Assignment 1
COMP 250, Fall 2021

Posted: Fri., Sept. 24, 2021
Due: Friday, Oct. 8, 2021 at 23:59 (midnight)

[document last modified: September 24, 2021]

General instructions

- T.A. office hours will be posted on mycourses under the "Office Hours" tab. Most office hours will be on zoom, but we're also trying to schedule some office hours in person too. If the zoom link is missing or doesn't work, please email the T.A. and the instructor, so the problem can be fixed ASAP. The T.A. emails can be found in the classlist. *Other than time critical emergencies such as this, you should not email the T.A.s!*
- Our plan is to use **Ed Lessons** both for assignment submission and grading. This is the first time a cs.mcgill course is using this software, and so there may be glitches. Please let us know if any issues come up so that we address them ASAP.
- Search for the keyword *updated* to find any places where the PDF has been edited since the original posting.
- This document provides you with a few examples that you can use to test your code. Your code will be evaluated on more challenging and comprehensive examples, however. Therefore we strongly encourage you to come up with more creative test cases and to share these test cases on the discussion board.

  There is a crucial distinction between sharing test cases and sharing solution code. We strongly encourage the former, whereas the latter is a serious academic offense.
- Get started early, so that you can avoid long T.A. lineups close to the deadline!
- **Policy on Academic Integrity:** See the Course Outline Sec. 7 for the general policies. We will release a checklist PDF that specifies more specifically what is allowed and what is not allowed on the assignment(s).
- **Late policy:** Late assignments will be accepted up to two days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Please see the submission instructions at the end of this document for more details.

1 Introduction

In this assignment you will get some valuable experience working with the arrays and the Java `LinkedList` class, and also practice your basic Java programming skills. The scenario for this assignment is that we have a set of "strings" stored in a memory.

We will use the term string here in two ways. One is a sequence of characters, which will be stored in the memory. The other is a Java `String` object, which will be used for getting or putting the string from/into memory.

The memory consists of an array of characters, and so each string will be stored in an interval of positions in that array. The goal is to put strings into the memory array, get strings from the memory array, or remove strings from the memory array. In doing so, we need to keep track of which strings are stored in memory (each string will have an id) and where the strings are. We will use a linked list for this bookkeeping task.

In terms of learning goals, the assignment in not really about strings. Rather, it is about how to store things (e.g. Java objects) in a memory, and how to keep track of starting and ending positions of each thing. We use strings because these things make program easier to write and debug.

2 Your task: implement the `Memory` class

Write a class `Memory` as described below. The class defines a memory array and a linked list of objects. These objects define the intervals in the array where the strings are stored. The class has various methods for putting, getting, and removing a string to/from the memory.

2.1 `StringInterval` inner class

The `Memory` class has an inner class `StringInterval`. An instance of this string interval class defines a location in memory where the characters of a string are stored. The `StringInterval` class has three member fields:

- `int id` – this is a unique integer identifier for the string; no two strings that are simultaneously in memory have the same id.
- `int start` – This is the index in memory holding the first character of the string.
- `int length` – This is the number of characters in the string.

Note that we do not store the characters in the `StringInterval` object. Instead, we store the characters in the array.

The `StringInterval` class also has a contructor method. The parameters of the constructor are `int` variables `id`, `start` and `length` in that order. Normally one would make the inner class and its fields private. But for grading purposes, we make them public.

2.2 `Memory` class fields

The fields of the `Memory` class are as follows, and for grading purposes they must also all be public.

- `LinkedList<StringInterval> intervalList` – a list of intervals, stored in order of their location in memory i.e. the `start` values must be increasing.
- `char[] memoryArray` – an array that holds the characters in memory
- `static int idCount` – a field that counts the number of ids that have been issued; this field is used to give a new string an id. This field *must* be initialized to 0, and so the first string that is put into a memory will have id 0.

2.3 `Memory` class methods

The methods of the `Memory` class that you are required to implement also must all be `public`. These methods are:

- `Memory(int length)` – a constructor that has one `int` parameter which specifies the length of the underlying `char` array. *This constructor must reset the* `static idCount` *variable to 0*. This is important because our tester code will test multiple memories. Note that string intervals in different memories will share the same id, which is fine.
- `get(int id)` – returns a Java `String` object that corresponds to a string stored in memory, namely the string associated with that id. This method assumes the id is valid, namely it is one of the id's in the linked list; if the id is not valid, then the method returns `null`.
- `get(String s)` – returns the id associated with that string, assuming that exact string is stored in memory and is associated with some `String Interval` instance. (Substrings don't count.) If that exact string is not in memory, then the method returns the value -1.
- `remove(int id )` – removes the string, if indeed the id corresponds to a string stored in the memory. Note that you do not need to "erase" the characters of the string from the array, e.g. by writing over them with the ASCII NUL (0) character. Rather, you just need to remove the string interval from the linked list. The method should return the Java `String` object associated with the given id, if it exists in the list, and otherwise return `null`.

  If the removed element is not the last in the list, then the remove method will leave a hole or gap in memory. This hole is captured by the linked list, namely the interval corresponding to the removed element is gone. As mentioned above, however, the characters of the removed string are not erased but rather they remain in memory; they are erased only if written over by a `put` operation (see below).
- `remove(String s)` – removes the argument string, if that string corresponds to a string stored in memory, and the id of that string should be returned. Otherwise, return -1. So the return value is an `int`.
- `put(String)` – adds a new string to memory. A string interval object is added to the linked list at the appropriate position, and an id must be generated. You *must* use the static field of the class to generate these `id`'s. See the examples.

  The characters of the string must also be written into memory. Specifically, the characters are written into the memory array starting at the *first available location(s)* in memory, without writing over any

of the other strings that are currently in memory. Note that the start positions of the intervals in the list must be increasing. Examples will be given below.

