Hashtables

David Kauchak
cs302
Spring 2013

Administrative

- Talk today at lunch
- Midterm
  - must take it by Friday at 6pm
- No assignment over the break

Hashtables

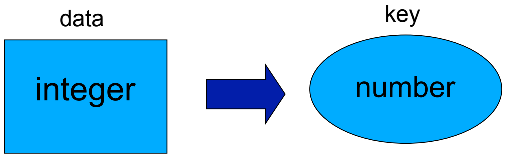
Constant time insertion and search (and deletion in some cases) for a large space of keys

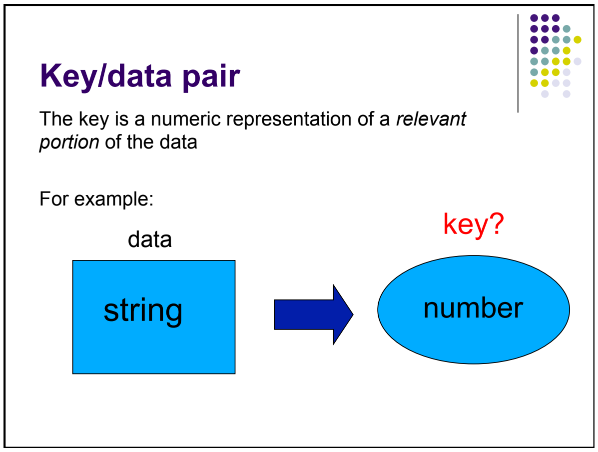
Applications
- Does *x* belong to *S?*
- I've found them very useful
- compilers
- databases
- search engines
- storing and retrieving non-sequential data
- save memory over an array

Key/data pair

The key is a numeric representation of a *relevant portion* of the data

For example:





Key/data pair

The key is a numeric representation of a *relevant portion* of the data

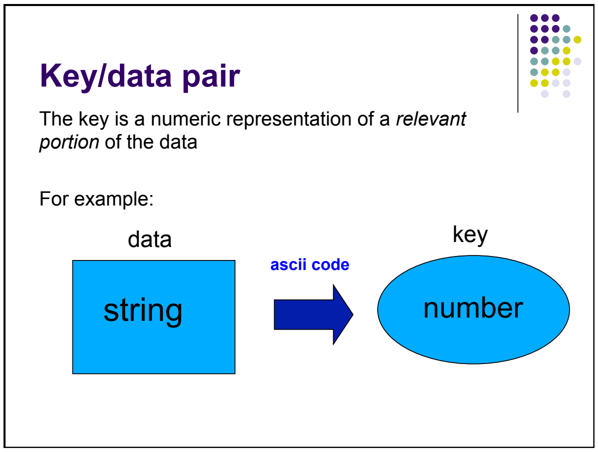
For example:

data

key?

string

number



Key/data pair

The key is a numeric representation of a *relevant portion* of the data

For example:

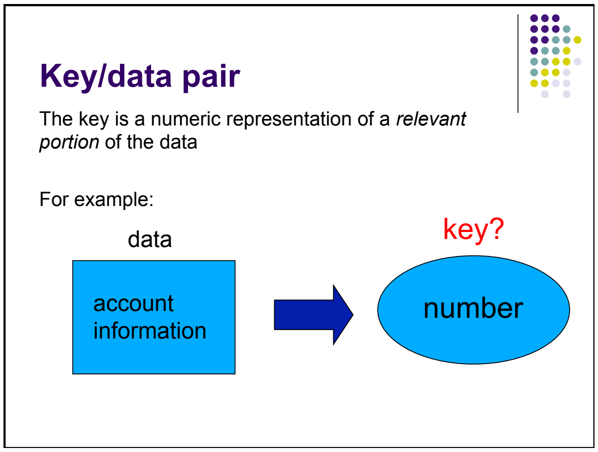
data

key

ascii code

string

number



Key/data pair

The key is a numeric representation of a *relevant portion* of the data

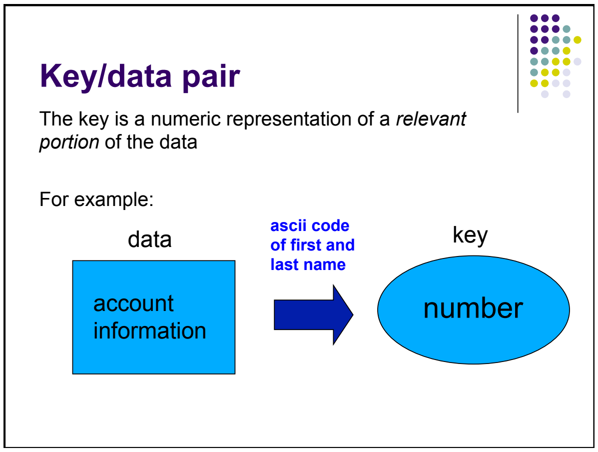
For example:

data

key?

