Hash Tables (Cont'd)

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!

Announcements

- This Wednesday (February 1) we have class at the make-up time slot, 12:30 (that is, we have two classes; 12:30 to 1:20, and then 4:30 to 5:20)

- I will be in my office tomorrow (January 31) from 9:00 to 10:15, in case you have any questions (e.g., any details with the lab project).

Hash Tables

- Today's class:
  - Investigate the other important technique to deal with collisions, namely, *open addressing* or *probing*.
    - If a bin is taken, probe other bins (following a given pattern) until we find one that is empty.
  - We'll deal with two variations (two patterns of selecting the sequence of bins to probe):
    - Linear probing
    - Double hashing

Hash Tables

- The basic idea is really quite easy — let's look at linear probing:
- If a bin is taken, check the next one (in loop, so that if the next one is also taken, you continue to the next one, and so on)
  - When an empty bin is found, place the element there

Hash Tables

- The basic idea is really quite easy — let's look at linear probing:
- If a bin is taken, check the next one (in loop, so that if the next one is also taken, you continue to the next one, and so on)
  - When an empty bin is found, place the element there
  - It's really *that* easy!!

Hash Tables

- The basic idea is really quite easy — let's look at linear probing:
- If a bin is taken, check the next one (in loop, so that if the next one is also taken, you continue to the next one, and so on)
  - When an empty bin is found, place the element there
  - It's really *that* easy!! Well, ok, except for the half-a-ton of related details that we have to go over ...

Hash Tables

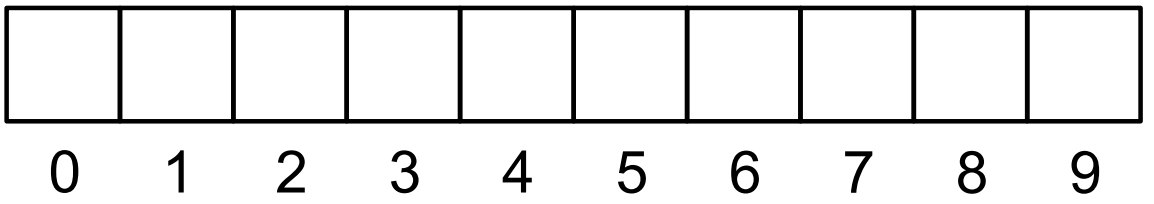
- Let's look at an example — as usual, our examples (intended for human brains to process, and not for CPUs) will use hash table size 10, storing non-negative integer values, and the hash will be the value modulo 10 (i.e., the last digit)

Hash Tables

- The example:
- Insert the numbers

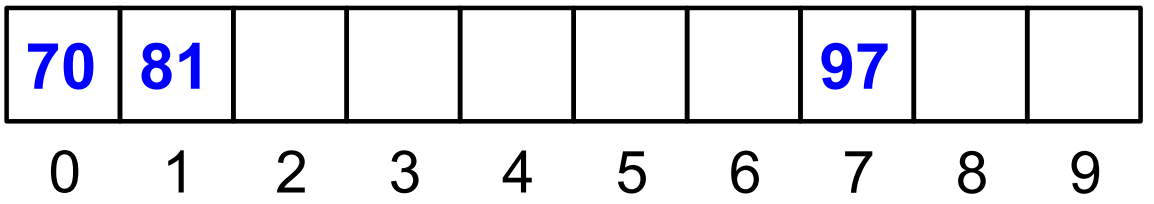
  81, 70, 97, 60, 51, 38, 89, 68, 24

  into the initially empty hash table:



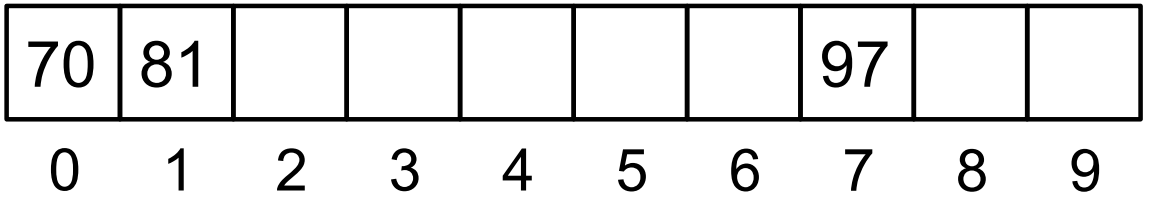
Hash Tables

- The first three are easy — no collision so far, so each element goes in its corresponding bin:



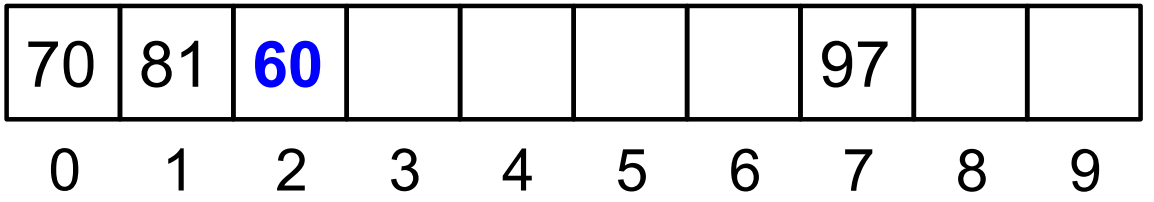
Hash Tables

- Inserting 60 causes a collision in bin 0, therefore, we check:
  - Bin 1 (also full), and
  - Bin 2 (empty)



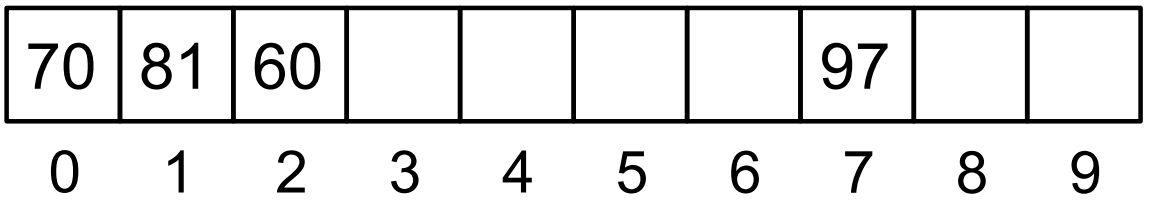
Hash Tables

- Inserting 60 causes a collision in bin 0, therefore, we check:
  - Bin 1 (also full), and
  - Bin 2 (empty)



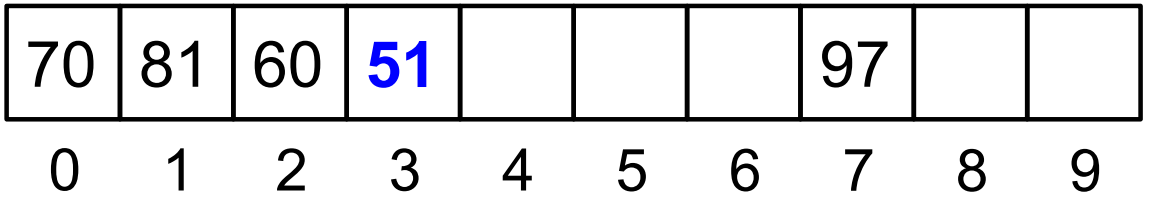
Hash Tables

- Inserting 51 also causes a collision, this time, in bin 1, therefore, we check:
  - Bin 2 (also full), and
  - Bin 3 (empty)



