DEQUE USING MEMORY EFFICIENT XOR DOUBLY LINKED LIST

TEAM DETAILS:

Team size>>5 Year/Semester>>II/II

S.NO.	ROLL NO.	NAME OF THE STUDENT	DEPT
1.	19H51A0473	K.Naga Sarvani	ECE
2.	19H51A04B0	P.Ruchitha	ECE
3.	19H51A0565	B.Vishnu Varshitha	CSE
4.	19H51A0567	D.Sai Prasanna	CSE
5.	19H51A0573	K.V.S.S.S.R.H Sri Harsha	CSE

MENTOR: Mr. Ashutosh K Akhouri

ABSTRACT:

Deque is a type of queue data structure. Double ended queue is abbreviated as deque. In general, deque uses two pointers and performs several operations like push front, push back, delete, sizeof operations with a time complexity of O(n).

The main aim of our project is to implement this deque data structure using a single pointer with the help of XOR linked lists and maintain a constant time complexity i.e O (1). We will test the code for sample cases and also consider various test cases to check whether it's printing the expected out or not. Then, we are plotting the graph for the executed operations. The graph is plotted by considering the time taken and input size. To improve the memory efficiency and time complexity we are implementing deque using a single pointer with the help of XOR linked lists.

college of engineering and technology

TABLE OF CONTENTS

1)PROJECT DESCRIPTION:

<<1.1>> Purpose of the project:

Double-ended queue performs various operation such as push_back (adding element at the end), push_front (adding element at the beginning), pop_back (delete last element), pop_front (delete first element), get_front (getting the front element), get_back (getting the last element), get_2nd_front, get_2nd_back, size (finding size of deque), empty (checking whether the deque is empty or not). The purpose of this project is to implement these operations in constant time complexity i.e, O (1) using a single pointer per node.

<<1.2>>Goals:

- To make the run time complexity of the push and pop functions to constant time.
- To decrease the space complexity by using a single pointer per node using XOR linked list.
- To check working of code for various test cases
- Calculating time complexity of push and pop functions by executing them and plotting graphs.

<<1.3>>Methodology:

1.3.1. Alternative approaches

The possible alternative approaches to implement deque data structure are listed below:

i) A conventional doubly linked list

Here we use two pointers per node for accessing the next node and previous node.so it can be converted into a double ended queue by maintaining a pointer at front(head) and back(tail) and traversing.

Time complexity for insertion: O (1)
Time complexity for deletion: O (1)
Time complexity for accessing: O (n)

ii) Single_Linked List

Similar to double linked lists but we need 2 pointers at head and one for tail for travelling so as to keep track of previous address since we only have one address i.e, next address at every node.

Time complexity for insertion: O (1)
Time complexity for deletion: O (1)(At front)/O(n)(back)
Time complexity for accessing: O (n)

iii) Two Queues

Similar to a queue.The two queues are combined so as to make two ends.

Time complexity for insertion: O (1)
Time complexity for deletion: O (1)
Time complexity for accessing: O (n)

iv) Two Stacks

one stack as the head of the queue, and one as the tail. The enqueue operations would just be a push to the respective stack, and the dequeue operations would just be a pop on the respective stack.However, if the stack we want to dequeue from is empty, we'd have to pop each element from the other stack and push it back to the stack we want to dequeue from, and then dequeue the last one. That's not really good performance, so the overall performance of this implementation strongly depends on the workload.

Time complexity for insertion: O (1)
Time complexity for deletion: O (1)
Time complexity for accessing: O (n)

v) Circular Array

To implement Deque employing a circular array we should track 2 pointers front and rear within the array, all the operations are on these 2 pointers.If (rear + 1) % n equals to the front then the Deque is full, else it is not. Here n is the maximum size of Deque.

Time complexity for insertion: O (1)
Time complexity for deletion: O (1)
Time complexity for accessing: O (n)

vi) Vectors/Arrays

In cpp the dequeue is implemented similarly by individually allocating fixed-size arrays, with additional storing.

Time complexity for insertion: O (1)
Time complexity for deletion: O (1)
Time complexity for accessing: O (n)

1.3.2. Current approach chosen

The chosen approach is implementing deque using XOR linked lists.To make a memory efficient linked list by using one pointer at a node.

