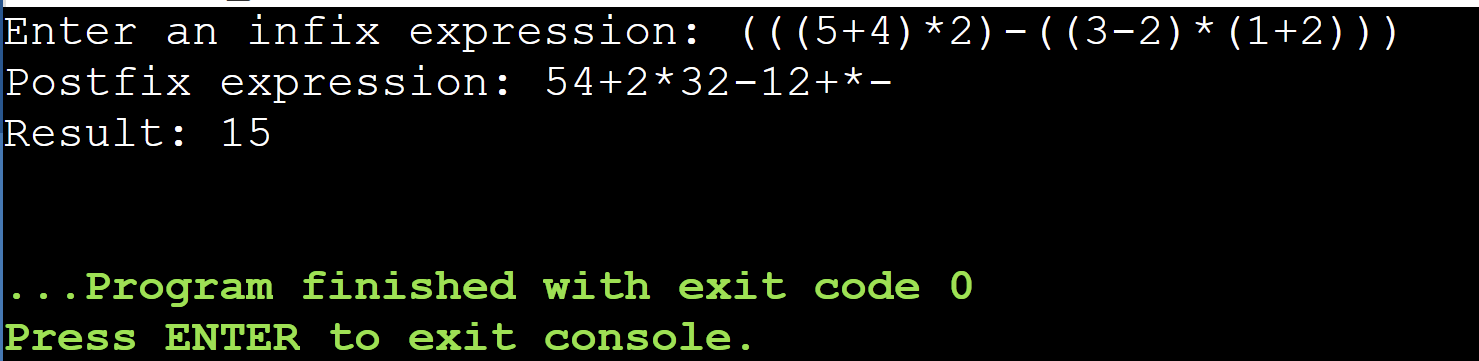
Result: 9

****

Enter an infix expression: (((5+4)\*2)-((3-2)\*(1+2)))

Postfix expression: 54+2\*32-12+\*-

Result: 15

****

**EXPERIMENT-7**

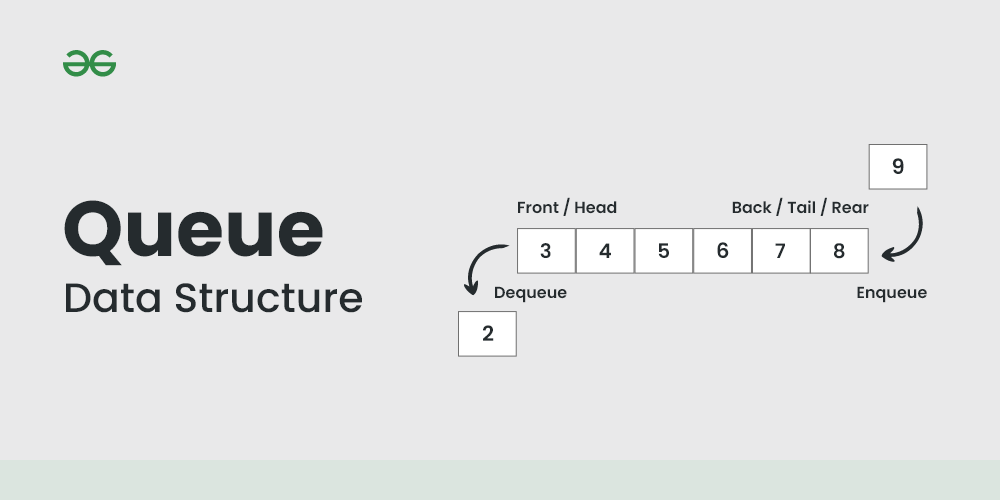
**Aim:**

Develop a program to perform operations of a Queue using the implementation in Arrays and Linked Lists

**Description:**

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

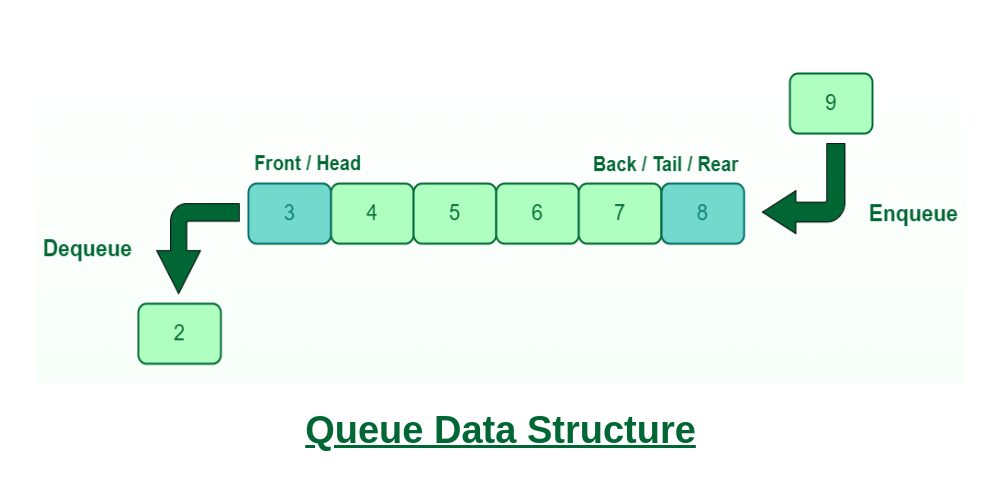
We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.



**FIFO Principle of Queue:**

A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).

Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue(sometimes, head of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue. See the below figure.



**Characteristics of Queue:**

Queue can handle multiple data.

We can access both ends.

They are fast and flexible.

**Advantages of Queue:**

* A large amount of data can be managed efficiently with ease.
* Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
* Queues are useful when a particular service is used by multiple consumers.
* Queues are fast in speed for data inter-process communication.
* Queues can be used in the implementation of other data structures.

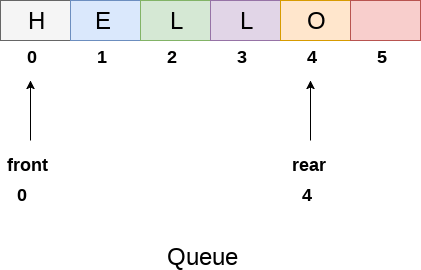
**Disadvantages of Queue:**

* The operations such as insertion and deletion of elements from the middle are time consuming.
* Limited Space.
* In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
* Searching an element takes O(N) time.
* Maximum size of a queue must be defined prior.

**Applications of Queue:**

* Multi programming: Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
* Network: In a network, a queue is used in devices such as a router or a switch. another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
* Job Scheduling: The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
* Shared resources: Queues are used as waiting lists for a single shared resource.

**Operations on queues using arrays:**



**INSERT (Enqueue):**

Algorithm: Check if the queue is full. If not, add an element at the rear.

if IsFull:

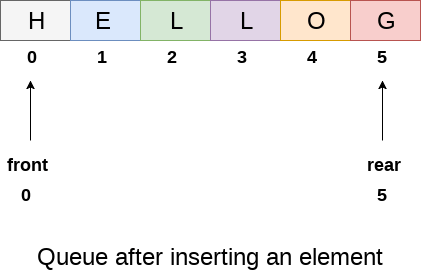
print "Queue Overflow"

else:

rear = (rear + 1) % capacity

queue[rear] = element

Time Complexity: O(1) - constant time for inserting an element.



**DELETE (Dequeue):**

Algorithm: Check if the queue is empty. If not, remove the element from the front.

if IsEmpty:

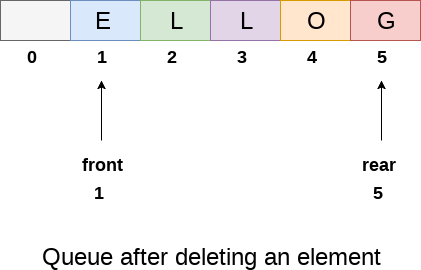
print "Queue Underflow"

else:

element = queue[front]

front = front + 1

Time Complexity: O(1) - constant time for deleting an element.

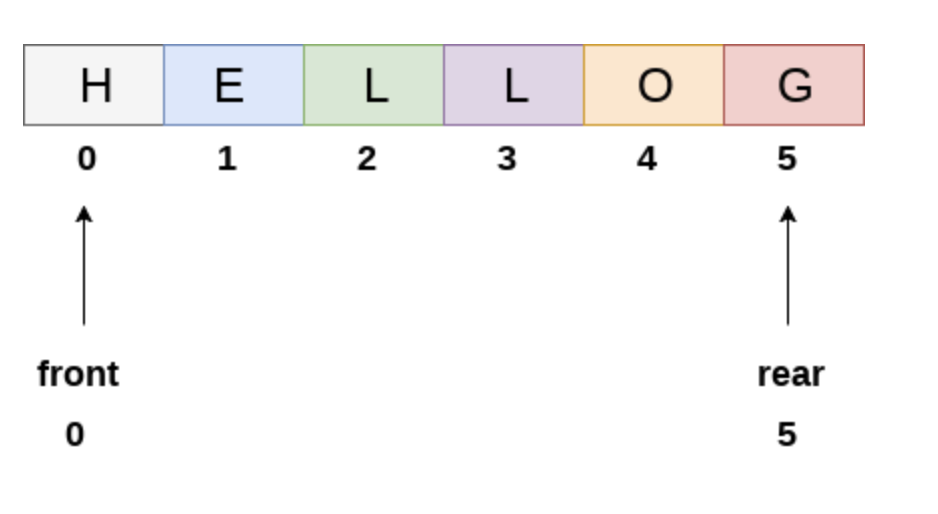


**SIZE:**

Algorithm: Calculate the size of the queue (number of elements).

size = rear - front + 1

Time Complexity: O(1) - constant time for calculating the size.

**Size=6**

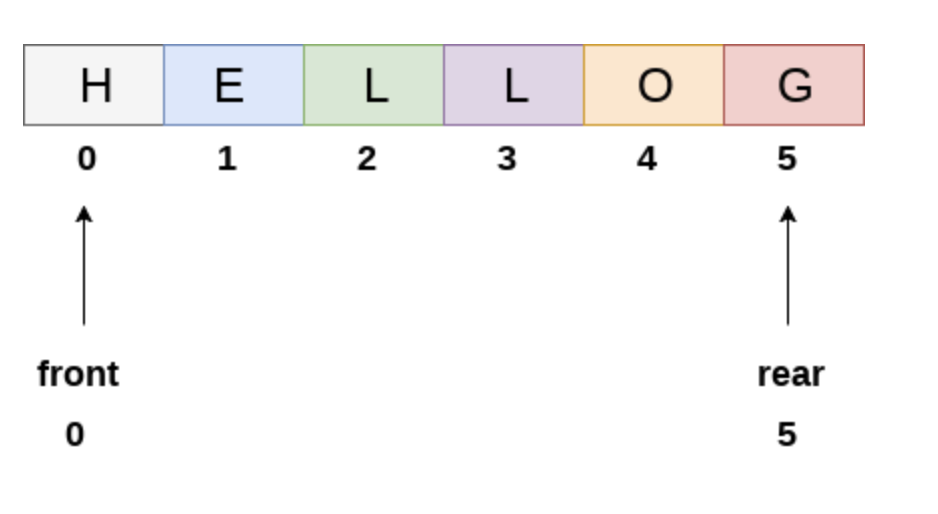
**Display:**

Algorithm: Print all elements in the queue.

for i from front to rear:

print queue[i]

Time Complexity: O(n) - linear time where n is the number of elements in the queue.



