COSC 21063

Data Structures and Algorithms

[Group 14]

PS/2020/010-A.E.P.P.Jayasekara

PS/2020/012-A.M.L.Athukorala

PS/2020/085-D.S.K.Wickramasinghe

PS/2020/024-K.D.C.D.Fernando

PS/2020/005-K.S.D.Wickramanayaka

PS/2020/078-K.G.Heenatigala

PS/2020/220-M.S.S.Jinadasa

PS/2020/095-W.M.P.G.Wijekoon

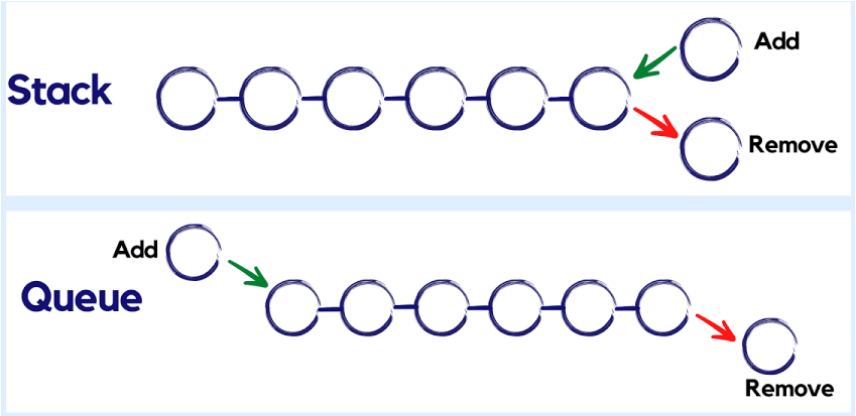
Dequeue Data Structure

Definition

Dequeue or double-ended queue is an generalized data structure of Queue that inherits the properties of both queues and stacks where the insertion and deletion operations are performs from both ends.

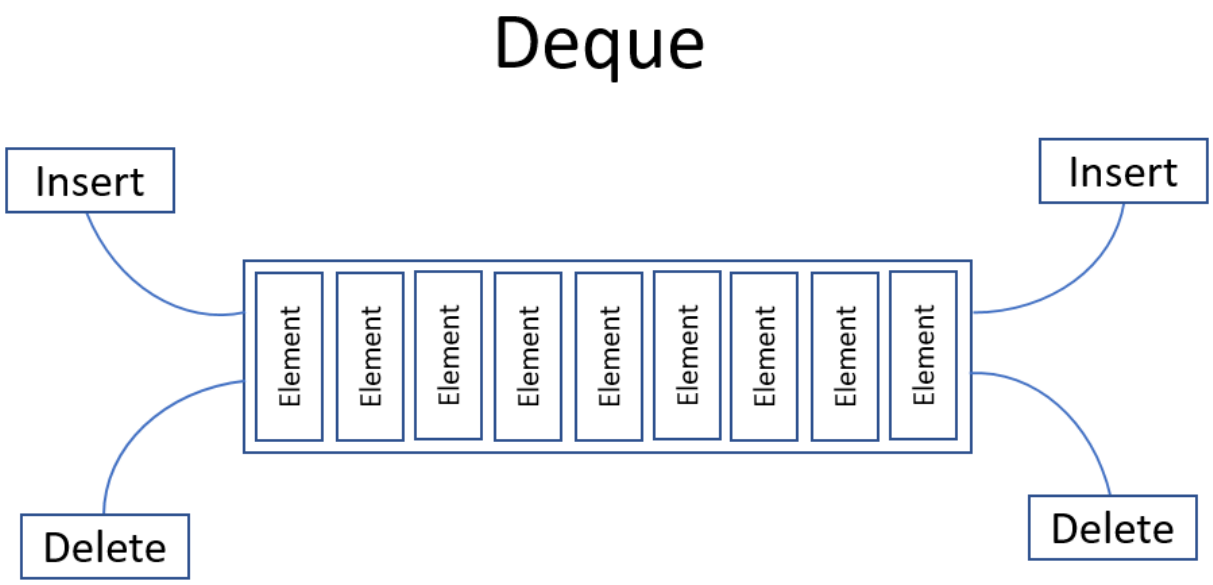
Introduction

The Dequeue(Double-ended Queue) appears as a crucial and adaptable linear data structure in the field of computer science and data management. The dequeue make a bridge for gap between stacks and queues by allowing insertion and removal of components either front or rear ends, in contrast to normal queues.

