CS2383

Assignment #4

Student Name: Michael Drost

Student Number: 3738984

Date: November 4, 2024

"I warrant that this is my own work."

**Part A and B D updated code(Yellow is added methods and changes)**

public class LinkedPositionalList<E extends Comparable<E>> implements PositionalList<E> {

//---------------- nested Node class ----------------

/\*\*

\* Node of a doubly linked list, which stores a reference to its

\* element and to both the previous and next node in the list.

\*/

private static class Node<E> implements Position<E> {

/\*\* The element stored at this node \*/

private E element; // reference to the element stored at this node

/\*\* A reference to the preceding node in the list \*/

private Node<E> prev; // reference to the previous node in the list

/\*\* A reference to the subsequent node in the list \*/

private Node<E> next; // reference to the subsequent node in the list

/\*\*

\* Creates a node with the given element and next node.

\*

\* @param e the element to be stored

\* @param p reference to a node that should precede the new node

\* @param n reference to a node that should follow the new node

\*/

public Node(E e, Node<E> p, Node<E> n) {

element = e;

prev = p;

next = n;

}

// public accessor methods

/\*\*

\* Returns the element stored at the node.

\* @return the stored element

\* @throws IllegalStateException if node not currently linked to others

\*/

public E getElement() throws IllegalStateException {

if (next == null) // convention for defunct node

throw new IllegalStateException("Position no longer valid");

return element;

}

/\*\*

\* Returns the node that precedes this one (or null if no such node).

\* @return the preceding node

\*/

public Node<E> getPrev() {

return prev;

}

/\*\*

\* Returns the node that follows this one (or null if no such node).

\* @return the following node

\*/

public Node<E> getNext() {

return next;

}

// Update methods

/\*\*

\* Sets the node's element to the given element e.

\* @param e the node's new element

\*/

public void setElement(E e) {

element = e;

}

/\*\*

\* Sets the node's previous reference to point to Node n.

\* @param p the node that should precede this one

\*/

public void setPrev(Node<E> p) {

prev = p;

}

/\*\*

\* Sets the node's next reference to point to Node n.

\* @param n the node that should follow this one

\*/

public void setNext(Node<E> n) {

next = n;

}

} //----------- end of nested Node class -----------

// instance variables of the LinkedPositionalList

/\*\* Sentinel node at the beginning of the list \*/

private Node<E> header; // header sentinel

/\*\* Sentinel node at the end of the list \*/

private Node<E> trailer; // trailer sentinel

/\*\* Number of elements in the list (not including sentinels) \*/

private int size = 0; // number of elements in the list

private int nodesV = 0;

private int numCompare = 0;

/\*\* Constructs a new empty list. \*/

public LinkedPositionalList() {

header = new Node<>(null, null, null); // create header

trailer = new Node<>(null, header, null); // trailer is preceded by header

header.setNext(trailer); // header is followed by trailer

}

// private utilities

/\*\*

\* Verifies that a Position belongs to the appropriate class, and is

\* not one that has been previously removed. Note that our current

\* implementation does not actually verify that the position belongs

\* to this particular list instance.

\*

\* @param p a Position (that should belong to this list)

\* @return the underlying Node instance at that position

\* @throws IllegalArgumentException if an invalid position is detected

\*/

private Node<E> validate(Position<E> p) throws IllegalArgumentException {

if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");

Node<E> node = (Node<E>) p; // safe cast

if (node.getNext() == null) // convention for defunct node

throw new IllegalArgumentException("p is no longer in the list");

return node;

}

/\*\*

\* Returns the given node as a Position, unless it is a sentinel, in which case

\* null is returned (so as not to expose the sentinels to the user).

\*/

private Position<E> position(Node<E> node) {

if (node == header || node == trailer)

return null; // do not expose user to the sentinels

return node;

}

// public accessor methods

/\*\*

\* Returns the number of elements in the list.

\* @return number of elements in the list

\*/

@Override

public int size() { return size; }

/\*\*

\* Tests whether the list is empty.

\* @return true if the list is empty, false otherwise

\*/

@Override

public boolean isEmpty() { return size == 0; }

/\*\*

\* Returns the first Position in the list.

\*

\* @return the first Position in the list (or null, if empty)

\*/

@Override

public Position<E> first() {

return position(header.getNext());

}

/\*\*

\* Returns the last Position in the list.

\*

\* @return the last Position in the list (or null, if empty)

\*/

@Override

public Position<E> last() {

return position(trailer.getPrev());

}

/\*\*

\* Returns the Position immediately before Position p.

\* @param p a Position of the list

\* @return the Position of the preceding element (or null, if p is first)

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public Position<E> before(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return position(node.getPrev());

}

/\*\*

\* Returns the Position immediately after Position p.

