MINI PROJECT

Deque using memory efficient linked list.

Team members:

Roll no	**Name**
19H51A0569	Suhasini
19H51A05A2	Iffat Maria
19H51A05C1	Yellaram Archana
19H51A05E3	Parinitha Pembarthi
19H51A05J2	V.Bhuvana Sri

Mentor: Mr. Ashutosh K Akhouri

Semester/Year: IV Semester/ II Year

Abstract

The aim of our project is to develop a Deque (Double ended queue), using memory efficient linked list. The developed program does the following operations to a deque: pushing to the front and rear of the deque, popping from the front and rear, getting the front and rear element, getting the second front and second rear element, returning the size of the deque and emptying all the elements. All of these operations take O(1) time to complete, the additional function to erase all the elements takes O(n) time to complete. Graphs are plotted by measuring the time taken for pushing elements into the front/ rear, as well as for getting the elements from the front/ rear.

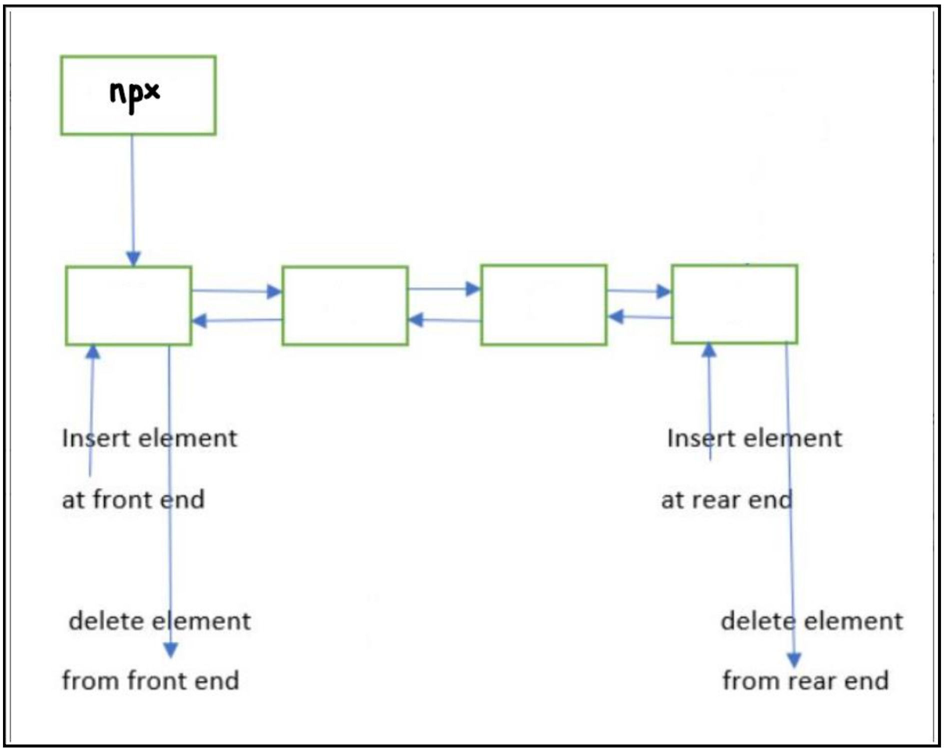


Table of contents

1.Project Description

1.1.Purpose of the project:

The aim of the project is to build a double ended queue using single pointer approach.

1.2.Goals/ requirements:

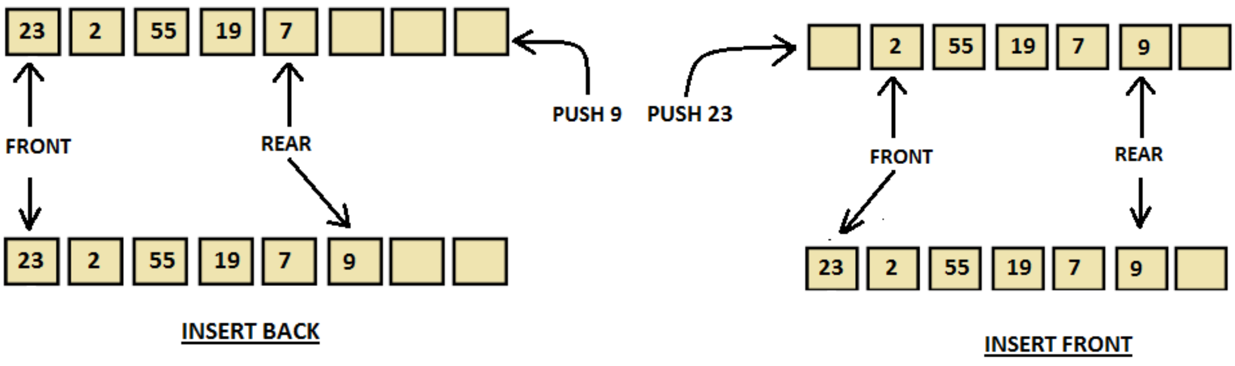
Writing the program using C++ and unit testing it after completion.
The operations performed on the deque must be done in O(1).

