IT5003 Oct-Dec 2023

Data Structures and Algorithms

Tutorial+Lab 05
Table ADT: Hash Table and BST

Document is last modified on: October 9, 2023

1 Introduction and Objective

In the first half of the session, we will review:

- `https://visualgo.net/en/hashtable` data structure for Table ADT, especially its Closed Addressing/Separate Chaining collision resolution technique versus what is likely used in Python set: Open Addressing/Linear Probing collision resolution technique, and
- `https://visualgo.net/en/bst` data structure, the unbalanced version, as another alternative data structure to implement Table ADT. However, since the full form (balanced BST) is only discussed briefly during recitation, this part is for theoretical interest only in this IT5003 module.

2 Questions

Hash Function Basics

Q1). 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 Python, we can use `dict` or `defaultdict` (to avoid the no key quirks). PS: For Python users, type in `hash(any-int)` to see that the default hash value of any small integer is itself (I don't know yet on what is the actual function used for big integer). You can also try to type in `hash('some-string')`, e.g., `hash('Steven Halim') = 8454583414177198303`. (I don't know yet on what is the actual function used for hashing string).

Hash Table Basics

Q2). 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 — only LP in IT5003 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 last_name or any good Open Addressing techniques (e.g., Double Hashing).

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.

Binary Search Tree

Q3). (Optional, only when many are still not comfortable with basic bBST operations): We will start this tutorial with a quick review of basic BST operations that is not necessarily balanced. The tutor will first open `https://visualgo.net/en/bst`, click Create → Random. Then, the tutor will ask students to Search for some integers, find Successor of existing integers, perform Inorder Traversal, Insert a few random integers, and also Remove existing integers.

This part is open ended, up to the tutor, and can be totally skipped (or only briefly mentioned) if the tutor detects that most students are roughly okay with basic BST tasks.

If need be, the review of basic operations can be done quickly and the tutor has to be creative enough to ask for various corner cases, starting from operations that do not change the underlying BST:

1. Search, vary the request between existing versus non-existing integer, and close to root versus as far as possible.

2. Successor, actually the tutor will ask students to answer Predecessor operation instead to verify understanding of this mirror operation of Successor. The tutor will vary the request between vertex that has Predecessor or not (has left child versus has no left child) and the tutor will also ask about the minimum element (it has no Predecessor). Optional: Discuss the subtle difference between these two strategies to find Successor of v if v has no right child: A). go up to parent(s) until we encounter a right turn (the current setup in VisuAlgo – but this REQUIRES us to maintain parent pointers) versus start from the root, search for v. If we go left, we keep the current vertex as 'potential successor' (the one currently implemented in BSTDemo.cpp – this eliminates the need to maintain parent pointers).

3. Inorder traversal, trivial. Show them shortcut that just echo the content of the BST in sorted order instead of manually performing Inorder traversal. The output of Inorder traversal of a BST is guaranteed to be sorted. Also take this opportunity to introduce two quick variants: Pre-order traversal and Post-order traversal of BST (or any tree) structure.

Then, the tutor can ask about two operations that modify the content of the BST:

1. Insert, vary the request between inserting to new leaf close to root versus as far as possible.

2. Delete/Remove, vary the request between the three deletion cases (leaf vertex, vertex with one child, vertex with two children).

Further Discussions

Q4). (Choose 2 out of 3 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 answer.

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. Which non-linear data structure should you use if you have to support the following three operations that can come in any order: 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, i.e., just say: remove existing value `v`.
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. 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 use (balanced) Binary Search Tree. The version that we learn in IT5003 is the standard (not self-balancing) ones to save time (otherwise we will not have enough time to discuss graph data structure and simple graph traversal algorithms). So, this possibly unbalanced BST is still not fully useful yet.

Hands-on 5

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

- PS4 Quick Debrief,
- Do a sample 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,bst`.
  PS: Skip parts that are skipped for this sem's IT5003.
- Finally, live solve two chosen (short) Kattis problems involving Table ADT.

Do PS4 quick debrief according to the context of your lab group.

Do VisuAlgo Online Quiz sample run in medium setting.
VA OQ sample run should skip topics that are not included in IT5003.

`https://nus.kattis.com/problems/keywords`.

Kattis /keywords, first: replace hyphens with spaces, transform string to lowercase, then it is a simple application of Hash Table (of strings) (Python set()) to avoid duplicates. At the end we report the size of the set.

`https://nus.kattis.com/problems/shoppinglist`.

Kattis /shoppinglist is an application of Hash Table (Python set()) again. For each shopping list, do set intersection with the running answers (items that appear in all lists). At the end, we need to call sort to output the answers in sorted (alphabetical) order.

Problem Set 5

We will end the tutorial with high level discussion of PS5 A+B.

For PS5 A (/variablenamn): this is a doable simulation task especially with help of a hash table (set of taken variable names). However, if we just simulate it like that, there is a huge chance you will get Time Limit Exceeded. There is a very inefficient step if we just do this simulation verbatim... What if we also remember what was the furthest multiple of any variable x too? From here it is very close to Accepted. PS: This task was used in final assessment last December 2022.

For PS5 B (/quickscope): Let's concentrate on the first two subtasks: when there is no ‘’ and ‘’, this becomes and easy Hash Table problem. Hash a variable to its type. For the third subtask, you can simulate the depth-10... For the last 10 points subtask, you need to think more and is reserved for those aiming to get A/A+ in this module... (can you think of the solution on your own under two hours)?