Department of Computing and Software Information Systems
The University of Melbourne
COMP20003 Algorithms and Data Structures
Assignment 1
Semester 2, 2013
Handed out: Tuesday, 20 August
Due: 12:00 noon, Thursday 5 September

Purpose

The purpose of this assignment is for you to:

- Increase your proficiency in C programming, including your facility of dynamic memory allocation and your understanding of linked data structures, through programming a dictionary.
- Increase your understanding of how computational complexity can affect the performance of an algorithm by conducting orderly experiments on the output from your dictionary program and comparing the results of your experimentation with theory.

Background

A dictionary is an abstract data type that stores and supports lookup of ⟨key,value⟩ pairs. For example, in a telephone directory, the (string) key is a person or company name, and the value is the phone number. In a student record lookup, the key would be a student ID number and the value would be a complex structure containing all the other information about the student.

A dictionary can be implemented in C using a number of underlying data structures. Any implementation must support the operations: `makedict` a new dictionary and return a pointer to the dictionary; `insert` a new item (key,value pair) into a dictionary and return a pointer to the updated dictionary; `search` for a key in the dictionary, and return the associated value. Most dictionaries will also support the operation `delete` an item, given its key, and return a pointer to the updated dictionary.

Your task

In this assignment, you will code a dictionary in C using an underlying list structure. You will then use the dictionary, counting comparisons, experiment with different inputs, and compare your results with what you would expect based on theory.

In Stage 1 you will use a sorted linked list as the underlying data structure, while in Stage 2 you will use a skip list. Experimentation should be done for each part.

Stage 1 (10 marks) In Stage 1 of this assignment, you will construct a dictionary using a sorted linked list, coded in C, as the underlying data structure. Obligatory operations for Stage 1 are `insert`, `search`, and `delete` operation. You might also find it helpful to support a `makedict` operation, that returns a (possibly empty) dictionary, and will be necessary in Stage 2.

Your code *must* count and output the the number of key comparisons made for: (1) the overall the series of insertions used to build the dictionary; (2) each search operation. You will use these data in your experiments.

Your code should be easily extensible to allow for multiple dictionaries, which means that the functions for insertion, search, and deletion take as arguments not only the item being inserted or a key for searching and deleting, *but also a pointer to a dictionary.*

Stage 2 (5 marks) In Stage 2 you will write a new set of functions implementing a dictionary as a skip list (Pugh 1990). You will keep your sorted list code, and add the skip list code. For the purposes of this assignment, you are to have two different main functions, one for the sorted list dictionary and one for the skip list dictionary. While there are certainly other ways to organize multiple implementations, you are expected to adhere to this specification. The supplied `Makefile` suggests a possible way to organize your code.

Your skip list dictionary should allow all the operations listed above for the simple sorted list, except that you do *not* need to implement the `delete()` operation.

It is suggested that you start with the following parameters:

- A maximum number of levels of 10.
- A probability p of adding another level to a new node of 0.05,
- A maximum list length of 1000, not necessarily set in your program, for your experimentation.

You may vary these parameters once your code is working, and including the results of these variations in your experimentation is desirable.

You should not attempt Stage 2 until your Stage 1 program is working correctly.

Bonus (2 marks) Still too easy? If you have completed both Stage 1 and Stage 2 to a high standard and are still looking for a challenge, you can extend your program by using a pointer to a function that compares keys. Using a pointer allows you the flexibility of using the same code to compare string keys or integer keys, without changing the code.

For this program, you will use a function pointer to comparison functions, one of which compares integers (as in Stages 1 and 2), the other of which compares strings. Your program will look at the value of the single command line argument, and will chose which comparison function to use for the keys based on this value. If the command line argument is `'i'`, the keys compared are integers. If the command line argument is `'s'` the keys are strings. If there is no command line argument, your program should assume the keys are integers.

Information about function pointers can be found in Kernighan and Ritchie, Section 5.11, and in Moffat, Sections 10.4 and 10.5.

You should only attempt this part of the assignment once Stage 1 and Stage 2 are both working correctly, well-structured, and well-documented, and your experimental results have been collated and discussed. *No bonus marks will be given for using function pointers unless both Stages 1 and 2 have been completed to a high standard. The bonus part of the assignment will* **not** *be supported in lectures, workshops, or by staff input on the Discussion Board.*

Implementation Requirements

The following implementation conventions *must* be used:

- Your program will first build your data structure by inserting into the dictionary *from a file that contains the data.* The data file will have one item per line, first the key, then the value. See section on **Input** in this specification for an example.
- The dictionary program should take as its first argument the name of the input file (use `argv` for this).
- After reading to the end of the input file and building the dictionary, you will then look up keys that are *input from* `stdin` *(keyboard input or redirection).*
- A `Makefile` is provided for you in `/home/subjects/comp20003/local/sample_code/assg1`. You may use the `Makefile` *as is* or modify it, but your target for the sorted list dictionary *must* be called `listdict` and your target for the skip list dictionary *must* be called `skipdict`. To build the `listdict` program, type `make listdict` at the command line. Similarly, to build the `skipdict`,

type `make skipdict`. *Hint:* If you haven't used `make` before, try it with very simple files first. If it doesn't work, read the error messages carefully. A common problem in compiling multi-file executables is in the included header files.

Your code should be modular; each operation should be implemented as a separate function, and related functions should be grouped together in separate files. For example, it makes sense to put all your list operations in a separate file, since you could then easily reuse the list operations in other programs.

