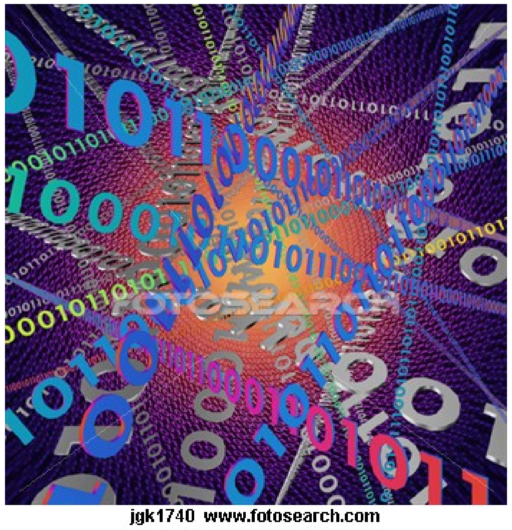
Hash Tables – Double hashing



Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

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

Hash Tables – Double hashing

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

Hash Tables – Double hashing

- Today's class:
  - We'll look at one of the issues with linear probing, namely clustering
  - Discuss double hashing:
    - Use one hash function to determine the bin
    - A second hash function determines the jump size for the probing sequence.
  - Look at some practical issues and approaches to deal with these issues.

Hash Tables – Double hashing

- One important problem with linear probing is clustering — as collisions start to occur, then blocks of contiguous occupied bins (clusters) appear.

Hash Tables – Double hashing

- One important problem with linear probing is clustering — as collisions start to occur, then blocks of contiguous occupied bins (clusters) appear.
- And a quite unfortunate aspect is that the longer these clusters, the longer our searches or insertions (or deletions) will take (and remember that we wanted them to be constant time *and* fast!)

Hash Tables – Double hashing

- An even more unfortunate aspect is the fact that the longer these clusters, *the more likely* it will be that they will grow with each insertion!
  - This is because a new value inserted will make the cluster grow if the hash falls anywhere in the interval $[C_S-1, C_E+1]$, where C_S, C_E are the beginning and the end of the cluster, respectively.
    - Any hash that falls in the cluster will end up taking the position C_E+1, as a result of the linear probing.

Hash Tables – Double hashing

- One idea that could come to mind is to do linear probing using a jump size *p*; that is, if there is a collision, instead of skipping to the next bin to probe it, skip *p* bins forward and probe there.

Hash Tables – Double hashing

- One idea that could come to mind is to do linear probing using a jump size *p*; that is, if there is a collision, instead of skipping to the next bin to probe it, skip *p* bins forward and probe there.
- The bad news: It turns out that if the jump size is fixed, this does not make the slightest difference with respect to our "standard" linear probing (i.e., with jump size $p = 1$)

Hash Tables – Double hashing

- This is a direct consequence of the jump size being fixed.
    - Jump size different from one just makes it a bit more difficult to visualize, but the problem is exactly the same

Hash Tables – Double hashing

- So... What if we could choose a different jump size for each insertion?

Hash Tables – Double hashing

- So... What if we could choose a different jump size for each insertion?
- For example, the first insertion uses jump size 1, second insertion jump size 2, and so on...

Hash Tables – Double hashing

- So... What if we could choose a different jump size for each insertion?

- For example, the first insertion uses jump size 1, second insertion jump size 2, and so on...
  - Would this work, and avoid the issue of clustering?

Hash Tables – Double hashing

- So... What if we could choose a different jump size for each insertion?
- For example, the first insertion uses jump size 1, second insertion jump size 2, and so on...
  - Would this work, and avoid the issue of clustering?

  - Clearly, not at all — how could we do lookups, if the probing sequence depends on the order in which the elements were inserted, which we don't (can't) keep track of !

Hash Tables – Double hashing

- However, if the jump size was a function of the value being inserted, things would work, right?

Hash Tables – Double hashing

- However, if the jump size was a function of the value being inserted, things would work, right?
- A function of the value being inserted ... sounds familiar, doesn't it?

