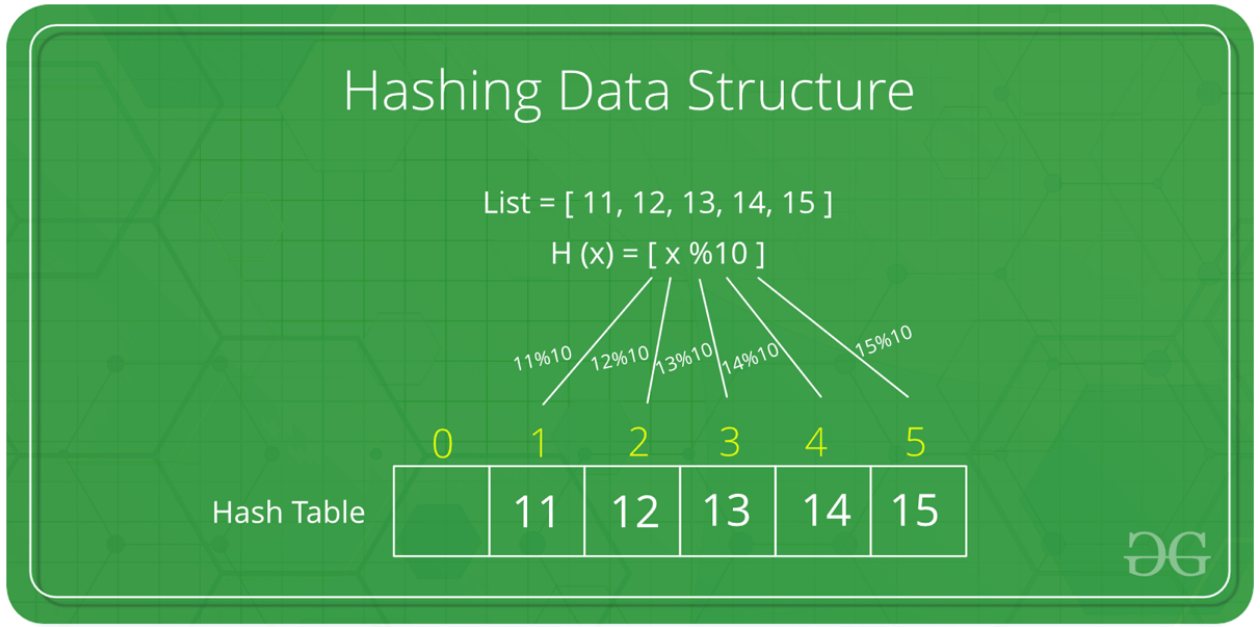
CSCI 3104: Algorithms

Lecture 9: Hash Tables

Rachel Cox

Department of Computer Science



What will we learn today?

