Hashing

The Search Problem

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key x, find the index i such as $x=A[i]$
  - As in the case of sorting, a key could be part of a large record.

example of a record

Key	**other data**

Applications

- Keeping track of customer account information at a bank
  - Search through records to check balances and perform transactions
- Keep track of reservations on flights
  - Search to find empty seats, cancel/modify reservations
- Search engine
  - Looks for all documents containing a given word

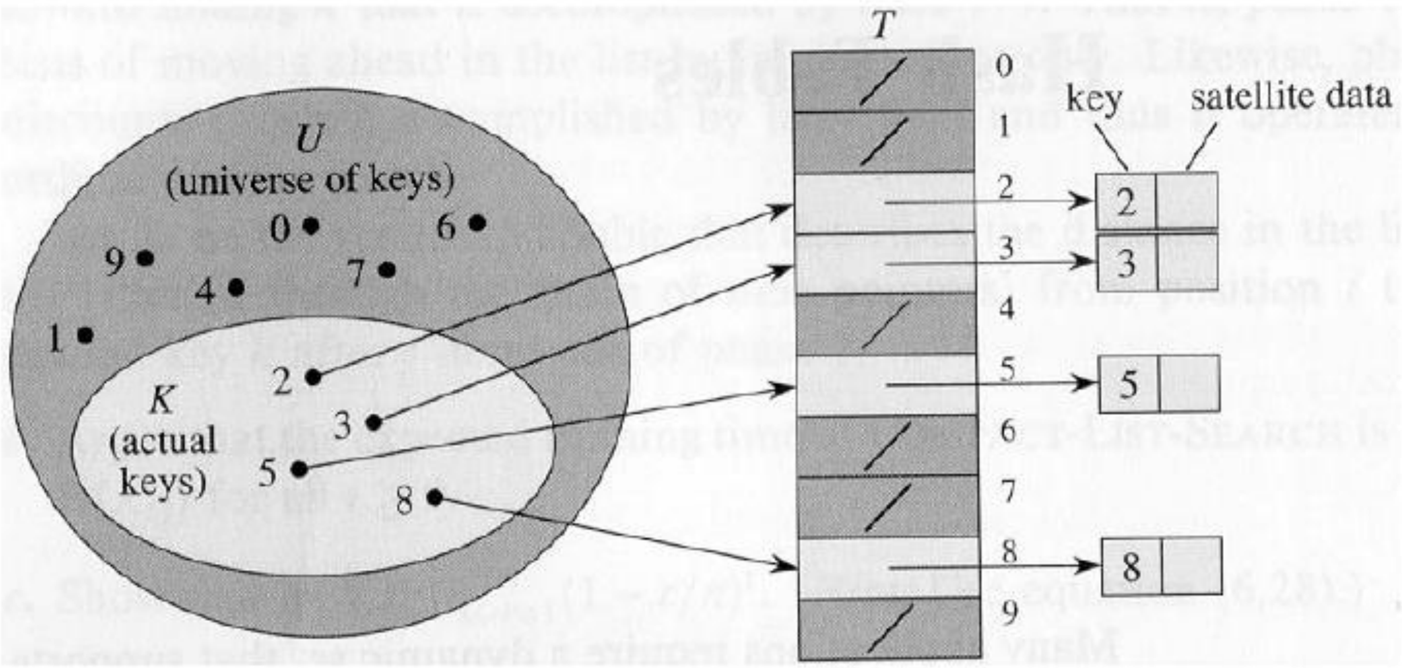
Special Case: Dictionaries

- **Dictionary** = data structure that supports mainly two basic operations: insert a new item and return an item with a given key
- Queries: return information about the set S:
  - Search (S, k)
  - Minimum (S), Maximum (S)
  - Successor (S, x), Predecessor (S, x)
- Modifying operations: change the set
  - Insert (S, k)
  - Delete (S, k) – not very often

Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe U = {0, 1, . . . , m - 1}
- Idea:
  - Store the items in an array, indexed by keys
- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NIL

Direct Addressing (cont'd)



(insert/delete in O(1) time)

Operations

Alg.: DIRECT-ADDRESS-SEARCH(T, k)

return T[k]

Alg.: DIRECT-ADDRESS-INSERT(T, x)

T[key[x]] $\leftarrow$ x

Alg.: DIRECT-ADDRESS-DELETE(T, x)

T[key[x]] $\leftarrow$ NIL

- Running time for these operations: $O(1)$

Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists

	Insert	Search
direct addressing	O(1)	O(1)
ordered array	O(N)	O(lgN)
ordered list	O(N)	O(N)
unordered array	O(1)	O(N)
unordered list	O(1)	O(N)

Examples Using Direct Addressing

Example 1:

(i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records

(ii) Create an array A of 100 items and store the record whose key is equal to i in $A[i]$

Example 2:

(i) Suppose that the keys are nine-digit social security numbers

(ii) We can use the same strategy as before but it very inefficient now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large
- $|K|$ can be much smaller than $|U|$

Hash Tables

- When K is much smaller than U, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the average case, not the worst case

Hash Tables

Idea:

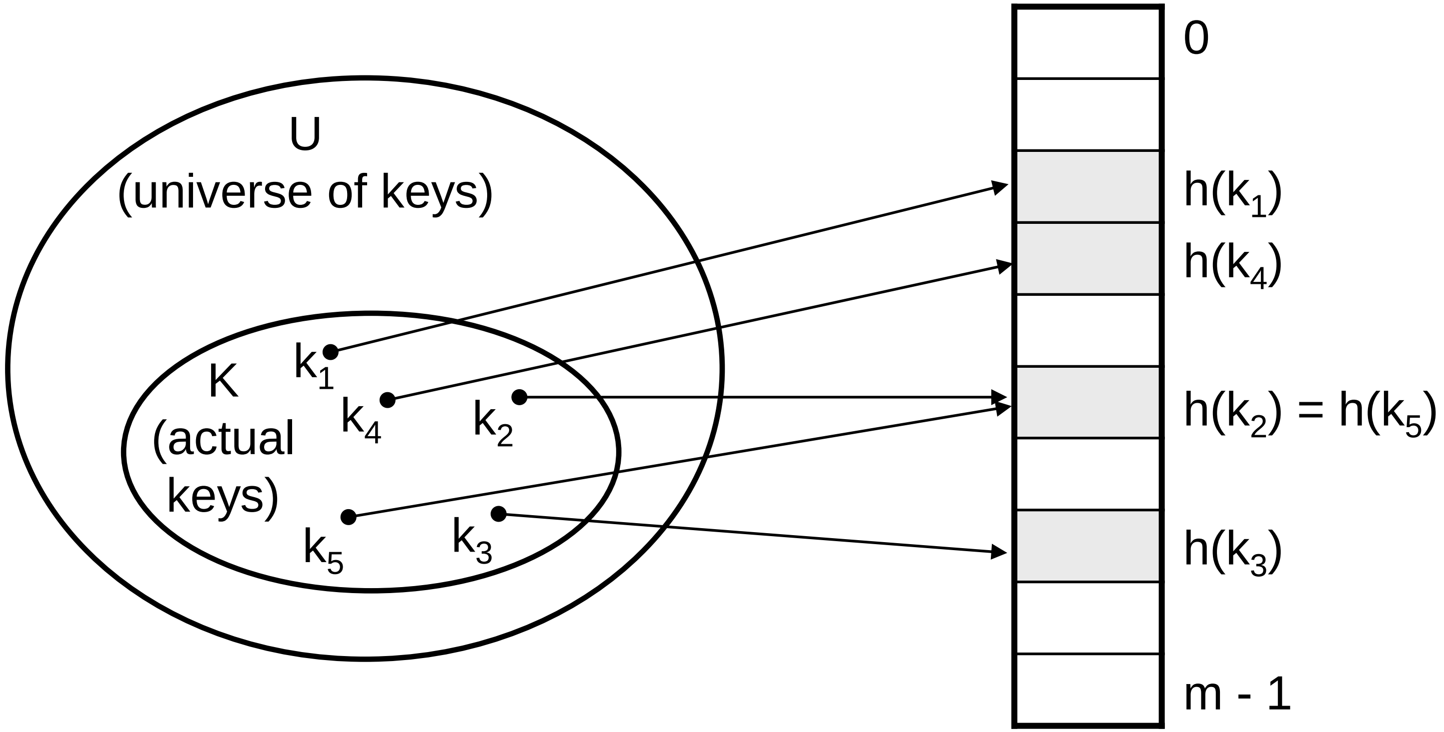
- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

$$h : U \rightarrow \{0, 1, . . . , m - 1\}$$

- We say that k **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: m instead of $|U|$
  - Storage is also reduced

Example: HASH TABLES



Revisit Example 2

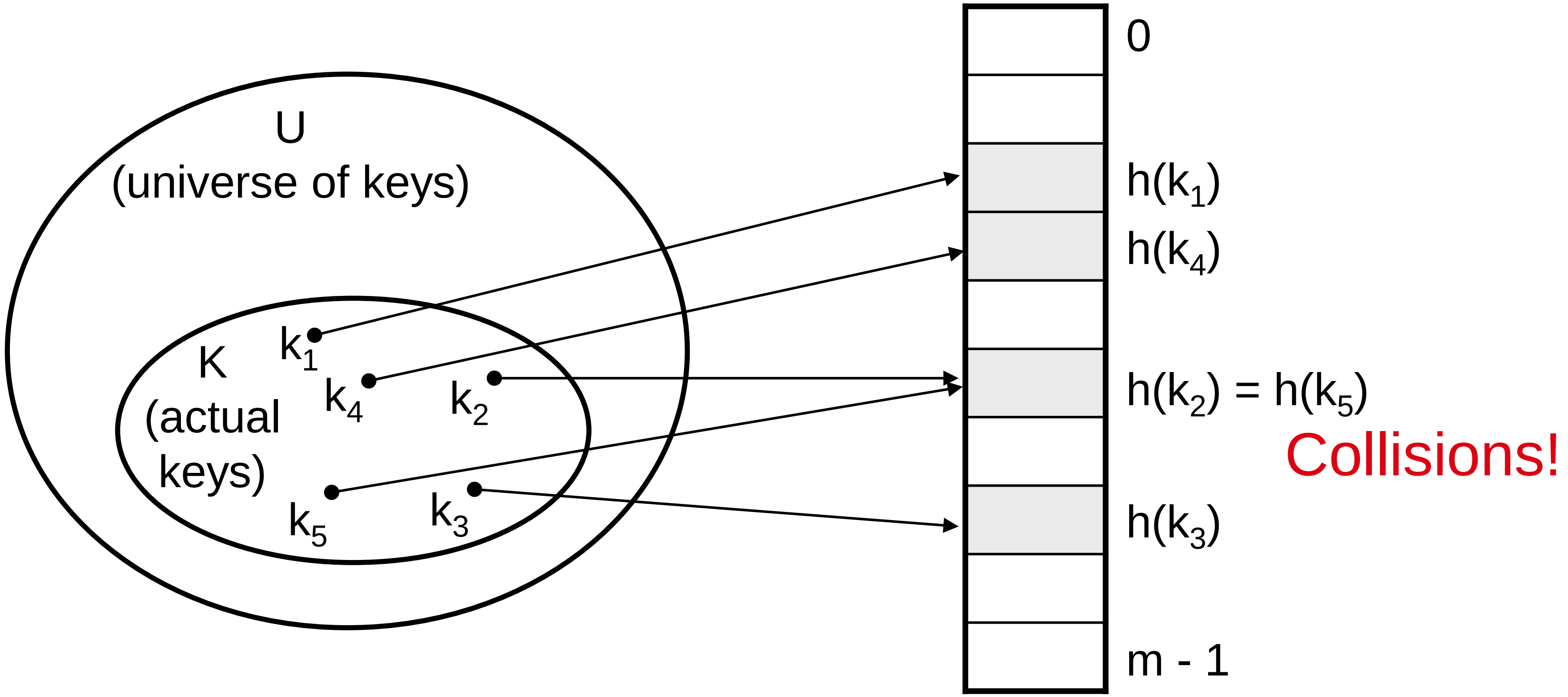
Suppose that the keys are nine-digit social security numbers

