Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №8
22.04.2010г.

ТЕМА: Свързана реализация на линеен списък

Задача: Да се реализира клас за представяне на 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 {@code 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> {

    private static final long serialVersionUID = 1L;

    /* 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;

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

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

    /*
     * Checks validity of index
     */
    private void checkIndex(int index) {
```

```
        if(index <0 || index > size)
                throw new IndexOutOfBoundsException("Index " + index +
                                                    " out of [0," + size + "]");
}

/*
 * Returns a reference to the node with given index
 */
private Node getNode(int index) {
        if (index < 0 || index > size) throw new IndexOutOfBoundsException("Index:
                                                    "+index+ ",  Size: "+size);
        Node tmp;
        if(index < size/2) {
                tmp = head;
                while(index-- > 0)
                        tmp = tmp.next;
        }
        else {
                tmp = head.prev;
                while(++index < size)
                        tmp = tmp.prev;
        }
        return tmp;
}

/*
 * Removes specified node
 */
private void removeNode(Node node) {
        if(node == null)
                throw new NullPointerException();
        if(size == 1) {
                head = null;
                size = 0;
        }
        else if(size > 1) {
                node.prev.next = node.next;
                node.next.prev = node.prev;
                if(node == head)
                        head = head.next;
                node.next = node.prev = null;
                size--;
        }
}
```

```
/*
 * Appends the specified element to the end of this list.
 * This method is equivalent to addLast(e).
 */
public boolean add(E e) {
        addLast(e);
        return true;
}

/*
 * Inserts the specified element at the specified position in this list.
 * Shifts the element currently at that position (if any) and any subsequent elements
 * to the right (adds one to their indices).
 * Throws IndexOutOfBoundsException
 */
public void add(int index, E e) {
        checkIndex(index);
        if(index == 0)
                addFirst(e);
        else if(index == size)
                addLast(e);
        else {
                Node tmpNext = getNode(index);
                Node tmp = new Node(e, tmpNext.prev, tmpNext);
                tmpNext.prev = tmpNext.prev.next = tmp;
                size++;
        }
}

/*
 * Inserts the specified element at the beginning of this list.
 */
public void addFirst(E e) {
        if (size == 0){
                head = new Node(e);
                head.prev = head.next = head;
        }
        else {
                head.prev = head.prev.next = new Node(e);
                head = head.prev;
        }
        size++;
}

/*
 * Appends the specified element to the end of this list.
 * This method is equivalent to add(E).
 */
```

```
public void addLast(E e) {
	if (size == 0){
		head = new Node(e);
		head.prev = head.next = head;
	}
	else
		head.prev = head.prev.next = new Node(e);
	size++;
}

/*
 * Returns the element at the specified position in this list.
 * Throws IndexOutOfBoundsException
 */
public E get(int index) {
	throw new UnsupportedOperationException();
}

/*
 * Returns the first element in this list. Throws NoSuchElementException -
 * if this list is empty
 */
public E getFirst() {
	if (size == 0)
		throw new NoSuchElementException();
	return head.data;
}

/*
 * Returns the last element in this list.
 * Throws NoSuchElementException - if this list is empty
 */
public E getLast() {
	if (size == 0)
		throw new NoSuchElementException();
	return head.prev.data;
}

/*
 * Retrieves and removes the head (first element) of this list.
 * Returns the head of this list, or null if this list is empty
 */
public E poll() {
	if (size == 0)
		return null;
	return removeFirst();

}
```

```
/*
 * Retrieves and removes the first element of this list, or returns null if list empty.
 * Returns the first element of this list, or null if this list is empty
 */
public E pollFirst() {
	return poll();
}

/*
 * Retrieves and removes the last element of this list, or returns null if list is empty.
 * Returns the last element of this list, or null if this list is empty
 */
public E pollLast() {
	if (size == 0)
		return null;
	return removeLast();

}

/*
 * Retrieves and removes the head (first element) of this list.
 * Returns the head of this list
 * Throws NoSuchElementException - if this list is empty
 */
public E remove() {
	return removeFirst();
}

/*
 * Removes and returns the first element from this list.
 * Throws NoSuchElementException - if this list is empty
 */
public E removeFirst() {
	if (size == 0)
		throw new NoSuchElementException();
	E tmp = head.data;
	if(size == 1)
		head = null;
	else {
		head.next.prev = head.prev;
		head.prev.next = head.next;
		head = head.next;
	}
	size--;
	return tmp;
}
```

