Introduction to Algorithms

Search Structures and Hashing

Dictionary/Table

Student ID	First Name	Last Name	GPA
0	Joe	Johnson	3.5
1	John	Jones	2.9
2	Mike	Smith	4.0
3	Jerry	Kennedy	3.4
4	John	Lincoln	2.3
5	Fred	Flinstone	3.5
6	Wilma	Flinstone	3.2

Operation supported: search

Given a student ID find the record (entry)

Data Format

Keys	***Entry***

What if the key is the ID number

- Well, we can still sort by the ids and apply binary search.
- If we have n students, we need O(n) space
- And O(log n) search time

What if new students come and current students leave

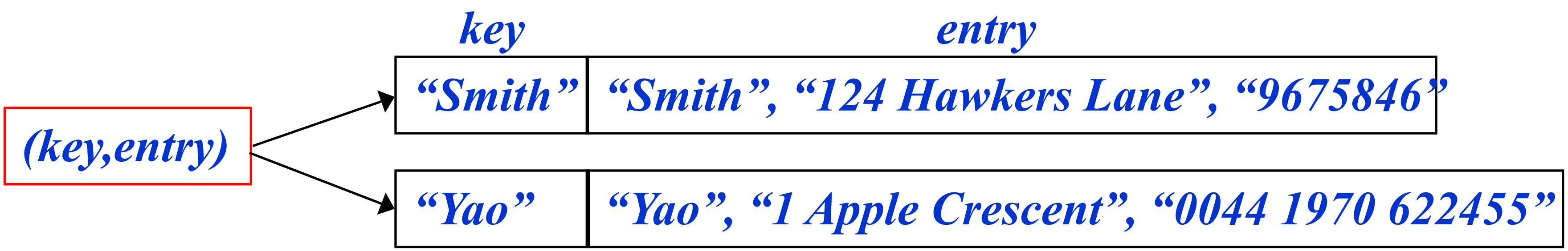
- Dynamic dictionary
- Operations to support
  - **Insert:** add a new (key, entry) pair
  - **Delete:** remove a (key, entry) pair from the dictionary
  - **Search:** Given a key, find if it is in the dictionary, and if it is, return the data entry associated with the key

How should we implement a dynamic dictionary?

- How often are entries inserted and removed?
- How many of the possible key values are likely to be used?
- What is the likely pattern of searching for keys?

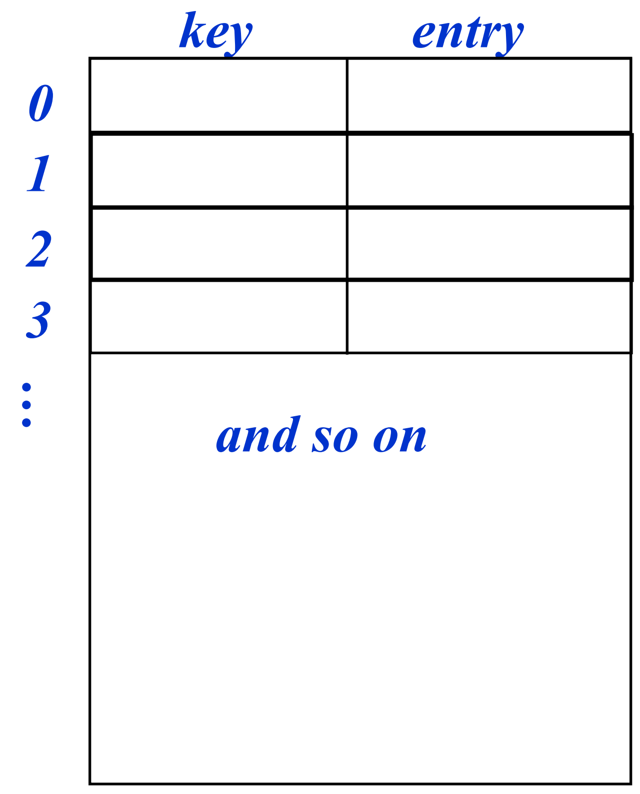
(Key,Entry) pair

- For searching purposes, it is best to store the key and the entry separately (even though the key's value may be inside the entry)