<u>Possible hash function</u>

$h(ssn) = sss \bmod 100$ (last 2 digits of ssn)

e.g., if $ssn = 10123411$ then $h(10123411) = 11$)

Do you see any problems with this approach?



Collisions

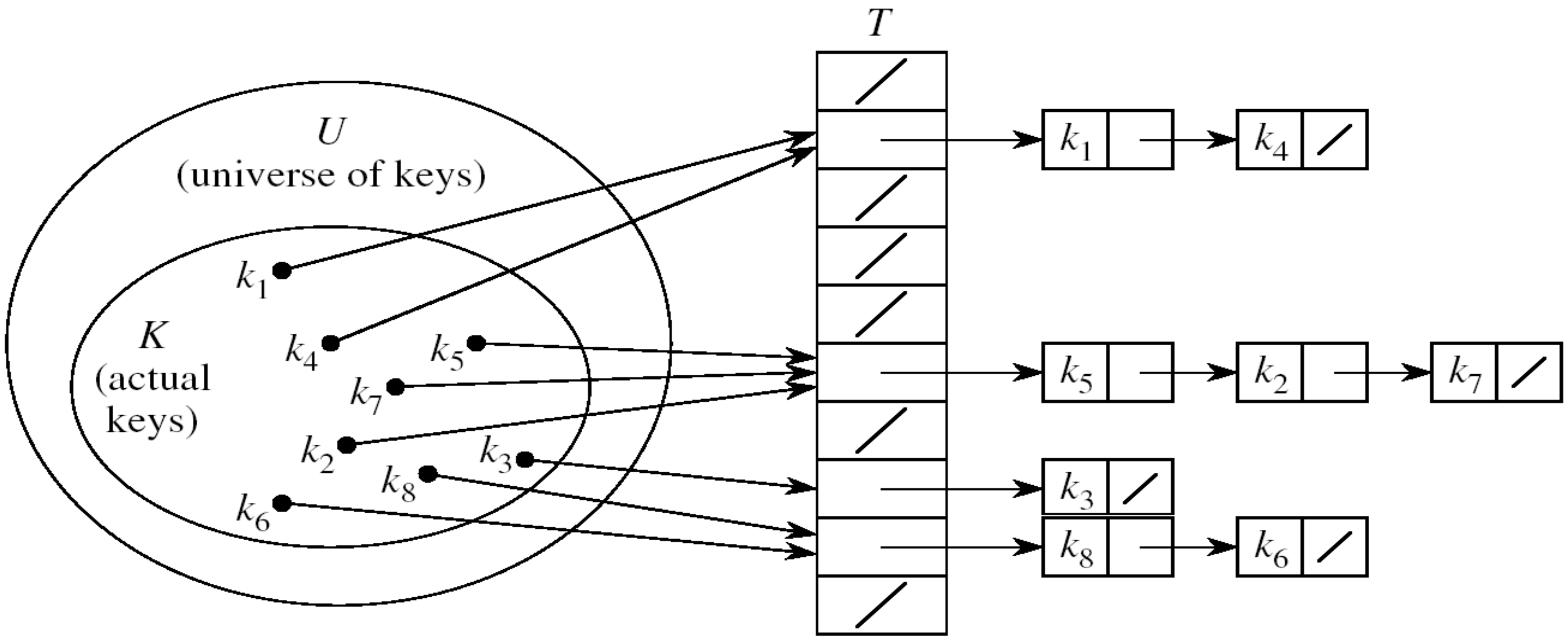
- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

Handling Collisions

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build "good" functions.