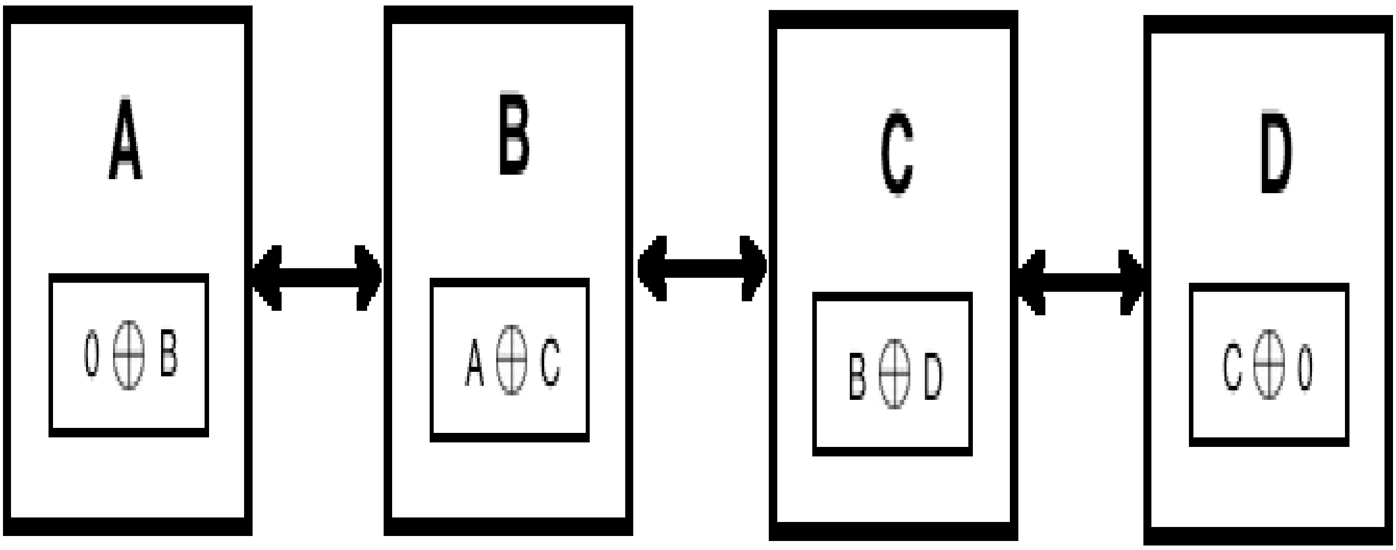
1.3.3. Detailed description of current approach

The current approach is a memory-efficient version of Doubly Linked List that can be created using only one space for the address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.

ILLUSTRATION:

Node A:
npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B:
npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C



Node C:
npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D

Node D:
npx = add(C) XOR 0 // bitwise XOR of address of C and 0
npx(C) XOR add(B)
=> (add(B) XOR add(D)) XOR add(B) // npx(C) = add(B) XOR add(D)
=> add(B) XOR add(D) XOR add(B) // a^b = b^a and (a^b)^c = a^(b^c)
=> add(D) XOR 0 // a^a = 0
=> add(D) // a^0 = a

So by using the current approach we save the storage by saving the one pointer space per node. This might not make much difference for smaller inputs,But for a larger input it will make a considerable change.

<<1.4>>Measurements to be done:

The time complexity of each operation must be O(1). We need to measure theoretical time complexity for push _back / push_ front and get_ front/get_ back. Graph must be plotted between time taken and input..

<<1.5>>Constraints:

The below mentioned points are the constraints of our project:
$$$ Only integers to be taken as input in the dequeue ranges between (-2147483648,2147483647).
$$$ The maximum number of elements in the dequeue is 10^7.

<<1.6>>Assumptions:

- Using integers
- Taking a single instance of input at a time
- Functions used to measure time
- xor(a,NULL)=a

<u>2)CODE</u>-- Link for source code: https://ideone.com/XrP1bv

```
bz_mini_project1 > bzdequeue.cpp > ...
#include <bits/stdc++.h>
using namespace std;

int c; //variable to count

struct node
{
    int data;
    struct node *npx;
};
struct node *head;
struct node *tail;

struct node *XOR(struct node *a, struct node *b)
{
    return reinterpret_cast<struct node *>(
        reinterpret_cast<uintptr_t>(a) ^ reinterpret_cast<uintptr_t>(b));
}

void push_front(int data)
{
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = data;
    new_node->npx = head; //Since head xor null=head
    if (head != NULL)
    {
        (head)->npx = XOR(new_node, (head)->npx);
    }
    else
    {
        tail = new_node;
    }
    head = new_node;
    c = c + 1;
}

```

```
bz_mini_project1 > bzdequeue.cpp > ...
void pop_front()
{
    if (head == NULL)
    {
        cout << "DELETION NOT POSSIBLE DEQUEUE IS EMPTY\n";
    }
    else
    {
        struct node *temp = head;
        head = (head)->npx; //the adress part of first node contains the adress of next node since previous adress is null
        if (head != NULL)
            (head)->npx = XOR(temp, (head)->npx);
        else
            tail = NULL;
        cout << temp->data << " deleted\n";
        c = c - 1;
    }
}

void pop_back()
{
    if (tail == NULL)
    {
        cout << "DELETION NOT POSSIBLE DEQUEUE IS EMPTY\n";
    }
    else
    {
        struct node *temp = tail;
        tail = (tail)->npx; //the adress part of last node contains the adress of previous node since the next adress is null
        if (tail != NULL)
            (tail)->npx = XOR(temp, (tail)->npx);
        else
            head = NULL;
        c = c - 1;
        cout << temp->data << " deleted\n";
    }
}
```