**Output:H E L L O G**

**FRONT:**

Algorithm: Return the front element without removing it.

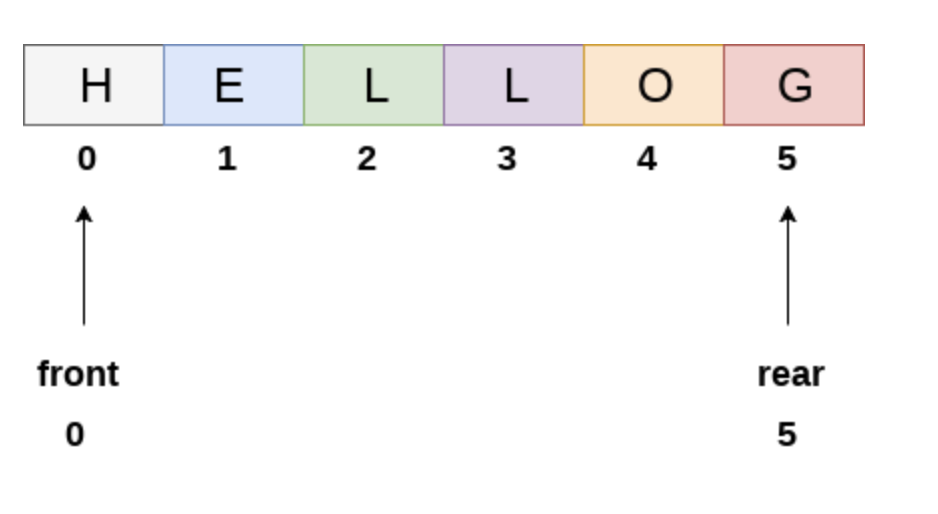
if IsEmpty:

print "Queue is Empty"

else:

print queue[front]

Time Complexity: O(1) - constant time to retrieve the front element.



**Front=H**

**REAR:**

Algorithm: Return the rear element without removing it.

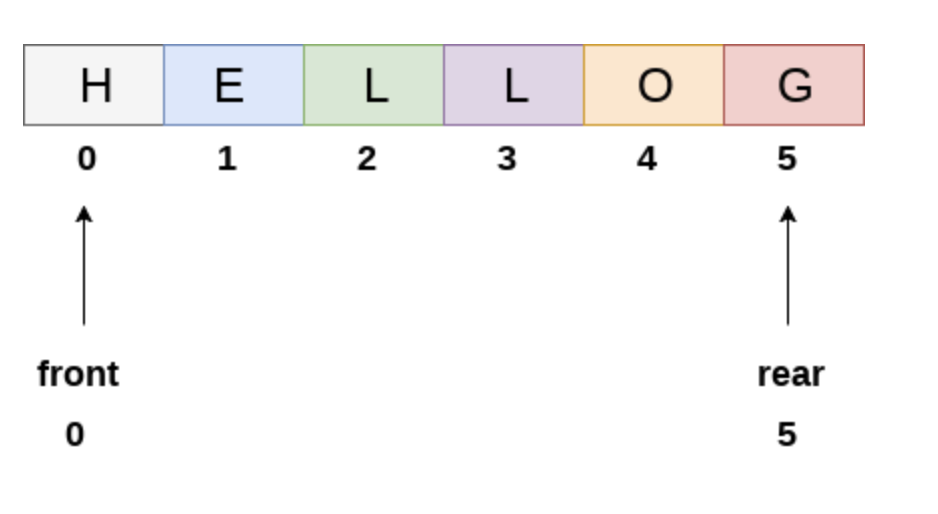
if IsEmpty:

print "Queue is Empty"

else:

print queue[rear]

Time Complexity: O(1) - constant time to retrieve the rear element.



**Rear=G**

**IsEmpty:**

Algorithm: Check if the queue is empty.

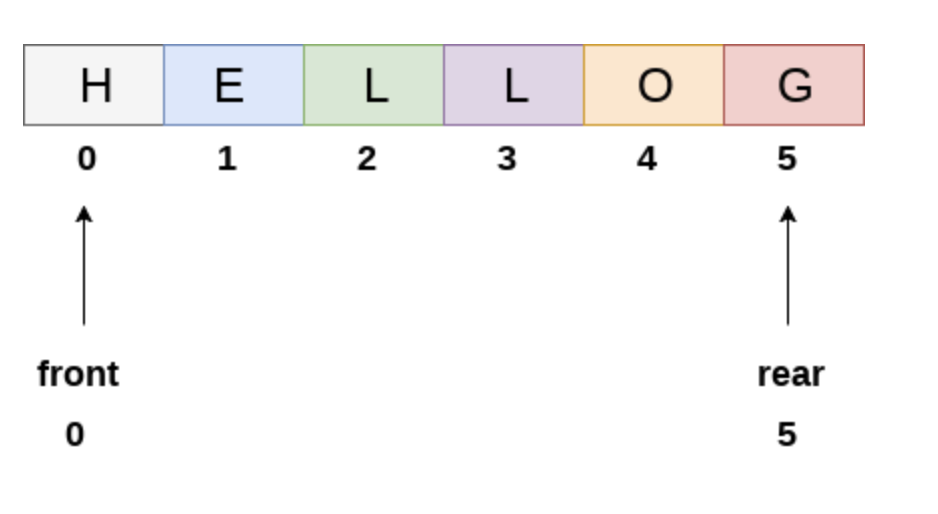
if front > rear:

return True

else:

return False

Time Complexity: O(1) - constant time to check if the queue is empty.



**IsEmpty=False**

**IsFull:**

Algorithm: Check if the queue is full.

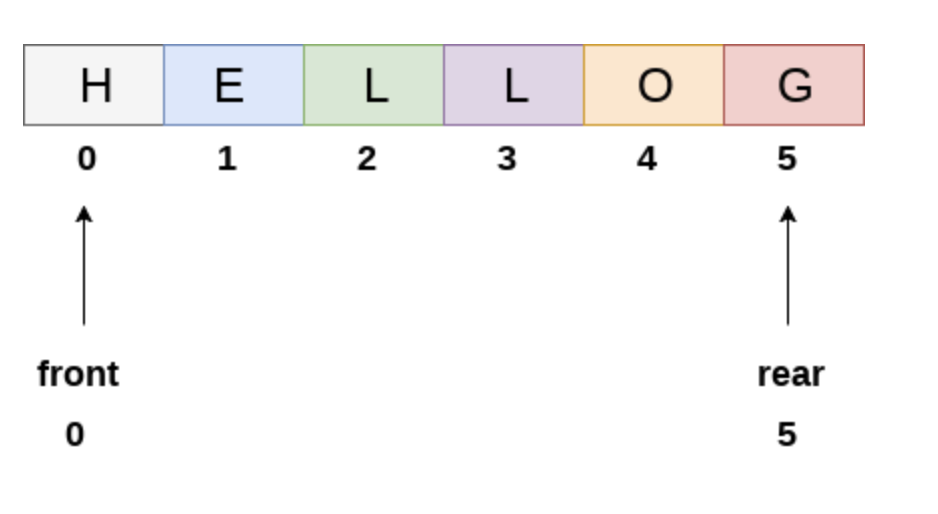
if rear == capacity - 1:

return True

else:

return False

Time Complexity: O(1) - constant time to check if the queue is full.



**IsFull=True**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define maxsize 5

void insert();

void delete();

void display();

int size();

int Front();

int Rear();

int isEmpty();

int isFull();

int queue[maxsize];

int front = -1, rear = -1;

void main()

{

int choice;

while (choice != 9)

{

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\n=================================================================\n");

printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n");

printf("4.Size of the queue\n5.Front element\n6.Rear element\n");

printf("7.Is Empty?\n8.Is Full?\n9.Exit\n");

printf("\nEnter your choice ?");

scanf("%d", &choice);

switch (choice)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

display();

break;

case 4:

printf("\nSize of the queue: %d\n", size());

break;

case 5:

printf("\nFront element: %d\n", Front());

break;

case 6:

printf("\nRear element: %d\n", Rear());

break;

case 7:

if (isEmpty())

printf("\nQueue is empty.\n");

else

printf("\nQueue is not empty.\n");

break;

case 8:

if (isFull())

printf("\nQueue is full.\n");

else

printf("\nQueue is not full.\n");

break;

case 9:

exit(0);

break;

default:

printf("\nEnter valid choice??\n");

}

}

}

void insert()

{

int item;

printf("\nEnter the element\n");

scanf("\n%d", &item);

if (isFull())

{

printf("\nOVERFLOW\n");

return;

}

if (isEmpty())

{

front = 0;

rear = 0;

}

else

{

rear = rear + 1;

}

queue[rear] = item;

printf("\nValue inserted ");

}

void delete()

{

int item;

if (isEmpty())

{

printf("\nUNDERFLOW\n");

return;

}

else

{

item = queue[front];

if (front == rear)

{

front = -1;

rear = -1;

}

else

{

front = front + 1;

}

printf("\nvalue deleted ");

}

}

void display()

{

int i;

if (isEmpty())

{

printf("\nEmpty queue\n");

}

else

{

printf("\nprinting values .....\n");

for (i = front; i <= rear; i++)

{

printf("\n%d\n", queue[i]);

}

}

}

int size()

{

if (isEmpty())

return 0;

else

return rear - front + 1;

}

int Front()

{

if (isEmpty())

{

printf("\nQueue is empty.\n");

return -1;

}

return queue[front];

}

int Rear()

{

if (isEmpty())

{

printf("\nQueue is empty.\n");

return -1;

}

return queue[rear];

}

int isEmpty()

{

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

}

int isFull()

{

return ((rear + 1) % maxsize == front);

}

**Output:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

=================================================================

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

6.Rear element

7.Is Empty?

8.Is Full?

