CS/IT 200

Lab 10: Autocomplete

Due Date: One week after assigned lab date

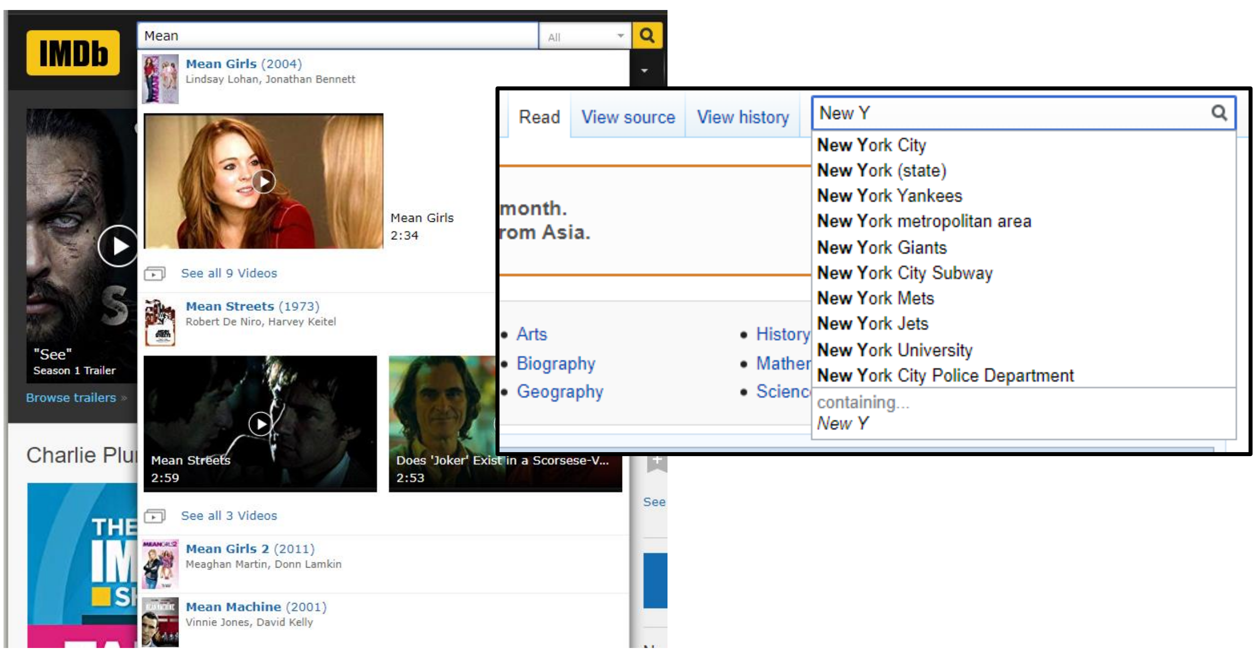
Submission

- Submit all code in a single zip file to the Lab 10 assignment folder on Kodiak.

Goal: Use binary search trees to write a program to implement autocomplete for a given set of *N* strings and positive weights. That is, given a prefix, find all strings in the set that start with the prefix, and present them in descending order of weight.

Problem

Autocomplete is an important feature of many modern applications. As the user types, the program predicts the complete *query* (typically a word or phrase) that the user intends to type. Autocomplete is most effective when there are a limited number of likely queries. For example, the Internet Movie Database uses it to display the names of movies as the user types; search engines use it to display suggestions as the user enters web search queries cell phones use it to speed up text input.



In these examples, the application predicts how likely it is that the user is typing each query and presents to the user a list of the top-matching queries, in descending order of weight. These weights are determined by historical data, such as box office revenue for movies, frequencies of search queries from other Google users, or the typing history of a cell phone user. For the purposes of this assignment, you will have access to a list of all possible queries and associated weights (and these queries and weights will not change).

Acknowledgement: This assignment is adapted from *Autocomplete Me*, created by Matthew Drabick and Kevin Wayne, Princeton University, and posted on EngageCSEdu (submitted by Ananda Gunawardena).

The performance of autocomplete functionality is critical in many systems. For example, consider a search engine which runs an autocomplete application on a server farm. According to one study, the application has only about 50ms to return a list of suggestions for it to be useful to the user. Moreover, in principle, it must perform this computation *for every keystroke typed into the search bar,* and *for every user!* (You won't have to handle each individual keystroke.)

In this assignment, you will implement autocomplete by using binary search trees to find the set of queries that start with a given prefix; and sorting the matching queries in descending order by weight.

Part I – Autocomplete Term

In `autocomplete.py`, write a class `Term` that represents an autocomplete term: a string query and an associated positive real-valued (float) weight. You must implement the following methods for the `Term` class:

- A constructor that takes a query and a weight as arguments. The constructor should raise a `TypeError` if query is `None`. It should raise a `ValueError` if the weight is negative.
- A `get_weight()` method that returns the weight.
- The `__eq__` method, which should return `True` if the two `Term`'s queries are equal.
- The `__lt__` method (which represents the < operator), which should return `True` if this `Term`'s query comes before the other `Term`'s query in lexicographic order. (Lexicographic order is based on the ASCII order of characters. That is, `A < Z < a < z`. Lexicographic order is the default ordering for Python strings.)
- The `__str__` method, which should return a string containing this `Term`'s weight and query.

Part II – Autocomplete

In this part, you will implement the functions that provide autocomplete functionality using the `Term` class and a binary search tree. To do so, you will add each term to the binary search tree (BST), use the tree's methods to find the set of terms that start with a given prefix, and sort the results in descending order by weight.

