Finish up from last time…
(Spelling)

Announcements

Announcements

- A user study!
  - Participants needed…
  - Contact: Shrirang Mare
  - FAQ: http://www.cs.dartmouth.edu/~brace

Participants wanted for a computer science user study

We are conducting a user study about smartphone authentication and we are recruiting volunteers for the user study. The study involves performing simple tasks on a smartphone for about 30 minutes. You will be compensated for your time. The study will take place in Sudikoff (computer science building). Send an email to uauth.study@gmail.com or shrirang@cs.dartmouth.edu to participate in the study.

For more information on the user study visit http://www.cs.dartmouth.edu/~brace

Do consider volunteering for the user study. You'll be helping Science! :)

Announcements (cont.)

- Caleb An (Xiaole.An.15@dartmouth.edu)
- User study — use their app…
  - Payment: get some extra help from 2 CS 10 experts!
  - Android phones only
  - Wednesday (today), 3-6pm

Hashing:
Hash Tables & HashFunctions

Hash Tables

Ex. — Hashing based on last 2 digits of Phone

- Sears catalog store in West Leb. used to keep track of catalog orders to be picked up in a set of 100 pigeonholes (boxes) numbered 0 - 99.

- Distribute orders for each person into one of the 100 boxes based on the *last* two digits of a given phone number.
  - ex. f(603-441-5555) = box 55
  - ex. f(603-846-4332) = box 32
  - etc.
- There may be some overlap, but probably not much — the clerk could search through the small set of orders in a given box.
- The trick is to find a *function* which distributes items fairly evenly. In this case, the *last* 2 digits of a Phone # worked well…
- **Question:** Why would the *first* 2 digits of Phone # be a bad idea?

Hash Tables

- One of the most efficient data structures for implementing a map.
- Most used in practice… :)
- The structure is known as a ***hash table****.*
- So far we've discussed maps as being something that holds a collection of key/value pairs. The *key* allows us to locate things in the structure. In an abstract way, we can think of ***keys*** as ***addresses***, and the corresponding ***values*** as being the ***contents*** at that address.
- Thus, In a very simple case, we could…
  - … think of the structure as an *array*
  - … think of an integer index as the *key*, and
  - … think of the data at some index as the corresponding *value*.

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Fig. from GTG

Lookup Table
N = 11

(1, D)
(3, Z)
(6, C)
(7, Q)

Hash Tables

- Good — basic map operations (get, put, remove) are O(1).
- If all keys mapped to their own index, we'd be done!
  - That would be like all Sears customers having the last two digits of their phone numbers be unique.
  - *But we aren't so lucky…*
- When multiple keys map to the same table index, we have a **collision**.
- You might be thinking: "with 100 slots in the table, we'd have to get close to 100 keys inserted into the table before we'd be likely to have a collision."
  - This isn't actually true…

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Fig. from GTG

Lookup Table

N = 11

(1, D)
(3, Z)
(6, C)
(7, Q)

Birthday Paradox

If you randomly select people, how many do you have to select before there is at least a 50% chance that at least two of them have the same birthday?

- You might think that the number is close to 365…?
- Or maybe it's around half of 365: 183…?
- Think smaller! Much, much smaller. In fact, it's just 23!
- [**DEMO**]
- There is a link in the notes if you want to dig deeper into the math :)
  At its core, there is just some clever probability theory at play.

Hash Tables

- The moral of the story is that collisions happen and we have to deal with them.
- **Big Idea:** To address this, we can think of each space in the table as consisting a "bucket", or collection, of entries.
- Some challenges (moving forward):
  - N would potentially need to be BIG to "prevent" collisions. Even then, it might happen still! Thus, we need a way to handle collisions…
  - So far we've talked about using integers as keys into the table — we don't, in general, require keys to be integers. In fact, the novel concept behind a hash table is the use of a ***hash function*** to map some (arbitrary) key to an index in a table.
  - We'd like our hash function to distribute entries fairly evenly throughout the table, but it could happen that more than one key maps to the same index!

