TITLE PAGE:

PROJECT TITLE:

ImplementingDeque using memory efficient XOR Doubly Linked List

TEAM MEMEBERS DETAILS:

1)19H51A05K3 - G. GANENDHAR

2)19H51A0522 - R. SATHYA RAJ

3)19H51A0420 - N. RISHITH RAO

4)19H51A0430 - ENOSH PAUL

MENTOR(S): K. Ashutosh Sir

SEMESTER/YEAR: IVth Semester/ IInd Year

ABSTRACT

We are doing a project on implementation of DEQUE using a XOR doubly linked list. The 10 functions given below code are performed here with time complexity of O(1). We are also going through graph plotting, constraints and detailed description of this project. By using XOR linked list, it reduces the space (i.e., memory efficient). Memory efficient double linked list has only 1 pointer to traverse the list back and front. The implementation is based only on pointer difference. It uses bitwise XOR operator to store the front and rear pointer addresses. Instead of storing actual memory address, every node stores the XOR address of previous node and next node. But the main disadvantages are debugging tools cannot follow XOR chain, previous node address must be remembered to get next nodes and pointers are not defined accurately. The conclusion we get is, it is generally used for applications that requires adding/removing elements from both the ends.

where the user can perform all the operations which a DEQUE performs

The operations are:

1. push\_Front()

2. push\_Back()

3. get\_front ()

4. get\_back()

5. get\_sec\_front()

6. get\_sec\_back()

7. Size()

8. Empty()

**Table of Contents:**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **CONTENTS** | **Page No** |
| **1**  1.1  1.2  1.3  1.3.1  1.3.2  1.3.3  1.4  1.5  1.6 | Project Description:  Purpose of the Project  Goals/Requirements  Methodology  Alternative Approaches  Current Approach Chosen  Detailed Description of Current Approach  Measurements to be done  Constraints  Assumptions |  |
| **2** | Code |  |
| **3**  3.1  3.2  3.3  3.4  3.5 | Test Plans:  Approach  Features to be Tested/Not Tested  Pass/Fail Criteria  List of Test Cases  Test programs Listing |  |
| **4**  4.1  4.2  4.3 | Measurement & Analysis:  Theoretical time Complexity Analysis of each operation  Tabular data for measured Time-taken vs N  Graph Plotting |  |
| **5** | Conclusions |  |
| **6** | Future Enhancements |  |
| **9** | Difficulties Faced |  |
| **10** | References links |  |

PROJECT DESCRIPTION:

Purpose of the project:

The main purpose of our project is to create a Data Structure using memory efficient Doubly Linked List (DLL), which can work as Dequeue. A Dequeue is a linear data structure where operations can be done on both sides. This project ensures that the elements in the queue can be added to or removed from either the front (head) or back (tail) and to get the memory efficient code using XOR doubly linked list.

Goals/Requirements:

The main goal of this project is to write a program in C++. We need to generate a Dequeue using memory efficient doubly linked list using XOR function.

The 10 functions are:

1. Push\_ back – An element is added to the end of the list.
2. Push\_ front – Element is added at the front of the list.
3. Pop\_ back – Element is deleted at the end of the list.
4. Pop\_ front – An element is deleted from the front of the list.
5. get\_ back – Function used to get the element from the back side of the list.
6. get\_ front – This function used to get the element from the front of the list.
7. get\_ second back – Function used to get the second back element from the list.
8. get\_ second front – Function used to get the second front element from the given list.
9. Size – The size function is used to get the size of the list.
10. Empty – Function used to display whether the given list is empty or not.

Methodology:

Alternative approach:

Array: An array can be used to implement Deque, but here the time complexity is not the optimal.

We can also use other data structures also to implement a Deque. We can use:

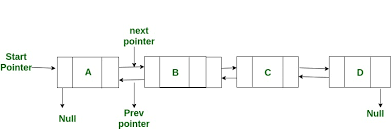
1. Single Linked List
2. Double Linked List
3. Two Queues
4. Two Stacks
5. Circular Array
6. Vectors

**Current Approach Chosen:**

DEQUE using XOR Doubly Linked List.

**Detailed Description of Current Approach:**

DEQUE is called as Memory efficient doubly linked list. It has one pointer to traverse the list back and front. The implementation is based on pointer difference. It uses bitwise XOR operator to store the front and rear pointer addresses. Instead of storing actual memory address, every node stores the XOR address of previous node and next node. XOR Linked Lists requires same time as that of the arrays would require.



