COMP20003 Algorithms and Data Structures
Second (Spring) Semester 2017

[Assignment 1]

Fast Auto-Complete!
as a Ternary Search Tree

Handed out: Friday, 18 of August
Due: 12:00 Noon, Monday, 4 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 Ternary Search Tree.
- 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

Autocomplete highly enhances user experience for two reasons: it minimises the number of keystrokes needed to specify a string input, while at the same time helps users with a suggestion of useful closely related inputs. Autocomplete typically wants to suggest the most likely strings that share the prefix specified so far. Typically the prefix is extended every time a new keyboard event is triggered, hinting that any successful autocomplete function should be as fast as the average human typist. Surveys suggest that the average person types 200 characters per minute, which means that ideally our autocomplete algorithm should not take longer than 0.3 seconds to suggest the correct autocompletion candidates. If autocomplete takes much longer, users will dislike our solution. Another important aspect is that more than one candidate should be suggested according to their likelihood. If we fail suggesting the correct autocompletions users will not like our implementation either.

Autocomplete is efficiently implemented in the industry by a ternary search tree, which is a concrete data type that stores and supports lookup of keys. Note that ternary tree is like a Binary tree but with three children (left,equal,right). Ternary search trees are used as an associative data structure (trie), where each node does not store the full key (stores a single character of the key). The position of the node in the tree defines the key associated with the node, where all its equal subtree descendants share the same key prefix. A key can have an associated weight to indicate its likelihood.

Autocompletion can be implemented in C using a number of abstract data structures and underlying concrete data structures. Any implementation must support the operations: `insert` a new item `<key, weight>` into a tree; `find_and_traverse` the tree given a prefix, and return (or print) a list of pairs `<key, weight>` where all keys share the same prefix.

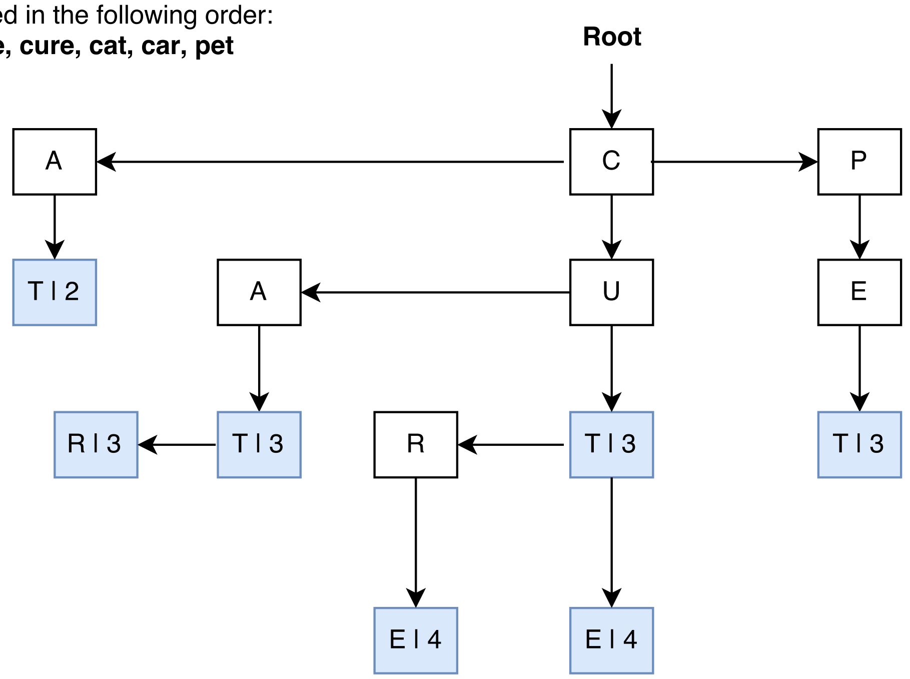
1 Ternary Search Tree

A ternary search tree has nodes with three children (left,equal,right). It is used as an associative data structure, meaning that a node does not need to contain the full key, but the position of the node in the tree determines the key of the node. For this reason, a node will have six components:

- A `single character`, containing 1 character of the key,
- An `integer` weight,
- A `boolean` flag marking whether this character is the `end_of_key`, i.e. the last character a complete key,
- Three `pointers` to nodes named left, equal and right.