\* @param p a Position of the list

\* @return the Position of the following element (or null, if p is last)

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public Position<E> after(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return position(node.getNext());

}

// private utilities

/\*\*

\* Adds an element to the linked list between the given nodes.

\* The given predecessor and successor should be neighboring each

\* other prior to the call.

\*

\* @param pred node just before the location where the new element is inserted

\* @param succ node just after the location where the new element is inserted

\* @return the new element's node

\*/

private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {

Node<E> newest = new Node<>(e, pred, succ); // create and link a new node

pred.setNext(newest);

succ.setPrev(newest);

size++;

return newest;

}

// public update methods

/\*\*

\* Inserts an element at the front of the list.

\*

\* @param e the new element

\* @return the Position representing the location of the new element

\*/

@Override

public Position<E> addFirst(E e) {

return addBetween(e, header, header.getNext()); // just after the header

}

/\*\*

\* Inserts an element at the back of the list.

\*

\* @param e the new element

\* @return the Position representing the location of the new element

\*/

@Override

public Position<E> addLast(E e) {

return addBetween(e, trailer.getPrev(), trailer); // just before the trailer

}

/\*\*

\* Inserts an element immediately before the given Position.

\*

\* @param p the Position before which the insertion takes place

\* @param e the new element

\* @return the Position representing the location of the new element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public Position<E> addBefore(Position<E> p, E e)

throws IllegalArgumentException {

Node<E> node = validate(p);

return addBetween(e, node.getPrev(), node);

}

/\*\*

\* Inserts an element immediately after the given Position.

\*

\* @param p the Position after which the insertion takes place

\* @param e the new element

\* @return the Position representing the location of the new element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public Position<E> addAfter(Position<E> p, E e)

throws IllegalArgumentException {

Node<E> node = validate(p);

return addBetween(e, node, node.getNext());

}

/\*\*

\* Replaces the element stored at the given Position and returns the replaced element.

\*

\* @param p the Position of the element to be replaced

\* @param e the new element

\* @return the replaced element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public E set(Position<E> p, E e) throws IllegalArgumentException {

Node<E> node = validate(p);

E answer = node.getElement();

node.setElement(e);

return answer;

}

/\*\*

\* Returns (but does not remove) the element at index i.

\* @param i the index of the element to return

\* @return the element at the specified index

\* @throws IndexOutOfBoundsException if the index is negative or greater than size()-1

\*/

public E get(int i) throws IndexOutOfBoundsException{

if (i<0 || i>=size){

throw new IndexOutOfBoundsException("index is larger than the list");

}

Node<E> current = header.next;

for (int index =0; index<i;index++){

current = current.next;

nodesV++;

}

return current.getElement();

}

/\*\*

\* Replaces the element at the specified index, and returns the element previously stored.

\* @param i the index of the element to replace

\* @param e the new element to be stored

\* @return the previously stored element

\* @throws IndexOutOfBoundsException if the index is negative or greater than size()-1

\*/

public E set(int i, E e) throws IndexOutOfBoundsException{

if (i<0 || i>size-1){

throw new IndexOutOfBoundsException("index is larger than the list");

}

Node<E> current = header;

for (int index =0; index<i;i++){

current = current.next;

}

E old = current.element;

current.element = e;

return old;

}

/\*\*

\* Inserts the given element at the specified index of the list, shifting all

\* subsequent elements in the list one position further to make room.

\* @param i the index at which the new element should be stored

\* @param e the new element to be stored

\* @throws IndexOutOfBoundsException if the index is negative or greater than size()

\*/

public void add(int i, E e) throws IndexOutOfBoundsException{

if (i<0 || i>size-1)

throw new IndexOutOfBoundsException("index is larger than the List");

if (i==0){

Node<E> newNode = new Node<>(e,header,header.next);

header.next = newNode;

newNode.next.prev = newNode;

size++;

}else if(i== size-1){

Node<E> newNode = new Node<>(e,trailer.prev,trailer);

trailer.prev = newNode;

newNode.prev.next = newNode;

size++;

}else{

Node<E> newNode = new Node<>(e,null,null);

Node<E> current = header;

for (int index =0;index<i;index++){

current = current.next;

}

newNode.prev = current.prev;

newNode.next = current;

current.prev = newNode;

}

}

/\*\*

\* Removes and returns the element at the given index, shifting all subsequent

\* elements in the list one position closer to the front.