account information

number



Key/data pair

The key is a numeric representation of a *relevant portion* of the data

For example:

data

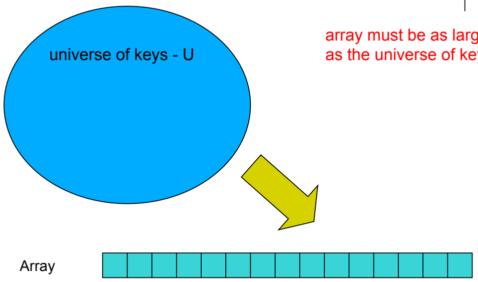
ascii code of first and last name

key

account information

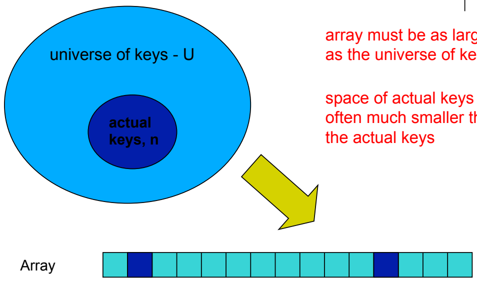
number

Why not just arrays aka direct-address tables?



array must be as large as the universe of keys

Why not just arrays?



array must be as large as the universe of keys

space of actual keys is often much smaller than the actual keys

Why not arrays?

Think of indexing all last names < 10 characters

- Census listing of all last names
  http://www.census.gov/genealogy/names/dist.all.last
  - 88,799 last names
- What is the size of our space of keys?
  - 26^{10} = a big number
- Not feasible!
- Even if it were, not space efficient

The load of a table/hashtable

m = number of possible entries in the table

n = number of keys stored in the table

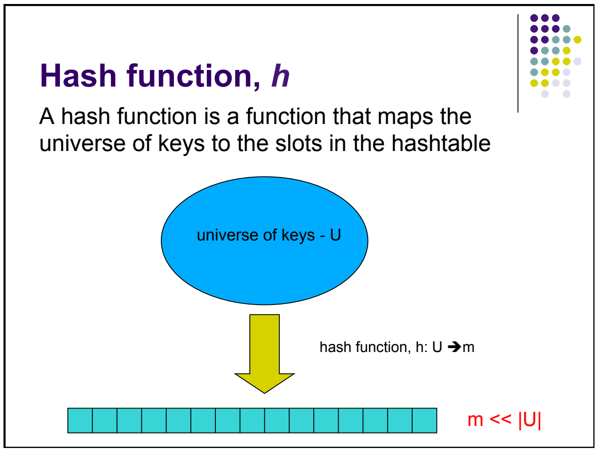
$\alpha = n/m$ is the **load factor** of the hashtable

What is the load factor of the last example?

- $\alpha = 88{,}799 / 26^{10}$ would be the load factor of last names using direct-addressing

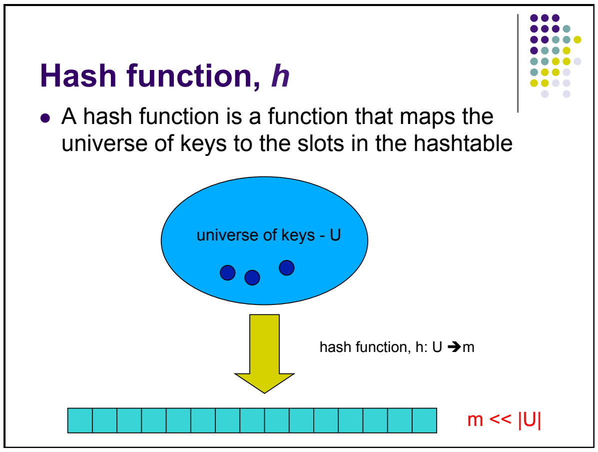
The smaller α, the more wasteful the table

The load also helps us talk about run time



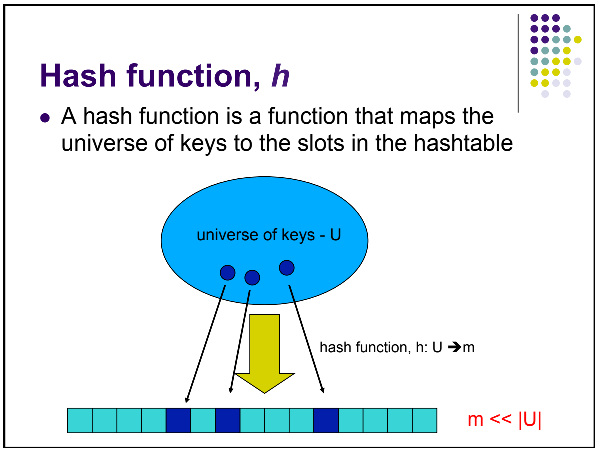
Hash function, *h*

A hash function is a function that maps the universe of keys to the slots in the hashtable