- ❑ Hash Tables
- ❑ Collisions
- ❑ Load Factor
- ❑ When to apply

Intro to Algorithms, CLRS: Sections 11.1, 11.2, 11.3, 11.4, 11.5

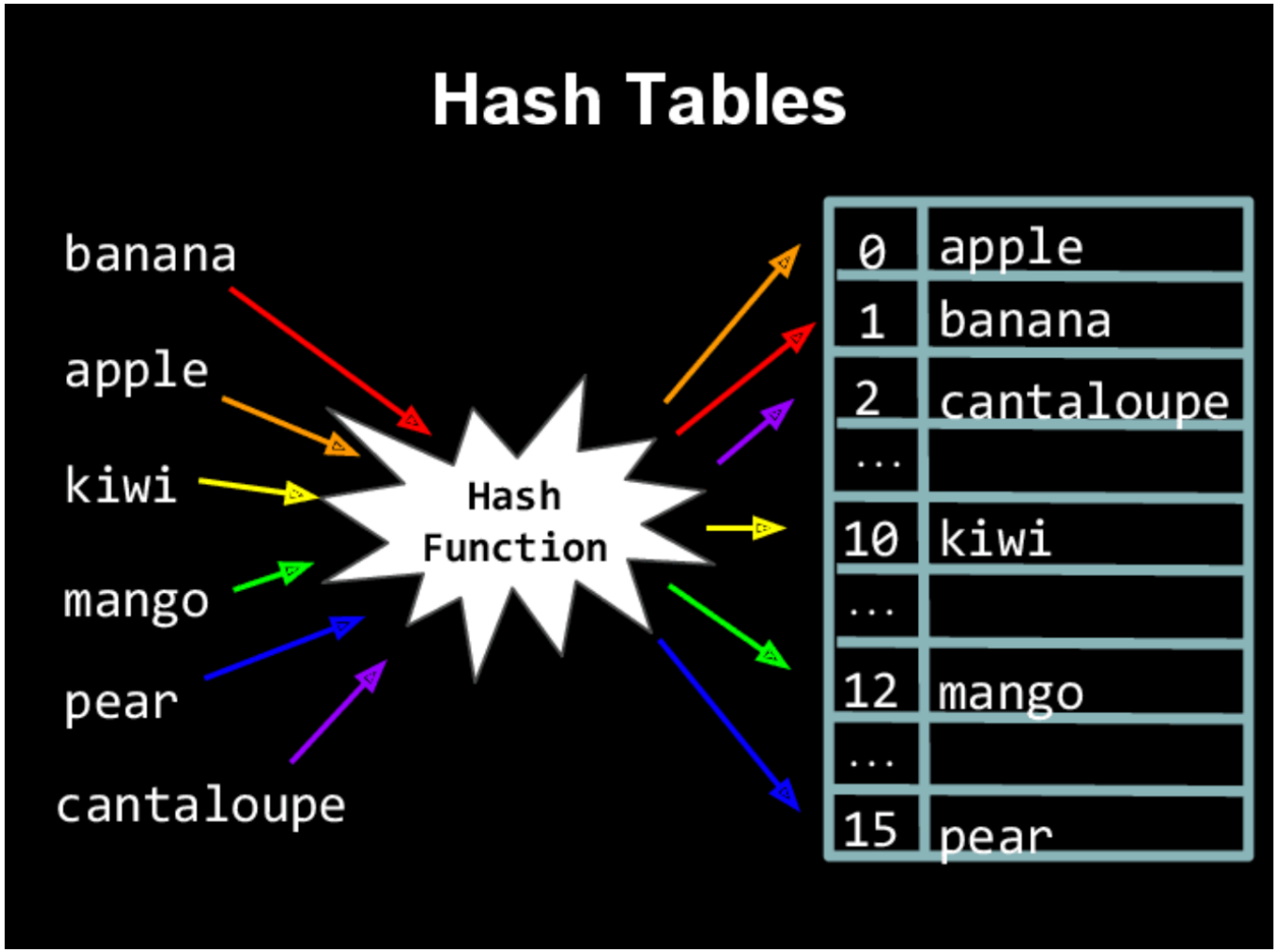
Hash Tables

Purpose: maintain a (possibly evolving) set of stuff

Insert/Add: Add a new record

Delete/Remove: Delete an existing record

Lookup/Find: Check for a particular record



- worst case: searching for an element can take as long as searching for an element in a linked list $\Theta(n)$

- in practice - $O(1)$

Hash Tables

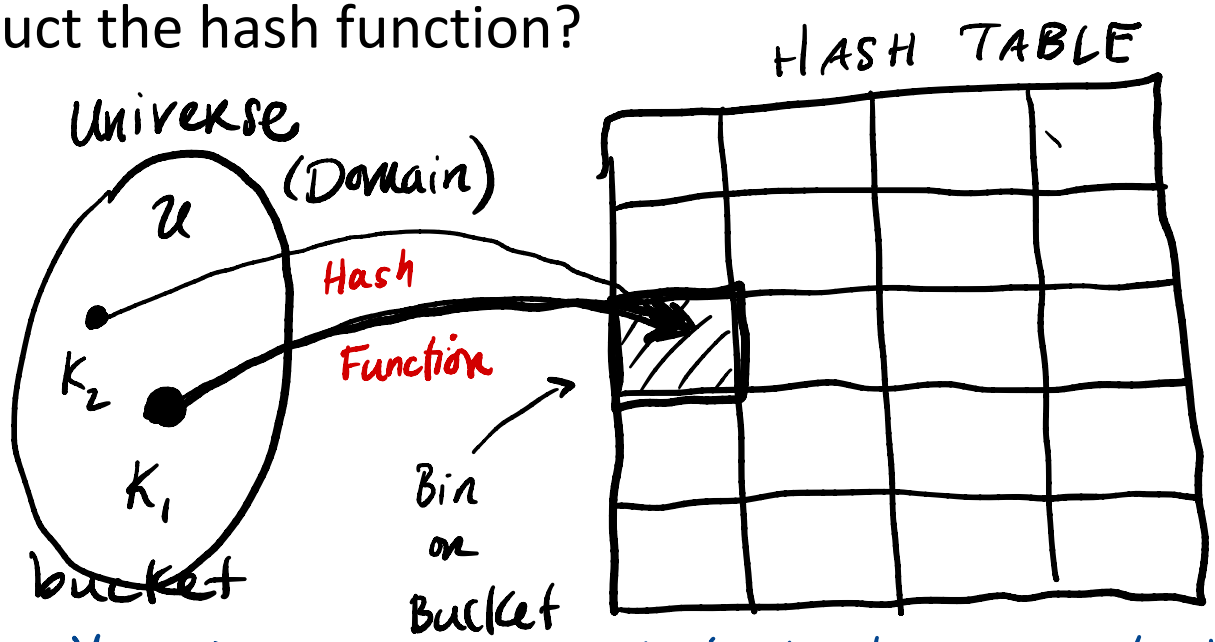
A **hash table** is basically a big array or look-up table where each row in the table stores some items.

