DD1339 Introduktion till datalogi 2013/2014

Uppgift nummer: 2

Namn: Marcus Larsson

Grupp nummer: 5

Övningsledare: Marcus Dicander

Betyg: ..... Datum: .... Rättad av: ....

Exercise 9.11

Device måste ha getName() eftersom objektet vi jobbar med för tillfället är en device och inte nödvändigtvis en printer. Därför letar Java-kompileraren efter metoden i Device. Eftersom vi måste garantera att metoden alltid ska finnas.

Exercise 9.12

Igen så jobbar vi med objektet av typen Device. Men när koden körs så kommer det objektet vi hanterar att vara en Printer. Den har en metod som overridar superklassens metod. Därför kommer metoden i Printer att exekveras.

Exercise 9.13

Eftersom alla klasser i java får Object som superclass utan att man specificerar det, så kommer toString() från Object att anropas. Så ja, koden kompilerar och toString() från klassen Object skriver ut en textrepresentation av objektet med på formatet:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Exercise 9.14

Ja den koden kompilerar också av samma anledning som i 9.13. När man anropar System.out.prinln(st); Så kommer anropet att hämta det som ska skrivas från st.toString(). Och detta hämtas från klassen Object.

Exercise 9.15

Ja koden kompilerar. Eftersom Student även har Object som superclass så är tilldelningen OK. Precis som i fråga 9.12 så måste Object ha toString() deklarerat, men när vi kör programmet så hanterar vi objektet som ett Object. Men när vi ber den köra toString() så kommer den köra sin egen toString() vilket är den i Student som overridar den i Object. Så metoden kommer skriva ut namnen på studenterna i listan.

Exercise 9.16

För att åstadkomma situationen ska vi delkarera varabeln x som T i koden och tilldela den ett objekt av typed D.

T x = new D();

Exercise web

Ordonotation

Alla metoder i LinkedList körs på konstant tid förutom isHealthy(), get(index) och toString(). Dessa tre går igenom hela listan 1 gång och har därför O(n) som tidskomplexitet.

I testklassen LinkedListTest testas metoderna med 0,1,2 och 3 element i listan förutom i testSize(). Därför har alla metoder konstant tidskomplexitet då antalet operationer ej varierar. De beror av antalet element, men eftersom antalet element är fast satta i testerna blir komplexiteten konstant.

testSize() har tidskomplexitet O(n) där n är antalet element som testas. Talet n genereras fram vid varje test till ett tal mellan 1 och 100000.