Hash Tables – Double hashing

- However, if the jump size was a function of the value being inserted, things would work, right?
- A function of the value being inserted ... sounds familiar, doesn't it?
- It would be a bad idea to re-use the same hash function that we used to obtain the bin
  - However, we could use a second (different) hash function

Hash Tables – Double hashing

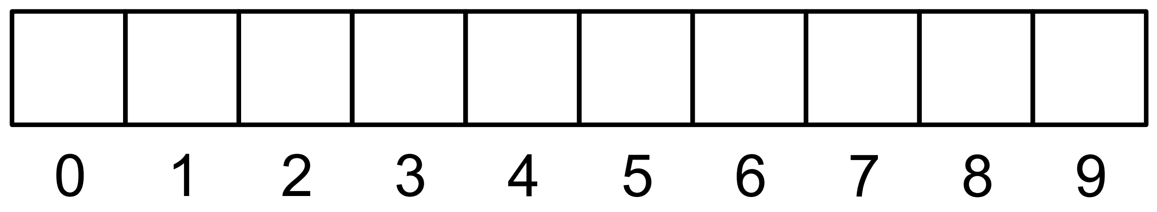
- We recall from two classes ago that we wanted to scramble the bits of the data and then select a subset of those bits (e.g., the m bits from the middle)
- What about taking advantage of the computation already done, and choose a *different* block of bits for the second hash function?

Hash Tables – Double hashing

- Let's look at an example, not with bits, but with something more human-brain-friendly:
  - The hash table uses size 10
  - For the hash function, multiply the value times 117 and keep the right-most digit
    - For the second hash function (jump size), just use the same result, and take the second digit

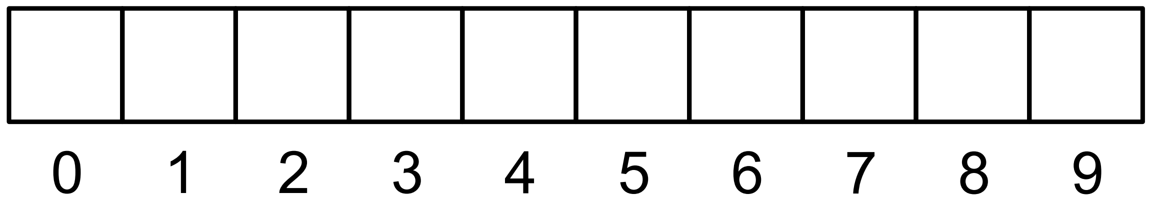
Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:



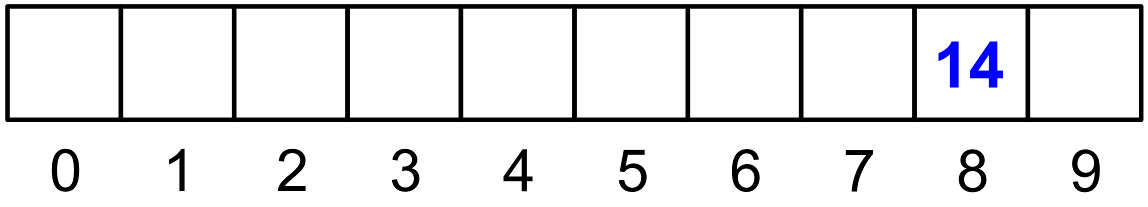
Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 14×117 = 1638 ⇒ bin 8 (and jump size 3 — not relevant now, since this insertion causes no collision)



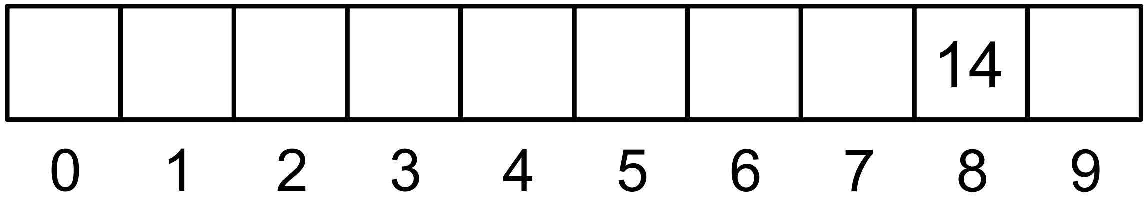
Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 14×117 = 1638 ⇒ bin 8 (and jump size 3 — not relevant now, since this insertion causes no collision)