1.3.Methodology:

1. Alternative approaches

We can implement a deque in C++ using arrays as well as a linked list. Apart from this, the Standard Template Library (STL) has a class "deque" which implements all the functions for this data structure.

- **Deque using Doubly linked list**
  - To implement a Deque using a doubly linked list. We maintain two pointers front and rear, where front points to the front of a doubly linked list and rear points to the end. Also we need to maintain an integer size that stores the number of nodes in the Deque.
  - It requires 2 pointers as compared to XOR Linked List.



- **Deque using circular array**
  As it's a double-ended queue circular arrays are used for implementation.
  - Deques implemented using arrays do not allow the use of NULL elements.
  - Deque will only hold as many or even lesser elements as the array's size is fixed. Unfilled space will not be utilized as the front pointer of the queue would have moved ahead.
  - Arrays are allowed to grow as per the requirements, with restriction-free capacity and resizable array support being the two most important features.

- **Deque using 2 stacks**
  Two different Stacks are used to perform the operations on front and rear end respectively, which consumes more space and is static.

```
Given : Stack A, Stack B
                    // based on requirement b will be reverse of a
add_first(e)
{
    A.push(e);
}
remove_first(e)
{
    A.pop()
}
add_last(e)
{
    B.push(e);
}
remove_last()
{
    B.pop();
}
```

2. Current approach chosen

Deque using XOR linked list.

3. Detailed Description of current approach

In the current approach, the two pointers are replaced by a single pointer. And all the operations on the deque are performed using bitwise XOR operator to save memory. Also each node doesn't store its actual address instead it stores the address of previous and next node, which helps in traversing the list in both the directions like it would be done using double linked lists.

1.4.Measurements to be done:

- **Direct measurements:**
  - Operations like push_f()/push_b(), get_f()/get_b(), get_2_f()/get_2_b(), pop_f()/pop_b(), sz() --- Time complexity=> **O(1)**
  - deleteAll() --- Time complexity=> **O(n)**
- **Indirect measurements:**
  - Constant graph for push_f(), push_b(), get_f(), get_b()

1.5.Constraints:

Two pointers per node are not allowed.

1.6.Assumptions:

The program successfully responds to operations on deque for all conditions which includes unique conditions like:

- Empty queue: all operations give output for this condition.
- Queue with single element (concentrates on operations like get_2_f(), get_2_b(),etc.)

2.Code

https://ideone.com/UJvXOv

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

class node{
    public: long long int data;
    node* npx;
};
node *head = NULL, *tail= NULL;
long long int s = 0;

node* XOR(node *x,node *y)
{
    return reinterpret_cast<node*>(reinterpret_cast<uintptr_t>(x) ^ reinterpret_cast<uintptr_t>(y));
}

node* create(long long int x)
{
    node *newnode = new node();
    newnode->data = x;
    newnode->npx = NULL;
    return newnode;
}

void push_f(long long int x)
{
    node* n = create(x);
    node* p = NULL;
    if(head == NULL&&tail==NULL)
    {
        n->npx = XOR(p,head);
        head = n;
        tail = n;
    }
    else
    {
        n->npx = XOR(p,head);
        head->npx = XOR(n,head->npx);
        head = n;
    }
    s++;
}
```

