OPERATING SYSTEMS

Assignment 1

Pavlos Ntais

Contents

1 General Information

The project has been implemented in C. I have organized the files into multiple directories and files for readability purposes and to allow separate compilation. The functionalities described in the assignment, have been implemented. I have also thoroughly tested the program, and it works as intended.

2 Explanation of Directories & Files

- **modules/linear_hashing.c**: Implementation of the linear hashing data structure.
- **modules/list.c**: Implementation of the list data structure.
- **src/commands.c**: Implementation of the application's commands (1-9).
- **src/database.c**: Helper functions for communication with the database (essentially wrappers of the data structures) where all information about the votes is stored.
- **src/mvote.c**: Contains the main() function where command line arguments are received, the database is created, and command parsing is done to select the correct function from commands.c, which is then executed.
- **src/utilities.c**: General utility functions needed in the application, mainly string manipulation.
- **include**: In this directory, I keep the header files. Apart from pieces of code:
- **include/utilities.h**: Here, in addition to function prototypes, I have the custom_malloc and custom_calloc functions, which act as wrappers for malloc and calloc, respectively. The functions implement error handling in case of failure resulting in the program termination. I chose to place them here instead of utilities.c for potential speed gains due to function inlining.

3 Explanation of Data Structure Composition

3.1 List

3.1.1 Overview

The list is implemented in a generic way (using void*). I made this decision made because a two-dimensional list is required, thus having two different data types stored in the list node. The first (I) is Postal Codes, and the second (II) is Voters. The node with Postal Codes contains a pointer to the list of its voters. In my opinion, this design is "cleaner" compared to having two different types of lists. For this implementation, the following functions need to be provided as arguments:
(I) DestroyFunction: destroy_postcode_node (found in database.c - line 11).
(II) DestroyFunction: None (NULL) - the hash table handles the deallocation of candidates (and voters consequently).

3.1.2 Functions

a) **list_create**: O(1).
b) **list_push**: Insert at the start of the list - O(1).
c) **list_sort**: Merge sort - O(nlogn).

3.2 Linear Hashing

3.2.1 Overview

The structure is fully modular, allowing the user to choose the initial size of buckets, the number of elements per bucket, the hash function, and the function providing the new number of buckets during expansion.

To store the elements, an array of buckets is used, specifically a lists of buckets (to handle overflows). Two variables, **curr_capacity** and **max_capacity**, provide information about the size of this array. **curr_capacity** indicates the number of buckets currently in use, and **max_capacity** indicates the number of buckets allocated. This is because, during a split, the current number may exceed the initially allocated number. In such cases, realloc is performed for a new, larger array, allowing the user to decide its new size through the mentioned functions. However, if no expand function is given, the default function is used, which expands by 1. The risk with this function is cases of frequent realloc, if we are not careful, it may lead to O(n) complexity. Nevertheless, the positive aspect is that only the required number of buckets is used each time, saving memory. To avoid the frequent O(n) complexity, doubling the size is an option, but this results in potential wastage of space. For this reason, I leave it up to the user to decide.

The **insert** process is straightforward. The element undergoes hashing with the appropriate hash function. We check for the element's existence to avoid duplicates and then insert the element into the correct bucket. If we exceed the maximum lambda (LAMDA_SPLIT), a new bucket is created, and a split operation occurs between the element and the element at position 'p.' We then take each element of the 'p' bucket, rehash it using **h_1**, and check if a new round of splitting is needed. If so, we prepare the new **h** and **h_1** functions. Note that the overflow bucket always goes to the beginning of the list instead of the end (like in the given paper) because:
a) traversing the list to put it at the end is time-consuming.
b) we need an extra pointer to the last element of the list to prevent the mentioned said inefficiency.
c) practically, there is no difference.

3.2.2 Functions

a) **hash_create**: O(1).
b) **hash_search**: The number of elements per bucket (including overflow buckets) does not grow proportionally with the total number of elements, so we can claim it is O(1).
c) **hash_insert**: O(n) worst case & O(1) amortized complexity. The worst case arises due to realloc, as in the case where adjacent space for our node array is not found, the old nodes

need to be copied to a new memory space. The amortized complexity arises because even in the split, the elements in a bucket (as mentioned in b.) are constant.

4 Commands

For breaking down all the arguments of the commands, I use strtok and then appropriate checks. The only commands that need explanation, in my opinion, are as follows:

- **Command 4 (voters_file)**: In case the voter has already voted, I still print "Marked Voted," but I do not count them again in the database.
- **Command 8 (postcode_voters)**: Here, I preferred to perform merge sort on the list, on the spot, instead of sorted insertion (with a doubly linked list), before printing. I made this decision because a) I assume this function is rare (i.e., possibly only done at the end of the vote), b) to avoid the (admittedly) few extra iterations, and c) the memory needed for the extra pointer in each node.
- **Command 10 (print_db)**: Print the hash table (my addition, mainly for debugging purposes) with 'p'.

5 Command Line Arguments

The following arguments can be provided to the program:
-f file_name: Read the file of voters.
-b bucket_size: Number of elements each bucket can hold.
-m starting_size: The initial size of the HT.
-e expand_function: **1** for expansion by one, **2** for doubling.

6 Technical Details

Since I'm not entirely sure about the format of the file from which we read on input (-b), based on responses on Piazza, I assume that the first name is given first, followed by the last name. If you prefer the opposite, in the utilities.c file, you can uncomment line 159 and comment line 158.

7 Usage

A makefile is provided with the following parameters:
make - Compilation and linking of the program.
make run - Execution of the program (with default command line arguments).
make help - Execution of the program using valgrind.
make clean - Deletion of object files & the executable created during compilation.