15-110: Principles of Computing, Spring 2018

Lab 7 – Thursday, March 1

Goals

This lab is aimed at developing your understanding of the importance of hash function design for the efficiency of search using a hash table. When you are done, you should be able to:

1. State the worst-case performance of search in a hash table (with an arbitrary hash function).
2. Explain what features of a hash function affect the performance of search in a hash table and why.
3. Explain why search in a hash table can be expected to take constant time, given a good enough hash function and a large enough table.

Part 1: TA Demonstrations (Hash Functions)

The correct operation of a hash table relies on a hash function that maps keys (in our case, these are strings) to non-negative integers that indicate which bucket the key belongs to. The hash table will work as long as the hash function maps each `key` to a non-negative integer less than the size of the hash table (`ht_size`), but it will only perform well if it distributes keys relatively uniformly across buckets.

Consider the following hash functions:

1. Perhaps the simplest possible hash function is one that just ignores the key and always returns 0:

```
def h0(key,ht_size):
    return 0
```

2. Another hash function obtains the ASCII code of the first character in the key, divides that code by the size of the hash table, and returns the remainder of that division. This can be coded in Python as:

```
def h1(key,ht_size):
    return ord(key[0]) % ht_size
```

3. Another hash function totals up the ASCII codes of all of the characters in the key, divides that total by the size of the hash table, and returns the remainder of that division:

```
def h2(key, ht_size):
    x = 0
    for i in range(len(key)):
        ascii_code = ord(key[i])
        x = x + ascii_code
    return x % ht_size
```

This scheme examines all the letters in the string, but is not sensitive to order. So the strings "act", "cat", and "tac" will all hash to the same value.

4. It is also possible to build a hash function that is position sensitive, in that each position is scaled by a power of some constant, like 128. So "ab" would be treated differently than "ba". In the first case, the numerical code for "a" would be multiplied by 128 and then the numerical code for "b" would be added, whereas in the second case, the numerical code for "b" would be multiplied by 128 and then the numerical code for "a" would be added. To make this into a Python hash function, one need only convert the string into a Python integer and then take the remainder modulo the size of the hash table:

```
def h3(key,ht_size):
    x = 0
    for i in range(len(key)):
        ascii_code = ord(key[i])
        x = 128 * x + ascii_code
    return x % ht_size
```

Self-Directed Activities

1. Hash Functions: For each of the hash functions, `h0`, `h1`, `h2`, and `h3`, find the result of hashing the following strings for a table of size 100,000.

   i. `"Hash table"`
   ii. `"Table hash"`
   iii. `"Towers of Hanoi"`

   You can do this easily by downloading the file `hash_table.py` from Autolab into your `lab7` directory. Then you can start python3 and load the hash functions and run them:

```
> python3 –i hash_functions.py
>>> h0("Hash table", 100000)
0
>>> etc.
```

   For each of these hash functions, would you get any collisions (i.e., more than one key hashing to the same bucket) if you were to insert these keys into a hash table?
   Record the results in a text file `experiments.txt` .

2. Hash Table Statistics: Our hash tables are implemented as Python lists where each element is a bucket. That bucket is a list that holds the entries that hashed to that bucket's index. In order to evaluate the effectiveness of different hash functions it is useful to be able to gather some statistics about the hash table after we've inserted some entries.

    i. Define a Python function `largest_bucket(ht)` in `largest_bucket.py` In your `lab7` folder that returns the maximum number of entries contained within any single bucket of the hash table `ht`. Remember that the hash table is implemented as a list of lists. Each interior list is a "bucket". You can follow this algorithm:

        1. Set *max_so_far* to 0.
        2. For each element *bucket* in the hash table, do the following:
           a. If the length of *bucket* is greater than *max_so_far*, then:
                set *max_so_far* to the length of *bucket*
        3. Return *max_so_far*.

    Example:

```
>>> largest_bucket([["a","b"],[],["w","x","y","z"],["c"]])
4
```

    ii. Define a Python function `mean_bucket(ht)` in `mean_bucket.py` in your `lab7` folder that returns the arithmetic mean of the number of entries in non-empty buckets of the hash table `ht`. You can follow this algorithm:

        1. Set *nonempty_buckets* to 0.
        2. Set *entries* to 0.
        3. For each element, *bucket*, in the hash table, do the following:
            a. If the length of *bucket* is greater than 0.
               i. Add one to *nonempty_buckets*.
               ii. Add the length of *bucket* to *entries*.
        4. Return *entries/nonempty_buckets*

    Example:

```
>>> mean_bucket([["a","b"],[],["w","x","y","z"],["c"]])
2.33333333333333
```

    iii. What is the point of gathering these statistics? Ideally, would we want the maximum bucket size and the mean bucket size to be small or large? Write a brief explanation in `experiments.txt` .

3. Hash Table Experiments: Download and save the code in `hash_table.py` from Autolab into your `lab7` folder. Look at the code. This code implements hash table creation (`new_hash_table`), insertion (`ht_insert`), search (`ht_search`), and imports the four hash functions (`h0`, `h1`, `h2`, `h3`) and your work from problem 2. There is a "main" function called `run(hash_fun)` that sets up a hash table, inserts a large number of words, and then does some searches. It measures the elapsed time for these operations, and reports some statistics about the buckets (using your functions `largest_bucket` and `mean_bucket` you wrote for problem 2).

   Load `hash_table.py` by typing

   ```
   python3 -i hash_table.py
   ```

   into the terminal, and execute `run("h0")`, `run("h1")`, `run("h2")`, and `run("h3")`, which will perform the timing and statistical measurements for hash tables with the respective hash function. In interpreting the results, take note of the fact that run() inserts 100,000 words but only searches 1000 times.

   Record the output of these runs in `experiments.txt` . How well do the different hash functions perform for the insertion and search operations? What correlation do you see between how evenly the hash function distributes keys across buckets and the search performance of the hash functions?

Submission

When you finish the lab, you should be inside the `lab7` folder, which is inside the `private/15110` directory. When you type `ls` and press the Enter key, you should see the following files: `experiments.txt`, `hash_functions.py`, `largest_bucket.py`, `mean_bucket.py`, and `hash_table.py`. Once you see all files, please type `cd ..` to move up one folder and press the Enter key. Then, zip your `lab7` folder by typing `zip –r lab7.zip lab7` and you should get a `lab7.zip` file in the current folder. Please submit the zipped file `lab7.zip` on Autolab under 'lab 7'.