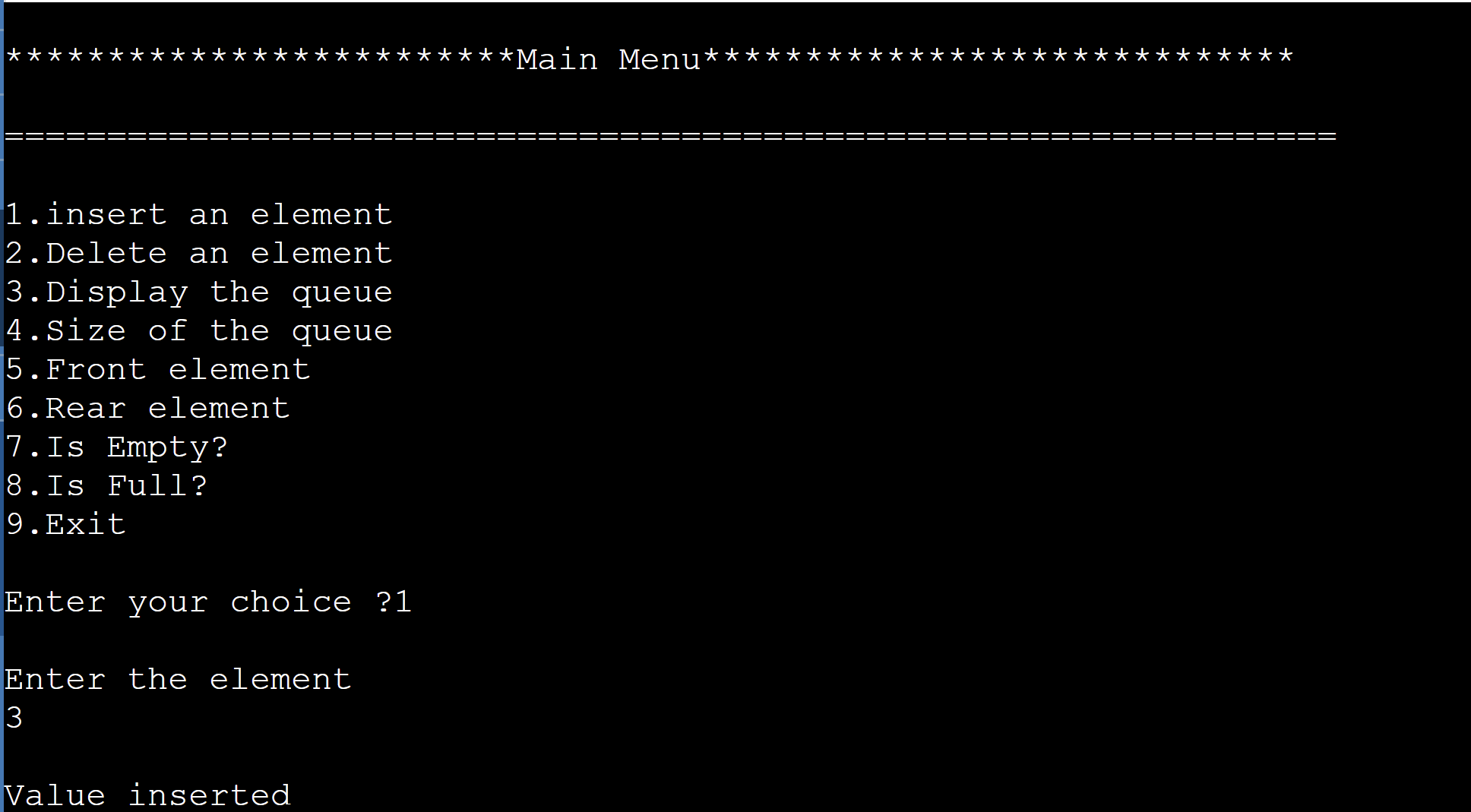
9.Exit

Enter your choice ?1

Enter the element

3

Value inserted

****

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

=================================================================

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

6.Rear element

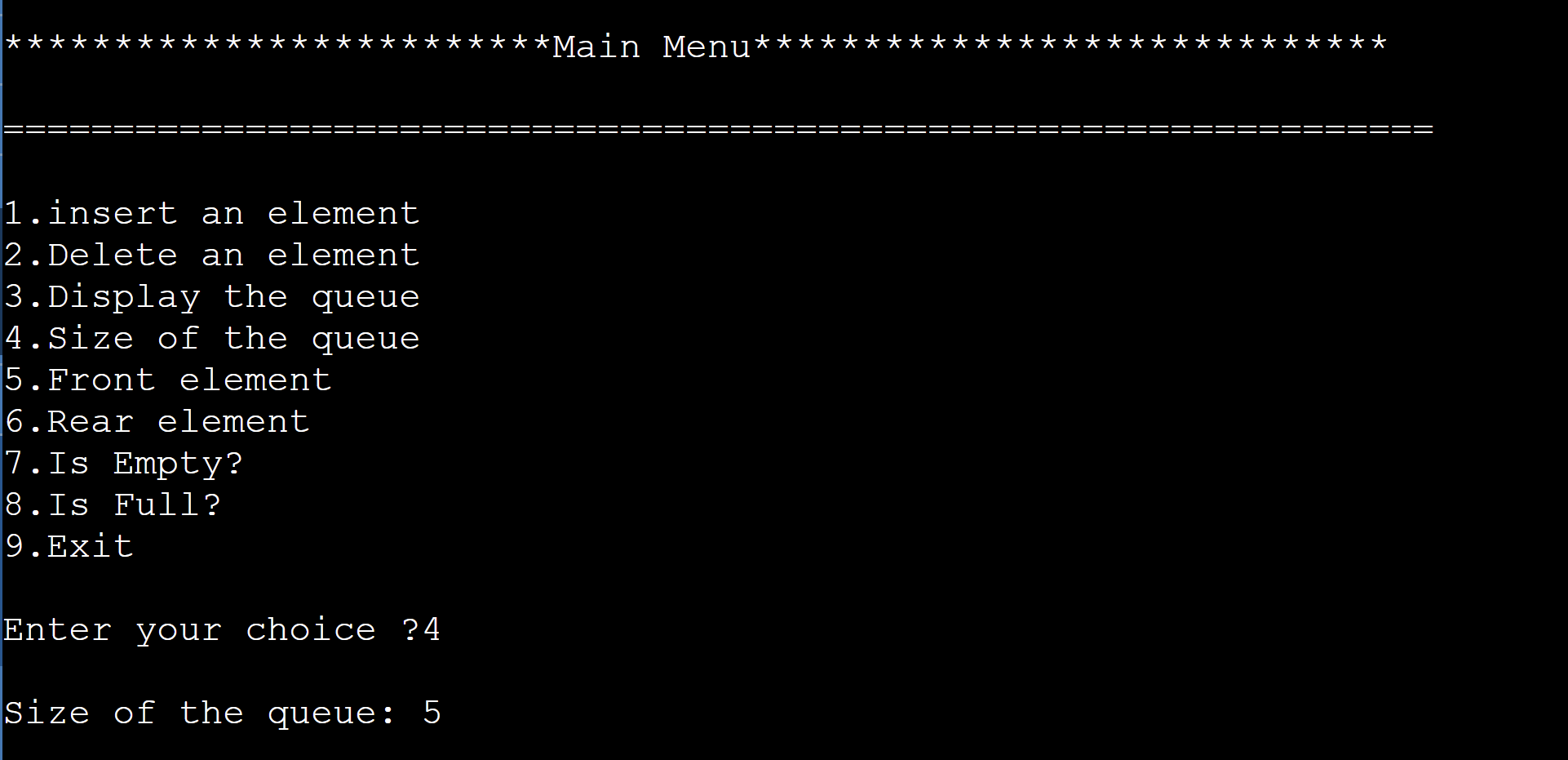
7.Is Empty?

8.Is Full?

9.Exit

Enter your choice ?4

Size of the queue: 5

****

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

=================================================================

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

6.Rear element

7.Is Empty?

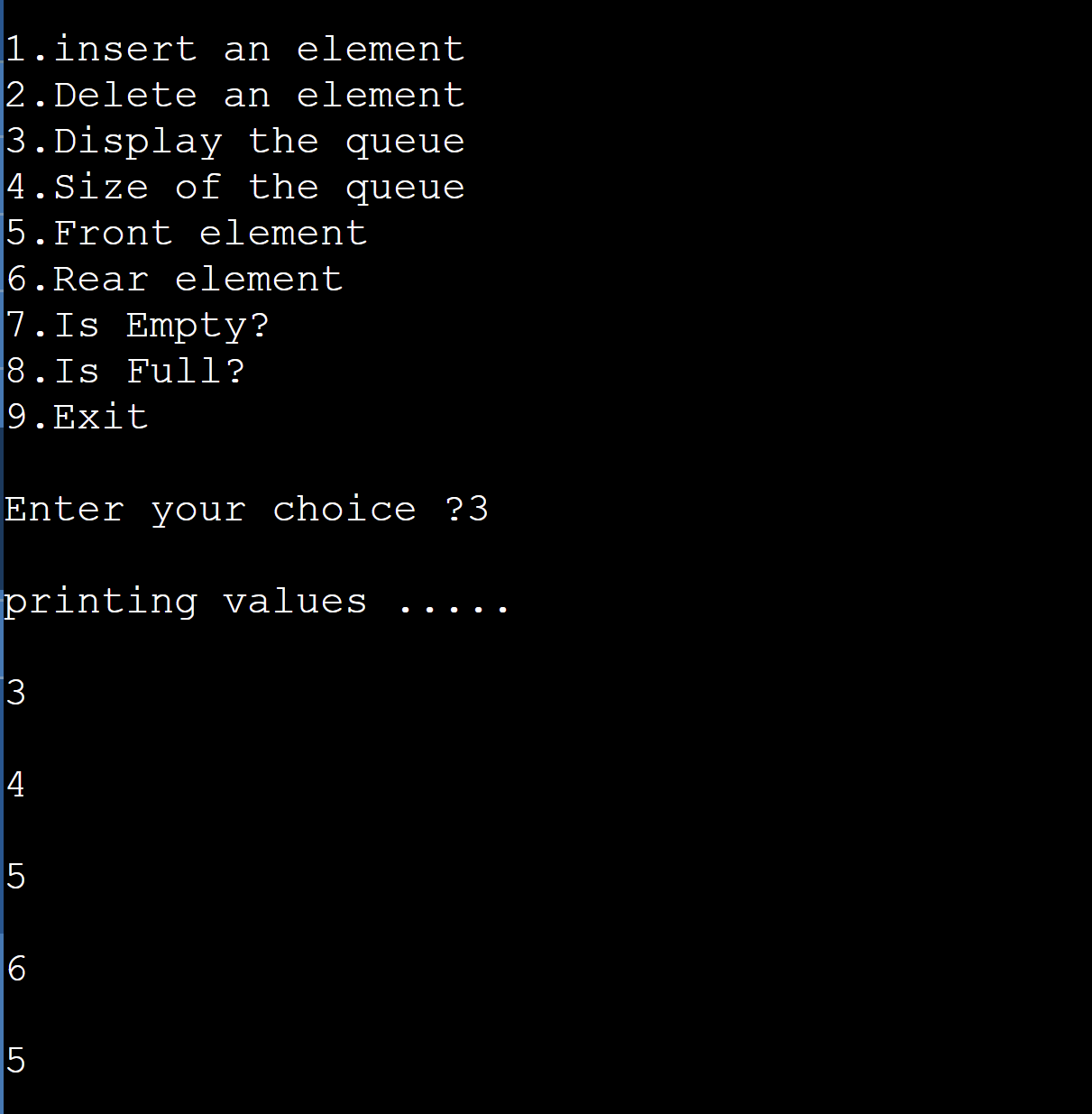
8.Is Full?

9.Exit

Enter your choice ?3

printing values .....

3 4 5 6 5



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

=================================================================

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

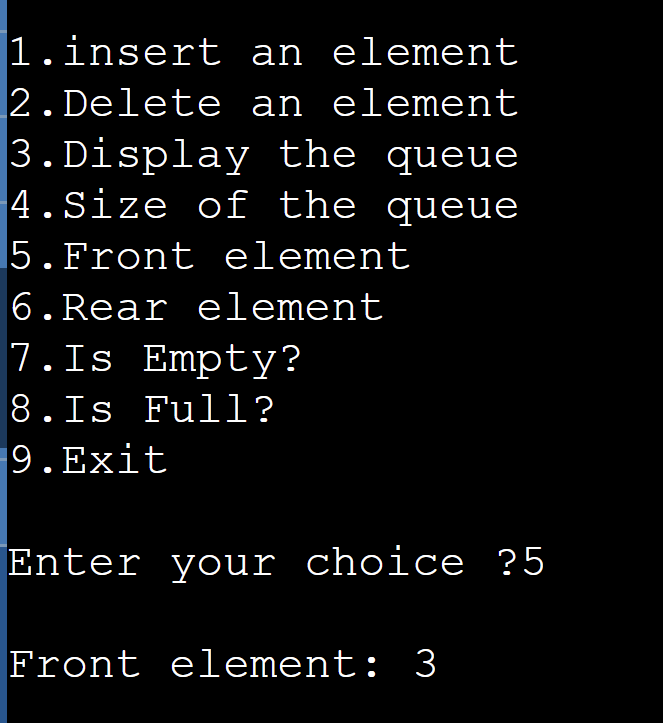
6.Rear element

7.Is Empty?

