Chapter11. Hash Tables

- Hash table
- Issue with hashing
- Collision Resolution Techniques
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Review

- Array Lists
  - O(1) access
  - O(N) insertion (average case), better at end
  - O(N) deletion (average case)
- Linked Lists
  - O(N) access
  - O(N) insertion (average case), better at front and back
  - O(N) deletion (average case), better at front and back
- Binary Search Trees
  - O(log N) access if balanced
  - O(log N) insertion if balanced
  - O(log N) deletion if balanced

Review

- What is hashing? Why is it useful to us?
  – There are lots of applications that need to support only the operations INSERT, SEARCH, and DELETE. These are known as "dictionary" operations.
- Applications:
  – data base search
    - books in a library
    - patient records, GIS data etc.
  – web page caching (web search)
  – combinatorial search (game tree)

Review : Performance goal for dictionary operations:

- *O(n) is too inefficient*.

Goal

- O(log n) on average
- O(log n) in the worst-case
- O(1) on average

Data structure that achieve these goals:

O(log n) on average ⇨ binary search tree(BST)

O(log n) in the worst-case ⇨ balanced BST(AVL tree)

O(1) on average ⇨ hashing. (but worst-case is O(n))

Hash

hash: transitive verb[1]

1. (a) to chop (as meat and potatoes) into small pieces
   (b) confuse, muddle
2. ...

Hash brown

Review

- Hashing
  - important and widely useful technique for implementing dictionaries
  - Technique supporting insertion, deletion, and search in <u>*average-case constant time: O(1)*</u>
  - Operations requiring elements to be sorted (e.g. find minimum) are not efficiently supported

Dictionary & Hash Tables

- **Dictionary:**
  - Dynamic-set data structure for storing items indexed using *keys*.
  - Supports operations Insert, Search, and Delete.
  - Applications:
    - Symbol table of a compiler.
    - Memory-management tables in operating systems.
    - Large-scale distributed systems.
- **Hash Tables:**
  - Effective way of implementing dictionaries.
  - Generalization of ordinary arrays.

- Direct-address Tables are ordinary arrays.
- Facilitate direct addressing.
  - Element whose key is k is obtained by indexing into the k^{th} position of the array.
- Applicable when we can afford to allocate an array with one position for every possible key.
  - i.e. when the universe of keys U is small.
- Dictionary operations can be implemented to take $O(1)$ time.

