

CS216: Introduction to Software Engineering Techniques (Spring, 2022)
Programming Assignment 2
(100 points)

Today's Date: Monday, March 21

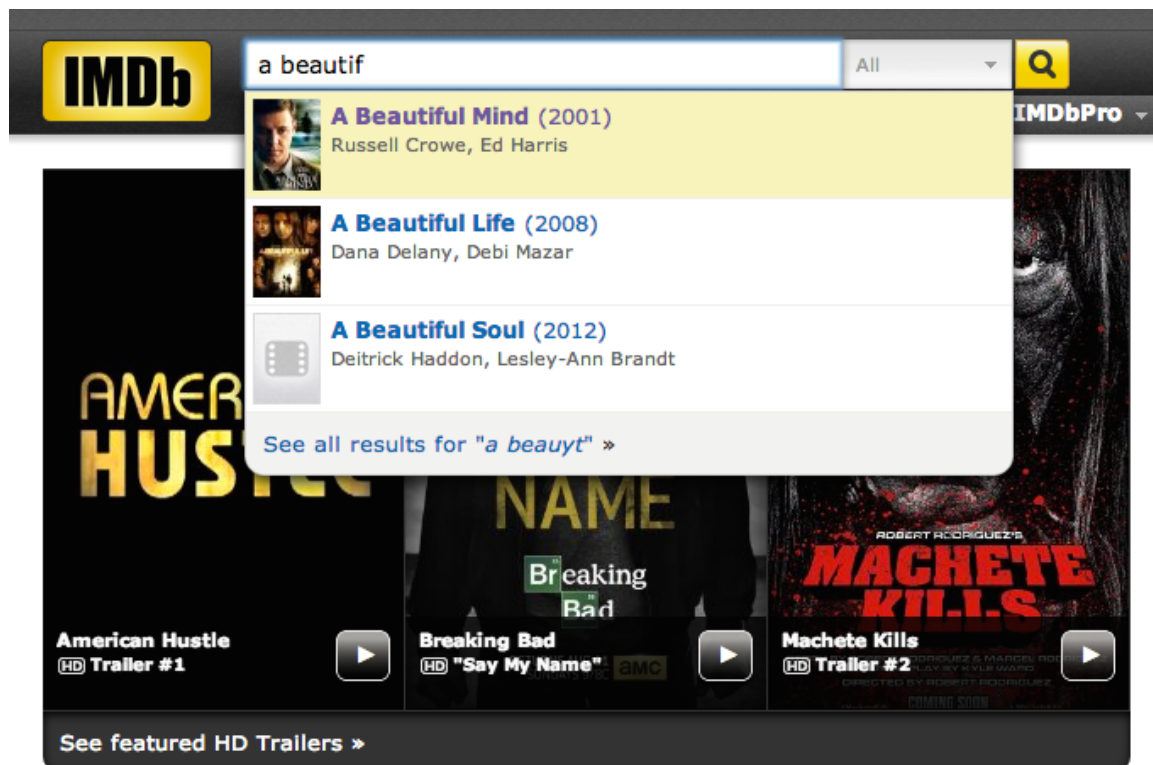
Due Date: Monday, April 4

Problem Statement

Write a C++ program to implement *Autocomplete Me* for a given set of N terms, where a term is a query string and an associated nonnegative weight. That is, given a prefix, find all queries that start with the given prefix, in descending order of weight.

Autocomplete Me is pervasive in modern applications. As the user types, the program predicts the complete *query* (typically a word or phrase) that the user intends to type.

Autocomplete Me is most effective when there are a limited number of likely queries. For example, the [Internet Movie Database](http://www.imdb.com) 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.