**MEMORY EFFICIENT DOUBLY LINKED LIST**

**Measurements to be done:**

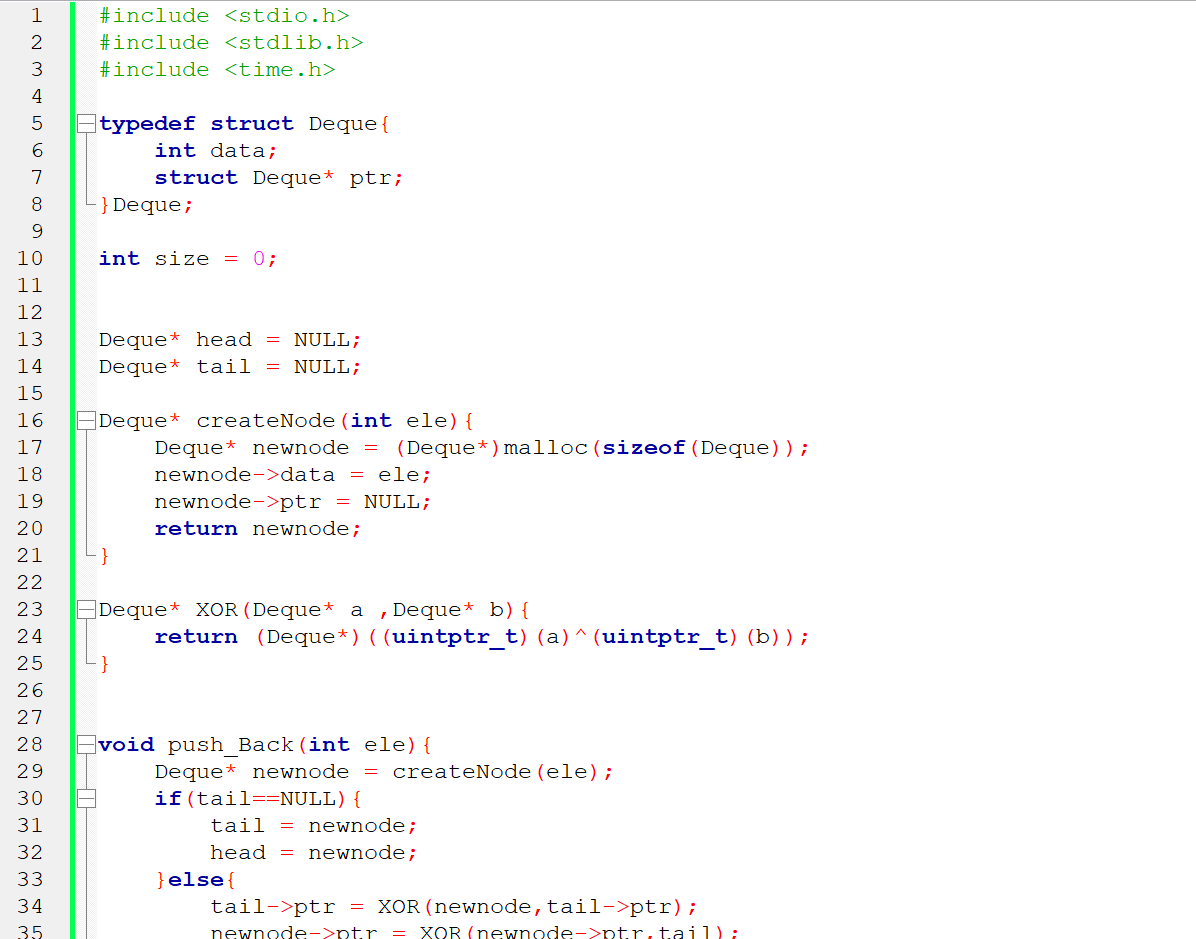
We have to ensure that the time complexity of each operation performed here must be constant O(1).This can be measured by using std::chrono library.Here, we need to take two time points (one at the first and other at the last) of the program. Then, by calculating the difference, we get the execution time of the program. We calculate the execution time of the insertion functions (i.e, push\_front() and push\_back()) by passing different inputs and plot a graph for no. of inputs vs execution time