Remember to initialize your pointers to `NULL` and the boolean flag to `false`.

Keys inserted in the following order:
cut, at, cute, cure, cat, car, pet



Every Time you traverse an "equal edge" (a vertical edge), you advance to the next character of the key.
Nodes with Blue Background represent the end of the key (they should have the end_of_string flag to true)
The number after "|" stands for the weight. In this Ternary search tree the weight is just the length of the key

Figure 1: An example of a Ternary Search Tree.

The insert function skeleton is as follows:

```
/**
* Given a call like root = insert(root, ”cat”, 3)
* the code below should return the tree containing the new key
*/
node* insert( node* pNode, char* word, int weight ){
   if (pNode is NULL) {
      /**
      * Create a new pNode, and save a character from word
      */

   }

   if (/* current char in word is smaller than char in pData */){
      /**
      * Insert the character on the left branch
      */
      pNode->left = insert(pNode->left, word, weight);
```

```c
   }
   else if (/* current char in word is equal to char in pData */) {
      if (/* next char in word is '\0' */){
         /**
         * set pNode end_of_key_flag to true and assign weight
         */
      }
      else{
         /**
         * If the word contains more characters, try to insert them
         * under the equal branch
         */
         pNode->equal = insert(pNode->equal, word+1, weight);
      }
   }
   else{
      /**
      * if current char in word is greater than char in pData
      * Insert the character on the right branch
      */
      pNode->right = insert(pNode->right, word, weight);
   }
   return pNode;
}
```

Figure 1 shows the resulting ternary search tree, after inserting the keys `cut, at, cute, cure, cat, car, pet`.

Once the tree is built, we have to implement the `find_and_traverse` function in order to print or retrieve all the possible keys following a prefix. The function below is the skeleton to find and print all the keys. It should be adapted if you require to return a list with all the keys instead of just printing them:

```c
void find_and_traverse( node* pNode, char* prefix ){
   /**
   * Find the node in the tree that represents the prefix
   * pNode will point there if we reached the '\0' symbol,
   * if prefix does not exist, then pNode should be NULL
   */
   while( *prefix != '\0' && pNode != NULL){
      /**
      * Find tree position for prefix
      */

      //left branch
      if( /* prefix char is smaller than pNode char */ ){
         /**
         * Go to left branch
         */
         continue;
      }

      //right branch
      if(/* prefix char is greater than pNode char */){
         /**
         * Go to right branch
         */
         continue;
      }

      //equal branch
      if( /*prefix char is equal to pNode char*/ ){
         /**
```

```
            * Go to equal branch, and check next char in prefix
            */
            continue;
        }
    }
    /**
    * At this point, pNode should be pointing to
    * the node where the prefix is contained.
    */

    if(/*pNode is not NULL*/){

        /**
        * Include the prefix itself as a candidate
        * if prefix is a key
        */

        if (/* pNode end of string flag is true*/)
        {
            buffer[strlen(prefix)+1] = '\0';
            printf( "%s\n", buffer);
        }

        /**
        * print all the keys that contain the prefix
        */

        traverse(pNode, buffer, strlen(prefix) );
    }
}

/**
* Tree traversal from a given node
*/
void traverse(struct Node* pNode, char* buffer, int depth)
{
    if (pnode is NULL) return;

    /**
    * Go first to the left most branch
    */
    traverse(pNode->left, buffer, depth);

    /**
    * If no more left branches, then save the character
    */
    buffer[depth] = pNode->data;

    if (/* pNode end of string flag is true*/)
    {
        buffer[depth+1] = '\0';
        printf( "%s\n", buffer);
    }

    /**
    * and go to the equal branch, advancing
    * to the next character of the key
    */
    traverse(pNode->equal, buffer, depth + 1);

    /**
    * Finally go to the branches that contain
    * characters greater than the current one in the buffer
    */
    traverse(pNode->right, buffer, depth);

```

```
}
```

Your task

In this assignment, you will create a simplified UNIX *AutoCompleMe* program, and will use it to autocomplete inputs given a variety of datasets: More than 2.2M Actors, 1.2M Actresses and 400K Directors from IMDB, 100K city names, and 10K words from a wiki in English.

There are two stages in this project. In each stage you will code in the C programming language. A ternary search tree and a linked list will be the underlying data structures for both stages.

