|  |  |
| --- | --- |
| **CSCI 2100E Spring 2020** | Student Name: \_Tam Yi Ki\_\_ Student Id: \_1155126127\_\_\_\_\_ |

# Lab Exercise #6

*Lab exercises are always due 2**weeks after day of the lab.*

*Please fill in the lab sheet and submit the completed Word doc file to blackboard.* *Places you need to fill in or work on are marked in red.*

## Problem 1

Below you will find an example implementation of a hash table in C. The collision function is currently very lousy, and the insertion function will refuse to insert an entry if collision happens! Please run the program once and see the result.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define HASHTABLE\_SIZE 10  typedef struct {  int key; // in this example, key is some long id  char\* value; // value is a string  } KeyValuePair;  void printHashtable(KeyValuePair\* table[]) {  // print out the hash table so that you will see if you have done  // things correctly!    int i;  for (i=0;i<HASHTABLE\_SIZE;i++) {  if (table[i] == NULL) {  printf("%2d: -\n",i);  } else {  printf("%2d: <%d,%s>\n",i,table[i]->key,table[i]->value);  }  }  }  int hashFunc(int key) {  // turns the key into hash and returns it  return key / 1000 % HASHTABLE\_SIZE; // a very bad hash function  }  void insert(KeyValuePair\* table[], int key, char value[]) {  // insert a key and value  printf("inserting %d,%s to the table...\n",key, value);  int hash = hashFunc(key);  int index = hash;  if (table[index] != NULL) {  printf("collision! I refuse to do anything!\n");  return;  } else {  // add the new key-value pair to the correct position  printf("Entry inserted at position %d\n",index);  KeyValuePair\* newEntry = (KeyValuePair\*) malloc(sizeof(KeyValuePair));  newEntry->key = key;  newEntry->value = value;  table[index] = newEntry;  }  }  char\* lookup(KeyValuePair \*table[], int key) {  // look up a key and return the value  int hash = hashFunc(key);  int index = hash;  if (table[index] != NULL) {  if (table[index]->key == key)  return table[index]->value;  else  // collision happened? should we do something?  return NULL;  } else {  return NULL;  }  }  int main()  {    KeyValuePair\* hashtable[HASHTABLE\_SIZE] = {NULL};  insert(hashtable,1450017, "Ted");  insert(hashtable,1450345, "Jerry");  insert(hashtable,1450191, "Bill");  insert(hashtable,1450677, "Perry");  insert(hashtable,1450922, "Claire");  insert(hashtable,1450957, "Arthur");  printHashtable(hashtable);  printf("Lookup %d - result: %s\n", 1450017, lookup(hashtable,1450017));  printf("Lookup %d - result: %s\n", 9999999, lookup(hashtable,9999999));  printf("Lookup %d - result: %s\n", 1450677, lookup(hashtable,1450677));  printf("Lookup %d - result: %s\n", 1450957, lookup(hashtable,1450957));  return 0;  } |

Let's warm up by changing the hash function to a less lousy one. According to lecture notes, the following function would be a universal hashing function:

h(k) = (5 \* k % 11) % m

where m is the size of the hash table. Please show your updated "int hashFunc(int key)" below:

|  |
| --- |
| int hashFunc(int key) {  // turns the key into hash and returns it  return (5 \* key % 11) % HASHTABLE\_SIZE; // change to universal hashing function  } |

If you have updated the function correctly, the table should look like this at the end of the program (the last two lookup fails because we haven't handled collisions yet):

|  |
| --- |
| 0: <1450922,Claire>  1: -  2: -  3: -  4: -  5: -  6: -  7: <1450017,Ted>  8: <1450345,Jerry>  9: -  Lookup 1450017 - result: Ted  Lookup 9999999 - result: (null)  Lookup 1450677 - result: (null)  Lookup 1450957 - result: (null) |

## Problem 2

We cannot allow collisions unhandled, right? Let's use **linear probing** to handle the collision. According to the lecture notes, in linear probing, if collision happens (the slot is already occupied by someone else), we will check for next slots one by one until an empty slot is found.

