Exercise 1: Set Implementations

A set is a data structure that stores unique elements, with no repetitions. In either Python or Java, implement sets of words. Follow the following steps:

* 1. Implement three sets using the following data structures:
     1. Linked list
     2. Binary tree
     3. Hash table
  2. For each set implementation, implement the following methods:
     1. **boolean add(String word)**: Takes a word as input and adds the word to the set if the word is not already there. Returns true if the word was added and returns false if the word is already present in the set.
     2. **boolean contains(String word)**: Takes a word as input and returns whether the word is present in the set (true) or not (false).
     3. **int size(): Returns the size of the set.**
  3. Read and tokenize an input text file (use [this one](https://canvas.eee.uci.edu/courses/39841/files/15593595/download?wrap=1)[Download this one](https://canvas.eee.uci.edu/courses/39841/files/15593595/download?download_frd=1) pride-and-prejudice.txt for your experiments). Insert all the words in all three sets. After inserting, print the number of words that are present in the sets. Note that, for each set, this number should be the same.
  4. Read and tokenize another text file (use [this one](https://canvas.eee.uci.edu/courses/39841/files/15593596/download?wrap=1)[Download this one](https://canvas.eee.uci.edu/courses/39841/files/15593596/download?download_frd=1) words-shuffled.txt). Search each of these words in each set, and, at the end, print out how many words do **not** exist in the sets.
  5. Conduct experiments to assess the relative performance of the set implementations. At least the following experiments should be done:
     1. Measure how much time (in nanoseconds) it takes to insert each word in the sets. You should plot a graph that shows as independent variable (x-axis) the number of words already inserted, and as dependent variable (y-axis) the time it took to insert it.
     2. Measure how much time it takes to search for a word. You should detect the worst case, the best case, and the average of all words.
  6. Write a one-page report with your findings. In your report you should also make a complexity analysis of your code and correlate that with the runtime performance you measured. Is it what you expected? If so, explain; if not, explain what factors may be at play that might affect the gap between theory and practice.
  7. Upload the zip file in the Canvas. Your zip file must contain both the source code and the report.

Note:

* The input file might contain Unicode characters.
* The input file can be very large and might not fit in the memory. Therefore, files should be read and tokenized line by line.
* While tokenizing the file you can consider anything other than alpha-numeric characters as delimiters. For example, *“Hello! Can I get a 7Up?*” can be tokenized as - *“Hello”, “Can”, “I”, “get”, “a”, “7Up”*.*“Twentieth-century”*can be tokenized as *“Twentieth”, “century”.*
* Data structures must be implemented from scratch. You should not use any library data structures. For example, you cannot use *List, ArrayList, HashTable, HashSet*in Java nor Dictionary, List, Set in python.
* You can re-use your own data structures across dictionaries. For example, you can re-use your own linked list to implement the hash table.
* For the experiments, you should repeat each one at least 10 times and report averages and standard deviations. Use Excel or some other tool that will help you with the statistics.
* It's not enough to mindlessly type lines of code copied from elsewhere. You need to make sure you understand what each line of code is doing. You will be quizzed about it.