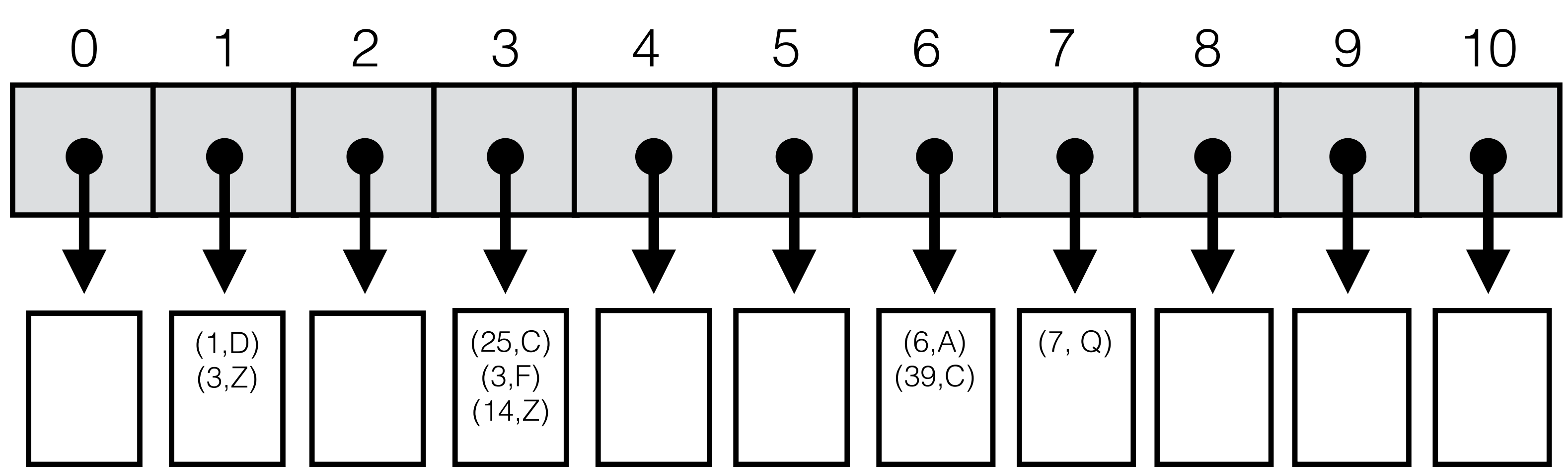


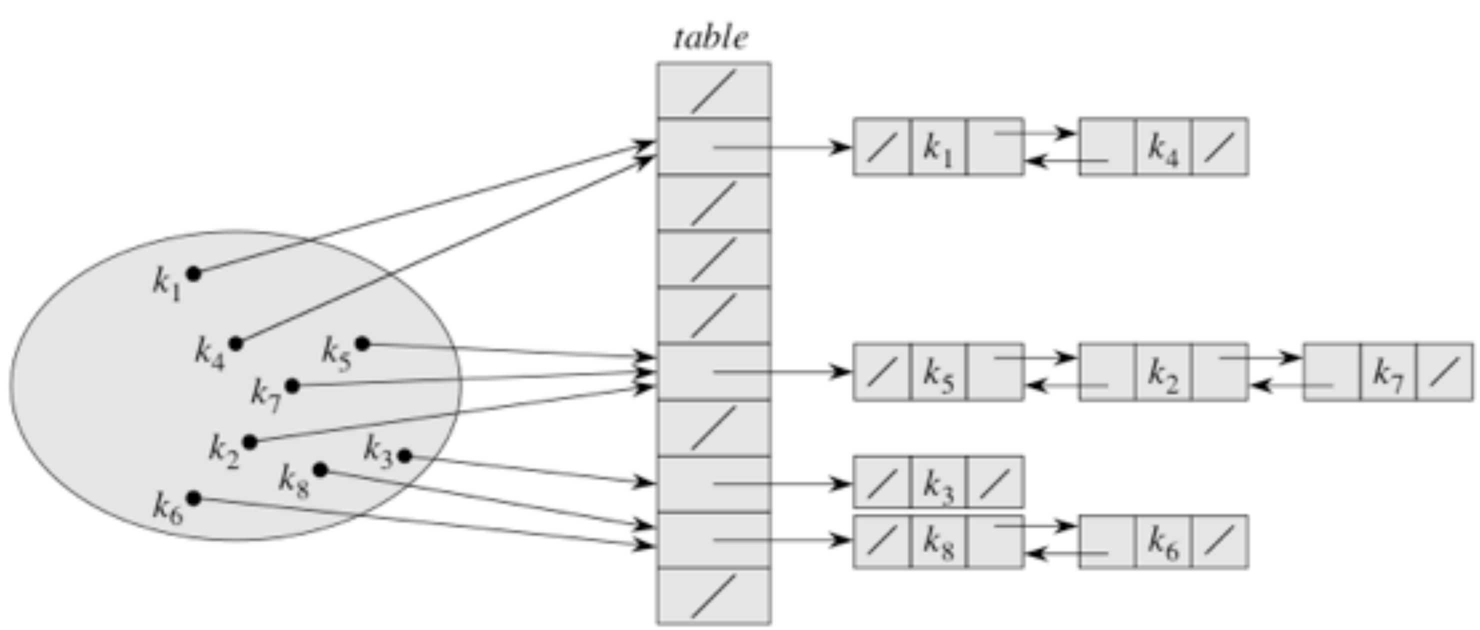
Fig. from GTG

Handling Collisions

There are a couple of ways that we can handle collisions.
Next, we will look at a few of them.

