CS202 -- Lab 4

- CS202 -- Data Structures and Algorithms I
- James S. Plank
- Fall, 2021
- Directory for files, makefile, executables and gradescript: **/home/jplank/cs202/Labs/Lab4**
- Last modified: *Mon Sep 13 23:58:29 EDT 2021*

You should read through the CS202 lecture notes on Hashing before you attempt this lab.

The Hash_202 Class, And Your Job

First, get yourself to a clean directory, and then do:

```
UNIX> cp -r /home/jplank/cs202/Labs/Lab4/src .
UNIX> cp -r /home/jplank/cs202/Labs/Lab4/include .
UNIX> cp /home/jplank/cs202/Labs/Lab4/makefile .
UNIX> mkdir obj
UNIX> mkdir bin
```

Your job is to implement the class `Hash_202`. This implements a hash table, where the keys are strings of hexadecimal digits, and the vals are arbitrary, non-empty strings. Most of the lab description is in the header file, **include/hash_202.hpp**. I'll give more information after putting the file here:

```
#pragma once
#include <vector>
#include <string>

class Hash_202 {

  public:
    /* There is no constructor or destructor here.  This allows you to simple declare
       these as variables, and not worry about memory allocation. */

    /* You must call Set_Up() to set up the table before using it.  This method
       takes the hash table size, a name of a hash function (either "Last7" or "XOR"), and
       the name of a collision resolution strategy (either "Linear" or "Double").

       You should error check your input, and look for the following errors, in the following
       order.  If you see one of these errors, return the string specified:

       - Table already set up:                "Hash table already set up"
       - Bad table size:                      "Bad table size"
       - Bad hash function:                   "Bad hash function"
       - Bad collision resolution strategy:   "Bad collision resolution strategy" */

    std::string Set_Up(size_t table_size, const std::string &fxn, const std::string &collision);

    /* Add() adds the given key/val pair to the hash table.  If successful, it should return
       an empty string.  Add should test for the following errors, in this order,
       and return the given strings:

           - Table not set up:                "Hash table not set up"
           - Empty string for the key:        "Empty key"
           - Key not composed of hex digits:  "Bad key (not all hex digits)"
           - Empty string for the val:        "Empty val"
           - The hash table is full:          "Hash table full"
           - Cannot insert key:               "Cannot insert key"
           - The key is already in the table  "Key already in the table"
     */

    std::string Add(const std::string &key, const std::string &val);

    /* Find() returns the val associated with the given key.  If the hash table has not been
       set up yet, or if the key is not in the hash table, or if the key is not composed of
       all hex digits, it should return an empty string.

       Find() is not const, because it sets the variable Nprobes to equal the number of
```

```
      probes that it took to find the answer. */

   std::string Find(const std::string &key);

   /* Print() prints all non-empty slots in the hash table, one per line.
      The index should be printed first, right justified and padded to five characters.
      Then a space, the key, a space, and the val. This should do nothing if the hash
      table has not been set up yet. */

   void Print() const;

   /* Total_Probes() should traverse the hash table, and for every key, caculcate how many probes
      it takes find that key.  It should return the total number of probes.  It should
      return 0 if the hash table has not been set up yet.  It is not const, because it
      uses Find() to find the number of probes for each key. */

   size_t Total_Probes();

 /* Obviously, you can use these however you want, but here is how I used them.  You may
    not modify or add to this header file, so you need to be able to implement the lab with
    just these member variables. */

 protected:
   std::vector <std::string> Keys;   // The hash table of keys.
   std::vector <std::string> Vals;   // The Vals[i] is the val associated with Keys[i]
   size_t Nkeys;                        // This is the number of keys that have been inserted.
   int Fxn;                          // The hash function.  I have 'X' = "Xor" and 'L' = "Last7".
   int Coll;                         // The collision resolution strategy.  I have 'L' and 'D'
   size_t Nprobes;                      // When I call Find(), I set this value to the number of probles.
};
```

This defines a class called **Hash_202** class. It will be used to store keys and values. Both are strings. The keys are strings of hexadecimal digits, just like the MD5 and SHA-1 hashes. They may be of arbitrary size, and must be composed solely of the characters 0-9, a-f or A-F. The values are strings.