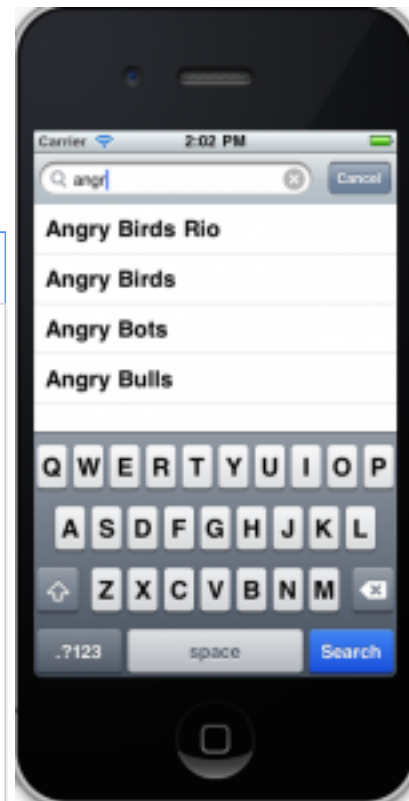




how to

how to tie a tie
how to screenshot on mac
how to get away with a murderer
how to write a check
how to hard boil eggs
how to make money
how to boil eggs
how to screenshot on pc
how to draw
how to write a cover letter

Google Search I'm Feeling Lucky



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 purpose of this assignment, you will have access to a set of all possible queries and associated weights (and these queries and weights will not change).

In this project assignment, you will implement “Autocomplete Me” by sorting the terms by query string first; then use *binary search* to find all matched terms whose query strings starting with a given prefix; and then generate the list of output by *sorting the matched terms, in descending order by weight*.

Part 1: “Autocomplete Me” term. Define a class, named **Term** that represents an autocomplete term: (a query string and an associated integer weight). Then you need to provide three member functions, which support comparing terms by three different sorting criteria respectively: **(1) in lexicographic order by query string; (2) in descending order by weight; and (3) in lexicographic order by query string but using only the first r characters**. The last order may seem a bit odd, but you will use it in **Part 2** to find all query strings that start with a given prefix (of length r). The first sorting criterion is defined through operator overloading for operator “**<**”. Other two sorting criteria are both defined as **static** member functions, and each of them returns an integer value: **-1, 0 or 1**. When comparing two terms, if the first term and the second term are in the right order according to the sorting

criterion, it returns **1**; if they are in the opposite order according to the sorting criterion, it returns **-1**; if they are the same, it returns **0**.

You should start with the following declaration of the class `Term`:

```
class Term
{
    public:
        // default constructor
        Term();

        // initialize with the given query string and weight
        Term(string query, long weight);

        // compare two terms in descending order by weight
        // if t1 and t2 are in descending order by weight, return 1
        // if they are of the same weight, return 0;
        // otherwise, return -1
        static int compareByWeight(Term t1, Term t2);

        // compares two terms in lexicographic order but using only
        // the first r characters of each query
        // if the first r characters of t1 and t2 are in lexicographic order, return 1
        // if they are of the same r characters, return 0;
        // otherwise, return -1
        static int compareByPrefix(Term t1, Term t2, int r);

        // define the operator "<" for Term class (as friend function)
        friend bool operator<(Term t1, Term t2);

        // define the operator "<<" for Term class (as friend function)
        // so that it can send the Term object directly to cout, in the following format:
        // the weight followed by a tab key, then followed by the query
        friend ostream& operator<<(ostream& out, const Term& t);

        // assign "friendship" to the class named Autocomplete
        // so that Autocomplete class can directly access the private data members
        // of Term class. Not the other way around.
        friend class Autocomplete;

    private:
        string query;
        long weight;
};
```

You will work on the complete definition of the above class, named **Term**, during **Lab8**.

Part 2: Autocomplete Me. Define a class, named **Autocomplete** that provides “Autocomplete Me” functionality for a given set of **Term** objects. For this assignment, use the binary search to efficiently find all query strings that start with a given prefix (they are the matched terms), and then sort the matched terms in descending order by weight. To do so, you need to **(1) sort the terms in lexicographic order by query first**, then **(2) use binary search to search for a given prefix, from a sorted sequence**. Since for a given prefix, the matched terms can be more than one, your program needs to get the first and the last such terms in the sequence, you may implement a member function named `search()` as follows:

```
void search(string prefix, int& firstIndex, int& lastIndex);
```

`firstIndex` and `lastIndex` are reference parameters, and they represent the index of the first query that matches the given prefix and the index of the last query that matches the given prefix respectively; or `-1` if no such prefix can be found.

