JOHANNES KEPLER
UNIVERSITÄT LINZ

Algorithms and Data Structures 2
Winter term 2019
Stefan Grünberger, Dari Trendafilov

Assignment 3

Deadline: **Mon. 18.11.2019, 23:59**
Submission via: **www.pervasive.jku.at/Teaching/**

(Digital) Sorting

1. HeapSort 4+2+4 points

Create a class **MinHeap** which implements the *interface MyPriorityQueue* (don't change the interface!) from assignment 2 and implement the following methods:

```
// create a MinHeap using bottom-up construction in-place
public MinHeap(Long list[]) throws IllegalArgumentException { ... }

// search of elements
public boolean contains(Long val) throws IllegalArgumentException { ... }

// sorting with HeapSort
public static void sort(Long list[]) throws IllegalArgumentException { ... }
```

Use **exactly** the specified interface for implementation.

1. When passing an array to the constructor **MinHeap,** this array should be used directly to create a valid heap using the *bottom-up construction* **in-place** (i.e. without creating an additional array). If *null* is passed, throw an **IllegalArgumentException** because a heap cannot be created from a *null* array.

2. The method **contains** should return *true*, if the element passed is stored in the heap (can also be contained multiple times). Ensure an efficient implementation (i.e. do not search the entire array sequentially, but use the properties of a heap)!

3. The two phases of the HeapSort algorithm (heap creation and repeated removal of the smallest element) should be executed by the **sort** method, to sort any passed array in *descending* order. For the heap construction in phase 1 the new constructor can be used. Phase 2 should also be executed in-place on the passed array.

If required use private help methods for implementing the specified methods.

2. RadixSort 14 points

Implement the **direct RadixSort** (base 10) for *ascending* sorting of Integer lists. Integer values can be considered as unsigned. Describe the runtime complexity of your algorithm in the function comment of the „sort function".

Use **exactly** the specified interface for implementation:

```
public class RadixSort {
 // O(...)
 public static MyLinkedList sort(Integer list[]) throws IllegalArgumentException { ... }
}
```

You can use private help methods to implement the specified functionality:

```
private static int getDigit(int val, int pos)                        // returns the digit (base10) of val at position pos
private static void bucketSort(int pos, MyLinkedList[] buckets)      // calculates Buckets for position pos
private static void merge(MyLinkedList[] buckets)                    // Merges bucket lists into one list
```

Use the following implementation for bucket lists. Make sure that you use the overloaded method *add* (existing nodes shall not be created again) correctly. Implement and use the function *public void link(MyLinkedList list)* which appends the list in the argument to the end of the calling list.

```
public class MyLinkedList {
 public MyNode head;
 public MyNode tail;

 public MyLinkedList() {
  head = null;
  tail = null;
 }
 public void clear() {head = null; tail = null;}

 public void add(Integer val) {
  if(head == null) {
   head = new MyNode(val);
   tail = head;
  } else {
   MyNode tmp = new MyNode(val);
   tail.next = tmp;
   tail = tail.next;
   tail.next = null;
  }
 }

public void add(MyNode node) {
 if(head == null) {
  head = node;
  tail = head;
  node.next = null;
 } else {
  tail.next = node;
  tail = tail.next;
  tail.next = null;
 }
}

 public void link(MyLinkedList list) {
  // To Do
 }
}
```

```
public class MyNode {
 Integer value;
 public MyNode next;

 public MyNode(Integer value) {
  this.value = value;
  next = null;
 }
}
```

Note for the entire assignment: Pay special attention to reusability and efficiency regarding runtime and memory requirements!