```cpp
void push_b(long long int x)
{
    node* n = create(x);
    node* p = NULL;
    if(head == NULL&&tail==NULL)
    {
        n->npx = XOR(p,tail);
        tail = n;
        head = n;
    }
    else
    {
        n->npx = XOR(tail,p);
        tail->npx = XOR(n,tail->npx);
        tail = n;
    }
    s++;
}

bool pop_f()
{
    if(head==NULL&&tail==NULL)
        return false;
    else
    {
        node* p = NULL;
        node* n = XOR(head->npx,p);
        if(n==NULL)
        {
            free(head);
            head = NULL;
            tail = NULL;
        }
        else
        {
            n->npx = XOR(head,n->npx);
            n->npx = XOR(p,n->npx);
            free(head);
            head = n;
        }
        s--;
        return true;
    }
}
```

```
bool pop_b()
{
    if(head==NULL&&tail==NULL) return false;
    else
    {
        node* p = NULL;
        node* n = XOR(tail->npx,p);
        if(n==NULL)
        {
            free(tail);
            head = NULL;
            tail = NULL;
        }
        else
        {
            n->npx = XOR(tail,n->npx);
            n->npx = XOR(p,n->npx);
            free(tail);
            tail = n;
        }
        s--;
        return true;
    }
}

long long int get_f()
{
    if(head==NULL&&tail==NULL)
    {
        cout<<"No elements in the queue ";
        return 0;
    }
    return head->data;
}

long long int get_b()
{
    if(tail==NULL&&head==NULL)
    {
        cout<<"No elements in the queue ";
        return 0;
    }
    return tail->data;
}
```

```
long long int get_2_f()
{
    if((tail==NULL&&head==NULL))
    {
        cout<<"Queue is empty ";
        return 0;
    }
    else if(head==tail)
    {
        cout<<"Less than 2 elements";
        return 0;
    }
    node *p = NULL;
    node *n = XOR(head->npx,p);
    return n->data;
}



long long int get_2_b()
{
    if((tail==NULL&&head==NULL))
    {
        cout<<"Queue is empty ";
        return 0;
    }
    else if(head==tail)
    {
        cout<<"Less than 2 elements";
        return 0;
    }
    node *p = NULL;
    node *n = XOR(tail->npx,p);
    return n->data;
}




long long int sz()
{
    return s;
}
```

```
bool deleteAll()
{
    if(head == NULL&&tail==NULL) return false;
    else
    {
        node* tmp = NULL;
        node* next;
        while(head!=tail)
        {
            next = XOR(head->npx, tmp);
            tmp = head;
            free(head);
            head = next;
        }
        free(tail);
        head = NULL;
        tail = NULL;
    }
    s=0;
    return true;
}

int main()
{
     long long int i;
    for(i=0;i<100;i++)
    {
        push_f(i);
        push_b(i+1);
    }
    cout<<get_f()<<"\n";
    cout<<get_b()<<"\n";
    cout<<get_2_f()<<"\n";
    cout<<get_2_b()<<"\n";
    cout<<sz()<<"\n";
    cout<<pop_f()<<"\n";
    cout<<pop_b()<<"\n";
    cout<<sz()<<"\n";
    cout<<deleteAll()<<"\n";
    cout<<sz()<<"\n";
    return 0;
    return 0;
}
```

Output:

Success #stdin #stdout 0s 5496KB

stdin