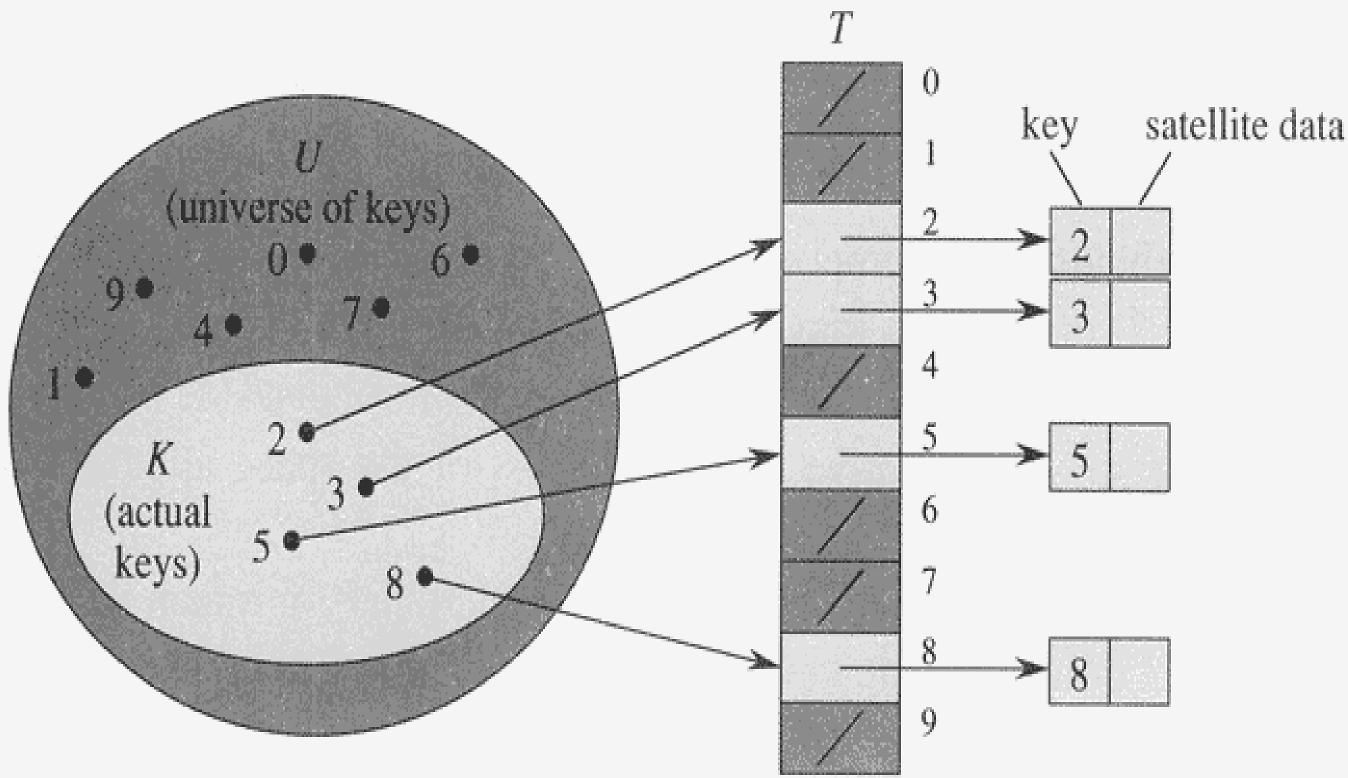
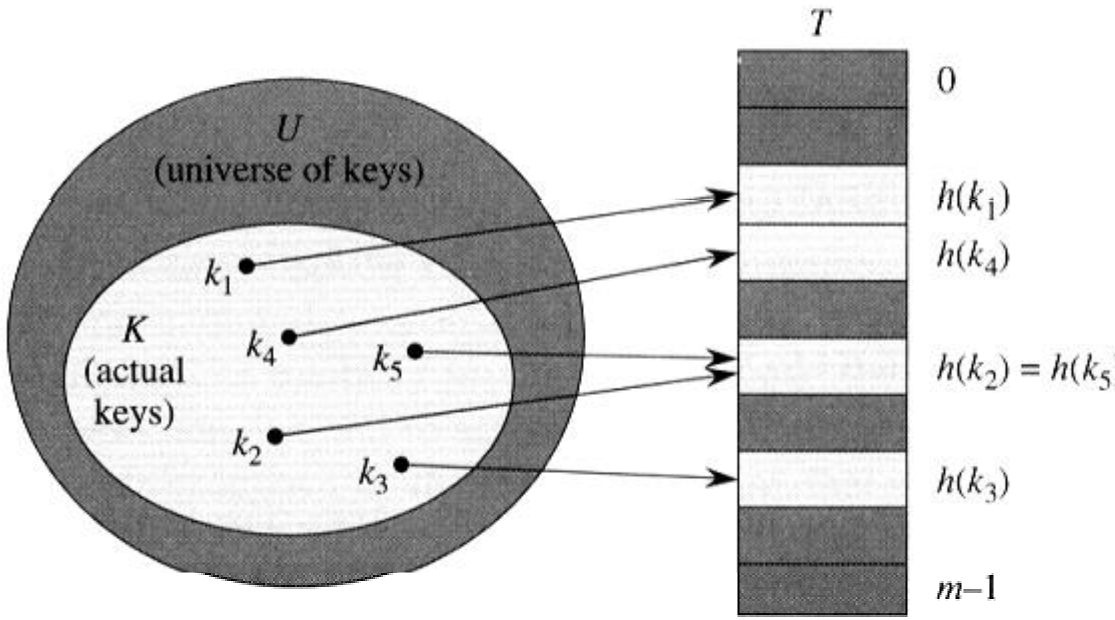


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Hash Table

- The difficulty with direct address is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible.
- Furthermore, the set K of keys actually stored may be so small relative to U. Specifically, the storage requirements can be reduced to $O(|K|)$, even though searching for an element in the hash table still requires only $O(1)$ time.



U
(universe of keys)
K
(actual keys)
k1
k4
k5
k2
k3
T
0
h(k1)
h(k4)
h(k2) = h(k5)
h(k3)
m–1

Hash Table

- **Notation:**
  - U : Universe of all possible keys.
  - K : Set of keys actually stored in the dictionary.
  - $|K| = n$.
- When U is very large,
  - Arrays are not practical.
  - $|K| << |U|$.
- Use a table of size proportional to $|K|$: *The hash tables.*
  - However, we lose the direct-addressing ability.
  - Define functions that map keys to slots of the hash table.

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

  $$h : U \rightarrow \{0,1,\ldots, m–1\}$$

- With arrays, key k maps to slot $A[k]$.
- With hash tables, key k maps or "hashes" to slot $T[h[k]]$.
- $h[k]$ is the *hash value* of key k.