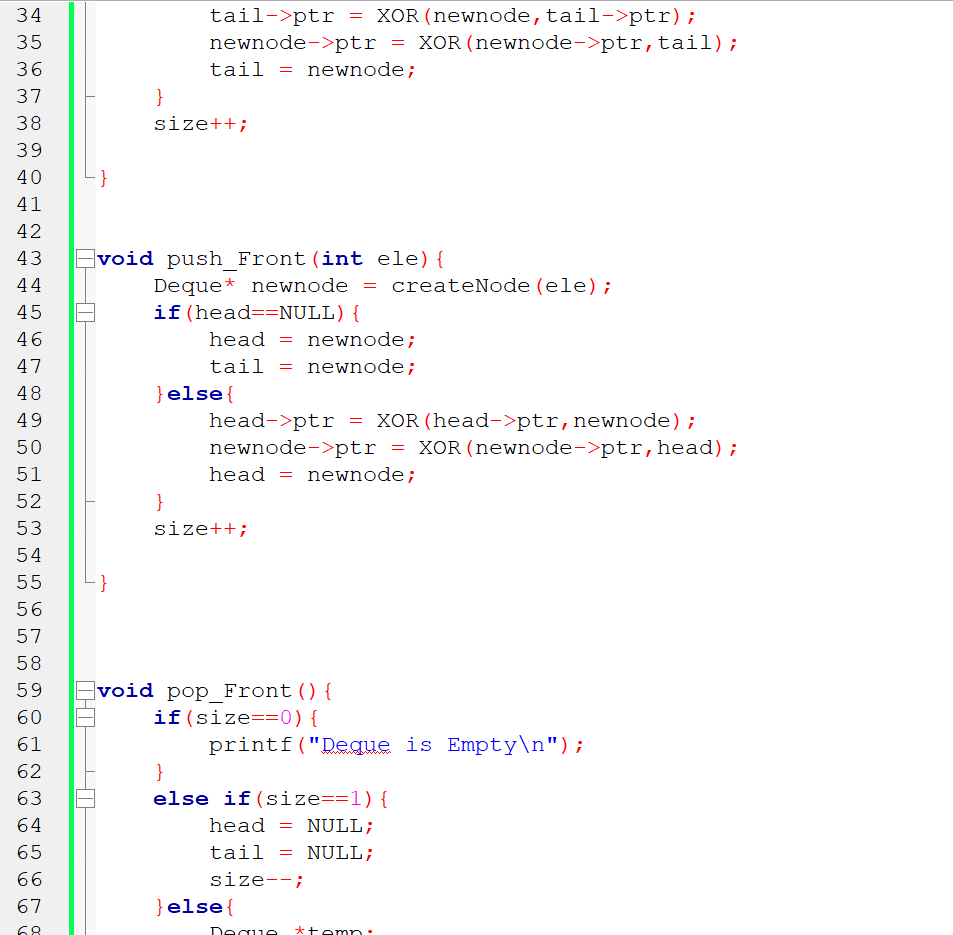
**Constraints:**

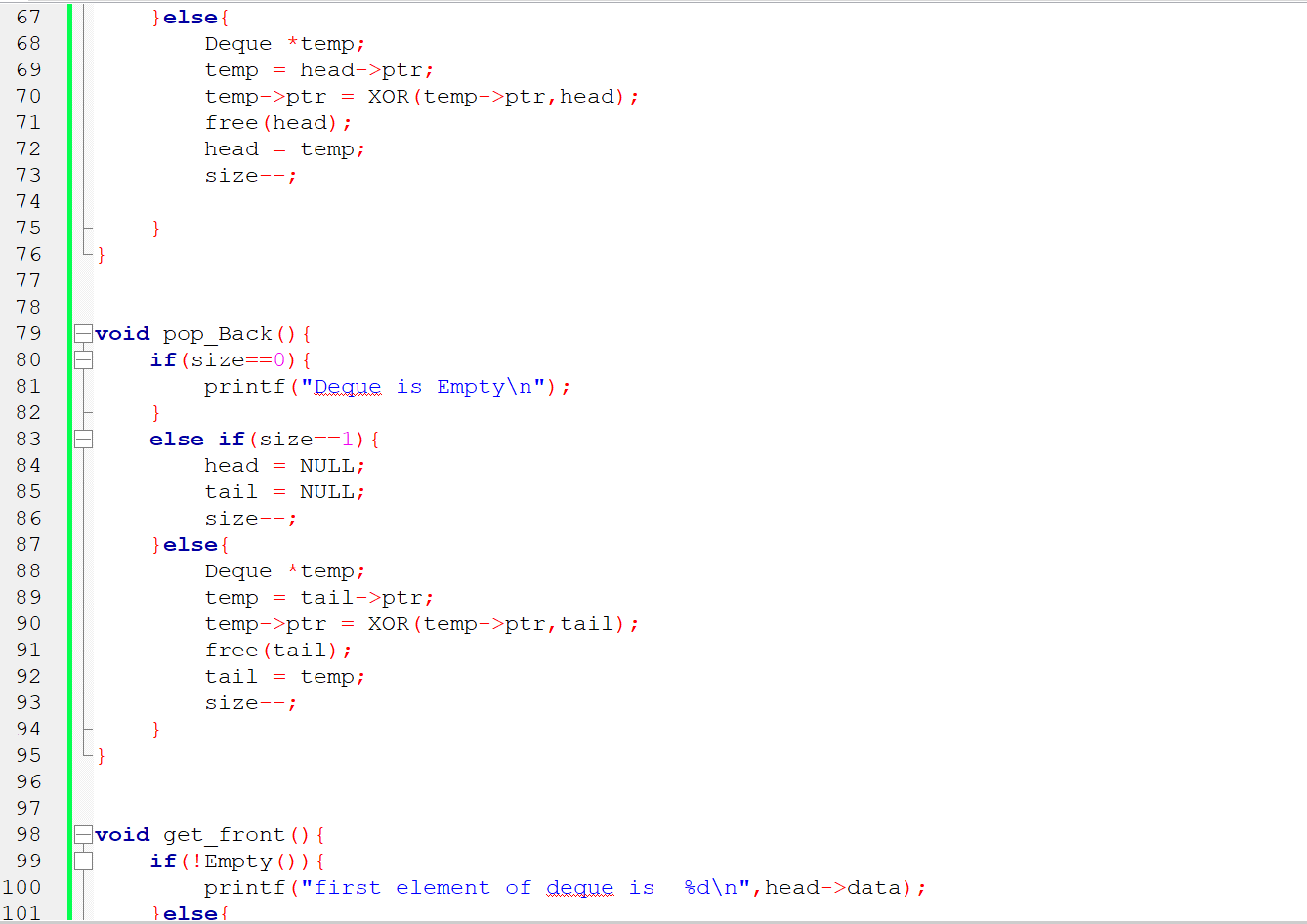
* A single Node cannot contain two pointers.
* Large integers values are not allowed to insert into the DEQUE.

