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

Searching a value in a relatively stable and infrequently changing data set is a common programming need. Hashing provides an efficient solution with time complexity approaching O(1).

To practice and learn the lessons related to hashing as technique for storing and retrieving data, we have created two files of words. First file of words, called `collection`, has words that we will add to a hash table. This collection has about 4250 words. Other collection of words has about 2000 words; the file is called `keys`. The words in file `keys` are to be searched in the collection put together in the hash table.

A brief description of hashing techniques is presented in this paragraph. To apply hash functions to creates signatures of the words we need a way to convert words into numerical values. Function `makeNumber()` performs this conversion in the program given in this document. The signature for a word is not necessarily different from the signatures of the other words in the collection. However, for a well-designed hash function not many signatures should collide. Therefore, to test if a key is in the collection, an array can gather the signatures of the words in the collection. The key is also hashed, and the hash value used to index into the array. If the indexed entry is empty, then key is not in the collection. If the location in the array is occupied, there is strong chance that the key is in the collection. However, the confirmation would require the actual comparison of the key with the few words that have the matching signature value. The hashing algorithm has avoided the need to search over every word in the collection.

Unlike the other data structures discussed in DSA, hashing technique is a probabilistic algorithm. Good performance requires hashing function that promises few collisions and is quickly computed.

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

1. Use and practice division method (See Reema Thareja: Section 15.4.1) for deriving hash value to store data.
2. Use and practice 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 performance data under various operational conditions listed as activities 1 through 5.

To assist students, a practically complete and working program is provided to the trainees in this document. The students are once again urged to manage their lessons and prepare for the lab work before their scheduled DSA lab session.

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

#define TBLSZ 6000

// Hashing choice related constants
#define DIVISION '%'
#define MULTIPLICATIVE '*'
#define LINEAR 1
// Step size for linear probing. Use only primes
#define STEP 1
#define DOUBLE_HASHING 2
#define QUADRATIC_REHASH 4

// To be set by the students as needed
#define PRIMARY_HASHING_METHOD MULTIPLICATIVE
#define PROBING_METHOD DOUBLE_HASHING

// The following are primes < TBLSZ
int usedSize, // Part of the table that will be used
     probeStepSz; // 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);
     assert(PROBING_METHOD >=0);
     assert(PROBING_METHOD < 3);
     assert(PRIMARY_HASHING_METHOD == MULTIPLICATIVE ||
          PRIMARY_HASHING_METHOD == DIVISION);
     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

	   Pointers to words/strings will be stored
	   in index range 0 to usedSize-1 only.
	*/
	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) {
			assert(usedSize > 1000);
			assert(usedSize < TBLSZ);
			break;
		}
	}

	/* The following code finds a prime to use to do
	probes over the table when there is a collision */
	probeStepSz = 2;
	while (probeStepSz*probeStepSz < usedSize)
		probeStepSz *= 2;
	/* Similar to the previous search for a prime */
	while (probeStepSz > 2) { // Keep trying
		i = 3;
		while (i*i <= probeStepSz) {
			if (probeStepSz%i == 0) {
				/* No good */
				probeStepSz -= 2;
				break;
			}
			i += 2;
		}
		if (i*i > probeStepSz) {
			assert(probeStepSz > 1);
			assert(probeStepSz < usedSize);
			break;
		}
	}
}

int goldenMultiplicative(int key, int size) {

	/* Returns the primary location for the data
```

```
            in the hash table.
            The location is computed using
            multiplication by golden number.
            Reference: Reema Thareja Edn 2 page 468
      */

      double multiplier = goldenNumber - 1;
      double frac = (key*multiplier);
      int index = frac;
      frac = frac-index; // Get fractional part
      index = frac*size;
      assert(index >= 0);
      assert(index < size);
      return index;
}

int makeNumber(char *str) {
      /* ******************************************
      Students to use code - implementation not important

      Construct a number from the string
      The function may return the same number for
      several different strings

      However, it will alway return the same number
      for a 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 indexAdvisedByHash(int key, int probe) {
      /* What arrangement has been chosen */
      int step, goldIndx;
```

```
	switch (PRIMARY_HASHING_METHOD) {
		case DIVISION:
			switch (PROBING_METHOD) {
				case LINEAR:
					return (key + STEP*probe)%usedSize;
				case DOUBLE_HASHING:
					step = key%probeStepSz;
					// step != 0 is necessary
					if (step < 2)
						step = 17;
					return (key + probe*step)%usedSize;
				default:
					assert(0); // Unexpected
					return 0;
			}
		case MULTIPLICATIVE:
			goldIndx = goldenMultiplicative(key, usedSize);
			switch (PROBING_METHOD) {
				case LINEAR:
					return (goldIndx + STEP*probe)%usedSize;
				case DOUBLE_HASHING:
					step = goldenMultiplicative(key,
									probeStepSz);
					// step != 0 is necessary
					if (step < 2)
						step = 1009;
					return (goldIndx + probe*step)%usedSize;
				default:
					assert(0); // Not expected
					return 0;
			}
		default:
			assert(0); // ERROR
	}
}


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);
```

```
	/* Read word from FILE collection and
		store pointer to word in array table[].

		Array index to be determined by a
		hashing method.
	*/
	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 is primary hashing attempt.
			reTry > 0 are attempted alternate locations.
		*/
		reTry = 0;
		while (rehash < 100) {
			loc = indexAdvisedByHash(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;
}
```

The students will notice that the code constructs a hash table, by inserting the words from file `collection`. The primary hashing method and probing method is set by setting constants at the start of the program.

Since words are not directly suitable for arithmetic computations, a function `makeNumber`() is defined to compute a numerical proxy for the word. The code in the function is not educative and best ignored.

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 a suitable prime number within the available space in the table.

Training Set 08: Task 01

The program provided to the students constructs a hash table from a set collection of words.

Use several different options and combinations of TBLSZ, PRIMARY_HASHING_METHOD, PROBING_METHOD and STEP values to record performance metrics. Is there a combination that may be stamped for very poor or superior performance?

The provided program does not implement quadratic hashing. This will be assigned as a task later in this training set.

Training Set 08: Task 02

Implement a quadratic probe option to handle collisions. To construct this option, you will need to determine two suitable prime numbers to be the coefficients in the quadratic expression.

For simplicity, we may augment the expression used in linear probe by adding a quadratic term: `probeStepSz*probe*probe`.

Record program performances under different settings of customisable parameters (listed above in Task 01) to compare this method against those recorded in the previous task.

Training Set 08: Task 03

Using the code and functions provided in the program (and writing fresh code as needed), bring in an improvement to the program. File `keys` contains keywords that need to be searched. Write code to perform this task taking account of the following recommendations:

1. Use `makeNumer()` to derive a numerical signature of the key.
2. Probe the hash table, repetitively as needed, to determine if the keyword is present in the table. This will require you to use function `strcmp()` to confirm the presence of the key in the table.
3. Student must set a suitable criterion to terminate the excessively long probing process if it fails to find the keyword in the table. A premature termination may miss to uncover keys that are in the collection. This error is called FALSE NEGATIVE. On the other hand, a reluctance to terminate the search may lead to unending run of the program.

Record program performance under various settings of the customisable values as in Task 01. Have you noticed an instance of false negative for your program?

Vishu Malhotra
03 May 2020
The LNMIIT, Rupa ki Nangal
Jaipur, Rajasthan 302031