You will organize your program into the following functions. Add these functions to `autocomplete.py`. (They should not be part of the `Term` class.)

- `build_tree(filename)` which takes a filename for the file containing weights and queries. This function should create a `Term` for each query/weight and add them to a `TreeMap` (our BST from class). Return that BST.
- `all_matches(tree, prefix)` which takes a BST and a string prefix. The function should return a list of `Terms` in the tree that start with the given prefix. That list should be sorted by the `Term's` weight in descending order (highest weights first).
    - Python's built-in `sort()` function for lists can be told how to sort a list by using the key keyword arguments. You can also specify descending sort using

the reversed keyword arguments. If you followed the naming conventions above, you should be able to sort a list of `Terms` (`termlist`) by their weight in descending order as follows:

```
termlist.sort(key=Term.get_weight, reversed=True)
```

- `main()` should do the following:
    - Ask the user for a file name and build the tree from that file.
    - Ask the user for a number of results to show.
    - Ask the user for a search term and display the appropriate number of results. The program should continue this step until the user enters a blank search term (by just pressing enter).
    - Time how long each query took to complete. (This should only be based on how long it took for `all_matches()` to execute.)

```
import timeit
start = timeit.default_timer()
# Code that you want to time
end = timeit.default_timer()
total_time_in_ms = end - start
```

Input Format: We provide two sample input files for testing. Each file consists of an integer N followed by N pairs of query strings and positive weights. There is one pair per line, with the weight and string separated by a tab. The query strings are in Unicode and can contain any Unicode characters (including spaces). Because these are Unicode files, you will have to open the file as follows:

```
file = open(filename, 'r', encoding='utf-8')
```

- The file `wiktionary.txt` contains the 10,000 most common words in Project Gutenberg, with weights equal to their frequencies.
- The file `cities.txt` contains nearly 100,000 cities, with weights equal to their populations.

Here are a few sample runs from our programs (user input in blue):

```
Enter file: wiktionary.txt
Enter number of results to show: 5
Enter search: auto
619695.0    automobile
424997.0    automatic
Query took 0 ms
Enter search: comp
13315900.0  company
7803980.0   complete
6038490.0   companion
5205030.0   completely
4481770.0   comply
Query took 0 ms
Enter search: the
```

```
Enter file: cities.txt
Enter number of results to show: 7
Enter search: M
12691836.0  Mumbai, India
12294193.0  Mexico City, Distrito
Federal, Mexico
10444527.0  Manila, Philippines
10381222.0  Moscow, Russia
3730206.0   Melbourne, Victoria,
Australia
3268513.0   Montréal, Quebec, Canada
3255944.0   Madrid, Spain
Query took 150 ms
Enter search: Al M
```

5627187200.0 the 334039800.0 they 282026500.0 their 250991700.0 them 196120000.0 there Query took 0 ms Enter search:	431052.0 Al Maḩallah al Kubrá, Egypt 420195.0 Al Manşūrah, Egypt 290802.0 Al Mubarraz, Saudi Arabia 258132.0 Al Mukallā, Yemen 227150.0 Al Minyā, Egypt 128297.0 Al Manāqil, Sudan 99357.0 Al Maţarīyah, Egypt Query took 0 ms Enter search:

Part III – Autocomplete with a Hash Table

Make a copy of `autocomplete.py` and save it as `autocomplete2.py`. In this copy, make the following changes:

- Change `build_tree()` to `build_map()`. Instead of returning a binary search tree, it should return a `ProbeHashMap` (our linear probing hash table from the `hash_tables.py` module) containing the same data as before. Make sure you update any calls to `build_tree()` to say `build_map()`.
- Re-write `all_matches()` so that it uses the hash table of `Terms` instead of the tree.

Part IV – Data Collection

Select eight search queries to run on both versions of our autocomplete code. Use the `cities.txt` input for these searches. Make a table that shows the query, how long it took in our BST version of our algorithm, and how long it took with the hash table version our algorithm.

Answer the following questions:

1. Overall, which version of our algorithm was faster, the one that used `TreeMap` (the BST) or `ProbeHashMap` (the hash table)?
2. Is this what you expected? Why or why not?
3. Is there any alternative to `TreeMap` or `ProbeChainMap` that you would expect to be superior to both? If so, why?

Extra Credit

Create a real-world data set (preferably large or huge – bigger than cities.txt!) for which autocomplete would be appropriate. Below are some possibilities. Note that some of the datasets are *massive* and you will need to filter them down to appropriate sizes and put them into our input format.

To receive credit, your submission must include:

- An input file in the format specified. (If the file is too large, Kodiak may not accept it. Come to Prof. O'Neill's office to hand off the file on a flash drive.)

- A `readme.txt` file that provides a citation to your data source, and briefly describes anything you did to filter the data.
- A text file showing the results of using your input file on 5 queries using either version of our autocomplete code (ideally whichever version was faster).

Some possibilities for data sets:

- Wikipedia – term = Wikipedia page, weight = number of hits per year
- Google Books Ngram Viewer – term = *n*-gram, weight = frequency of occurrence in corpus of books
- Corpus of Contemporary American English – term = *n*-gram, weight = frequency of occurrence in corpus
- The Internet Movie Database – term = movie, weight = number of reviews or average rating

What to Submit:

- Your code (`autocomplete.py, autocomplete2.py` and any other necessary modules, including modules given as class materials)
- Your table and answers from Part IV
- Any extra credit materials