Data Structures and Algorithms (Spring 2022)
Assignment №1

IT University Innopolis

Contents

1 About this homework

1.1 Coding part

For practical (coding) part you have to provide you program as a single Java class file or C++ file that contains all necessary definitions and a `main` method.

The program should read from standard input and write to standard output unless mentioned otherwise.

1.1.1 Preliminary tests

For some coding parts a CodeForces system will be used for preliminary tests. You will have to specify a submission number in those systems as a proof that your solution is correct. Only correct solutions will be assessed further by TAs.

1.1.2 Code style and documentation

The source code should contain adequate internal documentation in the form of comments. Internal documentation should explain how the code has been tested, e.g. the various scenarios addressed in the test cases.

Do not forget to include your name in the code documentation!

All important exceptions should be handled properly.

Bonus code style points might be awarded for elegant solutions, great documentation and the coverage of test cases. However, these bonus code style points will only be able to cancel the effects of some penalties.

1.2 Theoretical part

Solutions to theoretical exercises have to be elaborate and clear. Write all solutions in a single document, then save or compile it as a PDF for submitting.

Do not forget to include your name in the document!

1.3 Submission

You are required to submit your solutions to all parts via Moodle.

For this assignment you will need to submit:

- Java or C++ class file(s) for the coding parts;
- link(s) to accepted submission(s) in CodeForces as a proof of correctness; please verify that your link has the following format: `https://codeforces.com/group/.../contest/.../submission/<submission_id>`
- a PDF file with solutions for the theoretical part.

Submit files as a single archive with your name and group number. For example, if your name is *Ivanov Ivan* and your group is *B21-05* then your archive should be named `B21-05_Ivanov_Ivan.zip`.

2 Coding part (60 points)

2.1 Implementing Data Structures

2.1.1 Circular Bounded Queue (5 points)

Warning: In this assignment you are not allowed to use data structures from standard libraries (such as `java.util`*).*

A Circular Bounded Queue is an extension of a regular Queue ADT with two properties:

1. the queue has bounded capacity (it cannot store more that a fixed number of elements);
2. when the queue is full (its size reaches its maximum capacity), pushing new elements forces the removal of elements that were pushed the oldest.

For example, pushing elements 1, 2, 3, 4, 5 into a queue with maximum capacity 3 will result in a queue with elements 3, 4, 5, since 1 and 2 will have been pushed first.

Consider the following interface (or an equivalent abstract class) for Circular Bounded Queue:

```
public interface ICircularBoundedQueue<T> {
    void offer(T value); // insert an element to the rear of the queue
                         // overwrite the oldest elements
                         // when the queue is full
    T poll(); // remove an element from the front of the queue
    T peek(); // look at the element at the front of the queue
              // (without removing it)
    void flush();   // remove all elements from the queue
    boolean isEmpty();  // is the queue empty?
    boolean isFull();   // is the queue full?
    int size();     // number of elements
    int capacity(); // maximum capacity
}
```

You need to implement a Circular Bounded Queue. Specifically:

1. Declare `ICircularBoundedQueue` interface or an equivalent abstract

class;

2. Provide **one of** the following implementations of a `ICircularBoundedQueue`:
   (a) Implement `LinkedCircularBoundedQueue` using singly linked lists (defined from scratch).
   (b) Implement `DoublyLinkedCircularBoundedQueue` using doubly linked lists (defined from scratch). You may use doubly linked lists with or without sentinel.
   (c) Implement `ArrayCircularBoundedQueue` using primitive arrays.
3. Make sure that `offer(value)` and `poll()` have time complexity of $O(1)$ either worst-case, average case, or amortized.
4. Write down the time complexity for each of the implemented interface methods, specifying if it is given for worst case, average case, or amortized. Justification is not necessary, but is welcome.

2.1.2 Bounded Stack (5 points)

Warning: In this assignment you are not allowed to use data structures from standard libraries (such as `java.util`).

A Bounded Stack is an extension of a regular Stack ADT with two extra properties:

1. the stack has bounded capacity (it cannot store more that a fixed number of elements);
2. when the stack is full (its size reaches its maximum capacity), pushing new elements forces the removal of elements that were pushed the oldest.

