Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №12
20.05.2010г.

ТЕМА: Двоични дървета за търсене

Клас за представяне на двоично дърво за търсене.

```
package trees.binarysearchtrees;
import java.util.NoSuchElementException;
import java.util.Iterator;

public class BinarySearchTree<T extends Comparable<? super T>> {

    private static class Node<T> {
        T data;        // The data in the node
        Node<T> left;  // Left child
        Node<T> right; // Right child
        Node<T> parent; // Parent

        Node(T data) {
            this(data, null, null, null);
        }

        Node(T data, Node<T> left, Node<T> right, Node<T> parent) {
            this.data = data;
            this.left = left;
            this.right = right;
            this.parent = parent;
        }

        public String toString() {
            return data.toString();
        }
    }

    private class BSTreeIterator implements Iterator<T> {
        private Node<T> nextNode;
        private Node<T> lastReturned;

        BSTreeIterator() {
            nextNode = root;
            lastReturned = null;
            if(nextNode != null)
                while(nextNode.left != null)
                    nextNode = nextNode.left;
        }
```

```
        public boolean hasNext() {
                return nextNode != null;
        }

        public T next() {
                if (nextNode == null)
                    throw new NoSuchElementException();
                lastReturned = nextNode;  // save current value of next
                Node<T> tmp;  // set nextNode to the next node in order
                if (nextNode.right != null){ // successor is the leftmost node
                                             // of the right subtree
                    nextNode = nextNode.right;
                    while (nextNode.left != null)
                            nextNode = nextNode.left;
                }
                else {
                    // move up the tree, looking for a parent for
                    // which nextNode is a left child
                    tmp = nextNode.parent;
                    while (tmp != null && nextNode == tmp.right) {
                       nextNode = tmp;
                       tmp = tmp.parent;
                    }
                    nextNode = tmp;
                }
                return lastReturned.data;
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }
}

/** The tree root */
private Node<T> root;

 /** The size of the tree */
private int size;

/**
* Construct the tree.
*/
public BinarySearchTree( ) {
        root = null;
        size = 0;
}
```

```
/**
* Make the tree empty
*/
public void clear() {
        root = null;
        size = 0;
}

/**
 * @return the size of the tree.
*/
public int size() {
        return size;
}

/**
* Test if the tree is empty.
* @return true if empty, false otherwise.
*/
public boolean isEmpty() {
        return root == null;
}


/**
 * Find an item in the tree.
 * @param data the item to search for.
 * @return the matching data or null if not found.
 */
public T find(T data) {
        Node<T> tmp = findNode(data);
        if(tmp == null)
                return null;
        else
                return tmp.data;
}

/**
 * Internel method to find an item in the tree.
 * @param data the item to search for.
 * @return node containing data or null if not found.
*/
private Node<T> findNode (T data) {
        Node<T> tmp = root;
        int compared;
        while(tmp != null) {
                compared = tmp.data.compareTo(data);
                if(compared == 0)
```

```
                    break;
                if(compared < 0)
                    tmp = tmp.right;
                else
                    tmp = tmp.left;
        }
        return tmp;
}

/**
 * Insert into the tree.
 * @param data the item to insert.
 * @return false if data already present.
*/
public boolean insert(T data) {
        Node<T> tmp = root;
        Node<T> parent = null;
        int compared = 0;
        while(tmp != null) {
                parent = tmp;
                compared = tmp.data.compareTo(data);
                if(compared == 0)
                    return false;
                if(compared < 0)
                    tmp = tmp.right;
                else
                    tmp = tmp.left;
        }
        tmp = new Node<T>(data, null, null, parent);
        if(parent == null) // insert in an empty tree
                root = tmp;
        else if(compared < 0)
                parent.right = tmp;
        else
                parent.left = tmp;
        size++;
        return true;
}

/**
 * Remove from the tree.
 * @param data the item to remove.
 * @throws NoSuchElementException if data not found.
*/
public void remove(T data) {
        Node<T> toDelete = findNode(data);
        if(toDelete == null)
                throw new NoSuchElementException();
```

```
        removeNode(toDelete);
        size--;
}

private void removeNode(Node<T> toDelete) {
        Node<T> replaceWith = null;
        if(toDelete != null) {
                if(toDelete.left == null || toDelete.right == null) { // node has
                                                                      // at most one child
                        if(toDelete.left == null) // replace with right subtree
                                replaceWith = toDelete.right;
                        else
                                replaceWith = toDelete.left;  // replace with left subtree
                        if(root == toDelete)  // root deleted
                                root = replaceWith;
                        else {
                                Node<T> parent = toDelete.parent;
                                if(parent.right == toDelete) // to delete right child of parent
                                        parent.right = replaceWith;
                                else  // to delete left child of parent
                                        parent.left = replaceWith;
                                if(replaceWith != null)
                                        replaceWith.parent = parent;
                        }
                }
                else {  // node has two children
                        Node<T> tmp = toDelete.right;
                        Node<T> tmpParent = toDelete;
                        while(tmp != null){ // find smallest data of the right
                                            // subtree of toDelete
                                tmpParent = tmp;
                                tmp = tmp.left;
                        }
                        tmp = tmpParent;
                        toDelete.data = tmp.data;
                        removeNode(tmp);
                }
        }
}

 /**
 * Find the smallest element in the tree.
* @return smallest element or null if tree is empty.
*/
public T findMinimum() {
        if(root == null)
                return null;
        Node<T> tmp = root;
```

```
            while(tmp.left != null)
                tmp = tmp.left;
            return tmp.data;
        }

        /**
         * Find the largest element in the tree.
         * @return the largest element or null if tree is empty
        */
        public T findMaximum( ) {
            if(root == null)
                return null;
            Node<T> tmp = root;
            while(tmp.right != null)
                tmp = tmp.right;
            return tmp.data;
        }

        public Iterator<T> iterator() {
            return new  BSTreeIterator();
        }

        public String toString() {
            StringBuffer stb = new StringBuffer();
            if(root == null)
                stb.append("Empty Tree");
            else
                stb.append(preOrder(root, 0));
            return stb.toString();
        }

        private String preOrder(Node<T> node, int level) {
            StringBuffer stb = new StringBuffer();
            for(int i=0; i<level; i++)
                stb.append(" ");
            if(node == null)
                stb.append("---\n");
            else {
                stb.append(node.toString() + "\n");
                stb.append(preOrder(node.left, level+2));
                stb.append(preOrder(node.right, level+2));
            }
            return stb.toString();
        }
}
```