LinkedList source code

```
/**
 * A singly linked list.
 *
 * @author Marcus Larsson
 * @version 2014-01-27
 */
public class LinkedList<T> {
   private ListElement<T> first;   // First element in list.
   private ListElement<T> last;    // Last element in list.
   private int size;               // Number of elements in list.

   /**
    * A list element.
    */
   private static class ListElement<T> {
      public T data;
      public ListElement<T> next;

      public ListElement(T data) {
         this.data = data;
         this.next = null;
      }
   }

   /**
    * This TEST METHOD returns true if the following invariants hold:
    * <ul>
    *   <li> size equals the number of list elements, </li>
    *   <li> if size == 0, first == null and last == null, </li>
    *   <li> if size > 0, first != null and last != null, </li>
    *   <li> if size == 1, first == last, </li>
    *   <li> last.next == null. </li>
    * </ul>
    *
    * Computes the test in time complexity O(n).
    */
   public boolean isHealthy() {
      //checks so that the size matches the number of elements in the list.
      int numOfElements = 0;
      ListElement<T> current = first;
      while(current!=null){
         numOfElements++;
         current=current.next;
      }
      if(numOfElements!=size){
         return false;
      }

      //Test that if size==0 then first==null and last==null
      if(size==0 && (first!=null || last!=null)){
         return false;
      }

      // checks so that if size>0 then first and last is not null
      if(size>0 && (first==null || last==null)){
         return false;
      }

      //checks so that if size==1 then first==last
      if(size==1 && (first!=last)){
         return false;
      }

      //checks so that if last element exists it always has it's nextpointer equal to null.
      if(last!=null && last.next!=null){
         return false;
      }

      //if method made it here it passed all the tests and it will return true.
      return true;
   }

   /**
    * Creates an empty list.
    * Computes at constant time.
    */
   public LinkedList() {
   }

   /**
    * Inserts the given element at the beginning of this list and increment size.
    * Computes at constant time.
    */
   public void addFirst(T element) {
      ListElement<T> currentlyFirst = this.first;
      this.first= new ListElement<T>(element);
      this.first.next = currentlyFirst;
      if(this.last==null){
         this.last=this.first;
      }
      this.size++;
   }

   /**
    * Inserts the given element at the end of this list and increment size.
    * Computes at constant time.
    */
   public void addLast(T element) {
      ListElement<T> temp = new ListElement<T>(element);
      //Invariant last==first if size is 1 and both null if size is 0 means I only have to check if last is null.
      if(this.last==null){
         this.first=temp;
         this.last=temp;
      }else{
         last.next = temp;
         last=temp;
      }
      size++;
   }

   /**
    * Returns the first element of this list.
    * Returns <code>null</code> if the list is empty.
    * Computes at constant time.
    */
   public T getFirst() {
      if(first!=null){
         return first.data;
      }
      return null;
   }

   /**
    * Returns the last element of this list.
    * Returns <code>null</code> if the list is empty.
    * Computes at constant time.
    */
   public T getLast() {
      if(last!=null){
         return last.data;
      }
      return null;
   }

   /**
    * Returns the element at the specified position in this list.
    * Returns <code>null</code> if <code>index</code> is out of bounds.
    * Computes at time complexity O(n).
    */
   public T get(int index) {
      if(this.first!=null && index>=0){
         ListElement<T> current = this.first;
         int count = 0;
         while(count<index){
            current=current.next;
            count++;
            if(current==null){
                   return null;
               }
           }
           return current.data;
       }
       return null;
   }

   /**
    * Removes and returns the first element from this list and decrement size.
    * Returns <code>null</code> if the list is empty.
    * Computes at constant time.
    */
   public T removeFirst() {
       if(first!=null){
           T temp = first.data;
           if(size==1){
               first=null;
               last=null;
           }else{
               first=first.next;
           }
           size--;
           return temp;
       }
       return null;
   }

   /**
    * Removes all of the elements from this list.
    * Computes at constant time
    */
   public void clear() {
       // Only need to remove first and last and set size to 0. Because when first is removed, nothing
   points to second anymore and GC will take care of it.
       // Then it's the same for third etc. all the way to last. But last has an extra pointer from this.last,
   so that one is set to null manually.
       this.size=0;
       this.first=null;
       this.last=null;
   }

   /**
    * Returns the number of elements in this list.
    * Computes at constant time.
    */
   public int size() {
       return this.size;
   }

   /**
    * Returns <code>true</code> if this list contains no elements.
```

```
     * Computes at constant time
     */
    public boolean isEmpty() {
        if(first==null && last==null && size==0){
            return true;
        }
        return false;
    }

    /**
     * Returns a string representation of this list. The string
     * representation consists of a list of the elements enclosed in
     * square brackets ("[]"). Adjacent elements are separated by the
     * characters ", " (comma and space). Elements are converted to
     * strings by the method toString() inherited from Object.
     * Computes at time complexity O(n).
     */
    public String toString() {
        String result = "[";
        ListElement<T> element = first;

        for(int i=0;i<size;i++){
            if(element.data==null){
                result+="null";
            } else {
                result+=element.data;
            }
            if(element!=last){
                result+=", ";
            }
            element=element.next;
        }
        result+="]";
        return result;
    }
}
```

LinkedListTest source code

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.util.Random;

