Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №9
29.04.2010г.

ТЕМА: Свързана реализация на линеен списък.
Итератори

Задача 1: Да се реализира итератор за класа, представящ двойно свързан списък.

```
/**
 * An iterator for lists that allows the programmer
 * to traverse the list in either direction, modify
 * the list during iteration, and obtain the iterator's
 * current position in the list.
 */
public interface ListIterator<E> extends Iterator<E> {

    /**
     * Returns true if this list iterator has more elements when
     * traversing the list in the forward direction.
     */
    boolean hasNext();

    /**
     * Returns the next element in the list and advances the cursor position.
     * This method may be called repeatedly to iterate through the list,
     * or intermixed with calls to previous to go back and forth.
     */
    E next();

    /**
     * Returns true if this list iterator has more elements when
     * traversing the list in the reverse direction.
     */
    boolean hasPrevious();

    /**
     * Returns the previous element in the list and moves the cursor
     * position backwards.  This method may be called repeatedly to
     * iterate through the list backwards, or intermixed with calls to
     * next to go back and forth.
     */
    E previous();
```

```
    /**
     * Returns the index of the element that would be returned by a
     * subsequent call to next. (Returns list size if the list
     * iterator is at the end of the list.)
    */
     int nextIndex();

    /**
     * Returns the index of the element that would be returned by a
     * subsequent call to previous. (Returns -1 if the list
     * iterator is at the beginning of the list.)
     */
    int previousIndex();

    /**
     * Removes from the list the last element that was returned by
     * next or previous (optional operation).  This call can
     * only be made once per call to next or  previous.
     */
    void remove();

    /**
     * Replaces the last element returned by next or
     * previous with the specified element (optional operation).
     * This call can be made only if neither remove nor
     * add have been called after the last call to  next or previous.
     */
    void set(E e);

    /**
     * Inserts the specified element into the list (optional operation).
     * The element is inserted immediately before the next element that
     * would be returned by next, if any, and after the next
     * element that would be returned by previous, if any.  (If the
     * list contains no elements, the new element becomes the sole element
     * cursor: a subsequent call to next would be unaffected, and a
     * subsequent call to  previous would return the new element.
     */
    void add(E e);
}
```

```
import java.io.Serializable;
import java.util.Collection;
import java.util.Deque;
import java.util.Queue;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.List;
import java.util.NoSuchElementException;

public class LinkedList<E> implements Serializable, Cloneable, Iterable<E>,
                                      Collection<E>, Deque<E>, List<E>, Queue<E> {

    /* Class to represent a node of the List */
    protected final class Node    {
        E    data;
        Node next;
        Node prev;

        Node(E data) {
            this.data = data;
            next = prev = null;
        }

        Node(E data, Node prev, Node next) {
            this(data);
            this.prev = prev;
            this.next = next;
        }
    }

    /* Start of the List */
    protected Node head;

    /* Size of the List */
    protected int  size;

    /* Constructor to make an empty List */
    public LinkedList() {
        head = null;
        size = 0;
    }


    . . . . . .
```

```
/*
 * Returns a list-iterator of the elements in this list (in proper sequence),
 * starting at the beginning of the list.
 */
public ListIterator<E> listIterator() {
        return new ListIteratorImpl(0);
}

/*
 * Returns a list-iterator of the elements in this list (in proper sequence),
 * starting at the specified position in the list. Obeys the general contract of
 * List.listIterator(int).
 * Throws IndexOutOfBoundsException
 */
public ListIterator<E> listIterator(int index) {
        return new ListIteratorImpl(index);
}

/*
 * Returns an iterator over the elements in this list.
 * The elements will be returned in order from first (head) to last (tail).
 */
public Iterator<E> iterator() {
        throw new UnsupportedOperationException();
}

/*
 * Returns an iterator over the elements in this list in reverse sequential order.
 * The elements will be returned in order from last (tail) to first (head).
 */
public Iterator<E> descendingIterator() {
        throw new UnsupportedOperationException();
}



. . . . . .



/* Class to implement interface ListIterator */

private class ListIteratorImpl implements ListIterator<E> {

        private Node lastReturned = null; // node containing element last returned
        private Node next; // node after cursor
        private int nextIndex; // index of node after cursor
```

```
/**
 * Constructs an list iterator starting at index.
 */
ListIteratorImpl(int index) {
        checkIndex(index);
        if (index < (size >> 1)) {
                next = head;
                for (nextIndex = 0; nextIndex < index; nextIndex++)
                        next = next.next;
        }
        else {
                next = head;
                for (nextIndex = size; nextIndex > index; nextIndex--)
                        next = next.prev;
        }
}

/**
 * Returns true if this list iterator has more elements when
 * traversing the list in the forward direction.
 */
public boolean hasNext() {
        return nextIndex != size;
}

/**
 * Returns the next element in the list and advances the cursor position.
 * This method may be called repeatedly to iterate through the list,
 * or intermixed with calls to previous to go back and forth.
 */
public E next() {
        if (nextIndex == size)
                throw new NoSuchElementException();
        lastReturned = next;
        next = next.next;
        nextIndex++;
        return lastReturned.data;
}

/**
 * Returns true if this list iterator has more elements when
 * traversing the list in the reverse direction.
 */
public boolean hasPrevious() {
        return nextIndex != 0;
}
```

```
/**
 * Returns the previous element in the list and moves the cursor
 * position backwards.  This method may be called repeatedly to
 * iterate through the list backwards, or intermixed with calls to
 * next to go back and forth.
 */
public E previous() {
        if (nextIndex == 0)
                throw new NoSuchElementException();
        lastReturned = next = next.prev;
        nextIndex--;
        return lastReturned.data;
}

/**
 * Returns the index of the element that would be returned by a
 * subsequent call to next. (Returns list size if the list
 * iterator is at the end of the list.)
 */
public int nextIndex() {
        return nextIndex;
}

/**
 * Returns the index of the element that would be returned by a
 * subsequent call to previous. (Returns -1 if the list
 * iterator is at the beginning of the list.)
 */
public int previousIndex() {
        return nextIndex - 1;
}

/**
 * Removes from the list the last element that was returned by
 * next or previous (optional operation).  This call can
 * only be made once per call to next or  previous.
 */
public void remove() {
        if(lastReturned == null)
                throw new IllegalStateException();
        Node lastNext = lastReturned.next;
        LinkedList.this.removeNode(lastReturned);
        if (next == lastReturned) // after previous
                next = lastNext;
        else  // after next
                nextIndex--;
        lastReturned = null;
}
```

```
        /**
         * Replaces the last element returned by next or
         * previous with the specified element (optional operation).
         * This call can be made only if neither remove nor
         * add have been called after the last call to  next or previous.
         */
        public void set(E e) {
                if (lastReturned == null)
                        throw new IllegalStateException();
                lastReturned.data = e;
        }


        /**
         * Inserts the specified element into the list (optional operation).
         * The element is inserted immediately before the next element that
         * would be returned by next, if any, and after the next
         * element that would be returned by previous, if any.  (If the
         * list contains no elements, the new element becomes the sole element
         * cursor: a subsequent call to next would be unaffected, and a
         * subsequent call to  previous would return the new element.
         */
        public void add(E e) {
                lastReturned = null;
                LinkedList.this.add(nextIndex, e);
                nextIndex++;
      }
    } // end of class ListIteratorImpl

}
```

