Assignment 4 (100 points)

Anirudh Sivaraman

2021/11/02

1 Written Questions (70 points)

See Gradescope for the written questions.

2 Coding questions (30 points)

Use the starter code (hash_tables.py) to implement the three different schemes for exact matching we discussed in class: standard hashing (5 points), 2choice hashing (10 points), and 2left hashing (15 points).

2.1 Code walkthrough

hash_tables.py is the only Python file you'll need to work with for this assignment. It takes two arguments. The first one is the occupancy of the hash table (i.e., what fraction of the hash table is populated with entries). The second one is the name of the hashing scheme ("2choice", "2left", or "standard"). For instance, to run hash_tables.py with the standard hashing scheme with an occupancy of 30%, you would run the following command:

```
Anirudhs-Air:sims anirudh$ python3 hash_tables.py --occupancy 0.3 --hash_type standard
Fraction of trials in which slot size did not exceed capacity  0.671
```

The output of running this command tells you how often the hash table did not overflow. To understand how this is calculated, let's walk through hash_tables.py.

The values of the HashTable object are:

1. self.occupancy: The occupancy of the hash table as a fraction between 0 and 1. This is fed in by the user.

2. self.num_slots: The number of slots in the hash table. For all three hashing schemes, we structure the hash table as an array with a certain number of slots, where each slot can hold up to a certain number of elements.

3. self.cap_per_slot: The capacity of each slot. This reflects hardware realities of how many elements can be held in each slot. We have set this to 5 because it is in the same ballpark as the number of elements in each slot for real hardware implementations.

4. self.total_trials: The number of trials. For each trial, we check if the hash table overflowed: did the hash table run out of space because a particular slot that an element hashed into was already full? Note that we say that the hash table overflowed if *even a single* slot's capacity has been exceeded—even though the hash table might still have space in other slots. This is why the fraction of trials in which the hash table did not overflow reaches 0 long before the occupancy of the hash table reaches 1.

5. self.num_elements: The number of elements in the hash table. This is computed by multiplying the occupancy fraction with the product of the number of slots and the capacity of each slot (i.e., the maximum occupancy of the entire hash table).

After the constants have been initialized, the loop in run_hash_table runs (`for seed in range(TOTAL_TRIALS):`). This for loop runs a number of independent trials, each initialized with a different random seed to ensure the randomness in each trial is independent of other trials.

In each trial, we run through the total number of elements in the hash table in another for loop (`for i in range(self.num_elements):`). For each element, we compute the slot it hashes into depending on the hashing scheme, which is another user-supplied input. You need to implement three hashing schemes.

1. standard: We compute a single slot number for each new element regardless of the current occupancy of each slot.

2. 2choice: We compute 2 slot numbers using two independent hash functions, and then pick the slot number that has lower occupancy, breaking ties randomly.

3. 2left: We conceptually divide the hash table into two subtables with equal number of slots in each subtable. We compute a slot number in each subtable using two independent hash functions. Of these 2 slots in 2 subtables, we pick the slot that has lower occupancy, but always break ties in favor of one of the two subtables.

To compute a random slot number, you do not need to (and should not!) use an actual hash function for this assignment. Instead, you can use the Python function call (`random.randint(U, L)`) to generate a random slot number between U and L (both U and L included). You can test the occupancy at a particular slot number using the `occupancy_at_slot` array. For each algorithm, make sure that the slot number you pick is returned from its respective function because this is the index variable we use to update the `occupancy_at_slot` array.

2.2 Testing your code

The easiest way to test your code is to remember that for a given occupancy, the fracton of trials in which the hash table does not overflow is going to be highest for 2left, followed by 2choice, followed by standard. For instance, when I run my solutions for all three algorithms with an occupancy of 0.6, here's what I see.

```
Anirudhs-Air:sims anirudh$ python3 hash_tables.py --occupancy 0.6 --hash_type 2left
Fraction of trials in which slot size did not exceed capacity  0.999
Anirudhs-Air:sims anirudh$ python3 hash_tables.py --occupancy 0.6 --hash_type 2choice
Fraction of trials in which slot size did not exceed capacity  0.994
Anirudhs-Air:sims anirudh$ python3 hash_tables.py --occupancy 0.6 --hash_type standard
Fraction of trials in which slot size did not exceed capacity  0.0
```

Also, when the occupancy of the hash table goes up, the performance of all three schemes must degrade: the fraction of trials in which the hash table does not overflow should go down. When I run the three algorithms with an occupancy of 0.7, here's what I see.

```
Anirudhs-Air:sims anirudh$ python3 hash_tables.py --occupancy 0.7 --hash_type 2left
Fraction of trials in which slot size did not exceed capacity  0.95
Anirudhs-Air:sims anirudh$ python3 hash_tables.py --occupancy 0.7 --hash_type 2choice
Fraction of trials in which slot size did not exceed capacity  0.844
Anirudhs-Air:sims anirudh$ python3 hash_tables.py --occupancy 0.7 --hash_type standard
Fraction of trials in which slot size did not exceed capacity  0.0
```