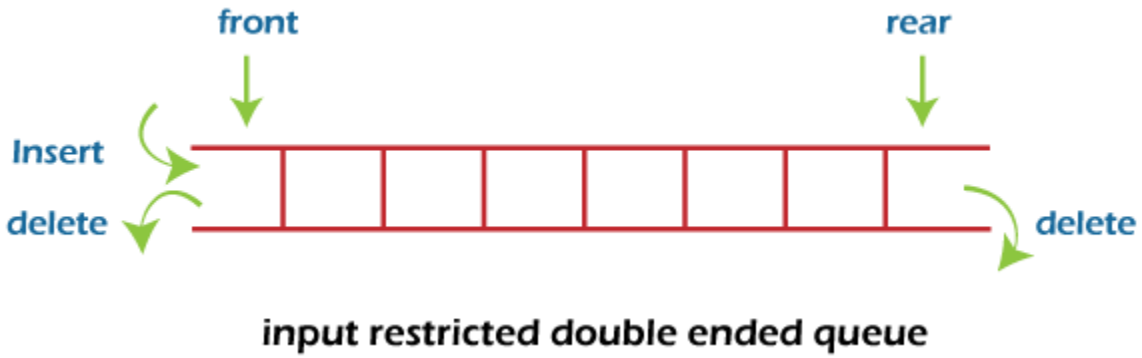


After recognizing that many real-world settings necessitate a more adaptable data processing tool led to development of the Dequeues dual -ended prowess.

Also dequeues dual-ended prowess use to make data management more flexible, such as when handling work scheduling effectively, controlling sliding data windows and creating sophisticated algorithms that necessitate swift element manipulation.



An also there are some special types of dequeues that can delete a element by both sides and only insert a element by one side.



Operation Specification

Dequeue support comprehensive set of operations that enable seamless management of elements.

isEmpty()

- input: a dequeue
- Output: isEmpty true or false
- Precondition: a dequeue has been created.
- Postcondition: isEmpty is true if dequeue is empty.

isFull()

- input: a dequeue
- Output: isFull true or false
- Precondition: a dequeue has been created.
- Postcondition: isFull is true if dequeue is full.

addFirst()

- input: a dequeue and x value
- Output: dequeue changed(x has added to the front)
- Precondition: a dequeue has been created and not full.
- Postcondition: dequeue has x as its front entry.

addLast()

- input: a dequeue and x value
- Output: dequeue changed(x has added to the rear)
- Precondition: a dequeue has been created and not full.
- Postcondition: dequeue has x as its rear entry.

removeFirst()

- input: a dequeue
- Output: dequeue changed and x the element removed
- Precondition: a dequeue has been created and not empty. If empty return Exception.
- Postcondition: The first entry in the front side of dequeue has been removed and returned as the value of x.

removeLast()

- input: a dequeue
- Output: dequeue changed and x the element removed
- Precondition: a dequeue has been created and not empty. If empty return Exception.
- Postcondition: The last entry in the rear side of dequeue has been removed and returned as the value of x.

pollFirst()

- input: a dequeue
- Output: dequeue changed and x the element removed
- Precondition: a dequeue has been created and not empty. If empty return null
- Postcondition: The first entry in the front side of dequeue has been removed and returned as the value of x.

pollLast()

- input: a dequeue
- Output: dequeue changed and x the element removed
- Precondition: a dequeue has been created and not empty. If empty return null
- Postcondition: The last entry in the rear side of dequeue has been removed and returned as the value of x.

getFirst()

- input: a dequeue
- Output: front element of dequeue returns
- Precondition: a dequeue has been created and not empty. If empty return Exception.
- Postcondition: dequeue is not change and return only current front.

getLast()

- input: a dequeue
- Output: rear element of dequeue returns
- Precondition: a dequeue has been created and not empty. If empty return Exception.
- Postcondition: dequeue is not change and return only current rear.

peekFirst()

- input: a dequeue
- Output: front element of dequeue returns
- Precondition: a dequeue has been created and not empty. If empty return null.
- Postcondition: dequeue is not change and return only current front.

peekLast()

- input: a dequeue
- Output: rear element of dequeue returns
- Precondition: a dequeue has been created and not empty. If empty return null.
- Postcondition: dequeue is not change and return only current rear.

getSize()

- input: a dequeue
- Output: return the size of dequeue

contains()

- input: a dequeue and x element
- Output: true or false.
- Postcondition: dequeue is not change and return true if there is a value equal to the x in dequeue.

removeFirstOccurence()

- input: a dequeue and x value.
- Output: if current front equals to x value, remove current front and return true, else return false.
- Postcondition: dequeue is change if and only current front equals x.

removeLastOccurence()

- input: a dequeue and x value.
- Output: if current rear equals to x value, remove current rear and return true, else return false.
- Postcondition: dequeue is change if and only current rear equals x.