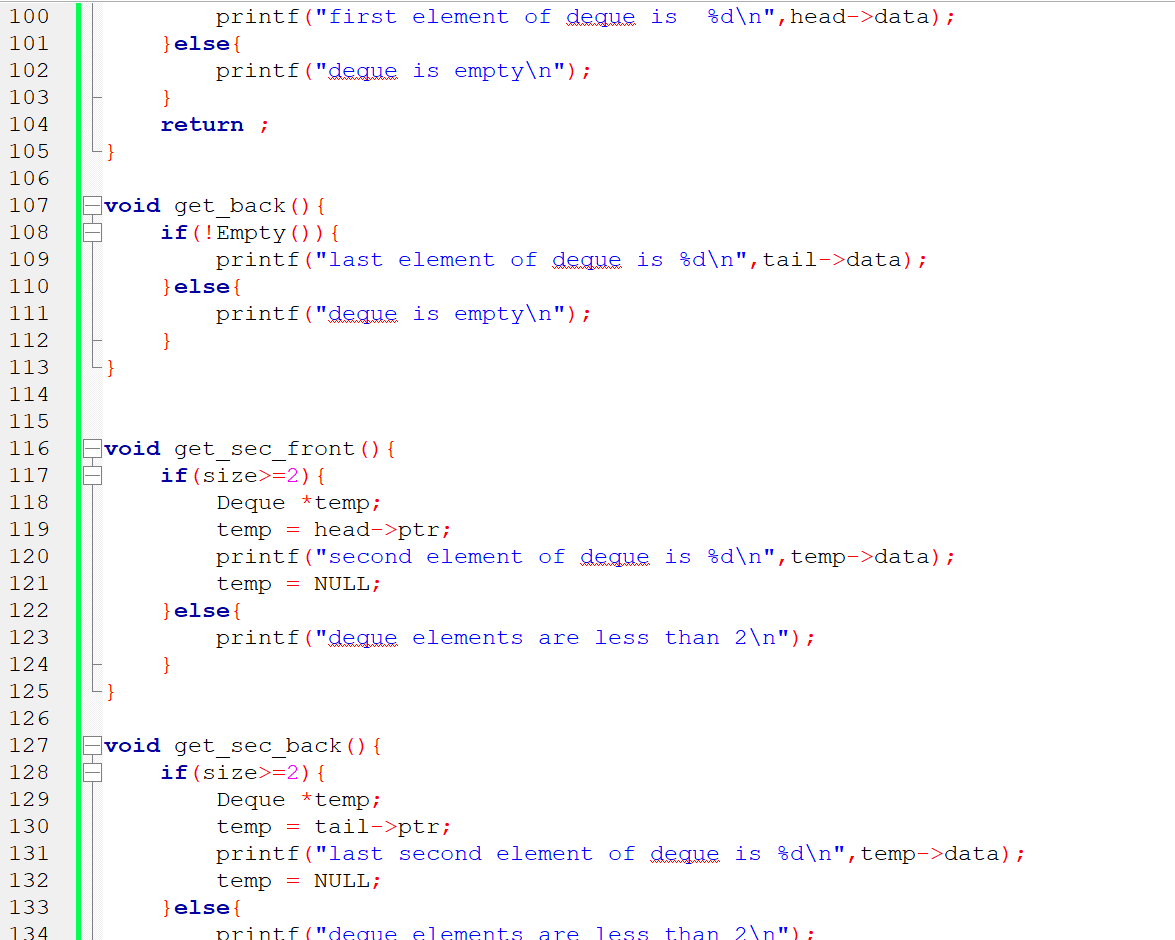
**Assumptions:**

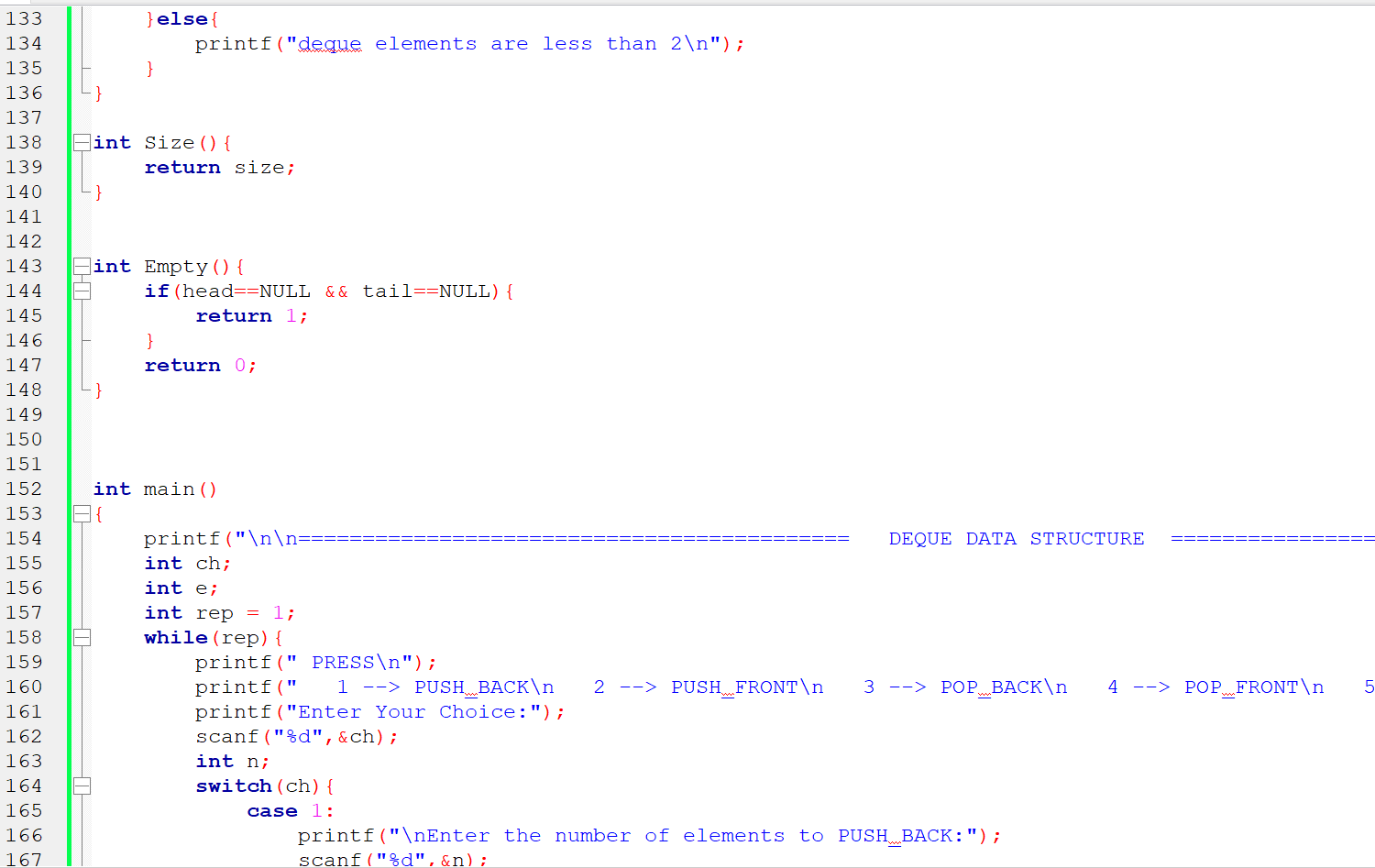
We assumed that a Deque will allow only the inputs in the range of integers only. It does not allow the long integers, strings, characters and user defined data types like structures and unions.

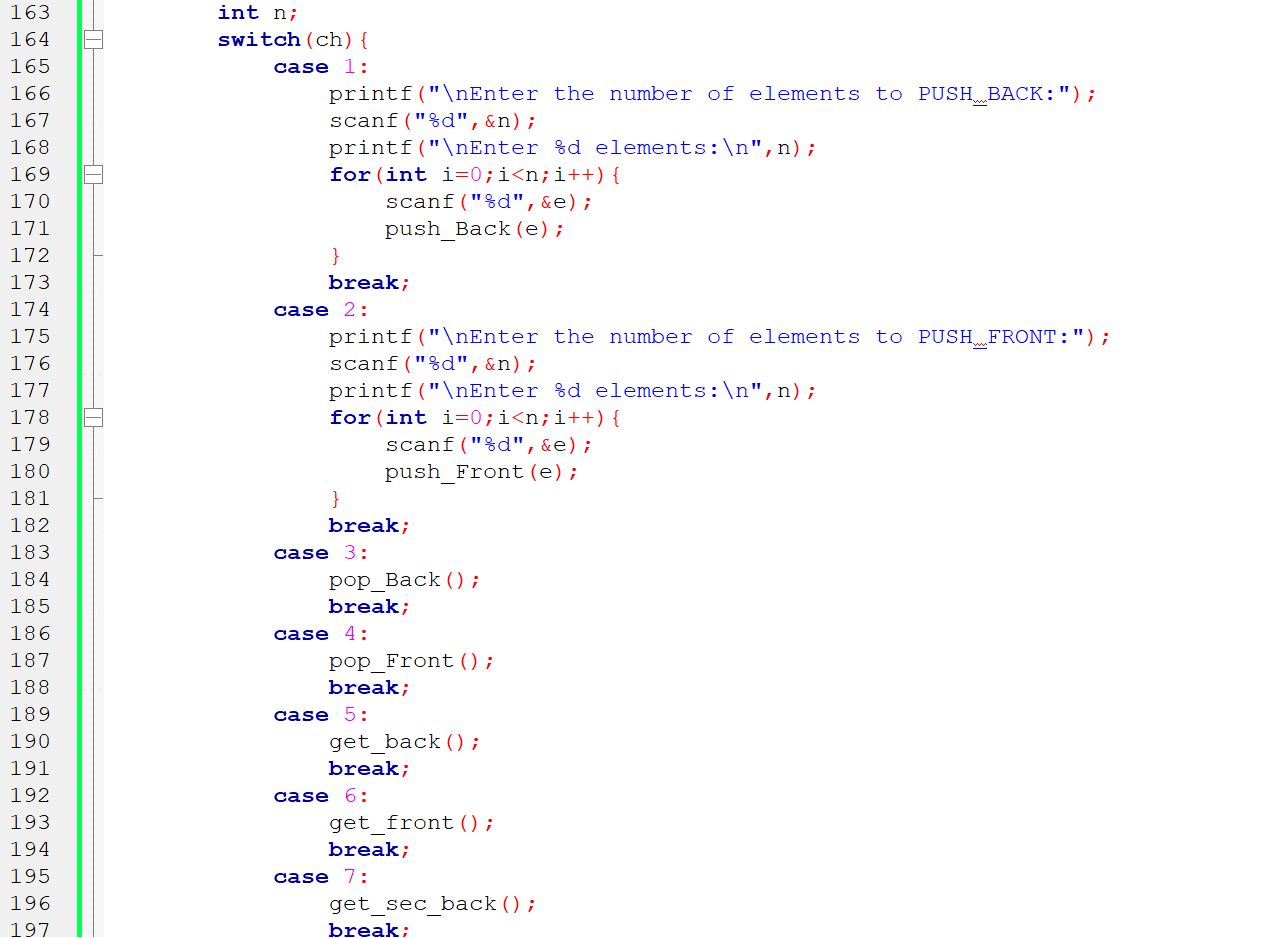
CODE: 





