Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Übungen zur Vorlesung Informatik II (D-BAUG) FS 2017

D. Sidler, F. Friedrich

http://lec.inf.ethz.ch/baug/informatik2/2017

Solution to exercise sheet # 8 **10.4.2017 – 25.4.2017**

Problem 8.1. Sliding Window

Open the task description at: https://codeboard.ethz.ch/inf2baugex08t01.

Goal of this 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 as discussed in the exercise session.

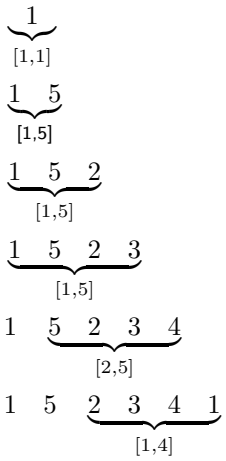

Figure 1: The sliding window w considers maximally the most recent 4 data points during insertion of $1, 5, 2, 3, 4, 1$. Insertion progress is shown top to bottom. Shown in brackets $[a, b]$ are then minimal and maximal values a and b within the window, respectively.

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 would 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
```

  Figure 1 shows the progress of the sliding window during insertion of the data points.

- 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 first. Afterwards you can either enter the `put`, `min`, or `max` command. The `put` command can take multiple integers as parameters. For example, input

```
3
put 1 2 3 4
min
max
put 8
min
max
```

should lead to the following output:

```
Min: 2
Max: 4
Min: 3
Max: 8
```

After implementing the `SlidingWindow` class, you can test your program by un-commenting the annotation `@RunTests` at the beginning of the class `Main`. Once you pass the test you can submit your program.

Solution of Problem 8.1.

```
class SlidingWindow {
    int buf[]; // data buffer
    int size; // how many elements of the buffer are used
    int position; // current insertion position of the buffer

    public SlidingWindow(int size) {
        buf = new int[size];
        position = 0;
        size = 0;
    }

    // post: value added to internal buffer
    public void Put(int value) {
        buf[position] = value;
        position = (position + 1) % buf.length;
        if (size < buf.length) {
            size++;
        }
    }

    // pre: some data have been provided via put
    // post: returns minimum of the most recent data in the buffer
    public int Min() {
        int min = buf[0];
        for (int i = 1; i < size; ++i) {
            if (buf[i] < min) {
                min = buf[i];
            }
```

```
        }
        return min;
    }

    // pre: some data have been provided via put
    // post: returns minimum of the most recent data in the buffer
    public int Max() {
        int max = buf[0];
        for (int i = 1; i < size; ++i) {
            if (buf[i] > max) {
                max = buf[i];
            }
        }
        return max;
    }

}
```

Problem 8.2. Open Hashing

We consider hash functions $h(k)$ for keys k for a hash table of size p, where p is a prime number.

a) Which of the following functions are useful hash functions? (Explain.)

- $h(k) =$ digit sum of k
- $h(k) = k(1 + p^3) \bmod p$
- $h(k) = \lfloor p(rk - \lfloor rk \rfloor) \rfloor$, $r \in \mathbb{R}^+ \setminus \mathbb{Q}$

b) Insert the keys 17, 6, 5, 8, 11, 28, 14, 15 in this order into an initially empty hash table of size 11. Use open addressing with the hash function $h(k) = k \bmod 11$ and resolve the conflicts using

(i) linear probing
(ii) quadratic probing, and
(iii) double hashing with $h'(k) = 1 + (k \bmod 9)$

in the following form.

```
17: h(17) = 6
__, __, __, __, __, __, 17, __, __, __, __

6: h(6) = 6
__, ...
```

c) Which problem occurs if the key 17 is removed from the hash tables in b), and how can you resolve it? Which problems occur if many keys are removed from a hash table?

Submission link: https://codeboard.ethz.ch/inf2baugex08t02

Solution of Problem 8.2.

a)
- $h(k) =$ digit sum of k: This function is not suitable for hashing. The value of the hash function must lie between 0 and $p-1$, but the digit sum can be arbitrarily large. Even if that sum was always below p it would not be a good choice because it does not lead to a uniform distribution of the keys.

- $h(k) = k(1 + p^3) \bmod p$: Since $p^3 \bmod p = 0$ we have $h(k) = k \bmod p$, which is a suitable hashing function as explained in the lecture. The only drawback of the definition as stated in this exercise are the unnecessary arithmetic operations.
- $h(k) = \lfloor p(rk - \lfloor rk \rfloor) \rfloor$, $r \in \mathbb{R}^+ \setminus \mathbb{Q}$: This is the so-called *multiplicative method* presented in the lecture, and it works well when r is chosen well.

b) All indices are considered modulo 11.

- Linear probing