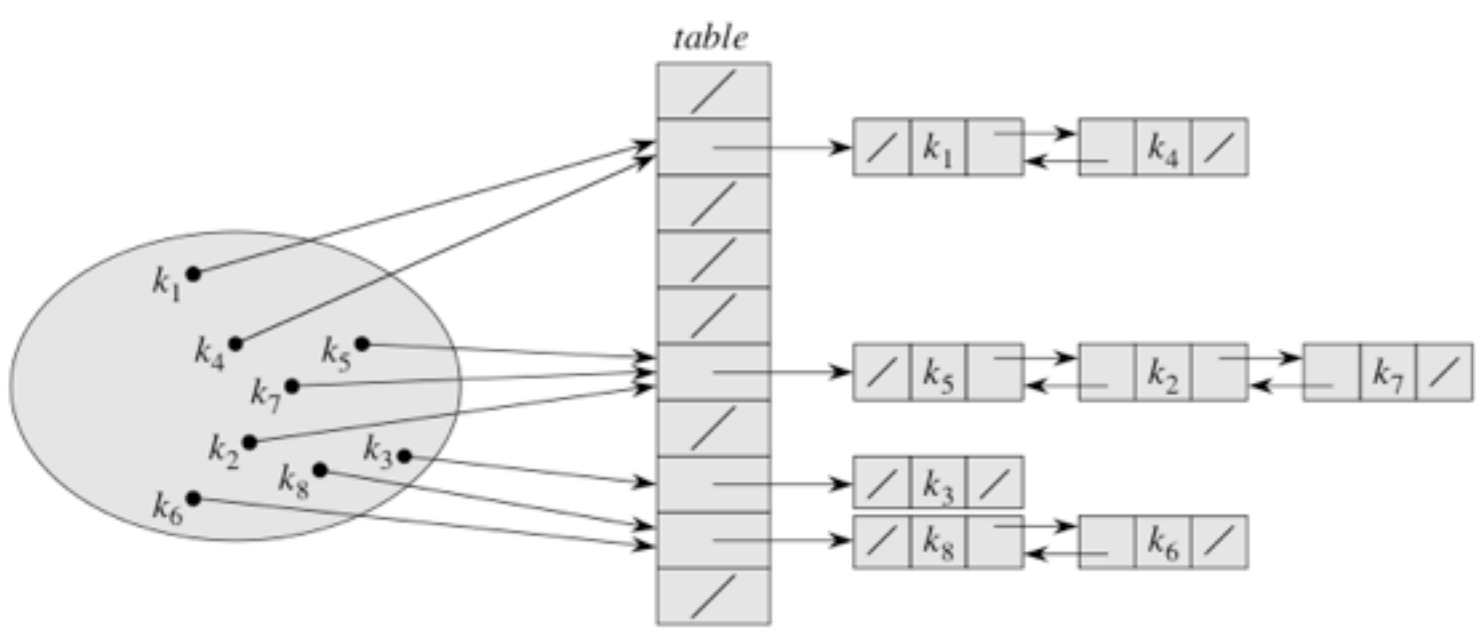
Chaining

- **Solution**: Instead of storing each element directly in the table, each slot in the table references a linked list.
- The linked list for slot *i* holds all the keys *k* for which *h(k) = i*.
- Spaces that have yet to have keys mapped to them can simply be *null*.
- Visually we show a DLL, but typically a SLL suffices.



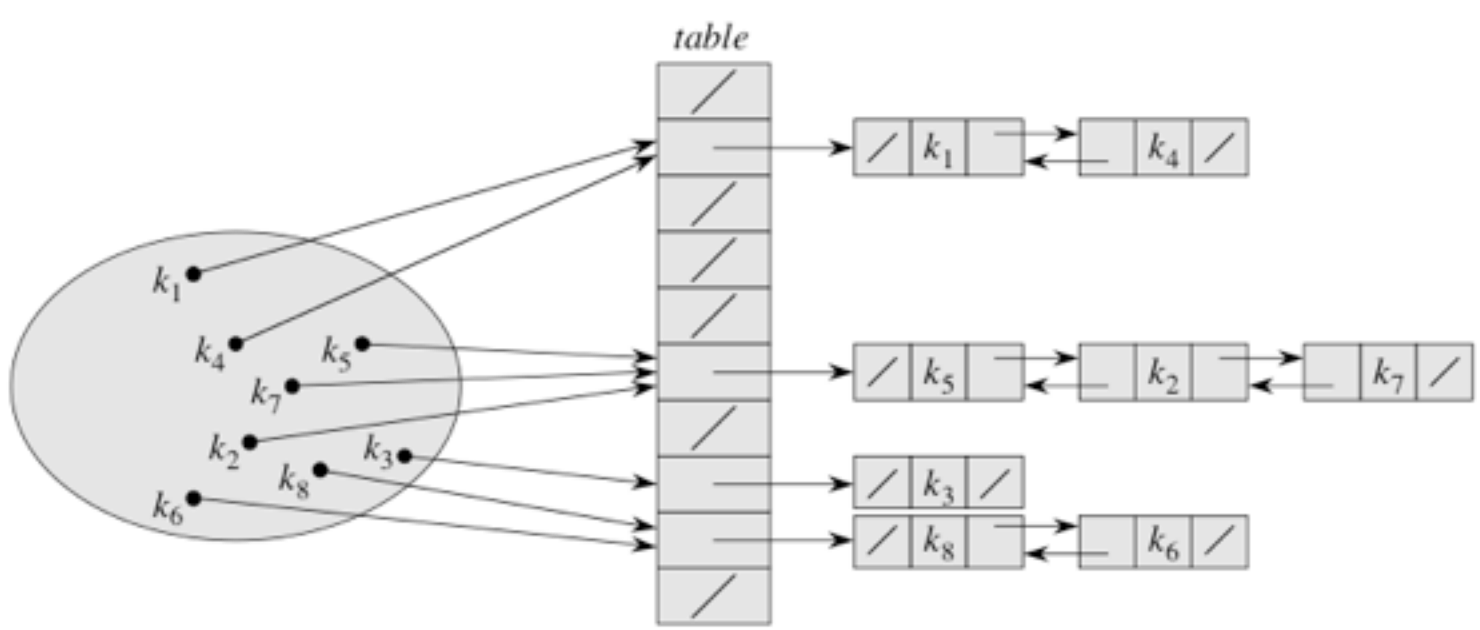
Chaining

- **Question**: How many items do we expect to look at when searching for a item?
- For **unsuccessful** search (not in the map or set), we would look at everything in the appropriate linked list.
- But how many elements is that?
  - If the table has ***N*** slots and there are ***n*** keys stored in it
  - there would be ***n/N*** keys per slot on average, and hence
  - ***n/N*** elements per list on average. We call this ratio the ***load factor***, and we denote it by **α** (alpha).
- Operations are O(1) expected time so long as α is O(1).