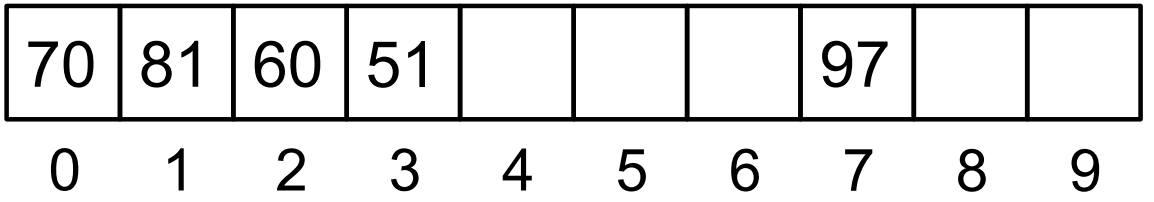
Hash Tables

- Inserting 51 also causes a collision, this time, in bin 1, therefore, we check:
  - Bin 2 (also full), and
  - Bin 3 (empty)



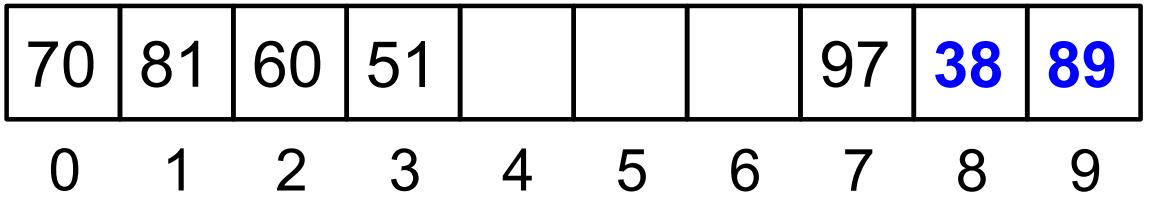
Hash Tables

- 38 and 89 can be placed into bins 8 and 9 respectively without collisions



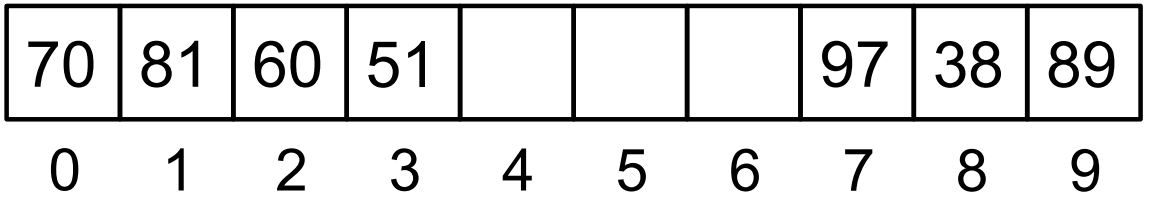
Hash Tables

- 38 and 89 can be placed into bins 8 and 9 respectively without collisions



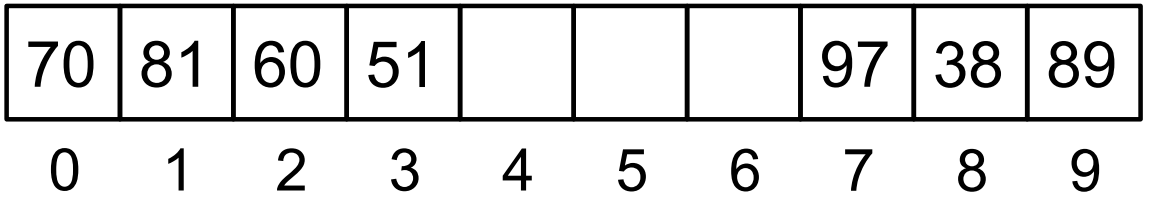
Hash Tables

- 68 causes a collision, and it looks like we're in pretty bad shape, since we run out of “next bins” to probe for available space...



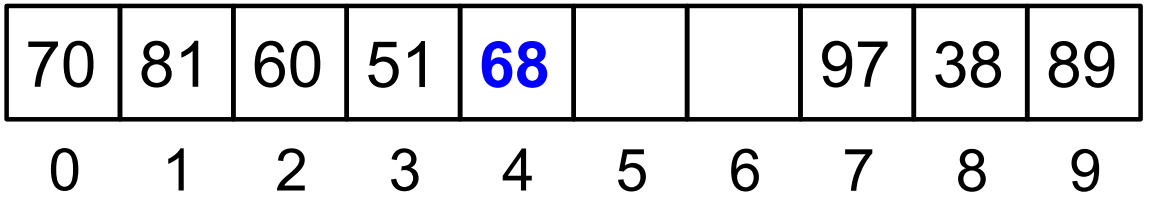
Hash Tables

- 68 causes a collision, and it looks like we're in pretty bad shape, since we run out of “next bins” to probe for available space...
- Any ideas?



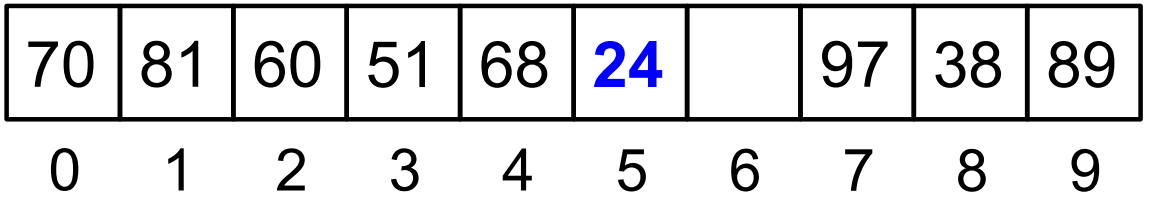
Hash Tables

- Of course — we can always use the underlying array similar to the way we use a *circular* array!
  - Check bins 9, 0, 1, 2, 3, and finally 4 which is empty
  - Insert 68 into bin 4



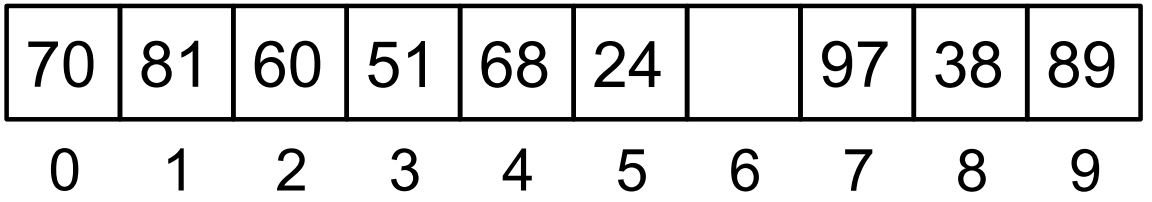
Hash Tables

- Inserting 24 causes a collision in bin 4, however the next bin is empty



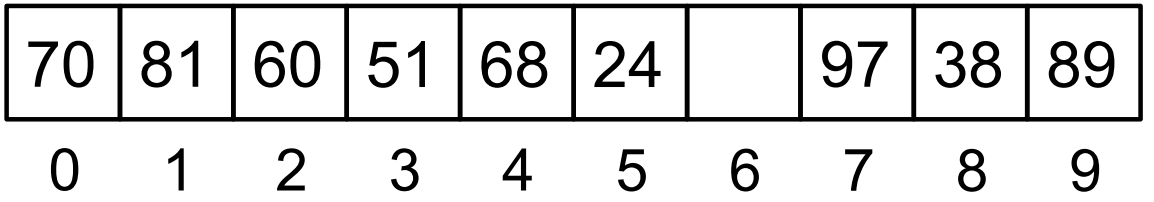
Hash Tables

- An observation:
  - If a hash table using linear probing is too crowded (i.e., a large fraction of the array is taken), we will spend a lot of time probing (back to this in a few minutes)



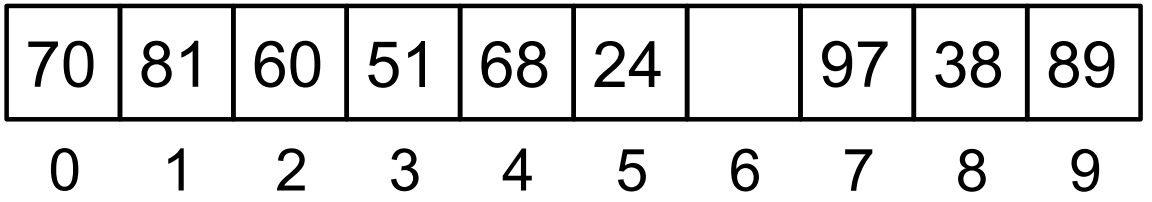
Hash Tables

- Another problem: (maybe?)
  - How do we look up values, if they maybe nowhere near where they're supposed to go? (e.g., what is 68 doing in bin 4??)



