5.1 Hashtables

In an exercise part of last week, we looked at the problem of storing a list of people that you want to invite to your birthday party together with a method to find out, who you already have invited. The method employed was a (growable) array containing a sorted list of entries. In a way, we did the first step of what you would also do in "normal life" or if you used an excel spreadsheet for this.

This week we extend and modify the task: We want to provide a method to store a data-set associated to each name (key), in order to be able to not only find out who you already have invited but also to view important data about your friends in the list.

Download the skeleton code for this exercise from http://lec.inf.ethz.ch/baug/informatik2/2016/ex/ex05/HashTable/Main.java

Class Friend

In the lecture the implementation of the class `Friend` was already shown. In this exercise you should implement this class yourself, such that it can be used later as an entry in the hash table. Your class `Friend` should implement the following interface:

```
class Friend {
  // constructor
  public Friend(String Name, String Email, String Favourite)

  // post: returns a string representation of this object
  public String toString()
}
```

Please implement the constructor and the function `toString()` such that when `f.toString()` on an object `f` of type `Friend` is called, it returns a string containing the friend's name, email address and favourite food seperated by comma and space ', '.

Here is an example:
`Kruemelmonster, monster@sesamstrasse.xyz, Kekse`

We have provided a skeleton here: Validate your solution here: https://challenge.inf.ethz.ch/team/websubmit.php?problem=IB16051

Hash Table

Your task is to implement the following HashTable

```
class HashTable {

    // constructor
    public HashTable(int size)

    // pre: string of length > 0
    // post: return hash value for string name
    int Hash(String name)
```

```
    // pre: key into the hash table
    // post: index of the key in the hash table,
    //   or the next free index if not found
    int Probe(String key)

    // pre: key and value to insert into hash table
    public void Set(String key, Friend value)

    // pre: key
    // post: return Friend corresponding to key or null
    public Friend Get(String key)
}
```

Functions `Grow` and `Hash` are already given. Implement functions `Probe`, `Set` and `Get`.

Probing should implement linear probing (Lineares Sondieren) as explained in the lecture. Functions `Set` and `Get` should make use of the `Probe` function to insert or retrieve an element.

Remark: from a correctness viewpoint, in this exercise it is sufficient to grow the hash table as soon as the number of used elements equals the number of available elements in the arrays. However, this is far from efficient: good hash tables grow as soon as a certain usage ratio (say, 25%) is exceeded. If you want your hash table implementation to be efficient, implement this.

Validate your solution here: https://challenge.inf.ethz.ch/team/websubmit.php?problem=IB16052

5.2 Sliding Window

Download the skeleton code for this exercise from http://lec.inf.ethz.ch/baug/informatik2/2016/ex/ex05/SlidingWindow/Main.java

Goal of this part of the exercise is to write a sliding window object that can return the maximum and minimum of $n > 0$ provided integer values. We do not provide the interface of the object. Rather we describe its behavior phenomenologically and with example code in the following. Your task is to implement the corresponding class such that the object's behavior is as expected. Tip: use a circular buffer discussed in the lecture.

Provide a class `SlidingWindow` with the following properties:

- A **sliding window** `w` can be instantiated with a **window size** n in the following way:

  ```
  // assume n of type int, n > 0
  SlidingWindow w = new SlidingWindow(n);
  ```

- A sliding window provides an **input method** `Put` that accepts integer numbers and has no return value

  ```
  int value;
  w.Put(value);
  ```

- Assume that a number of $m > 0$ input values have already been provided via `w.Put`. A sliding window provides a method `Max` that returns the maximum of the most recent $\min(m, n)$ inputs. That means that given 5 inputs and a sliding window of size 4 the oldest element will be discarded, and with a sliding window of size 8 all 5 elements will be taken into account. Example:

```
SlidingWindow w = new SlidingWindow(4); // windows size 4
w.Put(1); w.Put(5); w.Put(2):
System.out.println(w.Max()); // values: 1 5 2 => output 5
w.Put(3); w.Put(4); w.Put(1);
System.out.println(w.Max()); // values: 2 3 4 1 => output 4
```

- Assume that a number of $m > 0$ input values have already been provided via `w.Put`. A sliding window provides a method `Min` that returns the minimum of the most recent $\min(m, n)$ inputs.

To test your implementation, enter the window size followed by a list of numbers followed by `end`. For example, input

```
3 1 2 3 4 end
```

should lead for each new element to an output of the form [min max]:

```
[1 1]
[1 2]
[1 3]
[2 4]
```

Validate your solution here: https://challenge.inf.ethz.ch/team/websubmit.php?problem=IB16053