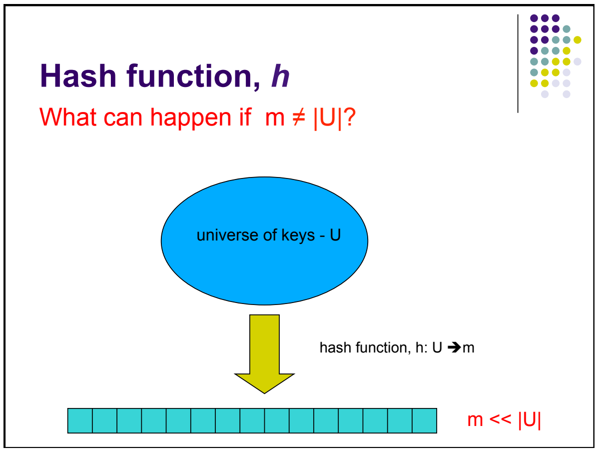
Hash function, *h*

- A hash function is a function that maps the universe of keys to the slots in the hashtable



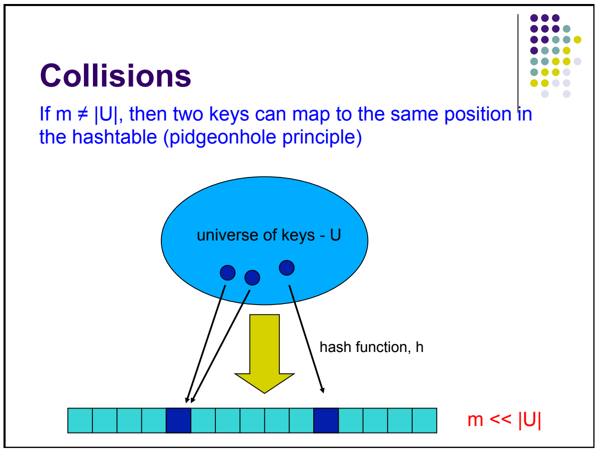
Hash function, *h*

- A hash function is a function that maps the universe of keys to the slots in the hashtable



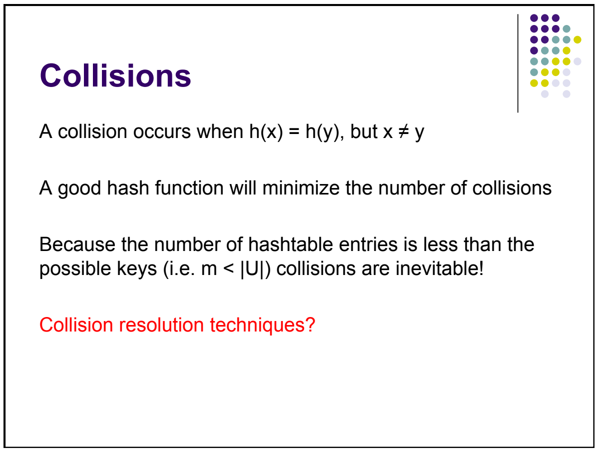
Hash function, *h*

What can happen if $m \neq |U|$?



Collisions

If m ≠ |U|, then two keys can map to the same position in the hashtable (pidgeonhole principle)



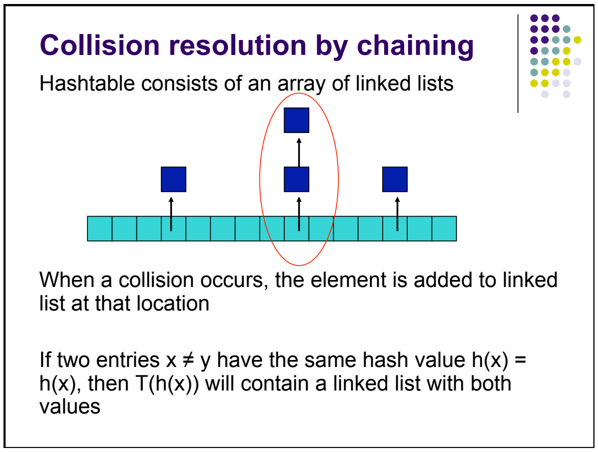
Collisions

A collision occurs when h(x) = h(y), but x ≠ y

A good hash function will minimize the number of collisions

Because the number of hashtable entries is less than the possible keys (i.e. m < |U|) collisions are inevitable!

Collision resolution techniques?