For example, pushing elements 1, 2, 3, 4, 5 into a stack with maximum capacity 3 will result in a stack with elements 3, 4, 5, since 1 and 2 will have been pushed first.

Consider the following interface (or an equivalent abstract class) for Bounded Stack:

```
public interface IBoundedStack<T> {
    void push(T value); // push an element onto the stack
                        // remove the oldest element
```

```
                        // when if stack is full
    T pop(); // remove an element from the top of the stack
    T top(); // look at the element at the top of the stack
             // (without removing it)
    void flush();   // remove all elements from the stack
    boolean isEmpty();  // is the stack empty?
    boolean isFull();   // is the stack full?
    int size();     // number of elements
    int capacity(); // maximum capacity
}
```

You need to implement a Bounded Stack. Specifically:

1. Declare `IBoundedStack` interface or an equivalent abstract class;
2. Implement `QueuedBoundedStack` using Circular Bounded Queue ADT. You will need two queues. *Hint: consider making either* push *or* pop *cheap (run in* $\Theta(1)$ *time).*
3. Explain your approach to the implementation of `QueuedBoundedStack`. Write down the explanation as a paragraph in the comments to the class `QueuedBoundedStack`.
4. Write down the time complexity for each of the implemented interface methods, specifying if it is given for worst case, average case, or amortized. Justification is not necessary, but is welcome.

2.1.3 Set with double hashing (5 points)

Warning: In this assignment you are not allowed to use data structures from standard libraries (such as ***java.util****).*

Consider a Set ADT[1] representing a collection of unique items:

```
public interface ISet<T> {
  void add(T item);          // add item in the set
  void remove(T item);       // remove an item from a set
  boolean contains(T item); // check if a item belongs to a set
  int size();                // number of elements in a set
  boolean isEmpty();         // check if the set is empty
}
```

You need to implement Set ADT using hashtables with open addressing. Specifically:

1. Declare `ISet` interface or an equivalent abstract class;
2. Implement the `DoubleHashSet` class that implements `ISet` interfaces using double hashing for collision handling. Use standard `hashCode()` method for the first hash function and expect a secondary `hashCode2()` method supported by the elements of a `DoubleHashSet`.
3. You are required to use primitive arrays to represent the hashtable. Dynamic resizing of the hashtable is not necessary, but is welcome. That is, you can initialize the hashtable with a fixed capacity, and throw exceptions when trying to insert into a full `DoubleHashSet`.
4. Write down the time complexity for each of the implemented interface methods, specifying if it is given for worst case, average case, or amortized. Justification is not necessary, but is welcome.

[1] you can think of Set ADT as a Map ADT without values, only keys.

2.2 Command Shell with Rollbacks (CodeForces)

Command line tools/interfaces (CLI) are useful to execute commands immediately, without interacting with GUIs. These tools accept commands as a text and execute them in an interactive environment (command shell). It is common that these tools provide powerful utilities and functionalities, and also they are often faster in executing commands compared to GUI tools. One feature that commands shells do not support typically[2] rollbacks. For instance, if we run a command to delete a file, we don't have a command to undo this action, even though GUIs commonly provide such functionality. Such an undo/redo functionality is also common for database management systems and text editors.

In this exercise, you will implement a simple command shell simulator with rollbacks. The command shell will only support commands to create/remove files and directories. You will use Set ADT to represent a set of existing files and a set of existing directories. Then, you will use Bounded Stack ADT to store the history of *states* of the environment for the last K commands (with K being the maximum capacity of the history stack).

Implement a simple command shell simulator in 3 stages as described below.

[2]however, see `nix-shell` for an example of an environment where rollbacks are supported quite well.

2.2.1 Bounded Commands Queue (15 points)

For this stage, the command shell simulator takes a list of commands and simply prints the last K commands. Each command is given in a separate line. You should not interpret the commands at this stage. You must use `ICircularBoundedQueue` from Section 2.1.1 to keep track of the last K commands.

Input format. The first line of input contains two numbers N ($0 \leq N \leq 10^6$) and K ($0 \leq K \leq 10^6$). K determines the maximum capacity for the bounded queue of commands. Each of the next N lines contains a command in the format described above, where `<FILENAME>` and `<DIRNAME>` can consist of any printable, non-whitespace ASCII characters. `<FILENAME>` cannot end in a forward slash (`/`).

