CS2040S Semester 1 2023/2024

Data Structures and Algorithms

Tutorial+Lab 06
Table ADT 1: Hash Table; Midterm Debrief

For Week 08

Document is last modified on: October 9, 2023

1 Introduction and Objective

In this tutorial, we will continue our discussion about Hash Table, **one possible efficient** implementation of Table ADT (unordered). We will heavily use `https://visualgo.net/en/hashtable` in this tutorial. We will contrast and compare two collision resolution methods that each has its own strengths and weaknesses.

2 Tutorial 06 Questions

Hash Function Basics

Q1). Which of the following is the best (string) hash function?

1. `int index = (rand() * (key[0]-'A')) % N;`
2. `int index = (key[0]-'A') % N;`
3. `int index = hash_function(key) % N;`

where

- `rand()` is a function that returns a pseudo-random integral number in the range between 0 and `RAND_MAX` (This value is library-dependent, but is guaranteed to be at least 32767 on any standard library implementation).
- `key` is a string

- `N` is the hash table size, usually a prime number
- `hash_function(v)` is as shown in `https://visualgo.net/en/hashtable?slide=4-7`

Ans: The best answer given these three options is option 3, but see the reasons below.

1. This is a non-deterministic hash. Impossible to be used as hash function.

2. The hash values might not be uniformly distributed since it only uses the 1st letter and there are only 26 possible alphabets. Extreme example: What if `keys` are NUS matric numbers that all start with character 'A'?

3. `hash_function(v)` as shown in `https://visualgo.net/en/hashtable?slide=4-7` is a base 26 computation: `key[0]`$*26^{(N-1)}$+`key[1]`$*26^{(N-2)}+ ... +$`key[`$N-1$`]`. It is deterministic, that is, given the same string it will always give the same hashcode and it has a good distribution since it uses the entire string (each character and its position) to compute the hashcode (albeit limited to ['A'..'Z'] in its current form). Modify this hash function a bit if we deal with alphanumeric and/or mix of UPPER and lower cases.

Q2). A good hash function is essential for good Hash Table performance. A good hash function is easy/efficient to compute and will evenly distribute the possible keys (necessary condition to have good performing Hash Table implementation). Comment on the flaw (if any) of the following (integer) hash functions. Assume that for this question, the load factor α = number of keys N / Hash Table size $M \leq 10$ (i.e., small enough for our Separate Chaining or Linear Probing implementation) for all cases below:

1. $M = 100$. The keys are $N = 50$ positive even integers in the range of [0, 10 000].
   The hash function is h(key) = key % 100.
2. $M = 100$. The keys are $N = 50$ positive integers in the range of [0, 10 000].
   The hash function is h(key) = floor(sqrt(key)) % 101.
3. $M = 101$. The keys are $N = 50$ positive integers in the range of [0, 10 000].
   The hash function is h(key) = floor(key * random) % 101, where $0.0 \leq$ random ≤ 1.0 (C++ `rand()`/Python `random()`/Java `Random` class).

Possible answers:

1. **No key will be hashed directly to odd-numbered slots in the table, resulting in wasted space, and a higher number of collisions in the remaining slots than necessary. Aside from the hash function, the hash table size may not be good as it is not a prime number. If there are many keys that have identical last two digits, then they will all be hashed to the same slot, resulting in many collisions.**

2. **Keys are not uniformly distributed. Many more keys (except 10 000) are mapped to the higher-indexed slots, e.g., only 0 is mapped to 0; [1..3] are mapped to 1, [4..8] are mapped to 2, [9..15] are mapped to 3, [16..24] are mapped to 4,..., [9801..9999] are mapped to 99, only 10 000 is mapped to 100. PS: Although 101 is a prime, the % 101 operation is actually useless for this hash function.**

3. **This hash function is not-deterministic. The hash function does not work because, while using a given key to retrieve an element after inserting it into the hash table, we cannot always reproduce the same address. PS: Now 101 is a prime but it doesn't help due to non-deterministic nature of this hash function.**

Key message to be shared with students: If we are just hashing a simple built-in data type: an integer, a double (but may have its own problem), or a string, just use C++17/Python 3/Java17 LTS built-in hash functions for those native data types. For Java, we can use `HashMap<Integer, Integer> mapper1`, `HashMap<Double, Integer> mapper2`, or `HashMap<String, Integer> mapper3` directly without defining our own hash function.