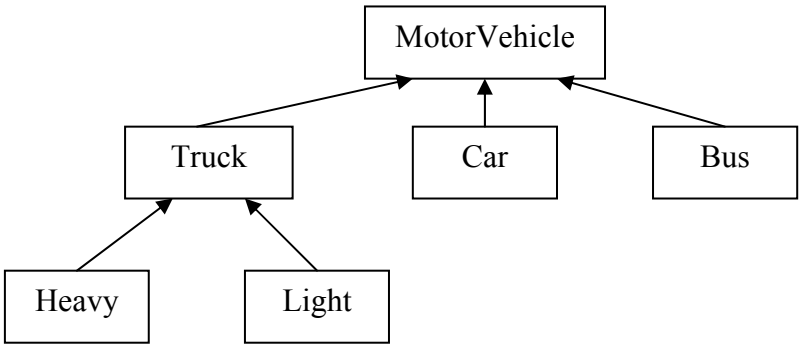
Задача 2: Да се създаде клас – наследник на класа ***LinkedList***, който позволява поддържане на сортиран списък.

```
public class SortedList<E extends Comparable<E>> extends LinkedList<E> {

    public void insert(E e) {
        Iterator<E> it = iterator();
        int index = 0;
        while(it.hasNext() && it.next().compareTo(e) < 0 )
            index++;
        add(index, e);
    }

    public boolean insertUniq(E e) {
        Iterator<E> it = iterator();
        E tmp = null;
        int index = 0;
        while(it.hasNext() && (tmp = it.next()).compareTo(e) < 0 )
            index++;
        if(tmp.compareTo(e) == 0)
            return false;
        else {
            add(index, e);
            return true;
        }
    }
}
```

