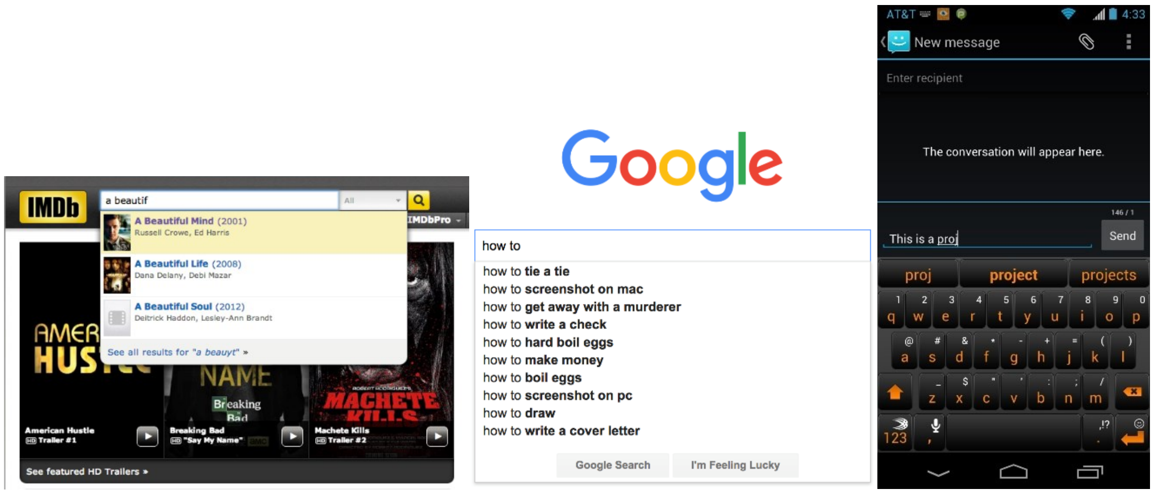
CSE247 Data Structures
Fall'24

Homework # 2 ***Due: Oct 6, 2024***

Assignment: Autocomplete

Overview: Write a program to implement autocomplete for a given set of `n` terms, where a term is a query string and an associated non-negative weight. Given a prefix, the program should find all queries that start with the given prefix and return them in descending order of weight.

Autocomplete is widely used in modern applications like search engines, text input on mobile phones, and websites that predict user queries. Your task is to implement this functionality efficiently for a fixed set of query strings and their associated weights.



a beautif

A Beautiful Mind (2001) Russell Crowe, Ed Harris

A Beautiful Life (2008) Dana Delany, Debi Mazar

A Beautiful Soul (2012) Deitrick Haddon, Lesley-Ann Brandt

See all results for "a beauyt" »

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

New message

Enter recipient

The conversation will appear here.

This is a proj

Send

proj | project | projects

The performance of autocomplete is critical in many systems, requiring quick responses even with large datasets. In this assignment, you'll implement autocomplete using binary search and sorting.

Part 1: Autocomplete Term

Create a C++ class `Term` to represent an autocomplete term, which includes a query string and an integer weight. This class should support comparison of terms by different criteria: lexicographic order, descending order by weight, and lexicographic order using only the first `r` characters.

API for Term class:

```
class Term {
public:
    std::string query;
    long weight;

    // Constructor: Initializes a term with the given query string and weight.
    Term(std::string query, long weight);

    // Compare two terms in lexicographic order by query (natural order).
    bool operator<(const Term& other) const;

    // Compares the two terms in descending order by weight.
    // Return a comparator that compares two terms in descending order by weight.
    static std::function<bool(const Term&, const Term&)> byReverseWeightOrder();

    // Returns a comparator that compares two Term objects by the first r
characters
    // of their query strings in lexicographic order (i.e., dictionary order).
    static std::function<bool(const Term&, const Term&)> byPrefixOrder(int r);

    // Returns a string representation of the term: "weight query".
    std::string toString() const;

    // Unit testing (required)
    static void test();
};
```

Requirements:

1. Throw an `invalid_argument` exception in the constructor if the query is empty or weight is negative.
2. Implement the `byReverseWeightOrder` function to compare two terms by descending weight.
3. Implement the `byPrefixOrder` function to compare two terms lexicographically, using only the first `r` characters of the query.
4. Overload the < operator to compare two terms lexicographically by their query strings.
5. The `toString` method should return a string representation of the term in the format: weight followed by a tab and the query.
6. Include a `unitTest` method for testing.

Performance: String comparison functions should take time proportional to the number of characters required to resolve the comparison.

Example:

Consider the following `Term` objects:

```
Term t1("apple", 500);
Term t2("application", 300);
Term t3("app", 700);
```

1. Lexicographic order:
    1. `t3` < `t1` < `t2` ("app", "apple", "application")
2. Reverse weight order:
    1. `t3`, `t1`, `t2` (700, 500, 300)
3. Prefix order (with `r = 3`):
    1. `t1` and `t3` are equal (`"app"` prefix).
    2. `t2` starts with `"app"`, but using only the first 3 characters, `"application"` is compared as `"app"`.

The `toString()` method for `t1` will output:

```
500 apple
```

Part 2: Binary Search

Create a `BinarySearchDeluxe` class that supports binary search for the first and last occurrence of a key in a sorted array.

API for BinarySearchDeluxe class:

```
class BinarySearchDeluxe {
public:
    // Returns the index of the first key in the sorted array that is equal to the search key.
    template <typename Key>
    static int firstIndexOf(const std::vector<Key>& a, const Key& key, std::function<bool(const Key&, const Key&)> comparator);

    // Returns the index of the last key in the sorted array that is equal to the search key.
    template <typename Key>
    static int lastIndexOf(const std::vector<Key>& a, const Key& key, std::function<bool(const Key&, const Key&)> comparator);

    // Unit testing (required)
    static void test();
};
```

Requirements:

1. Throw an `invalid_argument` exception if any argument to `firstIndexOf` or `lastIndexOf` is null.
2. Ensure the array is sorted with respect to the provided comparator.
3. Both methods must make at most `1 + ceil(log2(n))` comparisons in the worst case, where `n` is the length of the array.