Hash Table Basics

Q3). Hashing or No Hashing: Hash Table is a Table ADT that allows for `search(v)`, `insert(new-v)`, and `delete(old-v)` operations in O(1) average-case time, **if properly designed**. However, it is not without its limitations. For each of the cases described below, state if Hash Table can be used. If not possible to use Hash Table, explain why is Hash Table not suitable for that particular case. If it is possible to use Hash Table, describe its design, including:

1. The <Key, Value> pair
2. Hashing function/algorithm
3. Collision resolution (OA: LP/QP/DH or SC; give some details)

(Choose 2 out of 3 to be discussed live): The cases are:

1. A population census is to be conducted on every person in your (very large, e.g., population of 1 Billion) country. You can assume that no two person have the same name in this country. However, there can be two or more person with the same age. You can assume that age is an integer and within reasonable human age range [0..150] years old. We are only interested in storing every person's name and age. The operations to perform are: retrieve age by name and retrieve list of names (in any order) by age. Important consideration: Each year, everyone's age increases by one year, a bunch of new babies (age 0) are born and added into the database, some people unfortunately pass away and removed from the database. All these yearly changes have to be considered.

2. A different population census similarly contains only the name (in full name, again, guaranteed to be distinct) and the age of every person. The operation to perform is: Retrieve all person(s) (his/her/their full name(s) and age(s)) given a last (sur)-name. Note that although the full names are distinct, their last (sur-)names may not.

3. A grades management program stores a student's index number and his/her final marks in one GCE 'O' Level subject. There are 100,000 students, each scoring final marks in [0.0, 100.0] (the exact precision needed is not known). The operation to perform is: Retrieve a list of students who passed in ranking order (highest final marks to passing marks). A student passes if the final marks are more than 65.5. Whether a student passes or not, we still need to store all students' performance as the passing final marks can be adjusted as per necessary.

Possible Answers:

1. **Yes, we can use 2 (Hash) Tables for efficient lookup both ways (nobody is restricting us, so we can use more than one data structure if that simplifies our life :O).**
   Moreover, to handle the requirement that the age of everyone increases by one each year, we do not store the age, but store the birth_year instead and maintain an integer current_year = 2023 (that simply increases by +1 each year).
   This is because if we store the actual age, we need to loop through the entire keys (each year) to update the ages by +1 for everyone.

   The first Table ADT is for name to birth_year lookup:
   <Key, Value> Pair: <name, birth_year>, to easily find age = current_year - birth_year of a given (distinct) name
   If a baby is born, we add his/her entry with birth_year = current_year.
   If a person pass away, we delete his/her entry.
   Hashing Algorithm: h(name) = standard string hashing
   Collision Resolution: SC or OA: DH using a different h2(name) - either is fine.

   The second Table ADT is for age (use birth_year = current_year-age) to list of names lookup:
   Since birth_year is integer and of small range, we can actually use Direct Addressing Table (DAT).
   <Key, Value> Pair: <birth_year, vector<name>>, to find list of names (in any order) given a birth_year (but excluding those who have passed away, as we will lazily delete him/her from the second hash table if he/she pass away).
   If a baby is born, we add his/her entry with birth_year = current_year.
   If a person pass away, we can lazily choose to not immediately remove him/her.
   'Hashing' Algorithm: Direct Addressing, i.e., h(v) = v.
   'Collission Resolution' (no actual collision): SC, append additional names that have same birth_year at the back of vector. It is a bit not natural to use OA methods for this application.

Assuming the typical birth years are between [1873..2173], the implementation can be something like vector<string> age[301] where index 0/150/300 corresponds to birth_year 1873/2023/2173, respectively.

2. **Yes, we can extract the last_name (surname) from a full_name (using string tokenization).**
   <Key, Value> Pair: <last_name, vector<pair<full_name, age>>>
   Hashing Algorithm: h(last_name) = standard string hashing
   Collision Resolution: Separate chaining for people with the same h(last_name) or any good Open Addressing techniques (e.g., Double Hashing). Also notice that the value itself is another vector/list.

3. **Two issues:**
   First, the floating-point precision. If the precision is fixed, e.g., 1 or 2 decimal places, then the number can actually be converted to an integer, and stored as in part (2).
   Second issue is the requirement to retrieve a list of students who passed in ranking order. Hash Table is NOT designed for this ordered operations, we need to use the next data structure: balanced BST.

Q4). Quick check: Let's review all 4 modes of Hash Table (use the Exploration mode of `https://visualgo.net/en/hashtable`). During the tutorial session, the tutor will randomize the Hash Table size M, the selected mode (LP, QP, DH, or SC), the initial keys inside, and then ask student to `Insert(random-integer)`, `Remove(existing-integer)`, or `Search(integer)` operations. This part can be skipped if most students are already comfortable with the basics.

This part is open ended, up to the tutor.

Hash Table Discussions

Q5). Choose 2 out 4 to be discussed live: The following topics require deeper understanding of Hash Table concept. Please review `https://visualgo.net/en/hashtable?slide=1`, use the Exploration Mode, or Google around to help you find the initial answers and we will discuss the details in class. For some questions, there can be more than one valid answers.

1. What is/are the main difference(s) between List ADT basic operations (see `https://visualgo.net/en/list?slide=2-1`) versus Table ADT basic operations (see `https://visualgo.net/en/hashtable?slide=2-1`)?