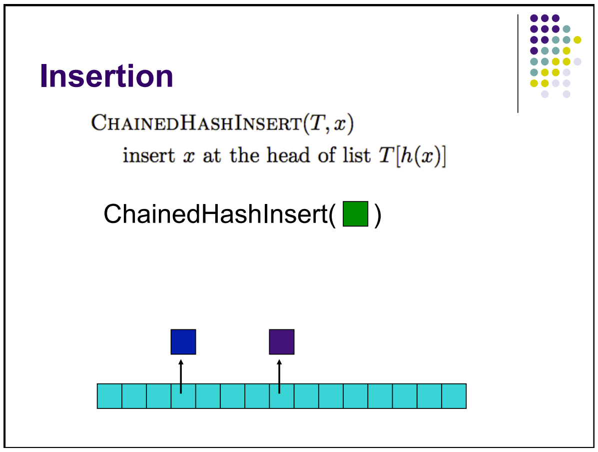


Collision resolution by chaining

Hashtable consists of an array of linked lists

When a collision occurs, the element is added to linked list at that location

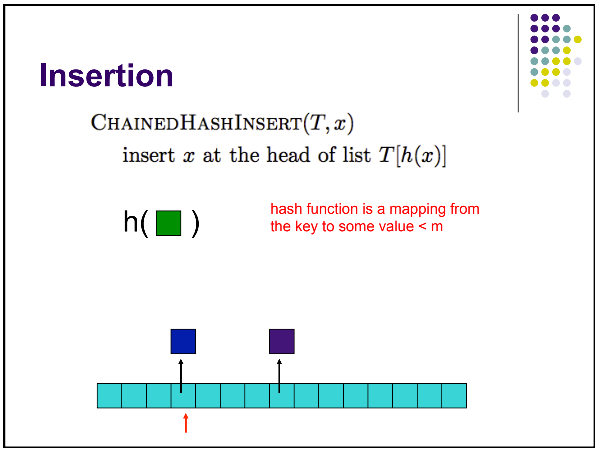
If two entries x ≠ y have the same hash value h(x) = h(x), then T(h(x)) will contain a linked list with both values



Insertion

```
CHAINEDHASHINSERT($T, x$)
    insert $x$ at the head of list $T[h(x)]$
```

ChainedHashInsert(🟩)

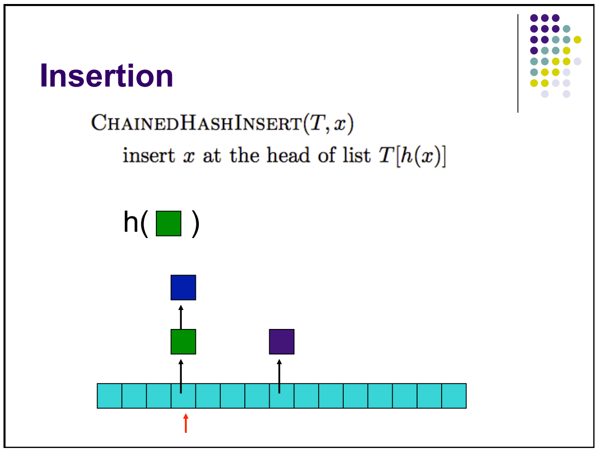


Insertion

```
CHAINEDHASHINSERT($T, x$)
    insert $x$ at the head of list $T[h(x)]$
```

h(🟩)

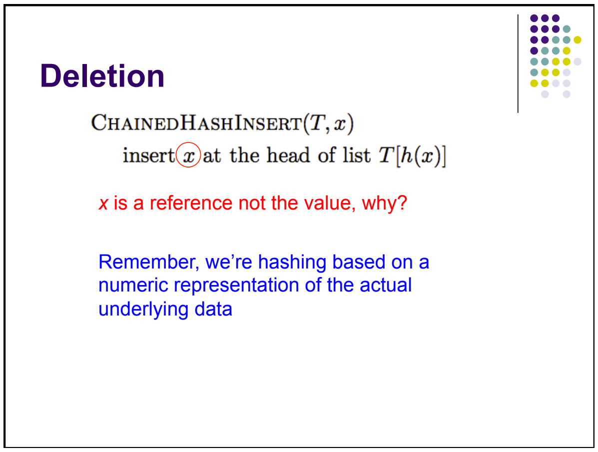
hash function is a mapping from the key to some value < m



Insertion

```
CHAINEDHASHINSERT($T, x$)
    insert $x$ at the head of list $T[h(x)]$
```

h(🟩)

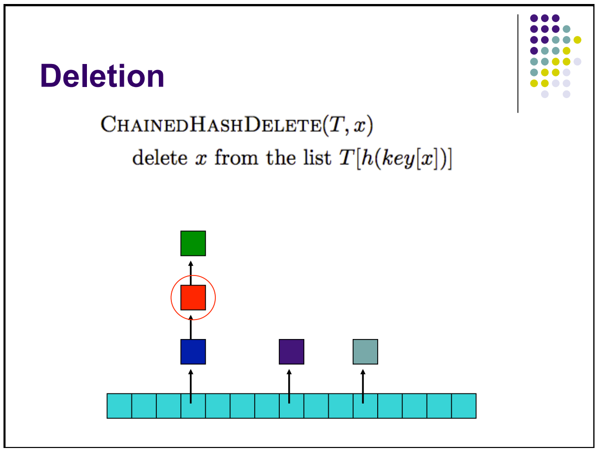


Deletion

```
CHAINEDHASHINSERT($T, x$)
    insert $x$ at the head of list $T[h(x)]$
```

