Database Project Report

Brett Gedvilas
CSCI 2421 Fall 2016

November 30, 2016

Project Summary

The goal of this project was to create a simple database in c++. Broadly speaking, a database is simply a structure that contains a set of (usually) related information. In our case, the database was to serve as an address book where each entry in the database can be considered a contact. Then, each entry contains information specific to that contact such as: name, address, phone number, email etc. The implementation of the database uses a binary search tree as the main data structure using a unique contact ID# as the key for correctly inserting and traversing the tree. Moving deeper into the structure of the tree, each contact is an instance of an abstract data type called Record that holds the various fields for the contact.

After defining the main structure of the database, functionality had to be added to allow the user to make meaningful queries to the contents of the database and generate output.

Summary of Functions

These functions all pertain to the user interaction with the database and demonstrate the functionality of the database. The parameters and some smaller functions have been omitted for clarity.

- searchTree()
- subSearch()
- visitNodes()
- modifyEntry()
- removeEntry()
- writeDatabase()
- readFile()
- sortList()
- selectFields()

Design Document

C++ Implementation of an Address Book Database

By: Brett Gedvilas
Student ID: 810-74-9234
Date: November 30, 2016

Problem Description

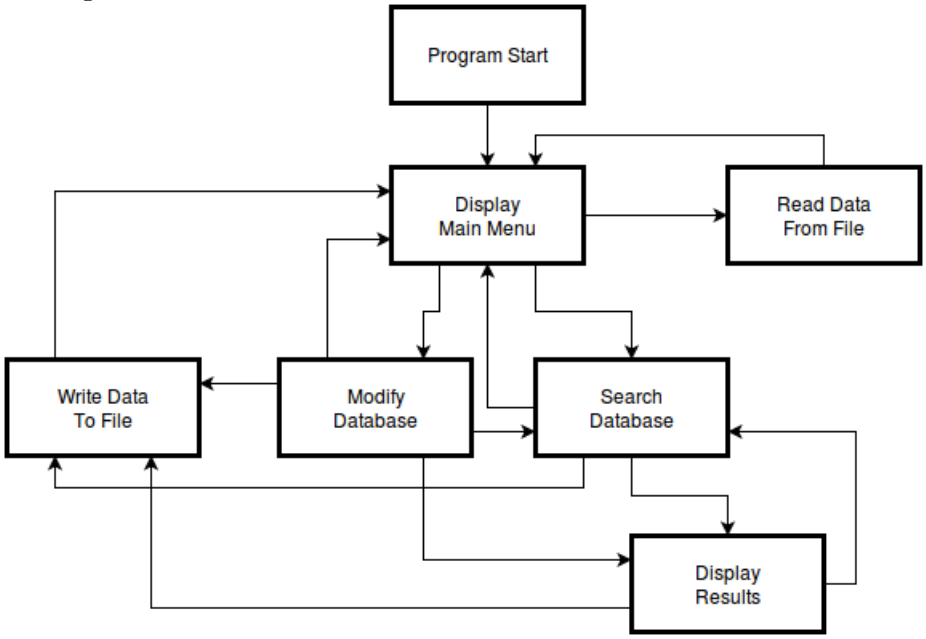
Being able to access and gain meaningful insights from large sets of data can be a very effective tool in a number of different academic or professional fields. Unfortunately, a data set that includes thousands or millions of entries, each having their own fields, presents a challenge of how to efficiently store, search, sort, and display the data as needed.

Our specific problem domain was to create a database that contains the contacts of an address book. This database must satisfy a number of functional goals which allow the user to easily navigate the database and direct output of their choice to a desired location. The program can read in a list of contacts from an external file and then allow the user to perform a variety of operations on the data such as:

- Modify the database manually by adding and deleting records in the database or modifying an existing entry.
- Search the database using a field of the users choice and perform sub-searches on the search results.
- Browse search results
- Sort the results from a database query using a given field.
- Display selected fields of search results.
- Output the database or search results to a file.