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 the same ternary search tree.

In both stages of the assignment, you will insert records into the ternary search tree from a file. You will then look up some prefix and output the keys and weights contained by the tree, counting and outputting the number of key comparisons used to find the prefix. In the second stage keys have to be sorted in descending order according to their weights.

You will report on the number of key comparisons used to find a prefix, compare the number of weight comparisons used to sort, and analyse what would have been expected theoretically. The report should cover each dataset file used to initialize the ternary search tree.

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

Stage 1 (9 marks)

In Stage 1 of this assignment, your `Makefile` will direct the compilation to produce an executable program called `autocomplete1`. The program `autocomplete1` 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 autocompletions located for each prefix. The file consists of an unspecified number of records, one per line, where the format of each record is:

```
<weight>;<key>
```

The field `<key>` is an alphanumeric string of varying length, containing the key (city, word or actor/actress full names). You may assume that this field contains no more than 250 characters. The `<weight>` field is a positive integer containing information about the importance of the key. For instance, the weight of a city is its population, the weight of a word is its frequency within a public wiki, and the weight of an actor/actress name is the amount of data associated with them in imdb, in terms of number of characters. Each field is separated by a semicolon ";".

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 key is 250 characters. This number could help you fixing a reading buffer size.

In this first stage of the assignment, you will:

- Construct a ternary search tree to store the information contained in the file specified in the command line argument.

- Search the ternary search tree for a key prefix. The key prefixes are read in from `stdin`, i.e. from the screen.

  For testing, it is often convenient to create a file of key prefixes 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:

  - `autocomplete1 datafile outputfile` then type in key prefix; or
  - `autocomplete1 datafile outputfile < keyfile`

- Your program will look up each key prefix and output all the keys that match the prefix (autocompletion candidates) 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`.

- Example output:

  - output file (information):

    ```
    Prefix:  Melb
    key:  Melbeck, Germany --> weight:  3358
    key:  Melbourn, United Kingdom --> weight:  4394
    key:  Melbourne Beach, Florida, United States --> weight:  3101
    key:  Melbourne, Arkansas, United States --> weight:  1848
    key:  Melbourne, Florida, United States --> weight:  76068
    key:  Melbourne, United Kingdom --> weight:  4404
    key:  Melbourne, Victoria, Australia --> weight:  3730206
    key:  Melbu, Norway --> weight:  2161

    Prefix:  Barcel
    key:  Barcellona Pozzo di Gotto, Italy --> weight:  41258
    key:  Barcelona, Philippines --> weight:  3541
    key:  Barcelona, Spain --> weight:  1621537
    key:  Barcelona, Venezuela --> weight:  424795
    key:  Barceloneta, Puerto Rico --> weight:  22322
    key:  Barcelonne-du-Gers, France --> weight:  1379
    key:  Barcelonnette, France --> weight:  3487
    key:  Barcelos, Brazil --> weight:  7353
    key:  Barcelos, Portugal --> weight:  19085

    Prefix:  barcel
    NOTFOUND
    ```

    Note that key prefixes are case sensitive (B vs. b)

  - `stdout` (comparisons):

    ```
    Prefix:  Melb found with 8 char comparisons
    Prefix:  Barcel found with 11 char comparisons
    Prefix:  barcel found with 9 char comparisons
    ```

Note that the key prefix 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 representing char comparisons per key prefix. Note that once the key prefix is found no more char comparisons are needed, retrieving the keys matching the prefix is only a traversal of the tree below the node representing the prefix.

Stage 2 (2 marks)

In Stage 2 of this assignment, your `Makefile` will direct the compilation to also produce an executable program called `autocomplete2`

In this stage, you will code a sorting algorithm to sort the key candidates in descending order, so we can suggest first the key with the highest weight, then the second best and so on. In stage 1 you just needed to print the key candidates while traversing the tree below the key prefix. In this stage the traversal has to return a list containing all the key candidates, so we can call our sorting algorithm. You have to implement `selection sort`, which given a list of nodes containing pairs `<key, weight>`, returns the same list but in descending weight order.

