Hash Tables

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Hash Tables

Standard reminder to set phones to silent/vibrate mode, please!

Hash Tables

- Today's class:
  - Introduce/discuss storage of unrelated/unordered data
    - Discuss the idea of indexing
  - Introduce the idea of hash functions
  - Introduce the use of hash functions for efficient storage of sets or associative containers.
  - Discuss chained hash tables

Hash Tables

- Let's start with an example:
  - Say that we need to store data about the students (for example, for evaluation purposes, automating grading, etc.)
  - Each student is identified by their student ID
  - Then, we need to store, for each student, a bunch of data (list of assignments/exams/etc. with grades and comments/feedback that the students can review)

Hash Tables

- Now, the user enters a student ID (or a student logs in, which implicitly supplies a student ID for the system to process), and we need to look up all the information.
  - Question: how do we do this efficiently?

Hash Tables

- Now, the user enters a student ID (or a student logs in, which implicitly supplies a student ID for the system to process), and we need to look up all the information.
  - Question: how do we do this efficiently?
  - Answer 1 (incorrect): we have a list of students, and we linearly scan that list until finding the given student ID.

    (*why is this an incorrect answer?*)

Hash Tables

- Now, the user enters a student ID (or a student logs in, which implicitly supplies a student ID for the system to process), and we need to look up all the information.
  - Question: how do we do this efficiently?
  - Answer 2 (better): we could store the student IDs in order, so that we don't scan linearly, but instead do binary search (logarithmic time is dramatically better than linear!)

Hash Tables

- Now, the user enters a student ID (or a student logs in, which implicitly supplies a student ID for the system to process), and we need to look up all the information.
  - Question: how do we do this efficiently?
  - Answer 2 (better): we could store the student IDs in order, so that we don't scan linearly, but instead do binary search (logarithmic time is dramatically better than linear!)

Hash Tables

- Follow-up question:
  - What is dramatically better than logarithmic time?

Hash Tables

- Follow-up question:
  - What is dramatically better than logarithmic time?
  - Ok (I assume that you did answer that question! :-)), so how can we obtain constant time for this look-up operation?

Hash Tables

- Follow-up question:
  - What is dramatically better than logarithmic time?
  - Ok (I assume that you did answer that question! :-)), so how can we obtain constant time for this look-up operation?
  - Hint: What data structure have we seen that offers constant time access to a given element?

Hash Tables

- And again, assuming that you answered that question....
- Follow-up question: Does it make sense to even consider using an array where the student ID is the subscript??
  - Hint: Look at your student ID ... That's 8 digits!!

Hash Tables

- And again, assuming that you answered that question....
- Follow-up question: Does it make sense to even consider using an array where the student ID is the subscript??
  - Hint: Look at your student ID ... That's 8 digits!!
  - We'd need an array of 100 million elements!! (and each student's data is perhaps a few kilobytes, so we're talking a few hundred GIGAbytes for this!!)

Hash Tables

- Sounds a bit excessive, given that we're trying to store the data of, say, 200 or 300 students (in general — in your case, it would be just a bit above 100)

Hash Tables

- Sounds a bit excessive, given that we're trying to store the data of, say, 200 or 300 students (in general — in your case, it would be just a bit above 100)

- Any ideas ... ?

Hash Tables

- Sounds a bit excessive, given that we're trying to store the data of, say, 200 or 300 students (in general — in your case, it would be just a bit above 100)

- Any ideas ... ?

  Unfortunately, I can't stop and wait for your answer, as the answer to this is immediately necessary to continue :-)

Hash Tables

- How about, we don't use the whole student ID as the subscript, but just several of its digits?
- We'd certainly address the issue of the (ridiculously) excessive amount of storage required!
- But...

Hash Tables

- Do we run into any problems?
- Let's see, say that we decide to use only two digits of the student ID — that way, we only need an array of 100 elements.
  - So, which two digits?
  - The first two digits?
    - Hint: My student ID's first two digits are 20 .... Are yours by any chance 20 ???

Hash Tables

- How about the last two digits?
  - Definitely better, but not good enough (not by a long shot!)
  - It's better because in a group of students selected randomly, the last two digits are more or less random(ish), so we have better chances that each student gets a unique subscript for the array.
  - However, the array only has 100 elements, and you guys are 110+, so there will necessarily be more than two students with the same last two digits in their student IDs.