\* @param i the index of the element to be removed

\* @return the element that had be stored at the given index

\* @throws IndexOutOfBoundsException if the index is negative or greater than size()

\*/

public E remove(int i) throws IndexOutOfBoundsException{

if (i<0 || i>size-1)

throw new IndexOutOfBoundsException("index is larger than the List");

Node<E> current = header;

for(int index =0;index<i;index++){

current = current.next;

}

current.prev.next = current.next;

current.next.prev = current.prev;

E removed = current.element;

current.element = null;

current.next = null;

current.prev =null;

return removed;

}

public int binarySearch(E target){

int left =0;

int right = size-1;

numCompare = 0;

nodesV = 0;

while (left <= right) {

int middle = left + (right - left) / 2;

int compare = get(middle).compareTo(target);

numCompare++;

if (compare == 0) {

return middle;

} else if (compare < 0) {

left = middle + 1;

} else {

right = middle - 1;

}

}

return -1; //YOU BETTER FUCKING WORK.

}

/\*\*

\* Removes the element stored at the given Position and returns it.

\* The given position is invalidated as a result.

\*

\* @param p the Position of the element to be removed

\* @return the removed element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public E remove(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

Node<E> predecessor = node.getPrev();

Node<E> successor = node.getNext();

predecessor.setNext(successor);

successor.setPrev(predecessor);

size--;

E answer = node.getElement();

node.setElement(null); // help with garbage collection

node.setNext(null); // and convention for defunct node

node.setPrev(null);

return answer;

}

// support for iterating either positions and elements

//---------------- nested PositionIterator class ----------------

/\*\*

\* A (nonstatic) inner class. Note well that each instance

\* contains an implicit reference to the containing list,

\* allowing us to call the list's methods directly.

\*/

private class PositionIterator implements Iterator<Position<E>> {

/\*\* A Position of the containing list, initialized to the first position. \*/

private Position<E> cursor = first(); // position of the next element to report

/\*\* A Position of the most recent element reported (if any). \*/

private Position<E> recent = null; // position of last reported element

/\*\*

\* Tests whether the iterator has a next object.

\* @return true if there are further objects, false otherwise

\*/

public boolean hasNext() { return (cursor != null); }

/\*\*

\* Returns the next position in the iterator.

\*

\* @return next position

\* @throws NoSuchElementException if there are no further elements

\*/

public Position<E> next() throws NoSuchElementException {

if (cursor == null) throw new NoSuchElementException("nothing left");

recent = cursor; // element at this position might later be removed

cursor = after(cursor);

return recent;

}

/\*\*

\* Removes the element returned by most recent call to next.

\* @throws IllegalStateException if next has not yet been called

\* @throws IllegalStateException if remove was already called since recent next

\*/

public void remove() throws IllegalStateException {

if (recent == null) throw new IllegalStateException("nothing to remove");

LinkedPositionalList.this.remove(recent); // remove from outer list

recent = null; // do not allow remove again until next is called

}

} //------------ end of nested PositionIterator class ------------

//---------------- nested PositionIterable class ----------------

private class PositionIterable implements Iterable<Position<E>> {

public Iterator<Position<E>> iterator() { return new PositionIterator(); }

} //------------ end of nested PositionIterable class ------------

/\*\*

\* Returns an iterable representation of the list's positions.

\* @return iterable representation of the list's positions

\*/

@Override

public Iterable<Position<E>> positions() {

return new PositionIterable(); // create a new instance of the inner class

}

//---------------- nested ElementIterator class ----------------

/\* This class adapts the iteration produced by positions() to return elements. \*/

private class ElementIterator implements Iterator<E> {

Iterator<Position<E>> posIterator = new PositionIterator();

public boolean hasNext() { return posIterator.hasNext(); }

public E next() { return posIterator.next().getElement(); } // return element!

public void remove() { posIterator.remove(); }

}

/\*\*

\* Returns an iterator of the elements stored in the list.

\* @return iterator of the list's elements

\*/

@Override

public Iterator<E> iterator() { return new ElementIterator(); }

// Debugging code

/\*\*

\* Produces a string representation of the contents of the list.

\* This exists for debugging purposes only.

\*/

public String toString() {

StringBuilder sb = new StringBuilder("(");

Node<E> walk = header.getNext();

while (walk != trailer) {

sb.append(walk.getElement());

walk = walk.getNext();

if (walk != trailer)

sb.append(", ");

}

sb.append(")");

return sb.toString();

}

**Part C)**

I believe the Big O for binarySearch is O(n log n ). Like mentioned the binarySearch itself is O(log n) as it cuts the results in half everytime, however with the use of the get method for the linked list this multiplies O(n) time complexity for every time the method is called, due to the get method needing to loop through the linked list to find the node with the correct element, making making the complexity O(log n) x O(n) making the final time complexity O(n log n).