8.Is Full?

9.Exit

Enter your choice ?5

Front element: 3****

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

6.Rear element

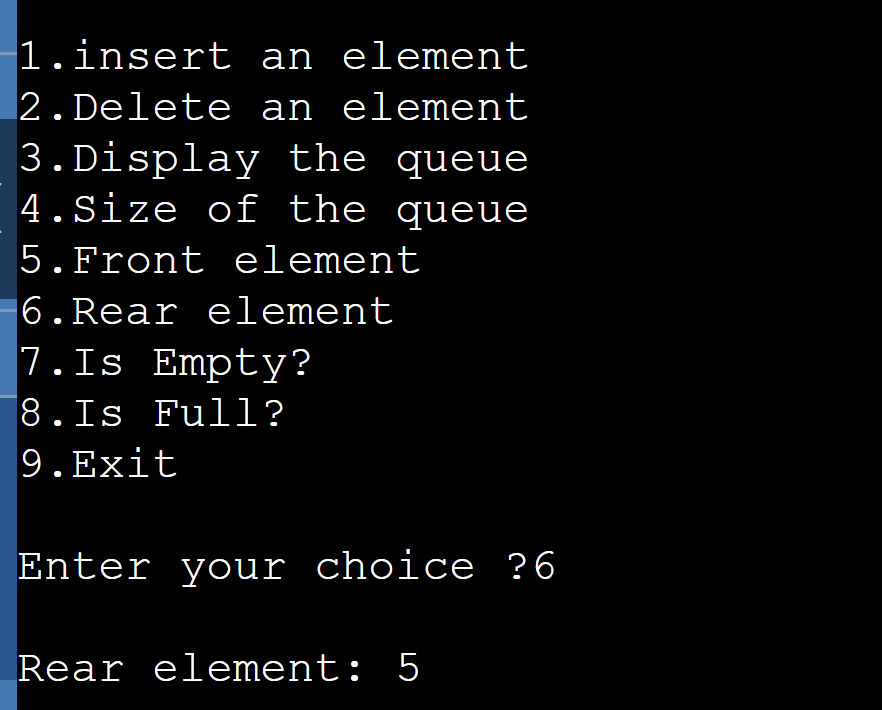
7.Is Empty?

8.Is Full?

9.Exit

Enter your choice ?6

Rear element: 5

****

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

6.Rear element

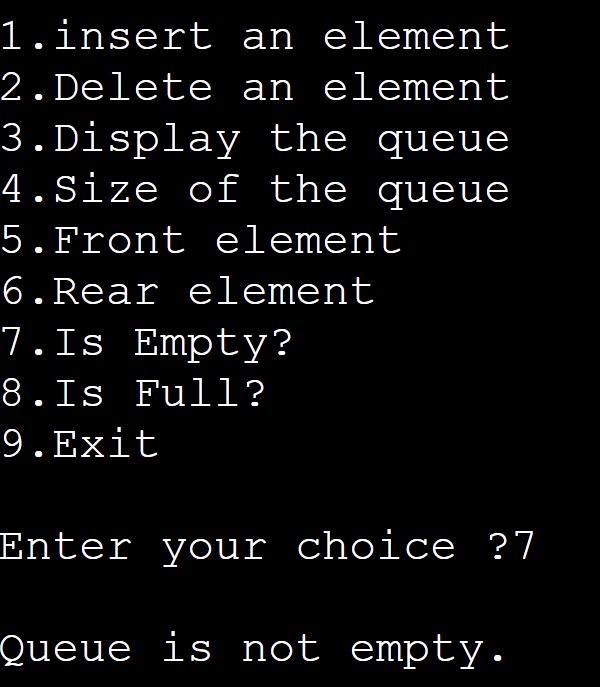
7.Is Empty?

8.Is Full?

9.Exit

Enter your choice ?7

Queue is not empty.

****

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

6.Rear element

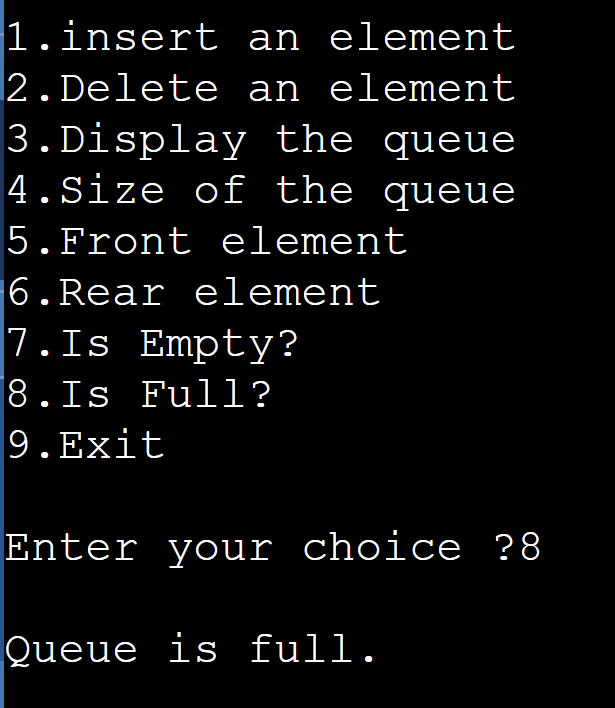
7.Is Empty?

8.Is Full?

9.Exit

Enter your choice ?8

Queue is full.

****

1.insert an element

2.Delete an element

3.Display the queue

4.Size of the queue

5.Front element

6.Rear element

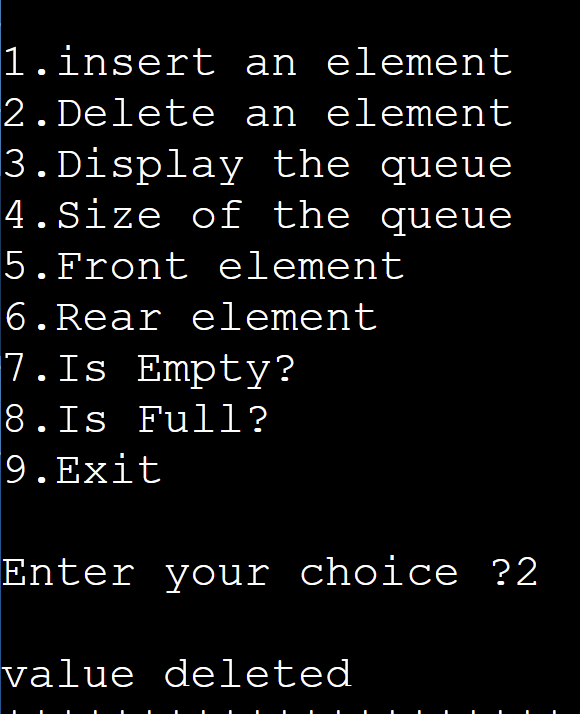
7.Is Empty?

8.Is Full?

9.Exit

Enter your choice ?2

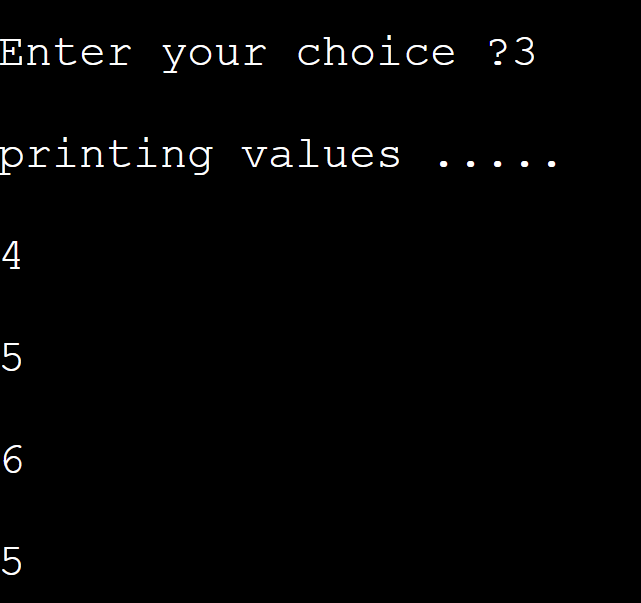
value deleted

****

Enter your choice ?3

printing values .....

4 5 6 5

**Operations on Queues using linked list:**

**Enqueue (Insertion):**

Pseudocode:

Enqueue(value):

newNode = createNode(value)

if rear is NULL:

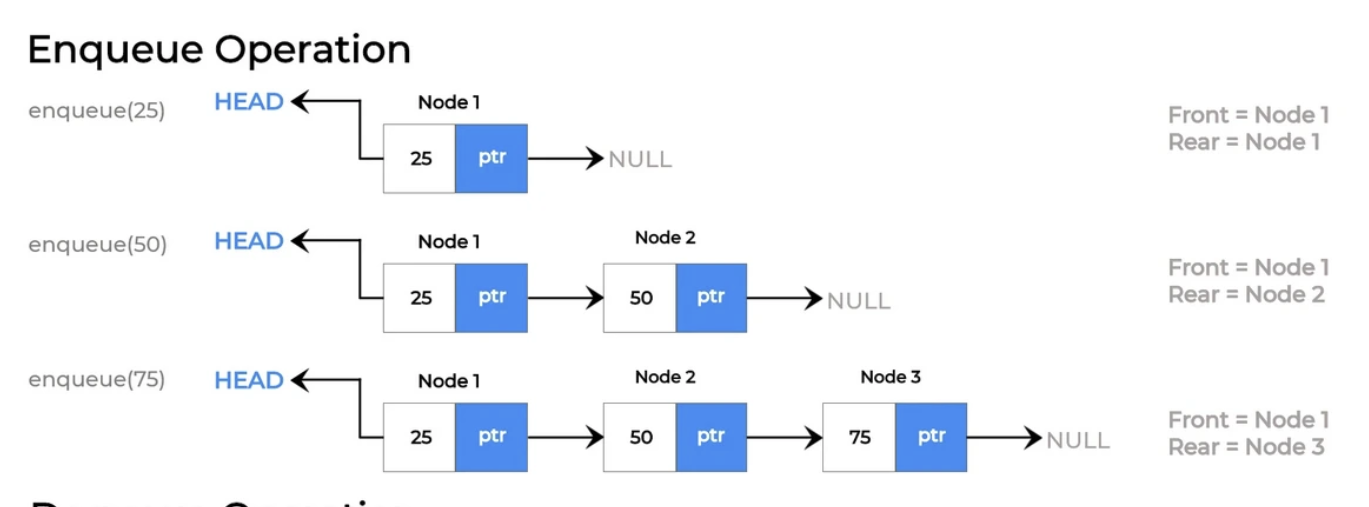
front = rear = newNode

else:

rear.next = newNode

rear = newNode

Time Complexity: O(1)



**Dequeue (Deletion):**

Pseudocode:

Dequeue():

if front is NULL:

print "Queue is empty"

return

else if front is equal to rear:

value = front.data

front = rear = NULL

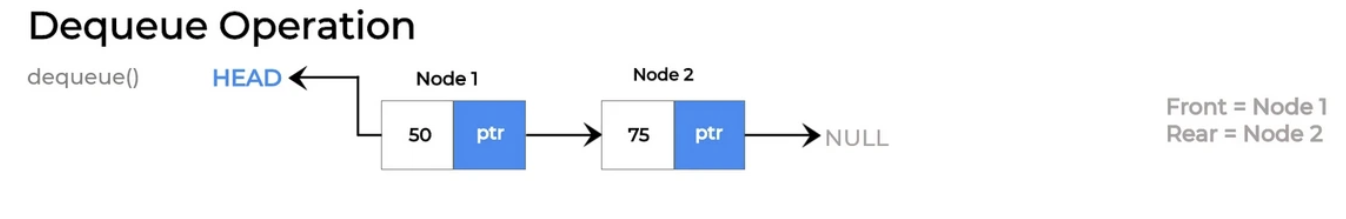
else:

value = front.data

front = front.next

return value

Time Complexity: O(1)

**Display:**

Pseudocode:

Display():

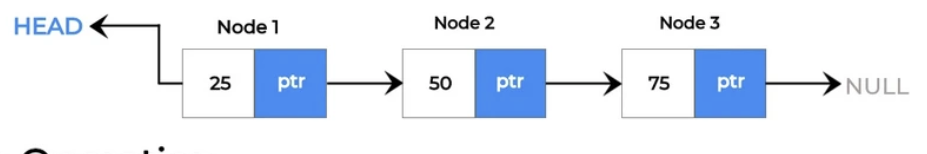
current = front

while current is not NULL:

print current.data

current = current.next

Time Complexity: O(n) where n is the number of elements in the queue.