Hash function example

- elements = Integers
- *h(i) = i % 10 (= i mod 10)*
- add 41, 34, 7, and 18
- constant-time lookup:
  - just look at *i % 10* again later
- Hash tables have no ordering information!
  - Expensive to do following:
    - getMin, getMax, removeMin, removeMax,
    - the various ordered traversals
    - printing items in sorted order

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Issue with Hashing

- Multiple keys can hash to the same slot
  - <u>Collisions</u>(two keys hash to same slot) are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations can be made to have an expected complexity of $\Theta(1)$.

Collision

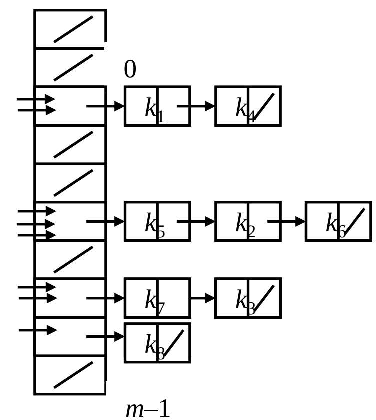
- Two or more keys hash to the same slot.
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely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

Collision Resolution Techniques

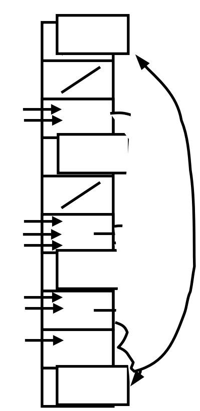
- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Collision Resolution Techniques

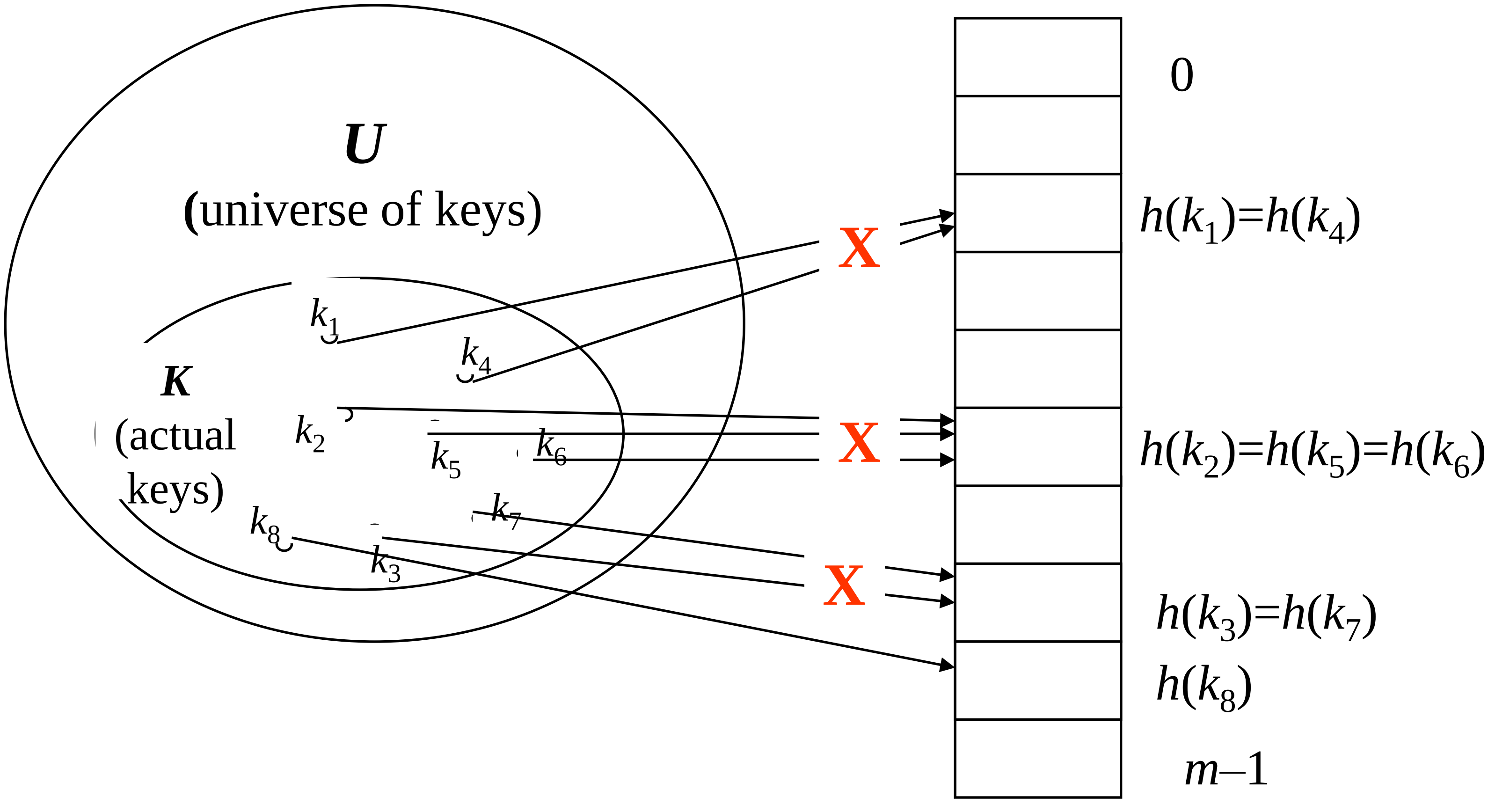
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