add()

- input: a dequeue and x value, Boolean condition true or false.
- Output: dequeue changed(x has added to the front if true as Boolean condition)
- Precondition: a dequeue has been created and not full.
- Postcondition: dequeue has x as its rear entry if false and dequeue has x as its front entry if true.

remove()

- input: a dequeue
- Output: dequeue changed and x the element removed
- Precondition: a dequeue has been created and not empty. If empty return Exception.
- Postcondition: The first entry in the front side of dequeue has been removed and returned as the value of x.

poll()

- input: a dequeue
- Output: dequeue changed and x the element removed
- Precondition: a dequeue has been created and not empty. If empty return null
- Postcondition: The first entry in the front side of dequeue has been removed and returned as the value of x.

element()

- input: a dequeue
- Output: front element of dequeue returns
- Precondition: a dequeue has been created and not empty. If empty return Exception.

Postcondition: dequeue is not change and return only current front.

peek()

- input: a dequeue
- Output: front element of dequeue returns
- Precondition: a dequeue has been created and not empty. If empty return null.
- Postcondition: dequeue is not change and return only current front.

toString()

- input: a dequeue
- Output: return all the elements in dequeue.

deleteDequeue()

- input: a dequeue
- Output: All the values in dequeue will be removed.

Contiguous Implementation

An array used to develop the dequeue data structure in contiguous implementation. The items are kept in the array in a sequential order and indices are kept to monitor both front and rear locations.

This method comes with the difficulty of potential resizing as the array fills up, yet providing constant time access to elements and simple memory management. Due to the constant copying of pieces, this might results in ineffective memory usage and performance loss.

Class DeQueue-:

```
package com.Assignment2;
//Contiguous Implementation of DeQueue Data Structure

import java.util.NoSuchElementException;

public class DeQueue<T> { // T is a Parameter,when create an instance of
Dequeue with an argument it will be replaced
    private Object[] arr;
    private int front;
    private int rear;
    private int size;

    public DeQueue(int capacity) {
        arr=new Object[capacity];
        front=-1;
        rear=0;
        size=0;
    }
    public boolean isEmpty()
    {
        return size==0;
    }
    public boolean isFull()
    {
        return size==arr.length;
    }
    public void addFirst(T data)
    {
        if (isFull())
        {
            throw new IllegalStateException("DeQueue is Full.Cannot insert");
```

```
        } else if (front==-1) {
            front=0;
            rear=0;
        } else if (front==0) {
            front=arr.length-1;
        }
        else {
            front--;
        }
        arr[front]=data;
        size++;

    }
    public void addLast(T data)
    {
        if (isFull())
        {
            throw new IllegalStateException("DeQueue is Full.Cannot insert");
        } else if (front==-1) {
            front=0;
            rear=0;
        } else if (rear==arr.length-1) {
            rear=0;
        }
        else {
            rear++;
        }
        arr[rear]=data;
        size++;
    }
    public T removeFirst()
    {
        if (isEmpty())
        {
            throw new NoSuchElementException("DeQueue is Empty.Cannot
delete");
        }
        T data=(T) arr[front];
        arr[front]=null;
        if (front==rear) {
            front=-1;
            rear=-1;
        } else if (front==arr.length-1) {
           front=0;
        }
        else {
            front++;
        }
        size--;
        return data;
    }
    public T pollFirst()
    {
        if (isEmpty())
        {
            return null;
        }
```

```
        T data=(T) arr[front];
        arr[front]=null;