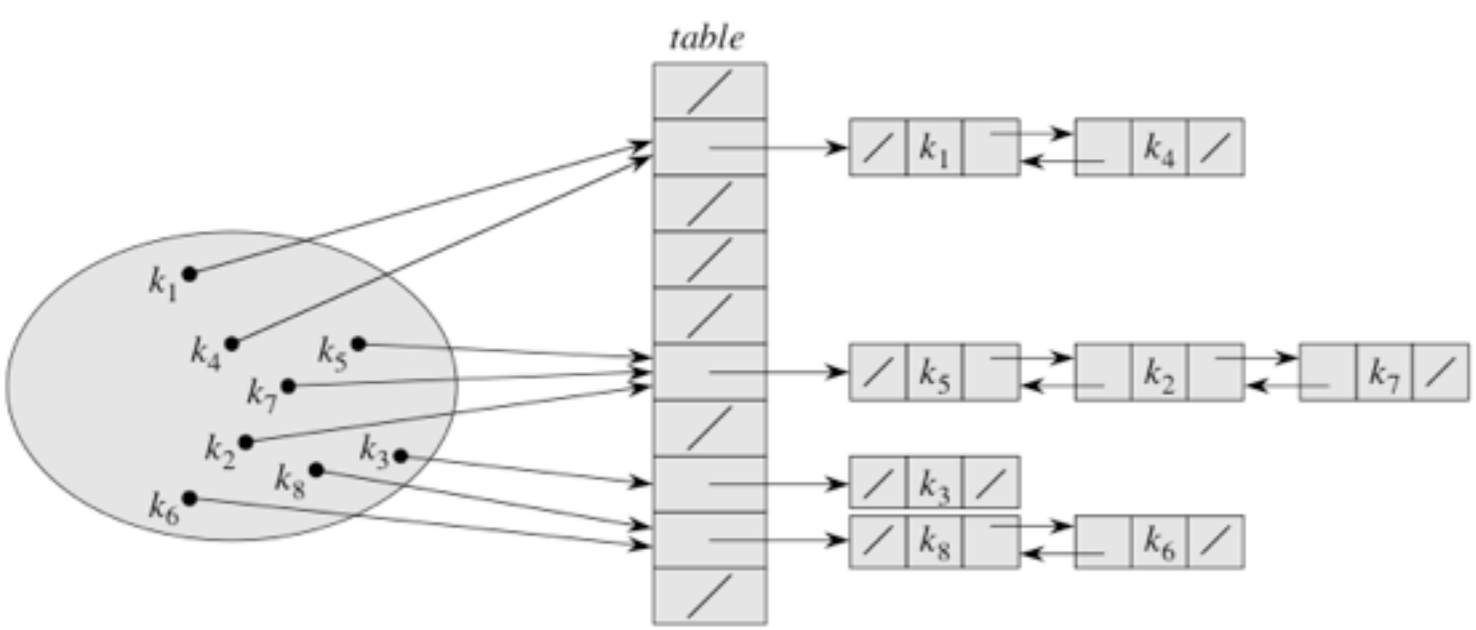
Chaining

- If the hash function did a ***perfect job*** and distributed the keys perfectly evenly among the slots, then each list has α elements
  - For **successful** search we know we will find the item, and on average we will go through half the list before we do so. So successful search takes about $1 + \alpha/2$ comparisons
  - **Unsuccessful** search takes α comparisons, on average. If $N = O(n)$, then this is a constant.
  - Either way, search takes $\Theta(1 + \alpha)$
    - Why "1 + "? Because even if $\alpha < 1$, we have to account for the time computing the hash function h, which we assume to be constant, and for starting the search.



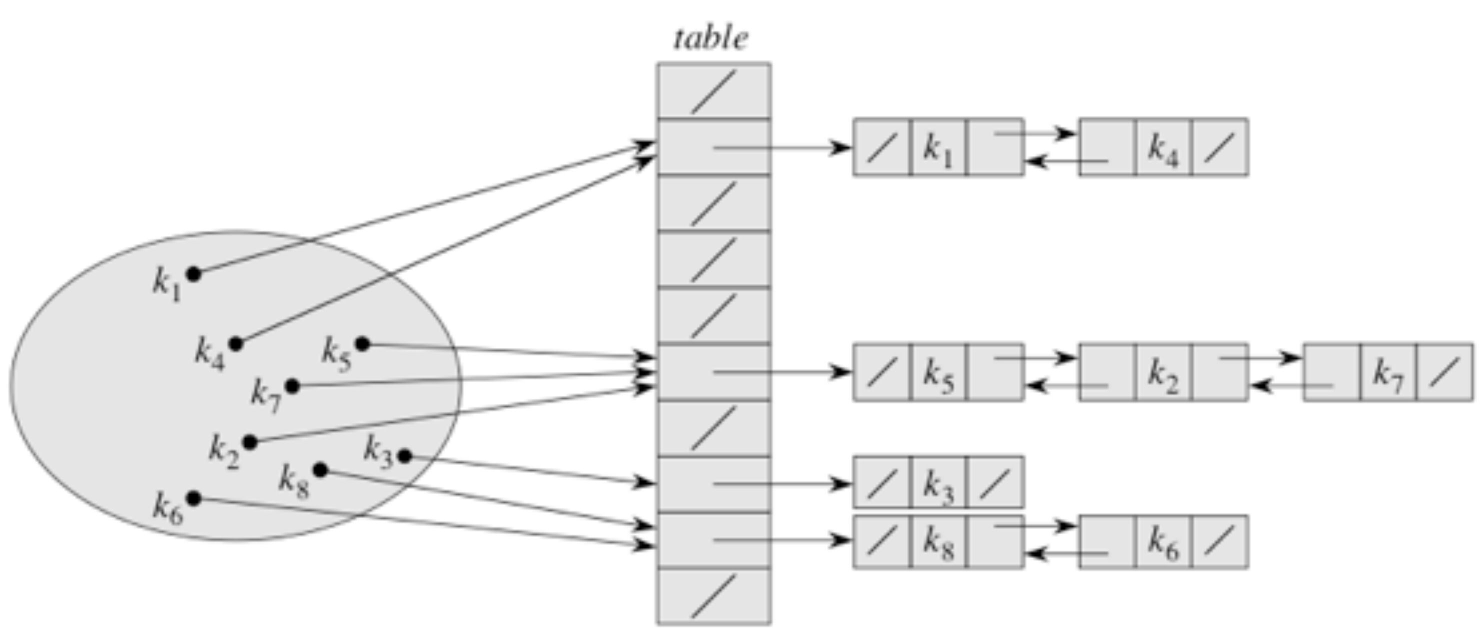
Chaining

- Now, what if the keys are ***not perfectly*** distributed?
- Things get a little trickier, but we operate on the assumption of ***simple uniform hashing***, where we assume that any given key is equally likely to hash into any of the *N* slots, without regard to which slot any other key hashed into.
- Under the assumption of simple uniform hashing, any search, whether successful or not, takes Θ(1 + α) time on average.