To choose the row in which to store a given item, we use a **hash function** $h(x): U \to [1, n]$

How do we choose or construct the hash function?

- A good hash function spreads things around evenly

- A bad hash function assigns everything to one bucket



you'd end up with a linked list in one bucket with n elements.

Direct-Address Tables

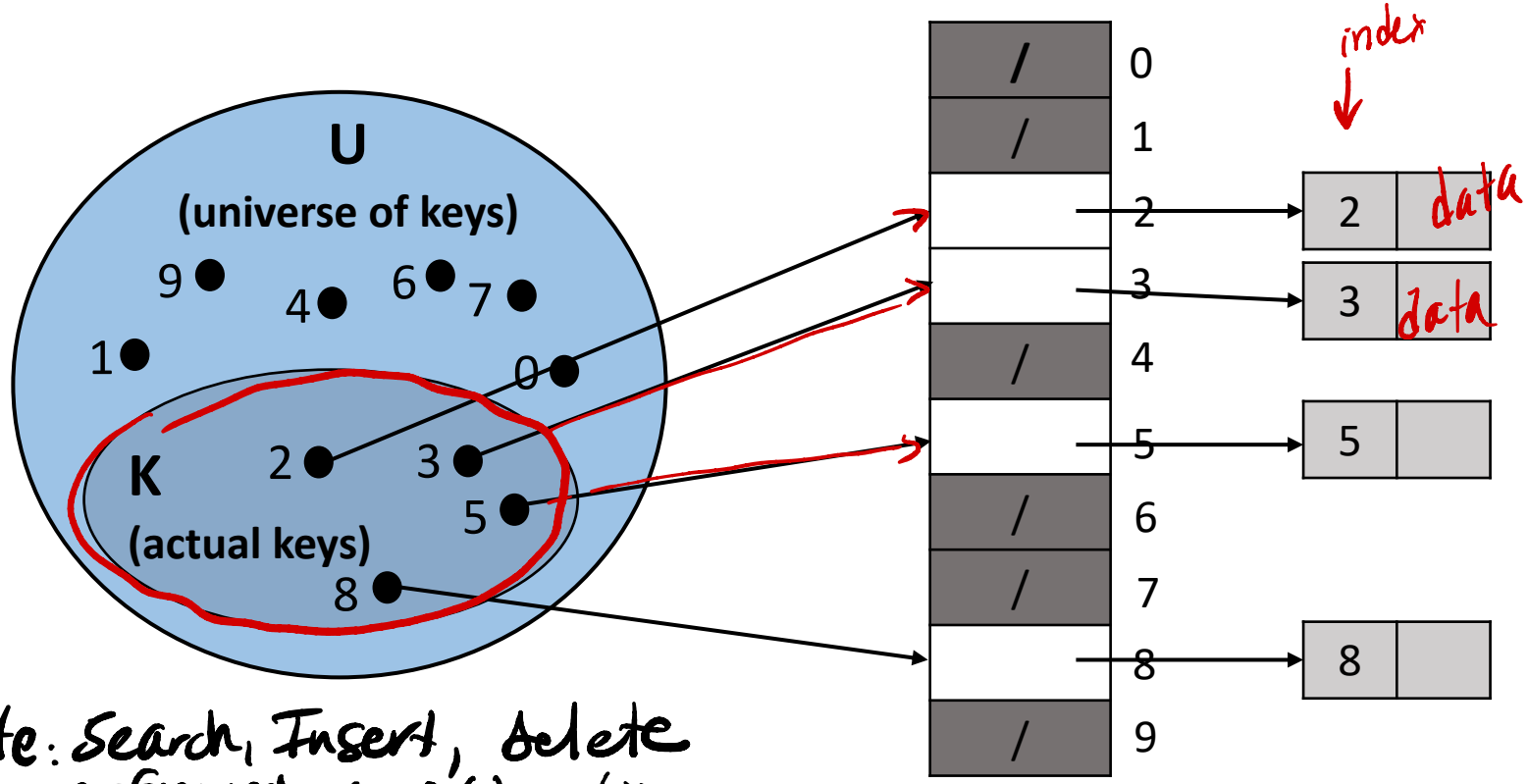
"hashing": mapping keys to bins with a hash function

Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large.

- Assume that no two elements have the same key.

perfect hashing: when a hash function maps no two keys to the same bucket (index) → every key has its own bucket

direct-address table: Each slot corresponds to a key in the universe



- This is just an array with a "fancy" mapping function for the index

- Recommendation – don't use a hash table when an array will do.

Note: Search, Insert, Delete performed in O(1) time

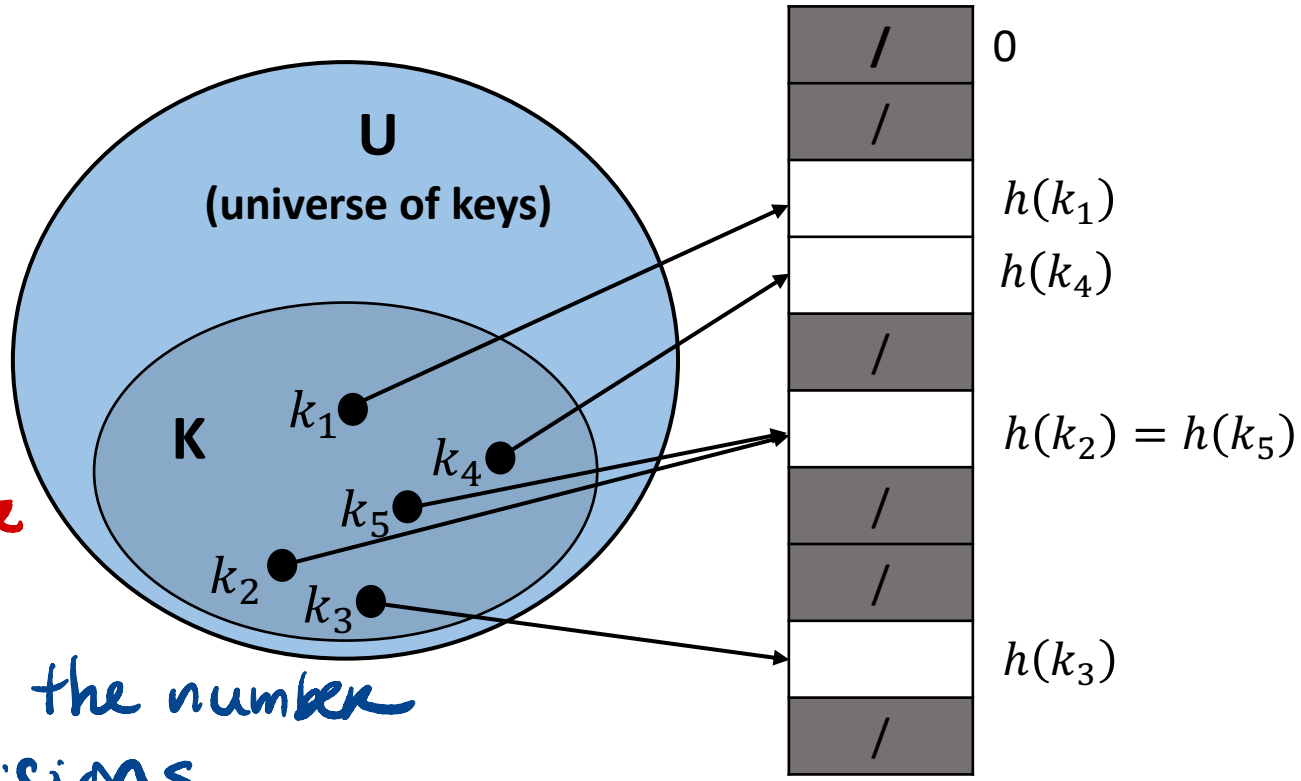
Hash Tables

Suppose that U is large, while K is much smaller. In this case, a hash table requires much less storage than a direct-address table.

