CSE312-L: DSA
LNMIIT, Rupa-ki-Nangal, Jaipur 302031
Training Set 08

Searching of a value in a relatively static data set is a problem that need programming solution frequently. Hashing promises a very efficient solution that has time complexity approaching O(1).

To practice and learn the lessons related to hashing, we have derived two data sets of words. First set of words called collection has words that we can add to a hash table. This is about 4250 words. Other collection of words is smaller that 2000 words called keys. The words in file `keys` are to be searched in the collection file stored in the hash table.

In this training set we will practice a combination of activities and collect performance data under the various scenarios. The tasks would include the following:

1. Use division method (See Reema Thareja: Section 15.4.1) for deriving hash value to store data.
2. Use multiplicative method (Reema Thareja: Section 15.4.2) for deriving hash value to store data.
3. Resolve collisions by linear probing
4. Resolve collision by double hash probing
5. Resolve collision by quadratic probing
6. Collect collision and probing count data under various operational parameters.

To support students, a partial working program is provided to the trainees in this document. The students are urged to prepare for the lab work before arriving at their DSA lab sessions.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <limits.h>
#include <ctype.h>
#include <math.h>

#define TBLSZ 6000
int usedSize, // Part of the table that will be used
      divider2ndHashing; // Step size for double hashing
double goldenNumber;

char *table[TBLSZ];

int initialise(int tableSize) {
      /* Find two suitable prime values for
            two division hashings funtions.
      */
      /* usedSize should be a prime that is nearly
            as big as TBLSZ
            This sets the part of the table that is actually
            used to store pointers to strings
      */
      assert(TBLSZ > 1000);
      goldenNumber = (sqrt(5.0)+1)/2;
      int testing, prime, i;
      if (TBLSZ%2 == 0)
```

```
            usedSize = TBLSZ-1;
      else
            usedSize = TBLSZ;


      /* Begin with usedSize as big as TBLSZ and
            progressively lower it till it is a prime
      */
      while (usedSize > 2) { // Keep trying
            i = 3;
            while (i*i <= usedSize) {
                  if (usedSize%i == 0) {
                        /* No good */
                        usedSize -= 2;
                        break;
                  }
                  i += 2;
            }
            if (i*i > usedSize) {
                  break;
            }
      }
      /* Pointers to data strings will be in indices 0 to
            usedSize.

            Now find a prime to do probing over the table */
      divider2ndHashing = 2;
      while (divider2ndHashing*divider2ndHashing < usedSize)
            divider2ndHashing *= 2;
      /* Similar to the previous search for a prime */
      while (divider2ndHashing > 2) { // Keep trying
            i = 3;
            while (i*i <= divider2ndHashing) {
                  if (divider2ndHashing%i == 0) {
                        /* No good */
                        divider2ndHashing -= 2;
                        break;
                  }
                  i += 2;
            }
            if (i*i > divider2ndHashing) {
                  break;
            }
      }
}

int goldenMultiplicative(int key) {
      /* Returns the primary location for the data
            using multiplication by golden number. See
            Reema Thareja Edn 2 page 468
      */

      double multiplier = goldenNumber - 1;
      double frac = (key*multiplier);
      int index = frac;
      frac = frac-index;
      index = frac*TBLSZ;
      return index;
}

int makeNumber(char *str) {
      /* Construct a number from teh string */
      assert(str != NULL);
      int num = 0;
      int el = strlen(str);
      int i = el - 1; // Next letter to process
      if (el%2 == 0) el--; // Make it odd
      assert(el < 20 && el > 0); // Too long a word

      /* Computer specific limit on while */
      while(num < INT_MAX/26/26) {
            num = num*26+toupper(str[i])-toupper('A');
            i = (i+2)%el; // Choose alternate letters
      }
      num = 26*num+el; // Include word length
      return num;
}

int divisionHash(int key, int probe) {
      int step = key%divider2ndHashing;

      /* Linear hashing */
      //return key%usedSize + probe;

      /* Double hashing */
      return key%usedSize + probe*step;
}

int main(void) {
      FILE *collection = fopen("collection", "r");
      assert(collection != NULL);

      char strg[100];
      char * dataP;
      int loc, collisions=0, noCollisions=0;
      int el, reTry, rehash, filled;
      initialise(TBLSZ);

      while (fgets(strg,90,collection)) {
            el= strlen(strg)-1;
            strg[el] = '\0';
            dataP = malloc(strlen(strg)+1);
                  assert(dataP != NULL);
                  strcpy(dataP, strg);

            reTry = 0;
            while (rehash < 100) {
                  loc = divisionHash(makeNumber(strg), reTry);
                  if (table[loc] != NULL) {
                        reTry++;
                        collisions++;
                        continue;
                  } else {
                        filled++;
                        table[loc] = dataP;
                        break;
                    }
                }
            }
        printf("Total collisions %d\n", collisions);
        printf("Unused %d\n", TBLSZ-filled);
        printf("Filled %d\n", filled);
        return 0;
}
```

Students will notice that the code constructs a hash table, by inserting the words in file collection. It uses division method for finding hash value. Since the words are not directly suitable for arithmetic computations, a function `makeNumber()` is defined to compute a numerical signature of the word. Study the program and understand how this program loads the words in the hash table.

Students will also notice size of the hash table is specified by constant TBLSZ. However, the actual parameters used for determining the hash and probing values are derived from this value by finding suitable prime numbers.

Training Set 08: Task 01

Use different TBLSZ values and plot the relationship between the TBLSZ and collisions. Next add code to function `main()` to read keys and determine if each key read from file keys is present in the hash table or not. Keep a record of the total number of table entries accessed in testing the memberships of keys in file `keys`.

To test the presence of a key in the hash table, the program needs to perform the following activities:

1. Use `makeNumer()` to derive a numerical signature of the key.
2. Probe the hash table, perhaps repetitively, to determine if the key word is present in the table. This will require the students to use function `strcmp()`.
3. Student must set a suitable criterion to determine if the search has failed to find the key in the hash table.

Training Set 08: Task 02

In this task the students must practice the use of linear and quadratic probe to handle collision situations. You will need to use a suitable triplet of primes numbers to construct useable hash table entries, and coefficients of first and second power of probe number.

In each of these examples spend some time to collect performance metrics to see relations between TBLSZ, number of collisions and probing.

Training Set 08: Task 03

Hashing function `int goldenMultiplicative(int key)` based on multiplicative hashing idea has been provided in the code. Test the performance characteristics of this function vis-à-vis those reported previously in Task 01.