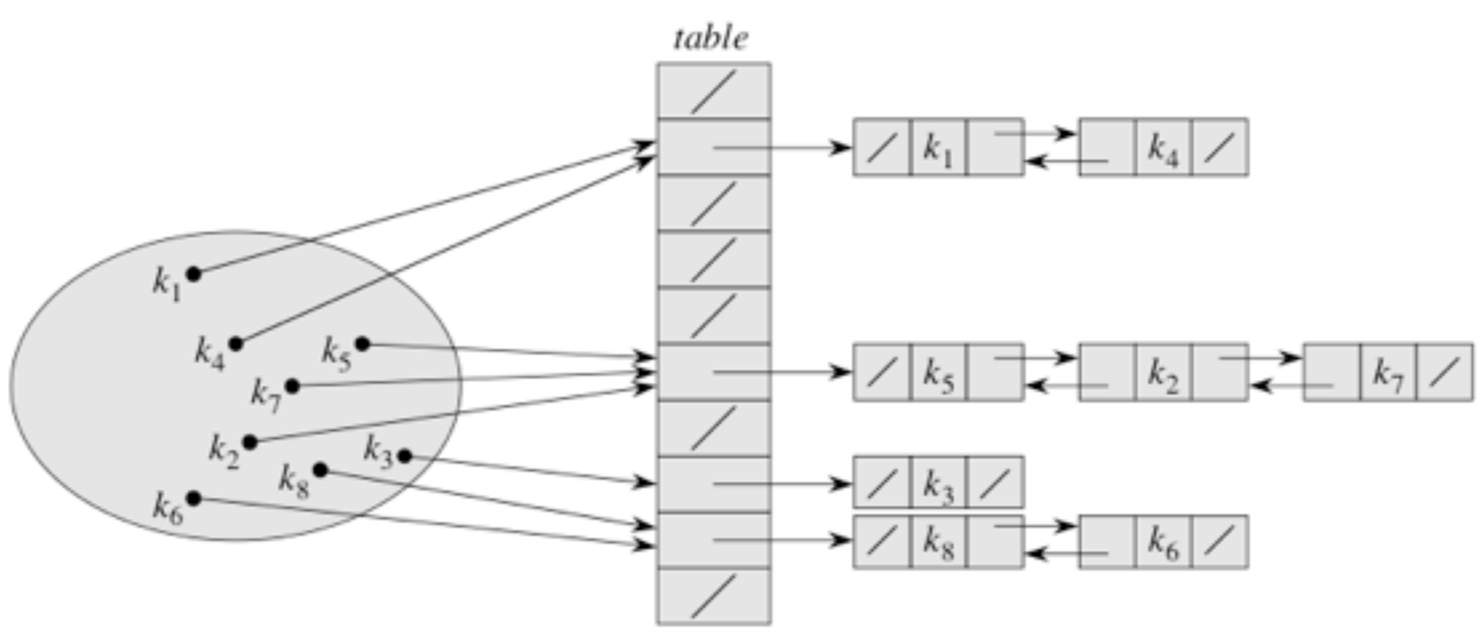
Chaining

- **Worst case** is bad — occurs when all keys hash to the same slot.
- Highly unlikely (though not impossible if your hash function is bad…)
- If an adversary puts *n*N* items into the table then one of the slots will have at least *n* items in it (pigeonhole).
  - Universal hashing is computes a different hash code every time you run the program.
- Worst case time for an ***unsuccessful search*** is Θ(n), since the entire list of n elements has to be searched
- For a ***successful search***, the worst-case time is still Θ(n), because the key being searched for could be in the last element in the list.



Chaining

- **Inserting**
  - Compute h(k) to find bucket, insert into LL —> Θ(1) time
- **Deleting**
  - If we assume that we have already searched for it and have a reference to its LL node, and that the list is a DLL, then removing takes Θ(1) time. Again, that's *after* having paid the price for searching.



Chaining: Wrap-up

- **Question:** If $n > N$, search time goes up… how to avoid this?
- **Answer:** Grow the table when it become "full" (similar to ArrayList).
    - What does "full" mean? —> double size when α exceeds 0.75.

- **Question:** How well are we using the table?
- **Answer:** Not necessarily very well at all…
    - Out table potentially has lots of empty slots.
    - We are (potentially) wasting lots of space!
    - If memory is valuable (as it is on embedded systems/mobile devices), we want to make better use of memory…

Open Addressing

- The second way to handle collisions is called **open addressing**.
- The idea is to store everything in the table itself, *even when collisions occur*. Thus, **there are no linked lists**!
  - We have the same problem as before though — collisions *can* happen so how do we deal with this?!
- Open addressing requires that the **load factor** be always ***at most*** 1, since entries are stored directly in the table itself (not in aux. data structures).

We will now look at a few of the collision resolution techniques for open addressing. The textbook has helpful and interesting details so please check those out.