```
Standard input is empty
```

stdout

```
99
100
98
99
200
1
1
198
1
0
```

3.Test Plans

3.1.Approach:

The program uses a single pointer approach and 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:

We plan to test the following features:

When the deque is empty:

- Output with pop_f()
- Output with pop_b()
- Output with deleteAll()
- Output with get_f()
- Output with get_b()
- Output with size()
- Output with get_2_f()
- Output with get_2_b()

When the deque has a single element:

- Output with get_2_f()
- Output with get_2_b()
- Output with pop_f() applied twice
- Output with pop_b() applied twice
- Output with pop_f() and pop_b() applied together
- Output with get_f() == get_b()

When the deque is filled using only push_f()

- Output with get_b()
- Output with get_2_b()

When the deque is filled using only push_b()

- Output with get_f()
- Output with get_2_f()

With random entries

- Output with get_f()
- Output with get_b()
- Output with get_2_f()
- Output with get_2_b()
- Output with sz()
- Output with deleteAll()

We couldn't test: how our code would handle operations on a fully filled deque.

3.3.Pass/ fail criteria

Test case	**Pass/fail criteria (Expected output)**
1. When deque is empty	
• Output with pop_f() • Output with pop_b()	`No elements in the queue, nothing to pop`
• Output with deleteAll()	`Empty Deque`
• Output with get_f() • Output with get_b()	`No elements in the queue`
• Output with size()	`0`
• Output with get_2_f() • Output with get_2_b()	`Queue is empty`
2. When deque has single element	
• Output with get_2_f() • Output with get_2_b()	`Less than 2 elements`
• Output with pop_f() applied twice • Output with pop_b() applied twice • Output with pop_f() and pop_b() applied together	`The queue is already empty.`
• Output with get_f() == get_b()	`/* prints the element given in input*/`
3. When the deque is filled using only push_f()	
• Output with get_b()	`/* prints the first element entered using push_f() */`
