Data Structures and Algorithms II
Fall 2024
Programming Assignment #1

For this assignment, first, you are going to create a hash table class. Then, you are going to write a program that uses your hash table class to load a "dictionary" and spell check a "document".

For the purposes of this assignment, a valid word is defined as any sequence of valid characters, and the valid characters are letters (capital A – Z and lowercase a - z), digits (0 - 9), dashes (-), and apostrophes ('). Every other character is considered a word separator.

A dictionary is defined as a list of recognized words. The dictionary is guaranteed to contain exactly one word per line, with no leading or trailing spaces, followed by a single, Unix-style newline character (\n). Some of the words in the dictionary might not be valid (i.e., they may contain invalid characters). When loading the dictionary, invalid words, as well as words that are too long (see below), can optionally be ignored. The dictionary does not specify the meanings of words; it just lists them.

The document to spell check may be any valid text file. Each line in the document will end with a single, Unix-style newline character. When spell checking the document, your program should indicate every unrecognized word, including the line number on which it occurs. Words should only be allowed to grow up to 20 characters. If a word in the document is too long, you should indicate the line number on which this occurs along with the first 20 characters of the word. The first line in the document is line 1. Words in the document that include digits (perhaps in addition to other valid characters) are technically valid but should not be spell checked (i.e., your program should ignore them). In the document, as previously stated, every character that is not a valid word character is a word separator; e.g., the string "abc@def" represents two valid words, "abc" and "def". Therefore, there cannot be invalid words in the document.

Your program should be case insensitive, and all capital letters in both the dictionary and the document should be converted to lowercase immediately upon seeing them.

Your program must be written in C++. In order to implement this task efficiently, you will use a hash table. You must implement a hash table class using separate files, including a header file and a source code file. Not every member function of the class will be necessary for this assignment, but you will reuse this class for your next two assignments. Since our textbook provides code for the separate chaining and quadratic probing collision resolution strategies, I am requiring that you use either linear probing or double hashing. (Linear probing is a bit simpler to implement, and you will not receive any extra credit if you choose double hashing.) You are welcome to look at the book's code for the other two strategies, but keep in mind that the instructions I am specifying for your hash table class make this different than the book's implementation in several ways. For example, the book uses templates for its hash table class, but you will not. Also, your hash table class will allow the programmer to associate additional

data with each entry, while the book's implementation does not. More details about the requirements for your hash table class will be discussed later in this handout and in class. I will also discuss the use of void pointers in class.

To process the dictionary, simply insert every word in the dictionary into the hash table. To spell check the document, locate every valid word in the document (keeping track of line numbers), and lookup (i.e., search for) each word in the hash table to see if it is recognized. You should assume that an average dictionary contains about 50,000 words, but that some might be as large as 1,000,000 words. This is my way of telling you that you should implement a rehash member function! A sample dictionary, a bit on the small side (approximately 25,000 words), will be posted on the course home page.

Your program should prompt the user for the name of the dictionary file, the name of the document file to be spell-checked, and the name of the file where output should be written. Your program should indicate how long, in seconds, it takes to read the dictionary and how long it takes to spell check the document, measured in terms of CPU time. (These times should be displayed to standard output, not to the output file.) Your program must compile and run correctly using the g++ compiler on either Cygwin or Ubuntu.

Your hash table implementation must include a header file called "hash.h" and a source code file called "hash.cpp". The spell-checking code, and the rest of the main program, should be included in a separate file. You should also provide a makefile named "Makefile" that I can use to compile your program. I will provide a sample makefile that I used for my version of the program (also shown on the final page of this handout). I will also provide my version of "hash.h" (also shown later in this handout). You may reuse these two files directly if you wish. These files will also be posted on the course home page and discussed in class.

