Hash Table

Jacob Tucker

CMPS 390

4/29/23

C hash-table.c ×

hash-table > C hash-table.c > insertName(char **, int, char *)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define HASH1SIZE 200
#define HASH2SIZE 400
#define HASH3SIZE 700

int genHashTableIndex(int nameHash, int tableSize) {
  int q = 1;

  if (tableSize == HASH1SIZE) { q = 460; }
  else if (tableSize == HASH2SIZE) { q = 220; }
  else if (tableSize == HASH3SIZE) { q = 125; }
  else { printf("Invalid table size: %d", tableSize); }

  int hashIndex = ((int)((nameHash - 351) / q)) * 5;

  return hashIndex;
}

int insertName(char** table, int index, char* name) {
  int collisionFlag = 0;
  while (table[index] != NULL) {
    collisionFlag = 1;
    index++;
  }
  table[index] = (char*)malloc(strlen(name) + 1);
  strcpy(table[index], name);

  return collisionFlag;
}

void showTable(char** table, int tableSize) {
  for(int j = 0; j != tableSize; ++j) {
    char* name = table[j];
    if (name != NULL) { printf("%d: %s\n", j, table[j]); }
  }
  printf("\n");
}

void freeTable(char** table, int tableSize) {
  for (int j = 0; j != tableSize; ++j) {
    if (table[j] != NULL) { free(table[j]); }
  }
}
```

First, I define constants for each hash table to reuse their values throughout the code. I create 4 functions to make the program more modular.

The first function is *genHashTableIndex* which takes in a *nameHash* and *tableSize* and generates a hash index based on that information. It checks the table size argument against the constant hash table sizes in order to decide how many times it should divide the *nameHash*. I first subtract the minimum *nameHash* value computed from the input for this assignment, which is 351. This offsets all of the indices such that they will begin at 0. I then divide by the variable that changes based on the table size and ensures that the largest index, when multiplied by 5, will still be less than the (table size – 5). I then multiply the values by 5 so that they will all be multiples of 5, creating a gap of 5 between each index for collisions.

The *insertName* function takes in a pointer to the hash table, which itself is an array of strings, the index to insert the name at, and the name as arguments. It returns an integer representing whether a collision was detected or not. To detect this, since the hash tables are initialized to have NULL values at all indices, it simply looks to see if the given index is NULL or not. If not, then a name has already been inserted there so it must check the next index. Once it finds a NULL value at an index, it allocates memory for the name and stores it there.

The *showTable* function prints the index and name for all indices with a value other than NULL in the given table.

The *freeTable* function iterates through the given table freeing any memory that has been allocated by checking for NULL values.

```c
int main() {
    char* hashTable1[HASH1SIZE] = {NULL};
    char* hashTable2[HASH2SIZE] = {NULL};
    char* hashTable3[HASH3SIZE] = {NULL};

    int collision1Count = 0;
    int collision2Count = 0;
    int collision3Count = 0;

    char fileName[] = "input.txt";
    char line[30];
    FILE* file;
    file = fopen(fileName, "r");
    while (fgets(line, sizeof(line), file)) {
        if (line[strlen(line) - 1] == '\n') { line[strlen(line) - 1] = '\0'; }

        int nameHash = (line[0] - 'a') * (int)pow(26, 2) + (line[1] - 'a') * 26 + (line[2] - 'a');

        int index1 = genHashTableIndex(nameHash, HASH1SIZE);
        collision1Count += insertName(hashTable1, index1, line);

        int index2 = genHashTableIndex(nameHash, HASH2SIZE);
        collision2Count += insertName(hashTable2, index2, line);

        int index3 = genHashTableIndex(nameHash, HASH3SIZE);
        collision3Count += insertName(hashTable3, index3, line);
    }
    fclose(file);

    showTable(hashTable1, HASH1SIZE);
    showTable(hashTable2, HASH2SIZE);
    showTable(hashTable3, HASH3SIZE);
    printf("Collision Count for Table 1: %d\n", collision1Count);
    printf("Collision Count for Table 2: %d\n", collision2Count);
    printf("Collision Count for Table 3: %d\n", collision3Count);
    freeTable(hashTable1, HASH1SIZE);
    freeTable(hashTable2, HASH2SIZE);
    freeTable(hashTable3, HASH3SIZE);
}
```

In the *main* function I initialize the hash tables values to all NULL values and the collision counts for each to 0. I then read the input file line by line, calculating the hash for each name and then inserting it into each table as they are read. The outputs are then printed and the tables are freed. Here are the collision counts:

```
Collision Count for Table 1: 46
Collision Count for Table 2: 33
Collision Count for Table 3: 24
```

Here is the output for *showTable* being called with the 3 tables (the outputs were copied form the terminal into Excel so that they can be more easily compared side-by-side and the screenshots will take less room in this report):

	A	B	C	D	E
1	**table 1**		**table 2**		**table 3**
2	0: barrack		0: apollo		0: apollo
3	1: apollo		1: anny		1: anny
4	2: anny		2: avery		5: avery
5	3: avery		5: barrack		10: barrack
6	5: bob		10: bill		20: bill
7	6: brian		11: billyjoe		21: billyjoe
8	7: bullwinkle		15: bob		25: bob
9	8: bill		16: brian		30: brian
10	9: billyjoe		17: bullwinkle		31: bullwinkle
11	10: carl		20: carl		40: carl
12	11: charles		25: charles		45: charles
13	12: chuck		26: chuck		46: chuck
14	13: clarence		27: clarence		50: clarence
15	15: dale		30: cris		55: cris
16	16: dan		35: dale		65: dale
17	17: cris		36: dan		66: dan
18	18: dewy		40: dianna		70: dewy
19	20: dianna		41: dewy		75: dianna
20	21: donna		45: donna		80: donna
21	22: dudz		50: dudz		85: dudz
22	25: ellis		60: ellis		105: ellis
23	26: ellie		61: ellie		106: ellie
24	30: eric		62: eric		110: eric
25	35: fred		75: fred		135: fred
26	36: francis		76: francis		136: francis
27	40: george		85: george		150: george
28	41: gertrude		86: gertrude		151: gertrude
29	42: gemini		87: gemini		152: gemini
30	43: ghassan		88: ghassan		155: ghassan
31	45: harry		90: greg		165: greg
32	46: greg		95: harry		175: harry
33	47: halley		96: halley		176: halley
34	50: howard		105: howard		190: howard
35	51: huey		106: hongkongfooey		191: hongkongfooey
36	52: hongkongfooey		110: huey		195: huey
37	60: jerry		125: issaac		220: issaac
38	61: issaac		130: jerry		230: jerry
39	65: joe		135: joe		240: joe

	A	B	C	D	E
40	66: karl		136: johnson		241: johnson
41	67: karla		140: judy		250: judy
42	68: johnson		141: junkun		251: junkun
43	69: judy		145: karl		255: karl
44	70: kerry		146: karla		256: karla
45	71: kim		147: kerry		260: kerry
46	72: junkun		150: kim		265: kim
47	75: larry		160: larry		280: larry
48	76: lala		161: lala		281: lala
49	80: mary		175: mary		310: mary
50	81: marvin		176: meriam		311: marvin
51	82: matt		177: marvin		312: matt
52	83: max		178: matt		313: max
53	85: meriam		179: max		315: meriam
54	86: mitzee		180: mitzee		316: mitzee
55	105: pam		220: pam		390: pam
56	106: paul		221: paul		391: paul
57	107: peter		225: peter		395: peter
58	120: raymond		250: raymond		445: raymond
59	125: ross		260: ross		460: ross
60	126: robert		261: robert		461: robert
61	127: roy		262: roy		462: roy
62	128: rocky		263: rocky		463: rocky
63	129: sammy		265: sammy		470: sammy
64	130: stewart		275: stewart		490: stewart
65	135: tom		285: thomas		500: tena
66	136: thomas		286: theresa		505: thomas
67	137: theresa		287: tena		506: theresa
68	138: tena		290: tom		510: tom
69	140: twirly		295: twirly		520: twirly
70	145: ulyssess		305: ulyssess		535: ulyssess
71	155: webster		330: webster		585: webster
72	175: zack		375: zack		660: zack
73	180: zeus		376: zeus		665: zeus
74	181: ziggy		380: ziggy		670: ziggy

From the collision data we can see that increasing the table size decreases the number of collisions. From the side-by-side comparison, we can also see that increasing the table size improves the ordering of the table so that it is closer to being in alphabetical order. However, the trade off is that increasing the table size is less memory efficient. It could be better to focus on improving the hashing function such that it distributes the data more efficiently, and having the hash table dynamically resize based on collision count.