For example, assume the following shows part of the sorted sequence of term objects (note that part of the first column (shown in orange color below) represents the index number of each **Term** object in the sequence named **terms**, which is the private data member of **Autocomplete** class):

```
.....
terms[956] 2809 Wackenhut
terms[957] 219812 Wal-Mart Stores
terms[958] 24623 Walgreen
terms[959] 1693 Wallace Computer Svcs.
terms[960] 25269 Walt Disney
terms[961] 1922 Walter Industries
terms[962] 17692 Washington Mutual
.....
```

Assume that the given **prefix** is “**Wal**”, after calling `binary_search()` inside `search()` function mentioned above, it should return one of the index numbers of whose query matches “**Wal**”. Then from this index number, your program should be able to decide the values for **firstIndex** and **lastIndex**, in this case, **firstIndex** should hold index number 957, and **lastIndex** should hold index number 961 after calling `search()` function. Then, **(3) your program needs to sort the matched terms** (from index **firstIndex** to **lastIndex**) **in descending order by weight**. (note that using each index number, it can directly access the corresponding **Term** object)

You need to use the following declaration of the class **Autocomplete**:

```
class Autocomplete
{
    public:
        // insert the newterm to the sequence
        // note that a SortingList<Term> object, named terms,
        // represents the sequence
```

```

void insert(Term newterm);

// sort all the terms by query in lexicographical order
// note that this function needs to be called before applying binary search
void sort();

// return the index number of the term whose query
// prefix-matches the given prefix, using binary search algorithm
// Note that binary search can only be applied to sorted sequence
// Note that you may want a binary search helper function
int binary_search(string prefix);

// first: the index of the first query whose prefix matches
//      the search key, or -1 if no such key
// last: the index of the last query whose prefix matches
//      the search key, or -1 if no such key
// key: the given prefix to match
// hints: you can call binary_search() first to find one matched index number,
//      say hit, then look up and down from hit, to find first and last respectively
void search(string key, int& first, int& last);

// return all terms whose queries match the given prefix,
// in descending order by weight
SortingList<Term> allMatches(string prefix);

// display all the terms in the sequence
void print() const;

// other member functions you may need...

private:
    SortingList<Term> terms;
};

```

The performance of “*Autocomplete Me*” functionality is critical in many systems. For example, consider a search engine, which runs an *Autocomplete Me* 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! Hence the decision of which data structure should be chosen to store the sequence of term objects in Autocomplete class is very important. Since sorting operation (either in lexicographic order by query and/or in descending order by weight) is needed and essential to the sequence of Term objects in this project, we think about the class named SortingList, which should provide a few member functions using different sorting algorithms, and it should also be

flexible enough to apply to different types of data items, such as integers, Building objects, Term objects and so on. This leads to designing the `SortedList` class to provide the following two features:

- Provide the function pointer to each sorting member function so that the sorting criteria (defined by different function) can be passed in as a parameter;
- Make the `SortedList` class into a template class, so that it can be used to store any type of data members or objects.

You can download the definition of the `SortedList` class from the following link (note that it is almost complete, however it still needs a little help from you) and spend some time understanding how to implement the two major features above.

http://www.cs.uky.edu/~yipike/CS216/PA2_SortedList.zip

You will complete the definitions of two classes, named **Autocomplete** and **SortedList** in Lab9.

Input format. You can download a number of sample input files for testing. Each file consists of N pairs of nonnegative weights and query strings. There is one pair per line, with the weight and query separated by a tab. A weight can be any integer between 0 and $(2^{63} - 1)$. A query string can be an arbitrary sequence of Unicode characters, including spaces (but not newlines).