A sample document, a sample run of the program using that document, and a sample output file appear on pages 3 and 4 of this handout. The document text file used for this sample run, as well as the dictionary used, will be available from the course home page. Your output file should adhere *exactly* to the format shown, and all output messages should be worded exactly the same way, with the same spacing. I will use "diff" to compare your output to mine, and you will lose points for any differences. (Of course, I will test your programs on multiple test cases involving different documents and dictionaries of various sizes.) The output displayed to standard output does not have to match my formatting, as long as the content is the same.

Pages 5 and 6 of this handout show my "hash.h" file. Your hash table class must implement the same public member functions as mine. Note that the "getPointer", "setPointer", and "remove" member functions will not be used for this assignment; however, they will be used for future assignments! It is OK if you do not implement these member functions now. The specifics of this header file, and also the "Makefile" (shown on page 7 of this handout), will be discussed in more detail in class. Both files will also be made available to you from the course home page.

When your assignment is complete, e-mail me (*carl.sable@cooper.edu*) your program, including your source code files, your header file(s), and your makefile (even if you use the provided files). Do NOT send me your executable or object files (.o files). In addition to correctness, your grade may also depend on the efficiency and elegance of your code and adherence to proper C++ style. Your program is due before midnight on the night of Tuesday, September 24.

Below are the lyrics to "Supercalifragilisticexpialidocious" from "Mary Poppins". This represents the contents of the document "lyrics.txt" used in the sample run shown on the next page. This file will also be posted on the course home page.

```
Um-deedledeedledeedle um-deedleday
Um-deedledeedledeedle um-deedleday
Um-deedledeedledeedle um-deedledeedle
Um-deedledeedledeedle um-um um-um um-um

For example...

Supercalifragilisticexpialidocious
Even though the sound of it is something quite atrocious
If you say it loud enough you'll always sound precocious
Supercalifragilisticexpialidocious

Um-deedledeedledeedle um-deedleday
Um-deedledeedledeedle um-deedleday
Um-deedledeedledeedle um-deedleday

Super-super
Supercali
Super Supercalifragi

So when the cat has got your tongue there's no need for dismay
Just summon up this word and then you've got a lot to say
But better use it carefully or it can change your life

For example...

Yes?

One day I said it to me girl and now me girl's me wife

Supercalifragilisticexpialidocious
Even though the sound of it is something quite atrocious
If you say it loud enough you'll always sound precocious
Supercalifragilisticexpialidocious

Supercalifragilisticexpialidocious
Even though the sound of it is something quite atrocious
If you say it loud enough you'll always sound precocious
Supercalifragilisticexpialidocious

Supercalifragilisticexpialidocious
```

```
Even though the sound of it is something quite atrocious
If you say it loud enough you'll always sound precocious
Supercalifragilistic
Supercalifragilistic
Supercalifragilisticexpialidocious
```

Below is a sample run using the sample dictionary provided on the course home page and a text file that contains the lyrics to "Supercalifragilisticexpialidocious" from "Mary Poppins".

```
Enter name of dictionary: DICT/wordlist_small
Total time (in seconds) to load dictionary: 0.031
Enter name of input file: FILES/lyrics.txt
Enter name of output file: out_lyrics_small.txt
Total time (in seconds) to check document: 0
```

The output file should look exactly like this:

```
Long word at line 1, starts: um-deedledeedledeedl
Unknown word at line 1: um-deedleday
Long word at line 2, starts: um-deedledeedledeedl
Unknown word at line 2: um-deedleday
Long word at line 3, starts: um-deedledeedledeedl
Unknown word at line 3: um-deedledeedle
Long word at line 4, starts: um-deedledeedledeedl
Unknown word at line 4: um-um
Unknown word at line 4: um-um
Unknown word at line 4: um-um
Long word at line 8, starts: supercalifragilistic
Long word at line 11, starts: supercalifragilistic
Long word at line 13, starts: um-deedledeedledeedl
Unknown word at line 13: um-deedleday
Long word at line 14, starts: um-deedledeedledeedl
Unknown word at line 14: um-deedleday
Long word at line 15, starts: um-deedledeedledeedl
Unknown word at line 15: um-deedleday
Unknown word at line 17: super-super
Unknown word at line 18: supercali
Unknown word at line 19: supercalifragi
Unknown word at line 21: has
Unknown word at line 21: there's
Unknown word at line 21: dismay
Unknown word at line 23: better
Unknown word at line 23: carefully
Unknown word at line 27: yes
Unknown word at line 29: girl's
Long word at line 31, starts: supercalifragilistic
```

```
Long word at line 34, starts: supercalifragilistic
Long word at line 36, starts: supercalifragilistic
Long word at line 39, starts: supercalifragilistic
Long word at line 41, starts: supercalifragilistic
Unknown word at line 44: supercalifragilistic
Unknown word at line 45: supercalifragilistic
Long word at line 46, starts: supercalifragilistic
```