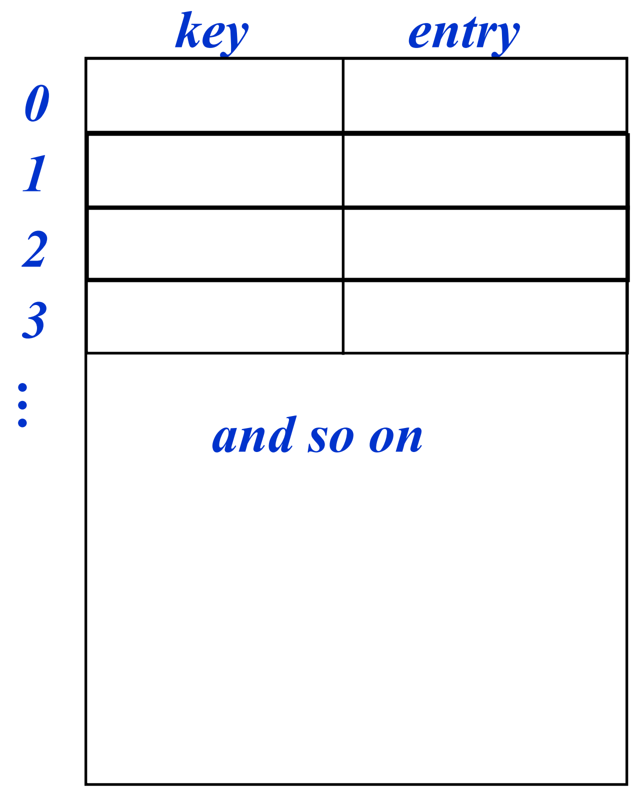
Implementation 1: unsorted sequential array

- An array in which (key,entry)-pair are stored consecutively in *any* order
- **insert**: add to the back of array; O(1)
- **search**: search through the keys one at a time, potentially all of the keys; O(n)
- **remove**: find + replace removed node with last node; O(n)



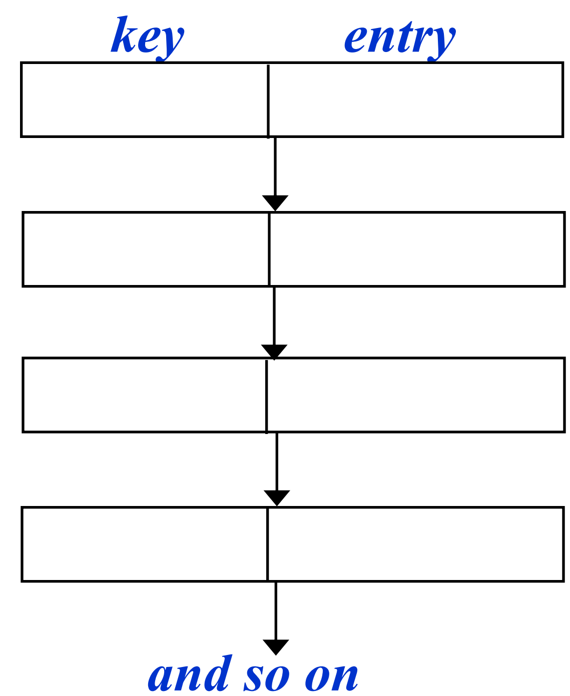
Implementation 2: sorted sequential array

- An array in which (key,entry) pair are stored consecutively, *sorted* by key
- **insert**: add in sorted order; O(n)
- **find**: binary search; O(log n)
- **remove**: find, remove node and shuffle down; O(n)



Implementation 3: linked list (unsorted or sorted)

- (key,entry) pairs are again stored consecutively
- **insert**: add to front; O(1)
  or O(n) for a sorted list
- **find**: search through potentially all the keys, one at a time; O(n)
  still O(n) for a sorted list
- **remove**: find, remove using pointer alterations; O(n)



Direct Addressing

- Suppose:
  - The range of keys is 0..m-1 (Universe)
  - 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

Direct-address Table

- **Direct addressing is a simple technique that works well when the universe of keys is small.**

Assuming each key corresponds to a unique slot.