```
	/*
	 * Removes and returns the last element from this list.
	 * Returns the last element from this list.
	 * Throws NoSuchElementException - if this list is empty
	 */
	public E removeLast()  {
		if (size == 0)
			throw new NoSuchElementException();
		E tmp;
		if(size == 1){
			tmp = head.data; // head = tail
			head = null;
		}
		else {
			tmp = head.prev.data;
			head.prev.prev.next = head;
			head.prev = head.prev.prev;
		}
		size--;
		return tmp;
	}

	/*
	 * Removes the element at the specified position in this list.
	 * Shifts any subsequent elements to the left (subtracts one from their indices).
	 * Returns the element that was removed from the list.
	 * Returns the element previously at the specified position.
	 * Throws IndexOutOfBoundsException - if the index is out of range (index < 0 ||
index >= size())
	 */
	public E remove(int index) {
		throw new UnsupportedOperationException();
	}

	/*
	 * Removes the first occurrence of the specified element in this list (
	 * when traversing the list from head to tail). If the list does not contain the element,
	 * it is unchanged. Returns true if the list contained the specified element
	 */
	public boolean removeFirstOccurrence(Object o){
		throw new UnsupportedOperationException();
	}

	/*
	 * Removes the last occurrence of the specified element in this list
	 * (when traversing the list from head to tail). If the list does not contain the element,
	 * it is unchanged. Returns true if the list contained the specified element
	 */
```

```
public boolean removeLastOccurrence(Object o) {
        throw new UnsupportedOperationException();
}

/*
 * Returns the index of the first occurrence of the specified element in this list,
 * or -1 if this list does not contain the element.
 */
public int indexOf(Object o) {
        throw new UnsupportedOperationException();
}

/*
 * Returns the index of the last occurrence of the specified element in this list,
 * or -1 if this list does not contain the element.
 */
public int lastIndexOf(Object o) {
        throw new UnsupportedOperationException();
}

/*
 * Replaces the element at the specified position with the specified element.
 * Returns the element previously at the specified position.
 * Throws IndexOutOfBoundsException
 */
public E set(int index, E element) {
        throw new UnsupportedOperationException();
}

/*
 * Returns true if this list contains the specified element.
 * More formally, returns true if and only if this list contains at least one element e
 * such that (o==null ? e==null : o.equals(e)).
 */
public boolean contains(Object o) {
        throw new UnsupportedOperationException();
}

/*
 * Removes the first occurrence of the specified element from this list,
 * if it is present. If this list does not contain the element, it is unchanged.
 */
public boolean remove(Object o) {
        throw new UnsupportedOperationException();
}

/*
 *  Returns a view of the portion of this list between the specified fromIndex,
```

```
 *  inclusive, and toIndex, exclusive. (If fromIndex and toIndex are equal,
 *  the returned list is empty.) The returned list is backed by this list, so
 *  non-structural changes in the returned list are reflected in this list, and vice-versa.
 *  The returned list supports all of the optional list operations supported by this list.
 *  Throws IndexOutOfBoundsException - for an illegal endpoint index value
 *  (fromIndex < 0 || toIndex > size || fromIndex > toIndex)
 */
public List<E> subList(int fromIndex, int toIndex){
        throw new UnsupportedOperationException();
}

/*
 * Removes all items from this list.
 */
public void removeAll() {
        throw new UnsupportedOperationException();
}

/*
 * Returns true if this collection contains no elements.
 */
public boolean isEmpty() {
        return size == 0;
}

/*
 * Removes all of the elements from this list.
 */
public void clear() {
        throw new UnsupportedOperationException();
}

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

/*
 * 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();
}

/*
 * Returns a shallow copy of this LinkedList.
 */
public Object clone() {
        throw new UnsupportedOperationException();
}

/*
 * Indicates whether some other object is "equal to" this one.
 * Returns true if this object is the same as the obj argument; false otherwise.
 */
public boolean equals(Object obj) {
        throw new UnsupportedOperationException();
}

/*
 *  Returns a string representation of the object.
 */
public String toString() {
        throw new UnsupportedOperationException();
}
```

```
        /*
         * Returns an array containing all of the elements in this list in proper sequence
        */
        public Object[] toArray() {
                throw new UnsupportedOperationException();
        }

        . . . . . .

}
```