**Output:25 50 75**

**Size:**

Pseudocode:

Size():

count = 0

current = front

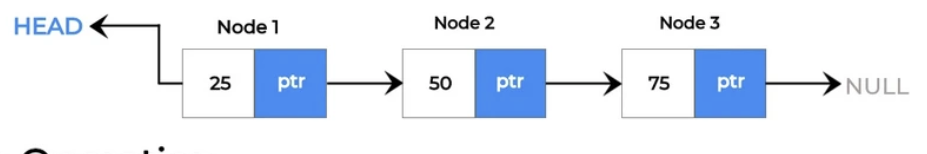
while current is not NULL:

count = count + 1

current = current.next

return count

Time Complexity: O(n) where n is the number of elements in the queue.

****

**Size=3**

**Front:**

Pseudocode:

Front():

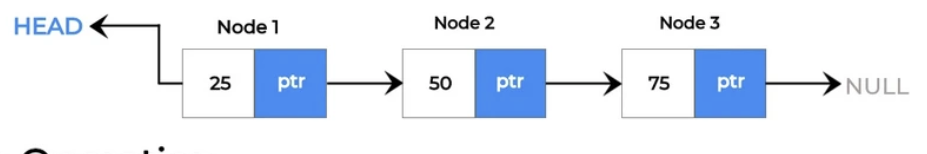
if front is NULL:

print "Queue is empty"

return

return front.data

Time Complexity: O(1)



**Front=25**

**Rear:**

Pseudocode:

Rear():

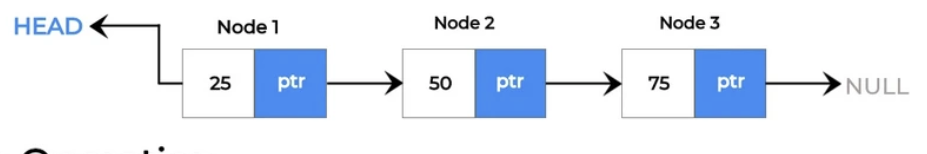
if rear is NULL:

print "Queue is empty"

return

return rear.data

Time Complexity: O(1)

****

**Rear=75**

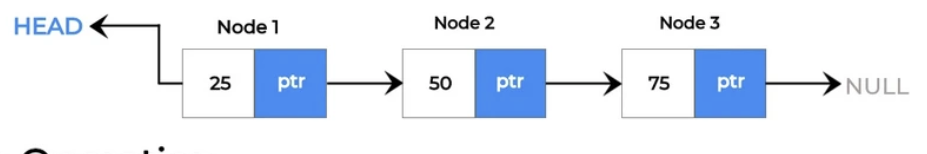
**Is Empty:**

Pseudocode:

IsEmpty():

return front is NULL

Time Complexity: O(1)



**IsEmpty=False**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* front = NULL;

struct Node\* rear = NULL;

void enqueue(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

printf("Queue is full. Unable to enqueue.\n");

return;

}

newNode->data = value;

newNode->next = NULL;

if (rear == NULL) {

front = rear = newNode;

} else {

rear->next = newNode;

rear = newNode;

}

printf("%d enqueued to the queue.\n", value);

}

void dequeue() {

if (front == NULL) {

printf("Queue is empty. Unable to dequeue.\n");

return;

}

struct Node\* temp = front;

front = front->next;

if (front == NULL) {

rear = NULL;

}

int value = temp->data;

free(temp);

printf("%d dequeued from the queue.\n", value);

}

void display() {

if (front == NULL) {

printf("Queue is empty.\n");

return;

}

struct Node\* current = front;

printf("Queue elements: ");

while (current != NULL) {

printf("%d ", current->data);

current = current->next;

}

printf("\n");

}

int size() {

int count = 0;

struct Node\* current = front;

while (current != NULL) {

count++;

current = current->next;

}

return count;

}

int frontValue() {

if (front == NULL) {

printf("Queue is empty.\n");

return -1; // Assuming -1 as an indicator of an empty queue

}

return front->data;

}

int rearValue() {

if (rear == NULL) {

printf("Queue is empty.\n");

return -1; // Assuming -1 as an indicator of an empty queue

}

return rear->data;

}

int isEmpty() {

return front == NULL;

}

int main() {

int choice, value;

while (1) {

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display\n");

printf("4. Size\n");

printf("5. Front\n");

printf("6. Rear\n");

printf("7. Is Empty\n");

printf("8. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to enqueue: ");

scanf("%d", &value);

enqueue(value);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

case 4:

printf("Size of the queue: %d\n", size());

break;

case 5:

printf("Front element: %d\n", frontValue());

break;

case 6:

printf("Rear element: %d\n", rearValue());

break;

case 7:

printf("Queue is %s\n", isEmpty() ? "empty" : "not empty");

break;

case 8:

printf("Exiting the program.\n");

exit(0);

default:

printf("Invalid choice. Please enter a valid option.\n");

}

}

return 0;

}

**Output:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Enqueue

2. Dequeue

3. Display

4. Size

5. Front

6. Rear

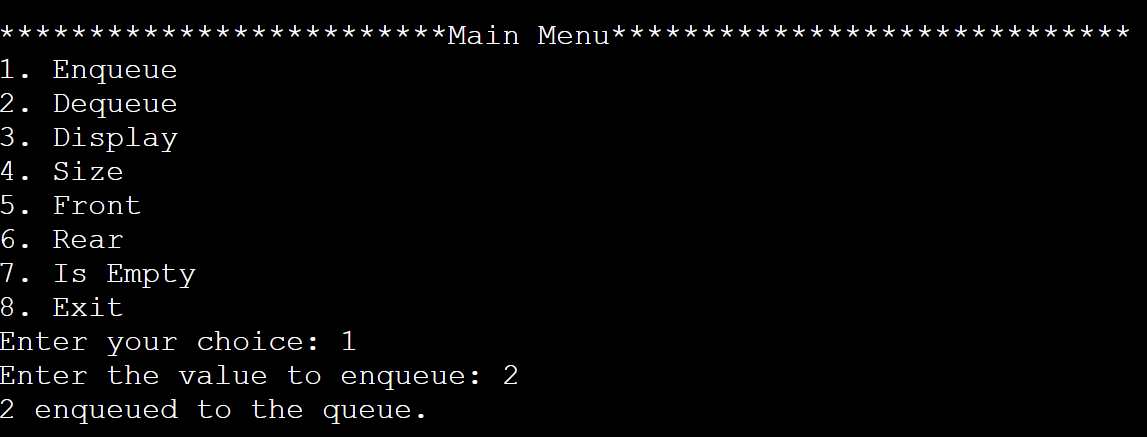
7. Is Empty

8. Exit

Enter your choice: 1

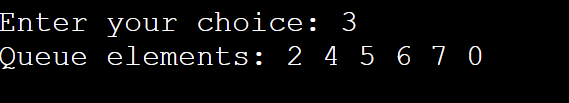
Enter the value to enqueue: 2

2 enqueued to the queue.

****

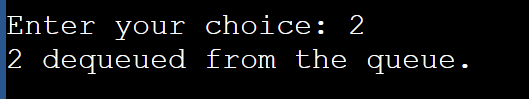
Enter your choice: 3

Queue elements: 2 4 5 6 7 0



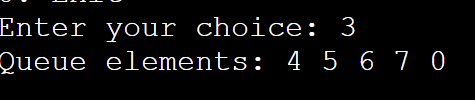
Enter your choice: 2

2 dequeued from the queue.



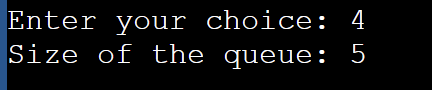
Enter your choice: 3

Queue elements: 4 5 6 7 0



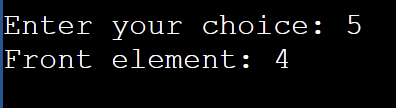
Enter your choice: 4

Size of the queue: 5



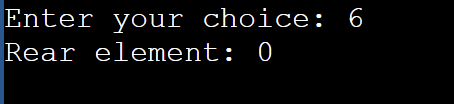
Enter your choice: 5

Front element: 4



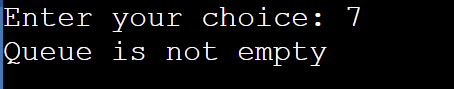
Enter your choice: 6

Rear element: 0

****

Enter your choice: 7

Queue is not empty



**Operations on CIRCULAR QUEUES using Arrays:**

**INSERT (Enqueue):**

Algorithm:

if IsFull:

print "Queue Overflow"

else:

rear = (rear + 1) % capacity

queue[rear] = element

Time Complexity: O(1) - constant time for inserting an element.

