COMP20003 Algorithms and Data Structures
Second (Spring) Semester 2018
[Assignment 1]
Olympics Datasets
Information Retrieval using Binary Search Trees

Handed out: Friday, 17 of August
Due: 8:00 AM, Monday, 3 of September

Purpose

The purpose of this assignment is for you to:

- Increase your proficiency in C programming, your dexterity with 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 with your program and comparing the results of your experimentation with theory.
- Increase your proficiency in using UNIX utilities.

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; `insert` a new item (key, value pair) into a dictionary; `search` for a key in the dictionary, and return the associated value. Most dictionaries will also support the operation `delete` an item.

Your task

In this assignment, you will create a simplified UNIX Information Retrieval system, a *search engine* as a concrete instance of a dictionary, and we'll use it to look up information about *Olympic athletes*. You'll search how many medals an athlete won overall the competitions.

There are two stages in this project. In each stage you will code a dictionary in the C programming language. A binary search tree will be the underlying data structure for both stages.

In this assignment the search keys are *not guaranteed to be unique*. In this assignment we use variants of the binary search tree designed to handle duplicates, i.e. by either dividing nodes using $<=$ $>$, or by using $<$ $>$ and a linked list for items with same key. You will use a `Makefile` to direct the compilation of two separate executable programs, one for Stage 1 and one for Stage 2, each of which uses a different variant of the binary search tree.

In both stages of the assignment, you will insert records into the dictionary from a file. You will then look up and output the records (data) contained by the dictionary, counting and outputting the number of key comparisons used in the search.

You will report on the number of key comparisons used for search, compare the number of key comparisons used by each stage, and analyse what would have been expected theoretically. The report should cover each file used to initialize the dictionary.

You are *not* required to implement the `delete` functionality.

Stage 1 (7 marks)

In Stage 1 of this assignment, your `Makefile` will direct the compilation to produce an executable program called `dict1`. The program `dict1` takes two command line arguments: the first argument is the name of the data file used to build the dictionary; the second argument is the name of the output file, containing the data located in the searches. The file consists of an unspecified number of records, one per line, with the following information:

ID – Unique number for each athlete
Name – Athlete's name
Sex – M or F
Age – Integer
Height – In centimeters
Weight – In kilograms
Team – Team name
NOC – National Olympic Committee 3-letter code
Games – Year and season
Year – Integer
Season – Summer or Winter
City – Host city
Sport – Sport
Event – Event
Medal – Gold, Silver, Bronze, or NA

The file was originally posted in kaggle. You can read more about the dataset here: Data source. Although it is not needed for this assignment, you can click on the overview and kernel tab on the web page to learn more about your dataset. We cleaned the dataset to remove nicknames embedded bewtween double quotes within the `<name>` field. The field `<name>` is an alphabetic string of varying length, containing the full name of an athlete. You may assume that this field contains no more than 128 characters. *The other columns* can be treated simply as the associated `<data>` field. Build a data structure of strings to save the associated data collected about the athlete. The maximum size of any string can be 128 characters. Each string is separated by a comma ",". It is a standard csv format where the delimiter used is a comma.

The dictionary key consists of the `<name>` field. The `<data>` is the information sought during lookup. Do not use the `<ID>`, the first column, as key. The ID is part of the `<data>`.

For the purposes of this assignment, you may assume that the input data is well-formatted, that the input file is not empty, and that the maximum length of an input record (a single full line of the csv file) is 512 characters. This number could help you fixing a reading buffer size.

In this first stage of the assignment, you will:

- Construct a binary search tree to store the information contained in the file specified in the command line argument. Each record should be stored in a separate Node.
- Search the binary search tree for records, based on their keys. The keys are read in from `stdin`, i.e. from the screen.

For testing, it is often convenient to create a file of keys to be searched, one per line, and redirect the input from this file. Use the UNIX operator < for redirecting input from a file.

- Examples of use:
  - `dict1 datafile outputfile` then type in keys; or
  - `dict1 datafile outputfile < keyfile`
- Your program will look up each key and output the information (the data found) to the output file specified by the second command line parameter. If the key is not found in the tree, you must output the word NOTFOUND.

  The number of key comparisons performed during both successful and unsuccessful lookups have to be written to `stdout`.
- Remember that the entries in the file do not necessarily have unique keys. Your search must locate *all* keys matching the search key, and output all the data found.