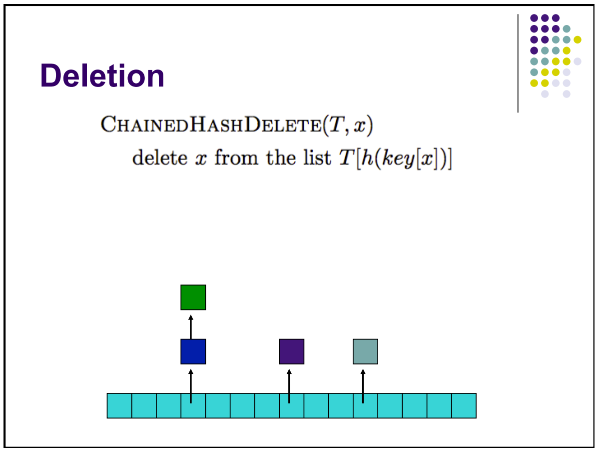
x is a reference not the value, why?

Remember, we're hashing based on a numeric representation of the actual underlying data



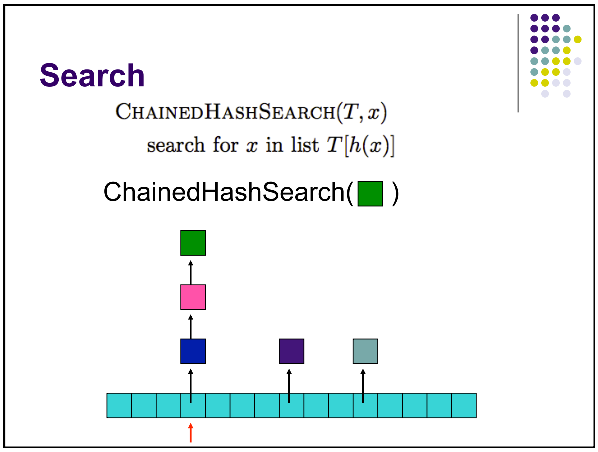
Deletion

```
CHAINEDHASHDELETE($T, x$)
    delete $x$ from the list $T[h(key[x])]$
```



Deletion

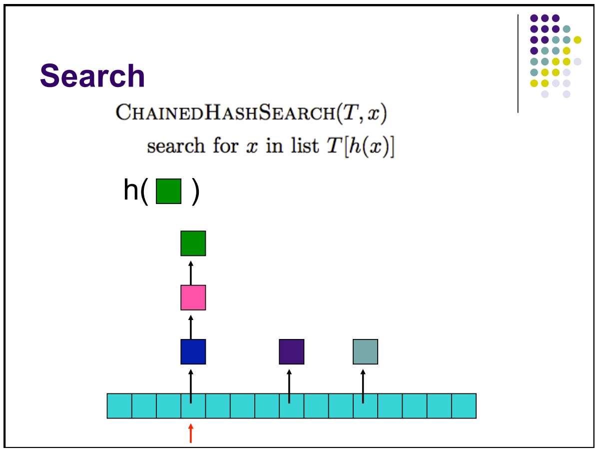
```
$\text{ChainedHashDelete}(T, x)$
    delete $x$ from the list $T[h(key[x])]$
```



Search

```
$\text{ChainedHashSearch}(T, x)$
    search for $x$ in list $T[h(x)]$
```

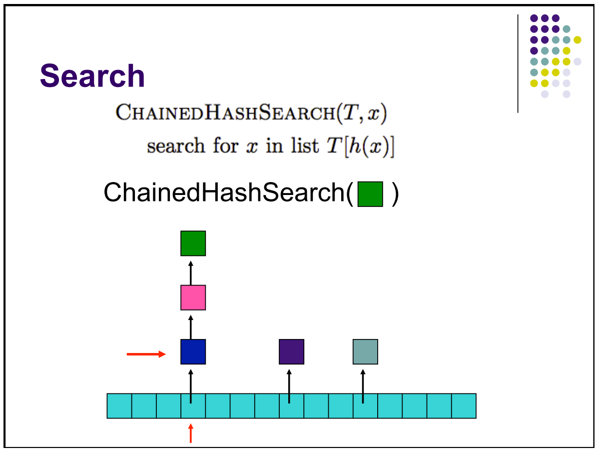
ChainedHashSearch(🟩)



Search

```
$\text{ChainedHashSearch}(T, x)$
    search for $x$ in list $T[h(x)]$
```

h(🟩)



Search

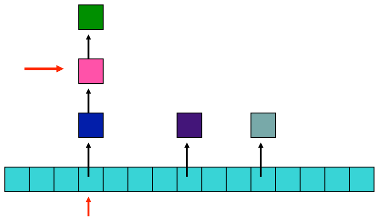
```
$\text{ChainedHashSearch}(T, x)$
    search for $x$ in list $T[h(x)]$
```

