COMP20003 Algorithms and Data Structures
Second (Spring) Semester 2020
[Assignment 1]
Melbourne Census Dataset
Information Retrieval using a Linked List

Handed out: Monday, 17 of August
Due: 8:00 AM, Friday, 28 of August
Marks: 10 (10% of total mark)

Purpose

The purpose of this assignment is for you to:

- Improve your proficiency in C programming and your dexterity with dynamic memory allocation.
- Demonstrate understanding of a concrete data structure (linked list).
- Practice multi-file programming and improve 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.

Your task

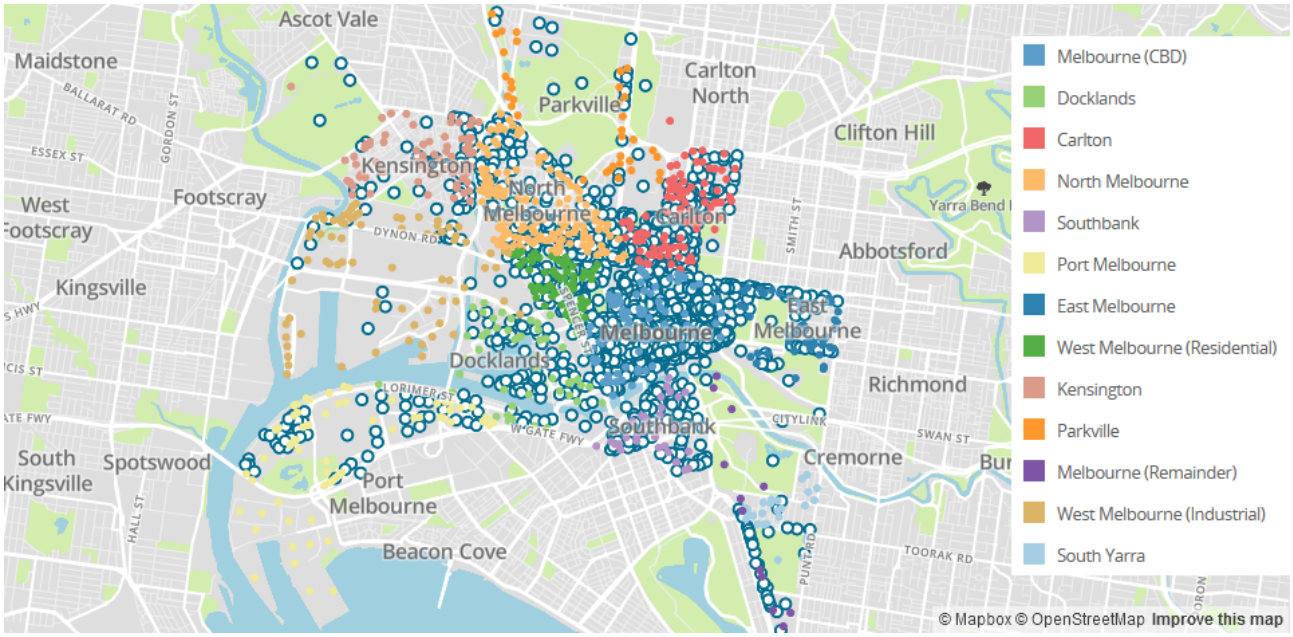
In this assignment, you will create a simple dictionary based on an unsorted linked list to store information from the City of Melbourne Census of Land Use and Employment (CLUE). A user will be able to search this dictionary to retrieve information about businesses in Melbourne using the business name (key).

Your implementation will build the dictionary by reading census data from a file and inserting each property record as a node in a linked list. You will also implement a method to search for a key in the list, outputting any records that match the key. Note that keys are not guaranteed to be unique!

Dataset

The dataset comes from the City of Melbourne Open Data website (https://data.melbourne.vic.gov.au/), which provides a variety of data about Melbourne that you can visualize online. The dataset used in this project is a subset of the **Business establishment trading name and industry classification 2018** dataset, accessed from:
https://data.melbourne.vic.gov.au/Business/Business-establishment-trading-name-and-industry-c/vesm-c7r2

The dataset describes businesses in the Melbourne area. Each row in the dataset includes the following fields:
Census year - the year in which surveying was completed (2018)
Block ID - an ID number to identify city blocks (about 606 in total)
Property ID - an ID number to identify an individual property



Visualization of the dataset. Colours indicate CLUE small area; open circles indicate locations with multiple business establishments. You can make your own interactive graphs and visualizations of this dataset here.

