Data Structures and Algorithms
Spring 2023
Homework Assignment №1

IT University Innopolis

February 13, 2023

Contents

1 About this homework

1.1 Coding part

For practical (coding) part you have to provide a program for each of the exercises as a single Java 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. Only submissions that pass all tests will be assessed further by TAs.

1.1.2 Code style and documentation

Code style (such as helpful comments, good naming and formatting, proper error handling, correctly used abstractions) will be graded in this assignment. More precisely, each coding exercise allocates 20% of its weight for the code style. Note that code style is graded only for correct solutions (that pass all tests on CodeForces).

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.

Data structures defined and implemented in the coding exercises should be generic (in Java) or template (in C++).

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

For this assignment you will need to submit:

Coding Part Submit on CodeForces.

1. The latest submission Submitted on CodeForces before the deadline will be considered for each exercise, so no additional submission on Moodle is required.
2. The submitted file must have student's name in the first line of the file in a comment.

3. Code style (such as helpful comments, good naming and formatting, proper error handling, correctly used abstractions) will be graded in this assignment.

Theoretical Part Submit on Moodle.

1. Typeset your solutions in Word or LaTeX
2. Include the original problem statements
3. Clearly indicate final answer in each problem.
4. Follow this file naming scheme: `<Firstname><Lastname>_homework_<N>.<ext>` Submit both the PDF and the corresponding source file (e.g. `NikolaiKudasov_homework_1.pdf` and `NikolaiKudasov_homework_1.tex`)
5. Do not submit archives!

2 Coding part (50 points)

2.1 Balanced Delimiters (20 points)

You are given a text that contains multiple types of delimiters that act like parentheses. Implement a program that checks if all delimiteres in the input text are properly balanced.

Requirements

Your solution **must** provide an implementation of `LinkedStack` (implementation based on linked lists) and use it to solve the problem. You **cannot** use any standard data structures such as `ArrayList` and `LinkedList` in Java or `std::vector` and `std::list` in C++.

Input format

First line of input contains numbers N ($0 \leq N \leq 100$) of types of delimiters and K ($1 \leq K \leq 10^5$). Each of the next N lines contains two tokens — an openning delimiter and closing delimiter of the the same type. The last K lines contain a sequence of tokens (arbitrary, not only delimiters) separated by whitespaces to check for balanced delimiters.

Output format

If the input sequence of tokens has properly balanced delimiter, then output:

`The input is properly balanced.`

Otherwise, you must specify the leftmost location in the sequence that contains a mistake in the input. The error message should be one of the following:

1. `Error in line <line>, column <column>: unexpected closing token <token>.`
2. `Error in line <line>, column <column>: expected <token> but got <token>.`
3. `Error in line <line>, column <column>: expected <token> but got end of input.`

Examples

Input:

```
3 1
( )
[ ]
{ }
function example ( int a [ ] , int n ) { return a [ n - 3 ] ; }
```

Output:

```
The input is properly balanced.
```

2.2 Accounting for a Café (30 points)

A local café would like to keep track of its earnings. Your task is to implement a simple accounting program for the café. The program is given a sequence of entries, one per sold item. Each entry contains a timestamp, receipt number, sold item title and its cost. The task is to compute for each date (year, month and day) total amount of money earned, as well as number of unique customers (number of unique receipts).

Requirements

You **must** provide an implementation of `HashMap` and use it to solve the problem. You **must** implement a rehashing strategy (dynamic hashtable) to keep the load factor constant. You may use primitive arrays, as well as `ArrayList`, `LinkedList` in Java and `std::vector`, `std::list` in C++.

Input format

First line of the input contains a single number N ($0 < N \leq 50000$) — the number of entries in the input. Following N lines each describe one entry. Each entry consists of 5 fields separated by spaces:

- date (`YYYY-MM-DD` format),
- time (`HH:MM:SS` format),
- receipt ID (`#<ID>` format, `ID` may contain digits, letters and dash (`-`)),
- cost (`$<number>` format) and
- item title (no specified format).

Entries are not guaranteed to appear in chronological order.

Output format

The program should output K lines, where K is the number of different dates in the input. For each date the program should output a line with the date (`YYYY-MM-DD`), total cost (`$<number>`), and the number of unique receipt IDs (`<number>`). Output is not required to appear in chronological order.