**DELETE (Dequeue):**

Algorithm:

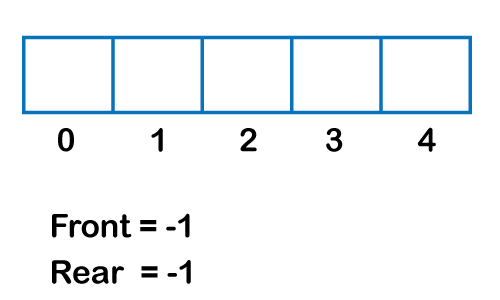
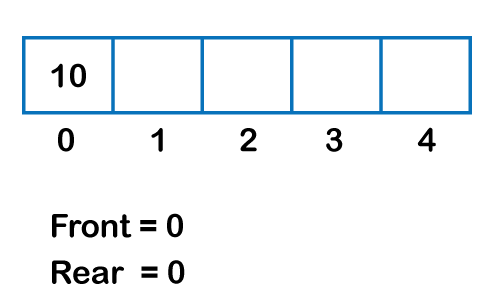
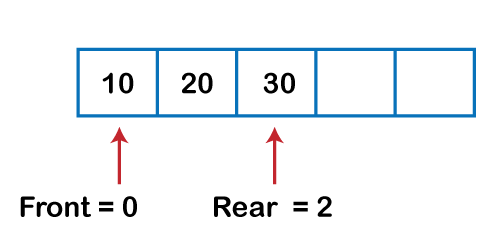
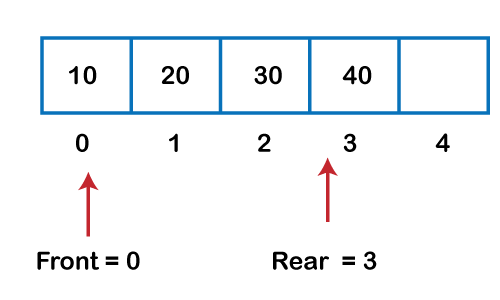
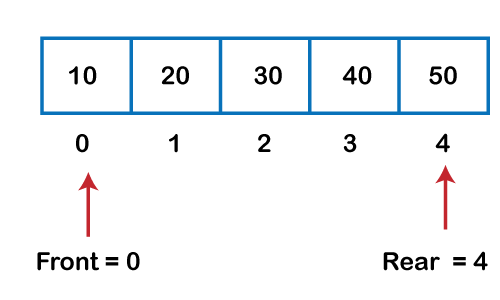
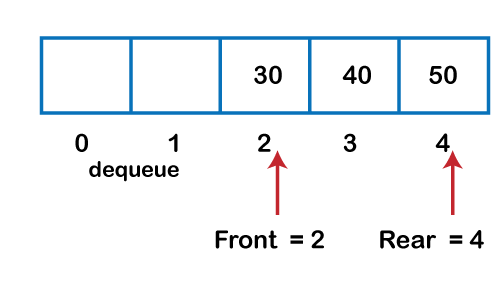
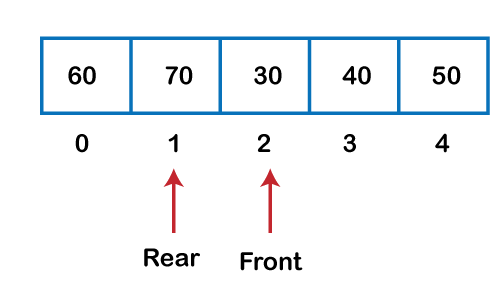
if IsEmpty:

print "Queue Underflow"

else:

element = queue[front]

front = (front + 1) % capacity

Time Complexity: O(1) - constant time for deleting an element.  
  
  
  
  
  


**SIZE:**

Algorithm:

size = (rear - front + capacity) % capacity

Time Complexity: O(1) - constant time for calculating the size.

**Display:**

Algorithm:

i = front

while i != (rear + 1) % capacity:

print queue[i]

i = (i + 1) % capacity

Time Complexity: O(n) - linear time where n is the number of elements.

**FRONT:**

Algorithm:

if IsEmpty:

print "Queue is Empty"

else:

print queue[front]

Time Complexity: O(1) - constant time to retrieve the front element.

**REAR:**

Algorithm:

if IsEmpty:

print "Queue is Empty"

else:

print queue[rear]

Time Complexity: O(1) - constant time to retrieve the rear element.

**IsEmpty:**

Algorithm:

if front == rear:

return True

else:

return False

Time Complexity: O(1) - constant time to check if the queue is empty.

**IsFull:**

Algorithm:

if (rear + 1) % capacity == front:

return True

else:

return False

Time Complexity: O(1) - constant time to check if the queue is full.

**Program:**

#include <stdio.h>

#define max 4

int queue[max]; // array declaration

int front = -1;

int rear = -1;

int isEmpty() {

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

}

int isFull() {

return (front == (rear + 1) % max);

}

int size() {

if (isEmpty())

return 0;

else

return (rear - front + max) % max + 1;

}

int Front() {

if (isEmpty())

{

printf("\nQueue is empty.\n");

return -1;

}

return queue[front];

}

int Rear() {

if (isEmpty())

{

printf("\nQueue is empty.\n");

return -1;

}

return queue[rear];

}

void enqueue(int element) {

if (isEmpty()) // condition to check if the queue is empty

{

front = 0;

rear = 0;

queue[rear] = element;

}

else if (isFull()) // condition to check if the queue is full

{

printf("Queue is overflow.\n");

}

else

{

rear = (rear + 1) % max; // rear is incremented

queue[rear] = element; // assigning a value to the queue at the rear position.

}

}

int dequeue() {

if (isEmpty()) // condition to check if the queue is empty

{

printf("Queue is underflow.\n");

}

else if (front == rear)

{

printf("The dequeued element is %d\n", queue[front]);

front = -1;

rear = -1;

}

else

{

printf("The dequeued element is %d\n", queue[front]);

front = (front + 1) % max;

}

}

void display() {

int i = front;

if (isEmpty())

{

printf("Queue is empty.\n");

}

else

{

printf("Elements in a Queue are: ");

while (i != rear)

{

printf("%d, ", queue[i]);

i = (i + 1) % max;

}

printf("%d\n", queue[rear]);

}

}

int main() {

int choice = 1, x; // variables declaration

while (choice < 10 && choice != 0) // while loop

{

printf("\n Press 1: Insert an element");

printf("\nPress 2: Delete an element");

printf("\nPress 3: Display the element");

printf("\nPress 4: Front element");

printf("\nPress 5: Rear element");

printf("\nPress 6: Size of the Queue");

printf("\nPress 7: Check if the Queue is full");

printf("\nPress 8: Check if the Queue is empty");

printf("\nPress 9: Exit");

printf("\nEnter your choice:");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("Enter the element which is to be inserted");

scanf("%d", &x);

enqueue(x);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

case 4:

printf("Front element: %d\n", Front());

break;

case 5:

printf("Rear element: %d\n", Rear());

break;

case 6:

printf("Size of the Queue: %d\n", size());

break;

case 7:

printf("Queue is %s\n", isFull() ? "full" : "not full");

break;

case 8:

printf("Queue is %s\n", isEmpty() ? "empty" : "not empty");

break;

case 9:

break;

default:

printf("\nEnter valid choice (1-9).\n");

}

}

return 0;

}

**Output:**

Press 1: Insert an element

Press 2: Delete an element

Press 3: Display the element

Press 4: Front element

Press 5: Rear element

Press 6: Size of the Queue

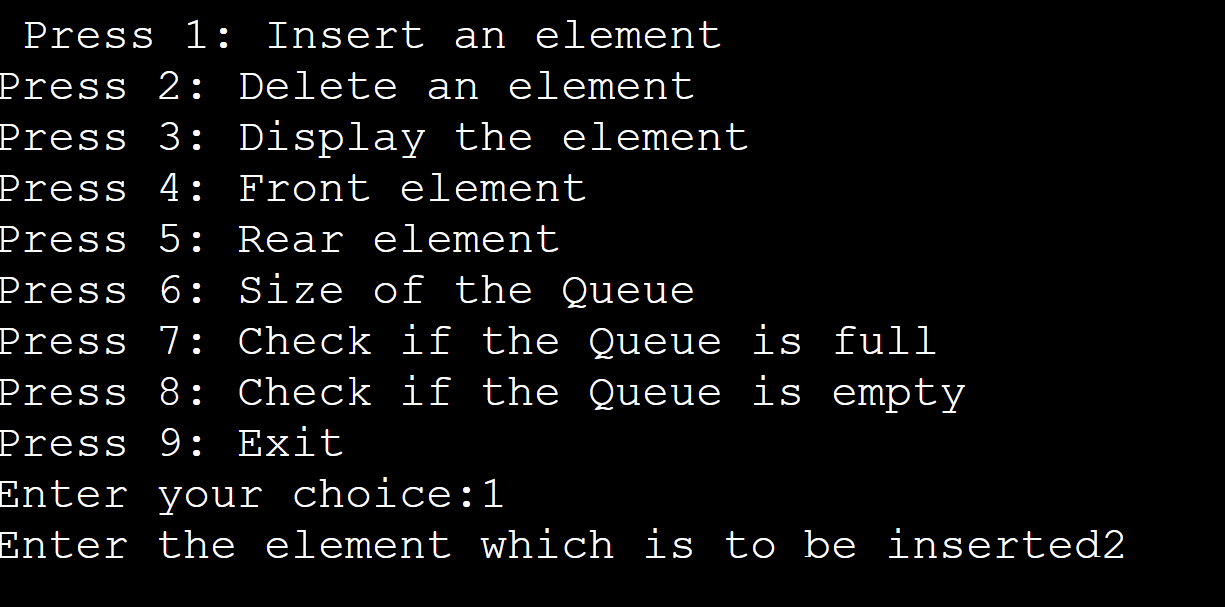
Press 7: Check if the Queue is full

Press 8: Check if the Queue is empty

Press 9: Exit

Enter your choice:1

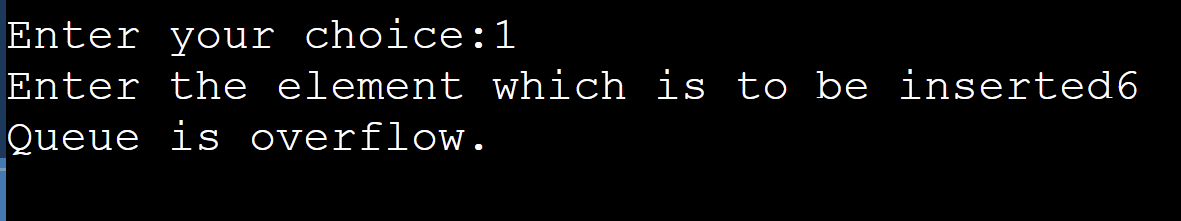
Enter the element which is to be inserted2



Enter your choice:1

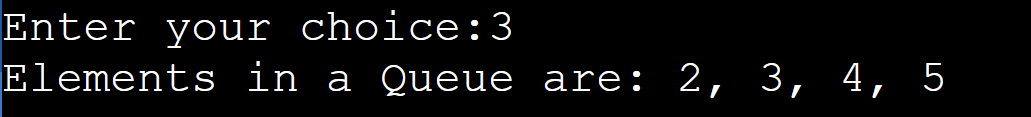
Enter the element which is to be inserted6

Queue is overflow.