Software Architecture

Basic Program Flow



Input Requirements

The inputs for the database come in three main flavours.

1. The bulk of the data for each record will be read in from a file and directly parsed and stored in the main binary search tree. The program will assume a specific file format for the records and any deviation from this will cause the file to be rejected.

2. The second input method is for the user to create a new record and directly add data to each field. In this case, the program will perform basic input validation to ensure the user has entered appropriate values.

3. The third type of input will come from the user selecting menu items and performing searches on the database. In the case of menu selection, the program will ask the user to select from a numeric list of menu options. The user must enter a valid integer corresponding to a menu option.

Output Requirements

The program can pipe information from the database to an output .txt file as ascii coded text. The format of the output mirrors the format of the input file for the database. Each record is separated by a |. The fields of each record are newline delimited, and within the affiliate field, each affiliate is separated by a semi- colon, and the sub-fields of each affiliate

(if used) are delimited by a comma. The user will have the option of which fields to include in the output file. The output can consist of the entire database, or the results of a search query. A detailed description of each output field is shown below.

ID# A unique 9 digit unsigned integer. Stored as a c++ unsigned int data type. Any 9 digit integer is valid, provided it is a non-zero positive value and does not match any existing ID#. The ID# 000000000 will be considered invalid. This gives the database up to 1 billion possible unique ID#'s

First Name String

Middle Name String

Last Name String

Company Name String

Home Phone String

Office Phone String

Email String

Mobile Phone String

Street Address String

City String

State String

Zip Code String

Country String

Affiliate Linked List - Each field of an affiliate (first name, last name, email and phone number) is stored as a string.

Problem Solution Discussion

The algorithm for my solution can be described in a number of logical steps. First, read in data from a file to populate the database. While the data is being read in, parse into contacts, insert each individual contact into a binary tree. To remove a contact, search binary tree for corresponding ID# and delete the node holding that contact. To search database, get which field to search for and search term, visit every node in the binary tree and compare selected field to search term. If match is found, insert the contact information into a linked list.

Data Structures

The main data structure used to organize the data base will take the form of a Binary Search Tree. Each record or contact in the address book will be represented by a node in the tree, organized by the unique ID number of each contact. We can leverage the power of the Binary Search Tree in order to efficiently search by ID number but searching using other fields will require each node in the tree to be visited.

Two main areas surface when considering the data structures to be used in the program: How to store the affiliates for each contact, and how to store search results from a database query.

1. I decided that the affiliates for each contact will be stored using a linked list, taking advantage of the c++ STL list container. Broadly speaking, a queue or stack does not translate well to this application. A user may choose to access an affiliate not at the top of the stack (or front of queue) and that alone excludes them from consideration. In this case, I decided that an array or vector were also not appropriate for a few reasons. First, a c++ array is constricted to a certain size and the number of affiliates for a contact is unknown until runtime. Furthermore, just creating an array with a size that would likely be large enough for the number of affiliates would lead to a large amount of wasted memory. Even a contact with no affiliates would still have an array of size n allocated for it. Similarly, while a vector is implemented as a dynamic array and would account for resizing to store more elements, they are still allocated with extra space. Again, since I'm making the assumption that the number of affiliates will remain rather relatively small this leads to a fair amount of wasted memory.

   The use of a linked list helps to alleviate some of these problems. Although a linked list is less efficient at accessing its elements than an array or vector, because the number of entries will remain small the difference between an access time on the order of O(1) or constant, and an access time on the order of O(n) or linear will be negligible. The trade off is one of speed vs. memory, but traversing a linked list of, say, 15 affiliates is trivial compared with the direct access afforded by a vector. That leads me to another consideration which is how and when the affiliates will be accessed. Most of the access to affiliates will be done by the program searching for a name or possibly a substring. In this case, each element will need to be visited anyway and so the number of times direct access to the affiliates is needed is assumed to be very low.

