COMP26120
Academic Session: 2018-19

Lab Exercise 8: Spellchecking using Trees and Hash-Tables

Duration: 2 lab sessions. **The deadline is at the start of Semester 2.**

For this lab exercise you should do all your work in your COMP26120/ex8 directory.

Copy the starting files from */opt/info/courses/COMP26120/problems/ex8*.

For this assignment, you can only get full credit if you do not use code from outside sources.

Learning Objectives

By the end of this lab you will be able to:

- Write C code to use ordered-binary-trees to store and search for strings.
- Implement a self-balancing tree data structure and explain how self-balancing works
- Write C code to use hash-tables to store and search for strings.
- Implement various collision resolution mechanisms and explain how they work
- Explain the asymptotic complexity of searching in either data structure

Introduction

The aim of this exercise is to write C code to complete two different versions of a simple spell-checking program: one using ordered-binary-trees and the other using hash-tables.

The completed programs will consist of several ".c" and ".h" files, combined together using *make*. You are given a number of complete and incomplete components to start with:

- *speller.h* - defines the facilities provided by *speller.c*
- *speller.c* - the driver for each spell-checking program, which:

1. reads strings from a dictionary file and *inserts* them in your data-structure (ordered-binary-tree or hash-table)
2. reads strings from a second text file and *finds* them in your data-structure
   - if a string is not found then it is assumed to be a spelling mistake and is reported to the user, together with the line number on which it occurred

- *dict.h* - defines the dictionary facilities that must be provided by *dict-tree.c* or *dict-hash.c* for the spell-checking program to function correctly.
- *dict-tree.c* - the starting point for your ordered-binary-tree version of the dictionary
- *dict-hash.c* - the starting point for your hash-table version of the dictionary

You are also given a makefile and a series of data-files to use for testing your code.

You are expected to complete *dict-tree.c* for part 1 and *dict-hash.c* for part 2. You should not change any of the other .c or .h files.

> **Note:** The code in *speller.c* that reads words treats any non-alphabetic character as a separator, so e.g. "non-alphabetic" would be read as the two words "non" and "alphabetic". This is intended to extract words from any text for checking (is that "-" a hyphen or a subtraction?) so we must also do it for the dictionary to be consistent. This means that your code has to be able to deal with duplicates (when building the tree or table) i.e. recognise and ignore them. For example, on my PC */usr/share/dict/words* is intended to contain 479829 words (1 per line) but is read as 526065 words of which 418666 are unique and 107399 are duplicates.

Description

The interface - Table

The simple spelling checker you are given uses a "Table" data-type defined by *dict.h*, in which to store all the words in a dictionary. You will implement the Table data-type in different ways for tree and for hash-table.

dict.h requires the following functions to be implemented:

- *initialize_table* The hash-table version uses a table-size parameter, which is ignored for the ordered-binary-tree version. You have to *malloc* space for the data structure(s), and set fields to, e.g., zero. For the hash-table this includes setting up an array of empty cells, whereas for the ordered-binary-tree you only need to initialise the pointer to the head (root) of the tree (e.g. to NULL).
  You don't need to write code to e.g. input the dictionary - this is already done for you in the main function in *speller.c*
- *find* a given key (i.e. string, alphabetic word) in a Table.

- *insert* a given key in a Table.

  If *insert* is called with a duplicate key (i.e. it is already in your Table) you should ignore it, so that every key in your Table is different.

  If all the keys seem to be the same, you are probably forgetting to make a copy of the array holding the key (e.g. using *strdup*) before you save it in your Table.

- *print_table* to list the contents of a Table. (You can also use this to help with debugging.)

- *print_stats* to output statistical information to show how well your code is working. For your hash table you need to count the number of collisions so you can calculate the average per access;
  For your tree you need to calculate the height and/or the average number of string compares per call to *insert* or *find*, so you can compare them with the theoretical minimum. You should add fields to the data structures in the code you are given to collect the information you need.

(You do *not* need to implement a function to delete a given key from a Table as this is not needed by the spell-checker.)

Running your code

To test your implementation, you will need a sample dictionary and a sample text file with some text that contains the words from the sample dictionary (and perhaps also some spelling mistakes). You are given several such files, and you will probably need to create some more to help debug your code.