Handling Collisions using Chaining

- **Idea**:
  - Put all elements that hash to the same slot into a linked list

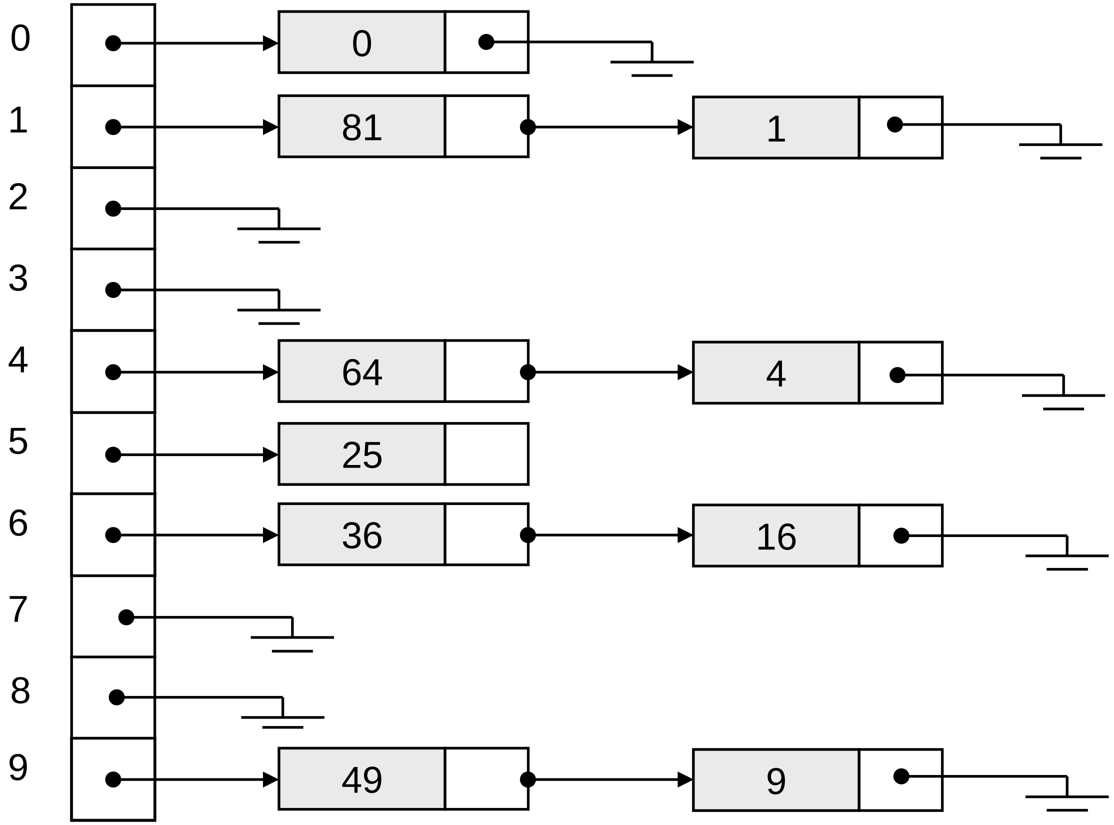


- Slot j contains a pointer to the head of the list of all elements that hash to j

Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.



Collision with Chaining - Discussion

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically, 1/5 or 1/10 of the total number of elements
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Insertion in Hash Tables

Alg.: CHAINED-HASH-INSERT(T, x)

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

- Worst-case running time is $O(1)$
- Assumes that the element being inserted is not already in the list
- It would take an additional search to check if it was already inserted

Deletion in Hash Tables

Alg.: CHAINED-HASH-DELETE(T, x)

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

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in Hash Tables

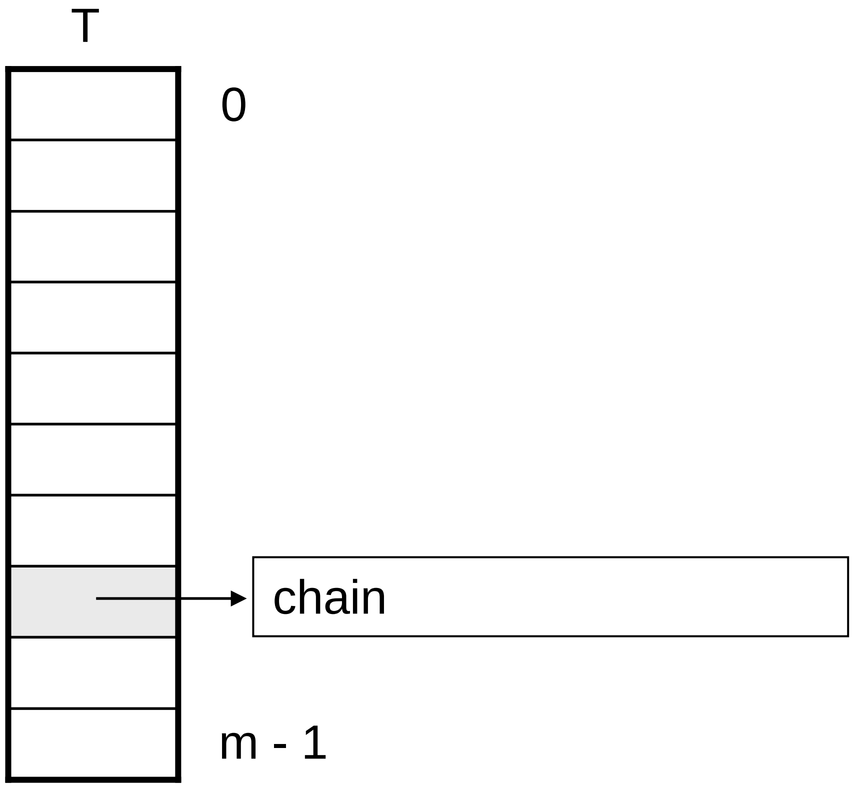
Alg.: CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list $T[h(k)]$