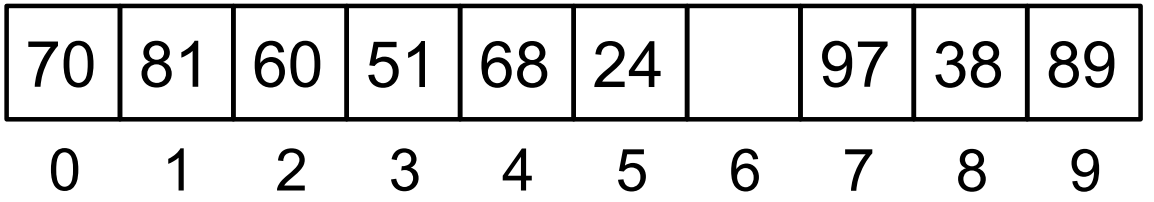
Hash Tables

- Actually, it should be pretty clear that this is not a problem — exactly as we determined that bin 4 is where it should go, we will be able to determine (following the same procedure) that bin 4 is where it is!



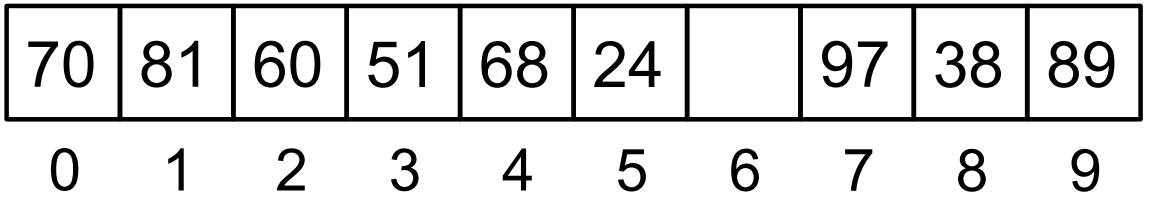
Hash Tables

- Compute the hash of the value that we're looking up
- Check the content of the bin given by that hash
  - If the value is not there, continue searching forward until:
    - The value is found
    - An empty bin is found (meaning that the value is not present)



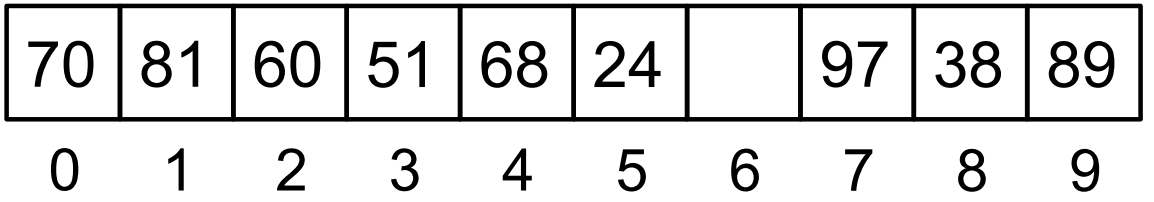
Hash Tables

- Searching for 68
  - Examine bins 8, 9, 0, 1, 2, 3, and 4, finding 68 in bin 4
- Searching for 23
  - Search bins 3, 4, 5, and 6
  - The last bin is empty; therefore 23 is not in the table



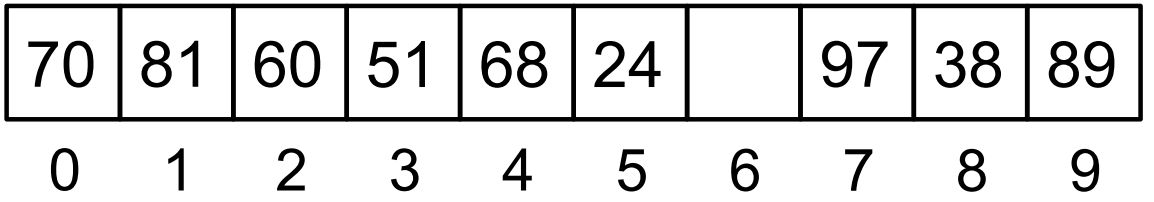
Hash Tables

- Another operation that we'd like:
  - How do we remove an element from the hash table?



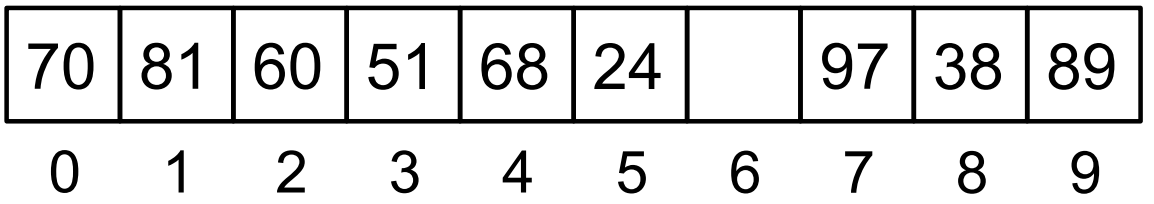
Hash Tables

- Another operation that we'd like:
  - How do we remove an element from the hash table?
  - Easy enough, right? The exact same way that we add them, or find them, we can remove them!



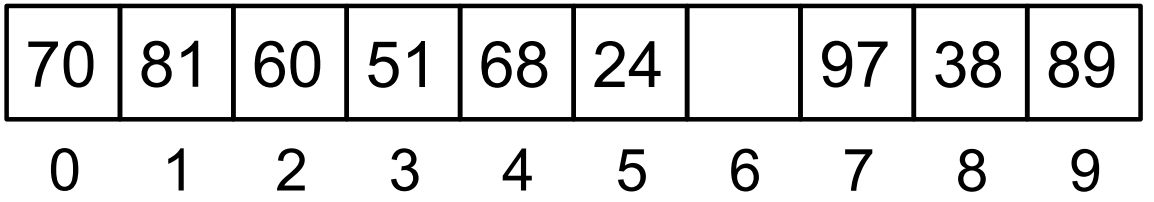
Hash Tables

- Another operation that we'd like:
  - Errrm... Are we sure?



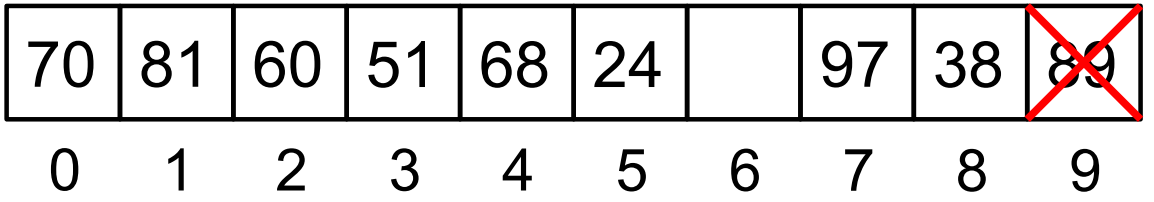
Hash Tables

- Another operation that we'd like:
  - Errrm... Are we sure?
  - Let's try one — let's remove 89...