Sample input:

```
6 3
NEW tmp.txt
NEW tmp/
NEW students.xlsx
REMOVE non_existing_file
NEW assignment.pdf
REMOVE tmp.txt
```

Output format. Print the last K commands as given in input (in the same order).

```
REMOVE non_existing_file
NEW assignment.pdf
REMOVE tmp.txt
```

2.2.2 Command shell without rollbacks (15 points)

For this stage, the command shell simulator should support the following commands:

1. `NEW <FILENAME>` — create a new file (if a file or directory named `<FILENAME>` does not exist);
2. `NEW <DIRNAME>/` — create a new directory `<DIRNAME>` (if a file or directory named `<FILENAME>` does not exist);
3. `REMOVE <FILENAME>` — remove a file if it exists;
4. `REMOVE <DIRNAME>/` — remove a directory if it exists;
5. `LIST` — print the list of all existing files and directories (in any order).

If a command cannot be executed, an error message should be issued. For example, when trying to execute `NEW some_file` in an environment where `some_file` or `some_file/` (directory with the same name) already exists, you should issue an error, mentioning that file already exists. Similarly, attempting to remove a non-existing file or directory leads to an error.

You must use `DoubleHashSet` from Section 2.1.3 to store the set of created files and a set of created directories.

Input format. The first line of input contains a number N ($0 \leq N \leq 10^6$). Each of the next N lines contains a command in the format described above, where `<FILENAME>` and `<DIRNAME>` can consist of any printable, non-whitespace ASCII characters. `<FILENAME>` cannot end in a forward slash (`/`).

Sample input:

```
7
NEW tmp.txt
NEW tmp/
NEW students.xlsx
REMOVE non_existing_file
NEW assignment.pdf
REMOVE tmp.txt
LIST
```

Output format. For each `LIST` command in the input there should be a corresponding line in the output, containing names of all files and directories, separated by whitespaces. Names of files and directories may appear in any order. Names of directories in the output must end with a forward slash (`/`).

Also, for each command that cannot be executed, an error message should be printed: `ERROR: cannot execute <COMMAND>`.

Sample output (matching sample input above):

```
ERROR: cannot execute REMOVE non_existing_file
tmp/ students.xlsx assignment.pdf
```

2.2.3 Command shell with rollbacks (15 points)

For this stage, the command shell simulator should be extended to support rollbacks. This includes an extra command in two variations:

1. `UNDO <N>` — undo the last N commands (not counting `UNDO` and `LIST` commands);
2. `UNDO` — undo the last command (equivalent to `UNDO 1`);

If an `UNDO` command cannot be executed (not enough history to undo), an error message should be issued without any effect on the state of the environment.

You must use `QueuedBoundedStack` from Section 2.1.2 to store the history of states (each state consists of a set of files and directories).

Input format. The first line of input contains two numbers N ($0 \leq N \leq 10^6$) and K ($0 \leq K \leq 10^6$). K determines the maximum capacity for the bounded stack of states (maximum history depth). Each of the next N lines contains a command in the format described above, where `<FILENAME>` and `<DIRNAME>` can consist of any printable, non-whitespace ASCII characters. `<FILENAME>` cannot end in a forward slash (`/`).

Sample input:

```
9 5
NEW tmp.txt
NEW tmp/
NEW students.xlsx
UNDO
NEW assignment.pdf
REMOVE tmp.txt
LIST
UNDO 2
LIST
```

Output format. For each `LIST` command in the input there should be a corresponding line in the output, containing names of all files and directories, separated by whitespaces. Names of files and directories may appear in any order. Names of directories in the output must end with a forward slash (`/`).

Also, for each command that cannot be executed, an error message should be

printed: `ERROR: cannot execute <COMMAND>`.

Sample output (matching sample input above):

```
tmp/ assignment.pdf
tmp.txt tmp/
```

3 Theoretical part (40 points)

3.1 Big-O notation (10 points)

Use big-O notation give a tight asymptotic upper bound for the following functions. For full grade, you must provide a complete proof using definition of big-O notation. You may use properties of big-O only if you reference them. It is not necessary to prove that the provided bound is tight.