bz_mini_project1 > bzdequeue.cpp > ...

```
void push_back(int data)
{
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = data;
    new_node->npx = tail;
    if (tail != NULL)
    {
        (tail)->npx = XOR(new_node, (tail)->npx);
    }
    else
    {
        head = new_node;
    }
    tail = new_node;
    c = c + 1;
}

void printList()
{
    struct node *curr = head;
    struct node *prev = NULL;
    struct node *next;
    if (head == NULL)
    {
        cout << "Dequeue is empty";
    }

    while (curr != NULL)
    {
        cout << curr->data << " ";
        next = XOR(prev, curr->npx);
        prev = curr;
        curr = next;
    }
    cout << "\n";
}
```

```
int size()
{
    return c;
}

bool IsEmpty()
{
    if (c == 0)
        return true;
    return false;
}

struct node *get_front()
{
    if (head == NULL)
        cout << "Dequeue is empty so no front\n";
    return head;
}

struct node *get_2front()
{
    if (head == NULL)
    {
        cout << "Dequeue is empty so no 2nd front\n";
        return head;
    }
    if (head->npx == NULL)
    {
        cout << "Dequeue has only one element so no 2nd front\n";
    }
    return head->npx;
}
```

```
struct node *get_back()
{
    if (tail == NULL)
    {
        cout << "Dequeue is empty so no back\n";
    }
    return tail;
}

struct node *get_2back()
{
    if (tail == NULL)
    {
        cout << "Dequeue is empty so no 2nd back\n";
        return tail;
    }

    if (tail->npx == NULL)
    {
        cout << "Dequeue has only one element so no 2nd back\n";
    }
    return tail->npx;
}
```

```
int main()
{
    int a;
    cout << "MENU\n1.PUSH FRONT\n2.PUSH BACK\n3.POP FRONT\n4.POP BACK\n5.SIZE\n6.IS_EMPTY\n";
    cout<<"7.GET FRONT\n8.GET 2ndFRONT\n9.GET BACK\n10.GET 2ndBACK\n11.PRINT ALL ELEMENTS PRESENT\n0.EXIT\n";
    cout << "ENTER YOUR CHOICE:";
    cin >> a;
    while (true)
    {
        if (a == 1)
        {
            cout << "Enter data to be inserted\n";
            int data;
            cin >> data;
            push_front(data);
            cout << data << " Inserted\n";
        }
        else if (a == 2)
        {
            cout << "Enter data to be inserted\n";
            int data;
            cin >> data;
            push_back(data);
            cout << data << " Inserted\n";
        }
        else if (a == 3)
        {
            pop_front();
        }
        else if (a == 4)
        {
            pop_back();
        }
        else if (a == 5)
        {
            cout << "The size of Dequeue:" << size() << "\n";
        }
```

```
        else if (a == 6)
        {
            if (IsEmpty())
                cout << "Dequeue is empty\n";
            else
                cout << "Dequeue is not empty\n";
        }
        else if (a == 7)
        {
            struct node *temp = get_front();
            if (temp != NULL)
                cout << "The data in the front:" << temp->data << "\n";
        }
        else if (a == 8)
        {
            struct node *temp = get_2front();
            if (temp != NULL)
                cout << "The data in the 2ndfront:" << temp->data << "\n";
        }
        else if (a == 9)
        {
            struct node *temp = get_back();
            if (temp != NULL)
                cout << "The data in the back:" << temp->data << "\n";
        }
        else if (a == 10)
        {
            struct node *temp = get_2back();
            if (temp != NULL)
                cout << "The data in the 2ndback:" << temp->data << "\n";
        }
        else if (a == 11)
        {
            printList();
        }
```

```
        else if (a == 0)
        {
            cout << "\nThank you";
            exit(0);
        }
        else
        {
            cout << "RECHECK AND TRY AGAIN\n";
        }
        cout << "Enter your Choice:";
        cin >> a;
    }
    return 0;
}
```

3)TEST PLANS:

<<3.1>>Approach:

Using a single pointer approach and using bitwise XOR operation to save space for one address, so that every node stores the XOR of addresses of previous and next nodes.

<<3.2>>Features to be tested/not tested:

- Output with push_back function
- Output with push_front function
- Output with pop_back function
- Output with pop_front function
- Output with get_front function
- Output with get_back function
- Output with get_2nd_front function
- Output with get_2nd_back function
- Output with size function
- Output with empty function
- Output with printList function

<<3.3>>Pass/fail criteria:

Test Cases	Pass Criteria(Output)
1.When dequeue is empty	
size() pop_front() pop_back() print_list() get_front() get_2front() get_back() get_2back() IsEmpty	The size of Deque:0 Deletion not possible since dequeue is empty Deletion not possible since dequeue is empty Dequeue is empty so no printing Dequeue is empty so no front Dequeue is empty so no 2nd front Dequeue is empty so no back Dequeue is empty so no 2nd back Dequeue is empty
2. When deque has only one element(Data 1)	
push_front() push_back() pop_front() pop_back() printList() size() is empty() get_front() get_back() get_2front() get_2back ()	Data 1 inserted Data 1 inserted Data 1 deleted Data 1 deleted Data 1 The size of Dequeue is 1 Dequeue is not empty data 1 data 1 Dequeue has only one element so no 2nd front Dequeue has only one element so no 2nd back
3 .when deque has 2 elements(both by push_front-data1,data2)	
size() pop_front() pop_back() print_list() get_front() get_2front() get_back() get_2back() IsEmpty()	The size of Dequeue:2 data2 deleted data1 deleted data2 data1 The data in the front:data2 The data in the 2nd front:data1 The data in the back:data1 The data in the 2ndback:data2 Dequeue is not empty
4. when deque has 3 elements(both by push_front-data1,data2 one by push_back-data3)	
size() pop_front() pop_back() print_list() get_front() get_2front() get_back() get_2back() IsEmpty()	The size of Dequeue:3 data2 deleted data1 deleted data2 data1 data3 The data in the front:data2 The data in the 2nd front:data1 The data in the back:data3 The data in the 2ndback:data1 Dequeue is not empty.
5 . when deque has 2 elements(both by push_front-data1,data2) performing more than 1 function together	
size() Two times pop_front() Two times pop_back() Two times get_front() Two times push_front() For data1,data2 pop_front Size()	The size of Dequeue:2 data2 deleted data1 deleted data2 deleted data1 deleted The data in the front:data2 The data in the front:data2 data1 inserted data2 inserted data2 deleted The size of Dequeue:1

<<3.4>>List of test cases

1. When deque is empty
tc1:size()
tc2:pop_front()
tc3:pop_back()
tc4:print_list()
tc5:get_front()
tc6:get_2front()
tc7:get_back()
tc8:get_2back()
tc9:IsEmpty()
2. When deque has only one element
tc1:push_front()
tc2:Output with push_back()
tc3: Output with pop_front()
tc4:Output with pop_back()
tc5:Output with printList()
tc6:Output with size()
tc7:Output with is empty()
tc8:Output with get_front()
tc9:Output with get_back()
tc10:Output with get_2front()
tc11:Output with get_2back ()
3. When deque has 2 elements(both by push_front-data1,data2)
tc1:size()
tc2:pop_front()
tc3:pop_back()
tc4:print_list()
tc5:get_front()
tc6:get_2front()
tc7:get_back()
tc8:get_2back()
tc9:IsEmpty()
4 .when deque has 3 elements(both by push_front-data1,data2 one by push_back-data3)
tc1:size()
tc2:pop_front()
tc3:pop_back()
tc4:print_list()
tc5:get_front()
tc6:get_2front()
tc7:get_back()
tc8:get_2back()
tc9:IsEmpty()
5.when deque has 2 elements(both by push_front-data1,data2) performing more than 1 function together
tc1():size()
tc2():Two times pop_front()
tc3():Two times pop_back()
tc4():Two times get_front()
tc5():Two times push_front()   pop_front() Size()

<<3.5>>Test programs listing

1.https://ideone.com/OljR50

2.https://ideone.com/lF4XIm

3.https://ideone.com/BkDxo1

4.https://ideone.com/15pAss

5.https://ideone.com/9wrL7u

4)MEASUREMENT AND ANALYSIS:

<<4.1>>Theoretical time complexity analysis for each operation

1.push_back/push_front

We first create a node by malloc function and assign values to its data and pointer.Then we check for condition and insertion is done.since there are no loops involved the theoretical time complexity is 0(1)-constant time.

2.pop_back /pop_front

We first create a temp pointer so as to store the data for later use.Then we check for condition and deletion is done.since there are no loops involved the theoretical time complexity is 0(1)-constant time.

3.get_front/get_back

There is always a head pointer at the starting node of the dequeue.we just return the head for get front so time complexity is O(1).There is always a tail pointer at the end node of the dequeue.we just return the tail for get back so time complexity is O(1).

3.get_2front/get_2back

There is always a head pointer at the starting node of the dequeue and we know that the address of the first node always contain the address of second node so we just return the (head->npx) for get 2front so time complexity is O(1).There is always a tail pointer at the starting node of the dequeue and we know that the address of the last node always contain the address of last but one node so we just return the (tail->npx) for get 2back so time complexity is O(1).

4.PrintList

since we need to traverse all elements and print the time complexity for this is O(n).

5.count/IsEmpty

The count is a global variable while push operation it adds up and while pop operation it is reduced by 1.so the time complexity is O(1) for returning a variable.Is empty checks for the count the number of elements in the dequeue if it is zero it returns true otherwise false.since there are no loops it is also of O(1) complexity.