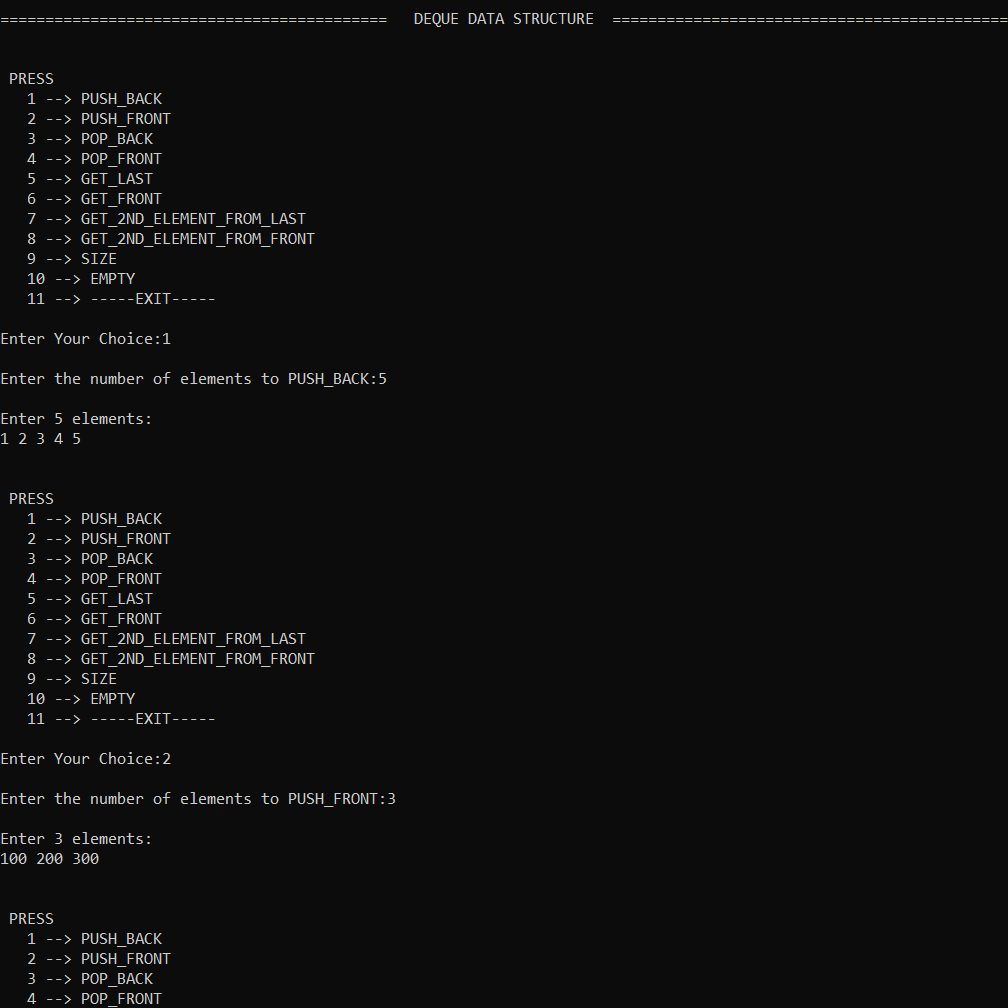


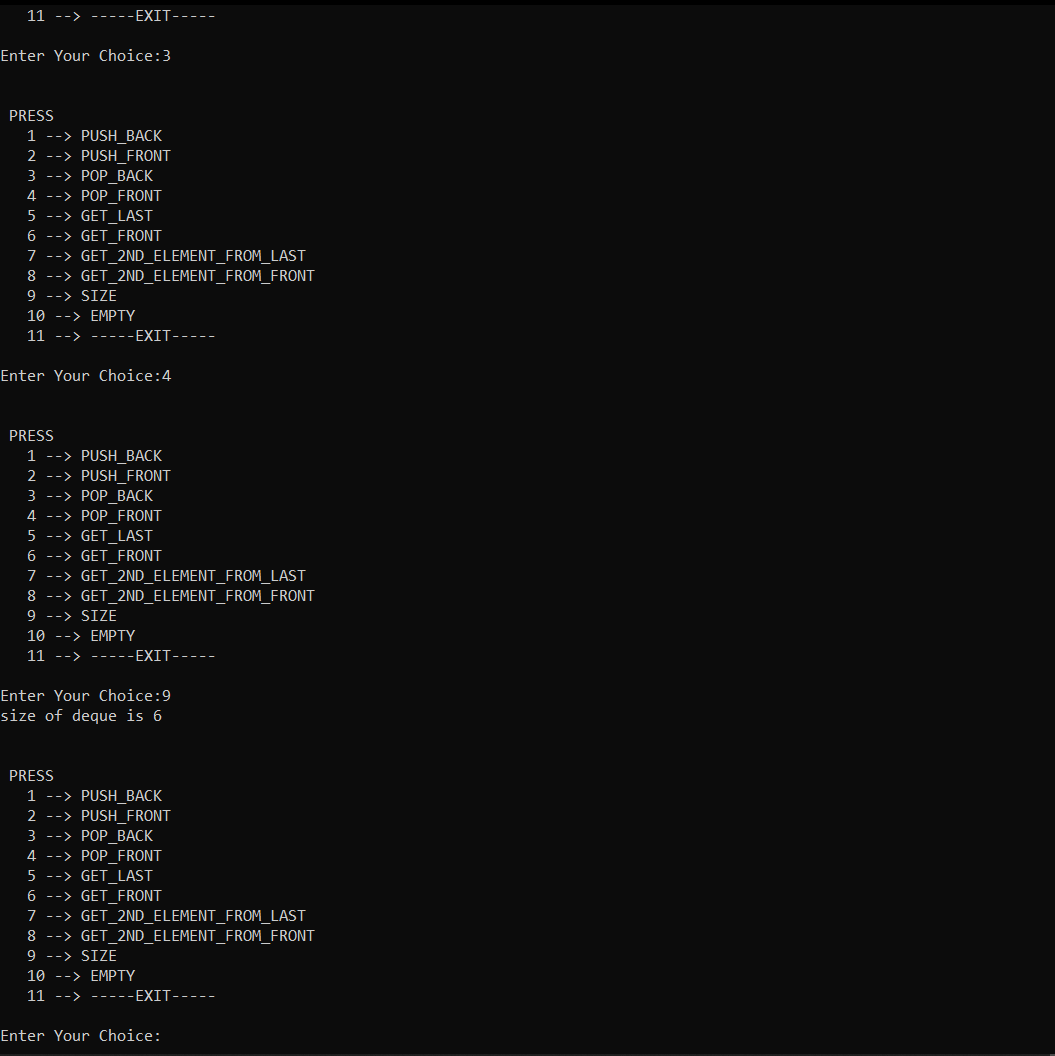
