Hash Tables

Dr. Chung-Wen Albert Tsao

Overview

- Hash tables
- Hashing functions
- Conflict Resolutions:
  - Chaining
  - Open addressing

- Part of the slides are based on material from Prof. David Kauchak, Middlebury College

Hash Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We typically don't care about sorted order
- Other applications
  - Databases
  - Search engines
  - …

Hash Tables

- More formally:
  - Given a *hash table* *T* and a record *x*, with key (= symbol) we need to support:
    - Insert (*T*, ***x***) in O(1) expected time!
    - Delete (*T*, ***x***) in O(1) expected time!
    - Search(*T*, ***k***) in O(1) expected time!
  - We want these to be fast, but don't care about sorting the records

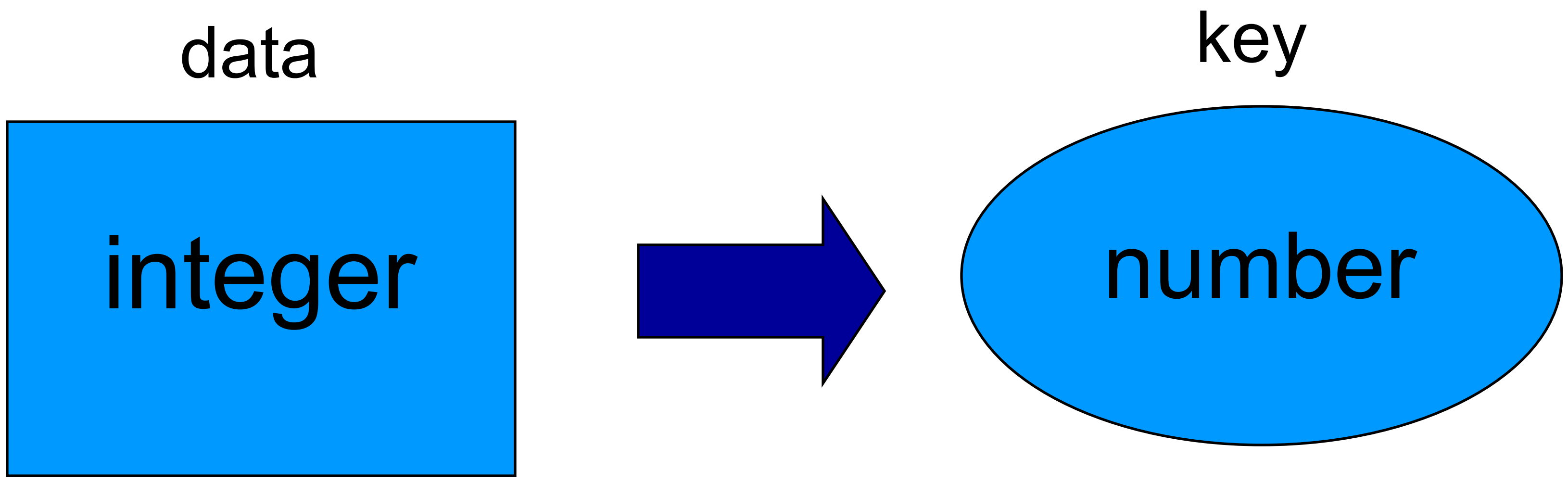
Hashing: Keys

- In the following discussions we will consider all keys to be (possibly large) natural numbers
  - When they are not, have to interpret them as natural numbers.
- *How can we convert ASCII strings to natural numbers for hashing purposes?*
  - Example: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
    - ASCII values: C=67, L=76, R=82, S=83.
    - There are 128 basic ASCII values.
    - So, CLRS = $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141{,}764{,}947$.

Key/data pair

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

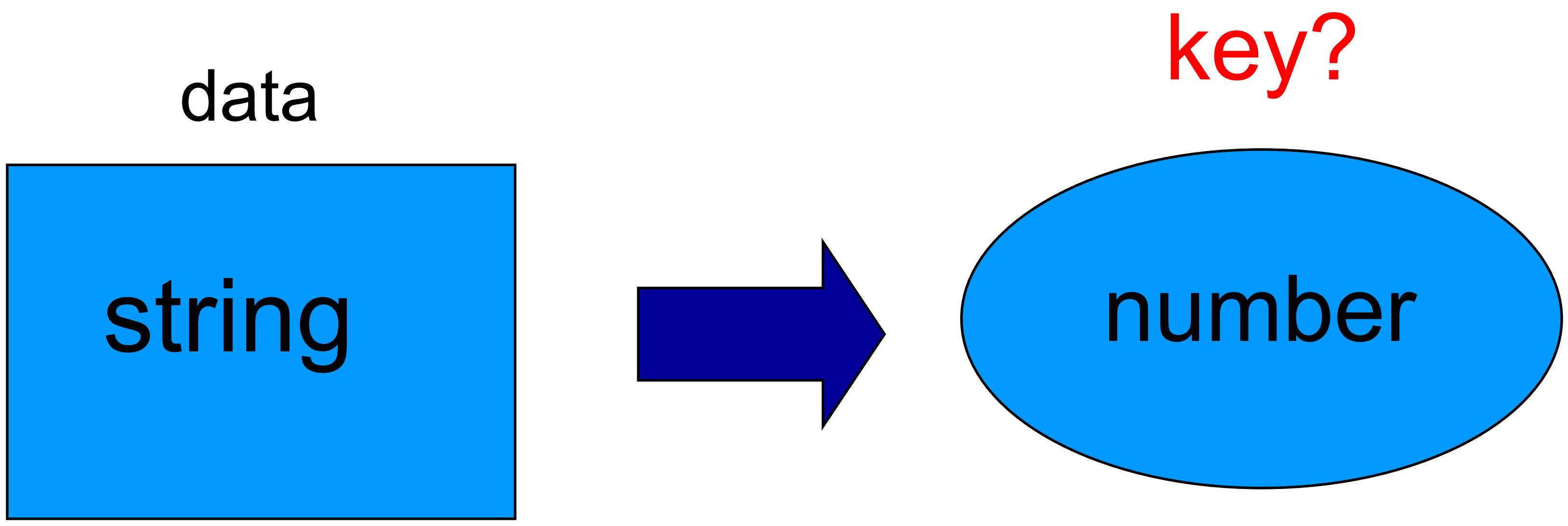
For example:



Key/data pair

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

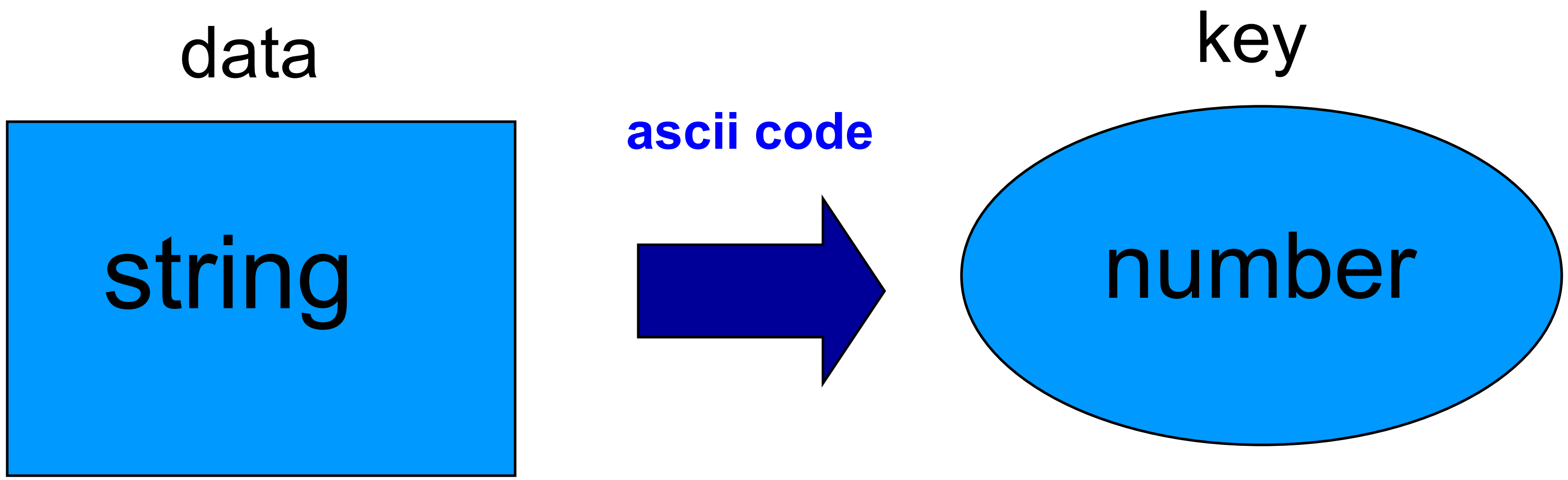
For example:



Key/data pair

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

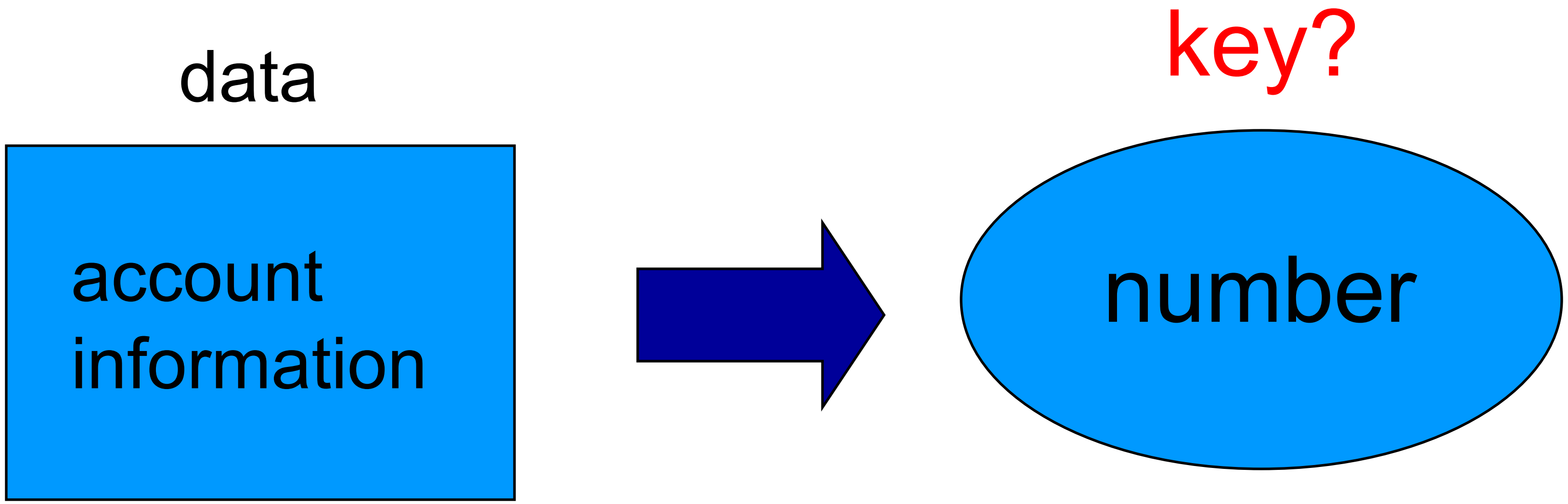
For example:



