ACS-2947-001
Assignment 4
Due by Tuesday, December 5, 11:59 PM

Instructions

- Submit your `.java` files (together in an `Assign4.zip` file) via Nexus.
- Include your name and student number as a comment in every file.
- Document the classes using Javadoc notation.
- Include comments as needed and use exception handling where necessary.

PART A – Linear Probing (35 marks)

Develop a program named that analyzes the words in the given file `PartA.txt` to determine the following:

- The letters with the lowest and highest frequencies.
- The words with the lowest and highest frequencies.
- The personal pronouns with the lowest and highest frequencies. The following words are considered personal pronouns: i, we, you, he, she, it, they

Note that your program must exclude all sentence punctuation characters when parsing.

1. Using the provided `Map` and `Entry` interfaces, provide the `ProbeHashMap` implementation. Use the `AbstractHashMap` and `AbstractMap` classes from your notes/text as a base.
   - In a `PartA_Driver` class, read the file `PartA.txt` and use your `ProbeHashMap` to store each word and its frequency and another map to store each character and its frequency.

2. Create an `OrderWordsByFrequency` comparator that orders entries of words and their frequency based on the frequency (lower frequency is ordered first, i.e., *ascending* order). Words with the same frequency must also be sorted in *ascending* order of their natural ordering.

3. Create an `OrderLettersByFrequency` comparator that orders entries of alphabetical letters and their frequency based on the frequency (lower frequency is ordered first, i.e., *ascending* order). Characters with the same frequency must also be sorted in *ascending* order of their natural ordering.

4. Create a class named `MergeSort` that uses the merge-sort algorithm (from the class) and a comparator to sort elements of an array. Overload your sorting algorithm to work both with a default comparator and with a specified comparator. Make sure to reuse code.

5. In your `PartA_Driver` class
   - Create a static generic `findMaxLeast` method that, given a map, can return the entry with either the maximum frequency or the least frequency based on the comparator provided to the method. This method is generic so that it can work with any map regardless of the key's type.
   - Create a static generic `findCategoryMaxLeast` method that works similarly to the `findMaxLeast` method but additionally accepts a list of possible Keys for which we want to find the entry with either maximum or least frequencies. For example, you will use this method to find which of the set of pronouns has the maximum or least frequencies.
   - Use the `findMaxLeast` method to find the words or letters with the maximum and least frequencies as shown in the sample. Use `findCategoryMaxLeast` to find the pronouns with the maximum and least frequencies.
   - Display your result for the analysis of the given text as in the sample output, including the frequencies of the most (or least) occurring letters, words, and pronouns.

Sample Output
```
Text Analyzer
Total number of distinct words: 8026
Total number of distinct letters: 26
Most occurring character: e
Least occurring character: z
Most occurring word: the
Least occurring word: ab
Most occurring pronoun: i
Least occurring pronoun: you
```

Notes:
- To remove sentence punctuation, use:
  - the useDelimiter() method of Scanner e.g.,
    `f.useDelimiter("[^a-zA-Z]+");`
  - or replaceAll() method of String e.g.,
    `word = word.replaceAll("[^a-zA-Z]+", "");`
- You will have to convert the Iterable of all entries to an array (iterate through and add each element)
- Before our lecture on Sorting, you can temporarily sort using `Arrays.sort`
- For the `findMaxLeast` and `findCategoryMaxLeast` methods, you may use a Boolean variable to specify whether the method returns the entry with the maximum or the least frequency when it is called. Note that these methods will use the `MergeSort` class to sort the entries.

PART B – Separate Chaining (45 marks)

Create a version of the `ChainHashMap` that **uses a linked list for each bucket.**

Note that the ChainHashMap implementation in Lab 7 used an UnsortedTableMap as bucket, so the implementation here would be different.

1. Implement the `Map` interface using the interfaces and abstract classes from Part A. Name your class `LinkedChainHashMap`.
    - Use a linked list as the auxiliary data structure that holds entries of colliding keys
        - You may choose to use either the LinkedPositionalList from the class or Java's LinkedList
    - Add a method named `getCollisions` that returns an integer representing the number of collisions that occurred in your hashmap.

2. Create a class named `PostalCode` that stores:
    - Strings for the `code`, `area`, and `province`
    - double for `latitude` and `longitude`
    - include a natural ordering of postal codes in alphabetical order of the `code`

3. Create an `OrderByLongitude` comparator that orders postal codes from *west to east*.

4. Create a class named `QuickSort` that uses the quicksort algorithm (from class) and a comparator to sort elements of a collection (stored as a queue or an array). Overload your sorting algorithm to work both with a default comparator and with a specified comparator. Make sure to reuse code.

5. In a `PartB_Driver` class, create a hash map of Postal Codes.
    - read in the `PartB.txt` file
    - set each line as an instance of `PostalCode`
    - store each instance in your map, using the `code` (stored in the first column of the given data) as the `key` and `PostalCode` instance as the `value`

6. In your output, display
    - The total number of postal codes
    - The number of collisions that occurred in the hashmap
    - An option for the user to sort by code or longitude

7. After displaying the output, include 2 lines of code to do the following. You may use any of the postal codes in the sample data for this.
    - Show the output of inserting an entry where the same key previously exists in the map.
    - Show the output of removing an entry where the key exists in the map.

Sample Output

```
Total Number of entries: 1649
Number of collisions: 218
Display by code (C) or Longitude (L) (any other key to quit)
...
```

[display the value associated with each postal code or quit accordingly]

Notes:

- Your table (array) will hold a linked list of map entries e.g., declare
  `private LinkedList<MapEntry<K, V>>[] table;`
    - Use the for-each loop, since elements are internal map entries you can make any updates directly
- Declare your hashmap as a `LinkedChainHashMap` in the driver (i.e. not `Map`) and understand why this is necessary
- Use `nextLine()` of Scanner and `split(",")` of String to parse the input data
- The number of collisions should be ~150 to ~450 but may vary
    - Run several times to check. Think about why the number of collisions may differ at each run
- The marker will be testing all methods, be sure to also test remove() and put() where an entry with key k exists
- You may choose to use either `quickSort` or `quickSortInPlace`
- You will have to convert the Iterable of all entries to a queue or an array (iterate through and add each element). Use the LinkedQueue from the lectures if using a queue.
- **Before our lecture on Sorting, you can temporarily sort using `Arrays.sort`**
- Overload your sorting algorithm to work both with a default comparator and with a specified comparator. Make sure to reuse code.

Submission

Submit your **`Assign4.zip`** file that includes all the assignment files (`Map.java, Entry.java, AbstractMap.java, AbstractHashMap.java, ProbeHashMap.java, PartA_Driver.java, MergeSort.java, DefaultComparator.java, OrderWordsByFrequency.java, OrderLettersByFrequency.java, PartA_Driver.java, LinkedChainHashMap.java, QuickSort.java, OrderByLongitude.java, PostalCode.java, PartB_Driver.java, any` other accompanying files/classes that you use e.g., `PositionalList.java`, `LinkedQueue.java`, and the provided data files for part A and part B) via **Nexus**.