Below and on the next page, I am providing you with the header file ("hash.h") for my hash table implementation. This file will also be posted on the course home page.

```
#ifndef _HASH_H
#define _HASH_H

#include <vector>
#include <string>

class hashTable {

 public:

  // The constructor initializes the hash table.
  // Uses getPrime to choose a prime number at least as large as
  // the specified size for the initial size of the hash table.
  hashTable(int size = 0);

  // Insert the specified key into the hash table.
  // If an optional pointer is provided,
  // associate that pointer with the key.
  // Returns 0 on success,
  // 1 if key already exists in hash table,
  // 2 if rehash fails.
  int insert(const std::string &key, void *pv = nullptr);

  // Check if the specified key is in the hash table.
  // If so, return true; otherwise, return false.
  bool contains(const std::string &key);

  // Get the pointer associated with the specified key.
  // If the key does not exist in the hash table, return nullptr.
  // If an optional pointer to a bool is provided,
  // set the bool to true if the key is in the hash table,
  // and set the bool to false otherwise.
  void *getPointer(const std::string &key, bool *b = nullptr);

  // Set the pointer associated with the specified key.
  // Returns 0 on success,
  // 1 if the key does not exist in the hash table.
  int setPointer(const std::string &key, void *pv);

  // Delete the item with the specified key.
```

```
// Returns true on success,
// false if the specified key is not in the hash table.
bool remove(const std::string &key);
```

```
 private:

  // Each item in the hash table contains:
  // key - a string used as a key.
  // isOccupied - if false, this entry is empty,
  //              and the other fields are meaningless.
  // isDeleted - if true, this item has been lazily deleted.
  // pv - a pointer related to the key;
  //      nullptr if no pointer was provided to insert.
  class hashItem {
  public:
    std::string key {""};
    bool isOccupied {false};
    bool isDeleted {false};
    void *pv {nullptr};

    hashItem() = default;
  };

  int capacity; // The current capacity of the hash table.
  int filled; // Number of occupied items in the table.

  std::vector<hashItem> data; // The actual entries are here.

  // The hash function.
  int hash(const std::string &key);

  // Search for an item with the specified key.
  // Return the position if found, -1 otherwise.
  int findPos(const std::string &key);

  // The rehash function; makes the hash table bigger.
  // Returns true on success, false if memory allocation fails.
  bool rehash();

  // Return a prime number at least as large as size.
  // Uses a precomputed sequence of selected prime numbers.
  static unsigned int getPrime(int size);
};

#endif //_HASH_H
```

This page shows the simple makefile, named "Makefile", that I used for my program. This will also be posted on the course home page. I will also explain the use of makefiles in class.

```
spell.exe: spellcheck.o hash.o
     g++ -o spell.exe spellcheck.o hash.o

spellcheck.o: spellcheck.cpp hash.h
     g++ -c spellcheck.cpp

hash.o: hash.cpp hash.h
     g++ -c hash.cpp

debug:
     g++ -g -o spellDebug.exe spellcheck.cpp hash.cpp

clean:
     rm -f *.exe *.o *.stackdump *~

backup:
     test -d backups || mkdir backups
     cp *.cpp backups
     cp *.h backups
     cp Makefile backups
```