Key/data pair

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

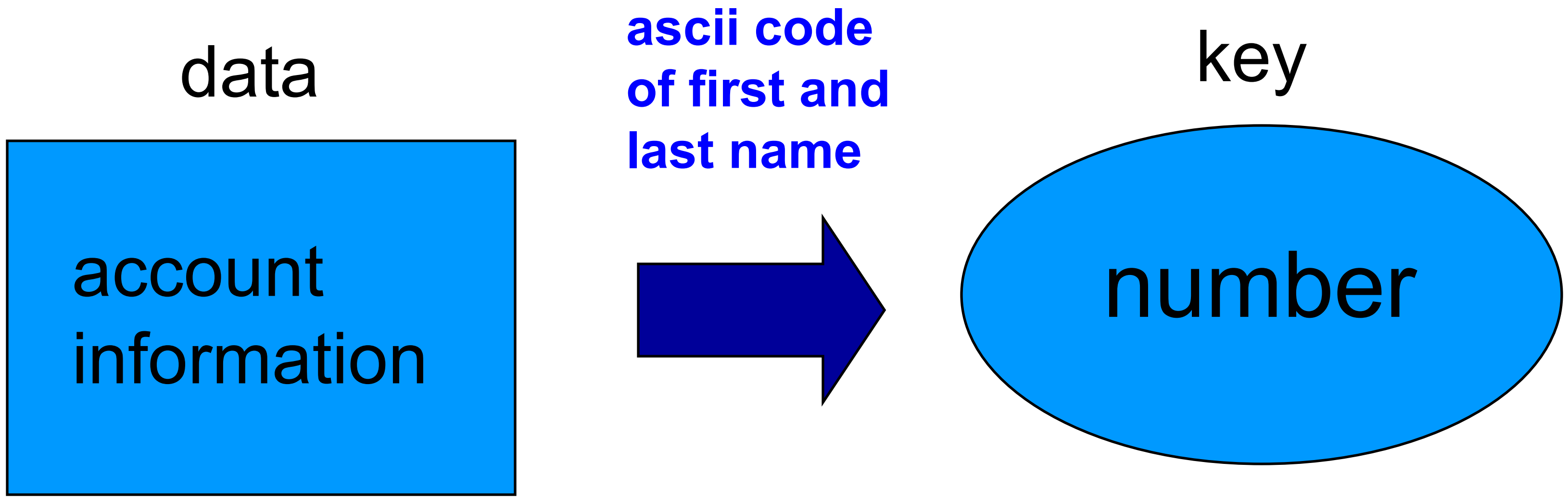
For example:



Key/data pair

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

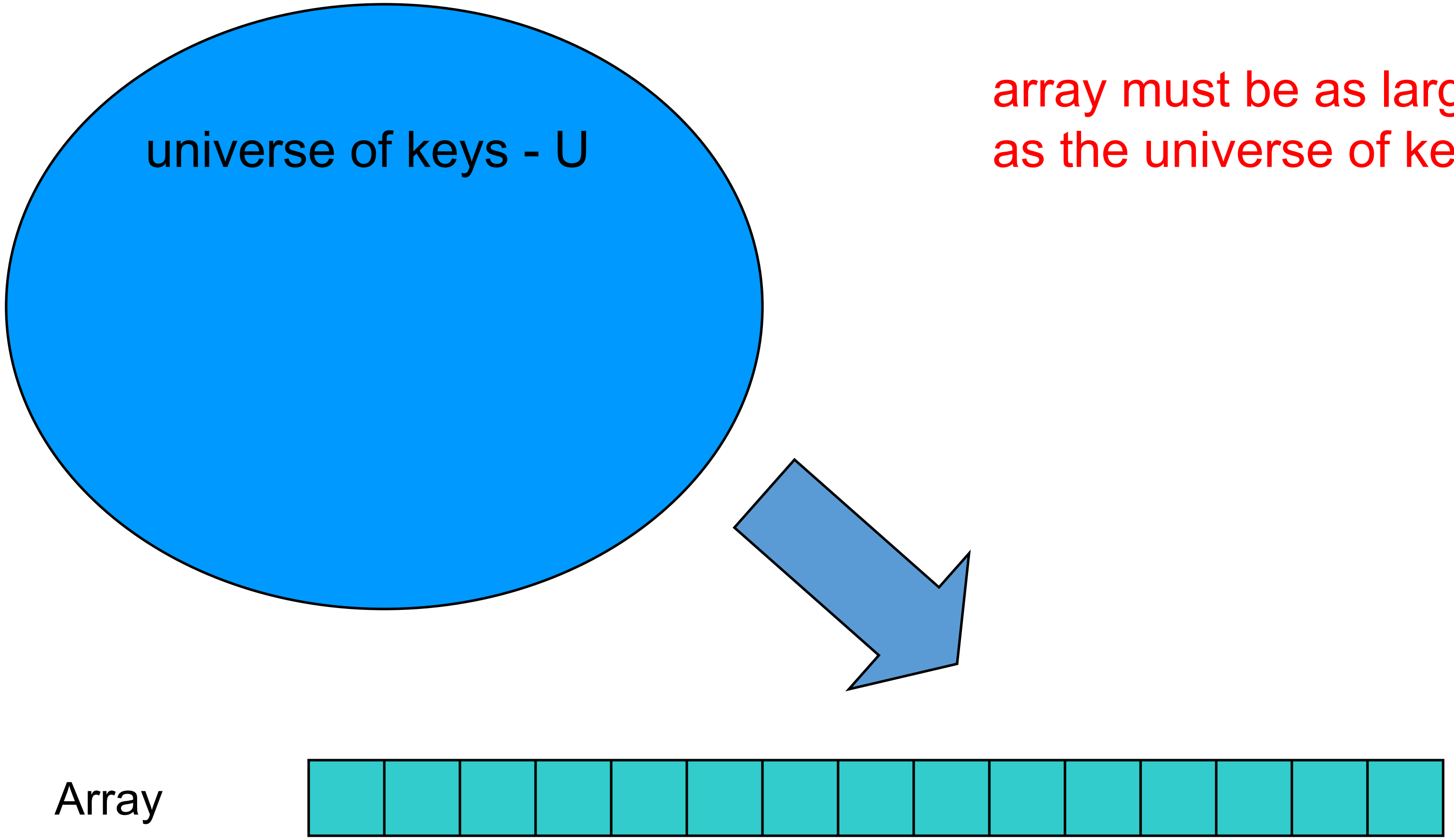
For example:



Direct Addressing

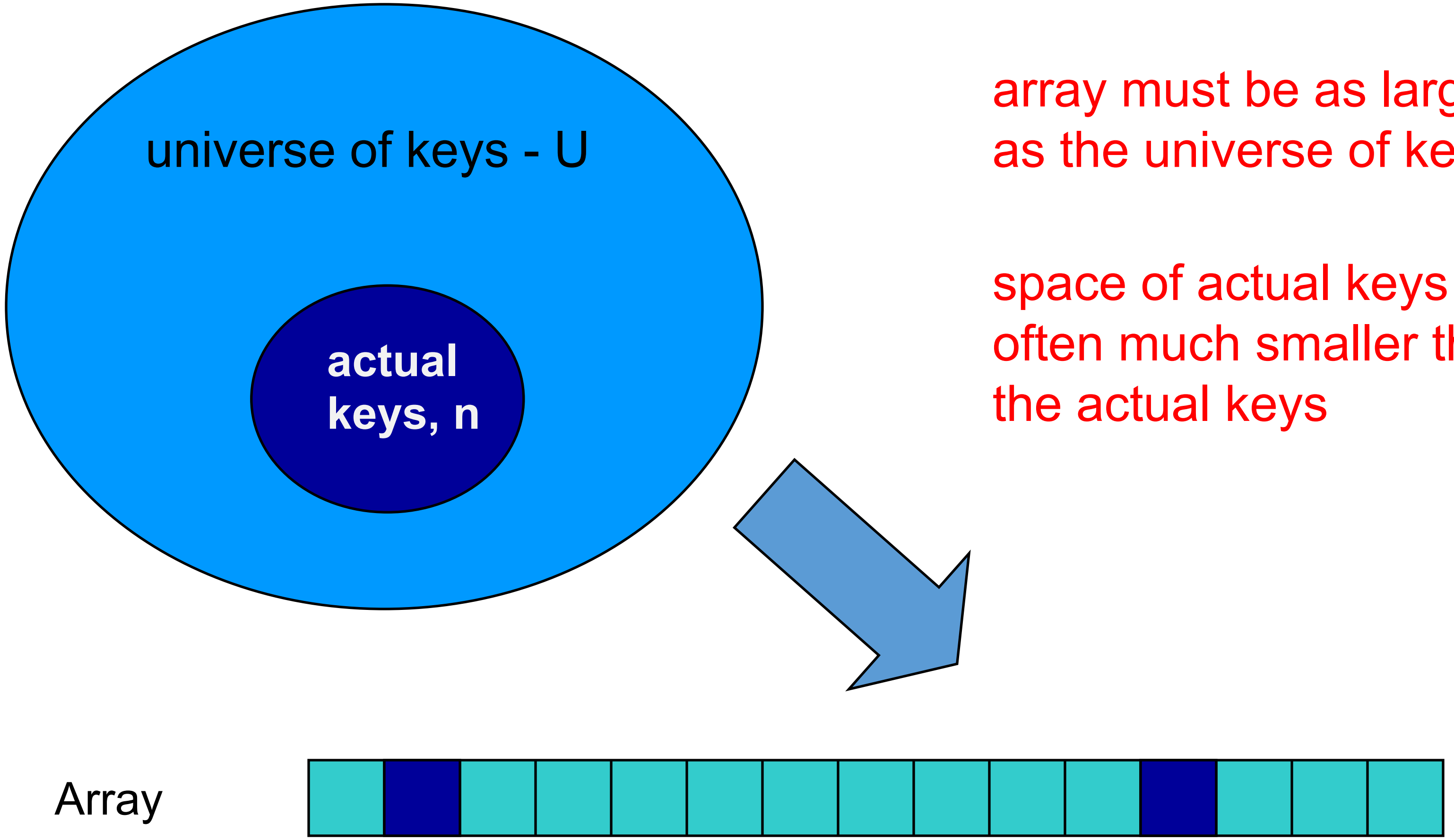
- Suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- The idea:
  - Set up an array T[0..m-1] in which
    - T[i] = x if $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

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?

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

The smaller α, the more wasteful the table

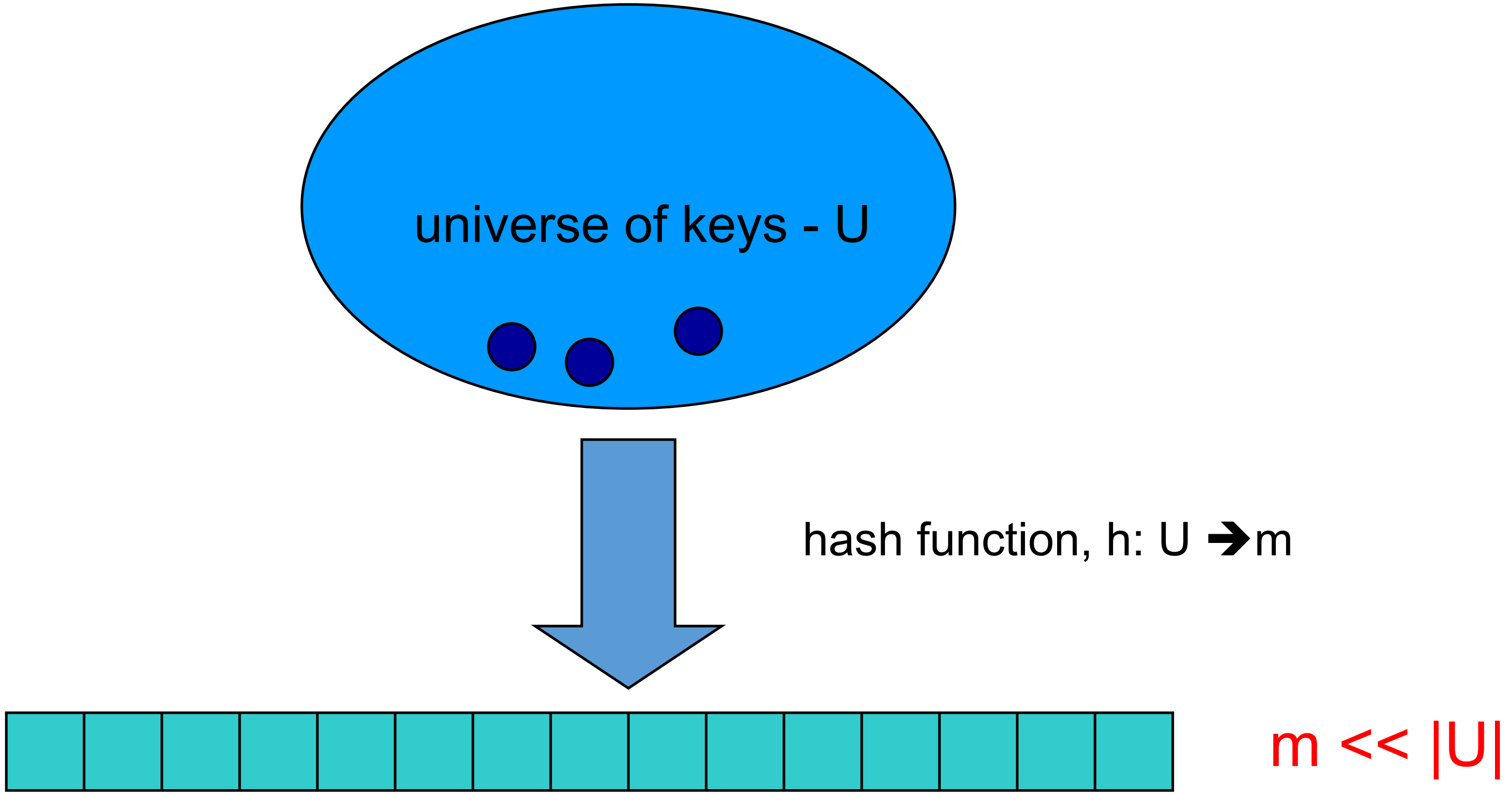
The load also helps us talk about run time