Direct-Address-Search(*T*,*k*)

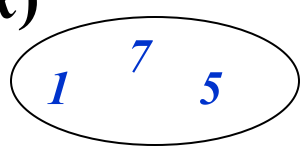
return $T[k]$

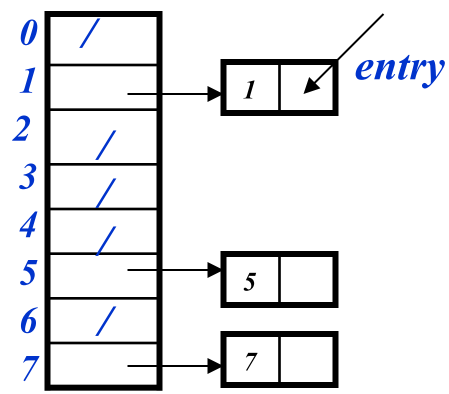
Direct-Address-Insert(*T*,*x*)

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

Direct-Address-Delete(*T*,*x*)

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





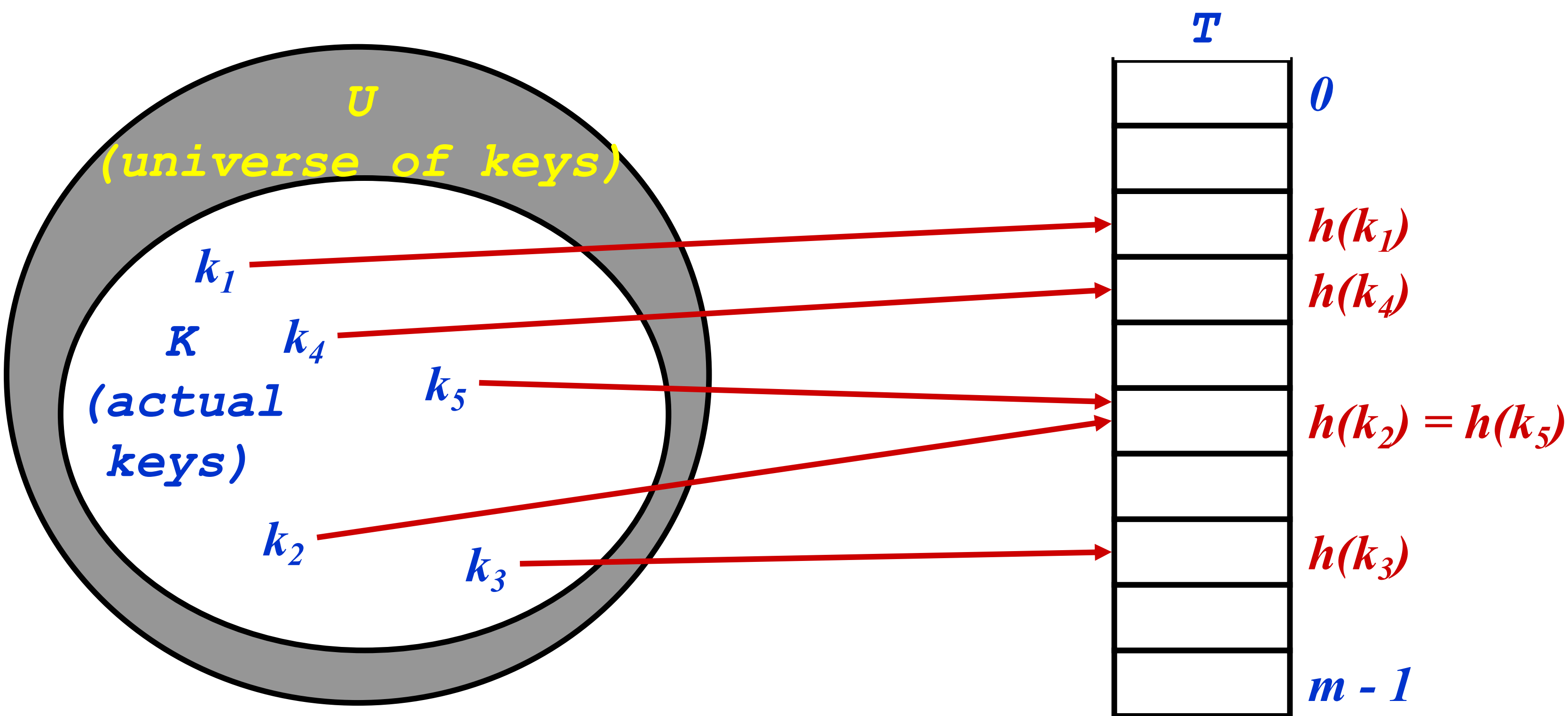