Example

Input:

```
5
2019-01-24 15:38:17 #495-GE $99.80 CAPPUCCINO
2019-01-24 15:38:17 #495-GE $34.95 ESPRESSO RISTRETTO
2021-08-15 01:46:42 #272-YZ $80.45 CAFFE MOCHA
2019-01-24 15:38:17 #495-GE $30.82 LATTE MACCHIATO
2019-01-24 15:38:17 #495-GE $50.00 AMERICANO
```

Output:

```
2021-08-15 $80.45 1
2019-01-24 $215.57 1
```

3 Theoretical part (50 points)

3.1 Asymptotics (32 points)

1. Prove or disprove the following statements. You must provide a formal proof. You may use the definitions of the asymptotic notations as well as their properties and properties of common functions, as **long as you properly reference** them (e.g. by specifying the exact place in Cormen where this property is introduced)!

   (a) $n^3 \log n = \Omega(3n \log n)$

   (b) $n^{\frac{9}{2}} + n^4 \log n + n^2 = O(n^4 \log n)$

   (c) $6^{n+1} + 6(n+1)! + 24n^4 2 = O(n!)$

   (d) There exists a constant $\varepsilon > 0$ such that $\frac{n}{\log n} = O(n^{1-\varepsilon})$.

2. For each of the following recurrences, apply the master theorem yielding a closed form formula. You must specify which case is applied, explicitly check the necessary conditions, and provide final answer using Θ-notation. If the theorem cannot be applied you must provide justification.

   (a) $T(n) = 2T(n/3) + \log n$

   (b) $T(n) = 3T(n/9) + \sqrt{n}$

   (c) $T(n) = 4T(n/9) + \sqrt{n} \log n$

   (d) $T(n) = 5T(n/5) + \frac{n}{1000}$

3.2 Segmented List (18 points)

Recall the methods of the List ADT:

```
public interface List<E>
{
  int size();
  boolean isEmpty()
  void add(int position, E element);
  E remove (int position);
  E get(int position);
  E set(int position, E element);
}
```

Consider a `SegmentedList` implementation, that consists of a doubly linked list of arrays (segments) of fixed capacity k:

1. all segments, except the last one, must contain at least $\lfloor \frac{k}{2} \rfloor$ elements;
2. when adding at the end of `SegmentedList`, the element is added in the last segment, if it is not full; otherwise, a new segment of capacity k is added at the end of the linked list;
3. when adding x at any position, the doubly linked list is scanned from left to right (or right to left) to find the corresponding segment; if the segment is full, it is split into two segments with $\frac{k}{2}$ elements each, and x is inserted into one of them, depending on the insertion position.
4. when removing an element, only elements in its segment are shifted; if the segment size becomes less than $\lfloor \frac{k}{2} \rfloor$, then we rebalance:
   (a) if it is the last segment, do nothing;
   (b) if it is a middle segment s_i and the sum $m = \mathsf{size}(s_{i-1}) + \mathsf{size}(s_i) + \mathsf{size}(s_{i+1}) < 2k$, then replace these three segments with two segments of size $\frac{m}{2}$;
   (c) if it is a middle segment s_i and the sum $m = \mathsf{size}(s_{i-1}) + \mathsf{size}(s_i) + \mathsf{size}(s_{i+1}) \geq 2k$, then replace these three segments with tree segments of size $\frac{m}{3}$;
   (d) if it is the first segment, and there exist two segments after, then proceed similarly to previous two cases;
   (e) if it is the first segment and only one segment after it exists, move as many elements from the last segment to the first one as possible; if the last segment becomes empty, remove it.

Perform the following analysis for the `SegmentedList`:

1. Argue that the worst case time complexity of `add(i, e)` is $O(\frac{n}{k} + k)$.
2. Argue that the worst case time complexity of `remove(i)` is $O(\frac{n}{k} + k)$.
3. What value of k should be chosen in practice? Why?

3.3 Segmented Queue (+1% extra credit)

Consider `SegmentedList` used as a Queue:

1. we enqueue (`offer(e)`) elements by adding them to the end of the segmented list;
2. we dequeue (`poll()`) elements by removing the first element of the segmented list;
3. in an attempt to make `remove` more efficient, we do not perfom shifts when removing an element in a segment (so there can be a gap on the left of the cells where values are stored); `add` will create a new segment when it reaches the right end of the last segment, regardless of whether there is empty space to the left;
4. rebalancing part of `remove` remains, to keep the invariant that each segment has at least $\lfloor \frac{k}{2} \rfloor$ elements.

Perform amortised analysis for arbitrary sequences of `offer(e)` and `poll()` operations applied to an initially empty queue, implemented using `SegmentedList` in a way described above. Show that amortized cost for any such sequence of length N is $O(N)$ (does not depend on the value of k). You **must** use the accounting method or the potential method.