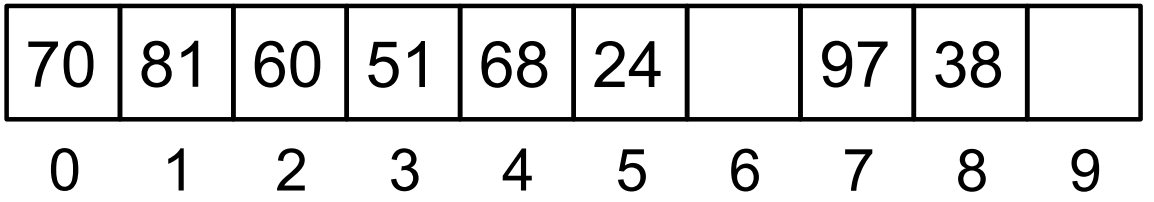
Hash Tables

- Another operation that we'd like:
  - Errrm... Are we sure?
  - Let's try one — let's remove 89... We locate its bin, 9, and erase it (search forward if needed):



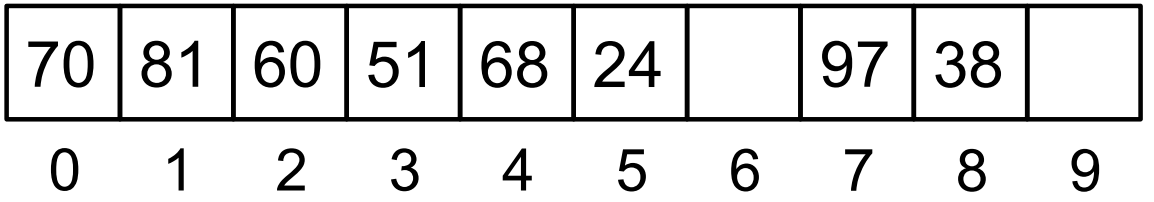
Hash Tables

- Another operation that we'd like:
  - Worked like a charm, right?



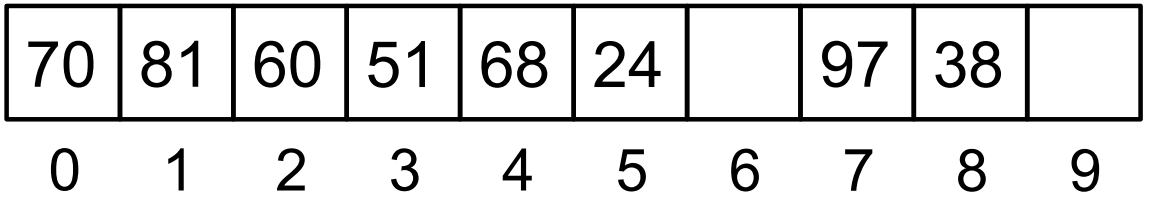
Hash Tables

- Another operation that we'd like:
  - Worked like a charm, right?
    - Really?



Hash Tables

- Another operation that we'd like:
  - Worked like a charm, right?
    - Really?
    - Hint: where's 68?



Hash Tables

- The positions of elements depend on chunks of occupied bins (with no holes).
- Opening holes on those chunks is likely to cause trouble (in this case, elements that are in the hash table can no longer be found).

Hash Tables

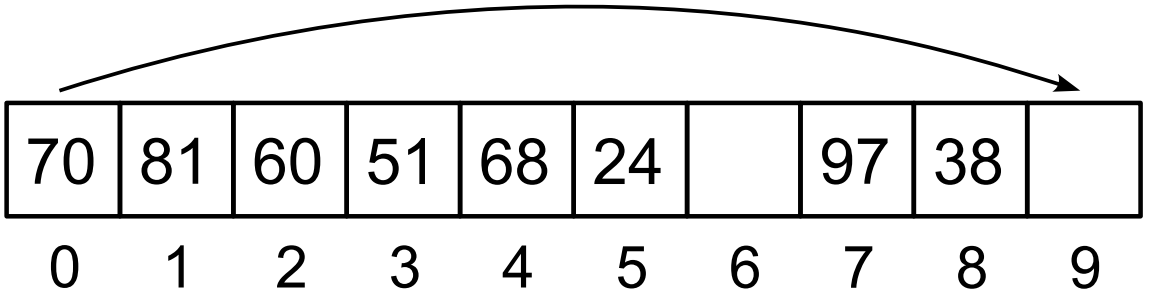
- So, what do we do?

Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?

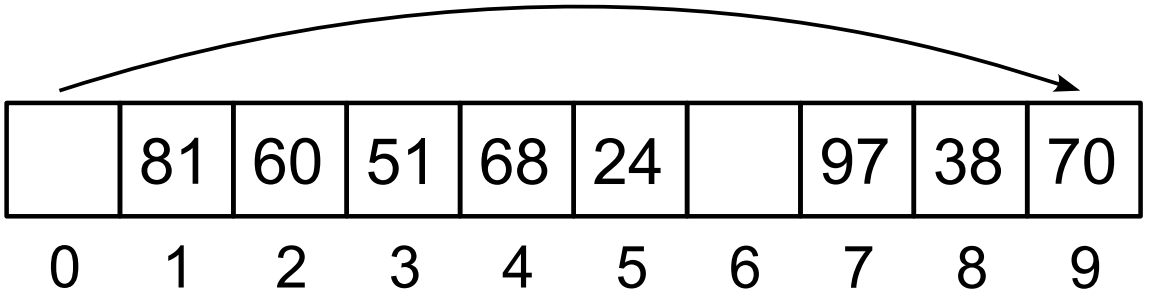
Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?



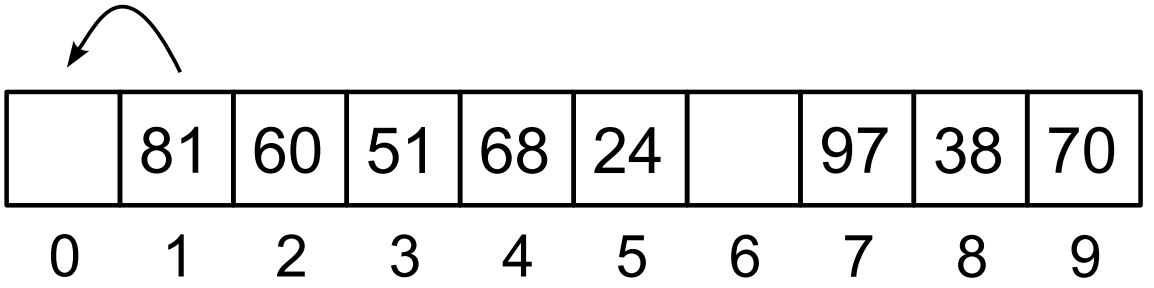
Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?



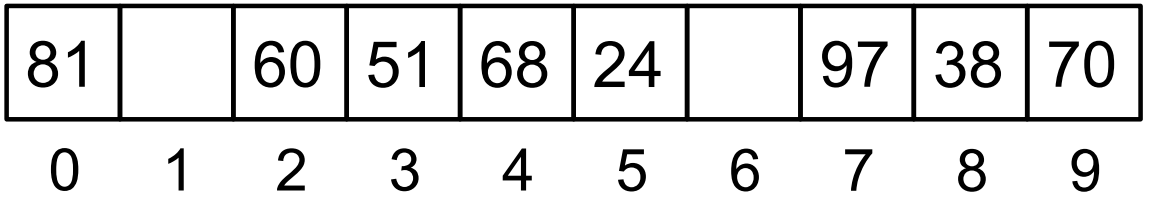
Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?