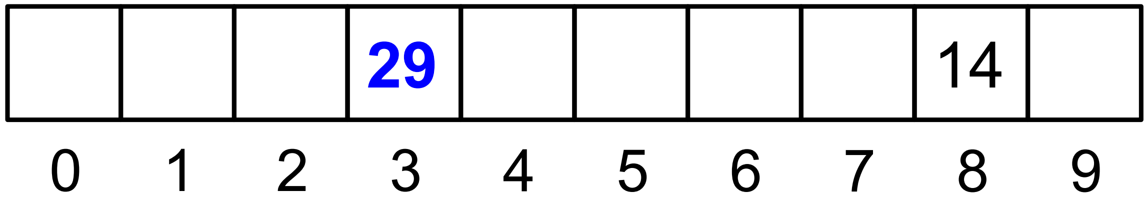
Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 29×117 = 3393 ⇒ bin 3 (jump size not relevant)



Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 29×117 = 3393 ⇒ bin 3 (jump size not relevant)



Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 43×117 = 5031 ⇒ bin 1 (jump size not relevant)

			29					14	
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 43×117 = 5031 ⇒ bin 1 (jump size not relevant)

	43		29					14	
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 19×117 = 2223 ⇒ bin 3, causing a collision (jump size given by the second digit, 2)
    - Probe bin 3 + 2 — available, so we're done:

	43		29					14	
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 19×117 = 2223 ⇒ bin 3, causing a collision (jump size given by the second digit, 2)
    - Probe bin 3 + 2 — available, so we're done:

	43		29		**19**			14	
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 5×117 = 585 ⇒ bin 5, causing a collision (jump size given by the second digit, 8)
    - Where would this one end up?

	43		29		19			14	
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 5×117 = 585 ⇒ bin 5, causing a collision (jump size given by the second digit, 8)
    - 5 + 8 (modulo 10, of course) is 3, which is already taken, so we check 3 + 8, which is 1, also taken, then bin 9!

	43		29		19			14	
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:
  - 5×117 = 585 ⇒ bin 5, causing a collision (jump size given by the second digit, 8)
    - 5 + 8 (modulo 10, of course) is 3, which is already taken, so we check 3 + 8, which is 1, also taken, then bin 9!

	43		29		19			14	**5**
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- There's a big (read: BIG!) problem with this. Let's try inserting 59:

	43		29		19			14	5
0	1	2	3	4	5	6	7	8	9

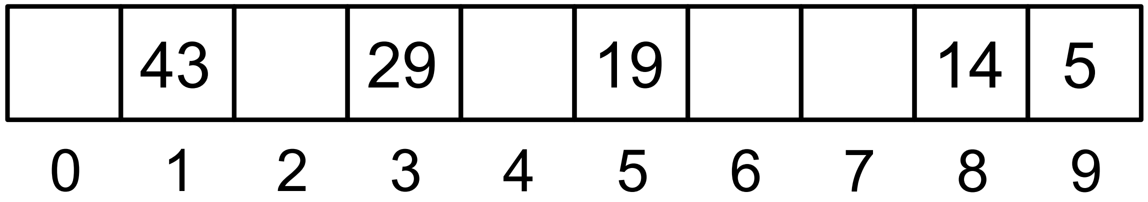
Hash Tables – Double hashing

- There's a big (read: BIG!) problem with this. Let's try inserting 59:
  - 59×117 = 6903 ⇒ bin 3, causing a collision, so we choose jump size ... Oops!