• Output with get_2_b()	`/* prints the 2nd element entered using push_f() */`
4. When the deque is filled using only push_b()	
• Output with get_f()	`/* prints the first element entered using push_b() */`
• Output with get_2_f()	`/* prints the 2nd element entered using push_b() */`

5. With random entries	ex:deque=> 4 5 6 2 1
• Output with get_f()	1
• Output with get_b()	4
• Output with get_2_f()	2
• Output with get_2_b()	5
• Output with sz()	5
• Output with deleteAll()	/* sz=0 */

3.4.List of test cases:

- **TC1 :**
  pop_f(underflow;sz=0)
- **TC2 :**
  pop_b(underflow;sz=0)
- **TC3 :**
  deleteAll(underflow;sz=0)
- **TC4 :**
  get_f(underflow;sz=0)
- **TC5 :**
  get_b(underflow;sz=0)
- **TC6 :**
  sz() with no elements in deque.
- **TC7 :**
  get_2_f(underflow;sz=0)
- **TC8 :**
  get_2_b(underflow;sz=0)
- **TC9 :**
  get_2_f(one element,sz=1)

- **TC10 :**
  get_2_b(one element;sz=1)
- **TC11 :**
  normal get_b()
- **TC12 :**
  get_2_f(4 elements)
- **TC13 :**
  get_2_f(5 elements)
- **TC14 :**
  get_2_b(6 elements)
- **TC15 :**
  get_2_b(5 elements)
- **TC16 :**
  single element get_f(), get_b(), sz=0
- **TC17 :**
  single element pop_f() twice
- **TC18 :**
  single element pop_b() twice
- **TC19 :**
  single element pop_f() and pop_b() together
- **TC20 :**
  checking get_b() by filling the queue using only push_f()
- **TC21 :**
  checking get_f() by filling the queue using only push_b()
- **TC22:**
  normal get_f()
- **TC23:**
  Checking get_2_b() by filling the queue using only push_f()
- **TC24:**
  Checking get_2_f() by filling the queue using only push_b()

3.5.Test programs listing

TC1: https://ideone.com/tOnhdA
TC2: https://ideone.com/tfhcC6
TC3: https://ideone.com/aqOvXd
TC4: https://ideone.com/b3rfYP
TC5: https://ideone.com/CW8lnM
TC6: https://ideone.com/7KJ6FI
TC7: https://ideone.com/c7UAXy
TC8: https://ideone.com/ZXcSGt
TC9: https://ideone.com/epvI9H