        if (front==rear) {
            front=-1;
            rear=-1;
        } else if (front==arr.length-1) {
            front=0;
        }
        else {
            front++;
        }
        size--;
        return data;
    }
    public T removeLast()
    {
        if (isEmpty())
        {
            throw new NoSuchElementException("DeQueue is Empty.Cannot
delete");
        }
        T data=(T) arr[rear];
        if(front==rear)
        {
            front=-1;
            rear=-1;
        } else if (rear==0) {
            rear=arr.length-1;
        }
        else {
            rear--;
        }
        size--;
        return data;
    }
    public T pollLast()
    {
        if (isEmpty())
        {
            return null;
        }
        T data=(T) arr[rear];
        if(front==rear)
        {
            front=-1;
            rear=-1;
        } else if (rear==0) {
            rear=arr.length-1;
        }
        else {
            rear--;
        }
        size--;
        return data;
    }

    public T getFirst() {
```

```
        if (isEmpty())
        {
            throw new NoSuchElementException("DeQueue is Empty");
        }
       else {
           return (T)arr[front];
        }
    }
    public T peekFirst() {
        if (isEmpty())
        {
           return null;
        }
        else {
            return (T)arr[front];
        }
    }

    public T getLast() {
        if (isEmpty())
        {
            throw new NoSuchElementException("DeQueue is Empty");
        }
       else {
           return (T)arr[rear];
        }
    }
    public T peekLast() {
        if (isEmpty())
        {
           return null;
        }
        else {
            return (T)arr[rear];
        }
    }

    public int getSize() {
        return size;
    }
    public boolean contains(T item) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == item) {
                return true;
            }

        }
        return false;
    }
   public boolean removeFirstOccurrence(T item)
   {
       for (int i=front;i!=rear;i=(i+1) % arr.length)
       {
           if(arr[i]!=null && arr[i].equals(item))
           {
               arr[i]=null;
               size--;
```

```
                return true;
            }
        }
        return false;
    }
    public boolean removeLastOccurrence(T item)//int i=(rear==0? arr.length-
1:rear-1);i!=front;i=(i-1+arr.length)%arr.length
    {
        for(int i=rear;i!=front;i=(i-1)%arr.length)
        {
            if(arr[i]!=null && arr[i].equals(item))
            {
                arr[i]=null;
                size--;
                return true;
            }
        }
        return false;
    }
    public void add(T item,boolean addtoFront)
    {
        if(isFull())
        {
            throw new IllegalStateException("DeQueue is Full");
        }
        else if(addtoFront)
        {
            addFirst(item);
        }
        else
        {
            addLast(item);
        }
    }
    public T remove()
    {
        return removeFirst();
    }
    public T poll()
    {
        return pollFirst();
    }
    public T element()
    {
        if (isEmpty())
        {
            throw new NoSuchElementException("DeQueue is Empty");
        }
        return getFirst();
    }
    public T peek()
    {
        return peekFirst();
    }
    public String toString()
    {
        StringBuilder sb=new StringBuilder();
```

```
        sb.append("Front-> [");
        if (!isEmpty())
        {
            int index=front;
            sb.append(arr[index]);
            index=(index+1)%arr.length;
            while (index!=rear+1)
            {
                sb.append(",").append(arr[index]);
                index=(index+1)%arr.length;
            }
        }
        sb.append("] <-Rear");
        return sb.toString();
    }
    public void deleteDequeue()
    {
        arr=new Object[arr.length];
        front=-1;
        rear=0;
        size=0;
    }

}
```

