**YELP Database as a Binary Search Tree Report**

There are two programs to implement a binary search tree in Yelp Database. Both of them are in the same structure: (Stage 2 adds an extra ‘2’ before .c)

| main.c

| input.c

| output.c

| construct.c

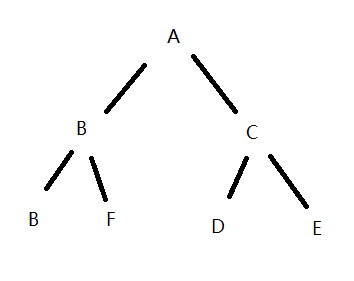
The main, input and output functions are identical except the header call, where the stage 1 called header.h and the stage 2 called header2.h. The input is also identical, with the format:

./yelp1 input\_file output\_file < keys\_file (use yelp2 in stage 2)

Or

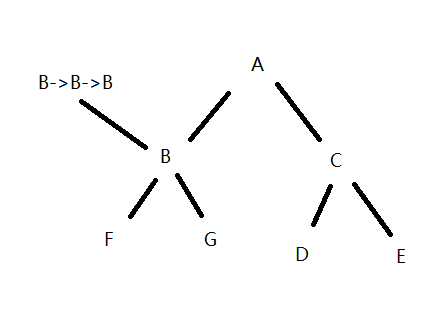
./yelp1 input\_file output\_file, then type keys (use yelp2 in stage 2)

Note that the input file has to be a standard csv file with only [name] and [data] per line. The name field have to <=64 characters and data field strict <=1465 characters. The algorithms, however, are different in processing duplicate records. Stage 1 inserts the duplicate records in the same dictionary (tree), which goes left after found a same key and looks for space in the left node.



Pic 1. Sample Structure of Stage 1

In Stage 2, the tree has one more node than in Stage 1, which is the head of a linked list, therefore when the program identifies a duplicate key, it is added into the linked list. The head of this linked list is node->head in the dictionary with the same key.



Pic 2. Sample Structure of Stage 2

In this report, I will analysis both algorithm in terms of space/time complexity.

The ideal complexity of Stage 1 should be:

O(n + log(n) + log(n) + 1) = O(n + n + n + 1) 🡪O(3n + 1)

(n for reading from file + n for inserting + n for searching + 1 for output file)

In the Stage 2, it should be:

O(n + log(n) + log(n) + 1) = O(n + 2(log(n) + m) + 1) 🡪 O(n + 2m + 2log(n) + 1)

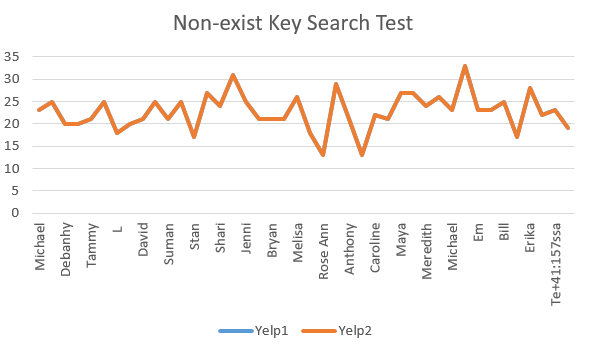
(n for reading from file + (log(n) + m) for inserting + (log(n) + m) for searching + 1 for output file)

Here n means number of non-duplicate keys and m means duplicate keys.

Testing File: yelp\_academic\_dataset\_business.csv, Number of Records: 77445

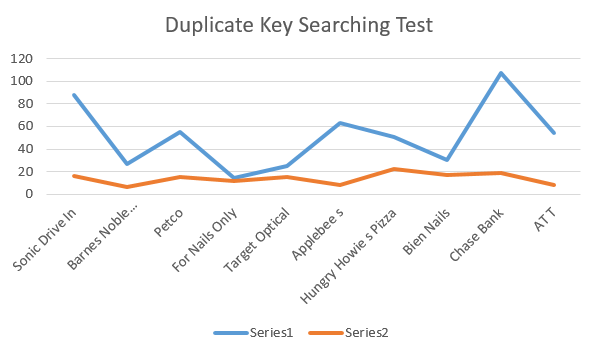
Graph 1. Random Search Keys Test

The first testing is that randomly takes 35 names in the database and 5 non-exist data name, run the program to see how many searching times would take. As the Graph 1 shows, the searching times in yelp1 generally takes more time to complete searching, although there are few exceptions.



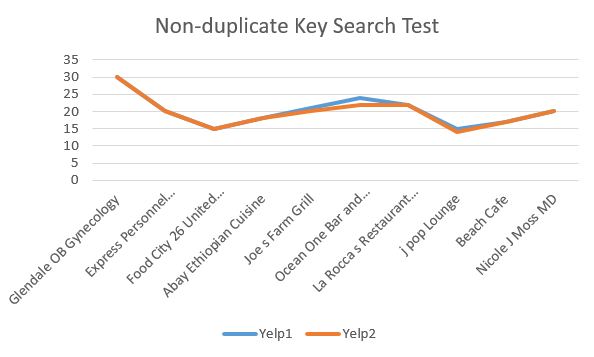
Graph 2. Non-exist Keys Searching Test

The second testing shows the searching efficiency on non-exist keys. Because 2 lines are overlapped, it is very clear from the graph that both of them take the same time to complete this kind of searching.



Graph 3. Duplicate Keys Searching Test, Series 1 = Yelp1, Series 2 = Yelp2

The third testing demonstrates the searching times taken by using duplicate keys. We can see from this graph that the duplicate key search in Stage 2 is much more effective than in Stage 1. This is because once the first key found, all other duplicate keys can print directly without more searching in Stage 2.



Graph 4. Non-duplicate Keys Searching Test

The last test is for non-duplicate keys, the Stage 2 is slightly effective than Stage 1, might because there are duplicates keys in the tree occupying the space.

Overall, the tests are support the theory, because m (the number of duplicate keys) is much less than n (the number of records), that is to say, O(3n + 1) takes more time than O(n + 2m + 2log(n) + 1), then we can conclude that the Stage 2 method is effective than Stage 1 when duplicate keys exist.