Push_back-O(1)

pop_back-O(1)

push_front-O(1)

pop_front-O(1)

printList-O(N)

size-O(1)

IsEmpty-O(1)

get_front-O(1)

get_2front-O(1)

get_back-O(1)

get_2back-O(1)

<<4.2>>Tabular data for measured time-taken vs N

Table for push_ back operation:

S.No.	Input Size	Time (seconds)
1.	1	0.000019
2.	10	0.000025
3.	100	0.000024
4.	1000	0.000048
5.	10000	0.000363
6.	100000	0.003031
7.	1000000	0.033336
8.	10000000	0.356852

II. Table for push_ front operation:

S.NO.	Input Size	Time (seconds)
1.	1	0.000018
2.	10	0.000024
3.	100	0.000024
4.	1000	0.000074
5.	10000	0.000396
6.	100000	0.003053
7.	1000000	0.029773
8.	10000000	0.293754

III. Table for push_ back n^{th} element operation:

S.NO.	Input Size	Time (seconds)
1.	1	0.000024
2.	10	0.00002
3.	100	0.000023
4.	1000	0.000019
5.	10000	0.000019
6.	100000	0.000025
7.	1000000	0.000037
8.	10000000	0.000037

IV. Table for push_ front n^{th} element operation:

S.NO.	Input Size	Time (seconds)
1.	1	0.000025
2.	10	0.000025
3.	100	0.000019
4.	1000	0.000019
5.	10000	0.000019
6.	100000	0.000037
7.	1000000	0.000037

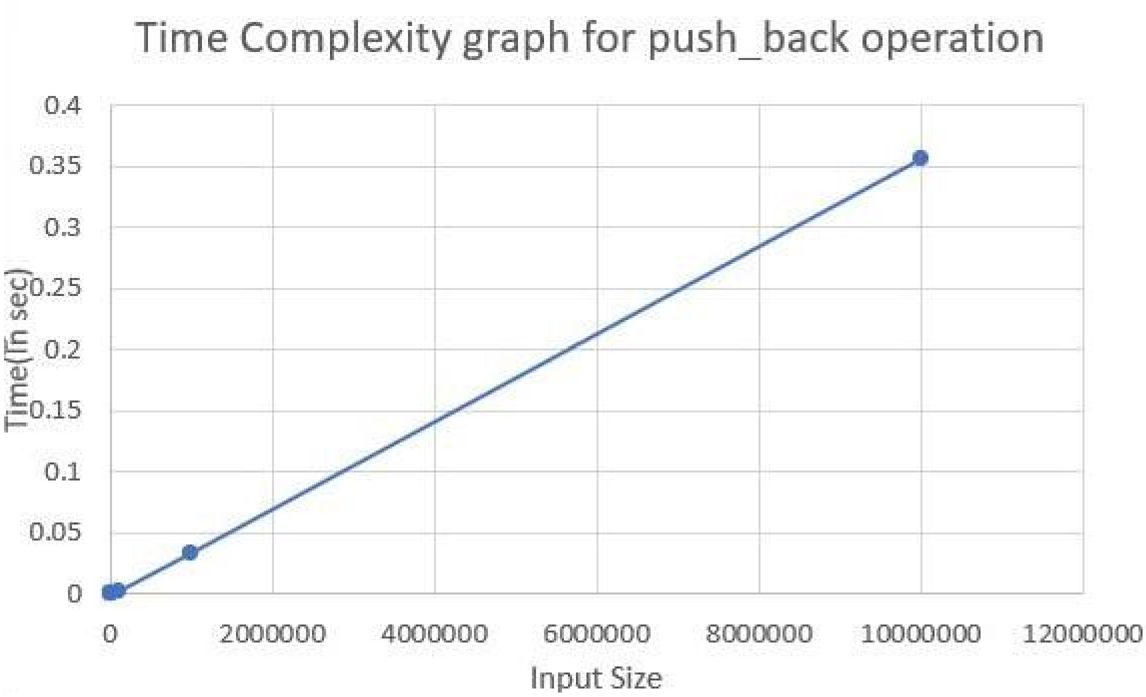
V. Table for pop_back n^{th} element operation:

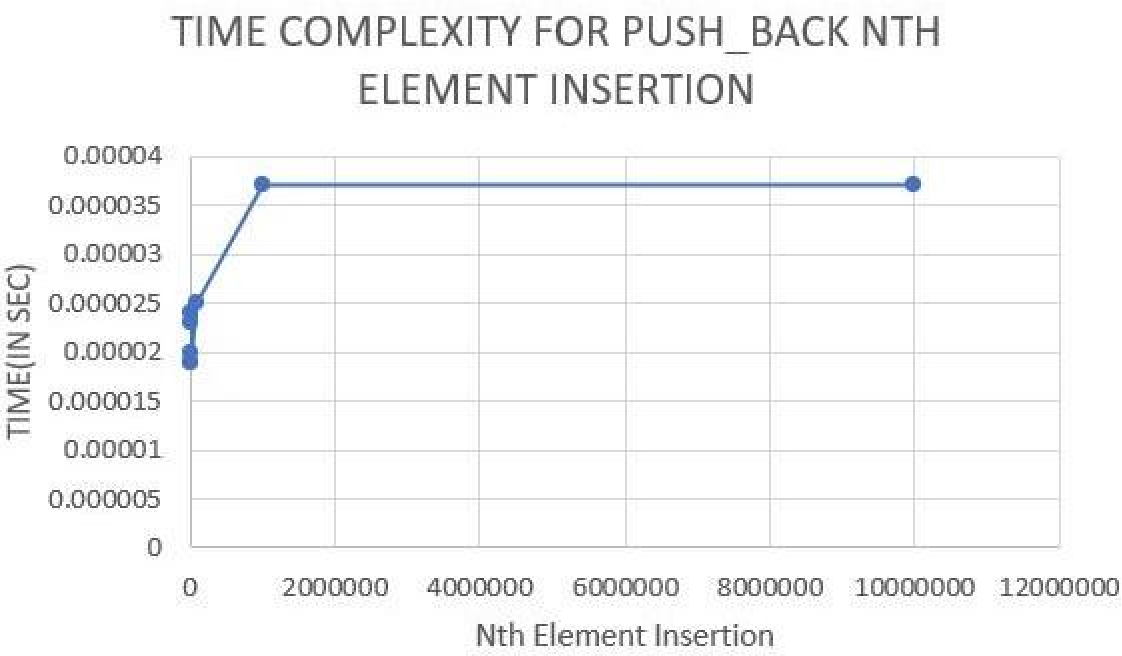
S.NO.	Input Size	Time (seconds)
1.	1	0.00003
2.	10	0.000028
3.	100	0.000031
4.	1000	0.000031
5.	10000	0.000037
6.	100000	0.000037
7.	1000000	0.000057
8.	10000000	0.000057

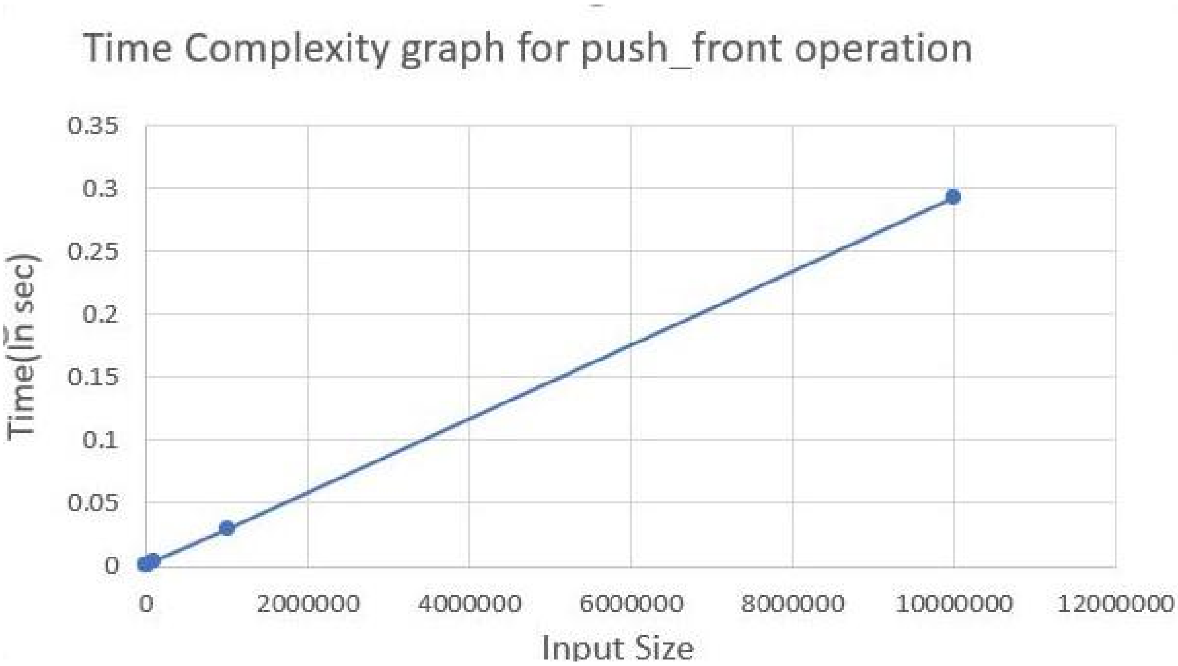
VI. Table for pop_front n^{th} element operation:

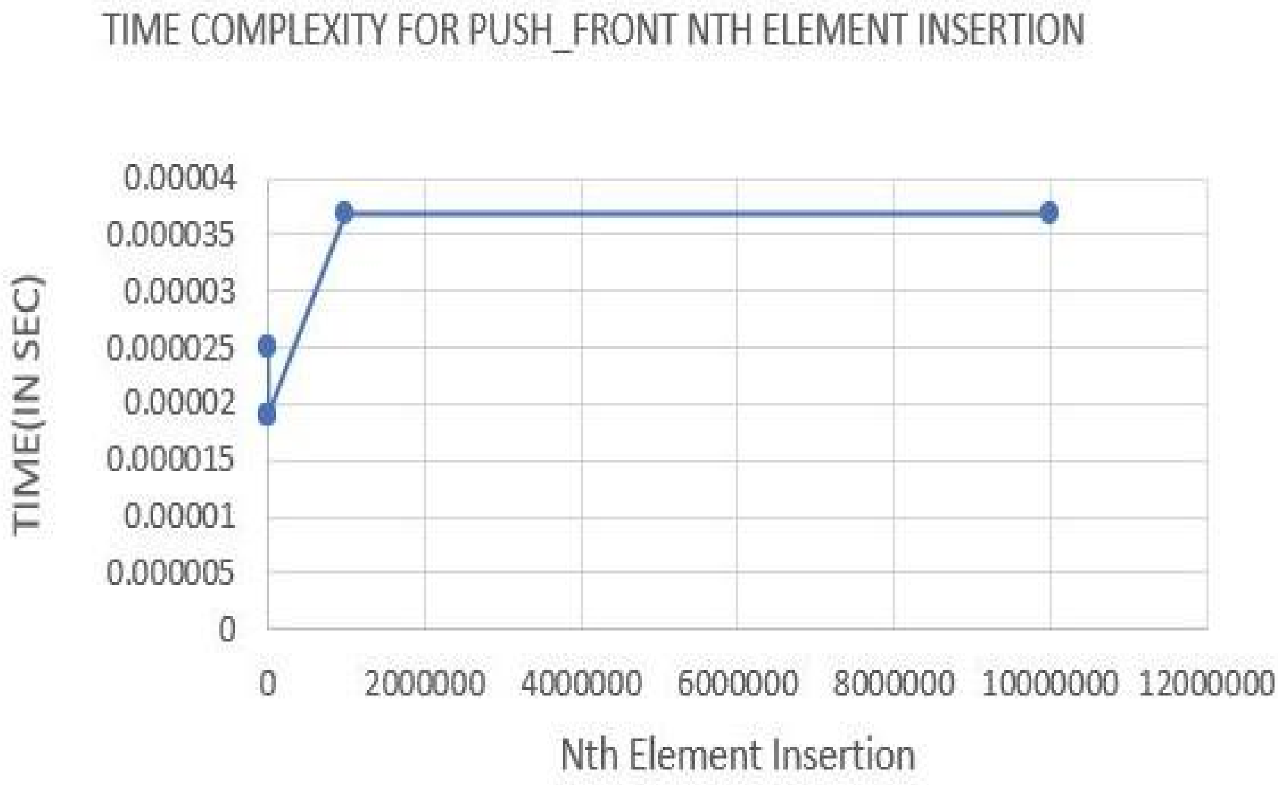
S.NO.	Input Size	Time (seconds)
1.	1	0.00003
2.	10	0.00003
3.	100	0.00003
4.	1000	0.000027
5.	10000	0.000026
6.	100000	0.000038
7.	1000000	0.000038
8.	10000000	0.000038