Linked Implementation

In the linked implementation of Dequeue there are nodes that containing data and each node is connected with before and next nodes excepting starting and ending nodes. This method eliminates the need for resizing and guarantees the effective insertion and deletion at both ends.

Class Node-:

```
package com.Assignment2.linkedImplementation;

public class Node<T> {
    T data;
    Node<T> next;
    Node<T> prev;

    public Node(T data) {
        this.data = data;
        this.next=null;
        this.prev=null;
    }
}
```

Class Dequeue-:

```
package com.Assignment2.linkedImplementation;
//Linked implementation of Dequeue Data Structure
import java.util.NoSuchElementException;

public class Dequeue<T>{  // T is a parameter
    private Node<T> front;
    private Node<T> rear;
    private int size;

    public Dequeue() {  //non parameterized constructor
        this.front = null;
        this.rear = null;
        size=0;

    }
    public boolean isEmpty()
    {
        return front==null;
    }  //isEmpty method
```

```
    public void addFirst(T data)  //addFirst method
    {
        Node<T> newNode=new Node<>(data);
        if (front==null){
            front=rear=newNode;
        }
        else {
            newNode.next=front;
            front.prev=newNode;
            front=newNode;
        }
        size++;
    }
    public void addLast(T data)  //addLast method
    {
        Node<T> newNode=new Node<>(data);
        if (rear==null){
            front=rear=newNode;
        }
        else {
            newNode.prev = rear;
            rear.next = newNode;
            rear = newNode;
        }
        size++;
    }
    public T removeFirst()  //removeFirst method.
    {
        if(isEmpty())
        {
            throw new IllegalStateException("Dequeue is Empty");
        }
        T data=front.data;
        front=front.next;
        if(front!=null)
        {
            front.prev=null;
        }
        else {
            rear=null;
        }
        size--;
        return data;
    }
    public T pollFirst()  //pollFirst method
    {
        if(isEmpty())
        {
           return null;
        }
        T data=front.data;
        front=front.next;
        if(front!=null)
        {
            front.prev=null;
        }
        else {
```

```
            rear=null;
        }
        size--;
        return data;
    }
    public T removeLast() //removeLast method
    {
        if (isEmpty())
        {
            throw new IllegalStateException("Dequeue is Empty");
        }
        T data=rear.data;
        rear=rear.prev;
        if(rear!=null)
        {
            rear.next=null;
        }
        else {
            front=null;
        }
        size--;
        return data;
    }
    public T pollLast() //pollLast method
    {
        if (isEmpty())
        {
            throw new IllegalStateException("Dequeue is Empty");
        }
        T data=rear.data;
        rear=rear.prev;
        if(rear!=null)
        {
            rear.next=null;
        }
        else {
            front=null;
        }
        size--;
        return data;
    }
    public T getFirst() //getFirst method
    {
        if(isEmpty())
        {
            throw new NoSuchElementException("Dequeue is Empty");
        }
        return front.data;
    }
    public T getLast() //getLast method
    {
        if (isEmpty())
        {
            throw new NoSuchElementException("Dequeue is Empty");
        }
        return rear.data;
    }
```

```
    public T peekFirst() //peekFirst method
    {
        if(isEmpty())
        {
           return null;
        }
        return front.data;
    }
    public T peekLast()
    {
        if (isEmpty())
        {
           return null;
        }
        return rear.data;
    }

    public int getSize() {
        return size;
    }
    public boolean contains(T item)
    {
        Node<T> current=front;
        while (current!=null)
        {
            if (current.data.equals(item))
            {
                return true;
            }
            current=current.next;
        }
        return false;
    }
    public boolean removeFirstOccurrence(T item)
    {
        Node<T> current=front;
        while (current!=null)
        {
            if (current.data.equals(item))
            {
                if (current==front) {
                  front=current.next;
                }
                else {
                    current.prev.next=current.next;
                    if(current.next!=null) {
                        current.next.prev = current.prev;
                    }
                }
                size--;
                return true;
            }
            current=current.next;
        }
        return false;
    }
    public boolean removeLastOccurrence(T item)
```

```
    {
        Node<T> current=rear;
        while (current!=null)
        {
            if (current.data.equals(item))
            {
                if (current==rear) {
                   rear=current.prev;
                   if(rear!=null)
                   {
                       rear.next=null;
                   }
                   else {
                       front=null;
                   }
                } else if (current==front) {
                   front=current.next;
                   if(front!=null)
                   {
                       front.prev=null;
                   }
                   else {
                       rear=null;
                   }
                }
                else {
                    current.prev.next=current.next;
                    current.next.prev=current.prev;
                }
                return true;
            }
            current=current.prev;

        }
        return false;
    }

    public void add(T item,boolean addtoFront)
    {
        if (isEmpty())
        {
            throw new IllegalStateException("Dequeue is Empty");
        } else if (addtoFront) {
            addFirst(item);
        }
        else{
            addLast(item);
        }
    }
    public T remove()
    {
        return removeFirst();
    }
    public T poll()
    {
        if (isEmpty())
        {
```

```
            return null;
        }
        return removeFirst();
    }
    public T element()
    {
        if (isEmpty())
        {
            throw new NoSuchElementException("Dequeue is Empty");
        }
        return getFirst();
    }
    public T peek()
    {
        if (isEmpty())
        {
            return null;
        }
        return peekFirst();
    }
    public String toString()
    {
        StringBuilder sb=new StringBuilder();
        sb.append("Front-> [");
        Node<T> current=front;
        while (current!=null)
        {
            sb.append(current.data);
            if (current.next!=null)
            {
                sb.append(",");
            }
            current=current.next;

        }
        sb.append("] <-Rear");
        return sb.toString();
    }
    public void deleteDequeue()
    {
        front=null;
        rear=null;
        size=0;
    }

}
```

Teamwork

- **PS/2020/010**-A.E.P.P.Jayasekara-:
  developed contiguous and linked implementation code of Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.

- **PS/2020/012**-A.M.L.Athukorala-:
  developed contiguous and linked implementation code Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.

- **PS/2020/085**-D.S.K.Wickramasinghe-:
  developed contiguous and linked implementation code Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.

- **PS/2020/024**-K.D.C.D.Fernando-:
  developed contiguous and linked implementation code Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.

- **PS/2020/005**-K.S.D.Wickramanayaka-:
  developed contiguous and linked implementation code Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.

- **PS/2020/078**-K.G.Heenatigala-:
  developed contiguous and linked implementation code Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.

- **PS/2020/220**-M.S.S.Jinadasa-:
  developed contiguous and linked implementation code Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.

- **PS/2020/095**-W.M.P.G.Wijekoon-:
  developed contiguous and linked implementation code Dequeue.
  Finding resources and requirements.
  Create dequeue data structure report.