  In Stage 1 of the assignment you will locate the duplicates by continuing your search until you reach a leaf node, regardless of whether or not you have already found a match or matches.
- Example output:
  - output file (information):

    ```
    Cornelia Aalten (-Strannood) --> ID: 8 Sex: F || Age: 18 || Height: 168 || Weight: NA || Team: Netherlands || NOC: NED || Games: 1932 Summer || Year: 1932 || Season: Summer || City: Los Angeles || Sport: Athletics || Event: Athletics Women's 100 metres || Medal: NA ||

    Cornelia Aalten (-Strannood) --> ID: 8 Sex: F || Age: 18 || Height: 168 || Weight: NA || Team: Netherlands || NOC: NED || Games: 1932 Summer || Year: 1932 || Season: Summer || City: Los Angeles || Sport: Athletics || Event: Athletics Women's 4 x 100 metres Relay || Medal: NA ||

    Nir Lipo --> NOTFOUND
    ```
  - `stdout` (comparisons):

    ```
    Cornelia Aalten (-Strannood) --> 423
    Nir Lipo --> 401
    ```

Note that the key is output to both the file and to `stdout`, for identification purposes. Also note that the number of comparisons is only output at the end of the search, so there is only one number for key comparisons per key, even when multiple records have been located for that key.

The format need not be exactly as above. Variations in whitespace/tabs are permitted. *The number of comparisons above has been made up, do not take it as an example of a correct execution.*

Stage 2 (2 marks)

In Stage 2, you will code a dictionary where all the duplicate keys in the dictionary are returned, as previously, and additionally where the search is more efficient than in Stage 1. Input and output are as for Stage 1, with the information or `NOTFOUND` written to a file and the number of comparisons made during the search written to stdout.

In Stage 2, however, you will structure your tree so that once a key is found, all duplicate keys can be found without further key comparisons. Note that comparing a key to `NULL` is not a full (costly) key comparison, and is not counted as a key comparison in Stage 2 of this assignment when building the report.

Experimentation (4 marks)

You will run various files through your program to test its accuracy and also to examine the number of key comparisons used when searching different files. You will report on the key comparisons used by your Stage 1 dictionary `dict1` for various data inputs and the key comparisons used by your Stage 2 dictionary `dict2` for various data inputs too. You will compare these results with each other and, importantly with what you expected based on theory *(big-O)*.

Your experimentation should be systematic, varying the size and characteristics of the files you use (e.g. sorted or random), and observing how the number of key comparisons varies. Repeating a test case with different keys and taking the average can be useful.

Some useful UNIX commands for creating test files with different characteristics include `sort, sort -R` (`man sort` for more information on the `-R option`), and `shuf`. You can randomize your input data and pick the first x keys as the lookup keywords.

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 the `UNIX` program `awk` for processing structured output. For example, if you pipe your output into `echo ''abc:def'' | awk -F ':' '{print $1}'`, you will output only the first column (`abc`). In the example, `-F` specifies the delimiter. Instead of using `echo` you can use `cat filename.csv | awk -F ';' '{print $1}'` which will print only the first column of the filename.csv file. You can build up a file of numbers of key comparisons using the shell append operator `>>`, e.x. `your_command >> file_to_append_to`.

You will write up your findings and `submit your results separately through the Turnitin system`. You will compare your results with the two dictionary implementations (stage1 and stage2) and also compare these results to what you know about the theory of binary search trees.

Tables and graphs are useful presentation methods. Select only informative data; more is not always better.

You should present your findings clearly, in light of what you know about the data structures used in your programs and in light of their known computational complexity. You may find that your results are what you expected, based on theory. Alternatively, you may find your results do not agree with theory. In either case, you should state what you expected from the theory, and if there is a discrepancy you should suggest possible reasons. You might want to discuss space-time trade-offs, if this is appropriate to your code and data.

You are not constrained to any particular structure in this report, but a useful way to present your findings might be:

- Introduction: Summary of data structures and inputs.
- Stage 1 and Stage 2:
  - Data (number of key comparisons)
  - Comparison of the two stages
  - Comparison with theory
- Discussion

Implementation Requirements

The following implementation requirements must be adhered to:

- You *must* code your dictionary in the C programming language.
- You *must* code your dictionary in a modular way, so that your dictionary implementation could be used in another program without extensive rewriting or copying. This means that the dictionary operations are kept together in a separate .c file, with its own header (.h) file, separate from the main program. The `main.c` of stage1 can perfectly be the same `main` for stage2, in terms of dictionary operations.
- Your code should be easily extensible to allow for multiple dictionaries. This 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 particular dictionary*, e.g. `insert(dict, item)`.
- In each stage, you must read the input file *once only*.
- Your program should store strings in a space-efficient manner. If you are using `malloc()` to create the space for a string, remember to allow space for the final end of string `'\0' (NULL)`.
- A `Makefile` is *not* provided for you. The `Makefile` should direct the compilation of two separate programs: `dict1` and `dict2`. To use the Makefile, make sure is in the same directory of your code, and type `make dict1` to make the dictionary for Stage 1 and `make dict2` to make the dictionary for Stage 2. You must submit your makefile with your assignment. Hint: If you havent used make before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces. It is *not* a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

Data

The data files are provided at `/home/subjects/comp20003/assg1/datafiles/` which can be reached via connection to the engineering university server hosts `nutmeg.eng.unimelb.edu.au` or `dimefox.eng.unimelb.edu.au`. You can copy the datafiles using `scp` or `sftp` commands, e.x. `scp your_username@host:path_to_file local_path` or use `sftp` instead.

The data format is as specified above in *Stage 1*.

No attempt has been made to remove or prevent duplicate keys to each original file, in fact, there should be several duplicates as athletes compete in several events across several years. Similarly, no attempt has been made to seed the file with duplicate keys. Our script only formatted the data correctly making sure it complies with a csv standard specification. The *file_alternative_x.csv* files have suffered some transformations. The Athletes dataset contains roughly 270,000 records, and User Database contains 560,000. Exact figures are not given to discourage static memory allocation.

Resources: Programming Style (2 Marks)

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 below and adapted version of the *LMS Resources → C Programming Style*, written for Engineering Computation COMP20005 by Aidan Nagorcka-Smith. *Be aware that your programming style will be judged with 2 marks.*

```
/** **********************
* C Programming Style for Engineering Computation
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
* Definitions and includes
* Definitions are in UPPER_CASE
* Includes go before definitions
* Space between includes, definitions and the main function.
* Use definitions for any constants in your program, do not just write them
* in.
*
* Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
* Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
* style. Both are very standard.
*/

/**
* GOOD:
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
 ...

/**
* BAD:
*/

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
 ...

/** ****************************
* Variables
* Give them useful lower_case names or camelCase. Either is fine,
* as long as you are consistent and apply always the same style.
* Initialise them to something that makes sense.
*/
```

```

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

  int i = 0;
  int num_fifties = 0;
  int num_twenties = 0;
  int num_tens = 0;

 ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

  int i = 0;
  int numFifties = 0;
  int numTwenties = 0;
  int numTens = 0;

 ...
/**
* BAD:
*/

int main(int argc, char **argv) {

  /* Variable not initialised - causes a bug because we didn't remember to
  * set it before the loop */
  int i;
  /* Variable in all caps - we'll get confused between this and constants
  */
  int NUM_FIFTIES = 0;
  /* Overly abbreviated variable names make things hard. */
  int nt = 0

  while (i < 10) {
    ...
    i++;
  }

 ...

/** ********************
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
* One tab of indentation within either a function or a loop.
* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

/**
* GOOD:
*/