Задача 3: Да се състави на езика Java клас **FrecDictionary** за представяне на честотен речник на текст. Речникът съхранява думите от текста заедно с броя на срещанията на всяка от тях. Класът да съдържа необходимите конструктори и следните методи:

- **void *add*(String *word*)** – добавя думата в речника ако я няма, иначе увеличава броя на срещанията й;
- **int *getCount*(String *word*)** - ако думата се среща в речника – връща броя на срещанията й, иначе връща 0;
- **String *toString*()** – извежда сортиран списък на думите от речника заедно с броя на срещания на всяка от тях.

Задача 4: Да се създаде следната йерархия от класове, представяща превозните средства, с които разполага транспортна фирма:



Класът ***MotorVehicle*** представя моторно превозно средство и съдържа описание на:

- регистрационен номер;
- месец на последен технически контрол;
- установено при контрола състояние (отлично, добро, незадоволително);
- признак за това дали е свободно в момента.

Класът ***Truck*** представя камион и има поле, задаващо неговата товароподемност.
Класът ***Heavy*** представя тежкотоварен камион и съдържа допълнително признак за това, дали има право да се движи в чужбина и дали е минал технически преглед след последното пътуване.
Класът ***Car*** описва лека кола и съдържа описание на нейната марка и мощност.
Класът ***Bus*** представя автобус и има данни за броя на местата му.
Техническият контрол на МПС се извършва по следните правила:

- за тежкотоворните камиони след всеки преход;
- за останалите – регулярно на определен период: 4 месеца за леките коли, 3 месеца за бусовете и един месец за лекотоварните камиони.

За изброените класове да се напишат конструктори и методи, необходими за реализиране на класа ***Transporter***, който представя транспортната фирма. Класовете, представящи МПС, които подлежат на регулярен преглед, да съдържат и метод *timeToControl(int now)*, който по текущия месец (параметъра now) определя дали превозното средство вече подлежи на преглед.
Класът ***Transporter*** съдържа информация за текущия месец и списък на всички МПС, притежание на фирмата. Класът да съдържа следните методи:

- метод *readyForUse(int now)*, който претърсва списъка на превозните средства и връща списък, съдържащ всички такива, които могат да бъдат използвани съгласно правилата за технически преглед към момента (параметъра now);
- метод *toBeControled(int now)*, който изработва и връща списък на всички превозни средства, подлежащи на регулярен контрол, които трябва да се явят на преглед.

Забележка: Навсякъде данната „месец” да се представя с тип int.