2413 Assignment 6

Due Date
3-7-16 11:59pm

For this assignment you will be implementing a Map using an open address hashtable. In order to get your output to match the expected output, you will need to follow the parameters of the assignment exactly.

Our map will be from Tokens to Tokens. A Token is a string which contains no whitespace.

Input Commands From Standard In:

```
Define Token1 Token2
Lookup Token1
Delete Token1
```

Sample Input:

```
Define ABCDEFG 123456
Define HAHAHAHA!!! Laughing
Define ABC123 DoReMe
Lookup ABC123
Lookup YabbaDabbaDo
Delete YabbaDabbaDo
Delete ABC123
```

1 Overview

For grading purposes, your implementation is going to be very very noisy. Whenever there is a collision, you will generate a statement on output indicating so. Whenever you resize the table or change the parameters of your hash function, you will indicate so. It is very important that you follow these directions exactly so that your output matches the expected output.

You will need an array of prime numbers. Please use this array exactly:

1. Primes[0] = 2
2. Primes[1] = 3
3. Primes[2] = 5
4. Primes[3] = 7
5. Primes[4] = 13
6. Primes[5] = 29
7. Primes[6] = 59
8. Primes[7] = 113
9. Primes[8] = 223

You will be using 3 primes as follows: When your table size is Prime[i], your hash function for the starting index is seeded with Prime[i-1] and your offset on collisions is the hash function seeded with Prime[i-2]. Your table will start at Prime[i=2].

You will need to resize the table and re-hash all of its contents when the density reaches > 0.5. If Prime[i] is your table size, then you resize when you are adding the element which pushes the number of occupied positions over Prime[i]/2 (integer division).

2 Map Data Structures

Token

1. char text[101];

The Key and Value types are both Tokens

We need a way to compare two different keys for equality.

define a function "int sameKey(Key* k1, Key*k2)" which returns 1 when k1 and k2 have the same text, otherwise returns 0.

Entry:

1. Key K;
2. Value V;

Open Address Entry (OAEntry):

1. Entry e;
2. int state; //Indicating type of occupancy in the open address table.

Hashtable

1. int * Primes;
2. OAEntry * array;
3. int i;
4. int op; //number of spots currently or previously occupied

Map is a hashtable

3 Create Hashtable (and destroy)

Algorithm:

1. Allocate Space for the meta data, pointed to by m
2. Allocate space for the prime array and initialize it to the above primes.
3. m→i = 2
4. Allocate space for the OAEntry Array of length m→Primes[m→i].
5. Initialize the state of the OAEntry to all be 1. (cell has not ever been occupied)
6. m→op = 0

In the destroy function, just free any allocated memory.

4 hash function

unsigned long hash(unsigned int P, unsigned char * c, unsigned int length, unsigned long tableSize)

1. unsigned long a = 0;
2. unsigned long i ;
3. for(i =0; i < length; i++)
   - a = a * P + c[i];
4. return a % tableSize;

5 Lookup

Lookup will take a map and a key and return a pointer to a value if it exists. In order to maximize utility of looking up entries, we are going to write a helper function which returns the index of the correct entry or -1 if it doesn't exist.

Value * Lookup(Map *m, Key k)

1. long index = Lookup2(m,k);
2. if index == -1,
   (a) `printf("Entry for Key:%s Not Found\n", (char*) &k);`
   (b) return NULL
3. else
   (a) `printf("Entry for Key:%s Found At Index:%d\n", (char*) &k, index);`
   (b) return a pointer to the value at the position of index in the array.

long Lookup2(Map * m, Key k)

1. int start = hash(m→i-1, (unsigned char*)& k, sizeof(Key), m→Primes[m→i]);
2. int offset = hash(m→i-2, (unsigned char*)& k, sizeof(Key), m→Primes[m→i]);
3. int index = start;
4. while(m→array[index].state != 1)
   (a) `printf("** Looking For Key:%s At Index:%d\n", (char*) &k, index);`
   (b) if(m→array[index].state == 2)
      i. if(sameKey(& k, & m→array[index].e.k)) return index;
   (c) index += offset % m→Primes[m→i];
5. return -1;

6 Define

This code will be slightly different than demonstrated in class. We want to make sure there are not multiple entries with the same key to ensure that resizing works properly.

void Define(Map *m, Key k, Value v)

1. int index = Lookup2(m, k)

2. if (index > -1)
   (a) m→array[index].e.v = v;
   (b) `printf("Updating Entry For Key:%s At Index:%d\n", (char*) &k, index);`
   (c) return;
3. int start = hash(m→i-1, (unsigned char*)& k, sizeof(Key), m→Primes[m→i]);
4. int offset = hash(m→i-2, (unsigned char*)& k, sizeof(Key), m→Primes[m→i]);
5. int index = start;
6. while(m→array[index].state != 1)
   (a) if(m→array[index].state == 3)
      i. m→array[index].e.k = k;
      ii. m→array[index].e.v = v;
      iii. m→array[index].e.state = 2;
      iv. `printf("Creating Entry For Key:%s At PO Index:%d\n", (char*) &k, index);`
      v. return;
   (b) index += offset % m→Primes[m→i];
7. m→array[index].e.k = k;
8. m→array[index].e.v = v;
9. m→array[index].e.state = 2;
10. `printf("Creating Entry For Key:%s At NO Index:%d\n", (char*) &k, index);`
11. m→op++;
12. if(m→op > m→Primes[m→i]/2) resize(m);

7 resize

In resizing, we essentially need to make a new map whose array is twice the size and move over every entry into the new map. To do this we are going to store the old array and old values.

void resize(Map * m)

1. OAEntry * old = m→array;
2. unsigned long oldSize = m→Primes[m→i];
3. m→i++;

4. `printf("RESIZING: FROM:%d TO:%d\n", m->Primes[m->i-1],m->Primes[m->i]);`
5. m→array = (OAEntry*)malloc(sizeof(OAEntry)*m→Primes[m→i]);
6. Initialize the state of the new OAEntries to all be 1. (cell has not ever been occupied)
7. For(i = 0; i ¡ oldSize; i++)
   (a) if (old[i].state == 2)
      i. Define(m, old[i].e.k, old[i].e.v)
8. free(old);
9. `printf("DONE RESIZING\n");`

8 Delete

The simplest way to delete is to mark the entry's cell as not occupied.

void delete(Map * m, Key k)

1. unsigned long index = Lookup2(m,k);
2. if index == -1,
   (a) `printf("Entry for Key:%s Not Found\n", (char*) &k);`
3. else
   (a) *m→array[index].state = 3;
   (b) `printf("Deleting Entry for Key:%s At Index:%d\n", (char*) &k, index);`

9 Main Loop

While you can read in commands.

1. command == "Define"
   (a) Read Key and Value tokens
   (b) `printf("Define Key:%s Value:%s\n", Key, Value);`
   (c) Define the key value pair in the map.
2. command == "Delete"
   (a) Read Key token
   (b) `printf("Delete Key:%s\n", Key);`

(c) Delete the key from the map

3. command == ”Lookup”

   (a) Read Key token

   (b) `printf("Lookup Key:%s\n", Key);`

   (c) Lookup The Key's Value

   (d) `printf("Found Value:%s\n", Value);`