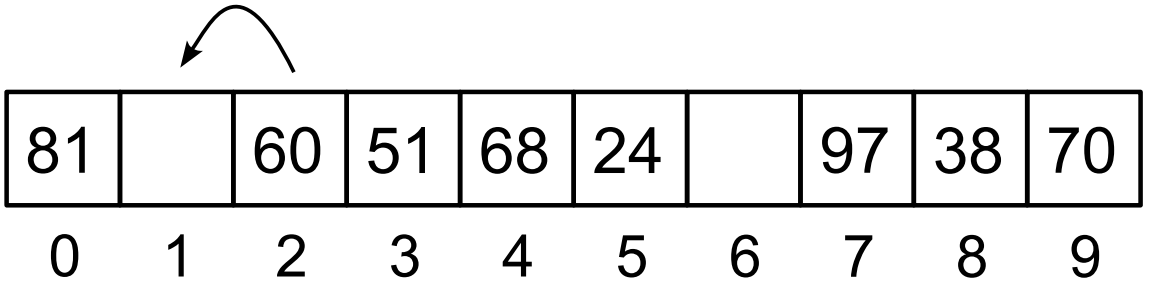
Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?



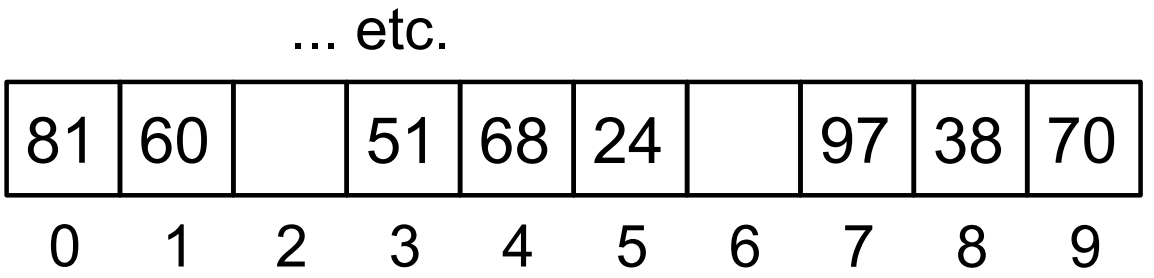
Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?



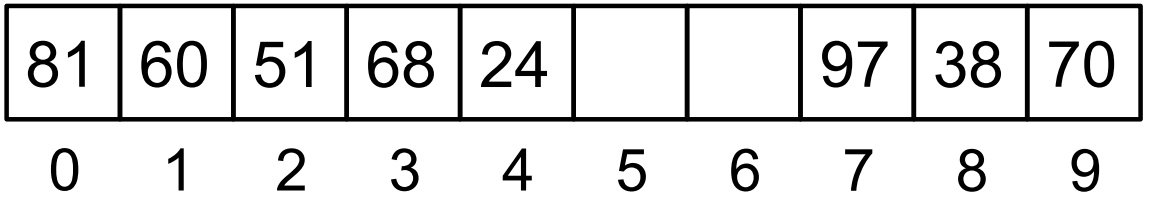
Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?



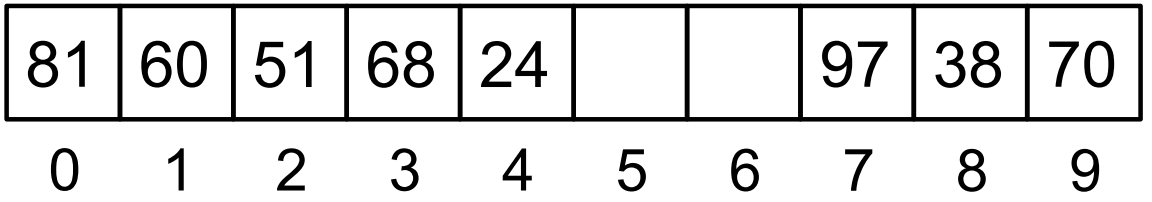
Hash Tables

- So, what do we do?
- How about we shift the elements that follow?
  - In this example, the table is almost full, so it looks like a linear time operation, but in practice, it should be just a few elements, right?



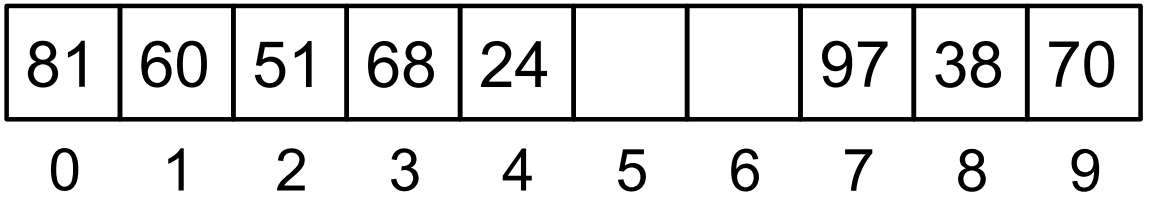
Hash Tables

- Perfect! Now 68 can be found again...



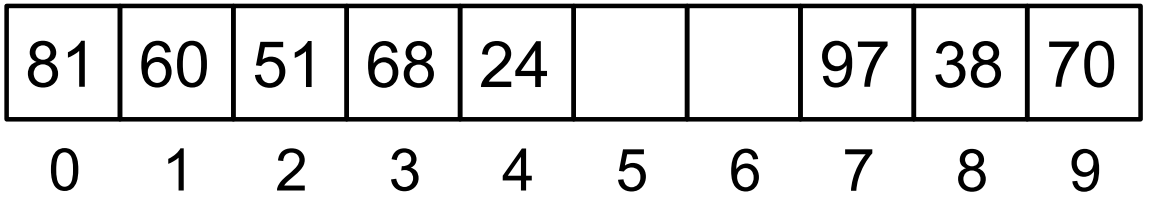
Hash Tables

- Perfect! Now 68 can be found again...
- Errm... Where's 70??



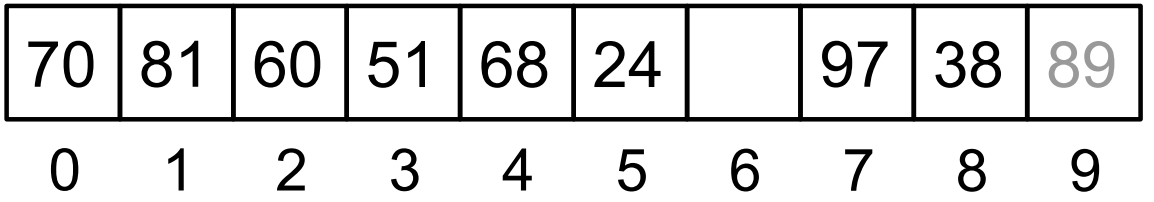
Hash Tables

- Perfect! Now 68 can be found again...
- Errm... Where's 70??
  - In fact, any element that was in the bin corresponding to its hash would now become impossible to find!



Hash Tables

- So, we undo and forget about this idea of shifting the elements that follow...
- Then... what do we do??



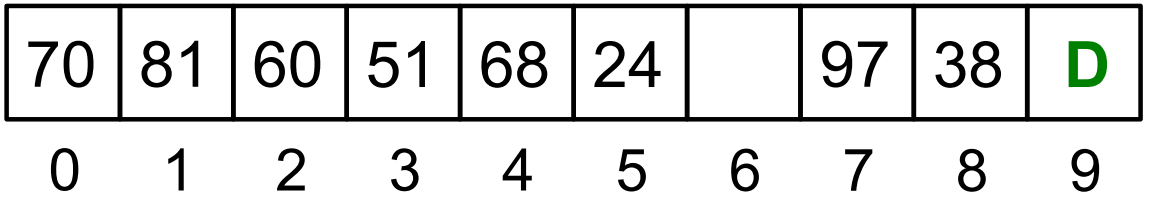
Hash Tables

- Class activity (hopefully 5 to 10 minutes ± ϵ):
  - Team up (groups of two to four people, maybe?) and brainstorm to try to come up with a solution
    - There are several possible solutions, not all of them good and efficient, but still valid! Hopefully some of you guys will find some of the efficient ones?