```
17: h(17) = 6
__, __, __, __, __, __, 17, __, __, __, __

6: h(6) = 6 -> h(6) - 1 = 5
s

5: h(5) = 5 -> h(5) - 1 = 4
__, __, __, __,  5,  6, 17, __, __, __, __

8: h(8) = 8
__, __, __, __,  5,  6, 17, __,  8, __, __

11: h(11) = 0
11, __, __, __,  5,  6, 17, __,  8, __, __

28: h(28) = 6 -> h(28) - 1 = 5 -> h(28) - 2 = 4 -> h(28) - 3 = 3
11, __, __, 28,  5,  6, 17, __,  8, __, __

14: h(14) = 3 -> h(14) - 1 = 2
11, __, 14, 28,  5,  6, 17, __,  8, __, __

15: h(15) = 4 -> h(15) - 1 = 3 -> h(15) - 2 = 2 -> h(15) - 3 = 1
11, 15, 14, 28,  5,  6, 17, __,  8, __, __
```

- quadratic probing, if this sequence was considered: $h(k) + \{0, -1, +1, -4, +4, ...\} \bmod m$

```
17: h(17) = 6
__, __, __, __, __, __, 17, __, __, __, __

6: h(6) = 6 -> h(6) - 1 = 5
__, __, __, __, __,  6, 17, __, __, __, __

5: h(5) = 5 -> h(5) - 1 = 4
__, __, __, __,  5,  6, 17, __, __, __, __

8: h(8) = 8
__, __, __, __,  5,  6, 17, __,  8, __, __

11: h(11) = 0
11, __, __, __,  5,  6, 17, __,  8, __, __

28: h(28) = 6 -> h(28) - 1 = 5 -> h(28) + 1 = 7
11, __, __, __,  5,  6, 17, 28,  8, __, __

14: h(14) = 3
11, __, __, 14,  5,  6, 17, 28,  8, __, __

15: h(15) = 4 -> h(15) - 1 = 3 -> h(15) + 1 = 5 -> h(15) - 4 = 0
h(15) + 4 = 8 -> h(15) - 9 = 6 -> h(15) + 9 = 2
11, __, 15, 14,  5,  6, 17, 28,  8, __, __
```

- quadratic probing, if this sequence was considered: $h(k) + \{0, +1, -1, +4, -4, ...\} \bmod m$

```
17: h(17) = 6
__, __, __, __, __, __, 17, __, __, __, __

6: h(6) = 6 -> h(6) + 1 = 7
__, __, __, __, __, __, 17,  6, __, __, __

5: h(5) = 5
__, __, __, __, __,  5, 17,  6, __, __, __

8: h(8) = 8
__, __, __, __, __,  5, 17,  6,  8, __, __

11: h(11) = 0
11 , __, __, __, __,  5, 17,  6,  8, __, __

28: h(28) = 6 -> h(28) + 1 = 7 -> h(28) - 1 = 5 -> h(28) + 4 = 10
11 , __, __, __, __,  5, 17,  6,  8, __, 28

14: h(14) = 3
11 , __, __, 14, __,  5, 17,  6,  8, __, 28

15: h(15) = 4
11 , __, __, 14, 15,  5, 17,  6,  8, __, 28
```

- double hashing

```
17: h(17) = 6
__, __, __, __, __, __, 17, __, __, __, __
```

```
6: h(6) = 6 -> h(6) - h'(6) = 10
__, __, __, __, __, __, 17, __, __, __,  6

5: h(5) = 5
__, __, __, __, __,  5, 17, __, __, __,  6

8: h(8) = 8
__, __, __, __, __,  5, 17, __,  8, __,  6

11: h(11) = 0
11, __, __, __, __,  5, 17, __,  8, __,  6

28: h(28) = 6 -> h(28) - h'(28) = 4
11, __, __, __, 28,  5, 17, __,  8, __,  6

14: h(14) = 3
11, __, __, 14, 28,  5, 17, __,  8, __,  6

15: h(15) = 4 -> h(15) - h'(15) = 8 -> h(15) - 2*h'(15) = 1
11, 15, __, 14, 28,  5, 17, __,  8, __,  6
```

c) The deletion of a key k is problematic if another key k' with $h(k) = h(k')$ is inserted after k. If k would simply be removed (e.g., by marking the position as free), the key k could no longer be found, because the probing stops once an empty position is found. In the example above the keys 6 and 28 could no longer be found. Therefore we must mark the position explicitly as deleted, and when searching for a key the probing must continue when such a position is found. Of course, when inserting a new key such a marking may be overwritten if necessary.

   If many keys are deleted, then it may happen that the search for a key gets very inefficient (because the probing may visit many positions that are marked as deleted). Therefore hashing is suitable especially if keys are mostly inserted and searched and only rarely deleted.

Problem 8.3. Hash Table

In the lecture you have seen the implementation of a hash table and different approaches to resolve collisions. Your task is to implement a hash table with chaining (mit Verkettung). In this hash table the key is of type string and the value of type integer.

Open the code template at https://codeboard.ethz.ch/inf2baugex08t03. In this project we have 3 files containing the classes `Main`, `HashTable`, and `ListNode`. First we have a look at the class `ListNode` which is similar to what you have seen in last week's exercise.

