1. Week 10/11 : Hashing

1.1 Double Hashing

Here we use double hashing, a technique for resolving collisions in a hash table.

Exercise 1.1 Use double hashing to create a spelling checker, which reads in a dictionary file from `argv[1]`, and stores the words.

Make sure the program:

- Use double hashing to achieve this.
- Makes no assumptions about the maximum size of the dictionary files. Choose an initial (prime) array size, created via malloc(). If this gets more than 60% full, creates a new array, roughly twice the size (but still prime). Rehash all the words into this new array from the old one. This may need to be done many times as more and more words are added.
- Uses a hash, and double hash, function of your choosing.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up. Assuming the program works correctly, this number is the only output required from the program.

■

60%

1.2 Separate Chaining

Separate chaining deals with collisions by forming (hopefully small) linked lists out from a base array.

Exercise 1.2 Adapt the previous program so that:

- A linked-list style approach is used.
- No assumptions about the maximum size of the dictionary file is made.
- The same hash, and double hash, functions are used as before.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up, on average. Assuming the program works correctly, this number is the only output required from the program.

■

30%

1.3 Extension

Exercise 1.3 Attempt an extension of your choosing. This could be to do with comparing with another hashing technique, using better hash functions, comparing with another search technique, using SDL to display the hashmap's occupancy, or something else. ■

10%

Submission

Please create a directory structure, so that I can easily find the different exercises. The p1 sub-directory will contain the files required the build the first part. These will typically be `spll.c`, `dhash.c` and `dhash.h`. If you have abstracted the hash data type, in part two, `spll.c` will change very little, but new hashing functionality will be accessed via `shash.c` and `shash.h`.

```
       Top Directory
        /      |     \
       /       |      \
      /        |       \
     /         |        \
    p1         p2       p3
    |          |         |
 spll.c    spll.c    extnd.c
 dhash.c   shash.c    ...
 dhash.h   shash.h    ...
                     extension.txt
```

It will probably be easier to simply bundle all of these up as a single `.zip` submission.

Simple Hash Functions

```
/* The 1st 2 & last 2 characters in the words are used.*/
unsigned int hash_ends(char *str)
{
   unsigned int c1, c2, cn1, cn2;
   unsigned int hash;
   unsigned int n;

   n = (unsigned int)strlen(str);
   c1 = str[0];
   c2 = str[1];
   cn1 = str[n-1];
   cn2 = str[n-2];
   hash = (c1*c2 + cn1*cn2)*n;

   return hash; /* You'll still need to mod it */
}

/* Every character in the word is used
   #define PRIME 31 */
unsigned int hash_all(char *str)
{
   int i;
   int n = strlen(str);
   unsigned int hash = 0;

   for(i=0; i<n; i++){
      hash = str[i] + PRIME*hash;
   }

   return hash; /* You'll still need to mod it */
}
```