O(1) time for all operations

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***

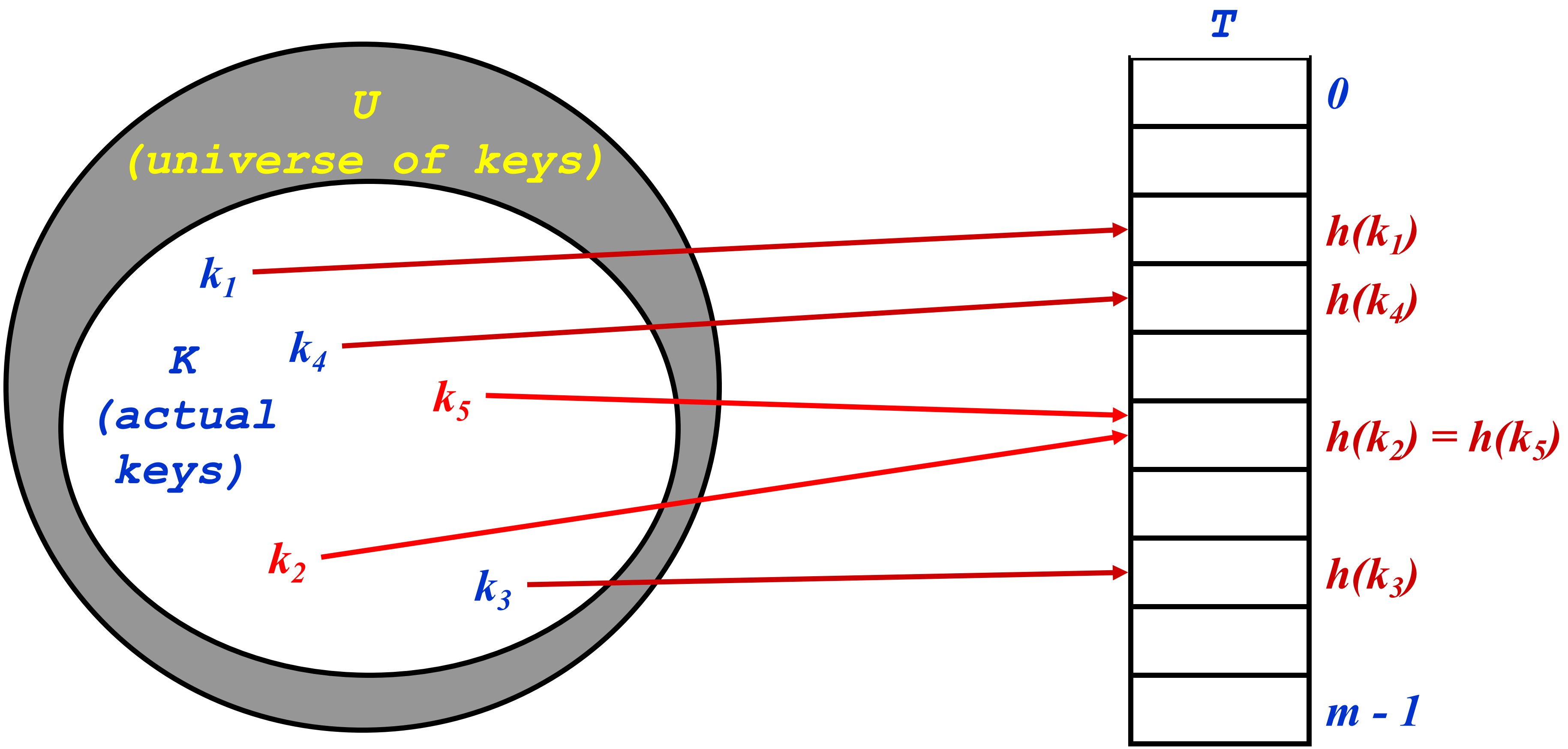
Hash function

- A hash function determines the slot of the hash table where the key is placed.
- Previous example the hash function is the identity function
- We say that a record with key k hashes into slot $h(k)$



