Assignment 13 01118

Problem statement:
A double-ended queue (deque) is a linear list in which additions & deletions may be made at either end. Obtain a data representation mapping a deque into a one-dimensional array. Write C++ program to simulate deque, with functions to add & delete elements from either end of the deque.

Objectives
1. Understand the double-ended queue and it's implementation details.
2. Understand various operations that are performed on deque & do there time & space complexity analysis.

Outcomes:
1. Implement deque data structure & its various operations
2. Do analysis of various deque operations.

Hardware requirement:
Manufacturer & Model: Acer Swift-3
Processor: Intel core i5 8th gen (8265U @ 1.6 GHz)
Installed memory: 8GB RAM, 512GB SSD
Architecture: 64-bit

Software Requirement:
Operating System: Ubuntu 20.04 LTS on oracle virtual machine (3 processors & 4096MB base memory is allocated)
C++ version: C++ 14
Compiler for C++: g++ (version: 10.1.0)
Code editor: Sublime Text (Build 2011)

Theory
Doubly ended Queue @ deque is a queue data structure

In which insertion & deletions operations are performed on both the ends (front & rear).

The operations of deque are:

insertfront(): add item at front of queue.
insertRear(): add item to last of queue
deletefront(): remove front item of queue
deleteRear(): remove last item of queue
getfront(): to get the front item of queue
getRear(): to get rear item of queue.

Pseudo Code:

Array based implementation.

```
Algorithm isEmpty() {
    return (front == rear)
}

Algorithm isfull() {
    if front+1 == rear
        (or) rear+1 = front
        return true.
    else return false
}

Algorithm insertfront(T x) {
    if (list is full)
        return.
    else
        arr[front] ← x
        front ← (front+1) %.D
}

Algorithm insertRear(T x) {
    if (list is full) return
    else rear ← (rear+1) %.D
```

arr[rear] = x

}

Algorithm getFront() {
    return arr[(front+1) % D];
}

Algorithm getRear() {
    return arr[rear]
}

Algorithm removeFront() {
    IF (isEmpty())
        return
    else {
        front ← (front+1) % D
    }
}

Algorithm removeRear() {
    IF (isEmpty())
        return
    else {
        rear ← (rear+D-1) % D
    }
}

ADT of class

* Dequeue class

class Dequeue {
private:
    Node *front, *rear;

```
public:
    Dequeue() {     //constructor
        //code
    }
    bool isEmpty() {    //checks if queue is empty
        //code
    }
    void insertFront(x) {   // inserts x at front of queue.
        //code
    }
    void insertRear(x) {    // inserts x at rear of queue
        //code
    }
    int getFront() {        // return element at front of queue
        //code
    }
    int getRear() {         // return element at rear of queue.
        //code
    }
    void removeFront() {    // removes element which is at front of queue
        //code
    }
    void removeRear() {     // removes element which is at
        //code                  rear of queue.
    }
```

Analysis of Algorithms:

	operation	time complexity	space complexity
①	Insert at front	constant time i.e. O(1)	const. space O(1)
②	Insert at rear	const. time i.e O(1)	const. space O(1)
③	Remove front	const time is O(1)	const. space O(1)
④	remove rear	const. tm i.e O(1)	const. space O(1)

⑤	get front element	const. time ie. O(1)	Const. space ie. O(1)
⑥	get rear element	const. time ie. O(1)	const. space ie. O(1)

Testcases

Sr No.	Testcase Description	Expected output	Actual Output
1)	Add element at front	Algorithm/ program should add element at front of queue.	Element is added successfully at front of queue.
2)	Add element at rear	Program should add element at rear of queue.	Element is added at rear succesfully.
3)	Remove front element	program should remove front element of queue	Program removes front element of queue.
4)	Remove rear element	program should remove rear element of queue.	Program removes rear element of queue successfully.
5)	Get front of queue	program should show/ print front element of queue.	Program prints front element of queue.
6)	Get rear of queue	program should print rear element of queue.	Program prints rear element of queue.
7)	Remove element from front from empty queue.	program should show appropriate message.	program shows message: Queue is empty
8)	get ~~front~~ rear element of empty queue.	program should show appropriate message.	program shows message: Queue is empty.

Applications

The applications of dequeue data structure are as follows:

① An internet browser's history

② storing a computer code applications list of undo operations.

③ In money control apps. Dequeue is used to show the stocks last visited, it'll take away the stocks when a while & can add the most recent ones.

Conclusion

Dequeue is simple yet very useful data structure which is used in computer science. This allows us to use functionalities of both stack & queue to be used in same data structure. This assignment provides insightful view of deque data structure & analysis of its common operations.

* * *