Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №13
27.05.2010г.

ТЕМА: Структура от данни ТАБЛИЦА.
Приоритетна опашка

Задача 1: Да се напише на езика Java клас **BookIndex** за представяне на абстрактна структура данни **Индекс на книга**.

Индекс на книга представлява множество от всички думи, които се срещат в книгата, лексикографски сортирани във възходящ ред. С всяка дума е асоцииран сортиран списък от номерата на всички страници в книгата, на които думата се среща.

- Класът да съдържа следните методи:
- метод ***add*** с параметри **дума** и **номер на страница**, който добавя думата в индекса, ако тя не е включена в него, в противен случай добавя номера на страницата в списъка от страници за думата;
- метод ***onPages*** с параметър **дума**, който връща списък от всички страници, на които се среща думата;
- метод ***wordsOnPage*** с параметър **номер на страница**, който връща сортиран списък на всички думи, които се срещат на страницата;
- метод ***printIndex,*** който извежда на екрана в подходящ формат индекс на книга.

Забележка: За реализиране на множеството от наредени двойки <дума, списък от страници> да се използва класът **TreeMap**, реализиращ интерфейса **SortedMap**.

```
package maps.bookindex;
import java.util.*;

public class BookIndex {
    private TreeMap<String, TreeMap<Integer, Integer>> map =
                                    new TreeMap<String, TreeMap<Integer, Integer>>();

    public void add(String word, int page) {
        TreeMap<Integer, Integer> tmp;
        if((tmp = map.get(word)) == null) {
            TreeMap<Integer, Integer> intMap = new TreeMap<Integer, Integer>();
            intMap.put(new Integer(page), new Integer(1));
            map.put(word, intMap);
        }
        else {
            Integer tmp1;
            if((tmp1 = tmp.get(new Integer(page))) == null)
                tmp.put(new Integer(page), new Integer(1));
```

```
                else {
                        int ii = tmp1.intValue()+1;
                        tmp.put(new Integer(page), new Integer(ii));
                }
        }
    }

    public LinkedList<Integer> onPages(String word) {
        TreeMap<Integer, Integer> tmp = map.get(word);
        if(tmp == null)
                return null;
        LinkedList<Integer> result = new LinkedList<Integer>();
        Set<Integer> set = tmp.keySet();
        Iterator<Integer> it = set.iterator();
        while(it.hasNext())
                result.add(it.next());
        return result;
    }

    public void printPagesOf(String word) {
        LinkedList<Integer> tmp = onPages(word);
        Iterator<Integer> it = tmp.iterator();
        //System.out.print(it.next());
        while(it.hasNext())
                System.out.print(", " + it.next());
        System.out.println();
    }

    public LinkedList<String> wordsOnPage(int page) {
        LinkedList<String> result = new LinkedList<String>();
        Set<Map.Entry<String,TreeMap<Integer, Integer>>> set = map.entrySet();
        Iterator<Map.Entry<String,TreeMap<Integer, Integer>>> it = set.iterator();
        Map.Entry<String, TreeMap<Integer, Integer>> tmp;
        TreeMap<Integer, Integer> tmp1;
        while(it.hasNext()) {
                tmp = it.next();
                tmp1 = tmp.getValue();
                if(tmp1.containsKey(new Integer(page)))
                        result.add(tmp.getKey());
        }
        return result;
    }

    public String toString() {
        return map.toString();
    }
}
```