TC10: https://ideone.com/RR0ZPJ
TC11: https://ideone.com/1DTBFD
TC12: https://ideone.com/ASsNkm
TC13: https://ideone.com/l8KFaq
TC14: https://ideone.com/NyRwX9
TC15: https://ideone.com/fFWn8M
TC16: https://ideone.com/aYKT1R
TC17: https://ideone.com/h8kH6k
TC18: https://ideone.com/UBR9c3
TC19: https://ideone.com/Pw2yuF
TC20: https://ideone.com/RK8ObN
TC21: https://ideone.com/jOKll7
TC22: https://ideone.com/yGSqaX
TC23: https://ideone.com/nFc09z
TC24: https://ideone.com/TncHdb

4.Measurement and Analysis

4.1.Theoretical Time Complexity analysis for each question

Function name	Time Complexity
push_f()	O(1)
push_b()	O(1)
get_f()	O(1)
get_b()	O(1)
get_2_f()	O(1)
get_2_b()	O(1)
sz()	O(1)
pop_f()	O(1)
pop_b()	O(1)
deleteAll()	O(n)

4.2.Tabular data for measured time-taken vs N

i. Tabular data for measured time vs N: **push_f()**

n - push_f()	Time taken (microsecs)
1	0
10	0
100	3
1000	37
10000	399
100000	3135
1000000	38997
10000000	381895

ii. Tabular data for measured time vs N: **push_b()**

n - push_b()	Time taken (microsecs)
1	0
10	0
100	4
1000	32
10000	393
100000	4450
1000000	32002
10000000	307872

iii. Tabular data for measured time vs N: **get_f()**

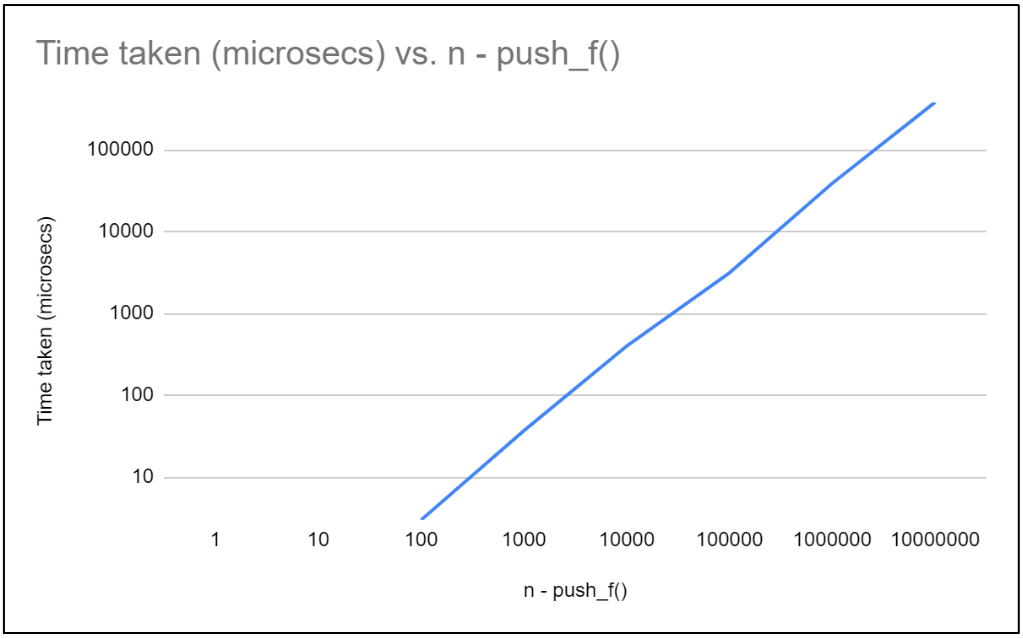
nth - get_f()	Time taken (microsecs)
1	8
10	9
100	9
1000	9
10000	9
100000	10
1000000	14
10000000	16

iv. Tabular data for measured time vs N: **get_b()**

nth - get_b()	Time taken (microsecs)
1	7
10	8
100	9
1000	10
10000	10
100000	10
1000000	14
10000000	21

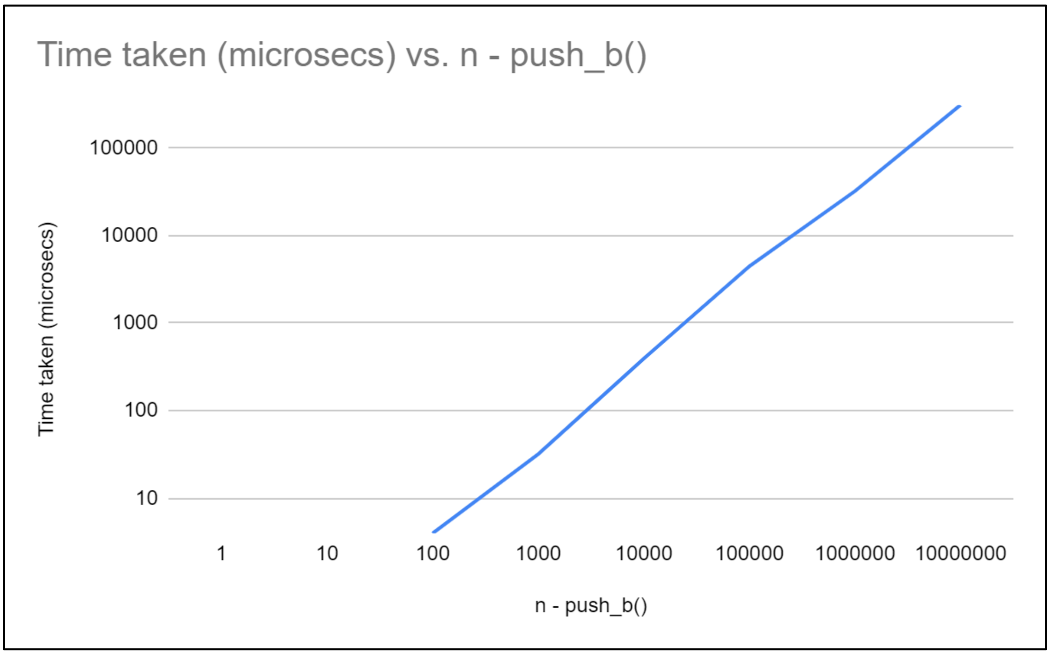
4.3.Graph plotting

1. Graph plotted using measured time for: **push_f()**
   https://ideone.com/nrHYHL



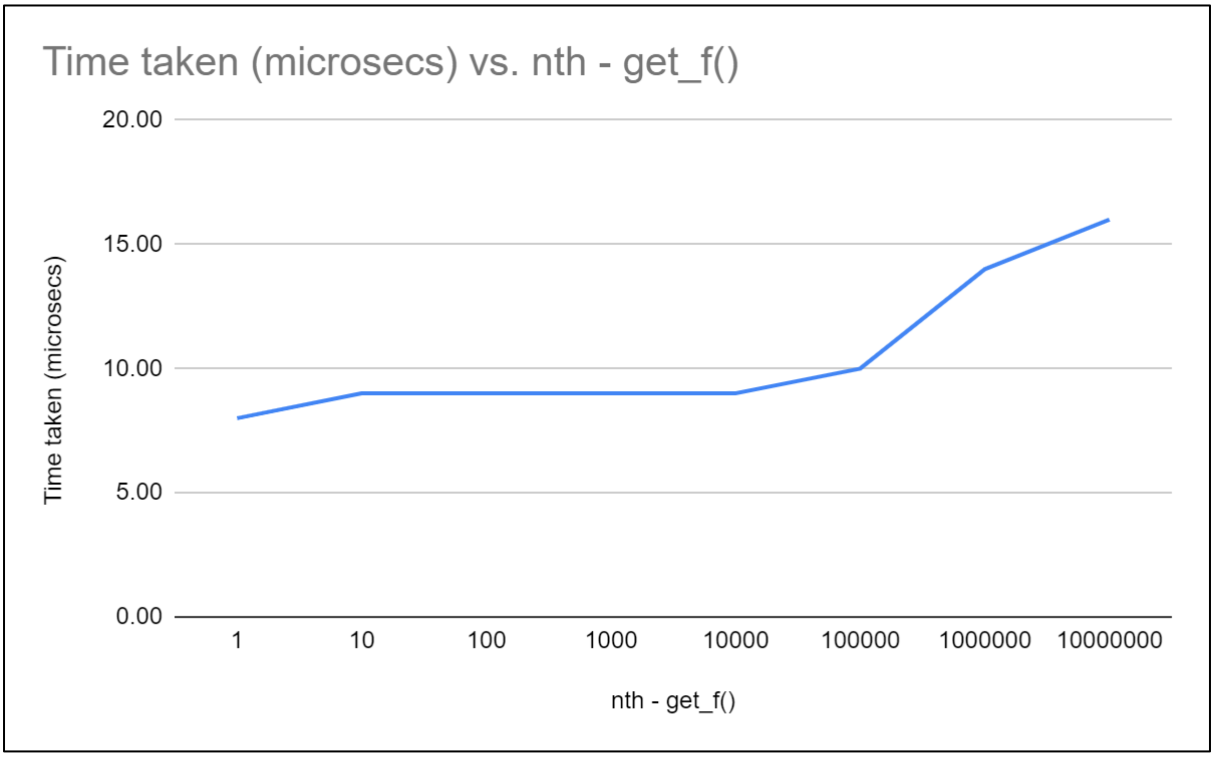
A linearly increasing graph=>O(1)

2. Graph plotted using measured time for: **push_b()**
   https://ideone.com/SkG2WW



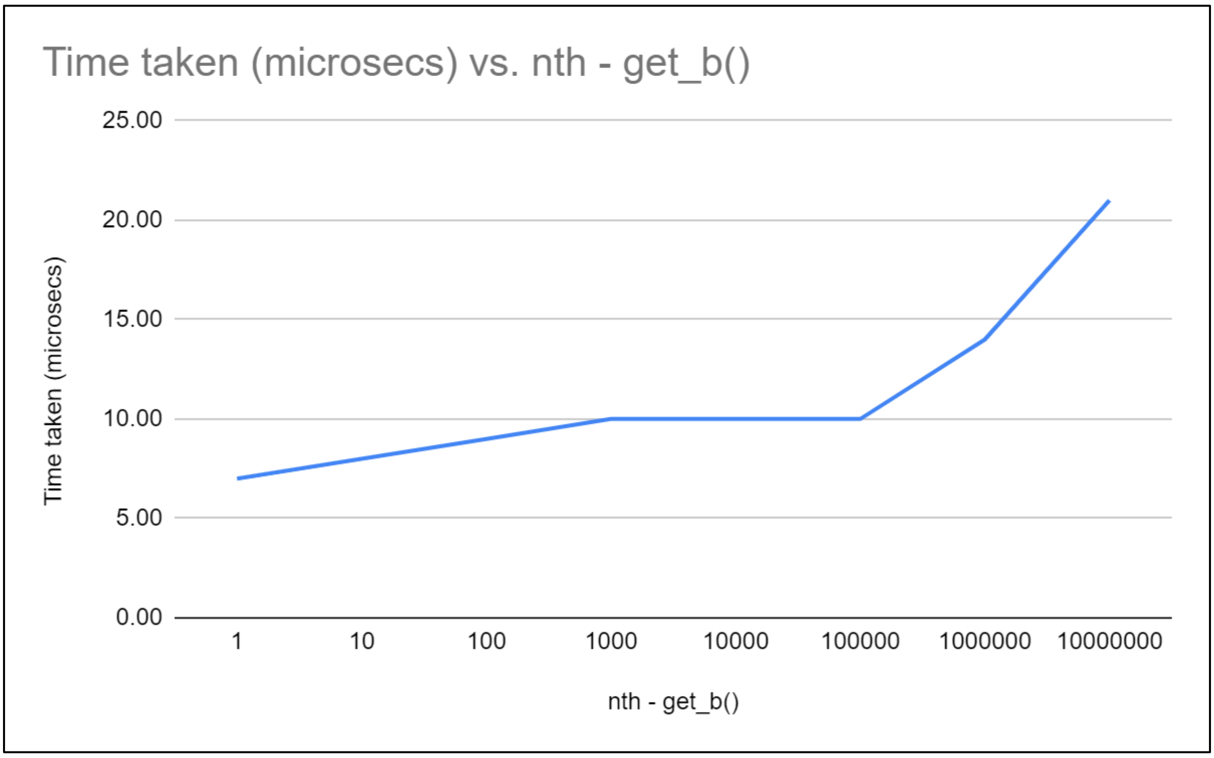
A linearly increasing graph=>O(1)

3. Graph plotted using measured time for: **get_f()**
   https://ideone.com/0ecKyt