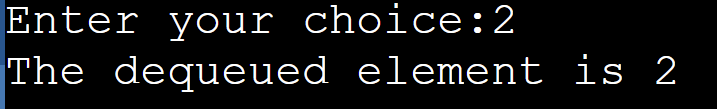
Enter your choice:3

Elements in a Queue are: 2, 3, 4, 5



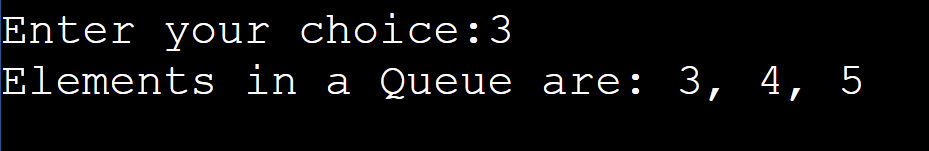
Enter your choice:2

The dequeued element is 2



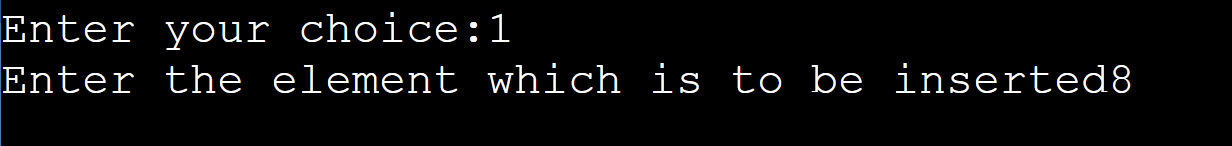
Enter your choice:3

Elements in a Queue are: 3, 4, 5



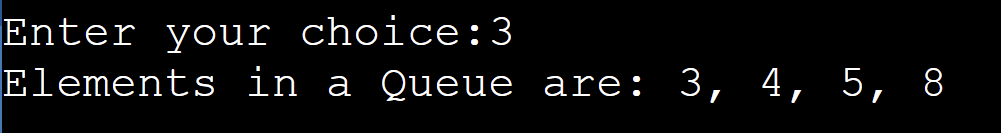
Enter your choice:1

Enter the element which is to be inserted8



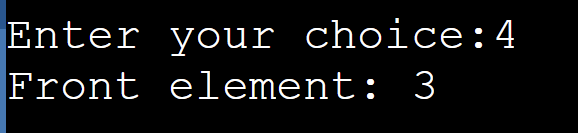
Enter your choice:3

Elements in a Queue are: 3, 4, 5, 8



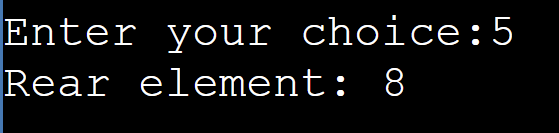
Enter your choice:4

Front element: 3



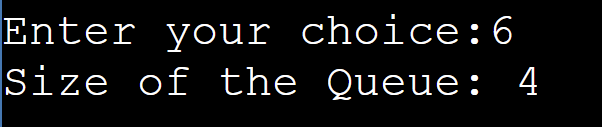
Enter your choice:5

Rear element: 8



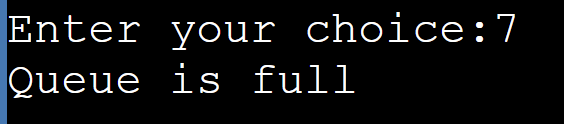
Enter your choice:6

Size of the Queue: 4



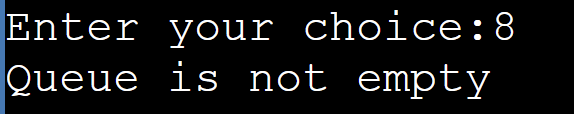
Enter your choice:7

Queue is full



Enter your choice:8

Queue is not empty



EXPERIMENT-8

**Problem**

You are given an array A of Q integers and Q queries. In each query, you are given an integer i (1≤i≤N).

Your task is to find the minimum index greater than i (1≤i≤N) such that:

1. Sum of digits of Ai is greater than the sum of digits of Aj
2. Ai < Aj

If there is no answer, then print **-1**.

**Program:**

#include <stdio.h>

int calculateDigitSum(int number) {

int digitSum = 0;

while (number > 0) {

int lastDigit = number % 10;

digitSum += lastDigit;

number -= lastDigit;

number /= 10;

}

return digitSum;

}

int main() {

int arraySize, queries;

scanf("%d %d", &arraySize, &queries);

int numbers[arraySize];

int digitSums[arraySize];

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

scanf("%d", &numbers[i]);

digitSums[i] = calculateDigitSum(numbers[i]);

}

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

int queryIndex;

scanf("%d", &queryIndex);

int isDone = 0;

for (int j = queryIndex - 1; j < arraySize; j++) {

if (digitSums[j] < digitSums[queryIndex - 1] && numbers[j] > numbers[queryIndex - 1]) {

isDone = 1;

printf("%d\n", j + 1);

break;

}

}

if (isDone == 0)

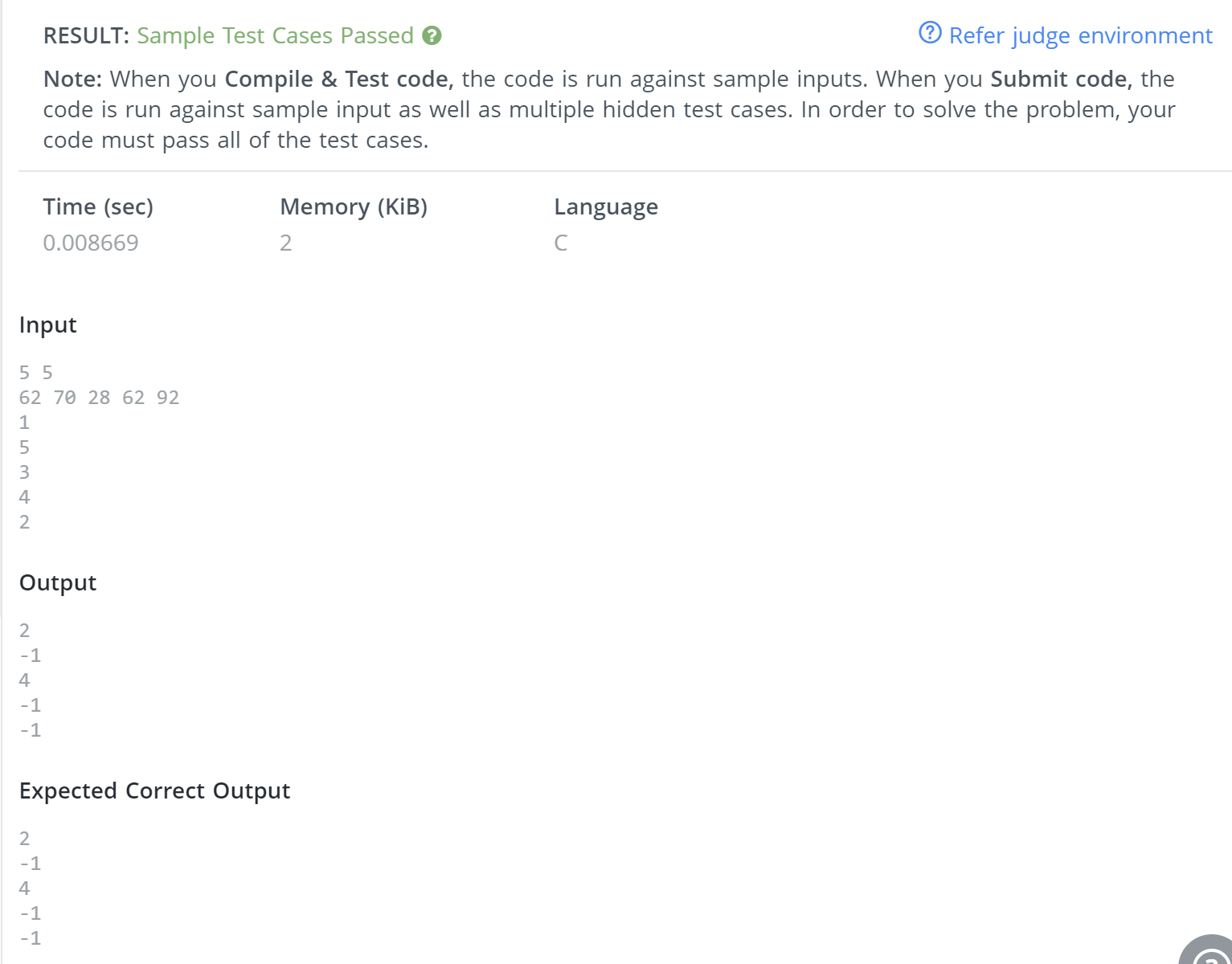
printf("-1\n");

}

return 0;

}

**Output:**

****

**Problem:**

You are given a stack of **N** integers such that the first element represents the top of the stack and the last element represents the bottom of the stack. You need to pop at least one element from the stack. At any one moment, you can convert stack into a queue. The bottom of the stack represents the front of the queue. You cannot convert the queue back into a stack. Your task is to remove exactly **K** elements such that the sum of the **K**removed elements is maximised.

**Program:**

#include <stdio.h>

#define MAX\_SIZE 2000000

typedef long long ll;

int main() {

ll size, windowSize;

scanf("%lld %lld", &size, &windowSize);

ll elements[size + 1];

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

scanf("%lld", &elements[i]);

}

ll cumulativeSum[size + 1];

cumulativeSum[0] = elements[0];

for (int i = 1; i < size; i++) {

cumulativeSum[i] = cumulativeSum[i - 1] + elements[i];

}

ll maxSum = 0;

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

ll tempSum = cumulativeSum[i] + cumulativeSum[size - 1] - cumulativeSum[size - windowSize + i];

maxSum = (maxSum > tempSum) ? maxSum : tempSum;

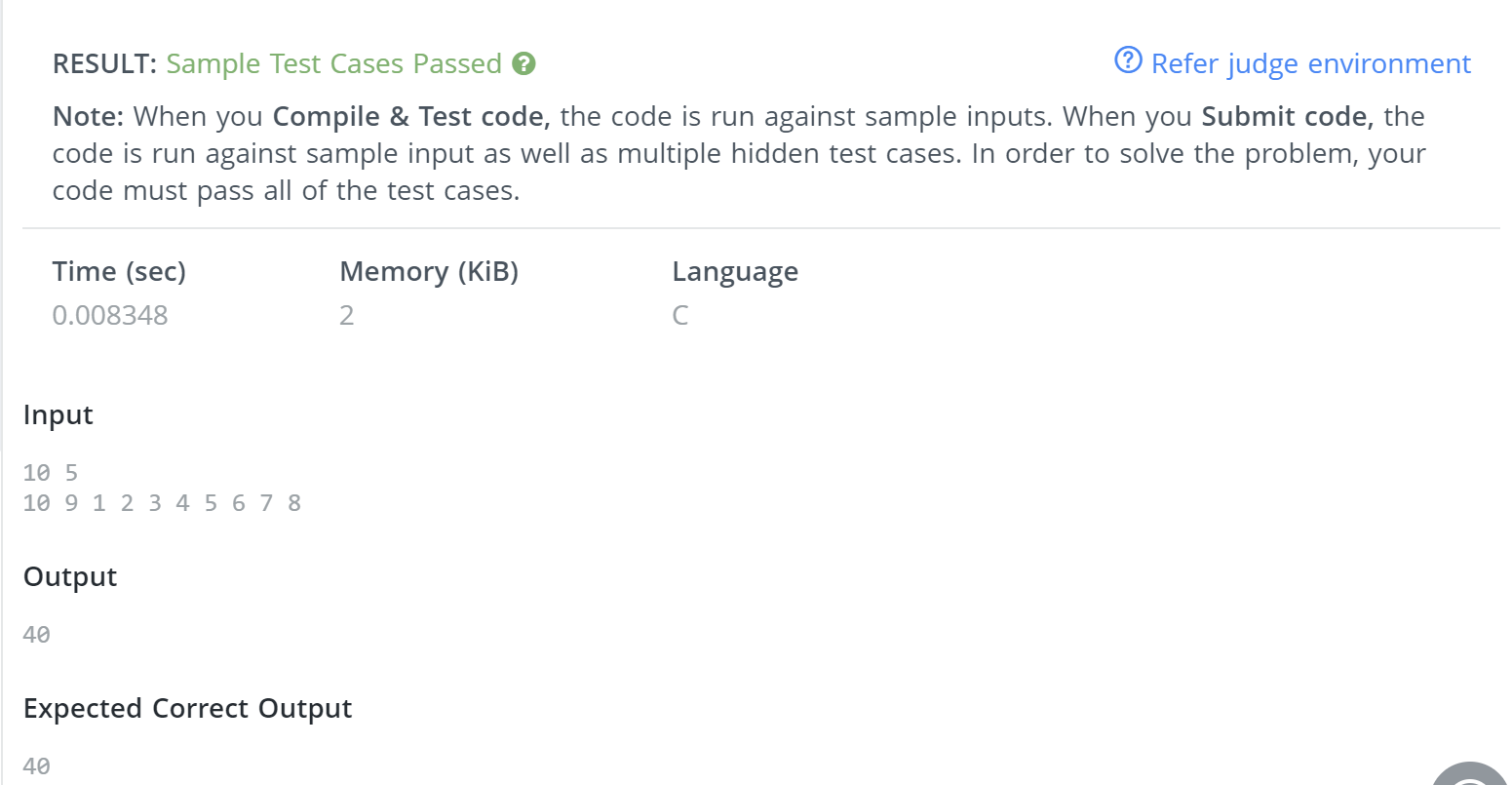
}

printf("%lld", maxSum);

return 0;

}

**Output:**

****

**PROBLEM:**

You are given two arrays each of size n, a and b consisting of the first n positive integers each exactly once, that is, they are permutations.