**Note:** If you probe beyond the maximum index of the hash table, go back to 0 and continue the probing.

Let's add linear probing to **insert** and **lookup** functions in the program in Problem 1. Please paste your updated functions below:

|  |
| --- |
| void insert(KeyValuePair\* table[], int key, char value[]) {  // insert a key and value  printf("inserting %d,%s to the table...\n",key, value);  int hash = hashFunc(key);  int index = hash;  if (table[index] != NULL) {  int FULL = 1;  for(int i = index + 1; i < (index + HASHTABLE\_SIZE); i++) { //index + 10 means it can move 10 times  int new\_index = i % HASHTABLE\_SIZE;  if(table[new\_index] == NULL) {  // add the new key-value pair to the correct position  printf("Entry inserted at position %d\n", new\_index);  KeyValuePair\* newEntry = (KeyValuePair\*) malloc(sizeof(KeyValuePair));  newEntry->key = key;  newEntry->value = value;  table[new\_index] = newEntry;  FULL = 0;  return;  }  } //check the hole table one time  printf("The table is full!\n");  } else {  // add the new key-value pair to the correct position  printf("Entry inserted at position %d\n",index);  KeyValuePair\* newEntry = (KeyValuePair\*) malloc(sizeof(KeyValuePair));  newEntry->key = key;  newEntry->value = value;  table[index] = newEntry;  }  }  char\* lookup(KeyValuePair \*table[], int key) {  // look up a key and return the value  int hash = hashFunc(key);  int index = hash;  //printf("index = %d\n", index);  if (table[index] != NULL) {  if (table[index]->key == key)  return table[index]->value;  else {  for(int i = index + 1; i < (index + HASHTABLE\_SIZE); i++) {  int new\_index = i % HASHTABLE\_SIZE;  if(table[new\_index] == NULL)  return NULL;  if(table[new\_index]->key == key)  return table[new\_index]->value;  }  printf("No this key! Please check!!\n");  return NULL;  }  } else {  return NULL;  }  } |

If you have updated the functions properly, the table should look like this at the output:

|  |
| --- |
| 0: <1450677,Perry>   1: <1450922,Claire>   2: <1450957,Arthur>   3: -   4: -   5: -   6: -   7: <1450017,Ted>   8: <1450345,Jerry>   9: <1450191,Bill>  Lookup 1450017 - result: Ted  Lookup 9999999 - result: (null)  Lookup 1450677 - result: Perry  Lookup 1450957 - result: Arthur |

## Problem 3

Unfortunately, as you see, there is **primary clustering** happening in our table. When primary clustering happens, the entries are lumped together and we need to go through a lot of probing to search for a key.

Let's use **quadratic probing** instead. With quadratic probing, instead of probing one by one, we will look at the +1 slot, +4 slot and +9 slot until a proper slot is found.

*e.g. Suppose the hash is 2 and there is collision; we will first look at 2+1=3, then 2+4=6, then 2+9 = 11 % 10 = 1 and on and on*

Please update again the **insert** and **lookup** functions in the program in Problem 1. Paste your updated functions below:

|  |
| --- |
| void insert(KeyValuePair\* table[], int key, char value[]) {  // insert a key and value  printf("inserting %d,%s to the table...\n",key, value);  int hash = hashFunc(key);  int index = hash;  if (table[index] != NULL) {  int FULL = 1;  for(int i = 1; i < 10; i++) { //index + 10 means it can move 9 times  int new\_index = (index + i \* i) % HASHTABLE\_SIZE;  if(table[new\_index] == NULL) {  // add the new key-value pair to the correct position  printf("Entry inserted at position %d\n", new\_index);  KeyValuePair\* newEntry = (KeyValuePair\*) malloc(sizeof(KeyValuePair));  newEntry->key = key;  newEntry->value = value;  table[new\_index] = newEntry;  FULL = 0;  return;  }  } //check the hole table one time  printf("The table is full!\n");  } else {  // add the new key-value pair to the correct position  printf("Entry inserted at position %d\n",index);  KeyValuePair\* newEntry = (KeyValuePair\*) malloc(sizeof(KeyValuePair));  newEntry->key = key;  newEntry->value = value;  table[index] = newEntry;  }  }  char\* lookup(KeyValuePair \*table[], int key) {  // look up a key and return the value  int hash = hashFunc(key);  int index = hash;  //printf("index = %d\n", index);  if (table[index] != NULL) {  if (table[index]->key == key)  return table[index]->value;  else {  for(int i = 1; i < 10; i++) {  int new\_index = (index + i \* i) % HASHTABLE\_SIZE;  if(table[new\_index] == NULL)  return NULL;  if(table[new\_index]->key == key)  return table[new\_index]->value;  }  printf("No this key! Please check!!\n");  return NULL;  }  } else {  return NULL;  }  } |

If you have updated the functions properly, the table should look like this at the output. We can see that the clustering seem to have improved a little bit. :)

|  |
| --- |
| 0: <1450922,Claire>   1: <1450677,Perry>   2: -   3: -   4: <1450957,Arthur>   5: -   6: -   7: <1450017,Ted>   8: <1450345,Jerry>   9: <1450191,Bill>  Lookup 1450017 - result: Ted  Lookup 9999999 - result: (null)  Lookup 1450677 - result: Perry  Lookup 1450957 - result: Arthur |

# Problem 4

One example is not good enough to verify if quadratic probing does indeed provide better resistance against clustering and thus reducing the amount of probing operations needed.

Using the **exact same data in the previous problems**, use 100 different universal hash functions (by randomization or otherwise) and calculate the average number collisions for inserting an entry using linear probing v.s. quadratic probing. In other words, you should:

1. Generate a new random hash function H
2. Clear the hashtable
3. Add all data to the hashtable with H using linear probing
4. Calculate the average number of probabing needed
5. Clear the hashtable again (!)
6. Add all data to the hashtable with H using quadratic probing
7. Calculate the average number of probabing needed
8. Go back to step 1 for 100 times

|  |
| --- |
| *Hint:* In lecture notes, one possible form of universal hashing function is in the form:  h(k) = (ak+b) mod p mod m  To get a random hashing function, you simply randomize a and b. Note that a should be greater than 0 and b should be greater than or equal to 0. |

**Discussion:** Does quadratic probing really help in this case as shown by empirical data? By how much, if so?

|  |  |  |
| --- | --- | --- |
|  | Linear Probing | Quadratic probing |
| Average number of probing needed for insert (over 100 different universal hash functions) | 2.36 | 1.98 |
| Comment on empirical result:  Yes. Average less than 0.38 | | |

# Problem 5 (Optional, Hard)

In reality, we never create a hashtable and stick with its size forever. When the table is too congested (high load factor), we enlarge it in a process called **rehashing**.

Implement a rehashing feature to your program. In your C program:

* Use dynamic memory allocation for the hashtable array so that we can specify size in run time
* Use a global variable for hashtable size, for convenience. At the beginning, use a size of 10, and assume maximum size is 160
* Your hash function should be aware of this variable hashtable size, and so should all other functions; please make sure your hash function remains valid as your hashtable grows
* Implement a CheckLoadFactor function that checks the load factor of the hashtable. The load factor is the ratio of occupied slots in the hashtable:
  + If the load factor is above 0.5, double the size of the hashtable array immediately, and rehash original data to this new table; note as your table size changes, your hash function also changes
* Check if your rehashing is correct by adding many key-value pairs, and retrieve some values via the keys to see if the table remains correct

Paste your program with rehashing here:

|  |
| --- |
|  |

|  |
| --- |
| Please finish other Lab Exercises if you haven't already! |