- Starting with smaller testing file first:
 - The file <http://www.cs.uky.edu/~yipike/CS216/fortune.txt> contains Fortune 1000 American largest companies, ranked by revenues. The weight equals to the company's revenue.
- When you think it is ready to challenge the sorting algorithm you choose for your project, you can test running your program with the following file:
 - The file <http://www.cs.uky.edu/~yipike/CS216/actors.txt> contains 100,000 famous American actors and actresses, with weights equal to their revenue.
 - The file <http://www.cs.uky.edu/~yipike/CS216/imdb.txt> contains over 80,000 movies from IMDB web site, with weights equal to the number of votes.

When you run your program, it should take the name of an input file and an integer k as command-line arguments. It reads the data from the file; then it repeatedly asks the user to enter autocomplete queries, and prints out the top k matching terms in descending order by weight, until the user enters "exit" to quit. If there are only j matching terms, where $(0 < j < k)$, then it prints out the top j matching terms in descending order by weight.

Notes on implementation:

- You are required to use the declaration of **Term** class described in this document, and binary search algorithm to find all query strings that start with a given prefix.
- You are also required to use the declaration of **Autocomplete** class described in this document, however, you are allowed to add other member functions if you want to. Do not modify the declaration of **SortingList** class you download from the instructor's web site. If you use other data structure to store the Term objects, you will get 10 points deduction. While making your decisions about data structure and algorithms, it should be based on the application: the size of the data items and the mix of operations you will perform on those data items.
- Your program should not prompt for the user input at the beginning, and it should just start reading the data from the input file before asking the user for input.
- Since there are a few sorting algorithms implemented as member functions in the **SortingList** class, it is your choice to decide which function to call to fit the sorting operations you apply to the sequence of data items. Which one is the most efficient one (to sort *a large number* of Term objects fast enough)? Which one allows you to pass in sorting criteria (defined by your own function) as a parameter so that I can sort the matched Term objects *in decreasing order by weight*?

Part 3 makefile. Create a make file (name it makefile) to build your C++ program from **Part 1** and **Part 2**. In the makefile, name the executable program **Project2** (NO .exe).

For all files:

- Test on your Virtual Machine. If (for whatever reason) it doesn't work there, you lose points. If your program cannot pass compilation, then you get zero credit.
- Your program is unzipped then compiled/linked by executing the command:

\$ make

therefore:

- All needed files are supplied and assumed to be on the current directory
- No hardcoded directory names in the .h file names
- Create your makefile (file name makefile)
- In the makefile, name the created program Project2
- Each class has a separate specification file (classname.h) and implementation file (classname.cpp) unless two classes have very closed relationship
- Comments for every file you write, lay out your source file in a readable style
- Name your zip file **Project2.zip** and zip using the following command:

```
$ zip Project2.zip makefile *.cpp *.h *.txt  
                                (or list the individual files)
```

NOT

```
$zip Project2.zip /PA2/*
```

Zip from the directory where the files are, not a parent directory.

After passing the compilation, please download the following sample output file to test running your program, and also think about how to design your own testing cases:

http://www.cs.uky.edu/~yipike/CS216/PA2Sample_auto.pdf

After testing your program with sample testing cases and testing cases you design by yourself, please follow the instructions in Part 3 to prepare your submission.

Extra Part: (Bonus 16 points)

In order to gain Bonus points for Project 2, you have to make sure that your program has already generated the correct output according to the Problem Statement in this document. Just as your own experience with *autocomplete me* feature, such as Google search engine or imdb movie searching:

- The prefix matching should be **case-insensitive**. For example, with the input file of actors' information, if the user types "toM", or "tOm", or "TOM", your program should be able to match the actors whose first name is "Tom". (10 points)
- If the user-input starts with quite a few blank spaces or tab key, your program should ignore them and start to match the prefix from the first non-space character. (3 points)
- If the user-input contains spaces or tab key in between two non-space characters, your program should consider it as a single space. For example, if the user types " tom H", your program should use "tom H" as the prefix to match. (3 points)

For the Extra Part of this project, based on what you have done for the Project 2, challenge your program with the above three features, correctly implement each feature helps you gain at maximum of $(10+3+3 = 16)$ bonus points.

