**Deque**

# Deque

Double-ended queue (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). It is **also called a head-tail linked list**.

# Distinctions and sub-types

This differs from the queue ADT or FIFO, where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

1. An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only
2. An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only

Both the basic and most common list types in computing, queues and stacks can be considered specializations of deques, and can be implemented using deques.

# Time Complexity

**In a doubly-linked list implementation**

* The time complexity of all deque operations is O(1)
* The time complexity of insertion or deletion in the middle, given an iterator, is O(1)
* The time complexity of random access by index is O(n)

**In a growing array**

* The amortized time complexity of all deque operations is O(1)
* The time complexity of random access by index is O(1)
* The time complexity of insertion or deletion in the middle is O(n)

# Operations

## Primary Operations

### enqueue at front

Adds an item at the front of Deque

### enqueue at rear

Adds an item at the rear of Deque

### dequeue front element

Deletes an item from front of Deque

### dequeue rear element

Deletes an item from rear of Deque

## Additional Operations

### peek front element

Gets the front item from queue

### peek rear element

Gets the last item from queue

### check empty

Checks whether Deque is empty or not

### check full

Checks whether Deque is full or not

# Applications

1. It can be used as both Stack and Queue
2. In Steal job scheduling algorithm

# Deque Implementation

## Using a doubly linked list

It is easy to implement and can be understand from the queue implementation using linked list.

## Using circular array

// C++ implementation of De-queue using circular array

#include<iostream>

using namespace std;

// Maximum size of array or Dequeue

#define MAX 100

// A structure to represent a Deque

class Deque {

int arr[MAX];

int front;

int rear;

int size;

public :

Deque(int size) {

front = -1;

rear = 0;

this->size = size;

}

// Operations on Deque:

void insertfront(int key);

void insertrear(int key);

void deletefront();

void deleterear();

bool isFull();

bool isEmpty();

int getFront();

int getRear();

};

// Checks whether Deque is full or not.

bool Deque::isFull() {

return ((front == 0 && rear == size-1)|| front == rear+1);

}

// Checks whether Deque is empty or not.

bool Deque::isEmpty () {

return (front == -1);

}

// Inserts an element at front

void Deque::insertfront(int key) {

// check whether Deque if full or not

if (isFull()) {

cout << "Overflow\n" << endl;

return;

}

// If queue is initially empty

if (front == -1) {

front = 0;

rear = 0;

}

// front is at first position of queue

else if (front == 0)

front = size - 1 ;

else // decrement front end by '1'

front = front-1;

// insert current element into Deque

arr[front] = key ;

}

// function to inset element at rear end

// of Deque.

void Deque ::insertrear(int key) {

if (isFull()) {

cout << " Overflow\n " << endl;

return;

}

// If queue is initially empty

if (front == -1) {

front = 0;

rear = 0;

}

// rear is at last position of queue

else if (rear == size-1)

rear = 0;

// increment rear end by '1'

else

rear = rear+1;

// insert current element into Deque

arr[rear] = key ;

}

// Deletes element at front end of Deque

void Deque ::deletefront() {

// check whether Deque is empty or not

if (isEmpty()) {

cout << "Queue Underflow\n" << endl;

return ;

}

// Deque has only one element

if (front == rear) {

front = -1;

rear = -1;

}

else

// back to initial position

if (front == size -1)

front = 0;

else // increment front by '1' to remove current

// front value from Deque

front = front+1;

}

// Delete element at rear end of Deque

void Deque::deleterear() {

if (isEmpty()) {

cout << " Underflow\n" << endl ;

return ;

}

// Deque has only one element

if (front == rear) {

front = -1;

rear = -1;

}

else if (rear == 0)

rear = size-1;

else

rear = rear-1;

}

// Returns front element of Deque

int Deque::getFront() {

// check whether Deque is empty or not

if (isEmpty()) {

cout << " Underflow\n" << endl;

return -1 ;

}

return arr[front];

}

// function return rear element of Deque

int Deque::getRear() {

// check whether Deque is empty or not

if(isEmpty() || rear < 0) {

cout << " Underflow\n" << endl;

return -1 ;

}

return arr[rear];

}

// Driver program to test above function

int main() {

Deque dq(5);

cout << "Insert element at rear end : 5 \n";

dq.insertrear(5);

cout << "insert element at rear end : 10 \n";

dq.insertrear(10);

cout << "get rear element " << " " << dq.getRear() << endl;

dq.deleterear();

cout << "After delete rear element new rear" << " become " << dq.getRear() << endl;

cout << "inserting element at front end \n";

dq.insertfront(15);

cout << "get front element " << " " << dq.getFront() << endl;

dq.deletefront();

cout << "After delete front element new " << "front become " << dq.getFront() << endl;

return 0;

}

Output:

insert element at rear end : 5

insert element at rear end : 10

get rear element : 10

After delete rear element new rear become : 5

inserting element at front end

get front element : 15

After delete front element new front become : 5

# END