Base property ID – an ID number to identify a parcel of land (which may contain multiple properties)
CLUE small area – city area name (e.g., Melbourne CBD)
Trading name – name of the business located at this property
Industry (ANZSIC4) code – numeric code to describe the industry in which the business operates
Industry (ANZSIC4) description – name of the industry corresponding to the code
x coordinate – longitude of the establishment
y coordinate – latitude of the establishment
Location – location as a (lat,long) pair (used for visualization)

The fields <CLUE small area>, <Trading name>, <Industry (ANZSIC4) description>, and <Location> are alphabetic strings of varying length. You may assume that none of these fields are more than 128 characters. The dataset is in csv format, with each field separated by a comma. Note that string fields may contain commas (the <Location> field always contains a comma); in these cases the string is enclosed in quotation marks.

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 choose a reading buffer size.

The <Trading name> should be used as the key in your dictionary implementation.

Implementation Details

Your Makefile should produce an executable program called dict. This program should take two command line arguments: (1) the name of the data file used to build the dictionary, and (2) the name of an output file.

Your `dict` program should:

- Construct an unsorted linked list to store the information contained in the data file specified in the command line argument. Each record (row) should be stored in a separate Node.
- Search the linked list for records, based on keys. The keys will be read in from `stdin`, i.e. from the screen. Remember that the entries in the file do not necessarily have unique keys, so your search must locate *all* keys matching the search key, and output all the data found.
- 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`.

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

Examples of use:

- `dict1 datafile outputfile` then type in keys; or
- `dict1 datafile outputfile < keyfile`

Example output

This is an example of what might be output to the file after searching for two keys:

```
In a Rush Espresso --> Census year: 2018 || Block ID: 44 || Property ID: 105956 || Base property ID: 105956 || CLUE small area: Melbourne (CBD) || Industry (ANZSIC4) code: 4511 || Industry (ANZSIC4) description: Cafes and Restaurants || x coordinate: 144.96174 || y coordinate: -37.81561 || Location: (-37.81560561, 144.9617411) ||

In a Rush Espresso --> Census year: 2018 || Block ID: 1101 || Property ID: 108973 || Base property ID: 108973 || CLUE small area: Docklands || Industry (ANZSIC4) code: 4511 || Industry (ANZSIC4) description: Cafes and Restaurants || x coordinate: 144.95223 || y coordinate: -37.81761 || Location: (-37.81761044, 144.9522269) ||

Tim Hortons --> NOTFOUND
```

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*.

Requirements

The following implementation requirements must be adhered to:

- You *must* write your implementation in the C programming language.
- You *must* write your code in a modular way, so that your implementation could be used in another program without extensive rewriting or copying. This means that the linked list operations are kept together in a separate .c file, with its own header (.h) file, separate from the main program.

- Your code should be easily extensible to different 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)`.
- Your implementation must read the input file *once only*.
- Each record should be stored in a struct with separate variables to store the separate data fields (e.g., separate variables for the record's `<Block ID>`, `<CLUE small area>`, `<Industry (ANZSIC4) code>`, etc.). Each of these variables should be an appropriate type and size for the data it stores.
- 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 your program. To use the Makefile, make sure is in the same directory of your code, and type `make dict` to make the dictionary. You must submit your makefile with your assignment.

Hint: If you haven't 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.