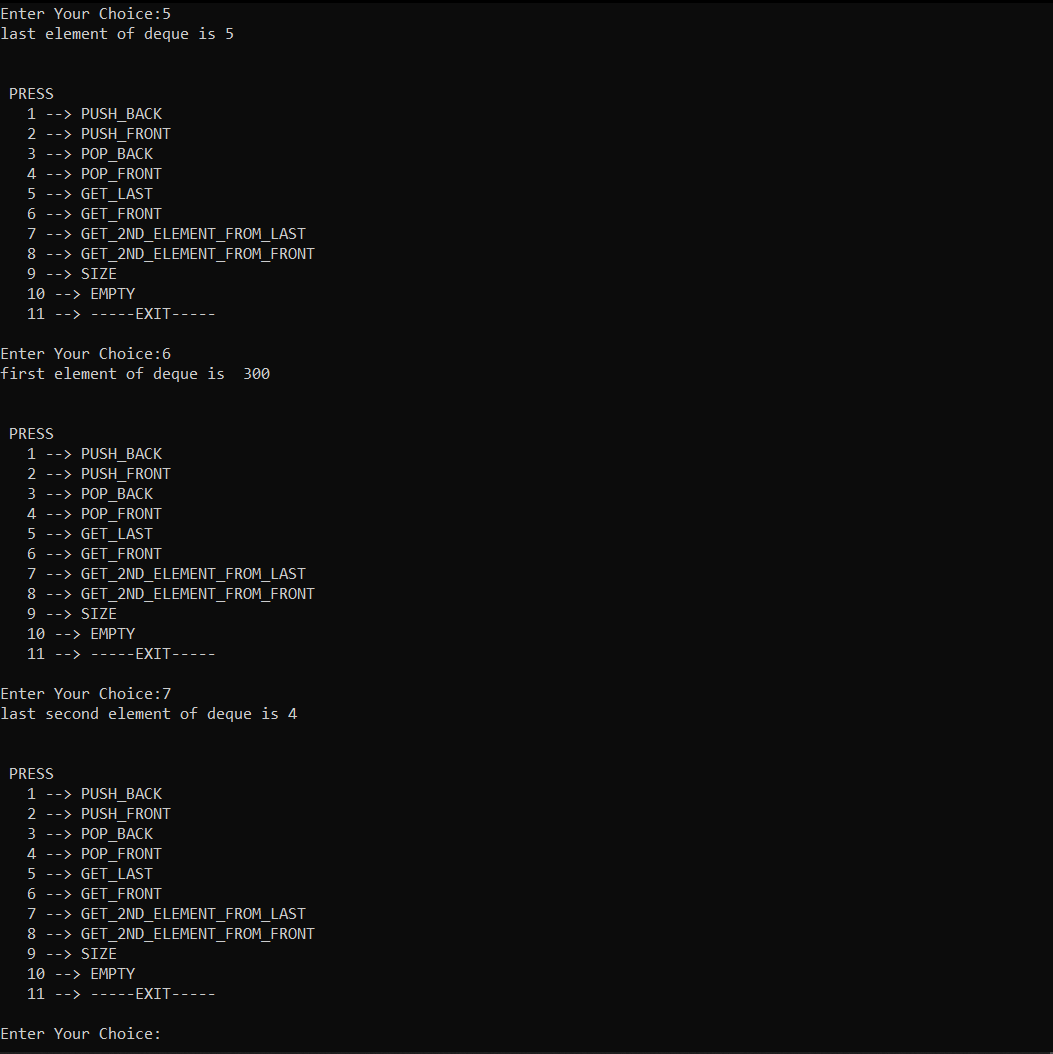
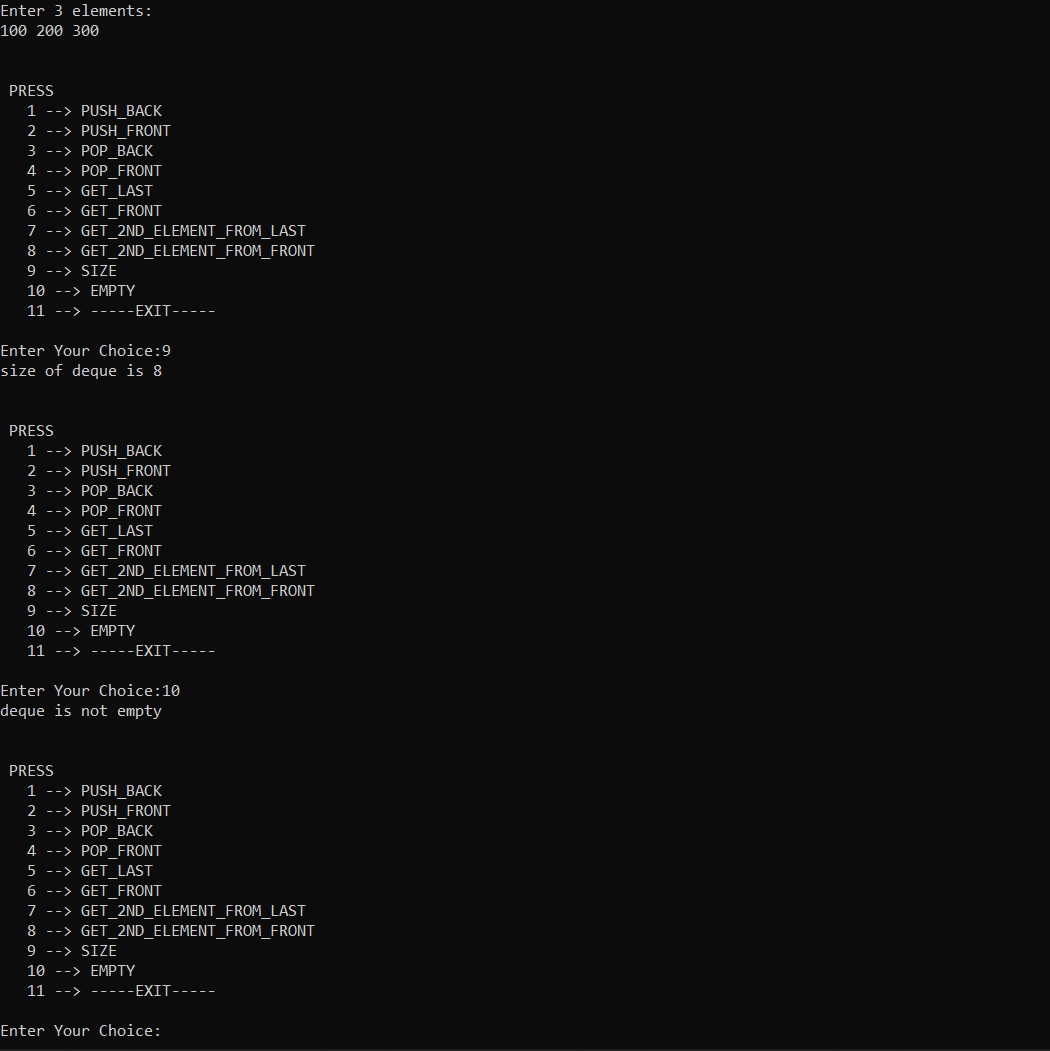