```
public class ListNode {
    String key;
    int value;
    ListNode next;

    ListNode (String k, int val, ListNode nxt) {
        key = k;
        value = val;
        next = nxt;
    }
}
```

Each entry in the hash table is a `ListNode` in case of a collision the overflow entries can be added (chained) through the `next` pointer in `ListNode`.

Your task is to complete the `put`, `get`, `contains` and `remove` function in the class `HashTable`. Open the file HashTable.java, as you can see the constructor and the function `computeHash` are already provided.

Put & Contains

First implement the contains function to retrieve the index in the hash table use the `computeHash` function. Then traverse the list at this index and check if the key is already in the list. The function returns true if the key is found and false otherwise.

Next implement the Put function which makes use of the `contains` function to check if the key is not yet in the hash table. Use again the `computeHash` function to get the index for this key, then insert the key and value in the list at this index.

Note: As described in the preconditions you can assume that the key is not yet in the hash table when the function is called.

After implementing this two functions you can test them with the commands described below are also run the automated tests by un-commenting the annotation `@RunTests` at beginning of the `Main` class in Main.java. You should already pass the first test.

Get

To implement the `get` function again make use of the `computeHash` function to obtain the index. Then search through the list at this index to find the `ListNode` with the corresponding key. Finally return the value.

Note: As described in the preconditions you can assume that the key and its value are in the hash table when the function is called.

After implementing this function you should pass the second automated test.

Remove

Use the `computeHash` function to retrieve the index for this key. Once you have the index you can traverse the list and look for this key. If the key is not in the list ignore it and return. If the key is in the list you have to remove the corresponding `ListNode` from the list.

After implementing this last function you should pass all automated tests.

To test your implementation, you can use the following commands which are implement in the `Main` class:

```
put Adam 23
get Adam
contains Adam
remove Adam
```

The `put` command takes a key (Adam) and a value (23). The `get` command takes a key (Adam) and returns the value stored in the hash table. The `contains` command checks if a key (Adam) is in the hash table. The `remove` command removes the entry with the given key (Adam).

Note: The `put` and `get` command throw an exception if the key is already in the hash table for the `put` and correspondingly if the key is not in the hash table for the `get`. This is checked by the `assert` statement at the beginning of these two functions (see code template). The remove function will ignore it if a key is not available.

Solution of Problem 8.3.

```
/**
 * HashTable class of the Java program.
```

```
 */


class HashTable {
    ListNode[] data;
    int m;

    HashTable(int size) {
        m = size;
        data = new ListNode[size];
    }

    //pre: String of length > 0
    //post: returns a string which represents the hash of the string
    int computeHash(String s) {
      int sum = 0;
      int b = 1;
      for (int k = 0; k<s.length(); ++k) {
        sum = (sum + s.charAt(k) * b) % m;
        b = (b * 31) % m;
      }
      return sum;
    }

    //pre: hash table with m elements, where m >= 0
    //post: return true if the key is already in the hash table, otherwise false
    boolean contains(String key) {
        int h = computeHash(key);
        ListNode n = data[h];
        while (n != null && !key.equals(n.key)) {
            n = n.next;
        }
        return n != null;
    }

    //pre: hash table which does not contain this key (yet)
    //post: hash table contains the key (and value)
    void put(String key, int value) {
        assert(!contains(key));
        int h = computeHash(key);
        data[h] = new ListNode(key, value, data[h]);
    }

    //pre: hash table contains the key
    //post: return the value stored with the key
    int get(String key) {
        assert(contains(key));
        int h = computeHash(key);
        ListNode n = data[h];
        while (n != null && !key.equals(n.key)) {
            n = n.next;
        }
        return n.value;
    }
```

```
    //pre: hash table with m elements, where m >= 0
    //post: hash table does not contain an entry with this key
    void remove(String key) {
        int h = computeHash(key);
        ListNode n = data[h];
        if (n == null) {
            return;
        }
        ListNode p = null;
        while (n != null && !key.equals(n.key)) {
            p = n;
            n = n.next;
        }
        if (p == null) {
            data[h] = n.next;
        } else {
            p.next = n.next;
        }
    }
}
```