Linearly Constant graph => O(1)

4. Graph plotted using measured time for: **get_b()**
   https://ideone.com/iCIvzJ



Linearly Constant graph => O(1)

5.Conclusions

The following conclusions can be drawn from the adapted model for building Deque using memory efficient linked list:

- **A memory efficient solution:**
  We were able to build standard functions and operations on deque using memory efficient linked list, i.e. by using single pointer.
- **A dynamic ADT:**
  The code runs efficiently for approximately 10^7 elements in the deque.
- **Successful test plans:**
  We carefully selected various test cases which check the functionality of all the operations performed on the Deque ADT and successfully cleared all the pass/fail criterias.
- We were able to plot the graph for push_f(), push_b(), get_f(), get_b() functions and were able to analyse the time complexity on various test cases.
- We theoretically analysed the time complexity for our code using the step count method
- After comparing the current solution with various alternative approaches to solve the problem, it can be concluded that the current approach is more memory efficient and simple.

6.Future Enhancements

- **Throwing errors:**
  For out of boundary inputs from users the program shall throw errors by using Object Oriented Programming.

For example:

Deque ADT	Interface java.util.Deque	
	throws exceptions	returns special value
first( )	getFirst( )	peekFirst( )
last( )	getLast( )	peekLast( )
addFirst(e)	addFirst(e)	offerFirst(e)
addLast(e)	addLast(e)	offerLast(e)
removeFirst( )	removeFirst( )	pollFirst( )
removeLast( )	removeLast( )	pollLast( )
size( )	size( )	
isEmpty( )	isEmpty( )	

7.Difficulties Faced

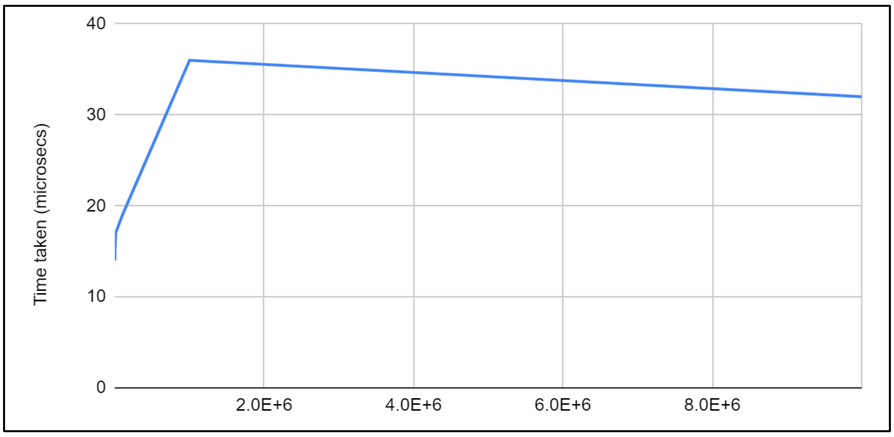
- One of the expected test cases was to check the push operations on a completely filled deque, but since the current deque is dynamic, attempts to overfill the queue made the computer crash.