Open Addressing: Linear Probing

- **Question:** How can we store everything in the table even when there's a collision?
- **Solution:** One simple scheme is called ***linear probing.***
- Suppose…
  - want to **insert** key *k* and that *h(k) = i — initially, the table is empty, go ahead and insert!*
    - *ex. h(13) = 2, h(26) = 4, h(21) = 10, h(5) = 5.*
  - linear probing suggests that, if there is a collision we start *probing for the next empty slot* (i.e., *i+1, i+2, …)*

0	1	2	3	4	5	6	7	8	9	10
		13		26	5					21

Fig. from GTG

Open Addressing: Linear Probing

- **Question:** How can we store everything in the table even when there's a collision?
- **Solution:** One simple scheme is called ***linear probing.***
- Suppose…
  - want to **insert** key *k* and that *h(k) = i — initially, the table is empty, go ahead and insert!*
    - *ex. h(13) = 2, h(26) = 4, h(21) = 10, h(5) = 5.*
  - linear probing suggests that, if there is a collision we start *probing for the next empty slot* (i.e., *i+1, i+2, …)*
    - *ex. h(37) = 4. Slot 4 is full already, so check slot 5. Slot 5 is full already, so check slot 6. Slot 6 is empty — insert here!*

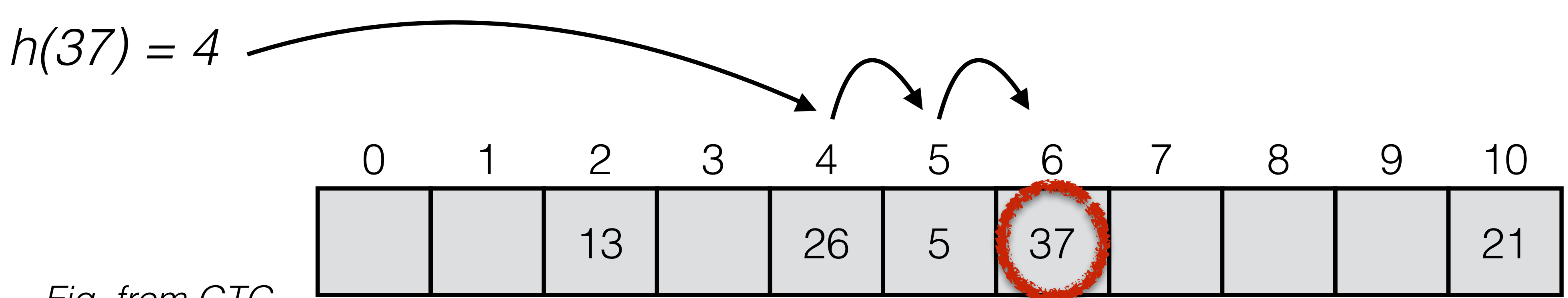


Fig. from GTG

Open Addressing: Linear Probing

- **Question:** How can we store everything in the table even when there's a collision?
- **Solution:** One simple scheme is called ***linear probing.***
- Suppose…
  - want to **insert** key *k* and that *h(k) = i — initially, the table is empty, go ahead and insert!*
    - *ex. h(13) = 2, h(26) = 4, h(21) = 10, h(5) = 5.*
  - linear probing suggests that, if there is a collision we start *probing for the next empty slot* (i.e., *i+1, i+2, …)*
    - *ex. h(37) = 4. Slot 4 is full already, so check slot 5. Slot 5 is full already, so check slot 6. Slot 6 is empty — insert here!*
    - *ex. h(16) = 4.*

**NOTE: If we reach the end of the array, we wrap around and start continuing probing from the beginning of the array.*

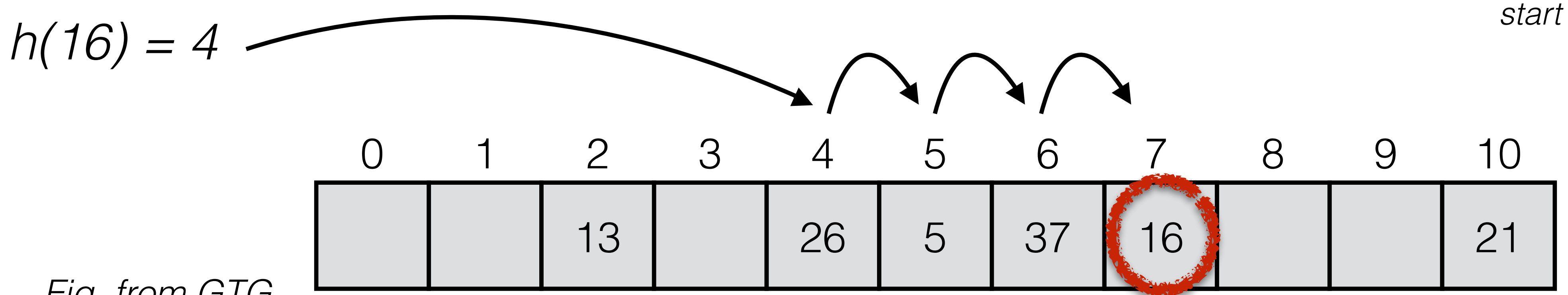


Fig. from GTG

Open Addressing: Linear Probing

- So long as the load factor, α, is less than one (i.e., n < N), we are guaranteed to find an empty spot (eventually) where we can insert.
- If α reaches 1, we must grow the table size and rehash the entries.