- Running time is proportional to the length of the list of elements in slot $h(k)$

Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All n keys hash to the same slot
  - Worst-case time to search is $\Theta(n)$, plus time to compute the hash function



Analysis of Hashing with Chaining: Average Case

- Average case
  - depends on how well the hash function distributes the n keys among the m slots
- **Simple uniform hashing** assumption:
  - Any given element is equally likely to hash into any of the m slots (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)
- Length of a list:

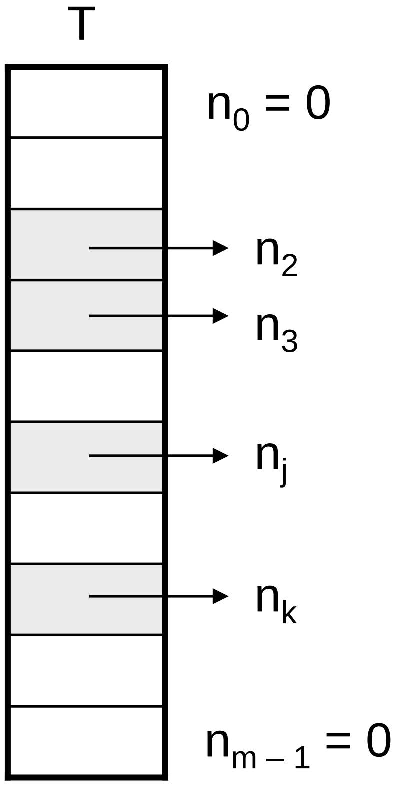
$$T[j] = n_j, \quad j = 0, 1, \ldots, m - 1$$

- Number of keys in the table:

$$n = n_0 + n_1 + \cdots + n_{m-1}$$

- Average value of n_j:

$$E[n_j] = \alpha = n/m$$

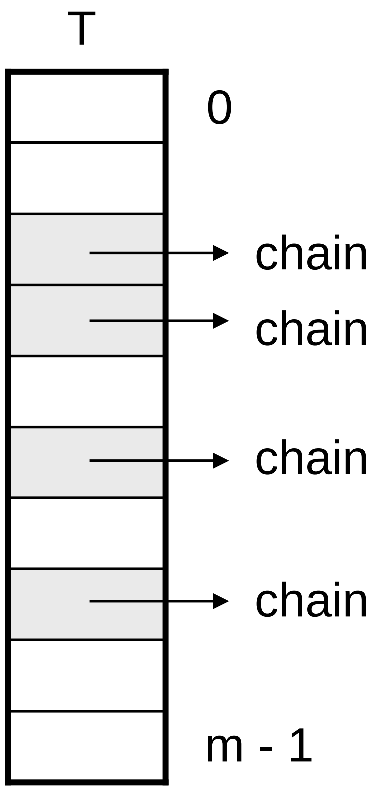


Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

  - n = # of elements stored in the table
  - m = # of slots in the table = # of linked lists
- α encodes the average number of elements stored in a chain
- α can be <, =, > 1



Case 1: Unsuccessful Search (i.e., item not stored in the table)

Theorem

An unsuccessful search in a hash table takes expected time under the assumption of simple uniform hashing : (i.e., probability of collision Pr(h(x)=h(y)), is 1/m) $\Theta(1+\alpha)$

Proof

- Searching unsuccessfully for any key k
  - need to search to the end of the list $T[h(k)]$
- Expected length of the list:
  - $E[n_{h(k)}] = \alpha = n/m$
- Expected number of elements examined in an unsuccessful search is α
- Total time required is:
  - O(1) (for computing the hash function) + $\alpha \rightarrow \Theta(1+\alpha)$

Case 2: Successful Search

- Successful search: $\Theta\ (1+\ \alpha/2) = \Theta\ (1+\ \alpha)$
  - On an average search half of a list of length α and O(1) time to compute h(k)

Analysis of Search in Hash Tables

- If m (# of slots) is proportional to n
- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$

$\Rightarrow$ Searching takes constant time on average

Summary

- The analysis shows us that the table size is not really important, but the load factor is.
- TableSize should be as *large* as the number of expected elements in the hash table.
  - To keep load factor around 1.
- TableSize should be *prime* for even distribution of keys to hash table cells.

Hash Functions

- A hash function transforms a key into a table address
- **What makes a good hash function?**

  (1) Easy to compute

  (2) Approximates a random function: for every input, every output is equally likely (simple uniform hashing)

- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., we don’t know in advance the probability distribution that keys are drawn from

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as `pt` and `pts` should hash to different slots
- **Derive a hash value that is independent from any patterns that may exist in the distribution of the keys**

Example :

Let m=2^3 (k=3)

$$\left.\begin{aligned} 00101 \bmod 2^3 &= 101 \\ 01101 \bmod 2^3 &= 101 \\ 10101 \bmod 2^3 &= 101 \\ 11101 \bmod 2^3 &= 101 \end{aligned}\right\}$$

Collision

$i.e.$

$m = 2^k$

$$1010111001110\underbrace{101010101}_{\textbf{K-bit}} \bmod 2^k = \underbrace{101010101}_{\textbf{LSB K-bit}}$$

The Division Method

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

$$h(k) = k \bmod m$$

- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

Example - The Division Method

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow h(k) = \{0, 1\}$, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow h(k) = \{0, 1, 2, 3\}$, least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

	m 97	m 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

The Multiplication Method

Idea:

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m \underbrace{(k\, A \bmod 1)}_{} \rfloor$$

fractional part of kA = kA - $\lfloor kA \rfloor$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, e.g., typically 2^p

Example – Multiplication Method

- The value of m is not critical now (e.g., $m = 2^p$)

assume $m = 2^3$

```
      .101101 (A)
       110101 (k)
--------------
1001010.0110011 (kA)
```

discard: 1001010

shift .0110011 by 3 bits to the left

011.0011

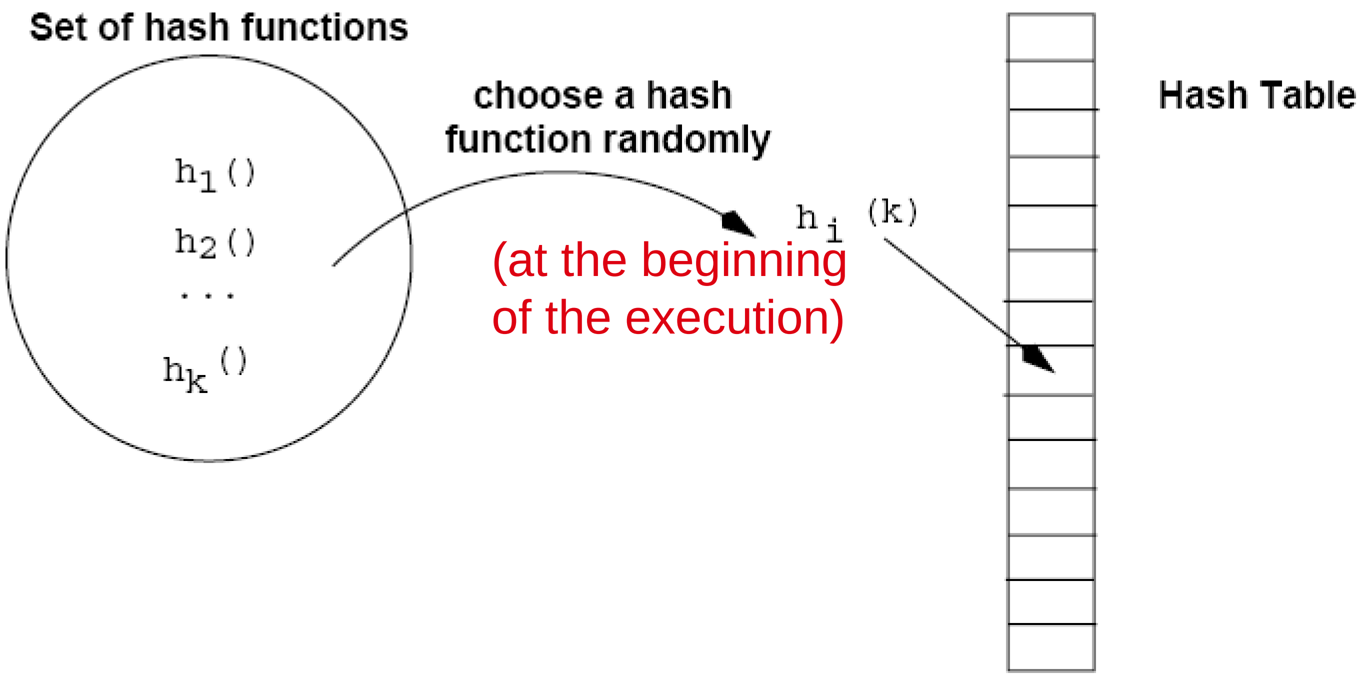
take integer part: 011

thus, h(110101)=011

Universal Hashing

- In practice, keys are not randomly distributed
- Any fixed hash function might yield $\Theta(n)$ time
- Goal: hash functions that produce random table indices irrespective of the keys
- Idea:
  - Select a hash function at random, from a designed class of functions at the beginning of the execution

Universal Hashing



Definition of Universal Hash Functions

$H=\{h(k): U \rightarrow (0,1,..,m-1)\}$

H is said to be universal if

$$\text{for } x \neq y, |(\mathbf{h}() \in \mathbf{H}: \mathbf{h}(\mathbf{x})=\mathbf{h}(\mathbf{y})|=|\mathbf{H}|/\mathbf{m}$$

(notation: $|H|$: number of elements in H - cardinality of H)

How is this property useful?

- What is the probability of collision in this case ?

It is equal to the probability of choosing a function $h \in U$ such that $x \neq y$ --> $h(x) = h(y)$ which is

$$\Pr(h(x)=h(y))= \frac{|H|/m}{|H|} = \frac{1}{m}$$

Universal Hashing – Main Result

With universal hashing the chance of collision between distinct keys k and l is no more than the $1/m$ chance of collision if locations $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m - 1\}$

Designing a Universal Class of Hash Functions

- Choose a prime number p large enough so that every possible key k is in the range $[0 \ldots p - 1]$

$$Z_p = \{0, 1, \ldots, p - 1\} \text{ and } Z_p^* = \{1, \ldots, p - 1\}$$

- Define the following hash function

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$$

$$\forall\, a \in Z_p^* \text{ and } b \in Z_p$$

- The family of all such hash functions is

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ and } b \in Z_p\}$$

The class $\mathcal{H}_{p,m}$ of hash functions is universal

- a , b: chosen randomly at the beginning of execution

Example: Universal Hash Functions

E.g.: $p = 17,\ m = 6$

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

$$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$$

$$= (28 \bmod 17) \bmod 6$$

$$= 11 \bmod 6$$

$$= 5$$

Advantages of Universal Hashing

- Universal hashing provides good results on average, independently of the keys to be stored
- Guarantees that no input will always elicit the worst-case behavior
- Poor performance occurs only when the random choice returns an inefficient hash function – this has small probability