There is no constructor, so when you create an instance if the class, it will contain an empty hash table. To create the hash table, you must call **Set_Up()**, which defines the size of the hash table, plus its hash function and collision resolution strategy. We will support two hash functions, "**Last7**" and "**XOR**". I'll define them later. The hash table will use open addressing with one of two collision resolution strategies: "**Linear**" for linear probing and "**Double**" for double hashing. More on that later too.

In the **protected** data of the class, you you represent the hash table with two arrays -- one of keys and one of values. They'll both be the same size, and the element in **Vals[i]** will correspond to the key in **Keys[i]**. You represent an empty slot in **Keys** and **Vals** with an empty string.

Add() adds the given key/val pair to the hash table with the proper hash function and collision resolution strategy. Please see the header file for information on error handling.

Find() looks for the given key in the hash table and returns its associated value. Please see the header file for information on error handling.

Print() prints the hash table -- see the header file for the format.

Total_Probes() traverses the hash table, and for every key, calculates how many probes it takes to find the key. It returns the total number of probes. I implemented this by calling **Find()** on every key, and having **Find()** set the **Nprobes** variable. This is the reason why both **Find()** and **Total_Probes()** are not **const** methods.

I've written a testing program in **src/hash_tester.cpp**. This follows the same methodology as your last lab. You call it with three arguments (or four if you want a prompt):

```
UNIX> bin/hash_tester
usage: hash_tester table-size fxn(Last7|XOR) collision(Linear|Double) [prompt]
UNIX>
```

Let's call it with an 8 element hash table, "Last7" and "Linear", and then see the commands:

```
UNIX> echo '?' | bin/hash_tester 8 Last7 Linear
A key val      Add the given key/val pair to the hash table.
F key          Find the given key in the hash table.
P              Print the hash table.
TP             Print the total probes for finding all keys.
```

```
Q               Quit.
?               Print comands.
UNIX>
```

That's striaghtforward. We'll use this to demonstrate examples later, but I need to do some more explanation next.

However, just to be explicit: *Your job is to write **src/hash_202.cpp**, which implements the methods of the `Hash_202` class. The only file that you submit is **src/hash_202.cpp**. You are not allowed to modify **include/hash_202.hpp** or **src/hash_tester.cpp**.*

Reading Hexadecimal

To read an integer **i** in hexadecimal from standard input, you do:

```
cin >> hex >> i;
```

Extending on this, **src/read_as_hex.cpp** reads strings from standard input and assumes that they are hexadecimal representations of integers. It prints out the decimal and hexidecimal values of the integers:

```
/* This program shows how you read a value from a
   string in hexadecimal using a stringstream.

   The printf() statement prints the value in
   decimal and then in hexadecimal. */

#include <string>
#include <cstdio>
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
  string s;
  istringstream ss;
  int i;

  while (cin >> s) {
    ss.clear();
    ss.str(s);
    if (ss >> hex >> i) printf("%d 0x%x\n", i, i);
  }
  return 0;
}
```

Giving it "a", "10" and "20" as input prints out the proper values:

```
UNIX> echo a 10 20 | bin/read_as_hex
10 0xa
16 0x10
32 0x20
UNIX>
```

Why do I tell you this? Well.....

Hash Functions

The keys that you are going to store in your hash tables are hexadecimal strings that can be any length. You need to hash these strings into integers. We are going to do this in two ways.

The first way is called **Last7**. It treats the last seven digits of the hash string as a number in hexadecimal. Thus, the hash of "**b9937df3fefbe66d8fcdda363730bf14**" will be 120635156, which is equal to 0x730bf14. If the string has fewer than 7 characters, then simply treat the whole string as a hexadecimal number. Thus, the hash of "**11**" will be 17, which is equal to 0x11.

The second hash function is called **XOR**. With this hash function, you break up the string into seven-character words, and then treat each word as a number in hexadecimal. The last word can be fewer than seven characters. You then calculate the bitwise exclusive-

or of each of these numbers. That is your hash function. So, let's take an easy example: "**a000000a0000101**". First, we break this up into seven-digit words (except for the last one): "**a000000**", "**a000010**" and "**1**". Their bitwise exclusive-or is equal to 0x11, so the **XOR** hash of "**a000000a0000101**" is equal to 17.

Collision Resolution

Linear probing is straightforward. With double hashing, you will use the specified hash function as the first hash function, and the other one as the second hash function. If the second hash function ends up yielding an increment of zero, set it to one.

