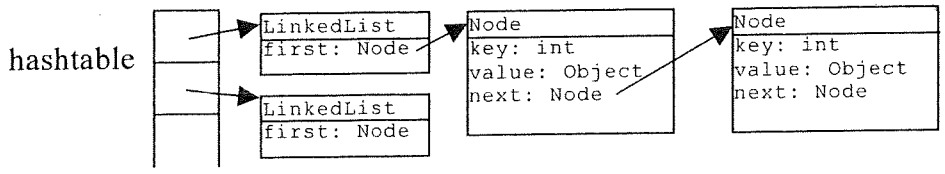
T-106.213 Basic Course in Programming L1: exam 14.01.2005 / Kary Främling

The class `IntHashMap` from Appendix 1 is a simple implementation of a *map* (or *dictionary*), in which *key-value mappings* can be stored. *Keys* are used to access the *values* of the map. The given class is different from the Java standard class `java.util.HashMap` (whose details are unimportant here) mostly in that in class `IntHashMap` only values of type `int` are used as keys.

Class `IntHashMap` has been implemented using a so-called *hashtable,* which works as follows. A hashtable has a constant size `n`, and its contents are stored in an array of that size. When one wants to add a certain value to the hashtable (under a certain key) a so-called *hash function* is used to calculate the array index (between 0 and (n-1)) at which the value will be placed. The hash function determines the index based on the key. In our case, all keys are integers and the hash function used in class `IntHashMap` is simply a modulo (remainder) of the key: `key % n`.

A hash function can assign the same index to various keys. For instance in an `IntHashMap` of size 11, the keys 5 and 27 are assigned the same index. The class `IntHashMap` solves this problem by using a *linked list* for each array index so that each list stores all those key-value mappings with that index (as determined by the hash function). The given classes `LinkedList` and `Node` serve as an implementation for linked lists. The following picture describes how a hashtable consists of an arrayful of linked lists, each of which contains key-value mappings in its nodes.



1. Fill in the methods `clear`, `get` ja `put` in the class `LinkedList` (which you'll find in Appendix 1) in such a way that the methods of class `IntHashMap` start to work as described by the documentation in Appendix 3. 20/100

2. Write a recursive implementation for the method `get`. Hint: you may have to write a private helper method in order to accomplish this. 10/100

3. There is an error in `IntHashMap` which manifests itself when the code in Appendix 2 is executed (no matter how you implemented class `LinkedList`). What is the error and how can it be fixed? 15/100

4. What happens if one tries to use negative values for keys? How can this behavior be corrected? 10/100

5. What is printed out when the main method in Appendix 2 is executed, assuming that the rest of the classes work as described by the documentation in Appendix 3 15/100

6. It would be possible to implement class `IntHashMap` using the Java standard classes `ArrayList`, `HashSet` (or some others) instead of the given classes `LinkedList` and `Node`. What changes would be needed in class `IntHashMap` in order to accomplish this? What good and bad sides might such a solution have? You don't need to provide any program code - just explain in general. 15/100

7. What kinds of changes would be needed in class `IntHashMap`, if one wanted it to work with any kind of keys (i.e. keys of type Object) in stead of working with just keys of type int, as it does now? You don't need to provide any program code - just explain in general. 15/100

Appendix 1

```
public class IntHashMap {
    public static final int SIZE = 11;

    private LinkedList[] heads;

    public IntHashMap() {
        this.heads = new LinkedList[IntHashMap.SIZE];
    }

    public void clear() {
        for ( int i = 0 ; i < heads.length ; i++ )
            this.heads[i].clear();
    }

    public boolean containsKey(int key) {
        if ( this.get(key) != null )
            return true;
        return false;
    }

    public Object get(int key) {
        return this.heads[this.getHash(key)].get(key);
    }

    public Object put(int key, Object value) {
        return this.heads[this.getHash(key)].put(key, value);
    }

    public String toString() {
        String s = "";
        for ( int i = 0 ; i < this.heads.length ; i++ ) {
            if ( this.heads[i] != null )
                s += this.heads[i].toString();
            else
                s += "\n";
        }
        return s;
    }

    private int getHash(int key) {
        return key % IntHashMap.SIZE;
    }
}

public class LinkedList {
    private Node first = null;

    public void clear() {
        // missing...
    }

    public Object get(int key) {
        // missing...
    }

    public Object put(int key, Object value) {
        // missing...
    }



    // continues on next page...
```

```
    public String toString() {
        String s = "";
        for ( Node n = this.first ; n != null ; n = n.getNext() )
            s += "(" + n.getKey() + "," + n.getValue() + ") ";
        return s + "\n";
    }
}

public class Node {
    private int key;
    private Object value;
    private Node next = null;

    public Node(int key, Object value, Node next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }

    public int getKey() {
      return this.key;
    }
    public void setKey(int key) {
      this.key = key;
    }
    public Object getValue() {
      return this.value;
    }
    public void setValue(Object value) {
      this.value = value;
    }
    public Node getNext() {
      return this.next;
    }
    public void setNext(Node next) {
      this.next = next;
    }
}
```

Appendix 2

```
public class IntHashMapTest {
    public static void main(String[] args) {
      IntHashMap ihm = new IntHashMap();
      ihm.put(77,"Olle");
      ihm.put(2,"Nina");
      ihm.put(23,"Kalle");
      ihm.put(56,"Vivan");
      ihm.put(1,"Pelle");
      ihm.put(10,"Katarina");
      ihm.put(56,"Rasmus");
      ihm.put(98,"Maria");
      ihm.put(56,"Ivan");
      ihm.put(74,"Olivia");
      System.out.println(ihm);
    }
}
```

`public boolean containsKey(int key)`

Returns `true` if this map contains a mapping for the specified key.

`public Object get(int key)`

Returns the value to which this map maps the specified key. Returns `null` if the map contains no mapping for this key.

`public int put(int key,`
`Object value)`

Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for this key, the old value is replaced by the specified value.

Returns:

previous value associated with specified key, or `null` if there was no mapping for key.

`public void clear()`

Removes all mappings from this map (optional operation).