The output for the same queries of the example shown above should be:

```
Prefix:  Melb
key:  Melbourne, Victoria, Australia --> weight:  3730206
key:  Melbourne, Florida, United States --> weight:  76068
key:  Melbourne, United Kingdom --> weight:  4404
key:  Melbourn, United Kingdom --> weight:  4394
key:  Melbeck, Germany --> weight:  3358
key:  Melbourne Beach, Florida, United States --> weight:  3101
key:  Melbu, Norway --> weight:  2161
key:  Melbourne, Arkansas, United States --> weight:  1848


Prefix:  Barcel
key:  Barcelona, Spain --> weight:  1621537
key:  Barcelona, Venezuela --> weight:  424795
key:  Barcellona Pozzo di Gotto, Italy --> weight:  41258
key:  Barceloneta, Puerto Rico --> weight:  22322
key:  Barcelos, Portugal --> weight:  19085
key:  Barcelos, Brazil --> weight:  7353
key:  Barcelona, Philippines --> weight:  3541
key:  Barcelonnette, France --> weight:  3487
key:  Barcelonne-du-Gers, France --> weight:  1379


Prefix:  barcel
NOTFOUND
```

`stdout` (comparisons):

```
Prefix:  Melb found with 8 char comparisons
Selection Sort:  36 weight comparisons
Prefix:  Barcel found with 11 char comparisons
Selection Sort:  45 weight comparisons
Prefix:  barcel found with 9 char comparisons
Selection Sort:  0 weight comparisons
```

The format need not be exactly as above. **Only Variations in whitespace/tabs are permitted**.

Experimentation (2 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 key prefixes across different files. You will report on the number of comparisons used by your Stage 1 ternary search tree `autocomplete1` for various data file inputs and the key prefixes. For Stage 2 you will report on the number of comparisons used by the sorting algorithm for various data file inputs and key prefixes. You will compare these results with what you expected based on theory *(big-O)*.

Your experimentation should be systematic, varying the size of the files you use and their characteristics (e.g. initially sorted data, randomly sorted initial data, total number of keys), and observing how the number of key comparisons varies given different sizes of key prefixes. 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 using the command `head <filename>`. If you use `tail <filename>` it will display the last x keys.

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 (stage1 and stage2) and also compare these results to what you know about the theory of Ternary 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 (datafile #keys and characteristics, prefix key queries, number of key or weight comparisons)

  - Comparison with theory

- Discussion

Implementation Requirements

The following implementation requirements must be adhered to:

- You *must* code your tree in the C programming language.
- You *must* code your tree in a modular way, so that your tree implementation could be used in another program without extensive rewriting or copying. This means that the tree operations are kept together in a separate .c file, with its own header (.h) file, separate from the main program. You can have two `main1.c`, one for stage1 and one `main2.c` for stage2.
- Your code should be easily extensible to allow for multiple trees. This means that the functions for insertion, search, and deletion take as arguments not only the item being inserted or a key for searching, *but also a pointer to a particular tree*, e.g. `insert(tree, 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: `autocomplete1` and `autocomplete2`. To use the Makefile, make sure is in the same directory of your code, and type `make autocomplete1` to make the tree for Stage 1 and `make autocomplete2` to make the tree 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 doesnt 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 through MobaX-Term, Filezilla, Cyberduck, etc.

The data format is, as specified above:

```
<weight>;<key>
```

No attempt has been made to remove or prevent duplicate keys to each original file (shouldn't be an issue, if a key that already exists is inserted again, you can safely overwrite the weight). 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, and that ";" is only used as a delimiter.

The datasets contain:

- More than 2.2M Actors, 1.2M Actresses and 400K Directors from IMDB. The weights are the size of the string containing all the movies they participated.

- 100K city names. The weights are their population.
- 10K words from a wiki in English. The weights are their frequency in the wiki.

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 *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.*

Additional Support

Your tutors will be available to help with your assignment during the scheduled workshop times. There is also a Discussion Forum *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 CIS machines `nutmeg.eng.unimelb.edu.au` and `dimefox.eng.unimelb.edu.au`, the excellent open source hackable editor `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*.

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.

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. No need to query every key on the datafiles.

Assessment

There are a total of 15 marks given for this assignment, 9 marks for Stage 1, 2 marks for Stage 2, and 2 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) on 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→subject information→subject outline page. If you decide to make a late submission, you should use the normal LMS submission method and notify by email the lecturer and head tutor (Grady). Our emails are available in the LMS.

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, 2017