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.*

```
1  /** ***********************
2  * C Programming Style for Engineering Computation
3  * Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
4  * Definitions and includes
5  * Definitions are in UPPER_CASE
6  * Includes go before definitions
7  * Space between includes, definitions and the main function.
8  * Use definitions for any constants in your program, do not just write them
9  * in.
10 *
11 * Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
12 * Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
13 * style. Both are very standard.
14 */
15
16 /**
17 * GOOD:
18 */
19
20 #include <stdio.h>
21 #include <stdlib.h>
22 #define MAX_STRING_SIZE 1000
23 #define DEBUG 0
24 int main(int argc, char **argv) {
25   ...
26
```

```c
/**
* 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++;
  }

 ...
```

```
93
94 /** ********************
95 * Spacing:
96 * Space intelligently, vertically to group blocks of code that are doing a
97 * specific operation, or to separate variable declarations from other code.
98 * One tab of indentation within either a function or a loop.
99 * Spaces after commas.
100 * Space between ) and {.
101 * No space between the ** and the argv in the definition of the main
102 * function.
103 * When declaring a pointer variable or argument, you may place the asterisk
104 * adjacent to either the type or to the variable name.
105 * Lines at most 80 characters long.
106 * Closing brace goes on its own line
107 */
108
109 /**
110 * GOOD:
111 */
112
113 int main(int argc, char **argv) {
114
115  int i = 0;
116
117  for(i = 100; i >= 0; i--) {
118    if (i > 0) {
119      printf("%d bottles of beer, take one down and pass it around,"
120      " %d bottles of beer.\n", i, i - 1);
121    } else {
122      printf("%d bottles of beer, take one down and pass it around."
123      " We're empty.\n", i);
124    }
125  }
126
127  return 0;
128 }
129
130 /**
131 * BAD:
132 */
133
134 /* No space after commas
135 * Space between the ** and argv in the main function definition
136 * No space between the ) and { at the start of a function */
137 int main(int argc,char ** argv){
138  int i = 0;
139  /* No space between variable declarations and the rest of the function.
140  * No spaces around the boolean operators */
141  for(i=100;i>=0;i--) {
142  /* No indentation */
143  if (i > 0) {
144  /* Line too long */
145  printf("%d bottles of beer, take one down and pass it around, %d
146 bottles of beer.\n", i, i - 1);
147  } else {
148  /* Spacing for no good reason. */
149
150  printf("%d bottles of beer, take one down and pass it around."
151  " We're empty.\n", i);
152
153  }
154  }
155  /* Closing brace not on its own line */
156  return 0;}
157
158 /** ***************
```

```
159 * Braces:
160 * Opening braces go on the same line as the loop or function name
161 * Closing braces go on their own line
162 * Closing braces go at the same indentation level as the thing they are
163 * closing
164 */
165
166 /**
167 * GOOD:
168 */
169
170 int main(int argc, char **argv) {
171
172   ...
173
174   for(...) {
175     ...
176   }
177
178   return 0;
179 }
180
181 /**
182 * BAD:
183 */
184
185 int main(int argc, char **argv) {
186
187   ...
188
189   /* Opening brace on a different line to the for loop open */
190   for(...)
191   {
192     ...
193     /* Closing brace at a different indentation to the thing it's
194     closing
195     */
196           }
197
198  /* Closing brace not on its own line. */
199       return 0;}
200
201 /** **************
202 * Commenting:
203 * Each program should have a comment explaining what it does and who created
204 * it.
205 * Also comment how to run the program, including optional command line
206 * parameters.
207 * Any interesting code should have a comment to explain itself.
208 * We should not comment obvious things - write code that documents itself
209 */
210
211 /**
212 * GOOD:
213 */
214
215 /* change.c
216 *
217 * Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
218 13/03/2011
219 *
220 * Print the number of each coin that would be needed to make up some
221 change
222 * that is input by the user
223 *
224 * 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:
*/

```

```c
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

Your C code files (including your `Makefile` and any other files needed to run your code) should be submitted through the LMS under **Assignment 1: Code** in the **Assignments** tab.
Your programs *must* compile and run correctly on JupyterHub. You may have developed your program in another environment, but it still *must* run on JupyterHub 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 JupyterHub 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.

Assessment

There are a total of 10 marks given for this assignment.

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.

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 integrity 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.

Late policy

The late penalty is 10% of the available marks for that project for each day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that departmental servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, **we are here to help!** There is information about getting help in this subject on the LMS. Frequently asked questions about the project will be answered in Piazza.