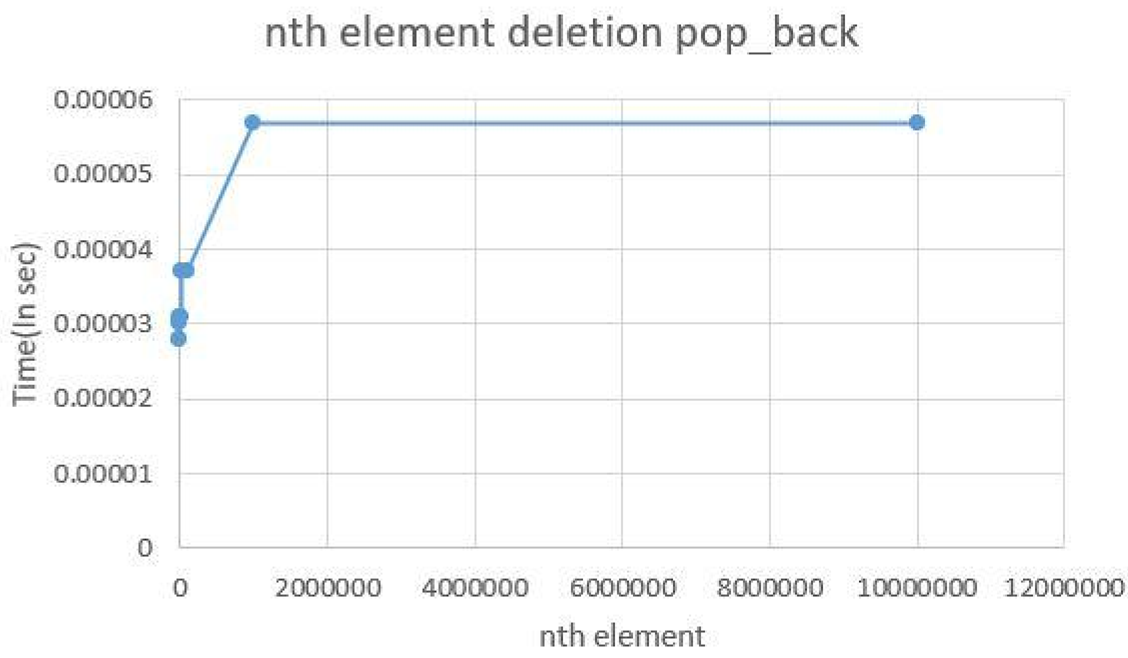
<<4.3>>Graph plotting

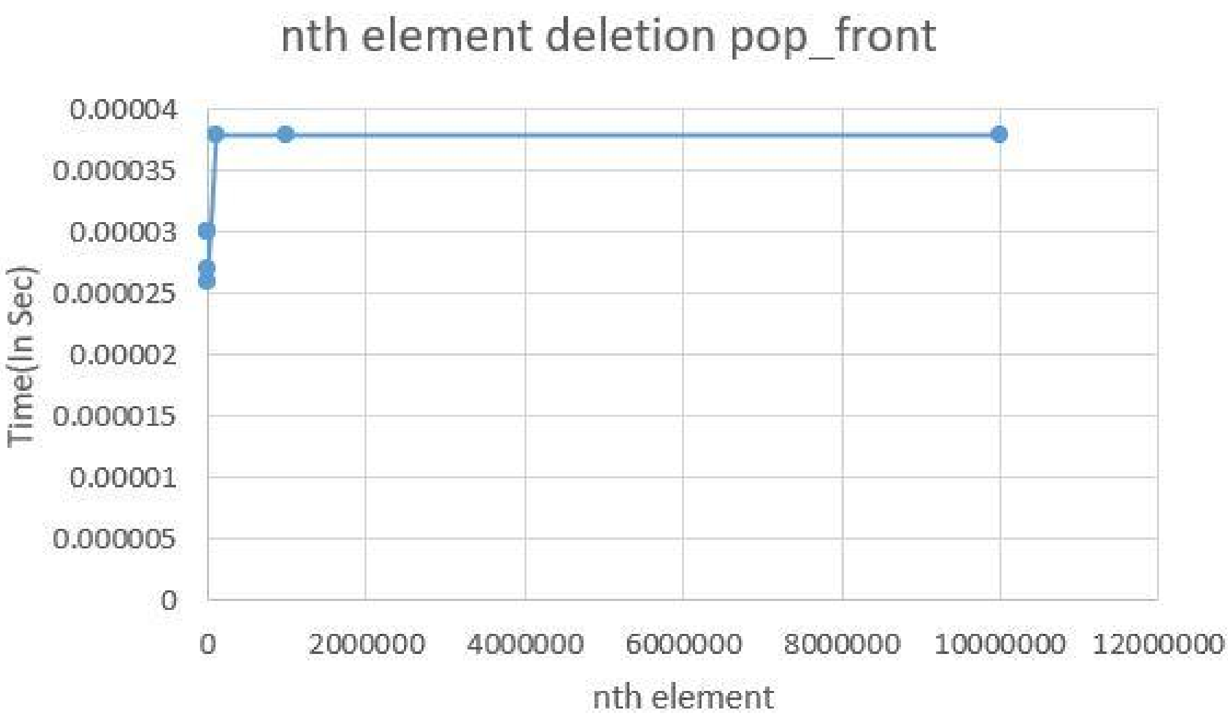





Time Complexity graph for push_front operation
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Time(in sec)
0
2000000
4000000
6000000
8000000
10000000
12000000
Input Size


TIME COMPLEXITY FOR PUSH_FRONT NTH ELEMENT INSERTION
0.00004
0.000035
0.00003
0.000025
0.00002
0.000015
0.00001
0.000005
0
TIME(IN SEC)
0
2000000
4000000
6000000
8000000
10000000
12000000
Nth Element Insertion


nth element deletion pop_back
0.00006
0.00005
0.00004
0.00003
0.00002
0.00001
0
Time(In sec)
0
2000000
4000000
6000000
8000000
10000000
12000000
nth element


nth element deletion pop_front
0.00004
0.000035
0.00003
0.000025
0.00002
0.000015
0.00001
0.000005
0
Time(In Sec)
0
2000000
4000000
6000000
8000000
10000000
12000000
nth element

5)CONCLUSIONS:

Deque is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front(head) or back(tail). We performed push _back/push_ front, pop_back /pop_front, get_ front/get_ back, get_2nd_front/get_2nd_back,size,empty operations with time complexity of O(1) using single pointer approach.We even found out that the maximum limit of elements in dequeue to be 10^7.

6)FUTURE ENHANCEMENTS:

Using other data structures like linked lists for additional bookkeeping so as to merge two or three dequeues so as to increase the size of input from a limit of 10^7.Take inputs of different data types (as we implemented only for integers). Taking multiple instances of input like creating 2-3 dequeues.

7)REFERENCE LINKS:

https://www.geeksforgeeks.org/time-complexities-of-different-data-structures/
https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/
https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/
https://en.cppreference.com/w/cpp/container/deque