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37	16			21

Fig. from GTG

Open Addressing: Linear Probing

- **Searching** with linear probing works much the same.
- Compute *h(k) = i*, and search slots *i, i + 1, i + 2, …,* wrapping around at slot *N – 1* until either we find key *k* or we hit an empty slot.
- If we hit an empty slot, then key *k* was not in the hash table.
- [*Discuss examples in class if necessary…*]

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37	16			21

Fig. from GTG

Open Addressing: Linear Probing

- **Searching** with linear probing works much the same.
- Compute $h(k) = i$, and search slots $i, i + 1, i + 2, \ldots,$ wrapping around at slot $N - 1$ until either we find key k or we hit an empty slot.
- If we hit an empty slot, then key k was not in the hash table.
- [*Discuss examples in class if necessary…*]

Linear probing has some nice ideas, but there are some glaring problems…

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37	16			21

Fig. from GTG

Open Addressing: Linear Probing

Problem

- **Question:** how to remove a key from the hash table?
  - Can’t just remove it… why not?
- Ex. Suppose h(26) = h(37) = h(16) = 4
  - Remove *k = 5* — now there is a “hole” at index 5.

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37	16			21

Fig. from GTG

Open Addressing: Linear Probing

<u>Problem</u>

- **Question:** how to ***delete*** a key from the hash table?
  - Can't just remove it… why not?
- Ex. Suppose h(26) = h(37) = h(16) = 4
  - Remove *k = 5* — now there is a "hole" at index 5.
  - Now suppose we search for *k = 37* ?
    - compute hash h(37) = 4
    - not there, probe slot *i+1*.
    - because slot *i+1* (5) is empty, we conclude 37 is not in the table.
    - *Wrong…*

0	1	2	3	4	5	6	7	8	9	10
		13		26		37	16			21

Fig. from GTG

Open Addressing: Linear Probing

Problem

- **Question:** how can we fix this?
- Need to be able to mark a slot as having held a key, but it has been removed…
- Then, it can be treated as full during ***searching***…
- But, it can be treated as empty during ***inserting***…
- If we remove a lot of keys though, searches start taking longer…

0	1	2	3	4	5*	6	7	8	9	10
		13		26		37	16			21

Fig. from GTG

Open Addressing: Linear Probing

Problem

- There is a bigger problem yet! — ***clustering***
- Longer and longer "runs" of occupied slots build up, increasing the avg. search time.
- Clusters form from…
  - prob. of an empty slot being filled is (t+1)/N, where t is the number of full slots preceding the empty slot.
  - coalescing clusters (two clusters growing together)

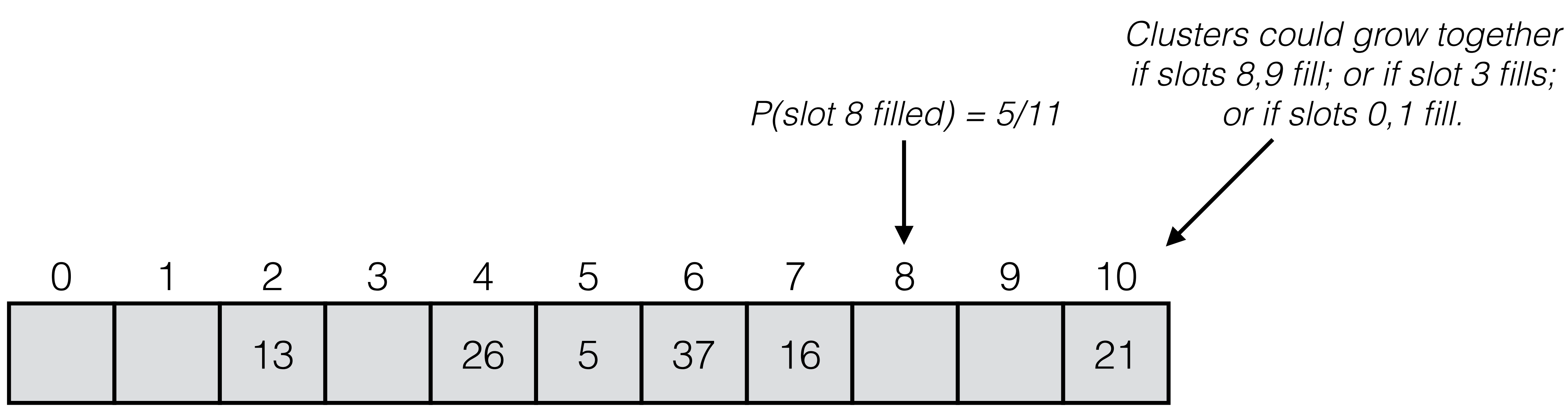


Fig. from GTG

Open Addressing

- Other schemes…
  - **quadratic probing** — similar to linear probing but rather than stepping in a linear fashion, steps are taken as
    - A[(h(k)+f(i)) mod N] for i = 0, 1, 2, …, where f(i) = i^2
    - Suffers from same/similar issues as linear probing
  - **double hashing** — better! Use two hash functions *h1* and *h2* to make a third *h'* which has some nice properties and makes clustering less likely.
    - *h1* — tells us the initial slot to check
    - *h2* — tells us how big of a step to take when probing
  - understanding implications of the **uniform hashing** assumption.
- Book and online notes have more details — check them out.