	43		29		19			14	5
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- There's a big (read: BIG!) problem with this. Let's try inserting 59:
  - Ok, so we could fix this by not allowing the second hash function to take value 0



0	1	2	3	4	5	6	7	8	9
	43		29		19			14	5

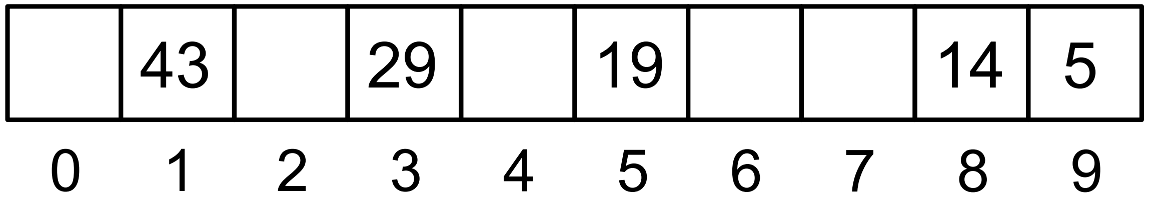
Hash Tables – Double hashing

- There's a big (read: BIG!) problem with this. Let's try inserting 59:
  - Ok, so we could fix this by not allowing the second hash function to take value 0 (could we do that with just an if? If the value is 0, just make it 1 — why not?)

	43		29		19			14	5
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- There's a big (read: BIG!) problem with this.
  - But that's not all — let's try inserting 74:



Hash Tables – Double hashing

- There's a big (read: BIG!) problem with this.
  - But that's not all — let's try inserting 74:
  - 74×117 = 8658 ⇒ bin 8, causing a collision, so we get jump size = 5, so we probe .... and... oops!

	43		29		19			14	5
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- Why does this happens?
- How do we fix it?

	43		29		19			14	5
0	1	2	3	4	5	6	7	8	9

Hash Tables – Double hashing

- The problem is that the cycle is given by the least-common-multiple of the two values.
- We'd like it to be the product of the two numbers, but if the numbers are not relatively prime (i.e., share common factors), then the LCM is lower.

Hash Tables – Double hashing

- The idea is that the jump size and the table size should be relatively prime (that is, they should have no common factors!)

Hash Tables – Double hashing

- The idea is that the jump size and the table size should be relatively prime (that is, they should have no common factors!)
    - This guarantees that every bin will be visited before cycling and start repeating bins.

Hash Tables – Double hashing

- The idea is that the jump size and the table size should be relatively prime (that is, they should have no common factors!)
  - This guarantees that every bin will be visited before cycling and start repeating bins.
  - And this is easy if one of the numbers (e.g., the table size) is prime (every number is relatively prime to a prime number, of course!)

Hash Tables – Double hashing

- But a prime table size has several problems:
  - Modulo operations become expensive (can't take advantage of bit shifts and bit and operations)

Hash Tables – Double hashing

- But a prime table size has several problems:
  - Modulo operations become expensive (can't take advantage of bit shifts and bit and operations)
  - Also, dynamically growing the size gets complicated and expensive (have to find prime numbers for the new size).

Hash Tables – Double hashing

- But wait !!! We had mentioned that we want table sizes that are powers of 2 anyway!

Hash Tables – Double hashing

- But wait !!! We had mentioned that we want table sizes that are powers of 2 anyway!
- In addition to all the good things about powers of 2 that we already saw.... Another great thing about powers of two is that their only factor is 2 ...

Hash Tables – Double hashing

- But wait !!! We had mentioned that we want table sizes that are powers of 2 anyway!
- In addition to all the good things about powers of 2 that we already saw.... Another great thing about powers of two is that their only factor is 2 ...
- So it's easy to find numbers relatively prime to them — any odd number is relatively prime to any power of 2 !!!

Hash Tables – Double hashing

- Problem: How do we get a hash function that is guaranteed to be an odd value?

Hash Tables – Double hashing

- Problem: How do we get a hash function that is guaranteed to be an odd value?
  - Hint: what does an odd value look in binary?