The Problem With Direct Addressing

- Direct addressing works well when the range *m* of keys is relatively small
- But what if the keys are 32-bit integers?
    - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
    - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range 0..*m*-1
- This mapping is called a *hash function*

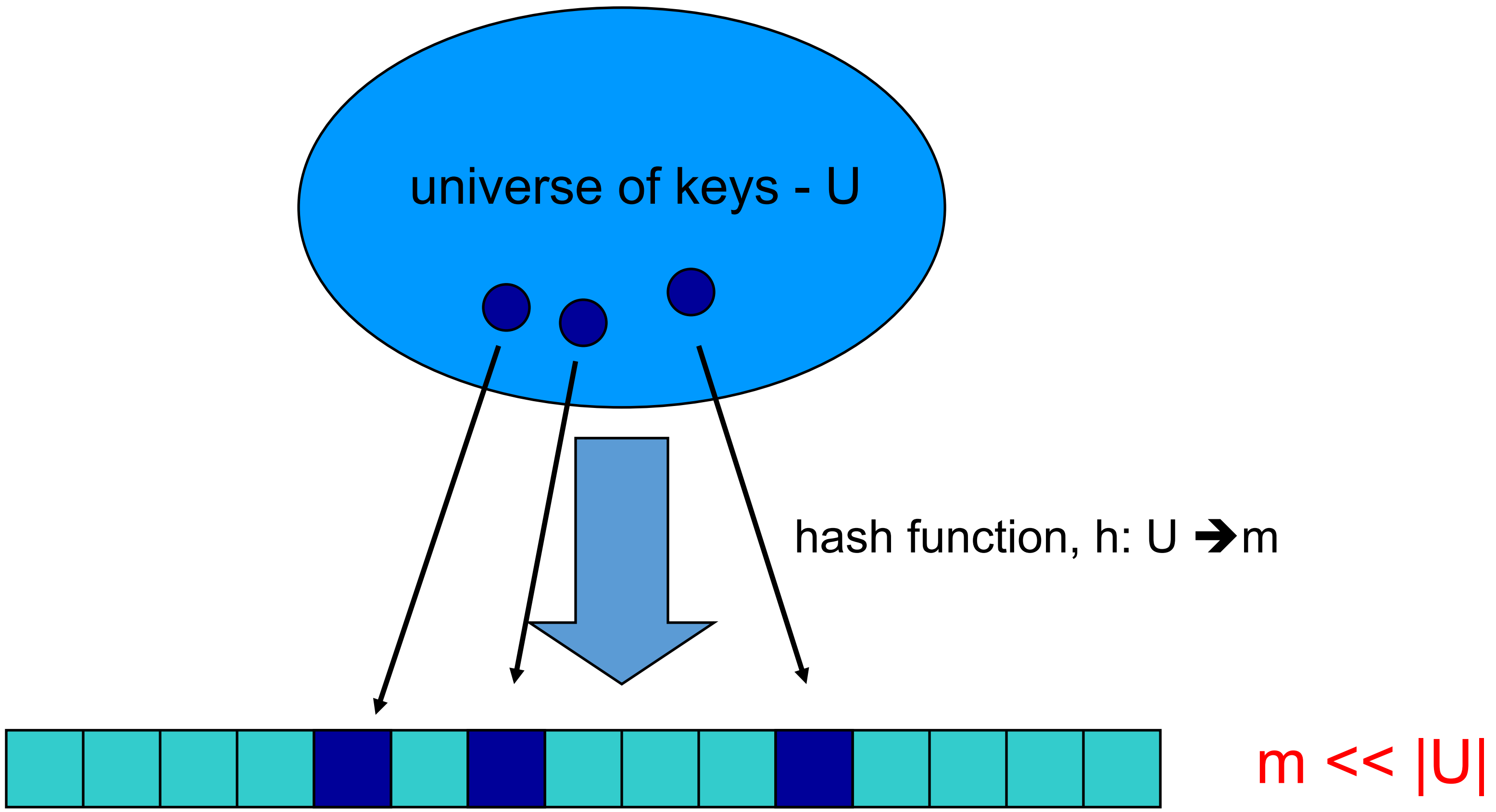
Hash function, *h*

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



Hash function, *h*

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



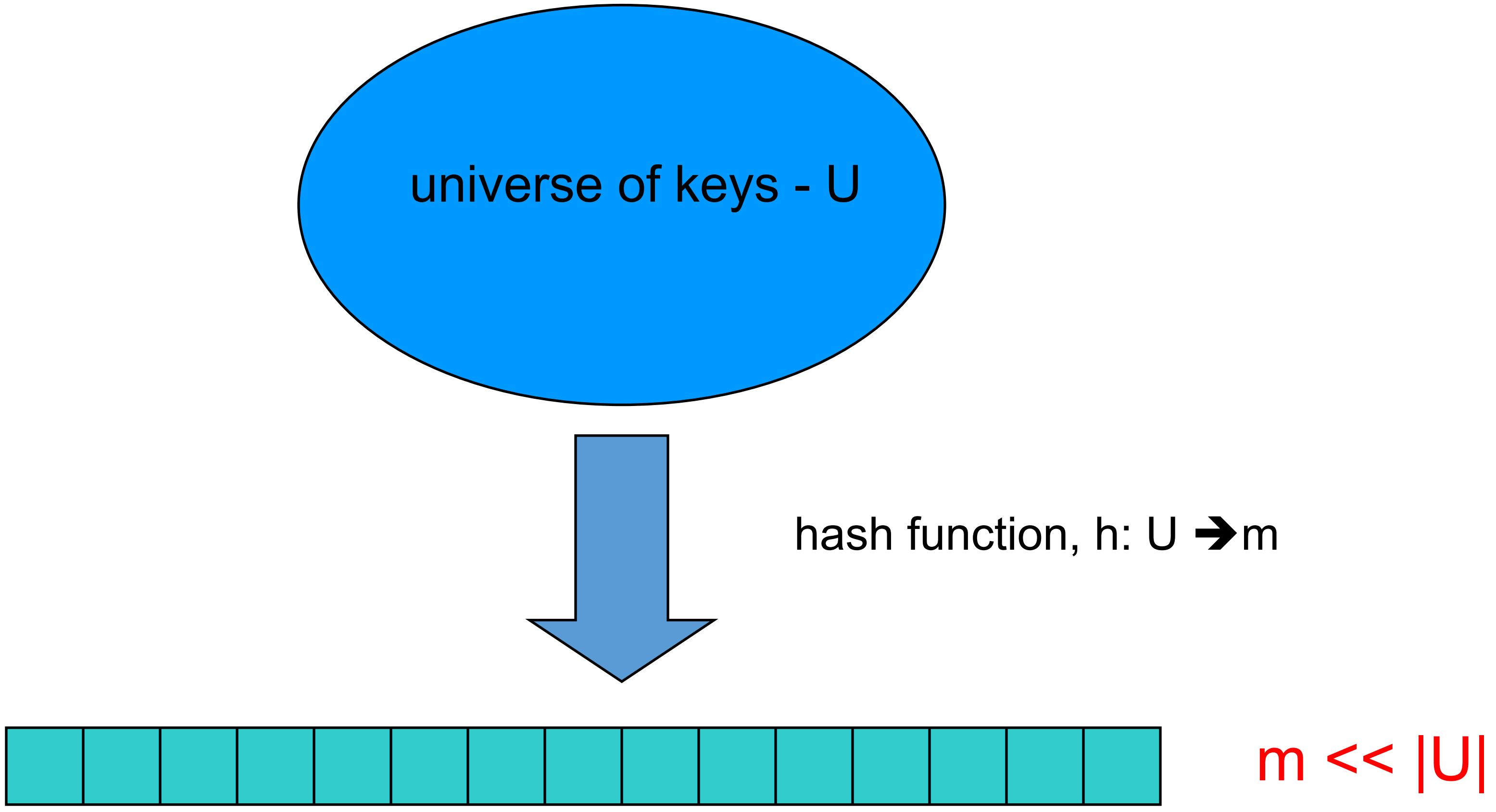
Hash Functions

- U – Universe of all possible keys.
- Hash function h: Mapping from U to the slots of a hash table $T[0..m-1]$.

  $$h : U \to \{0,1,\ldots, m-1\}$$
- With direct addressing, key k maps to slot $T[k]$.
- With hash tables, key k maps or "hashes" to slot $T[h[k]]$.
- $h[k]$ is the *hash value* of key k.

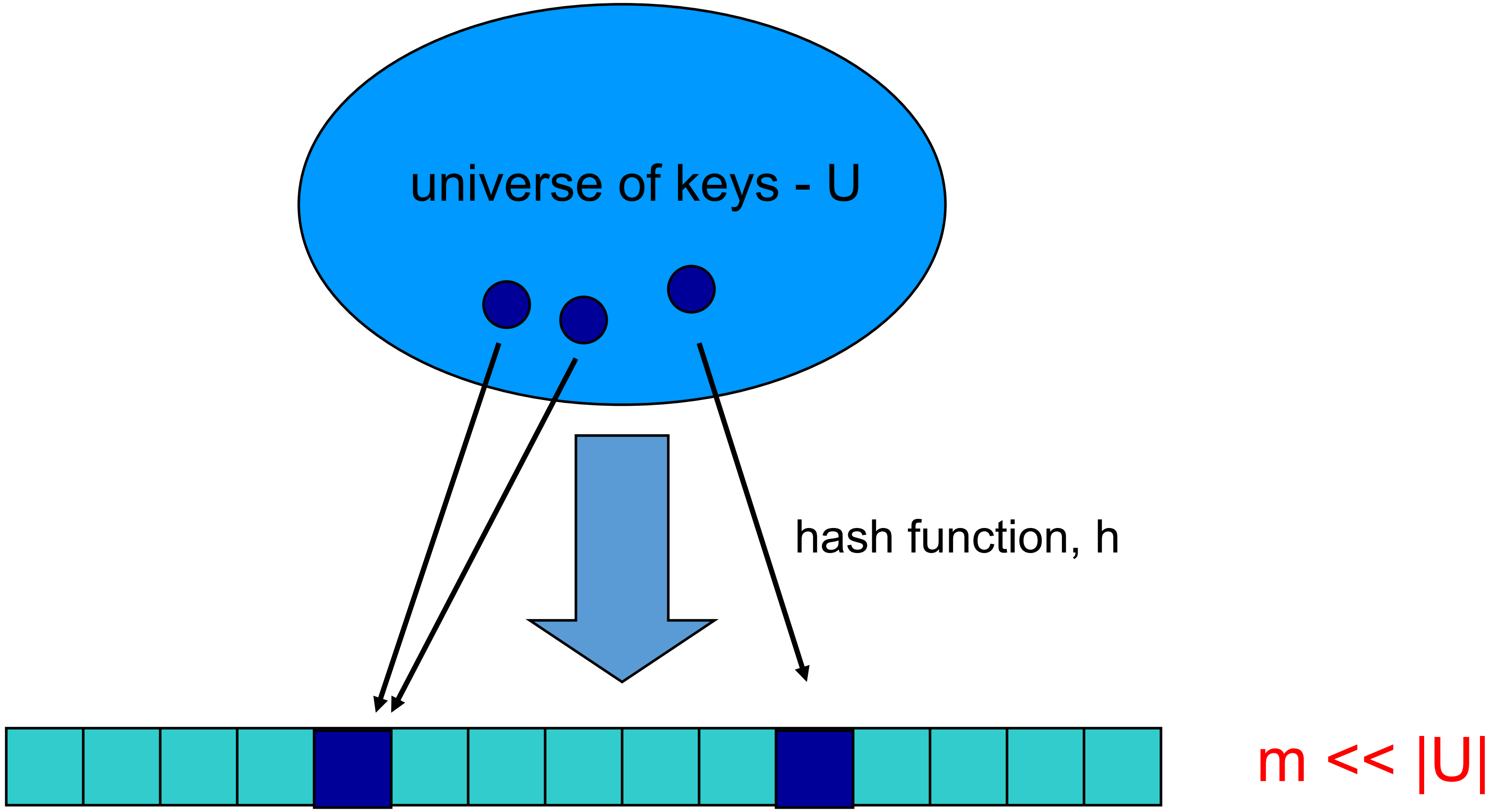
Hash function, *h*

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



Collisions

If $m \neq |U|$, then two keys can map to the same position in the hash table (pidgeon hole principle)