Next Problem

- *collision*

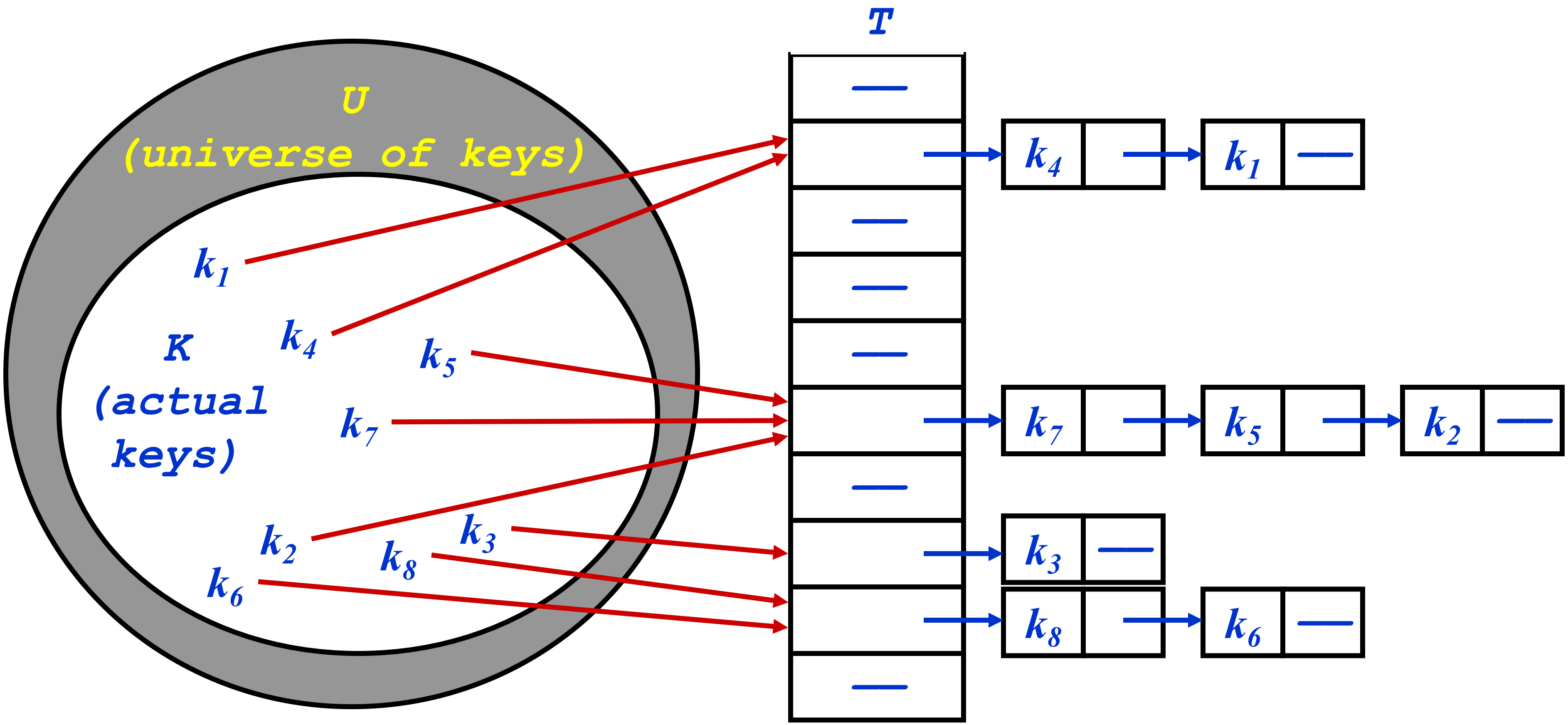


Resolving Collisions

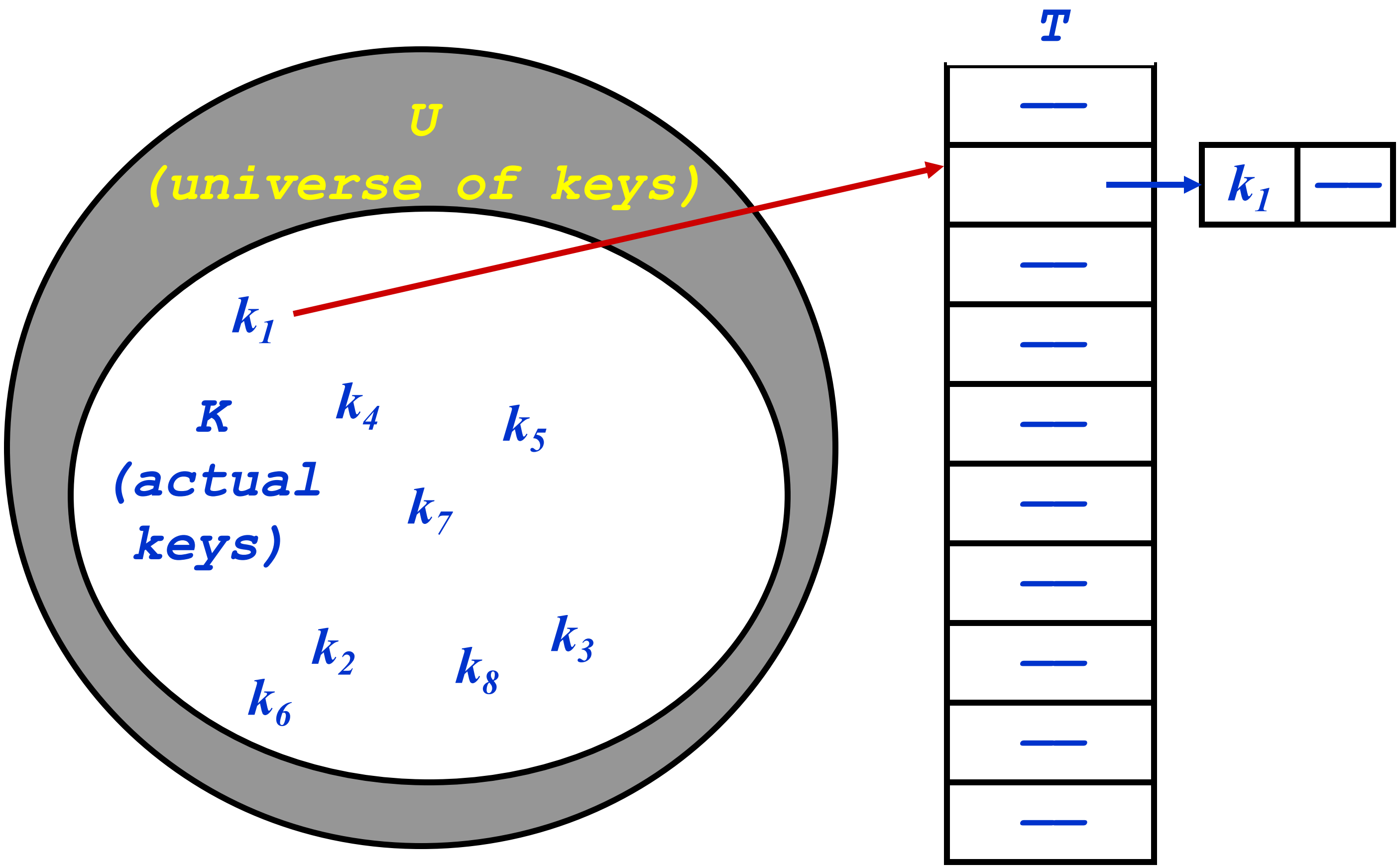
- **How can we solve the problem of collisions?**
- Solution 1: *chaining*
- Solution 2: *open addressing*