Examples

Let's start with a simple example -- putting one value into a hash table.

```
UNIX> bin/hash_tester 10 Last7 Linear 'Hash> '
Hash> A a Fred
Hash> P
    0 a Fred
Hash> Q
UNIX>
```

Since 0xa = 10, the hash value is 10, which means that the key/val go to index 0.

Let's try some of those example keys above:

```
UNIX> bin/hash_tester 10 Last7 Linear 'Hash> '
Hash> A b9937df3fefbe66d8fcdda363730bf14 Fred        # This hashes to 0x730bf14 = 120635156,
Hash> P                                              # So it goes into index 6.
    6 b9937df3fefbe66d8fcdda363730bf14 Fred
Hash> A 11 Luther                                    # This hashes to 0x11 = 17,
Hash> P                                              # So it goes into index 7.
    6 b9937df3fefbe66d8fcdda363730bf14 Fred
    7 11 Luther
Hash> F b9937df3fefbe66d8fcdda363730bf14
Found: Fred
Hash> F Fred                                         # Fred is a value, not a key, so we can't find it.
Not found.
Hash> Q
UNIX> bin/hash_tester 10 XOR Linear 'Hash> '
Hash> A a000000a0000101 Fred                         # Recall from above that this hashes to 0x11 = 17.
Hash> P
    7 a000000a0000101 Fred
Hash> Q
UNIX>
```

This example shows linear probing in action because "**001**", "**1**" and "**a000000b**" all hash to values that equal 1 mod 10:

```
UNIX> bin/hash_tester 10 XOR Linear 'Hash> '
Hash> A 001 Fred
Hash> A 1 Binky
Hash> A a000000b Baby Daisy
Hash> P
    1 001 Fred
    2 1 Binky
    3 a000000b Baby Daisy
Hash> TP
3
Hash> Q
UNIX>
```

The total number of probes is three, because finding "**001**", requires zero probes, "**1**" requires one and "**a000000b**" requires two. The total is therefore three.

Let's try a harder example with double hashing:

```
UNIX> bin/hash_tester 10 Last7 Double 'Hash> '
Hash> A 2 Fred
Hash> A 00000100000002 Binky
Hash> A c Luther
Hash> P
```

```
    0 00000100000002 Binky
    2 2 Fred
    4 c Luther
Hash> TP
2
Hash> Q
UNIX>
```

Each of these keys hashes to two. First, "**2**" goes into index two. Then "**00000100000002**" collides. Its hash value using **XOR** is 0x12 = 18. We take 18 mod 10 and that gives us an increment of 8. Thus, the first probe will try 2+8 = 10, which is 0 mod 10. Since that slot is empty, "**00000100000002**" goes into index 0. Finally, when "**c**" collides, its hash value using **XOR** is also 2. Therefore, it goes into index 2+2=4.

When we call **TP**, it returns 2, because "**2**" has zero probes, and "**00000100000002**" and "**c**" have one each.

Two final examples -- the first shows an example where the second hash produces an increment of zero. In that case, you set it to one:

```
UNIX> bin/hash_tester 10 Last7 Double 'Hash> '
Hash> A 0 Fred
Hash> A a Binky
Hash> P
    0 0 Fred
    1 a Binky
Hash> TP
1
Hash> Q
UNIX>
```

When we insert "**a**", both hash functions return 10, which equals 0 mod 10. Thus, we set the second hash function to 1, and the value goes into index 1.

The last example shows three keys that hash to 5 with both functions:

```
UNIX> bin/hash_tester 10 Last7 Double 'Hash> '
Hash> A 5 Fred
Hash> A f Binky
Hash> P
    0 f Binky
    5 5 Fred
Hash> A 19 Luther
A 19 Luther
Cannot insert key
Hash> Q
UNIX>
```

When we try to put the third key into the table, it can't go there, because it keeps trying indices five and zero. Your code needs to work in this case -- it needs to identify that the key cannot go into table.

The gradescripts for this lab simply call your **bin/hash_tester** and mine on input files that have commands. Those input files are in the directory **/home/jplank/cs202/Labs/Lab4/Gradescript-Examples**. Files that end with **.txt** are the commands, and files that end with **.cli** are the command line arguments for **bin/hash_tester**.