Example:

Given the array of `Term` objects (sorted lexicographically):

```
std::vector<Term> terms = {Term("app", 700), Term("apple", 500),
Term("application", 300)};
```

1. `firstIndexOf(terms, t1, Term::compareByPrefixOrder(3))` should return `0`, as "app" is the first match. Assume `t1` is Term t1("app", 1000).
2. `lastIndexOf(terms, t1, Term::compareByPrefixOrder(3))` should return `2` as "app" occurs in "application" as well.

Given a more complex array:

```
std::vector<Term> terms = {Term("apple", 500), Term("app", 700), Term("app",
600), Term("application", 300)};
```

1. `firstIndexOf(terms, "app", Term::compareByPrefixOrder(3))` should return `0` (the first "app").
2. `lastIndexOf(terms, "app", Term::compareByPrefixOrder(3))` should return `3` (the last "app").

Part 3: Autocomplete

Using `Term` and `BinarySearchDeluxe`, implement a data type that provides autocomplete functionality.

API for Autocomplete class:

```
class Autocomplete {
private:
    std::vector<Term> terms;
public:
    // Initializes the data structure from the given array of terms.
    Autocomplete(const std::vector<Term>& terms);

    // Returns all terms that start with the given prefix, in descending order of
weight.
    std::vector<Term> allMatches(const std::string& prefix);

    // Returns the number of terms that start with the given prefix.
    int numberOfMatches(const std::string& prefix);

    // Unit testing (required)
    static void test();
};
```

Requirements:

1. Throw an `invalid_argument` exception in the constructor if the argument array or any entry is null.
2. In `allMatches`, find all terms that match the given prefix and return them in descending order by weight.
3. In `numberOfMatches`, return the number of terms that match the given prefix.
4. Ensure performance requirements:
    1. `Autocomplete` constructor should make `O(n log n)` comparisons.
    2. `allMatches` should make `O(m log m + log n)` comparisons, where `m` is the number of matches.
    3. `numberOfMatches` should make `O(log n)` comparisons.

Example:

Given the following list of terms:

```
std::vector<Term> terms = {
    Term("apple", 500),
    Term("app", 700),
    Term("application", 300),
    Term("ape", 200),
    Term("apex", 100)
};
Autocomplete autocomplete(terms);

// Example 1: All terms that start with "app"
std::vector<Term> results = autocomplete.allMatches("app");
for (const auto& term : results) {
    std::cout << term.toString() << std::endl;
}

// Output:
// 700  app
// 500  apple
// 300  application

// Example 2: Number of terms that start with "ap"
std::cout << autocomplete.numberOfMatches("ap") << std::endl;

// Output:
// 5
```

Input Format:

You will be given a text file that contains an integer `n` followed by `n` pairs of query strings and their corresponding non-negative weights. The file format will have one pair per line, with the weight and string separated by a tab. A query string can be any sequence of Unicode characters, including spaces (but not newlines).

Example Main Function:

```
int main(int argc, char* argv[]) {
    std::string filename = argv[0];
    std::ifstream infile(filename);

    int n;
    infile >> n;

    std::vector<Term> terms;
    for (int i = 0; i < n; i++) {
        long weight;
        std::string query;
        infile >> weight >> std::ws;
        std::getline(infile, query);
        terms.emplace_back(query, weight);
    }

    Autocomplete autocomplete(terms);
    std::string prefix;
    while (std::getline(std::cin, prefix)) {
        std::vector<Term> results = autocomplete.allMatches(prefix);
        std::cout << results.size() << " matches\n";
        for (size_t i = 0; i < std::min(results.size(), static_cast<size_t>(5));
i++) {
            std::cout << results[i].toString() << "\n";
        }
    }
}
```

Sample Executions:

Example 1:

```
~/Desktop/autocomplete> ./autocomplete wiktionary.txt 5
auto
2 matches
        619695   automobile
        424997   automatic
comp
52 matches
      13315900   company
       7803980   complete
       6038490   companion
       5205030   completely
       4481770   comply
the
38 matches
    5627187200   the
     334039800   they
     282026500   their
     250991700   them
     196120000   there
```

Example 2:

```
~/Desktop/autocomplete> ./autocomplete cities.txt 7
M
7211 matches
    12691836  Mumbai, India
    12294193  Mexico City, Distrito Federal, Mexico
    10444527  Manila, Philippines
    10381222  Moscow, Russia
     3730206  Melbourne, Victoria, Australia
     3268513  Montréal, Quebec, Canada
     3255944  Madrid, Spain
Al M
39 matches
      431052  Al Maḩallah al Kubrá, Egypt
      420195  Al Manşūrah, Egypt
      290802  Al Mubarraz, Saudi Arabia
      258132  Al Mukallā, Yemen
      227150  Al Minyā, Egypt
      128297  Al Manāqil, Sudan
       99357  Al Maţarīyah, Egypt
```

These examples search through large datasets like city names and terms from Wiktionary. They demonstrate the program's ability to return query results sorted by weight and limited to a certain number of matches (in this case, 5 and 7).

Grading Breakdown:

File	**Points**
Term.cpp	13
BinarySearchDeluxe.cpp	10
Autocomplete.cpp	12
readme.txt	5
Total	40

The provided `unitTest` methods should perform appropriate testing on all public methods.