Collisions

A collision occurs when $h(x) = h(y)$, but $x \neq y$

A good hash function will minimize the number of collisions

Collisions are inevitable!

the number of hashtable entries < the possible keys (i.e. $m < |U|$)

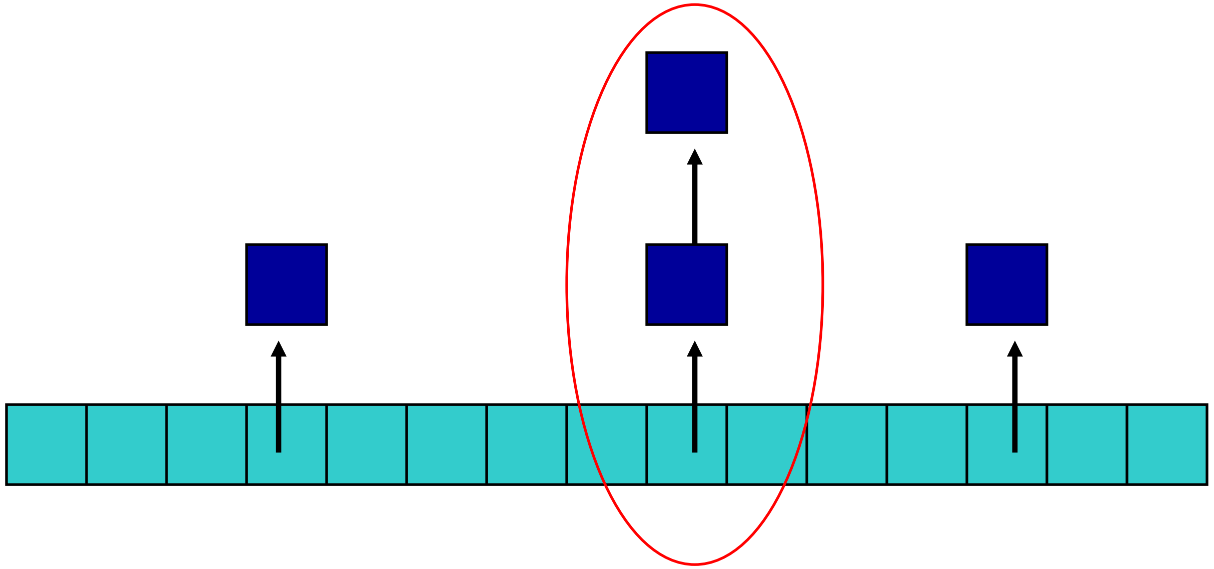
➔Collision resolution techniques?

Resolving Collisions

- *How can we solve the problem of collisions?*
- Solution 1: *chaining*
- Solution 2: *open addressing*

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

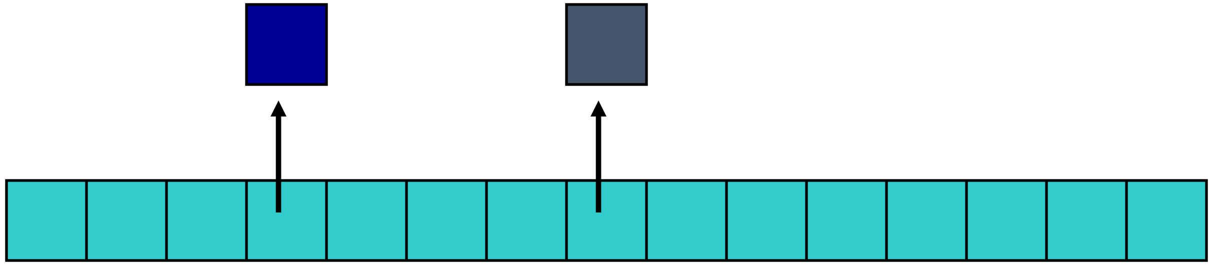
If two entries $x \neq y$ have the same hash value $h(x) = h(x)$, then $T(h(x))$ will contain a linked list with both values

Insertion

$\textsc{ChainedHashInsert}(T, x)$

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

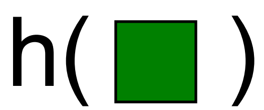
ChainedHashInsert()



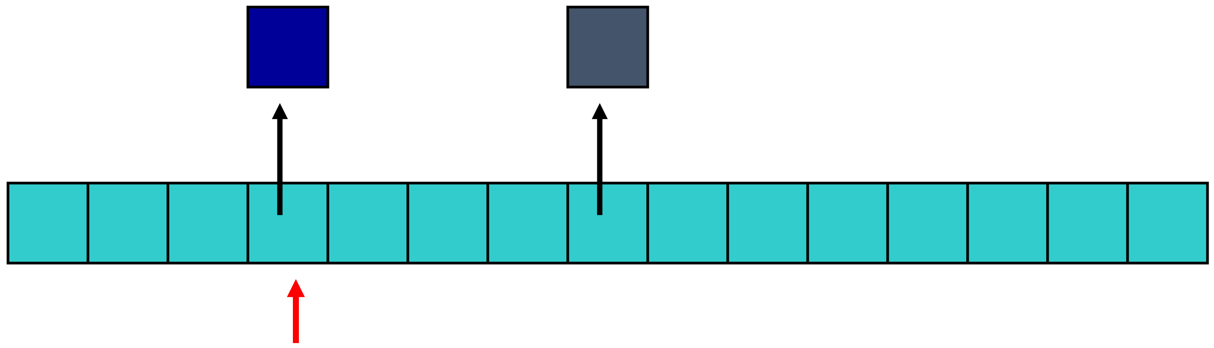
Insertion

$\textsc{ChainedHashInsert}(T, x)$

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



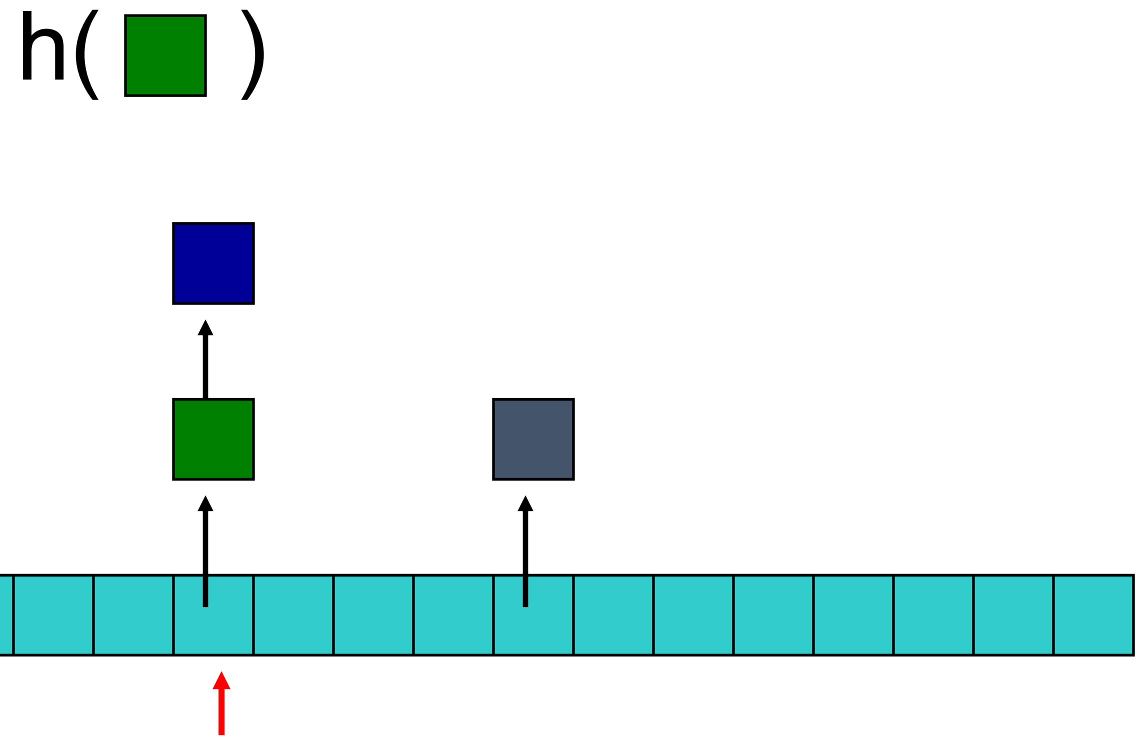
hash function is a mapping from the key to some value < m



Insertion

$\textsc{ChainedHashInsert}(T, x)$

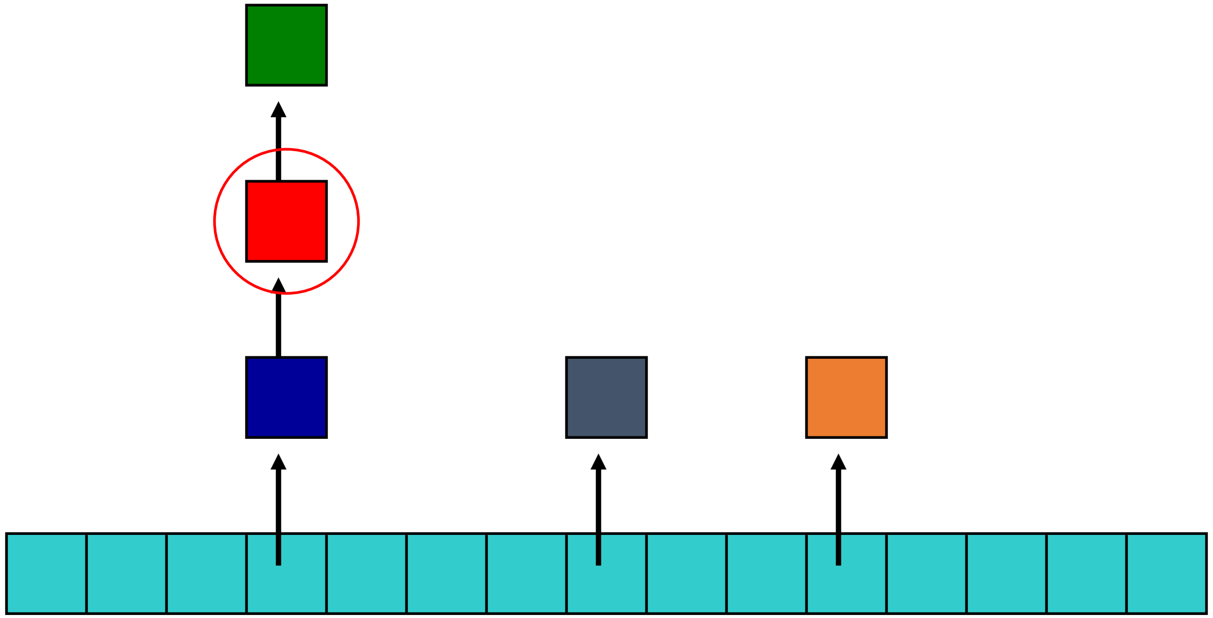
  insert x at the head of list $T[h(x)]$