Part D) TESTIGN CODE

public static void main (String args[]){

LinkedPositionalList<Integer> test = new LinkedPositionalList<Integer>();

final int from = 5;

int to = 100;

System.out.println("From\tto\tTarget\tn\tlogN\tNlogN\t#Iters\t#Nodes");

System.out.println("====\t==\t======\t=\t====\t=====\t======\t======");

for(int j =0;j<1000;j+=100){

for (int i = 5+j;i<=to;i+=5){

test.addLast(i);

}

test.binarySearch(3);

int iters = test.numCompare;

int nodes = test.nodesV;

int N = test.size();

int log2N = (int)(Math.log(N) / Math.log(2));

int nLog2N = N \* log2N;

System.out.println(from+"\t"+to+"\t3\t"+N+"\t"+log2N+"\t"+nLog2N+"\t"+iters+"\t"+nodes);//add math stuff still

to+=100;

}

LinkedPositionalList<Integer> test2 = new LinkedPositionalList<Integer>();

to = 100;

System.out.println("\nFrom\tto\tTarget\tn\tlogN\tNlogN\t#Iters\t#Nodes");

System.out.println("====\t==\t======\t=\t====\t=====\t======\t======");

for(int j =0;j<1000;j+=100){

for (int i = 5+j;i<=to;i+=5){

test2.addLast(i);

}

test2.binarySearch(to-3);

int iters = test2.numCompare;

int nodes = test2.nodesV;

int N = test2.size();

int log2N = (int)(Math.log(N) / Math.log(2));

int nLog2N = N \* log2N;

System.out.println(from+"\t"+to+"\t"+(to-3)+"\t"+N+"\t"+log2N+"\t"+nLog2N+"\t"+iters+"\t"+nodes);//add math stuff still

to+=100;

}

LinkedPositionalList<Integer> test3 = new LinkedPositionalList<Integer>();

to = 100;

System.out.println("\nFrom\tto\tTarget\tn\tlogN\tNlogN\t#Iters\t#Nodes");

System.out.println("====\t==\t======\t=\t====\t=====\t======\t======");

for(int j =0;j<1000;j+=100){

for (int i = 5+j;i<=to;i+=5){

test3.addLast(i);

}

test3.binarySearch((to/2)-1);

int iters = test3.numCompare;

int nodes = test3.nodesV;

int N = test3.size();

int log2N = (int)(Math.log(N) / Math.log(2));

int nLog2N = N \* log2N;

System.out.println(from+"\t"+to+"\t"+((to/2)-1)+"\t"+N+"\t"+log2N+"\t"+nLog2N+"\t"+iters+"\t"+nodes);//add math stuff still

to+=100;

}

}

**OUTPUT OF CODE:**

e2c2k\_6a933c31/bin As4.LinkedPositionalList

From to Target n logN NlogN #Iters #Nodes

==== == ====== = ==== ===== ====== ======

5 100 3 20 4 80 4 14

5 200 3 40 5 200 5 33

5 300 3 60 5 300 5 51

5 400 3 80 6 480 6 72

5 500 3 100 6 600 6 91

5 600 3 120 6 720 6 110

5 700 3 140 7 980 7 130

5 800 3 160 7 1120 7 151

5 900 3 180 7 1260 7 169

5 1000 3 200 7 1400 7 190

From to Target n logN NlogN #Iters #Nodes

==== == ====== = ==== ===== ====== ======

5 100 97 20 4 80 5 77

5 200 197 40 5 200 6 196

5 300 297 60 5 300 6 298

5 400 397 80 6 480 7 475

5 500 497 100 6 600 7 596

5 600 597 120 6 720 7 717

5 700 697 140 7 980 8 975

5 800 797 160 7 1120 8 1114

5 900 897 180 7 1260 8 1256

5 1000 997 200 7 1400 8 1395

From to Target n logN NlogN #Iters #Nodes

==== == ====== = ==== ===== ====== ======

5 100 49 20 4 80 5 34

5 200 99 40 5 200 6 93

5 300 149 60 5 300 6 144

5 400 199 80 6 480 7 232

5 500 249 100 6 600 7 291

5 600 299 120 6 720 7 353

5 700 349 140 7 980 8 479

5 800 399 160 7 1120 8 551

5 900 449 180 7 1260 8 620

5 1000 499 200 7 1400 8 690

(base) michaeldrost@Michaels-Air e2c2k %

**Explanation:**

The iterations among all three tests are consistent with O(log n). the number of iterations stays low and consistent, However the #Nodes is very different especially when list size is increased we see the # of nodes searched at its highest on the second test when the target is near the end of the list meaning that the # of nodes is reliant on the size of N making this part O(N) with both parts together this makes it O(log n) x O(N). And the final time complexity aligns with my observation from part C where I described the time complexity as O(N log N).