Direct-addressing: an element with key k is stored in slot k

Hashing: an element with key k is stored in slot $h(k)$

- apply the hash function

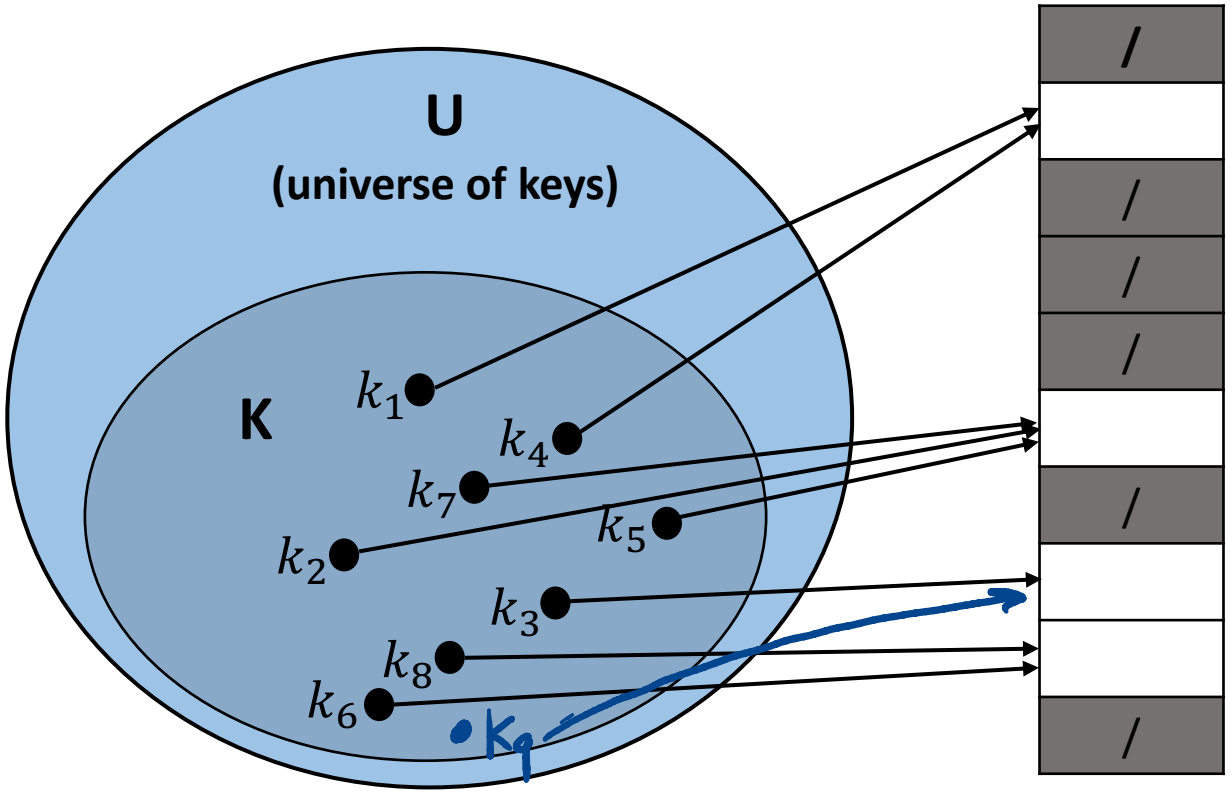


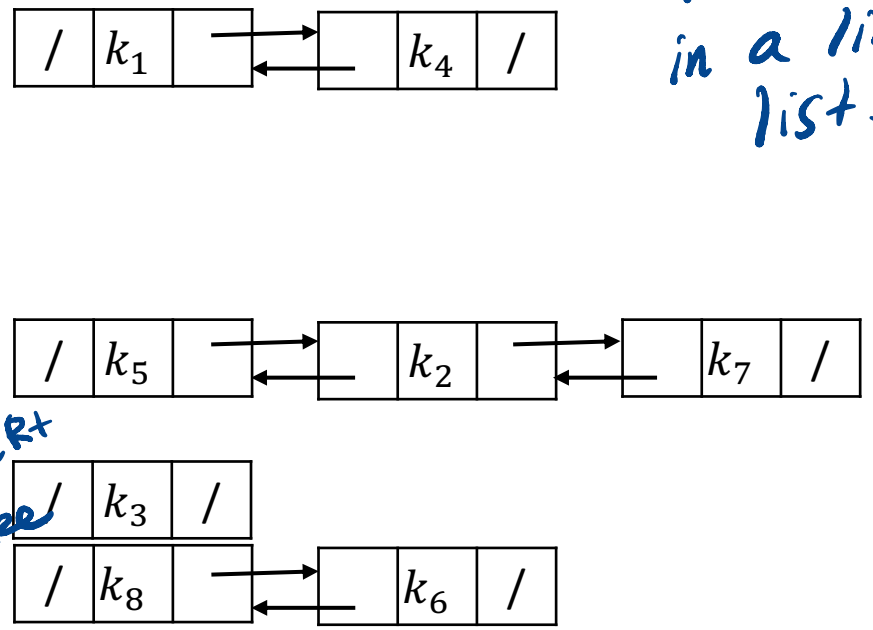
- with a Direct Address table, you know the physical address and you use it to directly locate the data
- often U is larger than l, the number of bins. → this causes collisions

Hash Tables

Collision: Multiple keys may "hash" to the same slot.