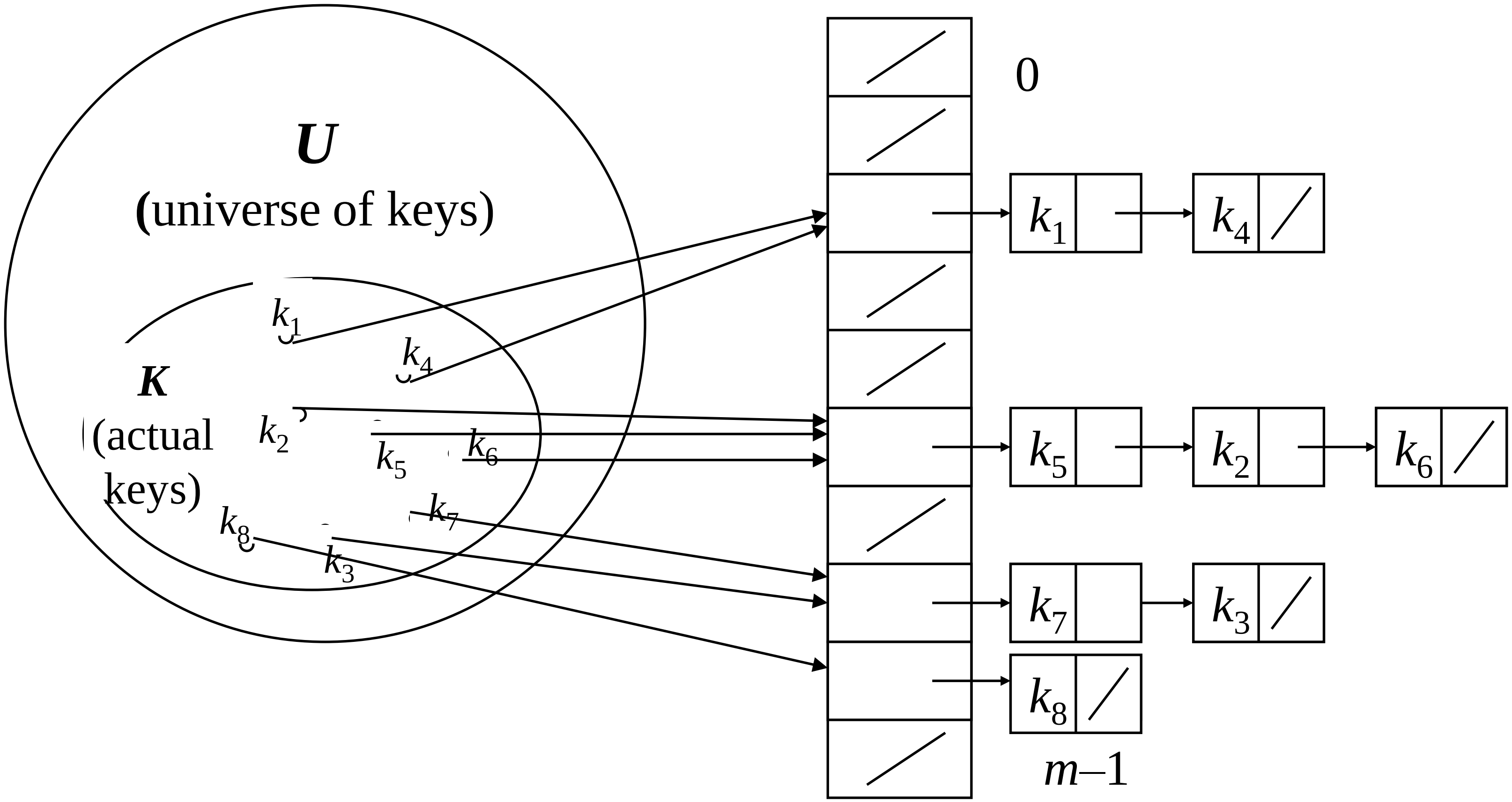
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