1. $10n \log n + 500n + n^2 + 123$
2. $n^{\frac{9}{2}} + 7n^4 \log n + n^2$
3. $6^{n+1} + 6(n+1)! + 24n^{42}$

Here's a sample proof:

Example 3.1.1. $n^2 + 2n + 1 = O(n^2)$.

Proof. By definition of big-O notation, it is sufficient to show that there exist constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$ we have $n^2+2n+1 \leq c \cdot n^2$. It is clear that $n^2 \geq 2n + 1$ for large enough n (more precisely, for $n \geq 3$). Thus, let $n_0 = 3$ and $c = 2$. Then, for $n \geq 3 = n_0$ we have

$$c \cdot n^2 = 2 \cdot n^2 = n^2 + n^2 \geq n^2 + (2n + 1)$$

□

Alternative proof. By Exercise 3.1-2 of Cormen et al. we have $(n+a)^b = \Theta(n^b)$. Thus,

$$n^2 + 2n + 1 = (n + 1)^2 = \Theta(n^2)$$

Then, by Theorem 3.1 of Cormen et al. we conclude that

$$n^2 + 2n + 1 = O(n^2)$$

□

3.2 Dynamic binary search (20 points)

Consider using several arrays to represent a set of n elements, supporting `insert` and `search` operations. Let $k = \lceil \log(n+1) \rceil$, and let the binary representation of n consist of k binary digits b_i, $n = \overline{b_{k-1} b_k \dots b_1 b_0}$. Then, for each digit $b_j = 1$ in the binary representation we have a full sorted array of size 2^j. A list of such arrays in increasing order of their sizes represents a set of n elements.

Hint: use k when performing amortized analysis.

3.2.1 Search

Consider the following implementation of `search` on an aforementioned representation of a set. Perform asymptotic analysis for the worst-case running time of `search`. For full grade, you must provide the details of your calculations:

1. cost and number of execution times per instruction;
2. worst-case running time as a function of input size (value of variable n);
3. asymptotic complexity analysis for the running time.

```
function search(value) {
    for array in arrays {
        int l = 0, r = array.length;
        while (l < r) {
            int mid = (l + r) / 2;
            if (array[mid] < value) { l = mid + 1; }
            else if (array[mid] > value) { r = mid; }
            else { return true; }
        }
    }
    return false;
}
```

3.2.2 Inserting

Consider the following implementation of `insert`. Perform amortized analysis for the running time of `insert`. For full grade, you must provide the details of your calculations:

1. cost and number of execution times per instruction;

2. worst-case running time as a function of input size (value of variable n);
3. asymptotic complexity analysis for the worst-case running time;
4. amortized time complexity analysis with a proof using either accounting method or potential method.

Hint: for similar (but simpler) amortized analysis see examples with binary counter in Chapter 17 of Cormen et al.

```
function insert(value) {
  values = new array of size 1;
  values[0] = value;
  insertMany(values);
}
function insertMany(values) {
    if (arrays is empty) { arrays.add(values); }
    else {
      head = arrays[0];
      if (arrays.head.size > values.size) {
        arrays.add(values)
      } else {
        merged = new array of size (values.size + head.size);
        i = 0; j = 0;
        for (k from 0 to merged.size - 1) {
          if (j >= head.size)              { merged[k] = values[i++];
          } else if (i >= values.size)     { merged[k] = head[j++];
          } else if (values[i] <= head[j]) { merged[k] = values[i++];
          } else                           { merged[k] = head[j++]; }
        }
        arrays.remove(0);
        insertMany(merged);
      }
    }
}
```

3.3 Recurrences and Master Theorem (10 points)

Running time $T(n)$ of processing n data items with some unspecified algorithm is described by the following recurrence (where k and c are constants):

$$T(n) = \sqrt{k} \cdot T\left(\frac{n}{k^2}\right) + c \cdot \sqrt[3]{n}; \quad T(1) = 0$$

It is also known that $c^6 > k^5$. Answer the following questions:

1. What is the computational complexity of this algorithm (express complexity in a closed form, i.e. without recurrence, in terms of c, n and k)?
2. Which case of Master Theorem applies (if any)? Why?

The answer should necessarily include calculations to justify your claim.