ChainedHashSearch(🟩)

Search

```
ChainedHashSearch$(T, x)$
    search for $x$ in list $T[h(x)]$
```

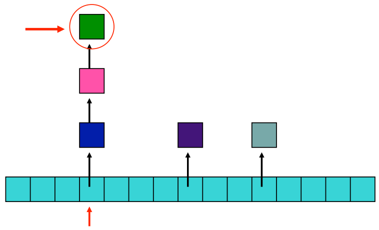
ChainedHashSearch(🟩)



Search

```
ChainedHashSearch$(T, x)$
    search for $x$ in list $T[h(x)]$
```

ChainedHashSearch(🟩)



Running time

```
ChainedHashInsert$(T, x)$
    insert $x$ at the head of list $T[h(x)]$
```

Θ(1)

```
ChainedHashDelete$(T, x)$
    delete $x$ from the list $T[h(key[x])]$
```

O(length of the chain)

```
ChainedHashSearch$(T, x)$
    search for $x$ in list $T[h(x)]$
```

O(length of the chain)

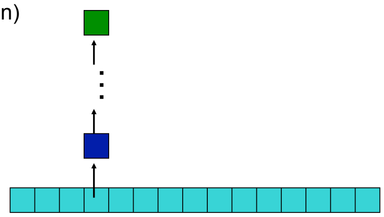
Length of the chain

Worst case?

Length of the chain

Worst case?

- All elements hash to the same location
- h(k) = 4
- O(n)



Length of the chain

Average case:

Depends on how well the hash function distributes the keys

What is the best we could hope for a hash function?

- simple uniform hashing: an element is equally likely to end up in any of the *m* slots

Under simple uniform hashing what is the average length of a chain in the table?

- *n* keys over *m* slots = *n* / *m* = α

Average chain length

If you roll a fair *m* sided die *n* times, how many times are we likely to see a given value?

For example, 10 sided die:

1 time

- 1/10

100 times

- 100/10 = 10

Search average running time

Two cases:

- Key is **not** in the table
  - must search all entries
  - Θ(1 + α)
- Key **is** in the table
  - on average search half of the entries
  - O(1 + α)

Hash functions

What makes a good hash function?

- Approximates the assumption of simple uniform hashing
- Deterministic – h(x) should always return the same value
- Low cost – if it is expensive to calculate the hash value (e.g. log n) then we don' t gain anything by using a table

Challenge: we don't generally know the distribution of the keys

- Frequently data tend to be clustered (e.g. similar strings, run-times, SSNs). A good hash function should spread these out across the table

Hash functions

What are some hash functions you've heard of before?

Division method

h(k) = k mod m

m	k	h(k)
11	25	
11	1	
11	17	
13	133	
13	7	
13	25	

Division method

h(k) = k mod m

m	k	h(k)
11	25	3
11	1	1
11	17	6
13	133	3
13	7	7
13	25	12

Division method

Don't use a power of two. Why?

m	k	bin(k)	h(k)
8	25	11001	
8	1	00001	
8	17	10001	

Division method

Don't use a power of two. Why?

m	k	bin(k)	h(k)
8	25	11001	1
8	1	00001	1
8	17	10001	1

if h(k) = k mod 2^p, the hash function is just the lower *p* bits of the value

Division method

Good rule of thumb for *m* is a prime number not to close to a power of 2

Pros:

- quick to calculate
- easy to understand

Cons:

- keys close to each other will end up close in the hashtable

Multiplication method

Multiply the key by a constant 0 < A < 1 and extract the fractional part of *kA*, then scale by *m* to get the index

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

extracts the fractional portion of *kA*

Multiplication method

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Common choice is for *m* as a power of 2 and

$$A = (\sqrt{5} - 1)/2 = 0.6180339887$$

Why a power of 2?

Book has other heuristics

Multiplication method

m	k	A	kA	h(k)
8	15	0.618		
8	23	0.618		
8	100	0.618		

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Multiplication method

m	k	A	kA	h(k)
8	15	0.618	9.27	floor(0.27*8) = 2
8	23	0.618	14.214	floor(0.214*8) = 1
8	100	0.618	61.8	floor(0.8*8) = 6

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Other hash functions

http://en.wikipedia.org/wiki/List_of_hash_functions

cyclic redundancy checks (i.e. disks, cds, dvds)

Checksums (i.e. networking, file transfers)

Cryptographic (i.e. MD5, SHA)

Open addressing

Keeping around an array of linked lists can be inefficient and a hassle

Like to keep the hashtable as just an array of elements (no pointers)

How do we deal with collisions?

- compute another slot in the hashtable to examine

Hash functions with open addressing

Hash function must define a **probe sequence** which is the list of slots to examine when searching or inserting

The hash function takes an additional parameter *i* which is the number of collisions that have already occurred

The probe sequence **must** be a permutation of every hashtable entry. Why?