70	81	60	51	68	24		97	38	89
0	1	2	3	4	5	6	7	8	9

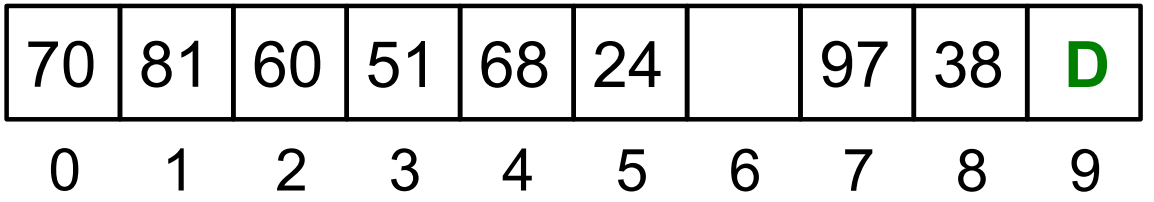
Hash Tables

- Several possible solutions:
  - Given that the problem is opening a hole where none is supposed to be, then do not leave a hole to begin with!
  - Mark the bin with a special value denoting *Deleted*



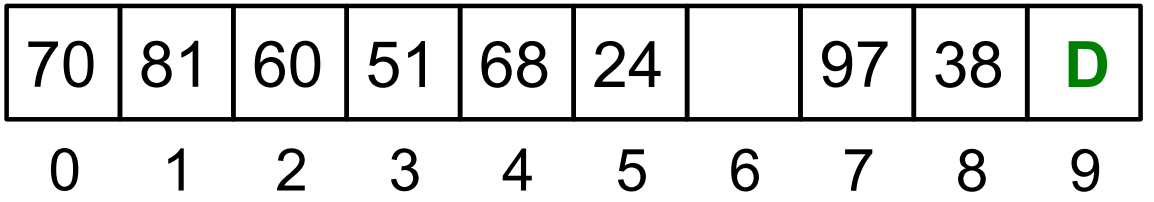
Hash Tables

- Several possible solutions:
  - The idea is that when searching, D counts as an occupied cell, but when inserting or iterating over all values, D counts as an empty cell!



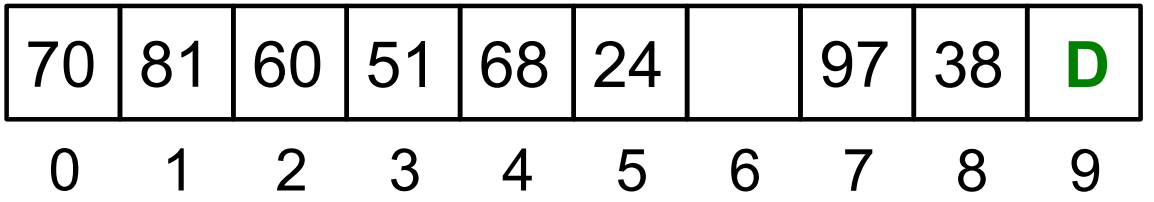
Hash Tables

- Several possible solutions:
  - How to get a suitable value to represent this D may or may not be easy.
  - For example, if we know we're storing positive values, then any negative value could denote this



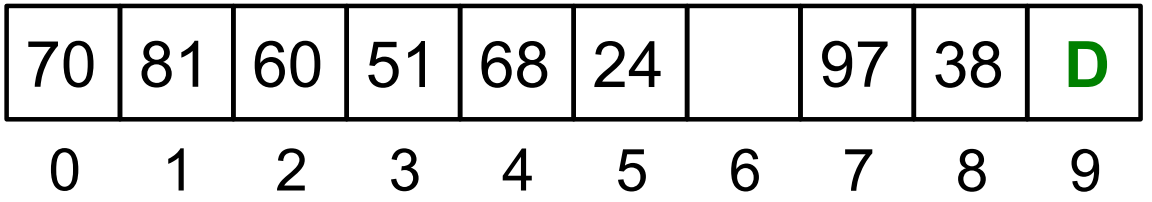
Hash Tables

- Several possible solutions:
  - And BTW, wasn't this already a problem, given that we need to distinguish *empty* bins?
  - If the bins are int and *any* value is a plausible element being stored, how do we distinguish an empty bin?



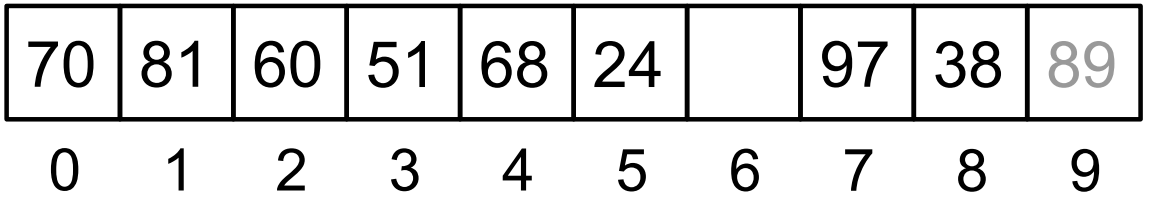
Hash Tables

- Several possible solutions:
  - A big problem with this solution is that if we have many insertions and removals, then the table is excessively populated even if few actual values are stored — making searches far less efficient than they could (should) be!



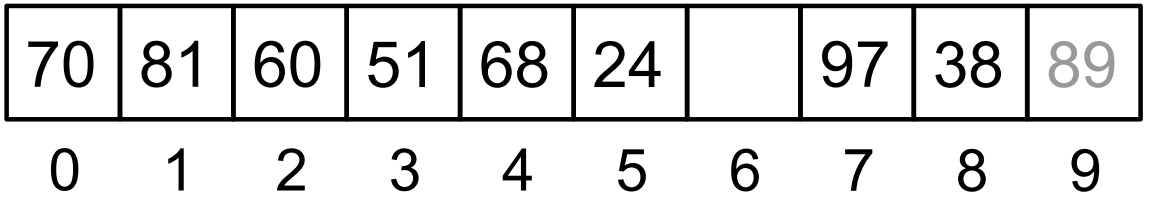
Hash Tables

- Several possible solutions:
  - Another solution is to reallocate and just copy all the elements without the one being deleted.
    - This definitely works, but it is too inefficient — *every* removal takes linear time!!



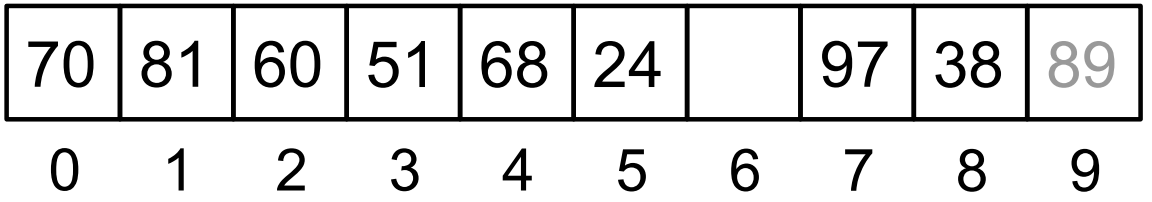
Hash Tables

- Several possible solutions:
  - What we really want to do is scan forward from the hole, looking for elements that could take that position, and move them to it.