Задача 2: Да се напише на езика Java клас **PriorityQueue** за представяне на приоритетна опашка на базата на двоична пирамида.

```
package queues.priorityqueues;

import java.util.Iterator;
import java.util.Arrays;
import java.util.NoSuchElementException;
import java.util.Comparator;

/**
 * A PriorityQueue holding elements on a priority heap
 * according to their natural order.
 */
public class PriorityQueue<E> {

    private E[] elements; // array holding  priority queue elements
    private int size; // number of active elments in the priority queue
    private int capacity; // number of active elments in the array holding priority queue
    private Comparator<? super E> comparator;

    /**
     * Creates an empty  priority queue that holds elements with the
     * specified generic type E.
     */
    public PriorityQueue() {
        this(10);
    }

    /**
     * Creates a  priority queue with given capacity
     * that holds elements with the specified generic type E.
     */
    public PriorityQueue(int capacity) {
        this(capacity, null);
    }

    /**
     * Constructs a priority queue with the specified capacity and comparator.
     */
    public PriorityQueue(int capacity, Comparator<? super E> comparator) {
        if (capacity < 1) {
            throw new IllegalArgumentException();
        }
        size = 0;
        this.capacity = capacity ;
```

```
        elements = (E[])new Object[capacity];
        this.comparator = comparator;
}

 /**
  * Returns the iterator of the priority queue, which will not return elements
  * in any specified ordering.
  */
public Iterator<E> iterator() {
        return new PriorityQueueIterator();
}

/**
 * Returns the size of the priority queue.
 */
public int size() {
        return size;
}

/**
 * Returns true if the priority queue is empty.
 */
public boolean isEmpty() {
        return size  ==  0;
}

/**
 * Removes all the elements of the priority queue.
 */
public void clear() {
        Arrays.fill(elements, null);
        size = 0;
}

/**
 * Inserts the element in the priority queue.
 * throws ClassCastException if the element
 * cannot be compared with the elements in the queue
 */
public boolean enqueue(E newElement) {
        if (newElement == null)
                throw new NullPointerException();
        if(capacity - size < 2)
                ensureCapacity();
        elements[size] = newElement;
        siftUp(size++);
        return true;
}
```

```
/**
 * Gets and removes the head of the queue.
 */
public E dequeue() {
        if (isEmpty())
                return null;
        E result = elements[0];
        removeAt(0);
        return result;
}

/**
 * Gets but does not remove the head of the queue.
 */
public E peek() {
        if (isEmpty())
                return null;
        return elements[0];
}

 /**
  * Removes the specified object from the priority queue.
  */
public boolean remove(Object o) {
        if (o == null)
                return false;
        for (int i = 0; i < size; i++) {
                if (elements[i].equals(o)) {
                        removeAt(i);
                        return true;
                }
        }
        return false;
}

/**
 * Cheks if there is an element in this queue equals to the object.
 */
public boolean contains(Object o) {
        for (int i = 0; i < size; i++)
                if(elements[i].equals(o))
                        return true;
        return false;
}

public E[] toArray() {
        E[] array = (E[]) new Object[size];
```

```
        for(int i = 0; i < size; i++)
                array[i] = elements[i];
        return array;
}

private void ensureCapacity() {
        capacity = 2 * capacity;
        E[] oldArray = elements;
        elements = (E[])new Object[capacity];
        for (int i = 0; i < size; i++)
                elements[i] = oldArray[i];
        oldArray = null;
}

private void removeAt(int index) {
        size--;
        elements[index] = elements[size];
        siftDown(index);
        elements[size] = null;
}

private int compare(E e1, E e2) {
        if (comparator != null) {
                return comparator.compare(e1,e2);
        }
        return ((Comparable<? super E>) e1).compareTo(e2);
}

private void siftUp(int child) {
        E newValue = elements[child];
        int parent;
        while (child > 0) {
                parent = (child - 1) / 2;
                E parentValue = elements[parent];
                if (compare(parentValue, newValue) > 0)
                        break;
                elements[child] = parentValue;
                child = parent;
        }
        elements[child] = newValue;
}

private void siftDown(int root) {
        E lastValue = elements[root];
        int child;
        while ((child = root * 2 + 1) < size) {
                if (child + 1 < size && // two children
                                compare(elements[child + 1], elements[child]) > 0)
```

```
                child++;  // second child greater
            if (compare(lastValue, elements[child]) > 0)
                break;
            elements[root] = elements[child];
            root = child;
        }
        elements[root] = lastValue;
    }

    private class PriorityQueueIterator implements Iterator<E> {

        private int current = -1;
        private boolean removePossible = false;

        public boolean hasNext() {
            return current < size - 1;
        }

        public E next() {
            if (!hasNext())
                throw new NoSuchElementException();
            removePossible = true;
            return elements[++current];
        }

        public void remove() {
            if (!removePossible)
                throw new IllegalStateException();
            removePossible = false;
            removeAt(current--);
        }
    }

}
```