Deletion

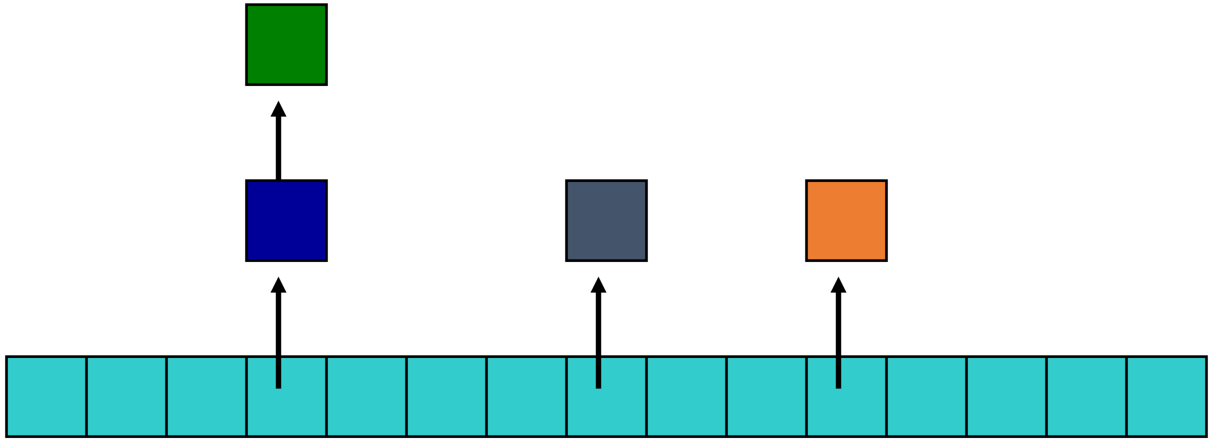
$\textsc{ChainedHashDelete}(T, x)$

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



Deletion

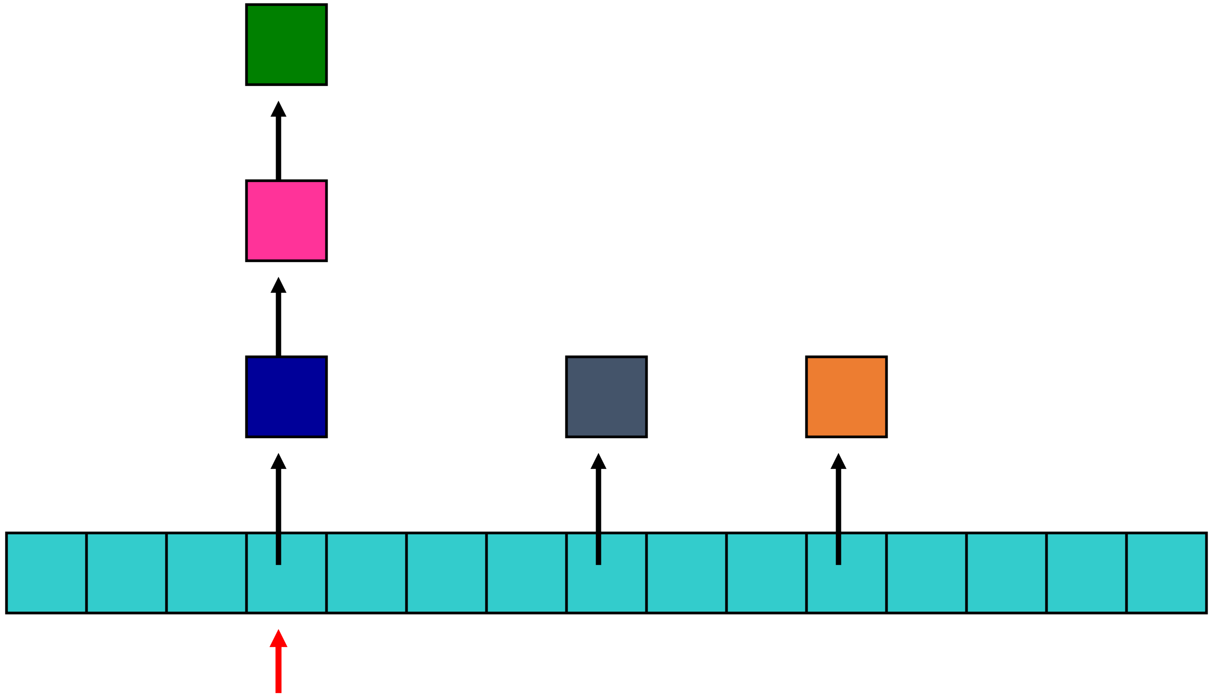
$$\textsc{ChainedHashDelete}(T, x)$$
$$\text{delete } x \text{ from the list } T[h(key[x])]$$



Search

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

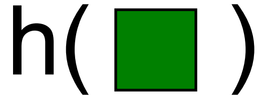
ChainedHashSearch()

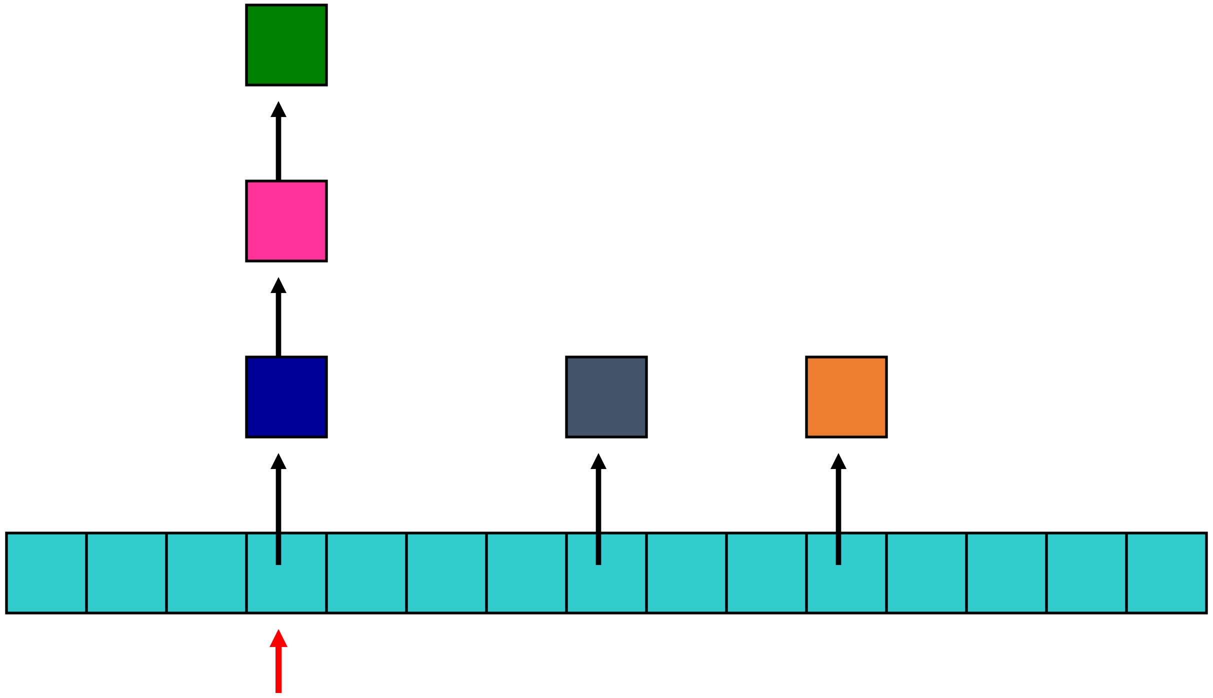


Search

$\textsc{ChainedHashSearch}(T, x)$

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

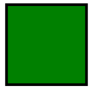


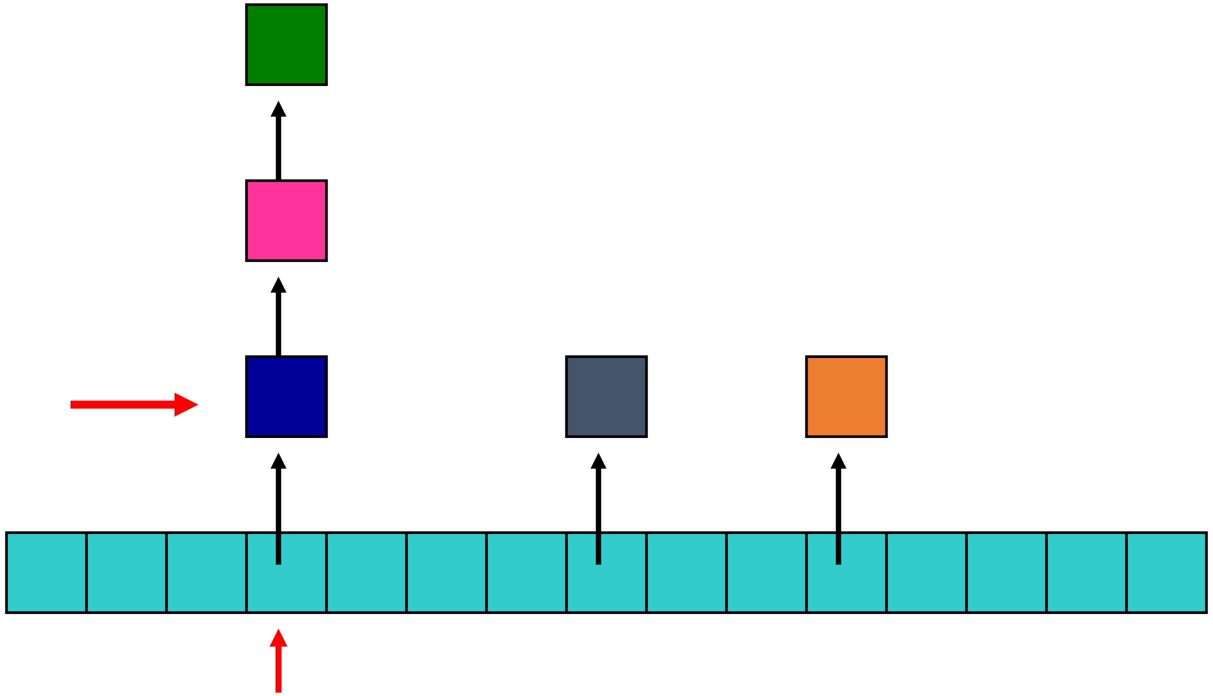


Search

$\textsc{ChainedHashSearch}(T, x)$

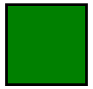
search for x in list $T[h(x)]$

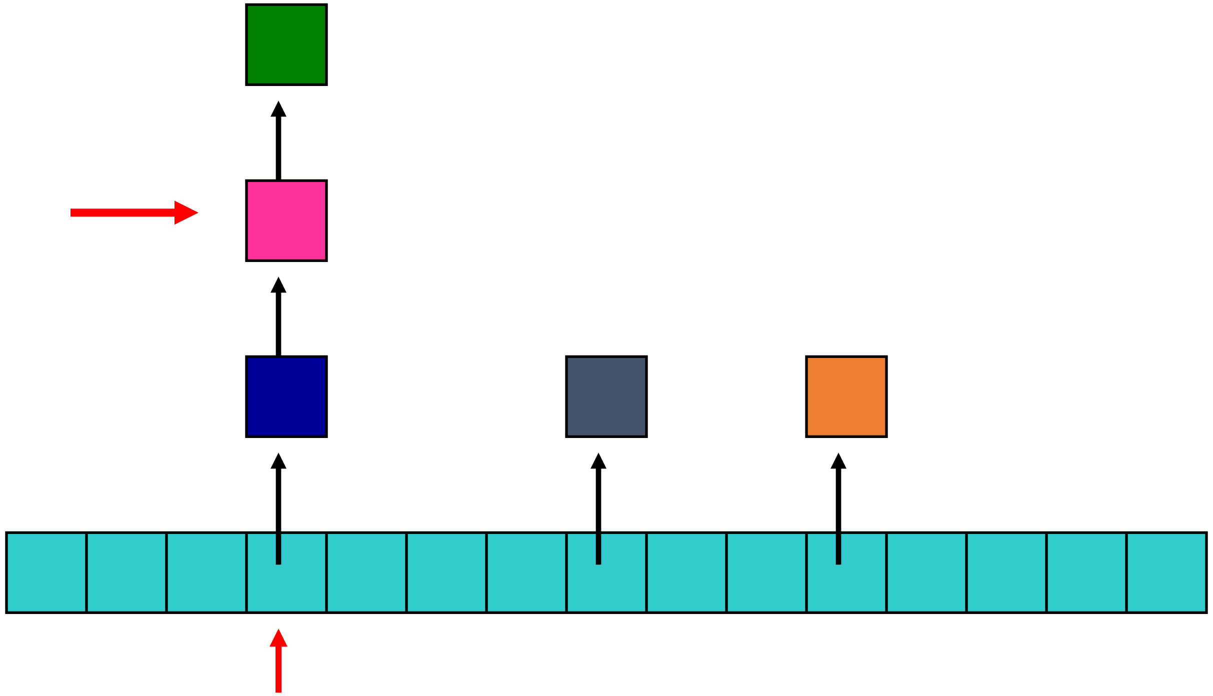
ChainedHashSearch()