**Output Interface:**





**Formal Deque Code Link:**

[**https://ideone.com/oO0Mg1**](https://ideone.com/oO0Mg1)

**Test Plans:**

**Approach:**

**Features to be tested/Not tested:**

**Pass/Fail Criteria:**

**List of Test cases:**

**Test Programs listing:**

**Measurement and Analysis:**

**Theoretical Time Complexing Analysis for each Operation:**

**Tabular Data:**

1. **PUSH FRONT OPERATION:**

|  |  |
| --- | --- |
| Input N | Time taken (milliseconds) |
| 100000 | 0 |
| 1000000 | 54 |
| 2500000 | 128 |
| 5000000 | 245 |
| 7500000 | 360 |
| 10000000 | 488 |
| 25000000 | 1199 |

2) **PUSH BACK OPERATION:**

|  |  |
| --- | --- |
| Input N | Time taken (milliseconds) |
| 100000 | 0 |
| 1000000 | 63 |
| 2500000 | 125 |
| 5000000 | 238 |
| 7500000 | 368 |
| 10000000 | 500 |
| 25000000 | 1202 |

3) **POP FRONT OPERATION:**

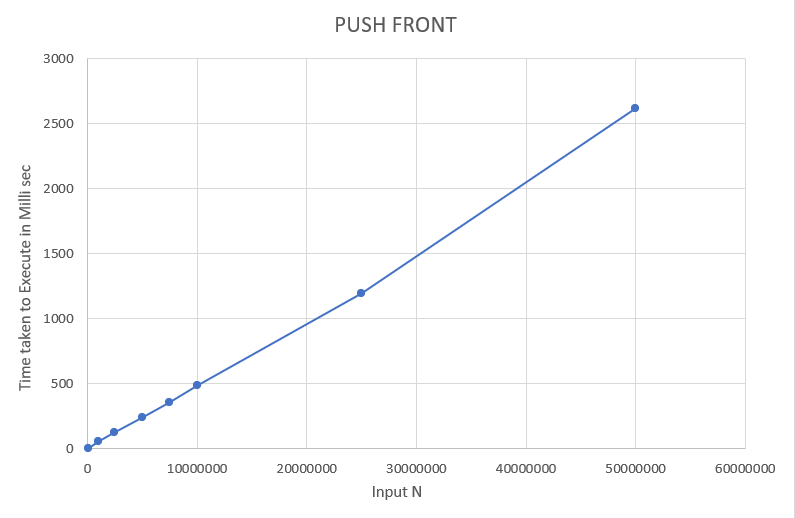
|  |  |
| --- | --- |
| Input N | Time taken (milliseconds) |
| 100000 | 0 |
| 1000000 | 15 |
| 2500000 | 77 |
| 5000000 | 171 |
| 7500000 | 255 |
| 10000000 | 327 |
| 25000000 | 880 |