- when multiple items "hash" to the same slot, you link the items in a linked list.





insert in here

insert : $\Theta(1)$

Hash Tables

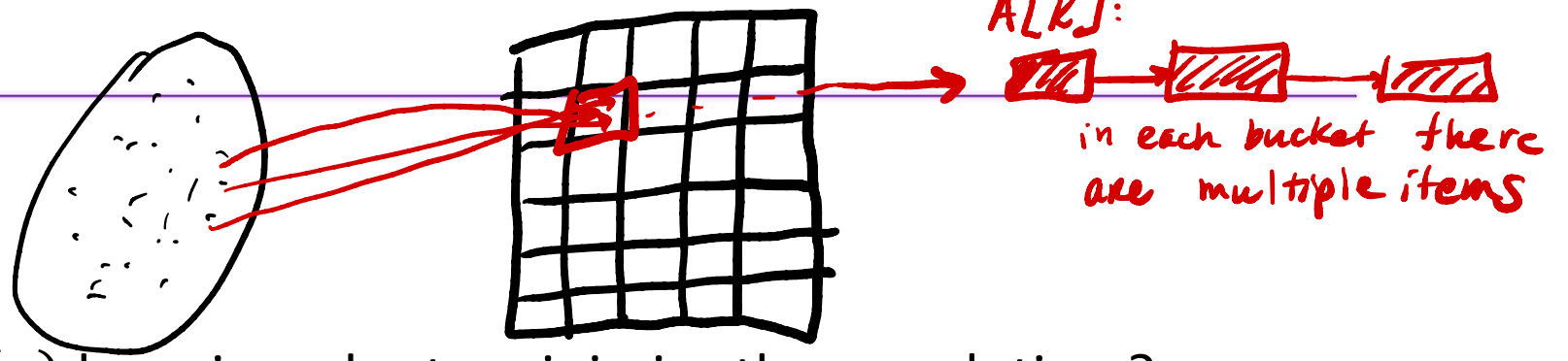
Chained Hash: In this version of the hash table, the array A has k elements, each of which is a linked list, sometimes called a "**bucket**".

Idea: If $h(x_i) = h(x_j) = k$ for some pair of items, then we simply add the second item to the bucket at $A[k]$.