Search

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

ChainedHashSearch()

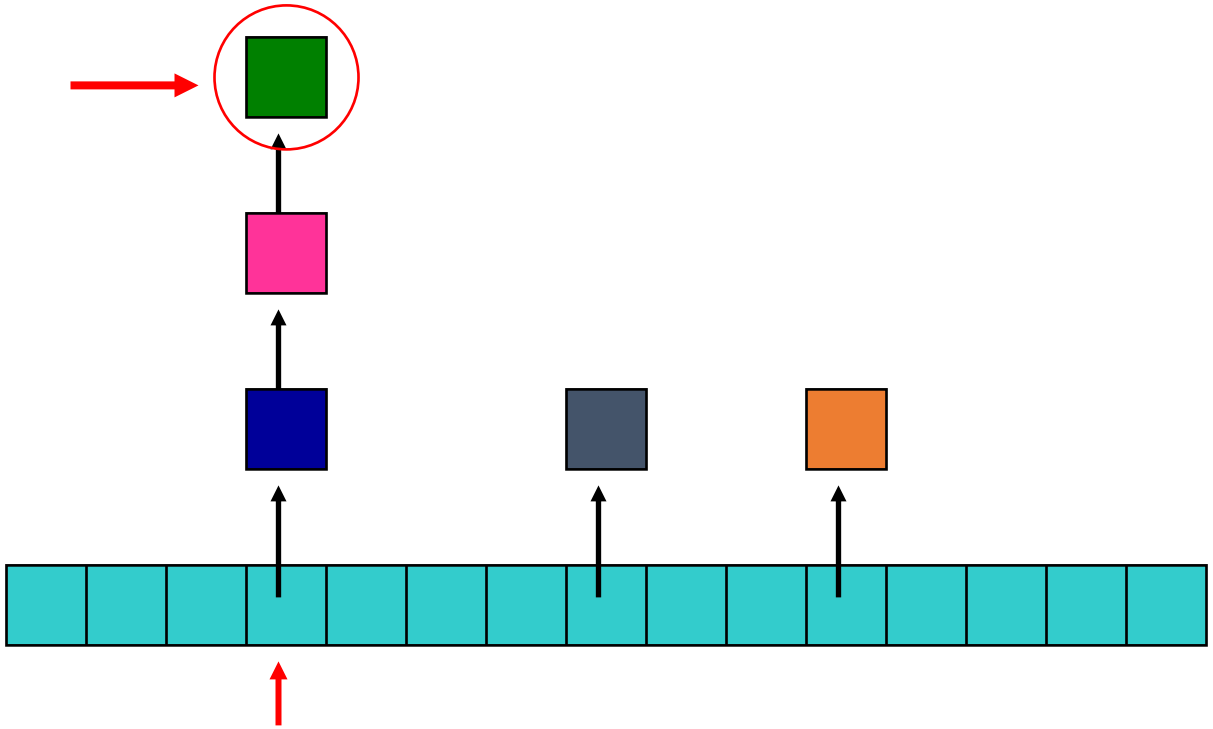


Search

$\textsc{ChainedHashSearch}(T, x)$

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

ChainedHashSearch( )



Hashing with Chaining

- Chained-Hash-Insert (T, x)
  - Insert x at the head of list $T[h(key[x])]$.
  - Worst-case complexity – $O(1)$.
- Chained-Hash-Delete (T, x)
  - Delete x from the list $T[h(key[x])]$.
  - Worst-case complexity – proportional to length of list with singly-linked lists. $O(1)$ with doubly-linked lists.
- Chained-Hash-Search (T, k)
  - Search an element with key k in list $T[h(k)]$.
  - Worst-case complexity – proportional to length of list.

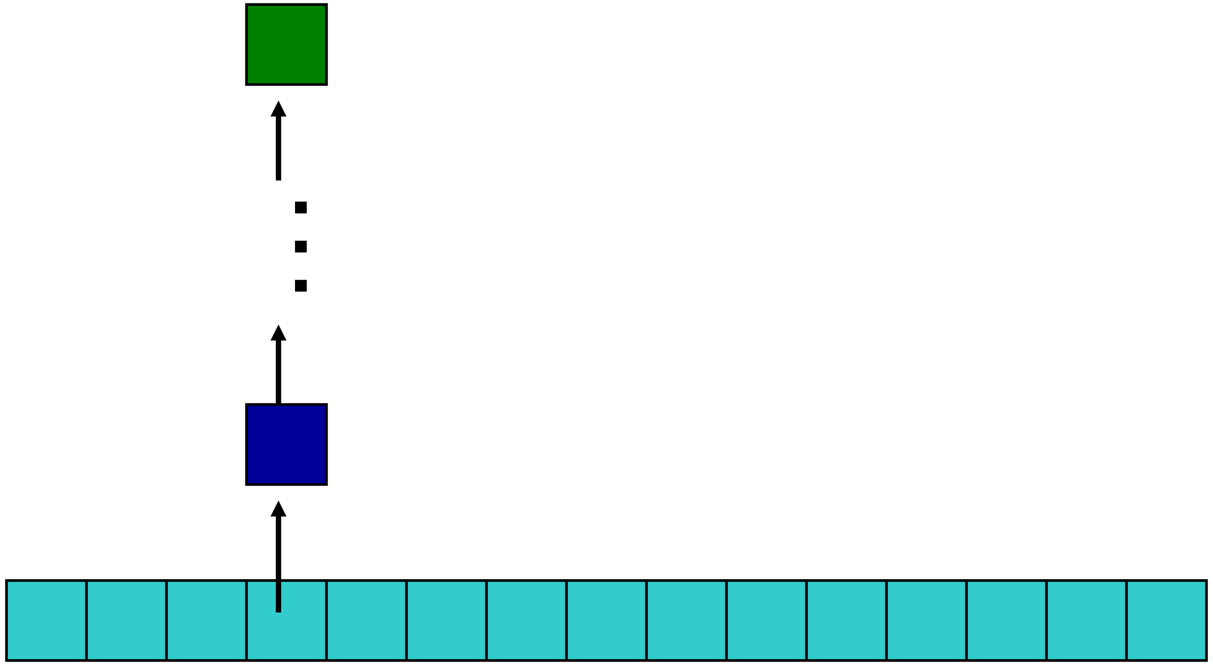
Length of the chain

Worst case?

Length of the chain

Worst case?

- All elements hash to the same location
- h(k) = 4 (a fixed number)
- 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 = $n / m = \alpha$

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
  - $\Theta(1 + \alpha)$
- Key **is** in the table
  - on average search half of the entries
  - $O(1 + \alpha/2)$

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

Choosing A Hash Function

- Clearly, choosing the hash function well is crucial
  - *What will a worst-case hash function do?*
  - *What will be the time to search in this case?*
- *What are desirable features of the hash function?*
  - Should distribute keys uniformly into slots
  - Should not depend on patterns in the data

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

Division method

- $h(k) = k \bmod m$
  - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
  - Example: m = 31 and k = 78, h(k) = 16.
- Advantage: fast
- Disadvantage: value of m is critical
  - Bad if keys bear relation to m
  - *Or if hash does not depend on all bits of k*
- *What happens to elements with adjacent values of k?*
  - Elements with adjacent keys hashed to different slots: good
- *What happens if m is a power of 2 (say 2^P)?*
- *What if m is a power of 10?*
- Pick m = prime number not too close to power of 2 (or 10)

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

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

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

Why a power of 2?

Suggested by Knuth: $A = (\sqrt{5} - 1) / 2 = 0.6180339887$

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

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

Hash Functions: The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m\,(kA \bmod 1) \rfloor = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\textit{What does this term represent?}} \rfloor$

Hash Functions: The Multiplication Method

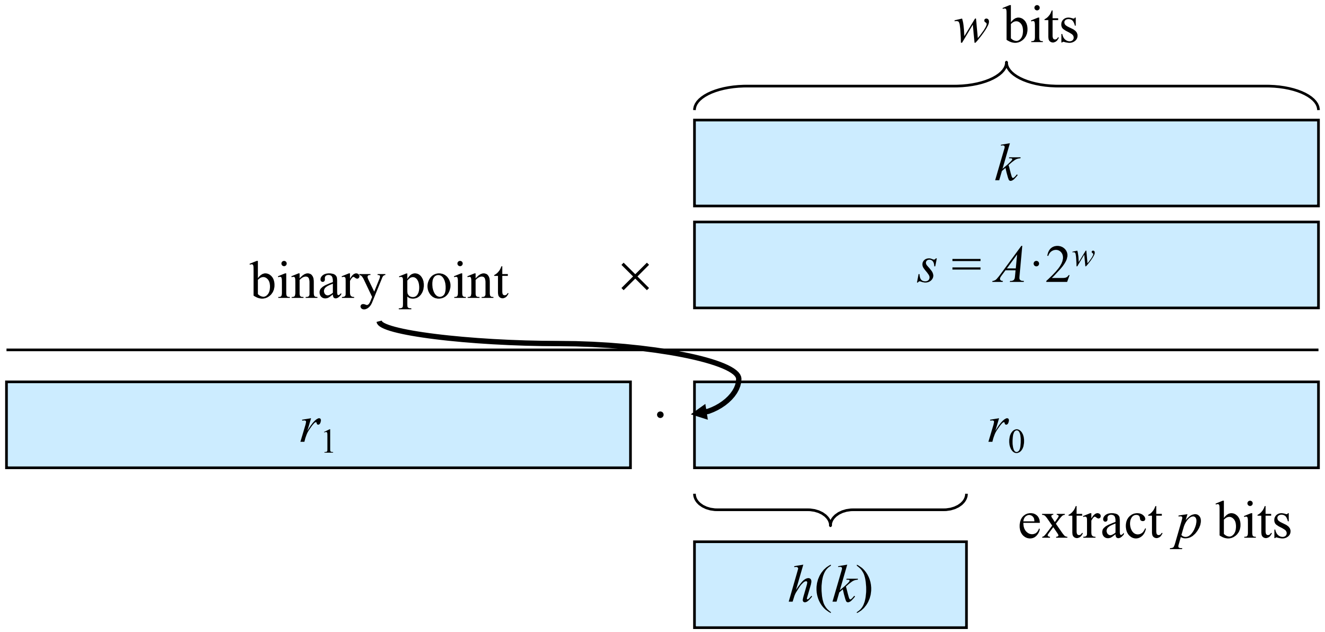
- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m\,(kA \bmod 1) \rfloor = \lfloor m\,\underbrace{(kA - \lfloor kA \rfloor)}_{\textit{Fractional part of } kA} \rfloor$
- Advantage: Value of m is not critical
- Disadvantage: relatively slower
- Choose $m = 2^P$, for easier implementation

Multiplication Method - Implementation

- Choose $m = 2^p$, for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let $0 < s < 2^w$. (s takes w bits.)
- Restrict A to be of the form $s/2^w$ ➔ $s = A \cdot 2^w$
- Let $k \times s = r_1 \cdot 2^w + r_0$.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($kA \bmod 1 = kA - \lfloor kA \rfloor$).
- We don't care about the integer part of kA.
  - So, just use r_0, and forget about r_1.

Multiplication Method – Implementation

- We want $\lfloor m\,(kA \bmod 1)\rfloor$.
- m = 2^p
- We could get that by shifting r_0 to the left by p bits and then taking the p bits that were shifted to the left of the binary point.
- But, we don't need to shift. Just take the p most significant bits of r_0.



Example

- As an example, suppose we have k = 123456, p = 14, m = 2^{14} = 16384, and w = 32.
- Let A be the fraction of the form $s/2^{32}$ that is closest to

  $A = (\sqrt{5} - 1)/2 = 0.6180339887$ ➔ A = 2654435769/ 2^{32}
- k * s = 327706022297664 = 76300 * 2^{32} + 17612864
- r_1 = 76300, r_0 = 17612864
- The 14 most significant bits of r_0 yield the value h(k) = 67.

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)

Hash Functions: Worst Case Scenario

- Scenario:
  - You are given an assignment to implement hashing
  - You will self-grade in pairs, testing and grading your partner's implementation
  - In a blatant violation of the honor code, your partner:
    - Analyzes your hash function
    - Picks a sequence of "worst-case" keys that all map to the same slot, causing your implementation to take O(n) time to search

Exercise 11.2-5: when $|U| > nm$, for any fixed hashing function, can always choose n keys to be hashed into the same slot.