2. At `https://visualgo.net/en/hashtable?slide=4-4`, Steven mentions about Perfect Hash Function. Now let's try a mini exercise. Given the following strings, which are the names of Steven's current family members: {"Steven Halim", "Grace Suryani Halim", "Jane Angelina Halim", "Joshua Ben Halim", "Jemimah Charissa Halim"}, design any valid **minimal perfect hash function** to map these 5 names into index [0..4] without any collision. Steven and Grace are not planning to increase their family size so you can assume that $N = 5$ will not change.

3. Thus far, which collision resolution technique is better (in your opinion or Google around): One of the Open Addressing technique (LP, QP, DH) or the Closed Addressing (Separate Chaining/SC) technique?

4. Which non-linear data structure should you use if you have to support the following three operations: 1). many insertions, 2) many deletions, and 3) many requests for the data in sorted order?

Possible Answers:

1. **In List ADT, we generally want to insert a new value `v` at a specific index `i` whereas in Table ADT, we let the ADT's underlying data structure to decide where to store `v` internally. Then in List ADT, we generally want to remove existing item at a specific index `i` whereas in Table ADT, as we don't specify where `v` should be located, we want to remove existing value `v` instead.**

2. **Well, we can use the first character after the first space in the name (recall basic string processing) and we will have 'H', 'S', 'A', 'B', 'C'. Those five characters are all different and can be mapped to [0..4] without any collision.**

3. **For camp SC: Separate Chaining looks like the implementation choice for C++ STL `unordered_map`, Python `dict/set`, and Java `HashMap` as it only uses one hash function (compared to two hash functions in Double Hashing Open Addressing technique). Also, there will be potentially lots of deletions (the library code need to at least provide such mechanism). Open Addressing techniques will require the usage of "DEL" markers, which will prolong the probing length after many interleaving insertions and removals. Although the time complexity of search (and deletion – that need search too) is $O(1 + \alpha)$, most of the time we can adjust $\alpha = n/m$ to be a small constant as we usually know what is the upperbound of keys that we ever need at one time n and thus we can set table size m 'appropriately' to make α small.**

   For camp OA (especially DH that minimizes primary and/or secondary clustering): If our Hash Table applications involve very few (or no) deletions (which is true for certain database applications, e.g., alumni database: once an alumni, always an alumni, no (or rare) deletion ever occur), then using Open Addressing technique is better as there is no extra memory overhead of using separate chains (usually linked lists), a bit faster (due to better cache memory usage), and has better time complexity if $\alpha = n/m$ is about 50% (not near full).

4. **Definitely not Hash Table. Hash Table is good for frequent insertions, weaker if there are many deletions (if we use Open Addressing as discussed earlier), and it totally cannot efficiently enumerate the data inside the Hash Table in sorted order, as the data is simply... unordered inside the Hash Table. For this, we will need to wait until next Tut07 when we discuss (balanced) Binary Search Tree.**

Hands-on 5

TA will run the second half of this session with a few to do list:

- Very quick review of Java HashSet and HashMap,
- Do a speed run of VisuAlgo online quiz that are applicable so far, e.g., `https://visualgo.net/training?diff=Medium&n=5&tl=5&module=hashtable`,
- Finally, live solve another chosen Kattis problem involving Table ADT (that does *not* require ordering) but has interesting time complexity analysis (to be fully understood in CS3230 later).

For PS3 Debrief, mention what is/are relevant for your tutorial/lab group.
1-2 slides for Java demonstrations, just show a few techniques or refer students to relevant references, e.g., `https://github.com/stevenhalim/cpbook-code/blob/master/ch2/nonlineards/unordered_map_unordered_set.java`.
Show off in 5m :), again (on another set of random questions).

Use Kattis /bokforing. The main issue is to avoid doing $O(N)$ per 'RESTART' (there can be many restarts). If we try to store money information in a vector/array of size $N + 1$ will be TLE due to the many restarts, $O(QN)$.

One way to do this is to use hash table to map index to value. We call `.clear()` to erase all contents of the hash table and store the default value (x) in a separate integer (to be used for each 'GET' of index that is not in the hash table). In a glance, this looks like a costly $O(N)$ operation. But notice that we only delete what was 'SET' before, so the total number of all 'RESTART' operations, combined, will overall be just $O(Q)$, not $O(QN)$:). One day in CS3230, you will learn about such amortized analysis.

Problem Set 4

We will end the tutorial with **last-minute algorithmic** discussion of PS4.
We are now allowed to discuss 100+100 solutions in high level.

Tutorial TA can now discuss in high level on what are required to get 100+100 points for PS4 A+B, respectively, in somewhat vague way.

For PS4A (/swaptosort), the required data structure is clearly an UFDS. For each pair of swap-able indices, we can merge merge into one 'connected component' (CC). Then we run a simple linear check on whether index 0 is swap-able (directly or indirectly, because they are in the same CC) with index $n - i - 1$, and so on (we can stop halfway).

For PS4B (/kaploeb), this is actually an easy Hash Table (plus a bit of sorting at the end). You probably need two hash tables (total time and lap count of each runner).