$\text{ChainedHash} - \text{Add}(A, x)$ { Insert x at the head of the list $A[h(x.key)]$ }
$\text{ChainedHash} - \text{Find}(A, k)$ { search for an element with key k in list $A[h(k)]$}
$\text{ChainedHash} - \text{Remove}(A, x)$ { delete x from the list $A[h(x.key)]$}

worst case for insertion is $O(1)$
worst case for searching - proportional to the length of the list
worst case for deletion - $O(1)$ for doubly linked list
search time for singly linked list

Hash Tables



Chained Hash:

What property should $h(x)$ have in order to minimize the search time?

Best Case: If we store n items in a hash table with l slots, the best we can do is have each bucket contain the same number of items $\alpha = n/l$

Worst Case: $h(x)$ assigns each of n items to the same bucket, and search takes $\Theta(n)$ time.

- dependent on how evenly we distribute the n elements in the Universe to the l slots in the Hash table.

α - "load factor"

$$\alpha = \frac{n}{l} = \frac{\text{number of elements}}{\text{number of slots}}$$

Knowing the load factor α tells us something about how long search will take

Hash Tables

Eg. 

$\frac{3+2+4}{3} = 3$

Chained Hash: The performance of a hash table depends on how well $h(x)$ can evenly distribute the set of elements.

Average Case: Make the uniform hashing assumption: $\forall x\ P(h(x) = k) = 1/l$

→ Each of the l slots has the same probability of being "chosen"

Under this assumption, the expected number of elements in any particular linked list $A[k]$ is

length of the list in any given bucket

$$E(A[k].\text{length}) = \sum_{i=0}^{l-1} \left(\frac{1}{l}\right) A[i].\text{length} = \frac{n}{l} = \alpha$$

Definition of Expected Value

$$E[x] = \sum_{i=1}^{k} x_i p_i$$

p_i ← probabilities.

x_i – finite outcomes

"center of mass"

$$= \sum_{i=0}^{l-1} P(h(x) = i) \cdot A[i].\text{length} \quad *$$

$$= \sum_{i=0}^{l-1} \frac{1}{l}(A[i].\text{length})$$

$$= \frac{1}{l}\underbrace{(A[0].\text{length} + A[1].\text{length} + \cdots A[l-1].\text{length})}_{=n} \quad *$$

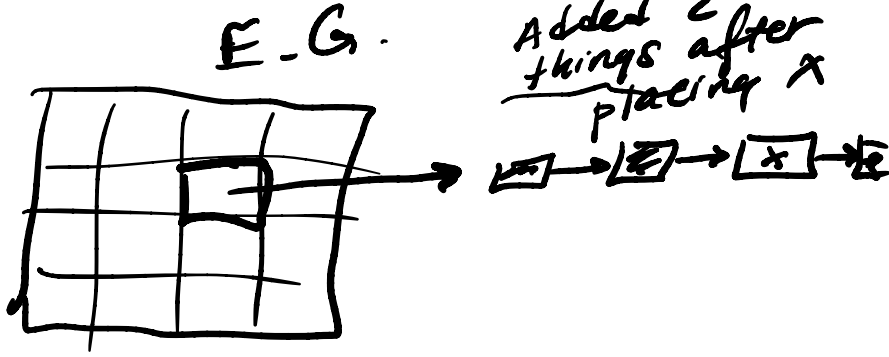
Hash Tables

Theorem: In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof: We assume the element being searched for is equally likely to be any of the n elements stored in the table.

- The number of elements examined in a successful search for 'x' is one more than the number of elements that appear before x in x's list

E.G.



- Let x_i denote the i^{th} element inserted into the table.
- For $i = 1, 2, \ldots, n$, let $k_i = x_i.key$

We define an indicator random variable: $X_{ij} = \begin{cases} 1 & h(k_i) = h(k_j) \\ 0 & \text{when } h(k_i) \neq h(k_j) \end{cases}$

Hash Tables

$\alpha = \frac{n}{\ell}$

(continued)

Theorem: In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

- By the simple uniform hashing. $P(h(k_i) = h(k_j)) = \frac{1}{\ell}$

$$\Rightarrow E[X_{ij}] = \sum_{i \neq j} 0 \cdot \frac{1}{\ell} + \sum_{i=j} 1 \cdot \frac{1}{\ell} = \frac{1}{\ell}$$

- $$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E[X_{ij}]\right)$$ By the linearity of Expected Value.