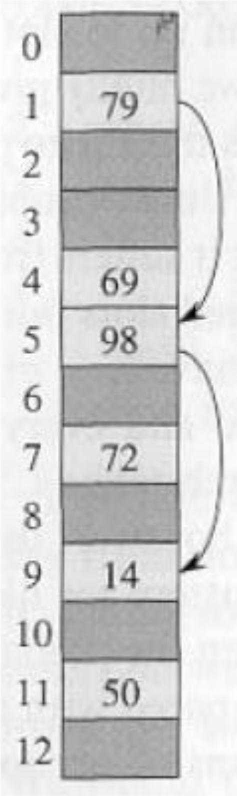
Collision Resolution with Open Addressing

- Separate chaining has the disadvantage of using linked lists.
  - Requires the implementation of a second data structure.
- In an open addressing hashing system, all the data go inside the table.
  - Thus, a bigger table is needed.
    - Generally, the load factor should be below 0.5.
  - If a collision occurs, alternative cells are tried until an empty cell is found.

Open Addressing

- If we have enough contiguous memory to store all the keys ($m > N$) $\Rightarrow$ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - Insertion: if a slot is full, try another one, until you find an empty one
  - Search: follow the same sequence of probes
  - Deletion: more difficult ... (we'll see why)
- Search time depends on the length of the probe sequence!

e.g., insert 14



Generalize hash function notation:

- A hash function contains two arguments now:

  (i) Key value, and (ii) Probe number

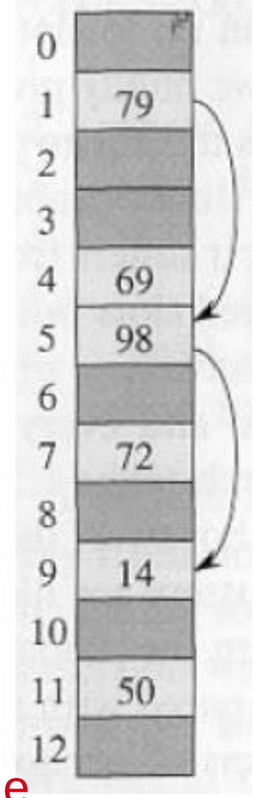
$$h(k,p), \quad p=0,1,\dots,m-1$$

- Probe sequences

  <h(k,0), h(k,1), ..., h(k,m-1)>

  - Must be a permutation of <0,1,...,m-1>
  - There are m! possible permutations
  - Good hash functions should be able to produce all m! probe sequences

insert 14



Example

<1, 5, 9>

Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

- Note: None of these methods can generate more than m^2 different probing sequences!

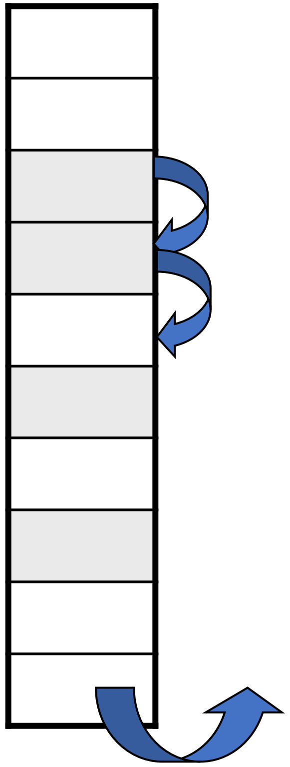
Linear probing: Inserting a key

- Idea: when there is a collision, check the next available position in the table (i.e., probing)

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

$$i=0,1,2,\ldots$$

- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$
- Third slot probed: $h_1(k)+2$, and so on



wrap around

probe sequence: < h1(k), h1(k)+1 , h1(k)+2 , ...>

- Can generate m probe sequences maximum, why?

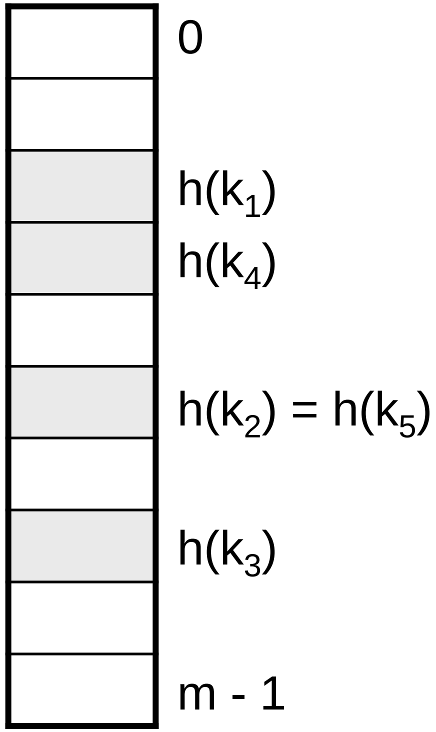
Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear probing: Searching for a key

- Three cases:
  - (1) Position in the table is occupied with an element of equal key
  - (2) Position in the table is empty
  - (3) Position in the table occupied with a different element
- Case 3: probe the next higher index until the element is found or an empty position is found
- The process wraps around to the beginning of the table



Linear probing: Deleting a key

- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys

Deletion: Linear Probing

- In linear probing deletion is difficult because deletion of one element, create trouble to other element

 e.g. delete **60**

Find element 19 ?..