After passing the compilation, please download the following sample output file to test running your program with extra features from Bonus Part:

http://www.cs.uky.edu/~yipike/CS216/PA2Sample_Extra.pdf

Submission:

Open the link to course Canvas page (<https://www.uky.edu/canvas/>), and login to your account using your linkblue user id and password. Please submit your file (Project2.zip) through the submission link for "Project 2". Note that only one file is allowed to upload and it should be your zip file. It is a good idea to check that your file is already uploaded successfully. If not, go back and submit again.

(Late assignment will be reduced 10% for each day that is late. The assignment will not be graded (you will receive zero) if it is more than 5 days late. Note that a weekend counts just as regular days. For example, if an assignment is due Friday and is turned in Monday, it is 3 days late.)

Academic Honesty:

All assignments in class are individual work. All work submitted as part of the class must be your own. You may not share work (whether from quizzes, lab assignments, or project assignments), nor may you use code provided to you by others, except for your instructor. You are allowed to use the source code provided by the instructor of this course only.

Always read the grading sheet for each project assignment. It lists typical errors. Check for these errors before submitting your source code. **Please note that your C++ program must compile in order to be graded.** If your program cannot pass the compilation, you will get 0 point.

Grading Sheet for Project Assignment 2

Total: 100 points (Bonus 16 points)

These are example errors. There are other ways to lose points. C++ programs must compile in order to be graded	Points	Deducted Points
C++ Program	80	
Provide the command line argument check while running your program	5	
Check whether the file can be open successful	5	
Correctly read pairs of (weight, query) from the input file and store into an Autocomplete object	5	
Correctly and efficiently sort all Term objects in lexicographical order by query (if your program cannot pass the test from imdb.txt as the input file, you lose 7 points for this category)	10	
Repeatedly allow the user to type the search query until the user enters "exit" to quit	2	
Generate N terms matched with the search query by prefix, where N is the smaller value between the second command line argument and the total number of matcher terms)	5	
Correctly display the matched terms in the decreasing order by weight	8	
Provide the function which uses binary search to find the matched term	6	
Correctly provide the complete definition of Term class	5	
Correctly provide the complete definition of Autocomplete class	16	
Complete the definition of SortingList class as a template class	8	
Provide separate .cpp files and header files for class(es).	3	
Quit the program correctly	2	

<p>makefile</p> <p>Generate the executable file correctly</p> <p>The dependency lines correctly track of files' dependencies</p> <p>The command lines correctly create targets</p> <p>Allow "make clean" to clean up the mess</p>	<p>2</p> <p>2</p> <p>2</p> <p>1</p>	7	
<p>Miscellaneous errors, or did not follow the directions in the program assignment, examples:</p> <p>(-10 if your program uses other data structure than SortingList class to store the sequence of Term objects in the Autocomplete class)</p> <p>-2 did not zip file or used tar or gzip instead</p> <p>-2 created subdirectory when unzipped</p> <p>-1 wrong names</p> <p>There may be other errors in the is category</p>	<p>2</p> <p>2</p> <p>1</p>	5	
<p>Style</p> <p>Lay out your program in a readable style</p> <p>C++ program comments</p> <p>-6 non</p> <p>-4 only a few</p>	<p>2</p> <p>6</p>	8	
<p>Bonus Extra Part</p> <p>Your program also provides the following feature:</p> <p>The prefix matching should be case-insensitive</p> <p>The prefix matching should start from the first non-space character(ignore space or tab at the beginning)</p> <p>The prefix matching should only keep one space between the non-space characters from user-input</p>	<p>10</p> <p>3</p> <p>3</p>	16	
Your Score for Project 2			