{ h(k,0), h(k,1), h(k,2), …, h(k, m-1) } is a permutation of { 0, 1, 2, 3, …, m-1 }

Probe sequence

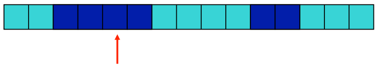
h(k, 0)

Probe sequence

h(k, 1)

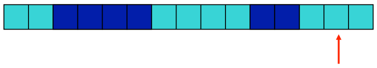
Probe sequence

h(k, 2)



Probe sequence

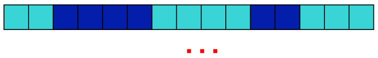
h(k, 3)



Probe sequence

h(k, …)

must visit all locations



Open addressing: Insert

```
HASH-INSERT($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$
        $i \leftarrow i + 1$
        $j \leftarrow h(k, i)$
if $T[j] = null$
        return $j$
else
        error "hash is full"
```

Open addressing: Insert

```
HASH-INSERT($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = null$
    return $j$
else
    error "hash is full"
```

get the first hashtable entry to look in

Open addressing: Insert

```
HASH-INSERT($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = null$
    return $j$
else
    error "hash is full"
```

follow the probe sequence until we find an open entry

Open addressing: Insert

```
HASH-INSERT($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = null$
    return $j$
else
    error "hash is full"
```

return the open entry

Open addressing: Insert

```
HASH-INSERT($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = null$
    return $j$
else
    error "hash is full"
```

hashtable can fill up

Open addressing: search

```
HASH-SEARCH($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$ and $T[j] \neq k$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = k$
    return $j$
else
    return $null$
```

Open addressing: search

```
HASH-SEARCH($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$ and $\boxed{T[j] \neq k}$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = k$
    return $j$
else
    return $null$
```

```
HASH-INSERT($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $T[j] \neq null$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = null$
    return $j$
else
    error "hash is full"
```

Open addressing: search

```
HASH-SEARCH($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $\boxed{T[j] \neq null}$ and $T[j] \neq k$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = k$
    return $j$
else
    return $null$
```

```
HASH-INSERT($T, k$)
$i \leftarrow 0$
$j \leftarrow h(k, i)$
while $i < m - 1$ and $\boxed{T[j] \neq null}$
    $i \leftarrow i + 1$
    $j \leftarrow h(k, i)$
if $T[j] = null$
    return $j$
else
    error "hash is full"
```

"breaks" the probe sequence

Open addressing: delete

Two options:

- mark node as "deleted" (rather than null)
  - modify search procedure to continue looking if a "deleted" node is seen
  - modify insert procedure to fill in "deleted" entries
  - increases search times
- if a lot of deleting will happen, use chaining

Probing schemes

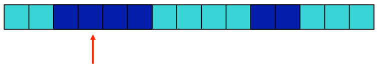
Linear probing – if a collision occurs, go to the next slot

- h(k,i) = (h(k) + i) mod m
- Does it meet our requirement that it visits every slot?
- for example, m = 7 and h(k) = 4

h(k,0) = 4
h(k,1) = 5
h(k,2) = 6
h(k,3) = 0
h(k,3) = 1

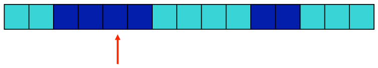
Linear probing: search

h(🟩, 0)



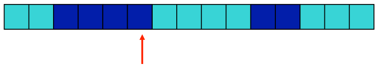
Linear probing: search

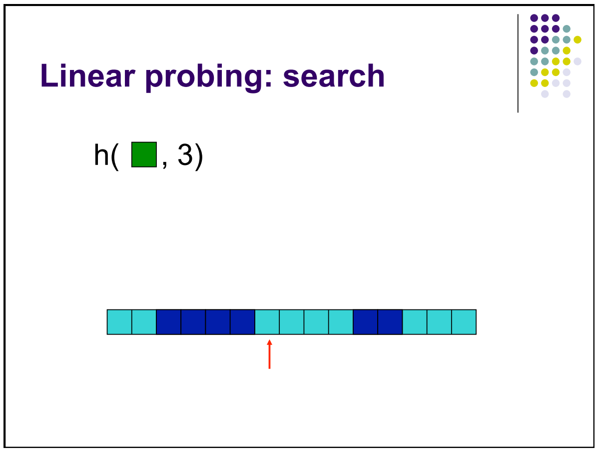
h(🟩, 1)