Hash Tables

- How about the last THREE digits, instead of just two?
  - Now, we have 1000 positions for the array, so each student will get a unique subscript in that array ...

Hash Tables

- How about the last THREE digits, instead of just two?
  - Now, we have 1000 positions for the array, so each student will get a unique subscript in that array ...

    Right...?

Hash Tables

- How about the last THREE digits, instead of just two?
  - Now, we have 1000 positions for the array, so each student will get a unique subscript in that array ...

    Right...?
  - Most definitely NO !! We have much better chances, but we do NOT have the guarantee that no two students will have the same last three digits. (in fact, we'll talk about the birthday paradox!)

Hash Tables

- Putting aside (just for a minute) this detail, do notice that we're talking about a trade-off here:
  - We increase the number of digits, and we improve our chances that the data structure will work.
  - However, we increase the amount of storage required, and the amount being “wasted” (in our example, we now need 1000 elements, to store a little over 100 students' data — not too efficient!)

Hash Tables

- However, this is the idea of Hash tables.
- In general, what we do is:
  - Take the data that we need to store (more specifically, the identifier, or “index” of the data that we need to store — in our case, the student ID as the identifier for a student's data) and transform it to a smaller and fixed size
  - Use that transformed data as the subscript in an array.

Hash Tables

- This transformation is called *hashing*, and it is done through a *hash function*.
- Two important aspects about this hash function:
  - It is computed in constant time (in general, it is a single expression; worst-case, a fixed number of iterations doing some fixed number of calculations at each round)
  - It is definitely not an injective function (hopefully you remember this from math? $f(x) = x^3$ is an injective function; $f(x) = x^2$ or $\sin(x)$ are not)

Hash Tables

- BTW ... Why can't this hash function be injective?

Hash Tables

- BTW ... Why can't this hash function be injective?
- And also, if it's not injective, meaning that different data (in our case, different student IDs) will produce the same hash value (this is known as *collisions*), then how can we make the hash table work?

Hash Tables

- An analogy:
  - Suppose we have a workplace with 50 employees, and we have mailboxes.
  - We could have just 26 mailboxes, one for each letter of the alphabet, and then, whatever correspondence for one of the employees will be placed in the mailbox with the first letter of their last name (for example, correspondence to me, Moreno, will be placed in the M mailbox)

Hash Tables

- Will this scheme work? Will I be able to find my correspondence?
- Notice a problem:
  - In the mailbox, my mail will be mixed with that of Anne Meyers, Peter Mathews, and Jane Martin.

Hash Tables

- Will this scheme work? Will I be able to find my correspondence?
- Notice a problem:
  - In the mailbox, my mail will be mixed with that of Anne Meyers, Peter Mathews, and Jane Martin.
    - Is this *really* a problem??

Hash Tables

- Will this scheme work? Will I be able to find my correspondence?
- Notice a problem:
  - In the mailbox, my mail will be mixed with that of Anne Meyers, Peter Mathews, and Jane Martin.
    - Is this *really* a problem??
  - Notice that it would definitely be a problem if the mail was identified just by the M, and not by my name.

Hash Tables

- Coming back to the issue of storing student's data indexed by student ID: the analogy here is that if we just take the last three digits of the student ID and just store the data of the student at that position, then things will not work:
  - We'll need to store several students' data at the same position (analogy with “in the same mailbox”); but if we just leave it as student 314, then we won't know if it is the data of student 20111314 or the data of student 201222314.

Hash Tables

- As long as we store the student's data at location 314, but then, as part of the data, we store the complete student ID, then we will be able to locate the data of student 20111314; even if the data of student 20222314 is also stored (in the same location).

Hash Tables

- BTW ... How do we store several students' data in the same location of the array?? (same subscript, therefore it is just one location).

- I will leave this one for you to think about it.

Hash Tables

- Next class, we'll take a look at two strategies to deal with collisions:
  - Chained hashing
  - Open addressing
    - Linear probing
    - Double hashing
- We'll also take a look at how to compute good hash functions (well, better — and more general — than just taking the last three digits of a student ID)