Sandro Costa

Professor Robert Alexander

CMSC-204

10-05-2021

Pseudocode

**BasicDoubleLinkedList**

**addToFront**

DECLARE new node with parameter

IF list is empty

SET head to new node

SET tail to new node

ELSE

SET head previous node to new node

SET new nodes next node to head

SET head to new node

INCREASE size by one

RETURN this

**addToEnd**

DECLARE new node with parameter

IF list is empty

SET head to new node

SET tail to new node

ELSE

SET tail next node to new node

SET new node previous node to tail

SET tail to the new node

INCREASE size by one

RETURN this

**getFirst**

IF list is empty

Return null

Return head data

**getLast**

IF list is empty

Return null

Return tail.data

**interator**

return ListerIterator item

**hasNext**

if current is not null return true

**next**

IF the list does not have next item

THROW no such element exception

DECLARE currentData to head data

SET last moved node to current node

RETURN currentData

**hasPrevious**

IF last move is not null return true

**previous**

IF previous value is null

THROW no such element exception

SET previous data to last move data

SET current to last move

SET last move to last moves previous

RETURN previousData

**retrieveFirstElement**

IF list is empty

Return null

SET first data to head data

SET head to head next

SET head previous to null

DECRESE size by one

RETURN first data

**retrieveLastElement**

IF list is empty

RETURN null

SET last data to tail data

SET tail to tail previous

SET tail next to null

DECREASE size by one

RETURN last data

**remove**

SET current node to head

WHILE current is not null

IF current data is equal to target data

IF current is the head

SET head to head next

DECREASE list size by one

BREAK

ELSE IF current is the tail

SET Tail to Tail previous

SET tail next to null

DECREASE size by one

BREAK

ELSE

SET current previous next node to current next node

SET current next node previous node to current node previous node

DECREASE size by one

BREAK

SET current to current next

RETURN this

**toArrayList**

CREATE new ArrayList of size size

SET current node to head node

WHILE current node is not null

ADD current node to list

SET current node to current next node

RETURN ArrayList

**SortedDoubleLinkedList**

**add**

DECLARE new node with data

IF list is empty

SET head and tail node to new node

ELSE IF data is less than head node data

SET new node next node to head

SET head previous node to new node

SET head to new node

ELSE IF data is greater than tail node data

SET new node previous node to tail node

SET tail node next node to new node

SET tail node to new node

ELSE

DECLARE current node to head next

WHILE current next node is not null and current node data is less than new node data

SET current node to current next

SET new node next node to current node

SET new node previous to current previous node

SET current previous next node to new node

SET current previous node to new node

INCREASE size by one

RETURN this

import java.util.ArrayList;

import java.util.Comparator;

import java.util.Iterator;

import java.util.ListIterator;

import java.util.NoSuchElementException;

public class BasicDoubleLinkedList <T> implements Iterator<Object>{

protected Node head;

protected Node tail;

protected int size=0;

public BasicDoubleLinkedList() {

head=null;

tail=null;

}

public BasicDoubleLinkedList<T> addToEnd(T data){

Node end = new Node (data);

if(head ==null) {

head=tail=end;

}

else {

tail.next=end;

end.previous=tail;

tail=end;

}

size++;

return this;

}

public BasicDoubleLinkedList<T> addToFront(T data){

Node front = new Node(data);

if(head==null) {

head=tail=front;

size++;

return this;

}

front.next = head;

front.previous = null;

head=front;

size++;

return this;

}

public BasicDoubleLinkedList<T> remove(T targetData, Comparator<T>comparator){

Node current=head;

if(head==null)

return null;

else{

while(comparator.compare(targetData, current.data)!=0) {

current=current.next;

}

if(current==head) {

head = head.next;

size--;

return this;

}

if(current==tail) {

tail = tail.previous;

size--;

return this;

}

current.previous=current.next.previous;

current.next=current.previous;

size--;

return this;

}

}

public T getFirst() {

return head.data;

}

public T getLast() {

return tail.data;

}

public int getSize() {

return size;

}

public ListIterator<T> iterator() throws UnsupportedOperationException,NoSuchElementException {

DoubleIterator nodeIterator = new DoubleIterator();

return nodeIterator;

}

public T retrieveFirstElement() {

Node first = head;

while(first!=null){

head.previous=null;

head=head.next;

size--;

return first.data;

}

return null;

}

public T retrieveLastElement() {

Node last = tail;

while(last!=null) {

tail.next=null;

tail=tail.previous;

size--;

return last.data;

}

return null;

}

public ArrayList<T> toArrayList(){

ArrayList<T> list = new ArrayList<>();

ListIterator<T> nodeIterator = new DoubleIterator();

while(((BasicDoubleLinkedList<T>.DoubleIterator) nodeIterator).itSize()<size) {

list.add(nodeIterator.next());

}

return list;

}

class Node{

protected T data;

protected Node previous;

protected Node next;

public Node (Node previousElement, T element, Node nextElement) {

this.data=element;

this.previous=previousElement;

this.next=nextElement;

}

public Node(T element) {

this.data=element;

this.previous=null;

this.next=null;

}

}

private class DoubleIterator implements ListIterator<T>{

protected Node current = null;

private int element=0;

public DoubleIterator() {

this.current=null;

}

@Override

public boolean hasNext() {

if(current==null&&head!=null) {

return true;

}

else if(element>0){

return current.next!=null;

}

else if(element ==size) {

return false;

}

else

return false;

}

@Override

public T next() {

if(hasNext()==false) {

throw new NoSuchElementException("no element");

}

if(current==null&&head!=null) {

current=head;

element++;

return current.data;

}

else if(current!=null) {

current=current.next;

element++;

return current.data;

}

else

throw new NoSuchElementException();

}

@Override

public boolean hasPrevious() {

if(element==size)

return true;

else if(element==1) {

current=head;

return true;

}

else if(element>1) {

return current.previous!=null;

}

else

return false;

}

@Override

public T previous() {

if(hasPrevious()==false)

throw new NoSuchElementException("no element");

if(element==size) {

element--;

return current.data;

}

else if(element==1) {

element--;

return current.data;

}

else if(element>1) {

current=current.previous;

element--;

return current.data;

}

else

throw new NoSuchElementException("no element");

}

@Override

public int nextIndex() {

throw new UnsupportedOperationException();

}

@Override

public void remove() {

throw new UnsupportedOperationException();

}

@Override

public void add(T e) {

throw new UnsupportedOperationException();

}

public int itSize() {

return element;

}

@Override

public int previousIndex() {

throw new UnsupportedOperationException();

}

@Override

public void set(T e) {

throw new UnsupportedOperationException();

}

}

public boolean hasNext() {

return false;

}

public Object next() {

return null;

}

}