Hash Tables – Double hashing

- Problem: How do we get a hash function that is guaranteed to be an odd value?
  - Hint: what does an odd value look in binary?
  - Clearly, a number is odd if and only if its binary representation has a 1 in the least-significant bit.
    - Proof:

Hash Tables – Double hashing

- Problem: How do we get a hash function that is guaranteed to be an odd value?
  - Hint: what does an odd value look in binary?
  - Clearly, a number is odd if and only if its binary representation has a 1 in the least-significant bit.
    - Proof:

$$N = \sum_{k=0}^{m} b_k 2^k \;=\; b_0 + \sum_{k=1}^{m} b_k 2^k \;=\; b_0 + 2\sum_{k=1}^{m} b_k 2^{k-1}$$

Hash Tables – Double hashing

- Problem: How do we get a hash function that is guaranteed to be an odd value?
  - Hint: what does an odd value look in binary?
  - Clearly, a number is odd if and only if its binary representation has a 1 in the least-significant bit.
    - Proof:

$$N = \sum_{k=0}^{m} b_k 2^k = b_0 + \sum_{k=1}^{m} b_k 2^k = b_0 + 2\sum_{k=1}^{m} b_k 2^{k-1}$$

    - If b_0 is 1, the number has the form 1 + 2n; otherwise, it has the form 2n.

Hash Tables – Double hashing

- So, how do we get a hash function that always has the LSB set to 1?

Hash Tables – Double hashing

- So, how do we get a hash function that always has the LSB set to 1?
  - It's quite straightforward, actually; if we want, say, 8 bits total, we take seven bits from the partial result with the "scrambled" bits, shift them to the second least-significant bit, and set bit 0 to 1.

Hash Tables – Double hashing

- So, how do we get a hash function that always has the LSB set to 1?
    - It's quite straightforward, actually; if we want, say, 8 bits total, we take seven bits from the partial result with the "scrambled" bits, shift them to the second least-significant bit, and set bit 0 to 1.
    - The last step is done with bitwise OR (vertical bar operator in C++):

```
hash = ((result & mask) >> shift) | 1;
```

Hash Tables – Double hashing

- Next, we'll look at removing elements.

Hash Tables – Double hashing

- Can we use the same (efficient) trick as we did with linear probing?

Hash Tables – Double hashing

- Can we use the same (efficient) trick as we did with linear probing?
- How do we determine the sequence of elements to scan, if they're not consecutive?

Hash Tables – Double hashing

- Can we use the same (efficient) trick as we did with linear probing?
- How do we determine the sequence of elements to scan, if they're not consecutive?
- How do we check in constant time if a given element is going to be in trouble due to the hole left by the deletion?

Hash Tables – Double hashing

- Any ideas?

Hash Tables – Double hashing

- Any ideas?
  - Hint: we already saw the approach that we need here!

Hash Tables – Double hashing

- Any ideas?
  - Hint: we already saw the approach that we need here!
  - Yes — we pretty much have no choice but to mark the deleted bins with a value designated to denote a bin from which an element was removed.

Hash Tables – Double hashing

- Any ideas?
  - Hint: we already saw the approach that we need here!
  - Yes — we pretty much have no choice but to mark the deleted bins with a value designated to denote a bin from which an element was removed.
    - Like we mentioned, when searching, this special mark is interpreted as “bin is occupied”. But when inserting, the mark is interpreted as “available”, and it can be of course overwritten by the value being inserted.

Hash Tables – Double hashing

- We may want to keep a counter of how many of these special marks we have, and if it exceeds some threshold, we just reallocate and copy the elements, re-hashing everything.

Hash Tables – Double hashing

- We may want to keep a counter of how many of these special marks we have, and if it exceeds some threshold, we just reallocate and copy the elements, re-hashing everything.
  - This (a linear time operation) should not affect the fact that on average, things happen in Θ(1).

Hash Tables – Double hashing

- We also want to keep track of the load factor, λ (the ratio of elements in the table to the size of the table), so that we double the size if λ exceeds a threshold (typically 2/3 or so)

Summary

- During today's class, we discussed:
  - Clustering with linear probing
  - Double hashing:
    - Use one hash function to determine the bin
    - A second hash function determines the jump size for the probing sequence.
  - How to make the second hash suitable (typically, table size 2^m and jump size always odd)