Your task is to find the minimum time required to make both the arrays empty. The following two types of operations can be performed any number of times each taking 1 second:

* In the first operation, you are allowed to rotate the first array clockwise.
* In the second operation, when the first element of both the arrays is the same, they are removed from both the arrays and the process continues.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 1000000

typedef struct {

    int elements[MAX\_SIZE];

    int front;

    int rear;

} JobQueue;

void enqueue(JobQueue\* jobQ, int x) {

    jobQ->elements[++jobQ->rear] = x;

}

int dequeue(JobQueue\* jobQ) {

    return jobQ->elements[jobQ->front++];

}

int peek(JobQueue\* jobQ) {

    return jobQ->elements[jobQ->front];

}

int isEmpty(JobQueue\* jobQ) {

    return jobQ->front > jobQ->rear;

}

int main() {

    int numJobs, jobNumber;

    scanf("%d", &numJobs);

    JobQueue jobQueue;

    jobQueue.front = 0;

    jobQueue.rear = -1;

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

        scanf("%d", &jobNumber);

        enqueue(&jobQueue, jobNumber);

    }

    int jobArray[numJobs];

    for (int i = 0; i < numJobs; i++)

        scanf("%d", &jobArray[i]);

    int totalTime = 0, executedJob = 0;

    while (!isEmpty(&jobQueue)) {

        int currentJob = peek(&jobQueue);

        if (currentJob == jobArray[executedJob]) {

            dequeue(&jobQueue);

            totalTime++;

            executedJob++;

        } else {

            enqueue(&jobQueue, dequeue(&jobQueue));

            totalTime++;

        }

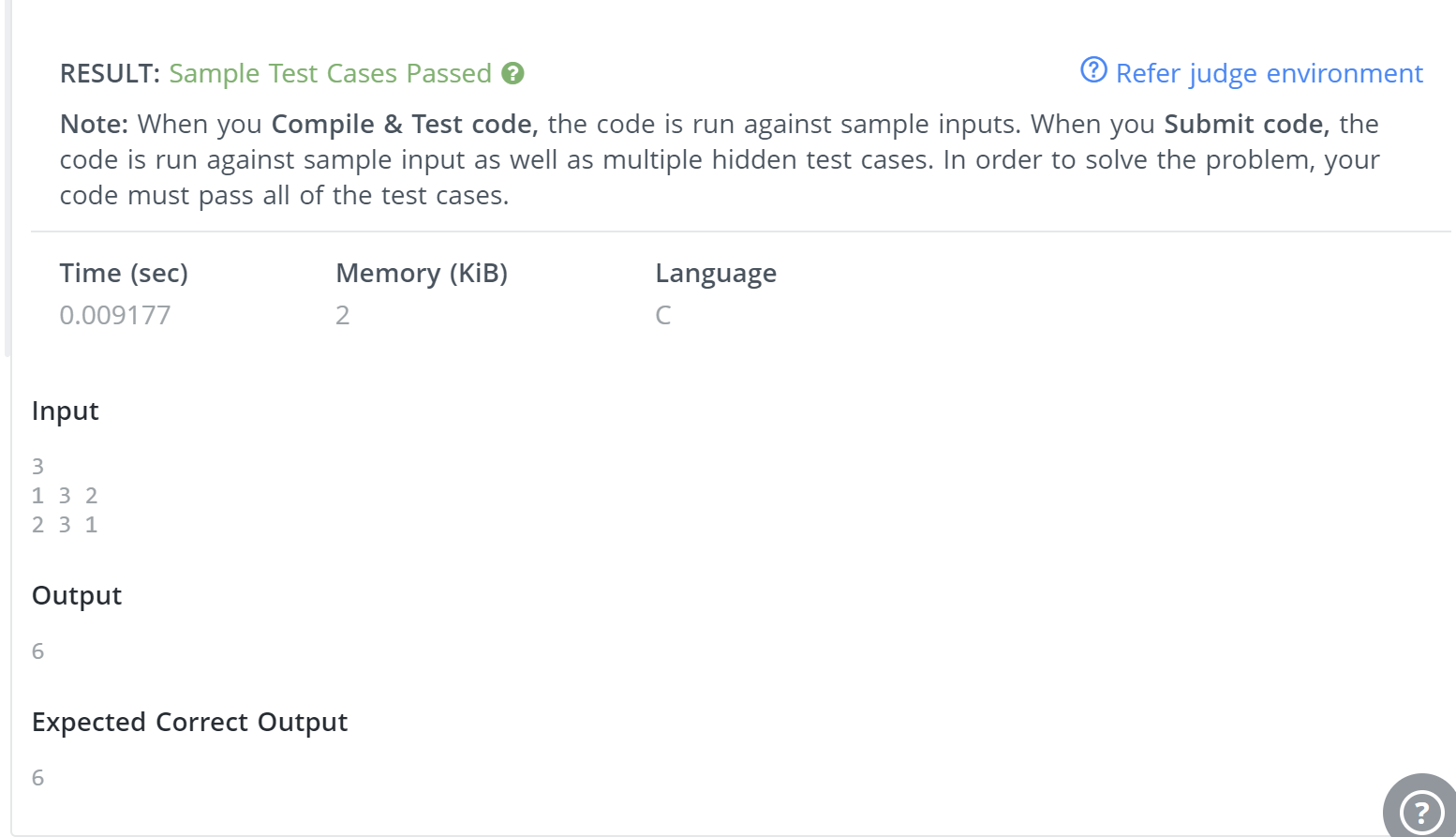
    }

    printf("%d\n", totalTime);

    return 0;

}

**Output:**

****

**Problem:**

Your task is to construct a tower in N days by following these conditions:

* Every day you are provided with one disk of distinct size.
* The disk with larger sizes should be placed at the bottom of the tower.
* The disk with smaller sizes should be placed at the top of the tower.

The order in which tower must be constructed is as follows:

* You cannot put a new disk on the top of the tower until all the larger disks that are given to you get placed.

Print N lines denoting the disk sizes that can be put on the tower on the ith day.

**Program:**

#include<stdio.h>

int main() {

    int diskSize, temp[100001] = {0};

    scanf("%d", &diskSize);

int minVal = diskSize, currentSize = diskSize;

    int currentInput;

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

        scanf("%d", &currentInput);

        temp[currentInput] = currentInput;

        if(currentInput == minVal) {

            while(temp[currentSize]) {

                printf("%d ", currentSize);

                currentSize--;

            }

            minVal = currentSize;

            printf("\n");

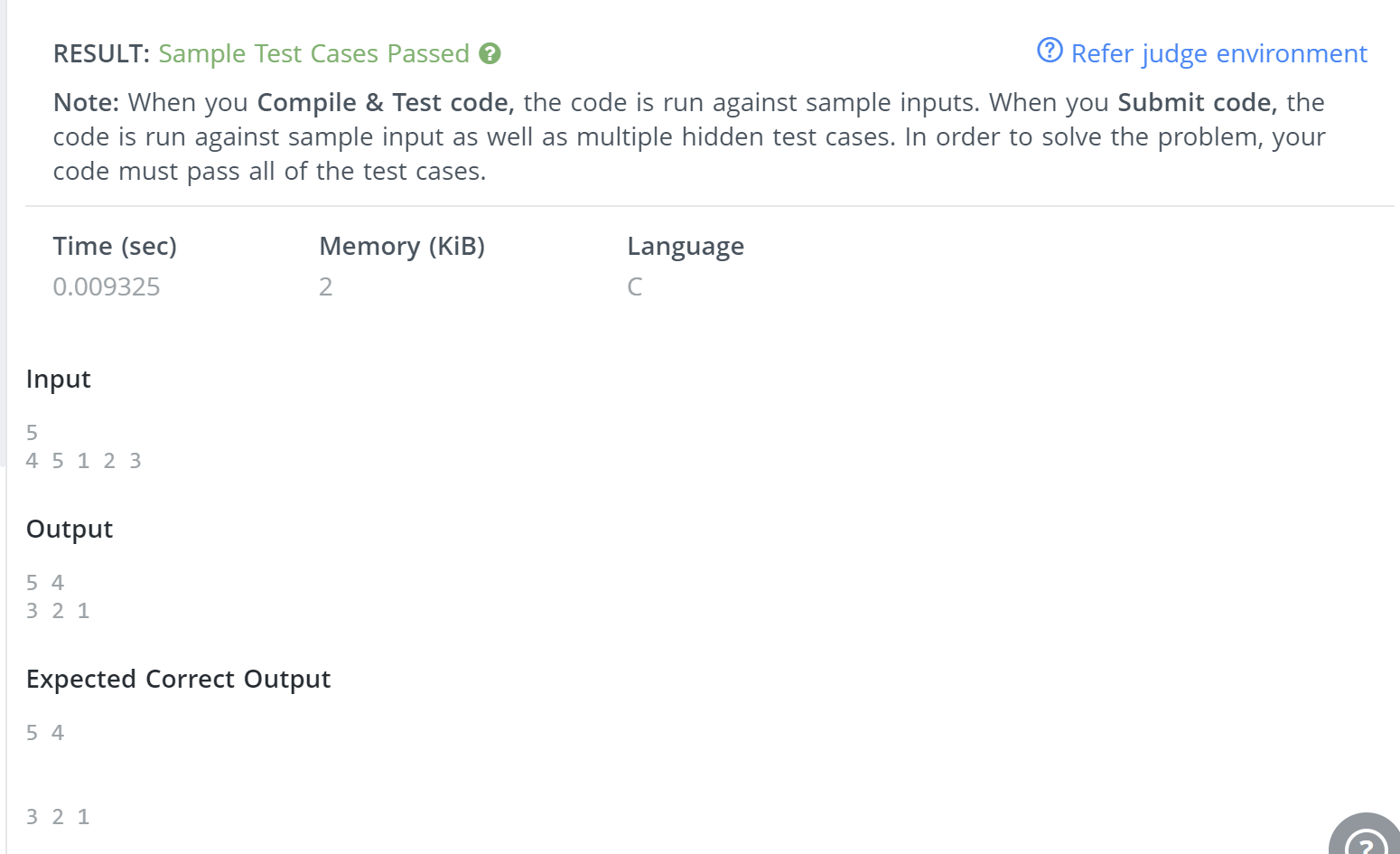
        }

    }

    return 0;

}

**Output:**

****