/**
 * The test class LinkedListTest. This class is constructed to test all public methods in LinkedList.
 *
 * @author  Marcus Larsson
 * @version 2014-01-24
 */
public class LinkedListTest
{
    /**
     * Tests the constructor in LinkedList.
     * Computes at constant time.
     */
    @Test
    public void testLinkedList(){
        LinkedList<String> list1 = new LinkedList<String>();
        assertTrue(list1.isEmpty());

        LinkedList list2 = new LinkedList(); //The type in the list will be Object.
        assertTrue(list2.isEmpty());

        assertTrue(list1.isHealthy());
        assertTrue(list2.isHealthy());
    }

    /**
     * Tests to add elements to the beginning of the list and checks so that the list is still healthy
between adding elements.
     * Computes at constant time.
     *
     */
    @Test
    public void testAddFirst(){
        LinkedList<String> list = new LinkedList<String>();

        //add element to empty list
        String s1 = "test";
        list.addFirst(s1);
        assertSame(s1, list.getFirst());
        assertTrue(list.isHealthy());

        //add element to list with 1 element
        String s2 = "test2";
        list.addFirst(s2);
        assertSame(s2, list.getFirst());
        assertTrue(list.isHealthy());

        //add element to list with 2 elements(more than 2 will behave the same since there are no more
variables changed.)
        String s3 = "test3";
        list.addFirst(s3);
        assertSame(s3, list.getFirst());
        assertTrue(list.isHealthy());

        //test to add NULL (list should still be healthy since only the value null is added in the first node.)
        list.addFirst(null);
        assertTrue(list.isHealthy());

    }

    /**
```

```
     * Tests to add elements to the end of the list and checks so that the list is still healthy between
adding elements.
     * Computes at constant time.
     */
    @Test
    public void testAddLast(){
        LinkedList<String> list = new LinkedList<String>();

        //add element to empty list
        String s1 = "test";
        list.addLast(s1);
        assertSame(s1, list.getLast());
        assertTrue(list.isHealthy());

        //add element to list with 1 element
        String s2 = "test2";
        list.addLast(s2);
        assertSame(s2, list.getLast());
        assertTrue(list.isHealthy());

        //add element to list with 2 elements(more than 2 will behave the same since there are no more
variables changed.)
        String s3 = "test3";
        list.addLast(s3);
        assertSame(s3, list.getLast());
        assertTrue(list.isHealthy());

        //test to add NULL (list should still be healthy since only the value null is added in the first node.)
        list.addLast(null);
        assertTrue(list.isHealthy());

    }

    /**
     * Test to get the first element in the list and checks so that it's the correct element.
     * Computes at constant time.
     */
    @Test
    public void testGetFirst(){
        LinkedList<String> list = new LinkedList<String>();
        //test so that it returns null when list is empty
        assertNull(list.getFirst());
        String test = "test";
        list.addFirst(test);
        //checks so that the element added is the first element.
        assertSame(test, list.getFirst());

    }

    /**
     * Test to get the last element in the list and checks so that it's the correct element.
     * Computes at constant time.
     */
    @Test
    public void testGetLast(){
        LinkedList<String> list = new LinkedList<String>();
        //test so that it returns null when list is empty
        assertNull(list.getLast());
        String test = "test";
        list.addLast(test);
        //checks so that the element added is the last element.
        assertSame(test, list.getLast());

    }

    /**
     * Test to get elements on a specific index in the list. Testing out of bounds index and valid indexes.
     * Computes at constant time.
     */
    @Test
    public void testGet(){
        LinkedList<String> list = new LinkedList<String>();
        //test the empty list first. All values should be out of bounds and return null
        assertNull(list.get(0));
        assertNull(list.get(1));
        assertNull(list.get(-1));