Hash Functions

Hash Functions

- Ideally, a hash function…
  - Computes the mapping from *keys* to *indices* in a hash table.
    - i.e., *h(k) = index into table —> (k,v) stored at A[h(k)]*
  - Computes this *fast!*
  - Distributes keys fairly *evenly* over the table.
  - Has the property that small changes to an input key result in a different hash code.
- Hash functions often consist of two steps:

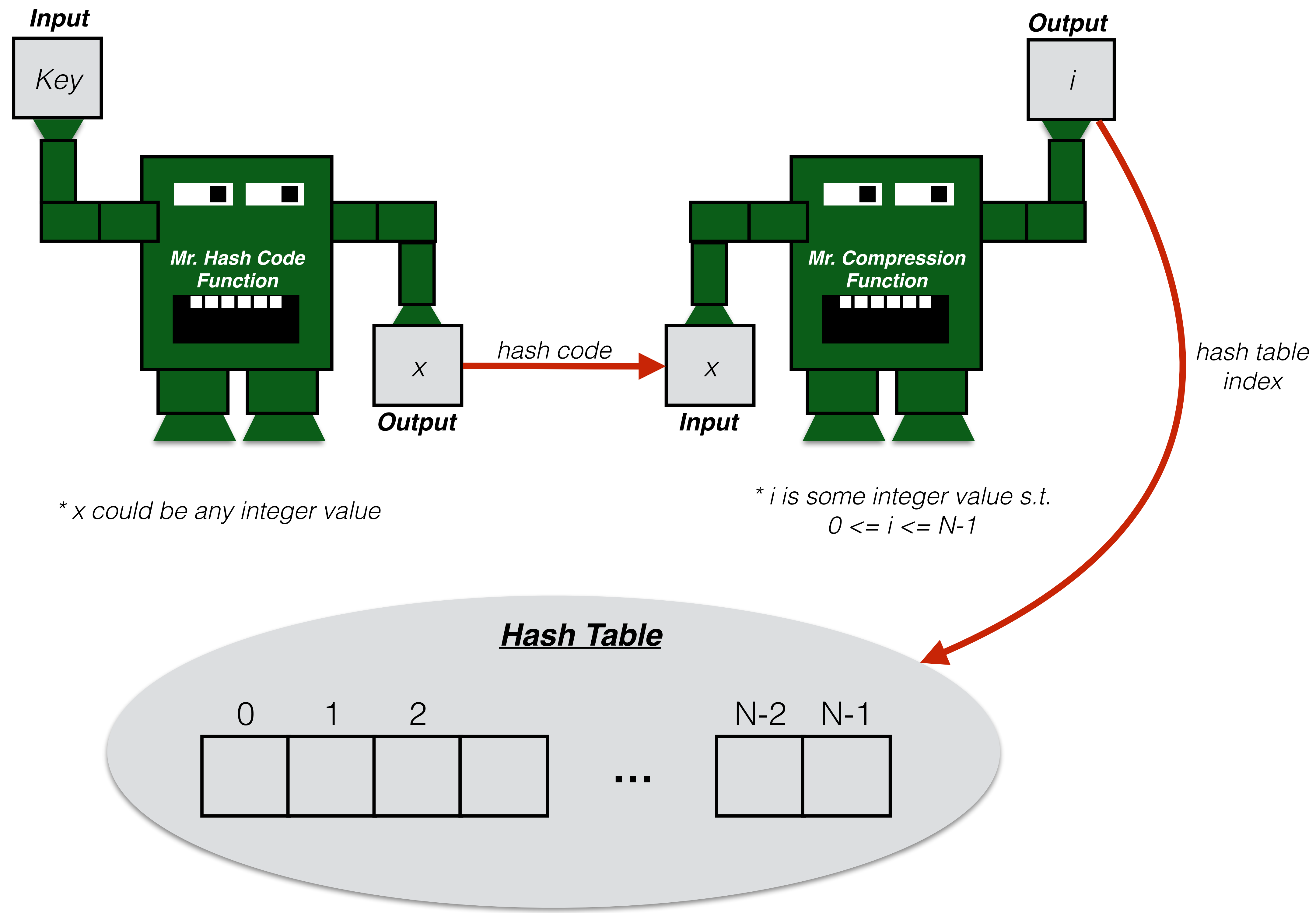
1. Compute a ***hash code*** from an arbitrary object (i.e., the key)
2. Use a ***compression function*** to map the hash code into the bounds of your table.

NOTE: *separating these two components allows you to compute a hash code independent of the table size. This allows for you to have a dynamically sized hash table (like an ArrayList) if you desire such a thing.*

Hash Functions

- Hash functions consist of two primary components…
- **Hash Code**
  - Java, by default, often uses the address in memory for an object.
  - Then this has to be mapped into the table…
- **Compression Function**
  - <u>Simple:</u> mod number by the size of the table
    - This works well if table size (i.e., N) is *prime* — not so well otherwise
  - <u>Better:</u> “MAD” method (Multiply-Add-and-Divide)
    - ((ai + b) mod p) mod N
      - i is the hash code,
      - p is prime > N
      - a,b are chosen at random (between 1 and p-1)
      - probability of collision is 1/N
  - NOTE: Java takes care of this stuff under the hood

Hash Functions



Hash Functions: Caution!!!

- While Java typically handles the *compression function* stuff for us, we are responsible for creating a “good” hash code.
- In particular, if you define **equals()** for an object it is *very important* that you override **hashCode()** so that two items considered equal have the same hash code.
  - if you don’t do this, then you’ll look for the item in the wrong index of the table!

Hash Functions

- **Q:** So how do we compute good hash codes for *composite* objects?! I.e.,
  - objects consisting of several instance variables, or
  - are Strings, or
  - are arrays or …
- **Simple:** sum all pieces or sum all hash codes of pieces
  - Problem: changing order doesn't impact the code (e.g., h("cs10") == h("01sc"))
- **Better:** compute a polynomial based on value, position, and some constants
  - [*Show real examples of hashCodes in the wild*]
    - String.java, Integer.java

```
// Ex. Compute polynomial hash code for String s using Horner's rule - see: String's hashCode() method.
public int hashcode() {
  final int a = 37;

  int sum = x[0];
  for (int j = 1; j < s; j++)
    sum = a * sum + x[j];

  return sum;
}
```