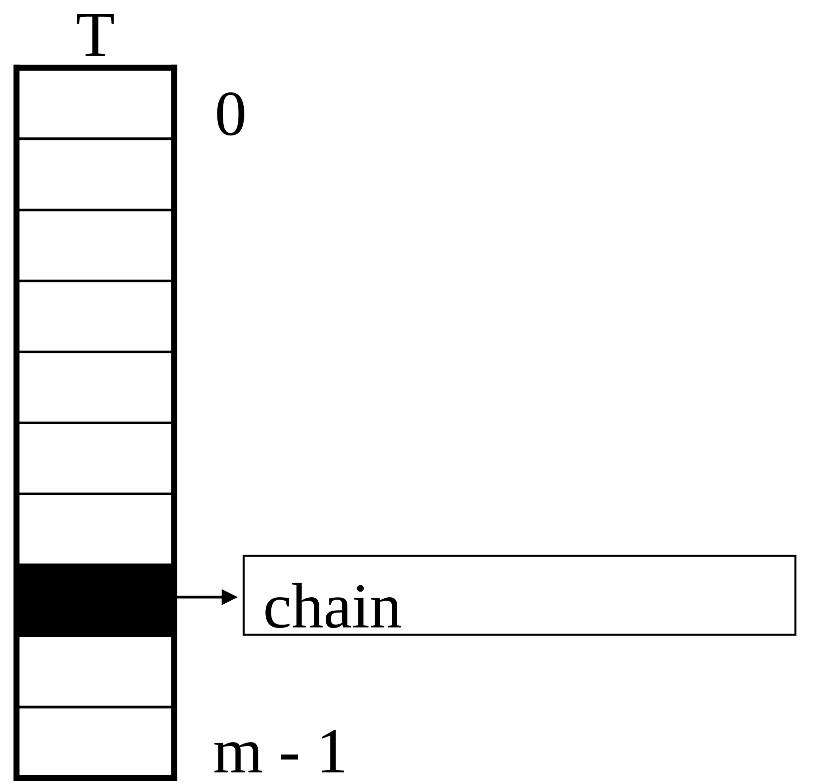
Hashing with Chaining

Dictionary Operation

- 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.

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:

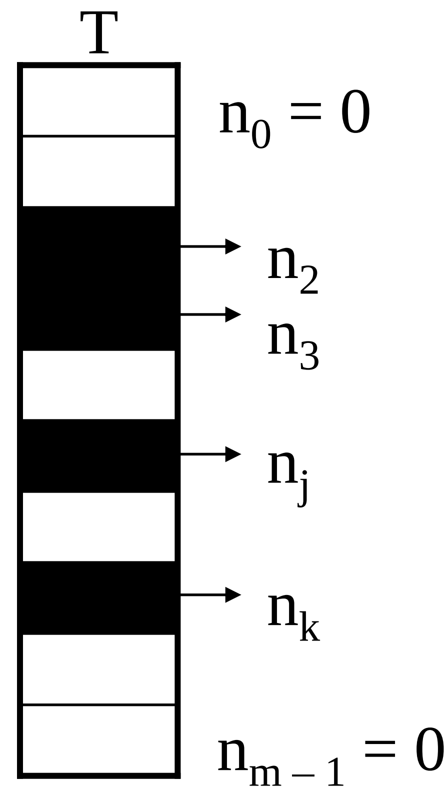
$$T[j] = n_j, \quad j = 0, 1, \ldots, m-1$$

- Number of keys in the table:

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

- Average value of n_j:

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

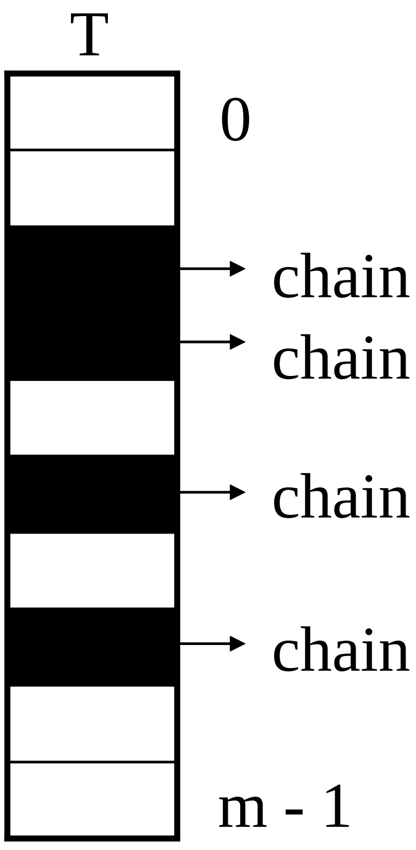


Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

  – n = # of elements stored in the table
  – m = # of slots in the table
- 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 $\Theta(1+\alpha)$ under the assumption of simple uniform hashing

(i.e., probability of collision $Pr(h(x)=h(y))$, is $1/m$)

Proof

- Searching unsuccessfully for any key k : $T[h(k)]$
- 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

Theorem: A successful search takes expected time $\Theta(1+\alpha)$.