You should also test your program using a larger dictionary. One example is the Linux dictionary that can be found in */usr/share/dict/words*.

Compile and link your code using *make tree* (for part 1) or *make hash* (for part 2).

When you run your spell-checker program, you can:

- use the $-d$ flag to specify a dictionary.
- (for part 2) use the $-s$ flag to specify a hash table size: try a prime number greater than twice the size of the dictionary (e.g. 1,000,003 for */usr/share/dict/words*).
- use the $-m$ flag to specify a particular mode of operation for your code (e.g. to use a particular hashing algorithm). You can access the mode setting by using the corresponding mode variable in the code you write.
- use the $-v$ flag to turn on diagnostic printing, or $-vv$ for more printing (or $-vvv$ etc. - the more "v"s, the higher the value of the corresponding variable *verbose*).

e.g.:

```
tree -d sample-dictionary -m 1 -vv sample-file
```

or:

hash -d /usr/share/dict/words -s 1000003 -m 2 -v sample-file

There are several such tests already provided in the *makefile*, and you should read it to see what they are, and also how to modify them by changing some of the variables in the *makefile* such as *MODE*. Then you can "make testtree" or "make test1" etc.

Part 1: Tree-based implementation

Implement the Table type using ordered-binary-trees (see, for example: M.T. Goodrich, R. Tamassia: Algorithm Design p. 145). Your code should be go in: *dict-tree.c*

You can start by using a simple insert technique to build the tree that constitutes the Table, although in this example it will lead to extremely sub-optimal tree shape (Why is this?).

because the tree may become very unbalanced even though it is ordered

If you have time, you should then improve your algorithm to produce more balanced trees using e.g. AA trees or AVL trees. (See, for example: M.T. Goodrich, R. Tamassia: Algorithm Design p. 152). *There are marks available for this.*

Write your code so that you can use the -m parameter, which sets the mode variable, to select the different algorithms (e.g. unbalanced or balanced) that you have implemented.

During the marking of this part you will be asked to explain why you chose the particular algorithm you used, and to discuss the asymptotic algorithm complexity of the function *find* that you implemented. You need to discuss the best and the worst case. You also need to compare this complexity to that when the dictionary is implemented as a list (see the lecture notes on complexity).

Part 2: Hash table-based implementation

Reimplement the Table data type using hash tables. Your code should go in: *dict-hash.c*

The hash-value(s) needed for inserting a string into your hash-table should be derived from the string. For example, you can consider a simple summation key based on ASCII values of the individual characters, or a more sophisticated polynomial hash code, in which different letter positions are associated with different weights. **(Warning: if your algorithm is too simple, you may find that your program is very slow when using a realistic dictionary.)**

The hashing strategy to be adopted is that of open addressing, so that collisions are dealt with by using a collision resolution function. You should attempt to implement several different instances of linear probing for collision handling using different values of the hash table size N (where N is a prime number). Then try to improve the collision handling procedure by implementing quadratic probing.

If you have time, you should try to implement double hashing. *There are marks available for this.*

Write your code so that you can use the $-m$ parameter, which sets the *mode* variable, to select the different hashing algorithms that you have implemented.

Your code should keep the *num_entries* field of the table up to date. Your *insert* function should check for a full table, and exit the program cleanly should this occur.

If you have time, increase (double?) the hash table size when the table is getting full and then rehash into a larger table. *There are marks available for this.*

You should experiment with the various hash functions described in the lectures, with different Table sizes and different collision resolution functions. You will need to add code to print_stats (and other places) to report your findings.

During the marking of this part you will be asked to discuss the asymptotic algorithmic complexity of your function *find*, and the potential problems that linear and quadratic probing may cause with respect to clustering of the elements in a hash table.