```
int main()
{
    long long int i;
    for(i=0;i<=100000000;i++)
    {
        push_f(i);
        push_b(i+1);
    }
    return 0;
}
```

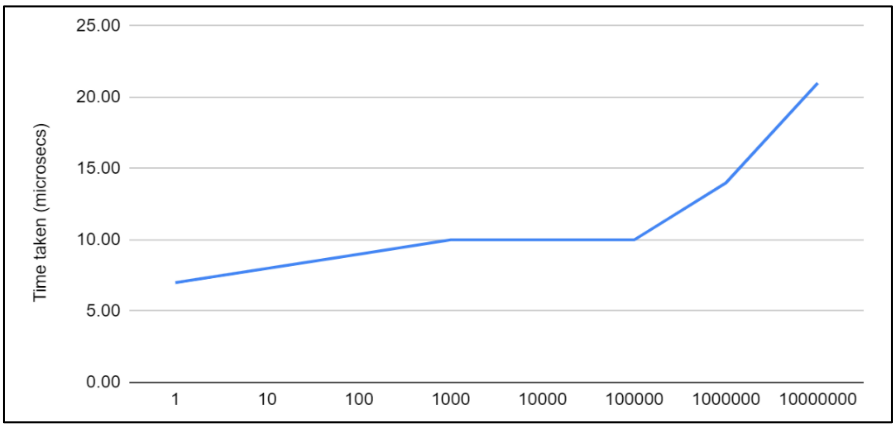
Runtime error #stdin #stdout 2.34s 2095728KB

- Theoretically, the get_f() and get_b() operations take unit time for execution, but for plotting graphs, the time taken for execution varied drastically. Which resulted in a rather non-linear, non-constant graph.

Expected graph



Attained graph



8.Reference links

https://stackoverflow.com/questions/66768578/c-xor-linked-list-deleting-data

https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/

xorlist/xorlist.c at master · kylelaker/xorlist · GitHub

Implementation of Deque using doubly linked list - GeeksforGeeks