Proof :

- Let x_i be the i^{th} element inserted into the table, and let $k_i = key[x_i]$.
- Define indicator random variables $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$, for all i, j.
- Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m$

  $\Rightarrow \mathrm{E}[X_{ij}] = 1/m$.
- Expected number of elements examined in a successful search is:

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

Case 2: Successful Search

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

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$ (linearity of expectation)

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

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

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{2m}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Expected total time for a successful search
= Time to compute hash function + Time to search
= $\boldsymbol{O(1+1+\alpha/2-\alpha/2n)=O(1+\alpha)}$.

Analysis of Search in Hash Tables

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

$\Rightarrow$ *Searching takes constant time on average.*

- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time when lists are do ubly linked.
- Hence, ***all dictionary operations take O(1) time on average with hash tables with chaining.***

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

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

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
- Good choice for *m*:
  - Primes, not too close to power of 2 (or 10) are good.

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

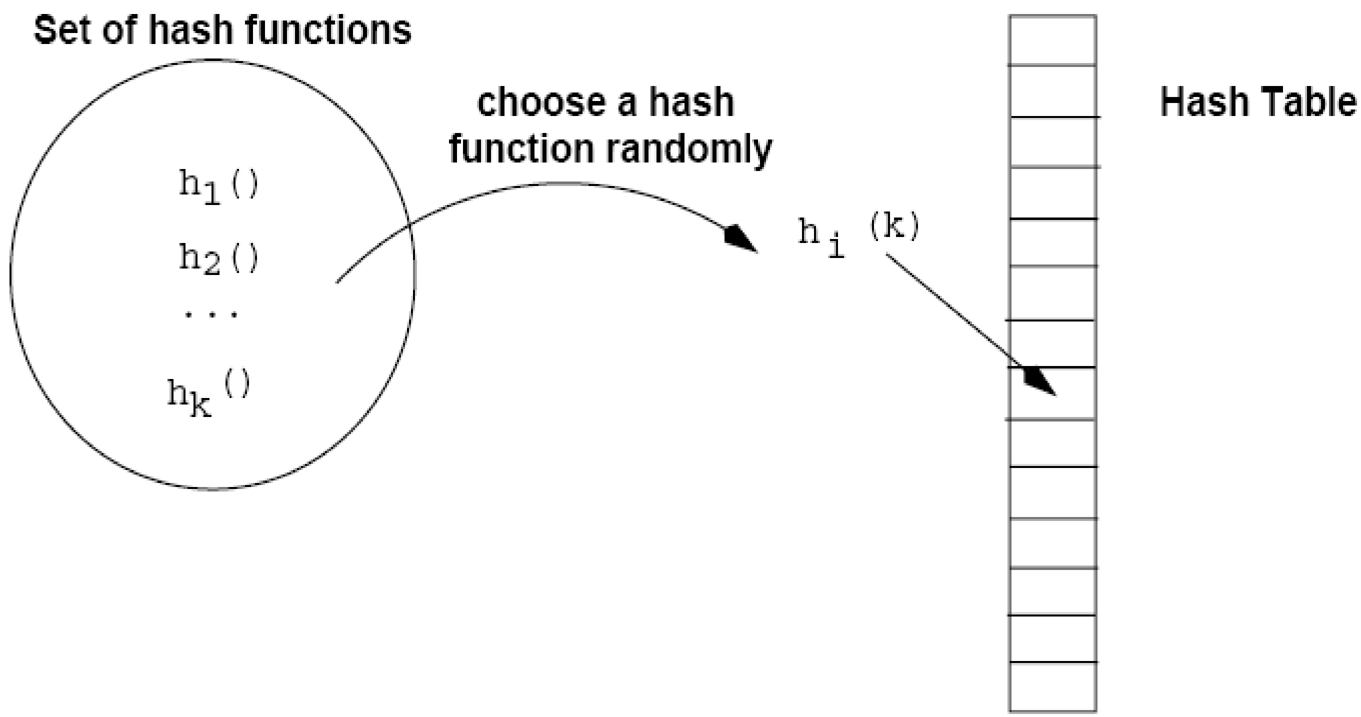
take integer part: 011

thus, h(110101)=011

Universal Hashing

- A malicious adversary who has learned the hash function chooses keys that all map to the same slot, giving worst-case behavior.
- Defeat the adversary using **Universal Hashing**
  - Use a different random hash function each time.
  - Ensure that the random hash function is independent of the keys that are actually going to be stored.
  - Ensure that the random hash function is "good" by carefully designing a class of functions to choose from.
    - Design a **universal** class of functions.

