COMP20003: Algorithms and Data Structures
Week 11 Workshop Exercise: Dictionaries: Hash table

Objectives

- To understand the dictionary Abstract Data Type.

- To work with several different dictionary implementations.

- To implement a simple hash table and use it to complete a small program.

The Programs

For this workshop you will write a program to implement a simple dictionary ADT for string values, using a hash table. To help you get started, we provide a complete program which implements this ADT using association lists.

Each program reads a file of key-value string pairs and inserts them into a dictionary. The programs take one command line argument, the name of the data file. Once the data is loaded, the user may issue lookup requests by specifying search keys visa stdin. The program then searches for the value associated with the key. If the key-value pair is in the dictionary then the value is printed out, otherwise the user is informed that the search failed. In either case, the number of lookups required for the search is printed.

The program assoc_list implements the dictionary as an association list, a linked list of key-value pairs. **This version of the program is already complete**, you should compile it and try it out on some of the provided data files.

Task 1: Dictionary operations (Required)

Your task is to complete the hash table implementation of the program. The basic data structures required are provided in the hash_table.c. The three functions that you need to implement are hash_table_init, hash_table_insert, and hash_table_lookup. Their interfaces are further described in the comments in the source code.

Every hashing scheme has two aspects: the hash function and the collision resolution method. The hash function specifies which table entry to use for a new record, while the collision resolution method tells us what to do when different keys has to the same value. Please use the hash function for strings -- take ord(c) to be the ASCII value for the characters:

```
h <- 0;
for i <- 0 to s - 1
```

```
do
  h <- (h * C + ord(ci)) mod m,
```

where C is a constant larger than any ord(ci)

For the collision resolution method, we will use the simplest of the open addressing methods, linear probing. Linear probing resolves collisions by checking the cell following the one where the collision occurs. If a collision occurs in that cell, we check the next cell, and so on. If during this process we reach the end of the table, then we wrap around to the front of the table. This wrapping behaviour can be implemented in C by referring to table entries modulo the table size.

Implement the three functions using the specified hash function and the linear probing collision resolution method. Test your function on some small data files; in the next section we will examine one way of expanding the table for larger data files.

Task 2: Rehashing (Optional)

Hash tables must be able to grow as the number of records they contain grows. This is not a problem for separate chaining. However, it does present a problem for open addressing. In this case, we must rehash. rehashing is the operation of resizing a hash table; it can be summarized as follows:

- choose a new table size. For this implementation we are always going to choose our table sizes to be prime numbers. (can you suggest why ?) Each time the table needs to be resized, we will double the table's size and then round up to the nearest prime number.

- Allocate memory for the new table. Traverse the old table and insert each entry into the new table. This will require computing a new hash value for the entry, hence the name: rehashing.

- Free memory used by the old table. After all of the entried are copied across to the new table, the old table can be freed.

Modify the hash_table_insert function so that it rehashes the table when the load factor exceeds 85%.

Task 3: Experimental Work (also optional)
We have provided a number of sample data files of varying sizes. For each of these files, you should compare how the association list dictionary implementation compares to the hashtable implementation. What are the theoretical efficiencies for insertion and lookup for both dictionary implementations? What are these efficiencies in practice?
What happens if you raise / lower the load factor limit for rehashing ?