If no available memory "gap" is found that is large enough for the new string, then the method should call a method defragment (described next) which removes all gaps by shifting the strings backwards in memory. Then, it should check again if there is room available, namely after the last string in the list. If there is room available, the string should be stored in memory directly after the last string, which is consistent with the "first available location" policy above.

If the string is successfully put into memory, then this method should return a unique id for that string. Otherwise (if there is no way to find room in memory for this string) then the method should return the value -1 to indicate that it failed to put the string in memory.

- `defragment()` – gets rid of the gaps that may appear in memory. It does so by shifting the string intervals backwards such that the first one starts at memory location 0 and there are no gaps between subsequent strings. Note that this requires moving the characters in memory, as well as changing the `start` field of the `StringInterval` objects. The method does not return anything. (Its return type is void.)

  As with `remove`, characters in memory do not have to be erased when defragmenting. The characters left behind do not matter. We mention this in case you would like to compare the contents of your memories on different examples.

Feel free to add any helper methods that you may need.

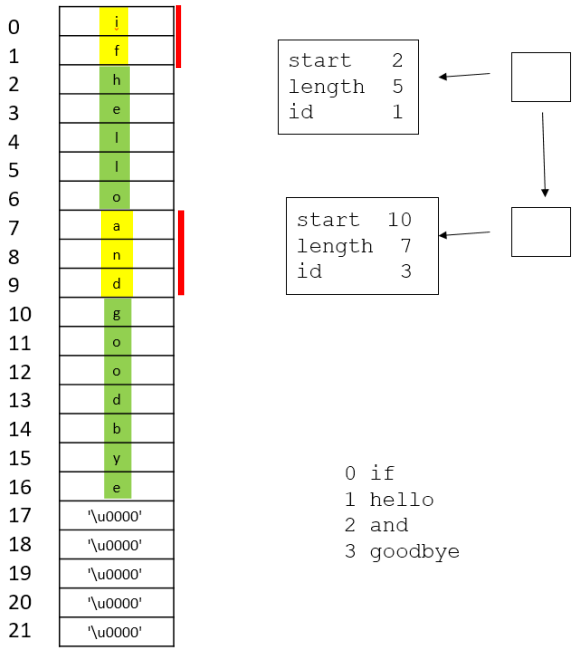
2.4 Point Distribution

- `StringInterval` inner class **(0 points)**
- `Memory(int)` constructor **(10 points)**
- `get(String)` **(10 points)**
- `get(int)` **(10 points)**
- `remove(int)` **(10 points)**
- `remove(String)` **(10 points)**
- `put(String)` **(40 points)**
- `defragment()` **(10 points)**

2.5 Examples

The first example adds (`put`'s) four strings and then removes two of them. See instructions on the left in the figure. Note that the remove method does not erase the characters in memory.

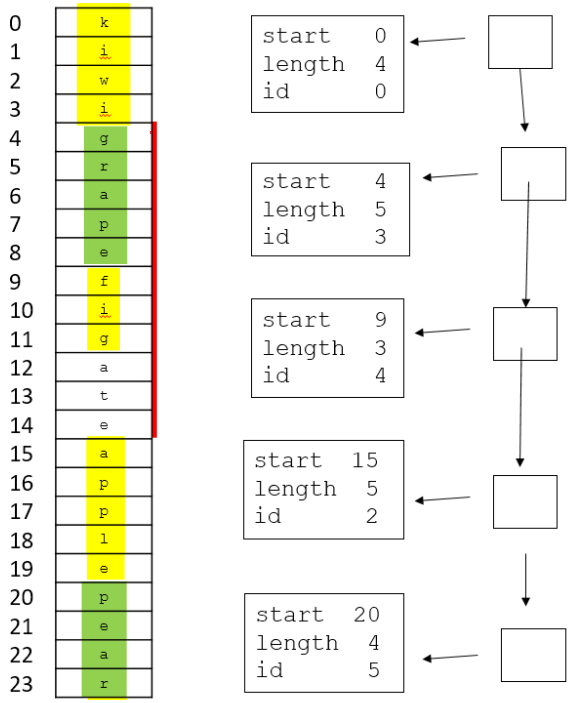
```
Memory memory = new Memory(22);
memory.put("if");
memory.put("hello");
memory.put("and");
memory.put("goodbye");
memory.remove(0);
memory.remove(2);
```



The second example adds six strings and removes one. Note that the new strings sometimes partly overwrite removed strings in memory. Also note where the new strings are added: `grape` and `fig` fit into gaps, whereas `pear` had to be placed at the end.

```
Memory memory = new Memory(24);
memory.put("kiwi");
memory.put("pomegranate");
memory.put("apple");
memory.remove("pomegranate");
memory.put("grape");
memory.put("fig");
memory.put("pear");
```

```
0 kiwi
1 pomegranate
2 apple
3 grape
4 fig
5 pear
```



3 Submission

- You should only submit one file on **Ed** as follows :
  - `Memory.java`
- This class must be part of the default package. Therefore you must remove any package declaration you have at the top. If you don't, your code will not compile.
- We will check that this file imports only what you need (`LinkedList`). If you use other imports, your code will not pass this check and you will not be able to submit.
- Do not submit any other files (like `.class` files).
- Whenever you submit your file to **Ed**, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. As mentioned earlier, we will test your code on a much more challenging set of examples. We highly encourage you to test your code thoroughly before submitting your final version.
- You may submit multiple times. Your assignment will graded using your most recent submission.
- If you submit code that does not compile, you will automatically receive a grade of 0. Since we grade only your latest submission, you must ensure that your latest submission compiles!
- If anything is unclear, pleasure clarify with us by posting a question on the Ed discussion board.

Good luck and have fun!