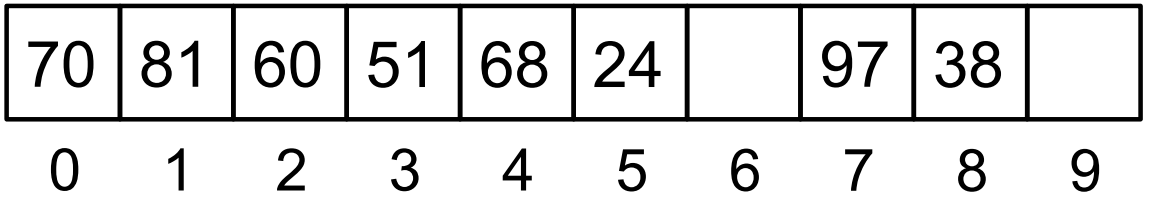
Hash Tables

- Several possible solutions:
  - What we really want to do is scan forward from the hole, looking for elements that could take that position, and move them to it.
    - Of course, we do this in loop, since moving the element creates a new hole where it was



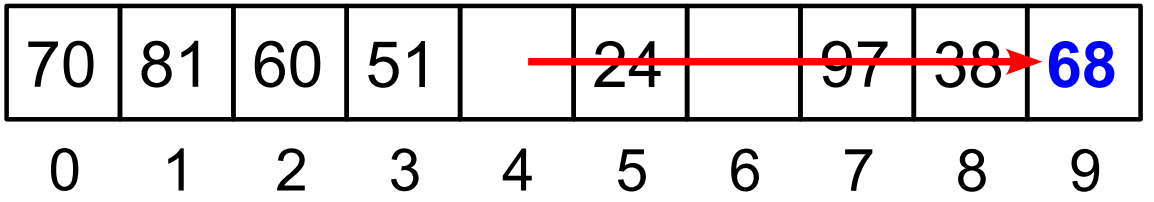
Hash Tables

- Let's see how this one works:
  - We can't move 70, or 81, or 60, or 51 (*why?*)
  - But we can move 68



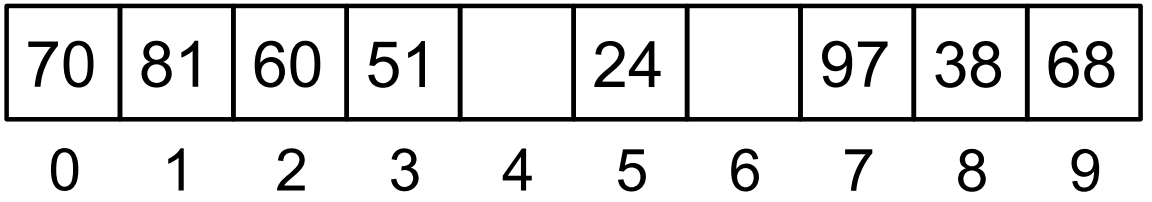
Hash Tables

- Let's see how this one works:
  - We can't move 70, or 81, or 60, or 51 (*why?*)
  - But we can move 68
    - Notice that we're really moving it “backwards” — but in a circular way, so from bin 0 we cycle to bin 9.



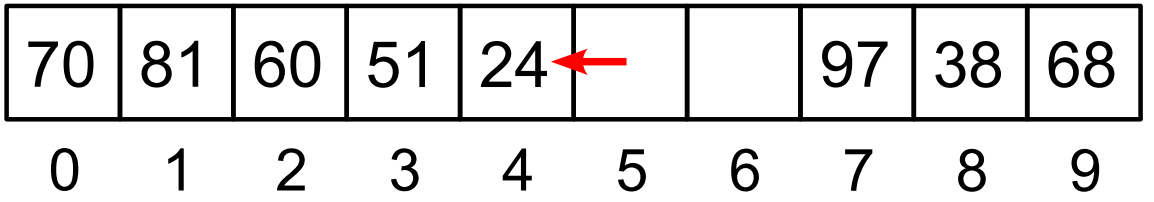
Hash Tables

- Let's see how this one works:
  - Now the hole is at bin 4, so we search forward looking for a value that can be moved to bin 4:



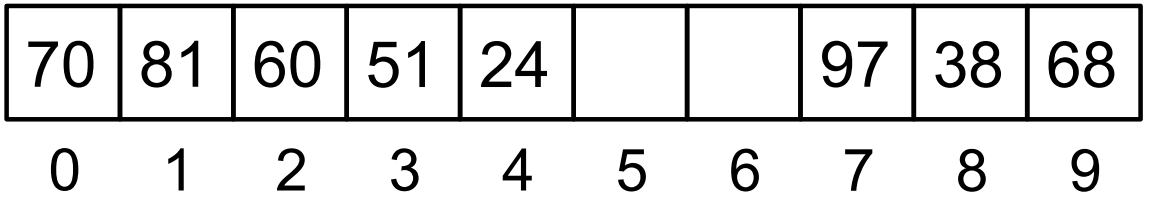
Hash Tables

- Let's see how this one works:
  - Now the hole is at bin 4, so we search forward looking for a value that can be moved to bin 4:
    - Clearly, 24 can be moved to bin 4:



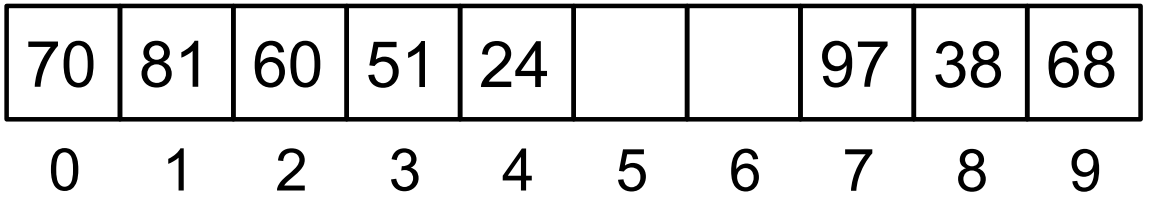
Hash Tables

- Let's see how this one works:
  - Now the hole is at bin 5 (well, not really, but in principle it could be). So, we search forward looking for a value that can be moved to bin 5.
    - Since bin 6 is empty, we are done, and now the table is in a consistent (valid) state.



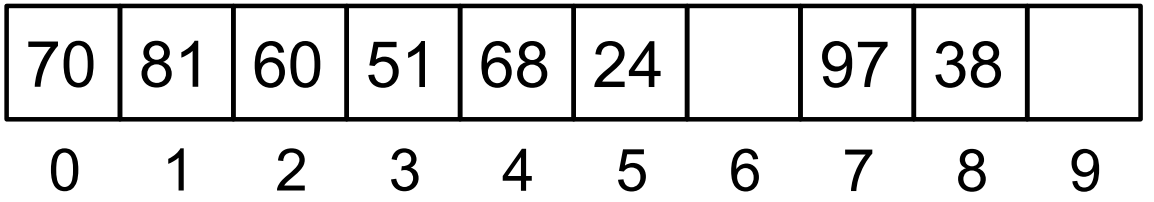
Hash Tables

- For fun:
  - How do we tell (and I mean a program that we write) that a value can or cannot be moved to take the position of the hole?



