Lab Exercise 9: Spellchecking using Trees and Hash-Tables

Duration: 3 sessions

Aims

To encourage you to find out about, and use, ordered-binary-trees and hash-tables for search.

Learning outcomes

On successful completion of this exercise, a student will:

- Have found out about the use of ordered-binary-trees.
- Have found out about the use of hash-tables.
- Have written C code to use oredered-binary-trees to store and search for strings.
- Have written C code to use hash-tables to store and search for strings.
- Understand the asymptotic complexity of their C code.

Summary

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 these 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 `insert`s them in your data-structure (ordered-binary-tree or hash-table)
    2. reads strings from a second text file and `find`s 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

  The code that reads words (both for the dictionary and for the text to be spell-checked) 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 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.
- `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 should copy all the files from `/opt/info/courses/COMP26120/problems/ex9` (use this path) to start.

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.

Deadlines: The unextended deadline is the end of your last scheduled lab for this exercise. If you need it, if you attend the lab you will get an automatic extension to the start of your next COMP26120 lab session (you must use submit to prove you finished in time and get it marked at the start of your next scheduled lab). You can also get an extension for good reason e.g. medical problems.

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

Description

Start by copying the starting code (described above) from `/opt/info/courses/COMP26120/problems/ex9` (use this path) into your `ex9` directory.

The interface - Table

The simple spelling checker you are given uses a "Table" data-type (data structures and functions) defined by `dict.h`, in which to store all the words in a dictionary. (The only data about each word is the word itself, stored as a string, and used as the key to locate the word in the data-structures). For each part of this exercise, you implement the Table data-type in different ways, so it can then be used by the code provided, and the results compared.

`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) - the nodes of the tree are created one-by-one by `insert`.
  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.
  You will need to decide what information to output and so what data to collect. E.g. for your hash table you might count the number of collisions so you can calculate the average per access; for your tree you might 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?).

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).

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.

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.

An optional extra could increase (double?) the hash table size when the table is getting full and then rehash into a larger table.

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 Process

You must use `labprint` and `submit` as normal.

`labprint` and `submit` will look for: `dict-tree.c` and `dict-hash.c`

There are 15 marks for each part (30 in total):

- The expected minimum for this lab involves using an unbalanced ordered-binary-tree (part 1), and single hashing using various algorithms (part 2).
  A correct implementation, with code that generally works is worth 10 marks for each part. Marks will be deduced if your implementation is far away from optimal.
- Bonus work for this lab involves using a self-balancing tree (part 1), and double-hashing for conflict resolution and automatically resizing a hash-table that is too small (part 2). Each part is worth up to an extra 3 marks.
- There are 2 marks for each part for a sensible discussion of the algorithm complexity (asymptotic properties) of the execution time for your function `find`.

You will get no credit for parts of the exercise that you have not implemented yourself.