```

```c
int main(int argc, char **argv) {

 int i = 0;

 for(i = 100; i >= 0; i--) {
   if (i > 0) {
     printf("%d bottles of beer, take one down and pass it around,"
     " %d bottles of beer.\n", i, i - 1);
   } else {
     printf("%d bottles of beer, take one down and pass it around."
     " We're empty.\n", i);
   }
 }

 return 0;
}

/**
* BAD:
*/

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
 int i = 0;
 /* No space between variable declarations and the rest of the function.
 * No spaces around the boolean operators */
 for(i=100;i>=0;i--) {
 /* No indentation */
 if (i > 0) {
 /* Line too long */
 printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
 } else {
 /* Spacing for no good reason. */

 printf("%d bottles of beer, take one down and pass it around."
 " We're empty.\n", i);

 }
 }
 /* Closing brace not on its own line */
 return 0;}

/** ****************
* Braces:
* Opening braces go on the same line as the loop or function name
* Closing braces go on their own line
* Closing braces go at the same indentation level as the thing they are
* closing
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

  ...

  for(...) {
    ...
  }

  return 0;
```

```
}

/**
* BAD:
*/

int main(int argc, char **argv) {

  ...

  /* Opening brace on a different line to the for loop open */
  for (...)
  {
    ...
    /* Closing brace at a different indentation to the thing it's
    closing
    */
             }

 /* Closing brace not on its own line. */
      return 0;}

/** *************
* Commenting:
* Each program should have a comment explaining what it does and who created
* it.
* Also comment how to run the program, including optional command line
* parameters.
* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
* GOOD:
*/

/* change.c
*
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
13/03/2011
*
* Print the number of each coin that would be needed to make up some
change
* that is input by the user
*
* To run the program type:
* ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
*
* To see all the input parameters, type:
* ./coins --help
* Options::
*   --help                  Show help message
*   --num_coins arg         Input number of coins
*   --shape_coins arg       Input coins shape
*   --bound arg (=1)        Max bound on xxx, default value 1
*   --output arg            Output solution file
*
*/

int main(int argc, char **argv) {

  int input_change = 0;

  printf("Please input the value of the change (0-99 cents
  inclusive):\n");
  scanf("%d", &input_change);
```

```
  printf("\n");

  // Valid change values are 0-99 inclusive.
  if(input_change < 0 || input_change > 99) {
    printf("Input not in the range 0-99.\n")
  }

  ...

/**
* BAD:
*/

/* No explanation of what the program is doing */
int main(int argc, char **argv) {

  /* Commenting obvious things */
  /* Create a int variable called input_change to store the input from
  the
  * user. */
  int input_change;

  ...

/** ***************
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
  printf("Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
  /* lots of computation here, pushing the else part off the screen.
  */
  ...
} else {
  fprintf(stderr, "Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}
```

Additional Support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Piazza forum, using the folder tag *assignment1* for new posts. 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 CIS machines `nutmeg.eng.unimelb.edu.au` and `dimefox.eng.unimelb.edu.au`, the excellent editor built into `MobaXterm` or `Atom`, and gcc on the department machines. While you are free to use the platform and editor of your choice, these are the only tools you can "expect" help with from the staff in this subject. We'll always do our best to help you learn. Your final program must compile and run on the department machines.

Submission

You will need to make *two* submissions for this assignment:

- Your C code files (including your `Makefile`) will be submitted through the LMS page for this subject: *Assignments → Assignment 1 → Assignment 1: Code*.
- Your experiments report file will be submitted through the LMS page for this subject: *Assignments → Assignment 1 → Assignment 1: Experimentation*. This file can be of any format, e.g. .pdf, text or other.

Program files submitted (Code)

Submit the program files for your assignment and your `Makefile`.

If you wish to submit any scripts or code used to generate input data, you may, although this is not required. Just be sure to submit all your files at the same time.

Your programs *must* compile and run correctly on the CIS machines. 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.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary.

Experiment file submitted using Turnitin

As noted above, your experimental work will be submitted through the LMS, via the Turnitin system. Go to the LMS page for this subject: *Assignments → Assignment 1 → Assignment 1 Experiments Submission* and follow the prompts.

Your file can be in any format. Plain text or `.pdf` are recommended, but other formats will be accepted. It is expected that your experimental work will be in a single file, but multiple files can be accepted. **Add your username to the top of your experiments file**.

Please do *not* submit large data files. There is no need to query every key on the dictionary.

Assessment

There are a total of 15 marks given for this assignment, 7 marks for Stage 1, 2 marks for Stage 2, and 4 marks for the separately submitted Experimentation Stage. **2 marks will be given based on your**

C programming style.

Your C program will be marked on the basis of accuracy, readability, and good C programming structure, safety and style, including documentation. Safety refers to checking whether opening a file returns something, whether mallocs do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting, by choosing descriptive variable names.

Your experimentation will be marked on the basis of orderliness and thoroughness of experimentation, comparison of your results with theory, and thoughtful discussion.

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 elses 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 and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

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

Administrative issues

When is late? What do I do if I am late? The due date and time are printed on the front of this document. The lateness policy is on the handout provided at the first lecture and also available on the subject LMS page. If you decide to make a late submission, you should send an email directly to the lecturer as soon as possible and he will provide instructions for making a late submission.

What are the marks and the marking criteria Recall that this project is worth 15% of your final score. There is also a hurdle requirement: you must earn at least 15 marks out of a subtotal of 30 for the projects to pass this subject.

Finally Despite all these stern words, **we are here to help!** There is information about getting help in this subject on the LMS pages. Frequently asked questions about the project will be answered in the LMS discussion group.

NL,
August 17, 2018