Задача 1: Компютърен магазин съхранява каталог с информация за продаваните видове преносима памет. За всеки артикул се пазят следните данни: име на фирма – производител, обем в MB и цена. Каталогът е организиран като двоично дърво за търсене, като артикулите са подредени по нарастване на обема на съответната памет, а при равен обем – по намаляване на цената.

Да се състави на езика Java клас ***RemovableStorage*** за представяне на описания каталог. Класът да съдържа необходими конструктори и следните методи:

- **void *insertDevice*(String *producer*, int *volume*, float *price*)** – въвежда сведение за нов артикул;
- **void *print*()** – извежда данните от каталога подредени по намаляване на обема на устройството, а при равен обем – по нарастване на цената;
- **void *changePrice*(String *producer*, int *volume*, float *price*, float *newPrice*)** – изменя цената на даден артикул;
- **void *deleteDevice*(String *producer*, int *volume*, float *price*)** – изтрива от каталога сведение за указания артикул;
- ***allDevices*(int *volume*)** – връща списък на всички устройства с обем, указан с параметъра на метода;
- ***allDevices*(float *price*)** - връща списък на всички устройства с цена, указана с параметъра на метода;

Задача 2: Да се напишат на езика Java следните класове:

1. Клас **Student** за представяне на студент, който има факултетен номер и списък от наредени двойки (дисциплина, оценка);

2. Клас **StudentSearchTree** – двоично дърво за търсене, което служи за представяне на множество от студенти, наредено съгласно средния успех на студентите (ако двама студенти са с равен успех, те да се считат наредени съгласно факултетните си номера). Класът да съдържа:
   - необходимите полета и конструктори
   - и следните методи:

   void *insertStudent*(String *name*, int *fakNumb*, String *disc*, float *mark*) – въвежда сведение за студент с дадено име и факултетен номер и оценка по една дисциплина. Ако студентът не е включен в дървото – данната за него се добавя. Ако вече е включен, то се добавя само данната да съответния изпит ;

   void *printSorted*() рекурсивен метод който извежда списък на всички студенти, наредени по намаляваща стойност на средния им успех.