4) **POP BACK OPERATION:**

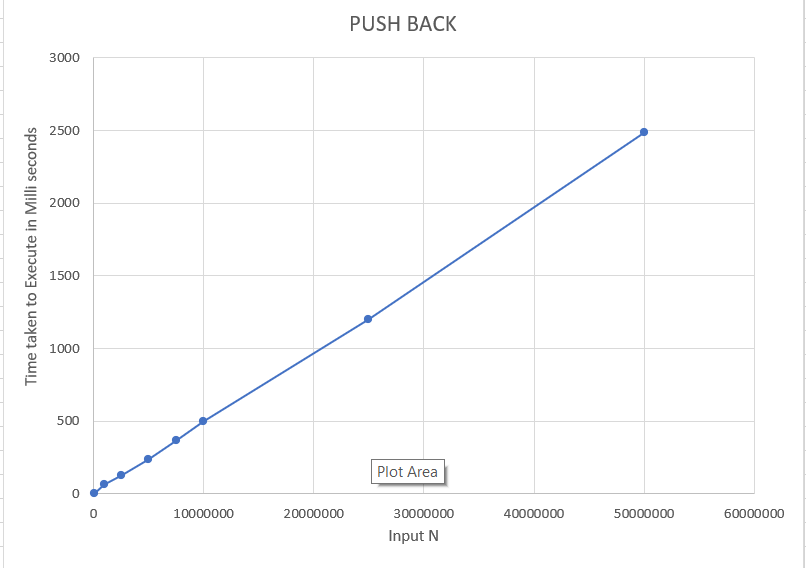
|  |  |
| --- | --- |
| Input N | Time taken (milliseconds) |
| 100000 | 0 |
| 1000000 | 22 |
| 2500000 | 80 |
| 5000000 | 163 |
| 7500000 | 245 |
| 10000000 | 337 |
| 25000000 | 878 |

**Graph Plotting:**

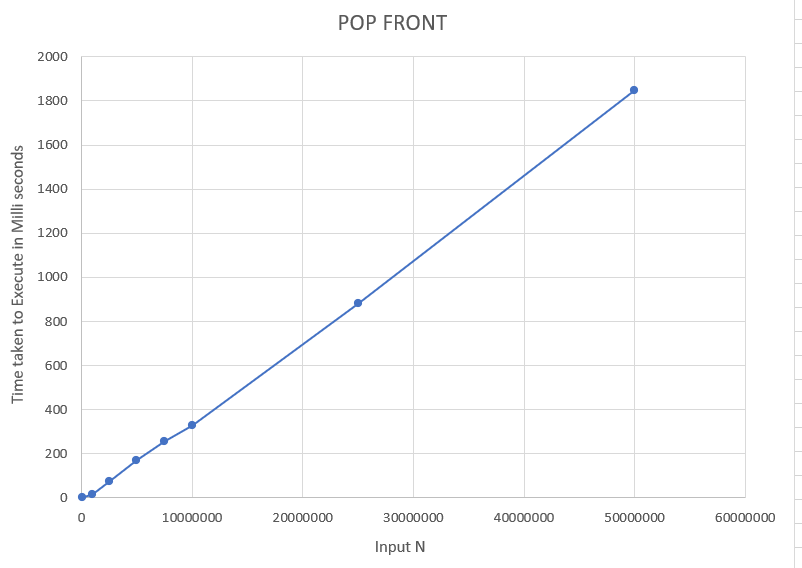
**TIME TAKEN (MILLI SECONDS) VS INPUT SIZE FOR PUSH FRONT OPERATION:**



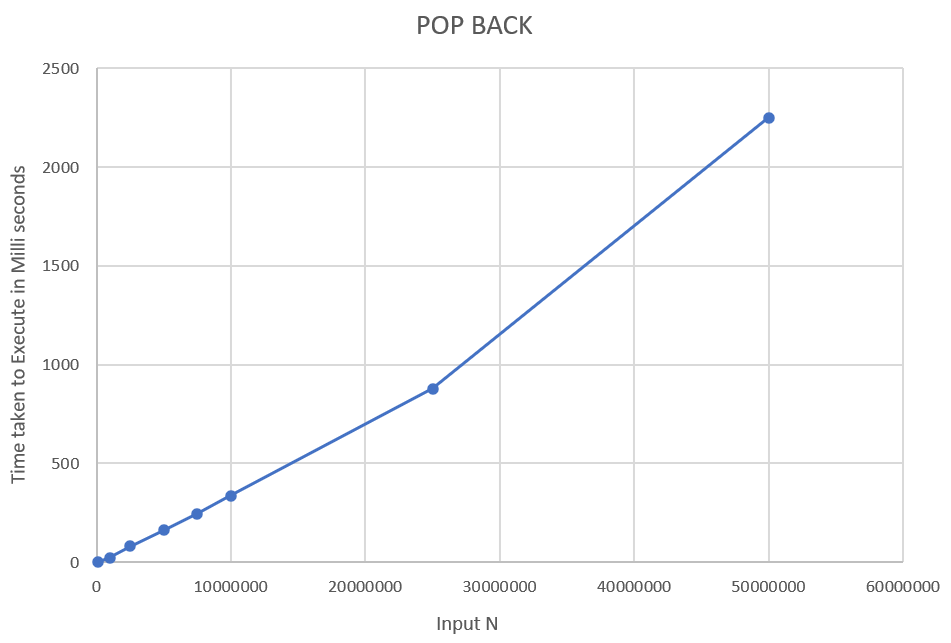
**TIME TAKEN (MILLI SECONDS) VS INPUT SIZE FOR PUSH BACK OPERATION:**



**TIME TAKEN (MILLI SECONDS) VS INPUT SIZE FOR POP FRONT OPERATION:**



**TIME TAKEN (MILLI SECONDS) VS INPUT SIZE FOR POP BACK OPERATION:**



**Conclusion:**

The conclusion is that it is used for applications that require adding/removing elements from both the ends and also in the scheduling of processors in multi-processor systems

**Future Enhancements:**

The following are the changes to be made:

* XOR linked list is not supported by several languages such as Java where conversion between pointers and integers is undefined.
* The price for the decrease in memory usage is an increase in code complexity, making maintenance more expensive.
* XOR linked lists do not provide some of the important advantages of doubly linked lists, such as the ability to delete a node from the list knowing only its address or the ability to insert a new node before or after an existing node when knowing only the address of the existing node.

**Difficulties Faced:**

**Reference Links:**