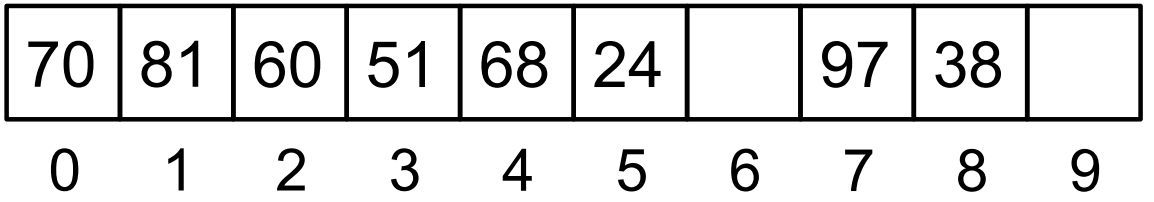
Hash Tables

- For fun:
  - How do we tell (and I mean a program that we write) that a value can or cannot be moved to take the position of the hole?
    - Back to the initial stage (right after removing 89):



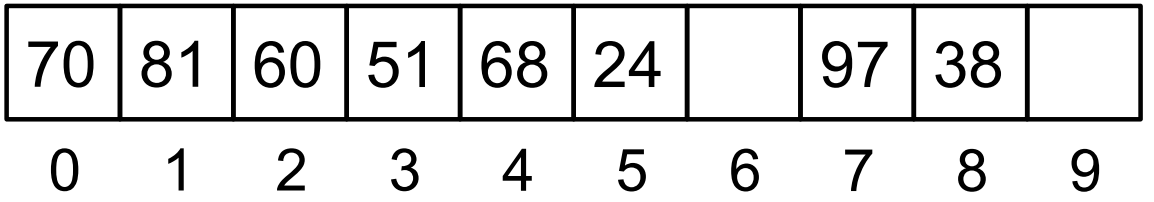
Hash Tables

- Solution 1 (veeeery simple to implement, but not efficient):
  - For each of the scanned elements, we use the lookup function that we have, and if the function says that the element is not in the table, then that element can (has to) be moved.



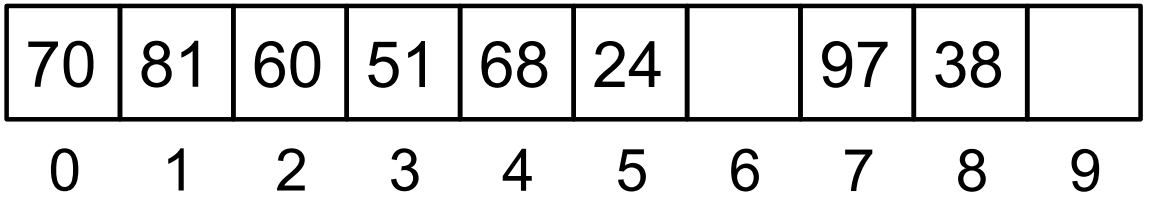
Hash Tables

- Solution 1 (veeeery simple to implement, but not efficient):
  - 70, 81, 60, and 51 are all unaffected by the hole at bin 9 — they're all found when following the lookup procedure.



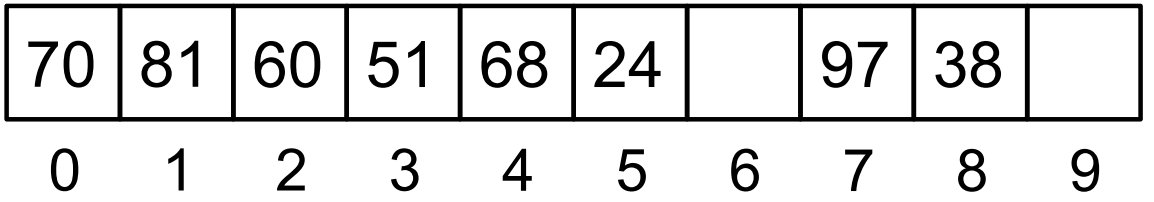
Hash Tables

- Solution 1 (veeeery simple to implement, but not efficient):
  - 68 fails — but since we know that 68 is in the table (we're looking at it!), then it must be that the hole is making the search fail for this value; thus, we move it.



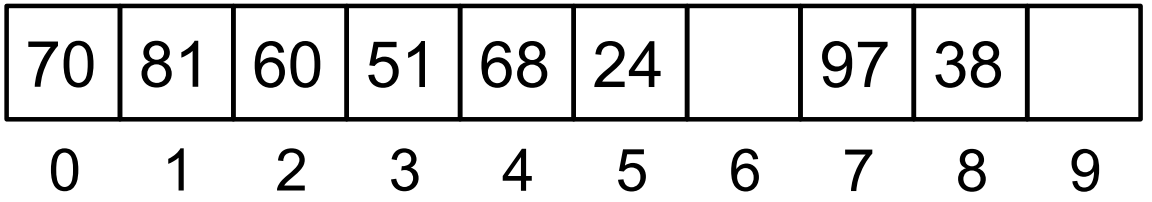
Hash Tables

- Problem: Lookups take linear time in worst case, and we need to do one lookup for each scanned element.



Hash Tables

- Problem: Lookups take linear time in worst case, and we need to do one lookup for each scanned element.
  - What if we could check *in constant time* whether an element should be moved to the position of the hole?



Hash Tables

- Remarkably enough, we can!
  - We take into account the following pieces of information:
    - The position of the hole (denoted by *H*)
    - The position of the element we're considering (denoted *P*)
    - The hash of the element we're considering (denoted h(*x*))

Hash Tables

- Remarkably enough, we can!
  - We take into account the following pieces of information:
    - The position of the hole (denoted by *H*)
    - The position of the element we're considering (denoted *P*)
    - The hash of the element we're considering (denoted h(*x*))
  - We move the element if and only if h(*x*) falls outside the interval (*H*, *P*]

Hash Tables

- Remarkably enough, we can!
  - An alternative way to look at this is: we move the element if (and only if) the sequence of bins going from h(x) to P includes H.

Hash Tables

- Remarkably enough, we can!
  - An alternative way to look at this is: we move the element if (and only if) the sequence of bins going from h(x) to P includes H.
    - This is quite intuitive — when looking up (or when we inserted the element being considered), the linear probing had to probe all the bins from h(x) (where the element in principle would go) all the way to P (which is where it ended up). This must be a contiguous block of occupied bins.
    - The hole would clearly prevent the element from being found.

Hash Tables

- Remarkably enough, we can!
  - An alternative way to look at this is: we move the element if (and only if) the sequence of bins going from h(*x*) to *P* includes *H*.
    - Thus, if the element was found at *P* before opening the hole between h(*x*) and *P*, then it means that the entire sequence from h(*x*) to *P* must be all occupied bins; thus, the element at *P* must be findable at *H*.

Hash Tables

- Remarkably enough, we can!
  - An alternative way to look at this is: we move the element if (and only if) the sequence of bins going from h(*x*) to *P* includes *H*.
    - Thus, if the element was found at *P* before opening the hole between h(*x*) and *P*, then it means that the entire sequence from h(*x*) to *P* must be all occupied bins; thus, the element at *P* must be findable at *H*.
    - BTW, a related detail... we know that the sequence between *H* and *P* is contiguous and have no holes (right? *why?*).

Hash Tables

- We want to do this check in constant time.
  - Not a problem — checking if a number is within a given interval involves two comparisons and one logical and.

Hash Tables

- We want to do this check in constant time.
  - Not a problem — checking if a number is within a given interval involves two comparisons and one logical and.
  - However, careful: the positions of bins work in a circular way, so being “before” or “after” can be tricky.
  - We consider two situations:
    - $H < P$ (less than in an “absolute” way)
    - $H > P$
      - Notice that $H \neq P$

Hash Tables

- We want to do this check in constant time.
  - The first case, $H < P$, is the "normal" one — we start at the position after the hole and search forward, so P is greater than H.
    - In this case, we move the element if $h(x) \leq H$ or $h(x) > P$

Hash Tables

- We want to do this check in constant time.
  - The first case, $H < P$, is the "normal" one — we start at the position after the hole and search forward, so P is greater than H.
    - In this case, we move the element if $h(x) \leq H$ or $h(x) > P$
  - However, when searching forward, P may have cycled back to bin 0, corresponding to the case $P < H$.
    - In this case, we move the element if $H \leq h(x) < P$

Summary

- During today's class, we discussed:
  - Linear probing as a mechanism to deal with collisions.
  - Difficulties associated with this technique.
  - Discussed techniques to efficiently delete elements from the hash table.