        //add elements to list
        String s1 = "1";
        String s2 = null;
        String s3 = "3";
        String s4 = "4";

        list.addLast(s1);
        list.addLast(s2);
        list.addLast(s3);
        list.addLast(s4);
        assertSame(s1, list.get(0));
        assertNull(list.get(1));
        assertSame(s3, list.get(2));
        assertNull(list.get(10));
        assertNull(list.get(-1));

    }

    /**
     * Test to remove the fist element in the list. Checks so that correct element is removed and that
the list is still healthy afterwards.
     * Computes at constant time.
     */
    @Test
    public void testRemoveFirst(){
        //need to test the method for 4 different lists. size=0,1,2,and>=3
        String s1 ="a";
        String s2 ="b";
        String s3 ="c";

        LinkedList<String> list0 = new LinkedList<String>();
        assertNull(list0.removeFirst());

        LinkedList<String> list1 = new LinkedList<String>();
        list1.addFirst(s1);
        assertSame(s1, list1.removeFirst());
        assertNotSame(s1, list1.getFirst());
        assertTrue(list1.isHealthy());


        LinkedList<String> list2 = new LinkedList<String>();
        list2.addFirst(s1);
        list2.addFirst(s2);
        assertSame(s2, list2.removeFirst());
        assertNotSame(s2, list2.getFirst());
        assertTrue(list1.isHealthy());

        LinkedList<String> list3 = new LinkedList<String>();
        list3.addFirst(s1);
        list3.addLast(s2);
        list3.addLast(s3);
        assertSame(s1, list3.removeFirst());
        assertNotSame(s1, list3.getFirst());
        assertTrue(list1.isHealthy());

    }

    /**
     * Tests to clear a list. Checks so that the list is really empty afterwards and still healthy.
     * Computes at constant time.
     */
    @Test
    public void testClear(){
        LinkedList<String> list = new LinkedList<String>();
        list.addFirst("a");
        list.addFirst("b");
        list.addFirst("c");
        list.clear();
        assertTrue(list.isEmpty());
        assertTrue(list.isHealthy());

    }

    /**
     * Tests so that the size variable is correct in the list. Adds a random number of elements and
checks so that it matches with the size variable in the list.
     * Computes at time complexity O(n) where n is the number of elements randomly chosen from 1
to 100000.
     */
    @Test
    public void testSize(){
        LinkedList<String> list = new LinkedList<String>();

        //test so that empty list has size 0.
        assertEquals(0, list.size());
        list.removeFirst();
        assertEquals(0, list.size());

        //test a random number of elements is correct.
        int numOfElements = (new Random().nextInt(100000))+1; //random between 1 and 100000
        int count=0;
        while(count<numOfElements){
            //test to add elements both first and last.
            if(count%2==0){
                list.addFirst("test"+count);
            }else{
                list.addLast("test"+count+1);
            }
            count++;
        }
        assertEquals(numOfElements, list.size());

        //test to remove elements and makes sure the size is still correct.
        list.removeFirst();
        assertEquals(numOfElements-1, list.size());

    }

    /**
     * Tests if method isEmpty is working correctly.
     * Computes at constant time.
     */
    @Test
    public void testIsEmpty(){
        LinkedList<String> list = new LinkedList<String>();
        assertTrue(list.isEmpty());
        list.addFirst("a");
        assertFalse(list.isEmpty());
        list.removeFirst();
        assertTrue(list.isEmpty());
        assertTrue(list.isHealthy());
    }

    /**
     * Tests the toString method of the list so that it computes the promised answer according to
documentation.
     * Computes at constant time.
     */
    @Test
    public void testToString(){
        LinkedList<String> list = new LinkedList<String>();
        assertEquals("[]", list.toString());

        //add elements
        String s1 = "test1";
        String s2 = "test2";

        //test several elements and null value
        list.addFirst(s1);
        assertEquals("[test1]", list.toString());
        list.addFirst(null);
        assertEquals("[null, "+s1+"]", list.toString());
        list.addFirst(s2);
        assertEquals("["+s2+", null, "+s1+"]", list.toString());

    }
}
```