0	**10**
1	**60**
2	**19**
3	
4	**44**
5	15
6	**75**
7	**35**
8	
9	**99**

Clustering Problem

- As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, even if the table is relatively empty, blocks of occupied cells start forming.
- This effect is known as *primary clustering*.
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Analysis of insertion

- The average number of cells that are examined in an insertion using linear probing is roughly

$$(1 + 1/(1 - \alpha)^2) / 2$$

  - Proof is beyond the scope of text book.
- For a half full table, we obtain 2.5 as the average number of cells examined during an insertion.
- Primary clustering is a problem at high load factors. For half empty tables, the effect is not disastrous.

Analysis of Find

- An unsuccessful search costs the same as insertion.
- The cost of a successful search of X is equal to the cost of inserting X at the time X was inserted.
- For $\alpha = 0.5$, the average cost of insertion is 2.5. The average cost of finding the newly inserted item will be 2.5 no matter how many insertions follow.
- Thus, the average cost of a successful search is an average of the insertion costs over all smaller load factors.

Average cost of `find`

- The average number of cells that are examined in an unsuccessful search using linear probing is roughly
  $(1 + 1/(1 - \alpha)^2) / 2$.
- The average number of cells that are examined in a successful search is approximately
  $(1 + 1/(1 - \alpha)) / 2$.
  - Derived from:

$$\frac{1}{\lambda}\int_{x=0}^{\lambda}\frac{1}{2}\left(1+\frac{1}{(1-x)^2}\right)dx$$

Summary

- Hash tables can be used to implement the insert and find operations in constant average time.
  - it depends on the load factor not on the number of items in the table.
- It is important to have a prime TableSize and a correct choice of load factor and hash function.
- For separate chaining, the load factor should be close to 1.
- For open addressing load factor should not exceed 0.5 unless this is completely unavoidable.
  - Rehashing can be implemented to grow (or shrink) the table.

Quadratic probing

i=0,1,2,...

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \text{ where } h' : U --> (0, 1, \ldots, m-1)$$

- Clustering problem is less serious but still an issue (*secondary clustering*)

- How many probe sequences quadratic probing generate ? m

(the initial probe position determines the probe sequence)

Double Hashing

(1) Use one hash function to determine the first slot

(2) Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,\ldots$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate m^2 probe sequences maximum

Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

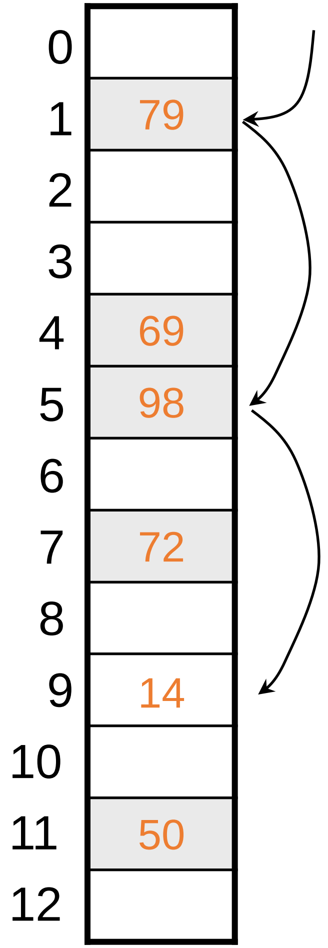
$h_1(14,0) = 14 \bmod 13 = 1$

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



Analysis of Open Addressing

- Ignore the problem of clustering and assume that all probe sequences are equally likely

Unsuccessful retrieval:

Prob(*probe hits an occupied cell*)= a (load factor)

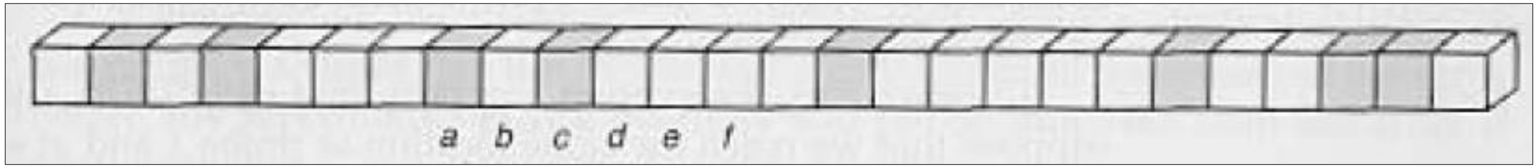
Prob(*probe hits an empty cell*)= $1-a$

probability that a probe terminates in 2 steps: $a(1-a)$

probability that a probe terminates in k steps: $a^{k-1}(1-a)$

What is the average number of steps in a probe ?

$$E(\#steps) = \sum_{k=1}^{m} ka^{k-1}(1-a) \le \sum_{k=0}^{\infty} ka^{k-1}(1-a) = (1-a)\frac{1}{(1-a)^2} = \frac{1}{1-a}$$



Analysis of Open Addressing (cont'd)

Successful retrieval:

$$E(\#steps) = \frac{1}{a}\, ln(\frac{1}{1-a})$$

Example (similar to **Exercise 11.4-4, page 244**)

Unsuccessful retrieval:

a=0.5 E(#steps) = 2

a=0.9 E(#steps) = 10

Successful retrieval:

a=0.5 E(#steps) = 3.387

a=0.9 E(#steps) = 3.670