Universal Hashing

- When attempting to defeat a malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly in a way that is independent of the keys that are actually going to be stored
    - pick a hash function randomly when the algorithm begins (*not* upon every insert!)
    - Guarantees good performance on average, no matter what keys adversary chooses
    - Need a family of hash functions to choose from

Universal Hashing

- Let H be a (finite) collection of hash functions
  - …that map a given universe U of keys…
  - …into the range {0, 1, …, m - 1}.
- H is said to be *universal* if:
  - for each pair of distinct keys $x, y \in U$, the number of hash functions h $\in$ H for which $h(x) = h(y)$ is at most $|H|/m$
  - In other words:
    - With a random hash function from H, the chance of a collision between x and y is at most $1/m$ $(x \neq y)$

Universal Hashing

- Theorem 11.3 (modified from textbook):
  - Choose *h* from a universal family of hash functions
  - Hash *n* keys into a table of *m* slots, $n \leq m$
  - Then the expected number of collisions involving a particular key *x* is less than 1
  - Proof:
    - For each pair of keys *y*, *x*, let c_{yx} = 1 if *y* and *x* collide, 0 otherwise
    - $E[c_{yx}]$ <= 1/*m* (by definition)
    - Let C_x be total number of collisions involving key *x*
    - $$E[C_x] = \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] \leq \frac{n-1}{m}$$
    - Since $n \leq m$, we have $E[C_x] < 1$
- Implication, expected running time of insertion is $\Theta(1)$

A Universal Hash Function

- Choose a prime number p that is larger than all possible keys
- Choose table size $m \geq n$
- Randomly choose two integers a, b, such that
  $1 \leq a \leq p - 1$, and $0 \leq b \leq p - 1$
- $h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$
- Example: $p = 17$, $m = 6$

  $h_{3,4}(8) = ((3*8 + 4) \% 17) \% 6 = 11 \% 6 = 5$

A universal hash function

- Theorem 11.5: The family of hash functions $H_{p,m} = \{h_{a,b}\}$ defined on the previous slide is universal
- Proof sketch:
  - For any two distinct keys x, y, for a given $h_{a,b}$,
  - Let r = (ax+b) % p, s = (ay+b) % p.
  - Can be shown that r$\neq$s, and different (a,b) results in different (r,s)
    - x and y collides only when r%m = s%m
    - For a given r, the number of values s such that r%m = s%m and r $\neq$ s is at most (p-1)/m
    - For a given r, and any randomly chosen s,

    prob(r $\neq$ s & r%m = s%m) = (p-1) / m / (p-1) = 1/m

Resolving Collisions

- *How can we solve the problem of collisions?*
- Solution 1: *chaining*
- Solution 2: *open addressing*

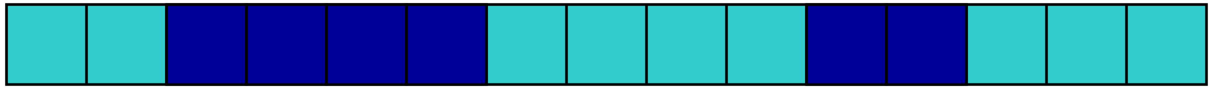
Open addressing

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

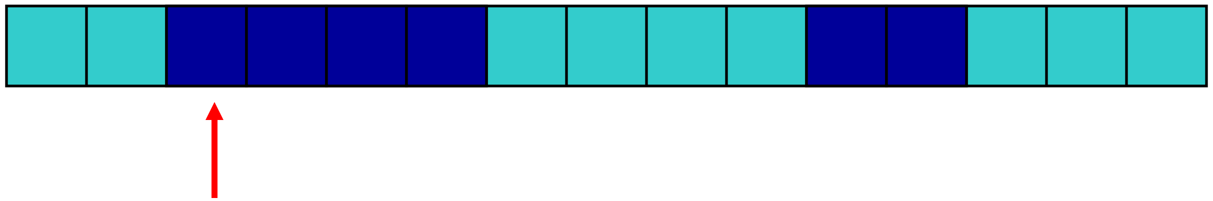
- Hash function must define a **probe sequence** which is the list of slots to examine when searching or inserting

- 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.

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

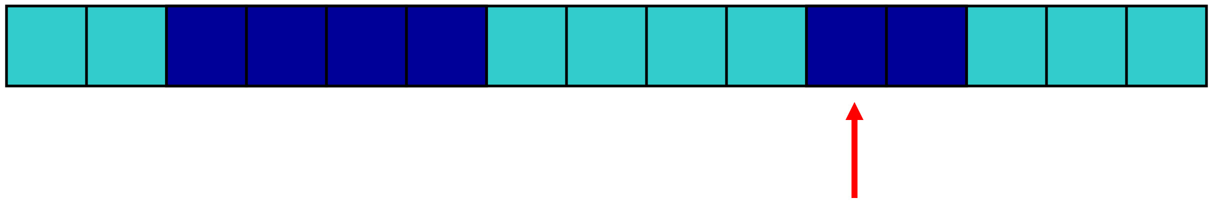
Probe sequence

h(k, 0)



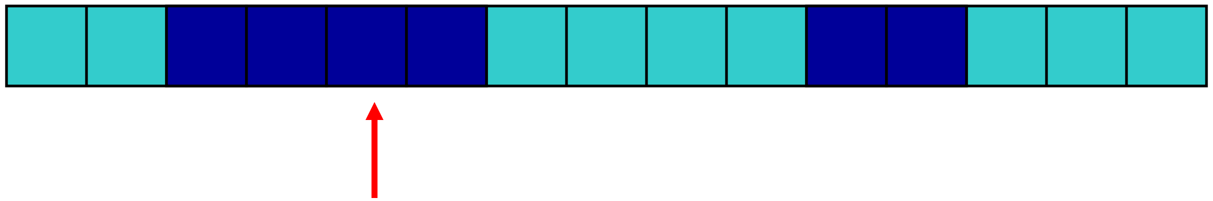
Probe sequence

$h(k, 1)$



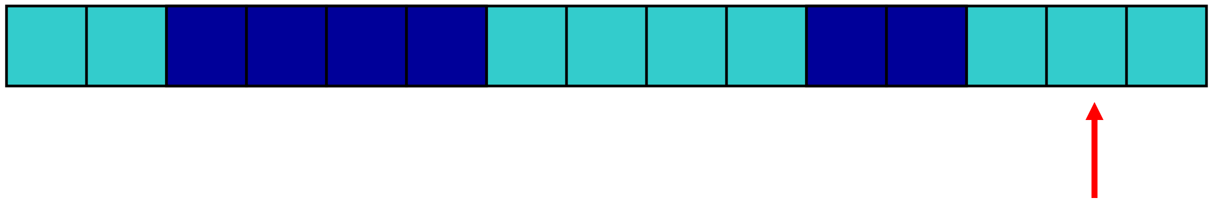
Probe sequence

h(k, 2)



Probe sequence

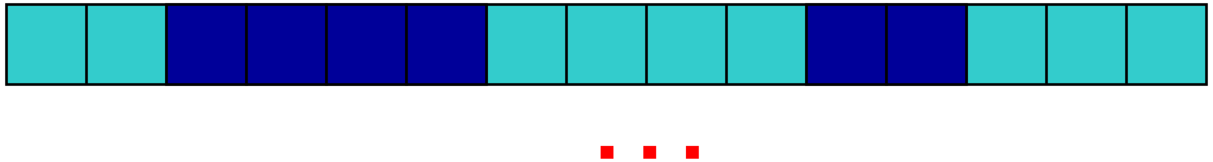
$$h(k, 3)$$



Probe sequence

h(k, …)

must visit all locations



Open addressing: Insert

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

Open addressing: Insert

```
Hash-Insert(T, k)
i ← 0
j ← h(k, i)
while i < m − 1 and T[j] ≠ null
        i ← i + 1
        j ← 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 ← 0
    j ← h(k, i)
    while i < m − 1 and T[j] ≠ null
        i ← i + 1
        j ← 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 ← 0
j ← h(k, i)
while i < m − 1 and T[j] ≠ null
        i ← i + 1
        j ← 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 ← 0
j ← h(k, i)
while i < m − 1 and T[j] ≠ null
        i ← i + 1
        j ← 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 $null$

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 ← 0
j ← h(k, i)
while i < m − 1 and T[j] ≠ null and T[j] ≠ k
        i ← i + 1
        j ← h(k, i)
if T[j] = k
        return j
else
        return null
```

HASH-INSERT(T, k)

```
i ← 0
j ← h(k, i)
while i < m − 1 and T[j] ≠ null
        i ← i + 1
        j ← h(k, i)
if T[j] = null
        return j
else
        error "hash is full"
```

"breaks" the probe sequence

Open addressing: Delete

- Problem:
  - If we simply delete a key, then search may fail.

- Solution:
  - slots of deleted keys are marked specially as “**deleted**” (not “null”).
  - modify search procedure to continue looking if a “deleted” node is seen
  - Insert can insert an item in a deleted slot, but search doesn’t stop at a deleted slot (but search time increases)

- if a lot of deleting will happen, use chaining

Probing schemes

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

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

h': U ➔ {*0, 1, ….., m-1*} is referred to as an auxiliary hash function

for example, m = 7 and h(k) = 4

$h(k,0) = 4$

$h(k,1) = 5$

$h(k,2) = 6$

$h(k,3) = 0$

$h(k,3) = 1$

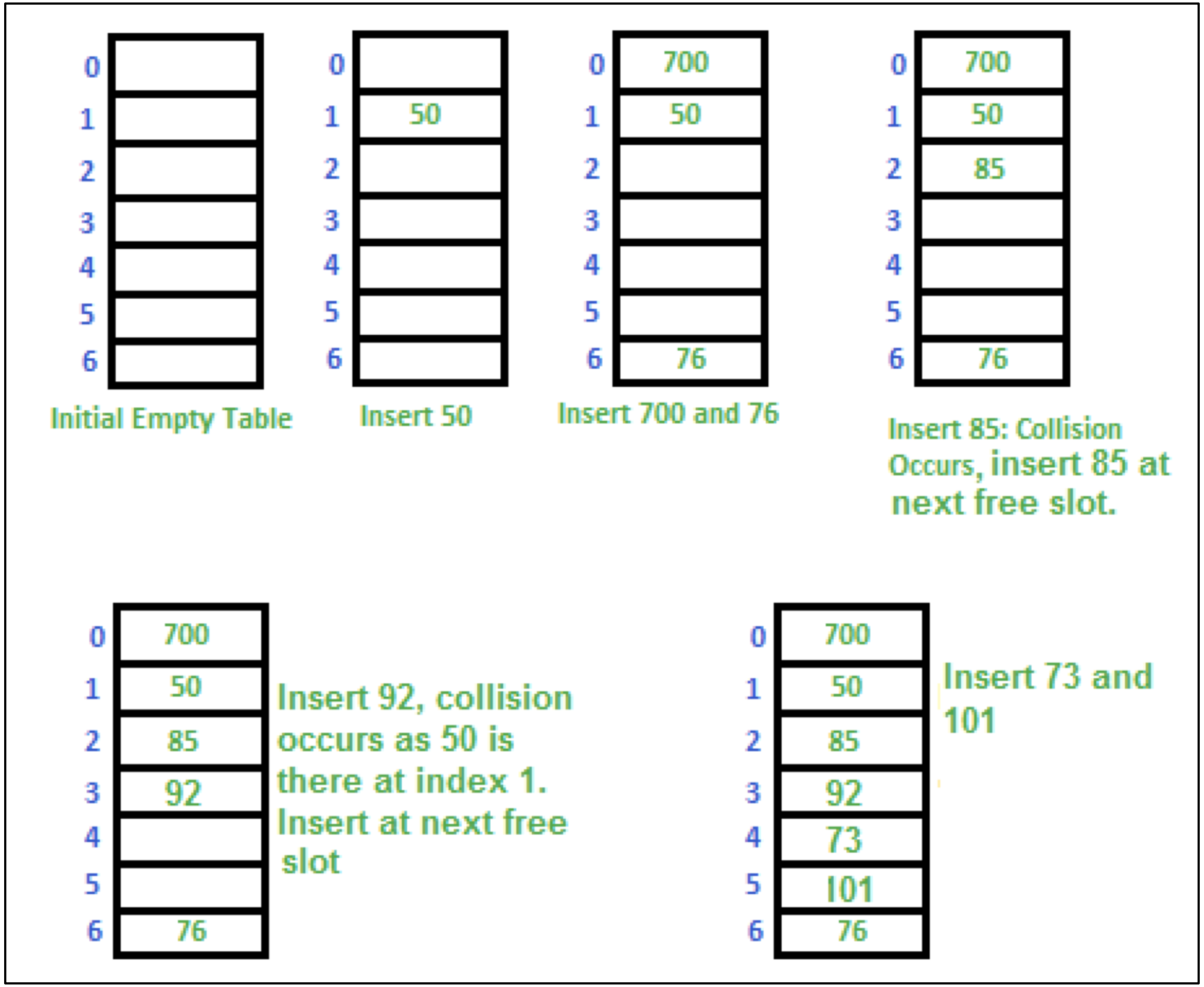
Example

- Given a hash table of size 16, with hash function h(x) = x mod 16, we want to insert prime numbers in sequence starting at 11 (i.e., 11, 13, 17, 19, 23, 29, $\cdots$) until two collisions occur. Show the evolution of the contents of the array with collision handled by linear probing.

