Week 6 worksheet: Complexity Analysis and Generics

Total points: 10

Out: February 25 (Tuesday)

Due: February 28 (Friday end of day [2359 CDT according to D2L])

## What to submit?

Upload only one Word or PDF file to the designated D2L folder.

## Running time assumptions

* Assume that basic arithmetic operations (+, -, \*, /, %) run in constant time.
* Assume that the System.out.println( … ) statement runs in constant time.

## Generic Node <E> assumptions

* Assume that generic class E has an equals method.

# Exercise 1: Elementary Sorts

[2 pts] Consider the following array. You will sort the array from least to greatest.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **6** | **4** | **9** | **7** |

Part A: Sort the array using insertion sort, showing the array after each insertion.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **6** | **4** | **9** | **7** |
| 2 | 4 | 6 | 9 | 7 |
| 2 | 4 | 6 | 7 | 9 |
|  |  |  |  |  |

Part B: Sort the array using selection sort, showing the array after each selection.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **6** | **4** | **9** | **7** |
| 2 | 4 | 6 | 9 | 7 |
| 2 | 4 | 6 | 7 | 9 |
|  |  |  |  |  |

# Exercise 2: Implement removeFirstTarget

[4 pts] Suppose that you have a linked list of Node<E>s of generic objects as part of a linked list of generic objects:

public class Node<E> {

E data;

Node<E> link;

public Node<E>( E data, Node<E> next ) {

this.data = data;

this.link = next;

}

public Node<E>() {

this.data = null;

this.link = null;

}

public E getData() { return data; }

public Node<E> getLink() { return link; }

public void setData( E data ) { this.data = data; }

public void setLink( Node<E> link ) { this.link = link; }

}

Suppose further you have a linked list with a head Node<E> named “head”.

Suppose further that class E implements the Comparable interface.

Write a method removeFirstTarget(…) that removes the first node of the linked list that contains a value “target” of type E. Of course, your method must then reconnect the list. If there is no such element in the list, return null. The signature of this method is

public static <E> E removeFirst( E target, Node<E> head ){

// Check if the head is null

if (head == null) {

return null; // List is empty

}

Node<E> current = head;

Node<E> previous = null;

// Traverse the linked list

while (current != null) {

// Check if the current node's data matches the target

if (current.getData().equals(target)) {

// If the node to remove is the head

if (previous == null) {

// Update head to the next node

head = current.getLink();

} else {

// Bypass the current node

previous.setLink(current.getLink());

}

// Return the data of the removed node

return current.getData();

}

// Move to the next node

previous = current;

current = current.getLink();

}

// If we reach here, the target was not found

return null;

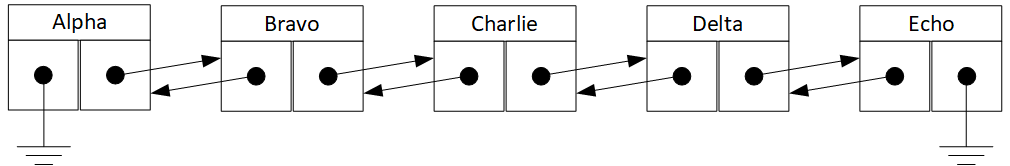
}

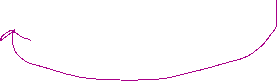
# Exercise 3: Doubly linked list removeFirst

[4 pts] Suppose that you have a doubly linked list of Nodes of generic objects. Each Node<E> has two references, one to the previous element in the list and one to the next element. It has two named nodes, “head” and “tail”. “head” has a null “prev” reference, and “tail” has a null “next” reference.

A sample such list with Strings would look like this:







public class Node<E> {



E data;

Node<E> prev;

Node<E> next;

public E getData() { return data; }

public Node<E> getPrev() { return prev; }

public Node<E> getNext() { return next; }

public void setData( E data ) { this.data = data; }

public void setPrev ( Node<E> prev ) { this.prev = prev; }

public void setNext ( Node<E> next ) { this.prev = next; }

}

Write a method removeLast that removes the last node of the linked list that contains a value “target” of type E, and then reconnects the list. If there is no such element in the list, return null. The signature of this method is

public E removeLast( E target, Node<E> head, Node<E> tail ){

Node<E> current = tail; // Start from the tail of the list

// Traverse the list backwards until we find the target or reach the head

while (current != null) {

if (current.getData().equals(target)) {

// Found the target, now remove this node

Node<E> prevNode = current.getPrev();

Node<E> nextNode = current.getNext();

// Adjust the pointers of the previous and next nodes

if (prevNode != null) {

prevNode.setNext(nextNode);

} else {

// If there is no previous node, we are removing the head

head = nextNode; // Update head if necessary

}

if (nextNode != null) {

nextNode.setPrev(prevNode);

} else {

// If there is no next node, we are removing the tail

tail = prevNode; // Update tail if necessary

}

return target; // Return the removed target

}

current = current.getPrev(); // Move to the previous node

}

return null; // Return null if the target was not found

}