Universal Hashing



Definition of Universal Hash Functions

H=(h(k): U --h(k)--> (0,1,...,m-1))

H is said to be universal if

for $x \neq y$, **|(h() $\in$ H: h(x)=h(y)|=|H|/m**

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

- The chance of a collision between two keys is the $1/m$ chance of choosing two slots randomly & independently.
- Universal hash functions give good hashing behavior

Universal Hashing

- 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

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

- 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, …, m – 1}

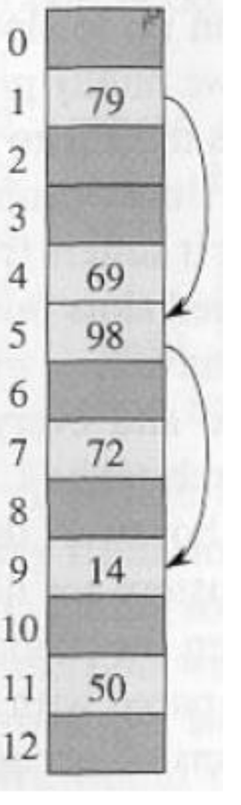
Advantages of Universal Hashing

- 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)

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 …
- 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

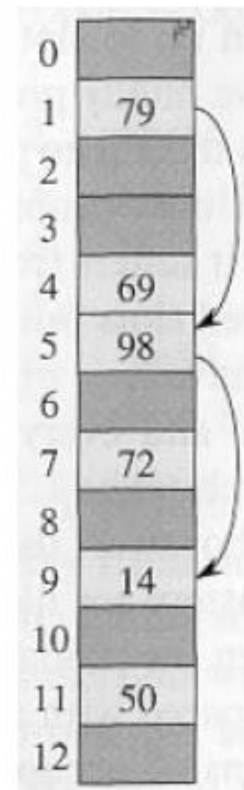
  $$h(k,p), \quad p=0,1,...,m-1$$

- Probe sequences

  $$\langle h(k,0), h(k,1), ..., h(k,m-1) \rangle$$

  - 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



Probe sequence: <1, 5, 9>

Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

Linear probing: Inserting a key

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

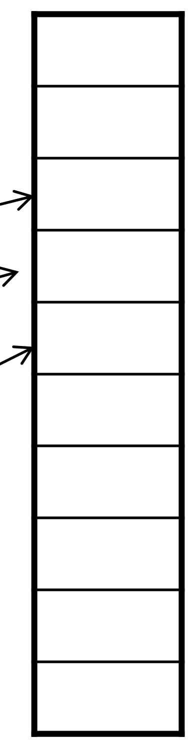
$$h(k,i) = (h_1(k) + i) \bmod m$$

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

First slot probed: $h_1(k)$

Second slot probed: $h_1(k) + 1$

Third slot probed: $h_1(k)+2$, and so on



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

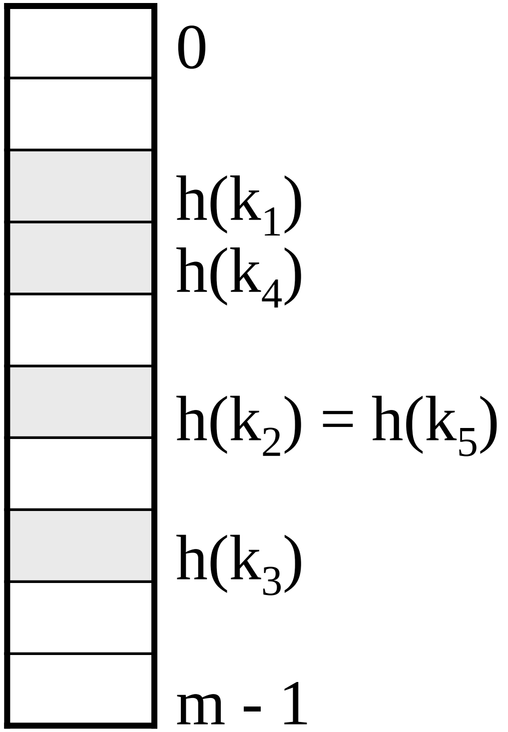
Linear probing: ***Searching*** for a key

- Three cases:

  (1) Position in table is occupied with an element of equal key

  (2) Position in table is empty

  (3) Position in table occupied with a different element
- Case 2: 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