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Part 1. Has the implementation of basic tree initialisation, insertion, and find been implemented completely and correctly? Note that code quality is addressed later.	
Initialisation produces a suitable empty tree. Insertion uses a suitable ordering on values to create an ordered binary tree. Duplicates are detected and ignored in insertion. Find uses the same ordering to correctly identify whether a value is in the tree.	(2)
As above but duplicates are not detected. However, the student can explain how to detect duplicates AND what the negative impact of not detecting duplicates is.	(1.5)
As above but duplicates are not detected	(1)
Some of the functionality is missing or does not work e.g. sometimes values are not inserted or sometimes find does not detect if a value is in the tree.	(0.5)
No attempt has been made	(0)

Part 1. Has an alternative balanced implementation been provided. This will most likely be an AVL tree but any other appropriate solution is okay.	
A good solution has been implemented that is clearly correct. The approach does not need to be optimal but should be clearly better than the unbalanced approach. Ideally the solution is a recognised algorithm that can be clearly referenced. Duplicates should be detected.	(2)
As above but there is a very minor flaw	(1.5)
The solution is a good attempt but fails to work (e.g. produce a balanced tree) in some cases.	(1)
A non-trivial attempt has been made but it is incomplete or clearly flawed	(0.5)
No attempt has been made	(0)

Part 1. The tree and usage statistics are printed in a sensible way	
The two printing functions are complete and sensible	(1)
There is a very minor issue with the printing functions e.g. the formatting is poor	(0.75)
The printing functions exist but miss an important bit of information	(0.5)
No attempt has been made	(0)

Part 1. The student understands how ordered binary trees work, how the balancing algorithm works, and the complexity of the associated algorithms.	
The student can give the complexity of find in an ordered binary tree and explain *why* this is the case. The student can compare this to a similar setting in lists. The student can explain the complexity of the balancing operations and the impact these have on the complexity of the find operation.	(3)
The student can state the complexities of the different operations but struggles to explain why these hold. There may be some confusion around how the balancing operations work.	(2)
The student can clearly explain everything related to the unbalanced operations but did not attempt the balanced tree part.	(2)
The student can attempt an explanation but the topic needs to be revised.	(1)
No attempt has been made	(0)

Part 2. Has the basic hash table been implemented including a suitable hash function and insertion/lookup using linear probing?	
The hash table is properly initialised. The hash function is sensible (unusual hash functions should be carefully justified). Insertion and lookup using linear probing has been implemented correctly (ignores duplicates, performs insertions when it should, always finds things that are in the table).	(3)
As above but there is a small flaw e.g. failure to detect duplicates	(2)
As above but a key component (e.g. the hash function) does not work as required or is far from optimal. For example, if the hash function does not produce a value in the required range or lookup using linear probing terminates too early.	(1)
No attempt has been made	(0)

Part 2. Has the hash table been improved with (i) quadratic probing, (ii) double hashing, and (iii) rehashing? Can the student explain how they work?	
All three optimisations have been implemented and work correctly. The student can explain how they work.	(3)
Two optimisations have been implemented and work correctly. The student can explain how they work.	(2)
Two or three optimisations have been implemented correctly but the student struggles to explain how they work sufficiently.	(1.5)
One optimisation has been implemented, works correctly, and can be explained.	(1)
An attempt at some optimisations has been made but they do not work correctly.	(0.5)
No attempt has been made	(0)

Part 2. The hash table and usage statistics are printed in a sensible way. Importantly, the number of clashes should be reported.	
The two printing functions are complete and sensible	(1)
There is a very minor issue with the printing functions e.g. the formatting is poor	(0.75)
The printing functions exist but miss an important bit of information	(0.5)
No attempt has been made	(0)

Part 2. Does the student understand the complexity of hash table operations and the issues related to probing?	
The student can explain the complexity of the insert and find functions. The student can discuss potential problems that linear and quadratic probing may cause with respect to clustering of the elements in a hash table.	(2)
The student can almost explain all of the concepts	(1.5)
The student can explain either the complexity or the problems with probing but not both	(1)
No attempt has been made	(0)

Is the code of good quality and does it handle errors properly? Note that many major errors are handled in the provided code.	
The quality of the code is good throughout with reasonable names for variables/functions, sufficient comments, and a sensible layout. All obvious errors are handled appropriately.	(2)
As above but there are a few minor issues	(1.5)
As above but there are some significant issues e.g. zero comments, completely illogical variable/function names	(1)
The quality of the code is poor throughout	(0.5)
The quality of the code is unacceptable	(0)

Is the code free of any memory errors?	
There are none	(1)
There are memory leaks only	(0.5)
There are other memory errors	(0)