  - Average over n elements
  - count $+1$ when we Reach x_i
  - looking at all the times after x_i was inserted that $h(k_i) = h(k_j)$ (things hashed to same slot)

$$= \frac{1}{n}\sum_{i=1}^{n} 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \frac{1}{n} \cdot n + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{\ell}$$

$$= 1 + \frac{1}{n\ell}\sum_{i=1}^{n}(n-i)$$

$$= 1 + \frac{1}{n\ell}\left[n^2 - \frac{n(n+1)}{2}\right] = 1 + \alpha\left(\frac{1}{2} - \frac{1}{2n}\right)$$

$\Rightarrow \Theta(1+\alpha)$

Hash Tables

Assuming the uniform hashing property, we can answer some specific questions about the expected performance of hash tables.

Question: How large does n need to be before we start seeing buckets with multiple items?

Birthday problem ⇒

Question: How large does n need to be before every bucket has at least one element in it?

Coupon Collector

Birthday Paradox

Assume there are n people in the room and l days in the year. Furthermore, assume that each person is born on a day chosen uniformly at random from the l days. What is the expected number of pairs of individuals that have the same birthday? And more generally, as a function of n and l, what is the probability of finding at least one pair of individuals with the same birthday?

Birthday Paradox

Example: As a function of n (number of people) and l (number of possible birthdays), what is the probability of finding at least 1 pair of individuals with the same birthday? (That is, what is the probability of a good hash function causing a collision?)

First we make some assumptions:

1. Birthdays of any two people are independent $P(A \cap B) = P(A) \cdot P(B)$
2. Being born on any day of the year is equally likely
3. There are 366 days in the year

Restatement: The probability that two people have the same birthday is 1 minus the probability that no two people have the same birthday

$P(\text{At least 1 pair sharing a birthday}) = 1 - P(\text{no pair shares a birthday})$

Birthday Paradox

$P(\text{no birthdays in common}) = P(\text{2 people not sharing a bday})\,P(\text{3rd person not sharing a bday})$:

$P(\text{n+th person not share with the } n-1 \text{ people})$

Now let's compute the probability that with n people in the room no two people have the same birthday

Think of asking people their birthdays 1 at a time

1. When the first person enters the room, the probability is 1 that they do not have the same birthday as anyone else

2. When the second person enters the room, the probability is $\frac{365}{366}$ that they do not have the same birthday as the first person

3. When the third person enters the room, the probability is $\frac{364}{366}$ that they do not have the same birthday as the first two people

n. When the n^{th} person enters the room, the probability is $\frac{366-(n-1)}{366}$ that they do not have the same birthday as the first two people

Birthday Paradox

Since birthdays are independent, the probability that no two people have the same birthday is

$$p_n = \frac{365}{366} \cdot \frac{364}{366} \cdots \frac{367 - n}{366}$$

And so the probability that two of the n people have the same birthday is

$$1 - p_n = 1 - \frac{365}{366} \cdot \frac{364}{366} \cdots \frac{367 - n}{366}$$

We want to know what is the first value of n such that $1 - p_n \geq \frac{1}{2}$

We could find n with Calculus, but let's just plug in some numbers

Birthday Paradox

Analogy for collissions in a hash table.

When $n = 3$ the probability is $1 - p_n \approx 0.027$

When $n = 10$ the probability is $1 - p_n \approx 0.117$

When $n = 22$ the probability is $1 - p_n \approx 0.475$

When $n = 23$ the probability is $1 - p_n \approx 0.506$

When $n = 50$ the probability is $1 - p_n \approx 0.970$

When $n = 100$ the probability is $1 - p_n \approx 0.9999996$

$\lambda = 366$

How to Choose a Hash Function

Properties of a Good Hash Function

- Should lead to good performance
- Should be easy to store/very fast to evaluate

Ideal: Use a super-clever hash function guaranteed to spread every data set out evenly.

Problem: Does not exist! For every hash function, there is a pathological data set.

❖ Check out Crosby and Wallach, USENIX 2003 further reading

How to Choose a Hash Function

- Use a cryptographic hash function -- make it infeasible to reverse engineer a pathological data set

- Use randomization -- design a family of hash functions such that for all data sets S, almost all functions spread S out “pretty evenly”

Next Time

- Greedy Algorithms