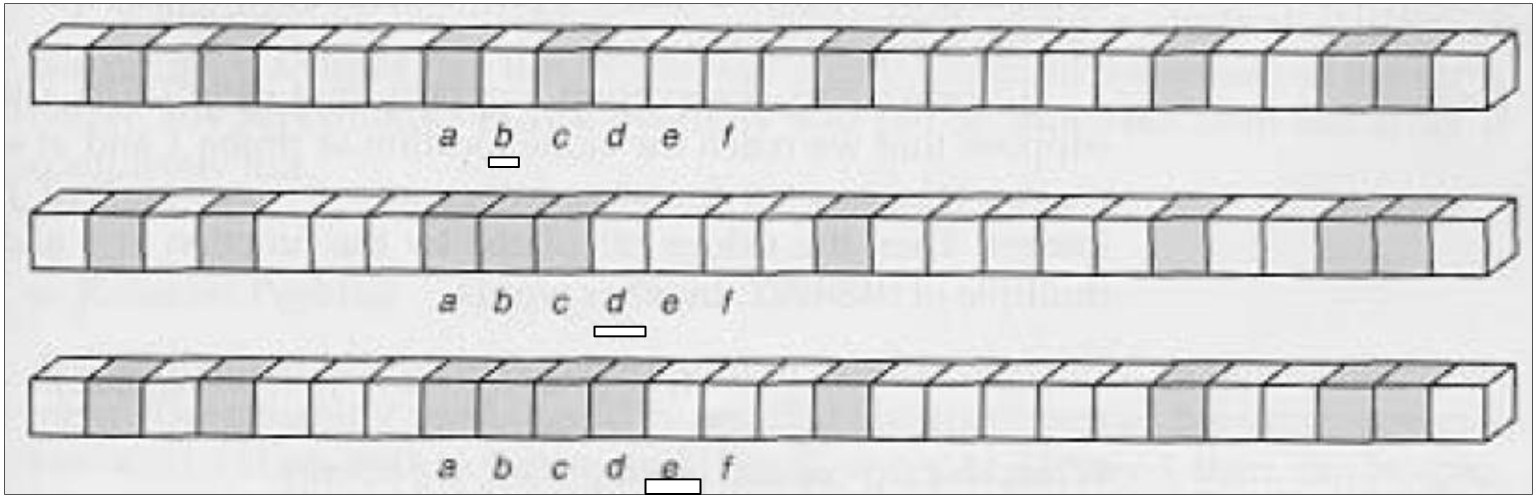
- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys

Primary Clustering Problem

- Some slots become more likely than others
- Long chunks of occupied slots are created

⇒ average insert & search time increases!!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

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

  key Probe number Auxiliary hash function

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number i.
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

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

$$h(k,i) = (h_1(k) + i\ h_2(k)\ )\ \mathrm{mod}\ m, \quad i=0,1,...$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k)$ mod m, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element

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:

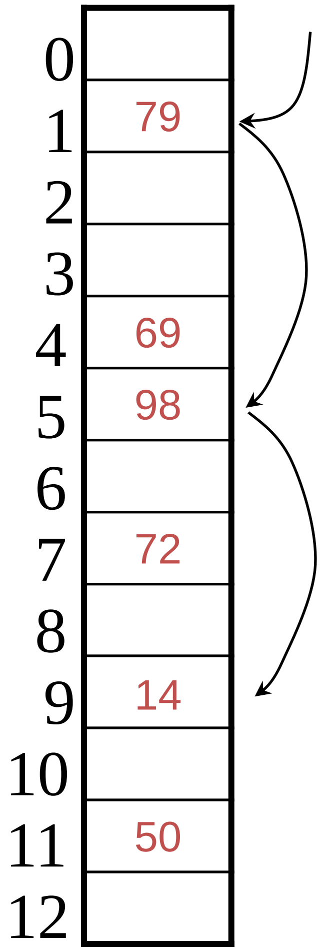
$h_1(14,0) = 14 \bmod 13 = 1$

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

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

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

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



Analysis of Open Addressing

- Analysis is in terms of load factor α.
- **Assumptions:**
  - Assume that the table never completely fills, so $n < m$ and $\alpha < 1$.
  - Assume uniform hashing.
  - <u>No</u> deletion.
  - All probe sequences are equally likely

Analysis of Open Addressing

- **<u>Unsuccessful retrieval</u>**:

  Prob(probe hits an occupied cell) = α

  Prob(probe hits an empty cell) = 1- α

  Probability that a probe terminates in 2 steps : $\alpha(1-\alpha)$

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

  What is the average number of steps in a probe?

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

Analysis of Open Addressing

- **successful retrieval**:
  The expected number of probes in a successful search in an open-address hash table is at most $(1/\alpha) \log (1/(1-\alpha))$.

Unsuccessful retrieval:

$\alpha = 0.5$ E(#steps) = 2

$\alpha = 0.9$ E(#steps) = 10

Successful retrieval:

$\alpha = 0.5$ E(#steps) = 3.387

$\alpha = 0.9$ E(#steps) = 3.670