Example

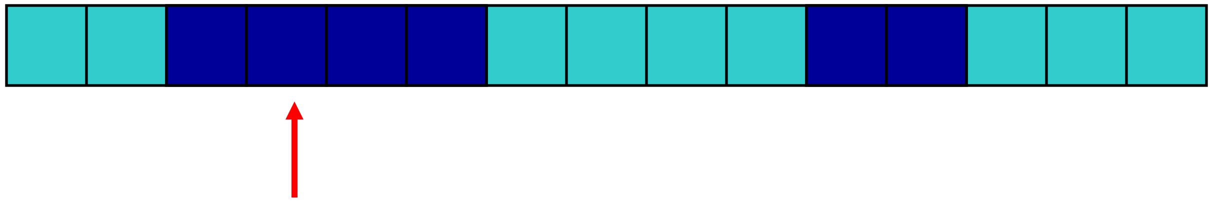
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
											11				
											11		13		
	17										11		13		
	17		19								11		13		
	17		19				23				11		13		
	17		19				23				11		13	29	
	17		19				23				11		13	29	31
	17		19		37		23				11		13	29	31
	17		19		37		23		41		11		13	29	31
	17		19		37		23		41		11	43	13	29	31

Example 2: consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



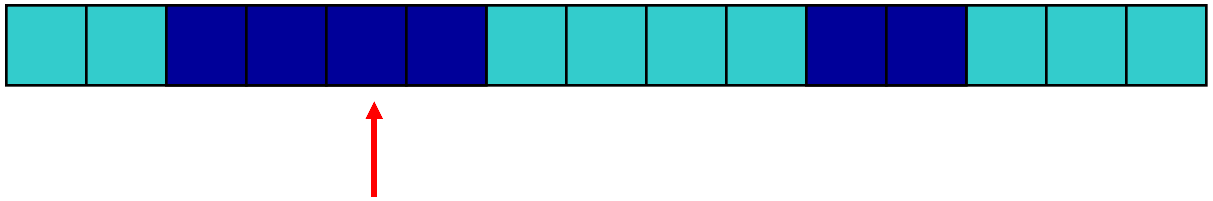
Linear probing: search

h(, 0)



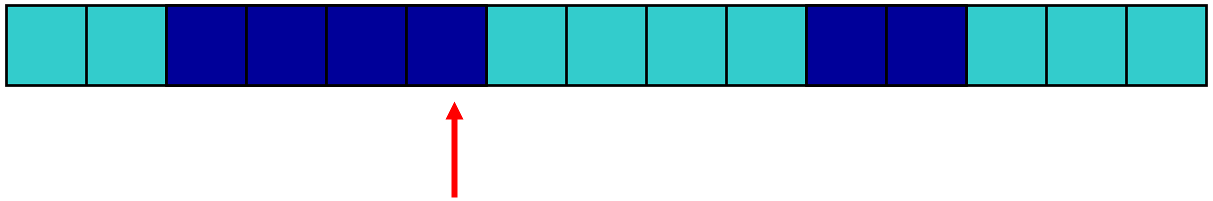
Linear probing: search

h(, 1)



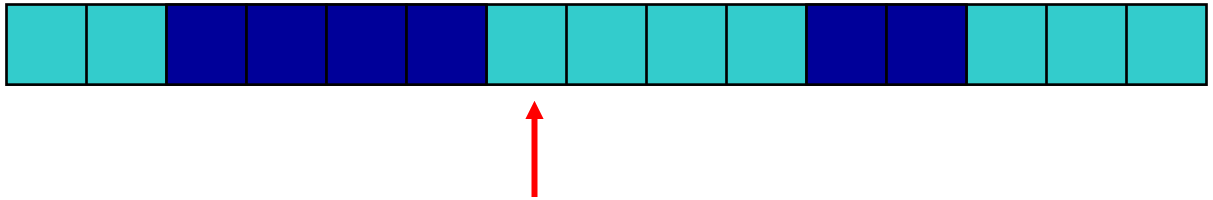
Linear probing: search

h(, 2)



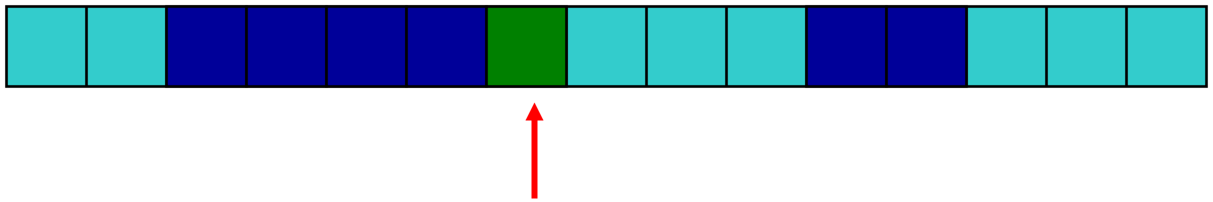
Linear probing: search

h(, 3)



Linear probing: search

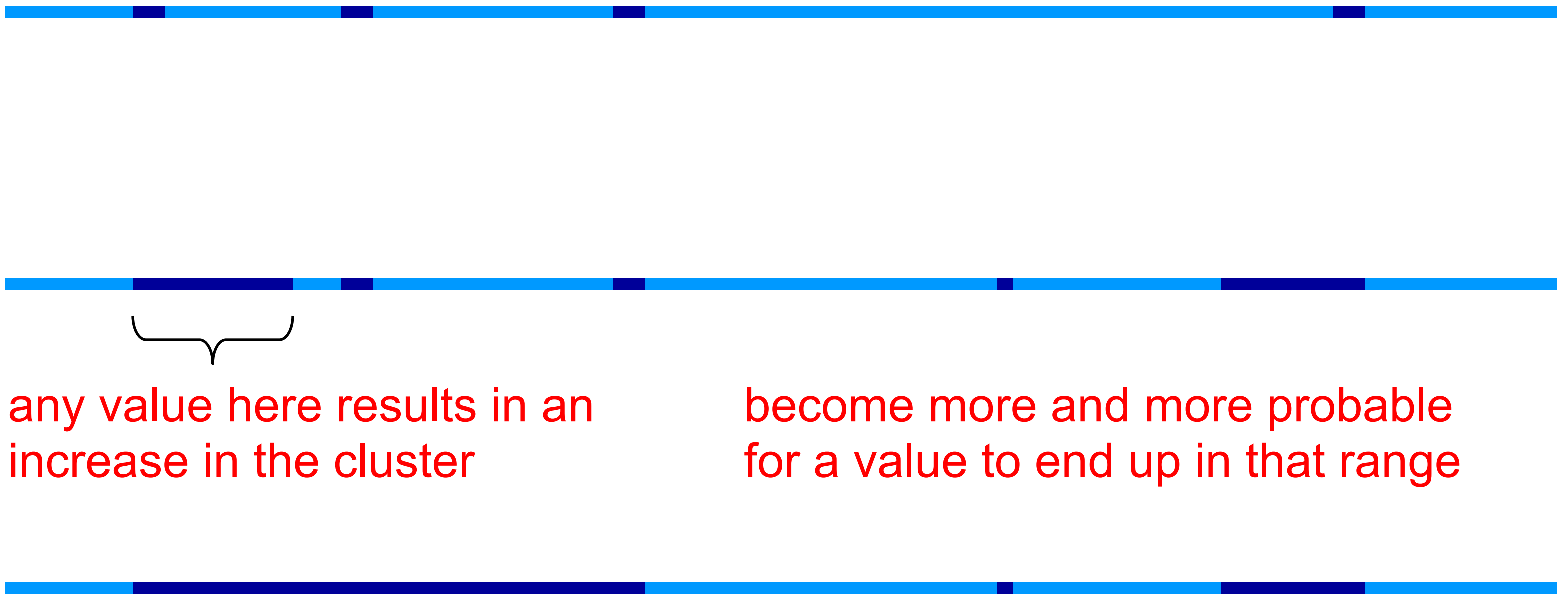
h(, 3)



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

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

- both h_1 and h_2 are auxiliary hash functions
- offers one of the best methods available for open addressing
- initial probe goes to position $T[\ h_1(k)]$
- successive probe positions are offset from previous positions by the amount $h_2(k)$ modulo *m*

Double hashing

Requirement:

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

➔$h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched

➔Let m be a power of 2 and to design h_2 so that it always produces an odd number

Example:

$h_1(k) = k \bmod m$;
$h_2(k) = 1 + (k \bmod m')$

m = 701, m'=700, k = 123456 ➔ $h_1(k) = 80$, $h_2(k) = 257$

Example

- Consider a table of size 13, with double hashing function:
- $h(k, i) = (h_1(k) + i\ h_2(k))\ mod$ 13, where i = 0, 1, 2, …, 12,
- $h_1(k) = k\ mod$ 13
- $h_2(k)$ = 1 + ($k\ mod$ 11).
- Discuss how the following sequence of keys will be mapped into the table: 61, 35, 23, 55, 49, 81, 68

Example

Key	$H_1(k)$	$H_2(k)$	$H_1(k)+H_2(k)$	$H_1(k)+2*H_2(k)$	$H_1(k)+3*H_2(k)$	$H_1(k)+4*H_2(k)$
61	9	7	3	10	4	11
35	9	3	12	2	5	8
23	10	2	12	1	3	5
55	3	1	4	5	6	7
49	10	6	3	9	2	8
81	3	5	8	0	5	10
68	3	3	6	9	12	2

Running time of insert and search for open addressing

Depends on the hash function/probe sequence

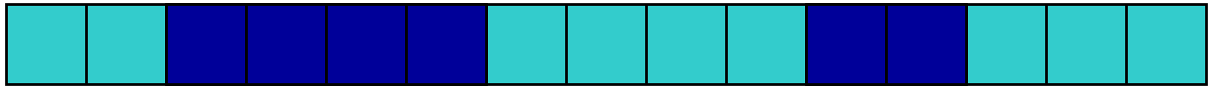
Worst case?

- 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

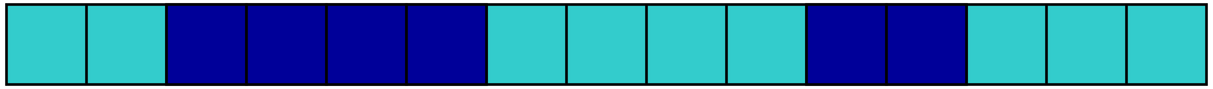


Running time of insert and search for open addressing

Average case?

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

α



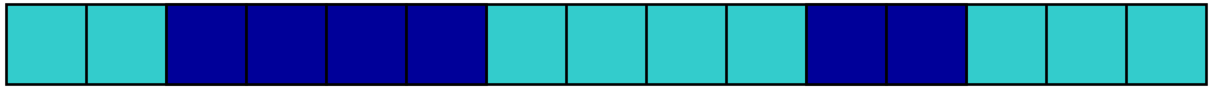
Running time of insert and search for open addressing

Average case?

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

$\sim\alpha^2$

why '~'?

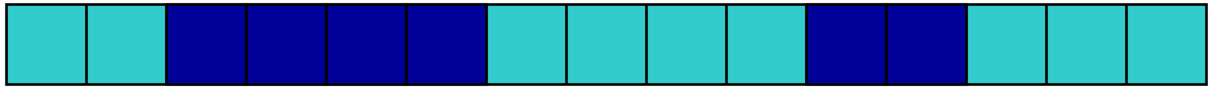


Running time of insert and search for open addressing

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 hash table should be around half full

What happens when the hash table 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