   Finally, using a linked list allows the container to grow only as much as is needed which helps to make efficient use of memory.

2. Search results from the database fall under one of two categories:

   (a) ID# Search - Because the binary search tree is ordered using the ID numbers and each contact has a unique ID, any ID# search will occur in log(n) time and return either a single Record, or nothing, if the ID doesn't exist in the database.

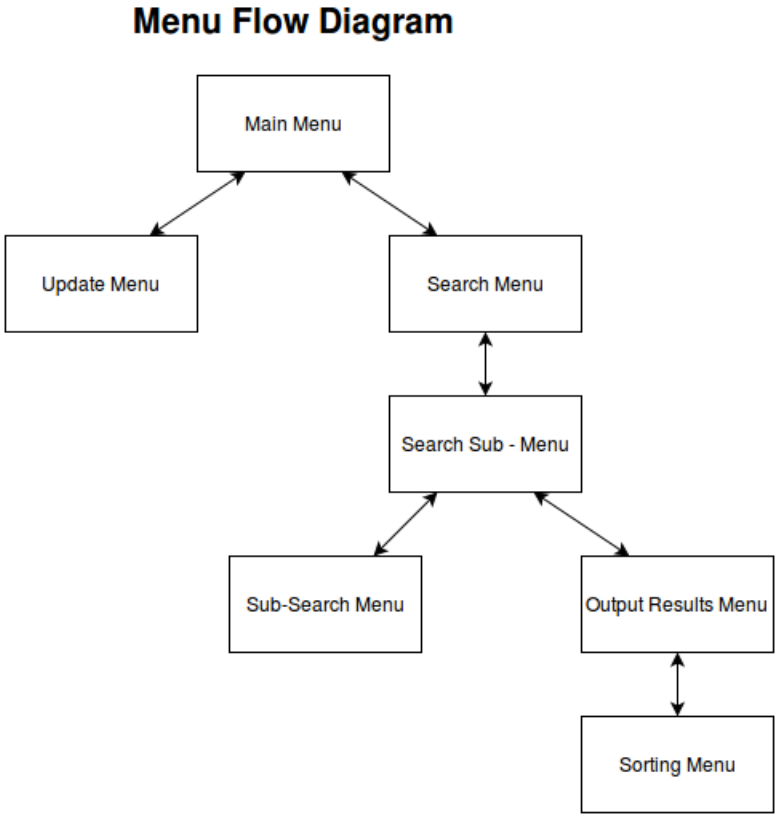
   (b) Other Fields - Data structures for other searches will by dynamically created and destroyed while the program is running. In my design, the user will not be limited to the number of searches they can perform on the database. Saving a

copy of every search result would create an unreasonably large use of memory while the program was running, and in extreme cases could deplete the available system memory and cause a crash. Instead, each instance of a search will only exist for as long as the user is still making queries on that search data.

The structure of each search result will once again be implemented as a linked list. For many of the same reasons as described above, a linked list once again maximizes our use of memory while reasonably managing the complexity of operating on the list. Any output of search results (to either a file or console) requires each item to be visited, so a linked list performs no worse than any other structure in this respect. Using the c++ STL list allows for easy checks for the number of elements in the list, and if sorted, accessing elements in alphabetical or reverse order.

User Interface Scheme

The program uses a menu-driven, text-based user interface scheme. The top level menu is the main menu, which then branches to sub-menus which allow the user to choose between different operations on the database. All menus require the user to input an integer to select which operation to perform. The majority of menus are governed by a switch() statement to execute the correct menu selection. Some menu options call other menus, some cause functions to execute, and some do both. The flow between menus is shown below.



Program Status

The program currently performs all required functions as I interpret them. The user can read in a .txt file containing database information and then perform searches and generate output based on a variety of criteria.

CSEgrid Status

The program successfully compiles and runs on the CSE grid using both databasesmall.txt and databaselarge.txt.