Linear probing: search

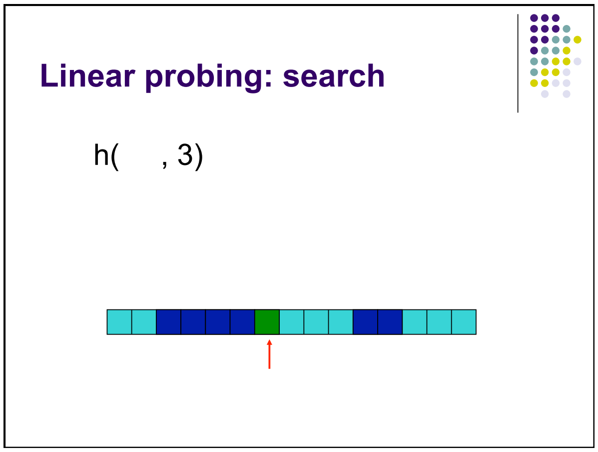
h(🟩, 2)





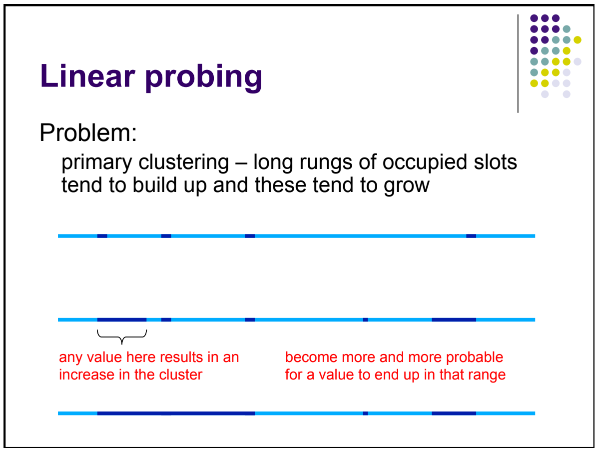
Linear probing: search

h(🟩, 3)



Linear probing: search

h(, 3)



Linear probing

Problem:

primary clustering – long rungs of occupied slots tend to build up and these tend to grow

Quadratic probing

$h(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod m$

Rather than a linear sequence, we probe based on a quadratic function

Problems:

- must pick constants and *m* so that we have a proper probe sequence
- if h(x) = h(y), then h(x,i) = h(y,i) for all i
- secondary clustering

Double hashing

Probe sequence is determined by a second hash function

$h(k,i) = (h_1(k) + i(h_2(k))) \bmod m$

Problem:

- $h_2(k)$ must visit all possible positions in the table

Running time of insert and search for open addressing

Depends on the hash function/probe sequence

Worst case?

- O(n) – probe sequence visits every full entry first before finding an empty

Running time of insert and search for open addressing

Average case?

We have to make at least one probe

Running time of insert and search for open addressing

Average case?

What is the probability that the first probe will **not** be successful (assume uniform hashing function)?

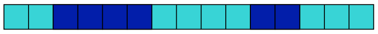
α

Running time of insert and search for open addressing

Average case?

What is the probability that the first **two** probed slots will **not** be successful?

why '~' ? → $\sim\alpha^2$



Running time of insert and search for open addressing

Average case?

What is the probability that the first **three** probed slots will **not** be successful?

$\sim\alpha^3$

Running time of insert and search for open addressing

Average case: expected number of probes

sum of the probability of making 1 probe, 2 probes, 3 probes, …

$$E[probes] = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

$$= \sum_{i=0}^{m} \alpha^i$$

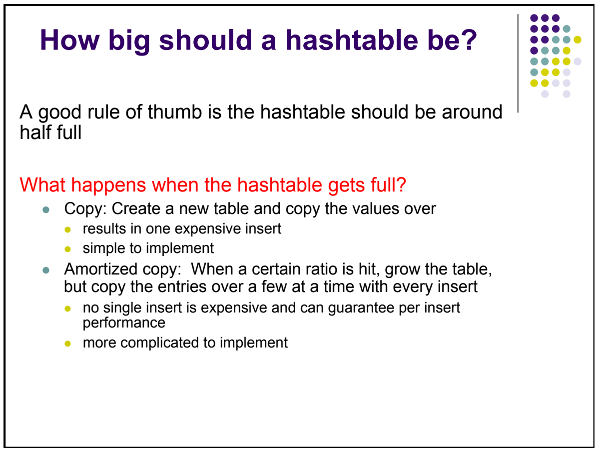
$$< \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}$$

Average number of probes

$$E[probes] = \frac{1}{1-\alpha}$$

α	Average number of searches
0.1	$1/(1-.1) = 1.11$
0.25	$1/(1-.25) = 1.33$
0.5	$1/(1-.5) = 2$
0.75	$1/(1-.75) = 4$
0.9	$1/(1-.9) = 10$
0.95	$1/(1-.95) = 20$
0.99	$1/(1-.99) = 100$



How big should a hashtable be?

A good rule of thumb is the hashtable should be around half full

What happens when the hashtable gets full?

- Copy: Create a new table and copy the values over
    - results in one expensive insert
    - simple to implement
- Amortized copy: When a certain ratio is hit, grow the table, but copy the entries over a few at a time with every insert
    - no single insert is expensive and can guarantee per insert performance
    - more complicated to implement