Input

The input format is key-value pairs, one pair to a line, *e.g.*:

```
16858     peer
 5778     crumple
10133     grippe
17535     poke
 6090     dazzle
```

Sample code for generating input data can be found in the directory `/home/subjects/comp20003/local/data_generation`. Some of this code can be used as is, while you might have to modify other code. The data above were generated using the C program `generate_key_value_pairs.c`. There is a `README` file in the directory, which points out the properties of the different programs.

For the purposes of this assignment, you can assume that the data are well-formatted, the input file is not empty, and the keys are unique.

Output Format

Your program output *must* be written to `stdout` and should adhere to the following format.

- Line 1 (written once only, after building the dictionary):
  $\langle Number\_of\_items\_inserted \rangle$ $\langle \text{“}Insertions\text{''} \rangle$ $\langle Number\_of\_comparisons\_made \rangle$
- Lines 2 and following, one line written for each search request:
  $\langle Key \rangle$ $\langle Value\_returned \rangle$ $\langle Number\_of\_comparisons\_made \rangle$

For example, with an input file of 100 data items, and three search requests, the output might be:

```
100  Insertions  2437
213  NOTFOUND     100
 57  abject        22
  2  NOTFOUND       1
```

Because this output format is not easily understandable on its own, your documentation should explain what the output means.

Experimentation

For both Stage 1 and Stage 2, you should report your experimental results in a file called `expts`, which you submit at the same time that you submit your code. Combine your results for Stage 1 and Stage 2 in a single `expts` file. For Stage 2 you are expected to compare your experimental results with your Stage 1 program, as well as to report on the skip list implementation results themselves.

Your experimentation should be orderly, that is you should systematically test a number of different inputs. Inputs can vary according to the number of data items n and according to arrangement (*random*, *reverse_sorted*, *etc.*). You *must* show your data: the size of the dictionary, the number of searches, the number of comparisons made during searching, *etc.* A table or a graph is often a good way of summarizing results.

If you use only keyboard input for searches, it is unlikely that you will be able to generate enough data to analyze your results. You should familiarize yourself with the powerful UNIX facilities for redirecting standard input (`stdin`) and standard output (`stdout`). You might also find it useful to familiarize yourself with Unix pipes `'|'` and possibly also `awk` for processing structured output. (For example, if your data output is structured in three columns, perhaps key, value, and comparisons, piping your output into `awk '{print $3 }'` will output only the third column.)

Resources

Two papers by William Pugh, the originator of the skip list, can be found on the LMS (*Assignments* → *Assignment 1*). For information about `argc` and `argv`, see Kernighan and Ritchie section 5.10 or Moffat section 7.11.

Sample C code that you can modify for specific tasks can be found in `/home/subjects/comp20003/local/sample_code/assg1`, including code for reading first from a file, as in your build, and subsequently reading from `stdin`(keyboard input).

Two locally-written papers containing useful guidelines on coding style and structure can be found on the LMS (→ *Resources*): *Project Coding Guidelines*, by Peter Schachte, and *C Programming Style*, written for Engineering Computation COMP20005 by Aidan Nagorcka-Smith. Thanks, Peter and Aidan! If you are interested in using Python to automate your experiments, tips on using Python to test, time, and graph your output can be found on `STILE`. (Thanks, Alex!) You may find this useful, or you may prefer some other way of displaying your data. For Unix tips, see the document *Introduction to Unix and MobaXterm* (LMS →Workshops). (Thanks, Ross!).

Additional Support

Your tutors will be available to help with your assignment at the scheduled workshop times. There is also a Discussion Forum entitled *Assignment 1* on the LMS, which you can use to post questions and answers. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the Discussion Forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking. For example, a question like, "How much data should I use for the experiments?", will not be answered; you must try out different data and see what makes sense.

In this subject, we support `MobaXterm` for `ssh` to the department machines `holly, lister, queeg, rimmer, toaster`, and `cat`, the excellent editor built into `MobaXterm`, and `gcc` on the department machines. While you are free to use whatever platforms and editors you prefer, these are the only tools you can expect help with from the staff in this subject.

Submission

Your programs *must* compile and run correctly on the CIS machines `cat, rimmer, holly, queeg, toaster,` and `lister` as listed above. You may have developed your program in another environment, but it still *must* run on the department machines at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on the department machines at reasonably frequent intervals.

Submit your project using the Unix command `submit comp20003 1`, and follow the prompts. Multiple files *must be submitted at the same time*, since each submission overwrites your previously submitted files. You should submit a `Makefile` (or `makefile`); if you don't submit a `Makefile` the supplied `Makefile` will be used. Your experimental results should be in a text file named `expts`, submitted at the same time as your code files.

Verify your submission with the command `verify comp20003 1`. You should save the verify file, to prove you have submitted, by using the command `verify comp20003 1 > verifyfile` *Note: Each time you submit, you will have to resubmit all your assignment files. As noted above, the submission program overwrites all the existing files from any previous submissions.* If in doubt about what you have submitted, check your `verifyfile` file.

Assessment

There are a total of 15 marks given for this assignment, 10 marks for Stage 1 and 5 marks for Stage 2. As detailed above, a further 2 bonus marks are available.

Your C program will be marked on the basis of accuracy, readability, and good C programming structure and style, including documentation. Your experimentation will be marked on the basis of orderliness, thoroughness, comparison with theory, and thoughtfulness.

Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

"Borrowing" of someone else's code without acknowledgment is plagiarism. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic honesty `http://academichonesty.unimemlb.edu.au` and details on plagiarism `http://academichonesty.unimemlb.edu.au/plagiarism`. Sometimes students plagiarize unintentionally. Please read the subject LMS page (*Assignments → Coding Style, Plagiarism, and other General Information → Unintentional Plagiarism*) and make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component on the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

References

Pugh,W., "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Communications of the ACM* **33**(6), 668–676, 1990.