Chaining

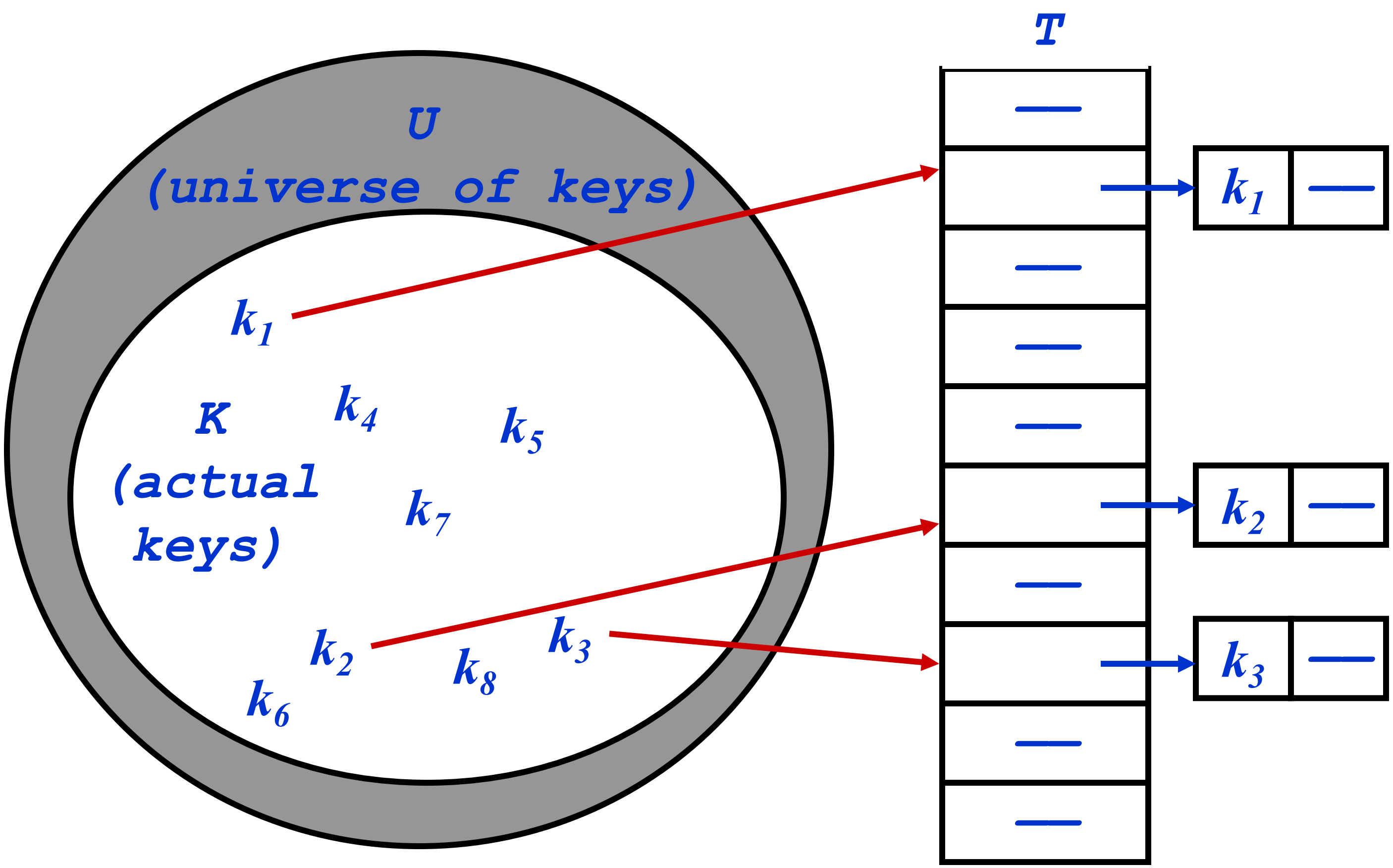
- Chaining puts elements that hash to the same slot in a linked list:



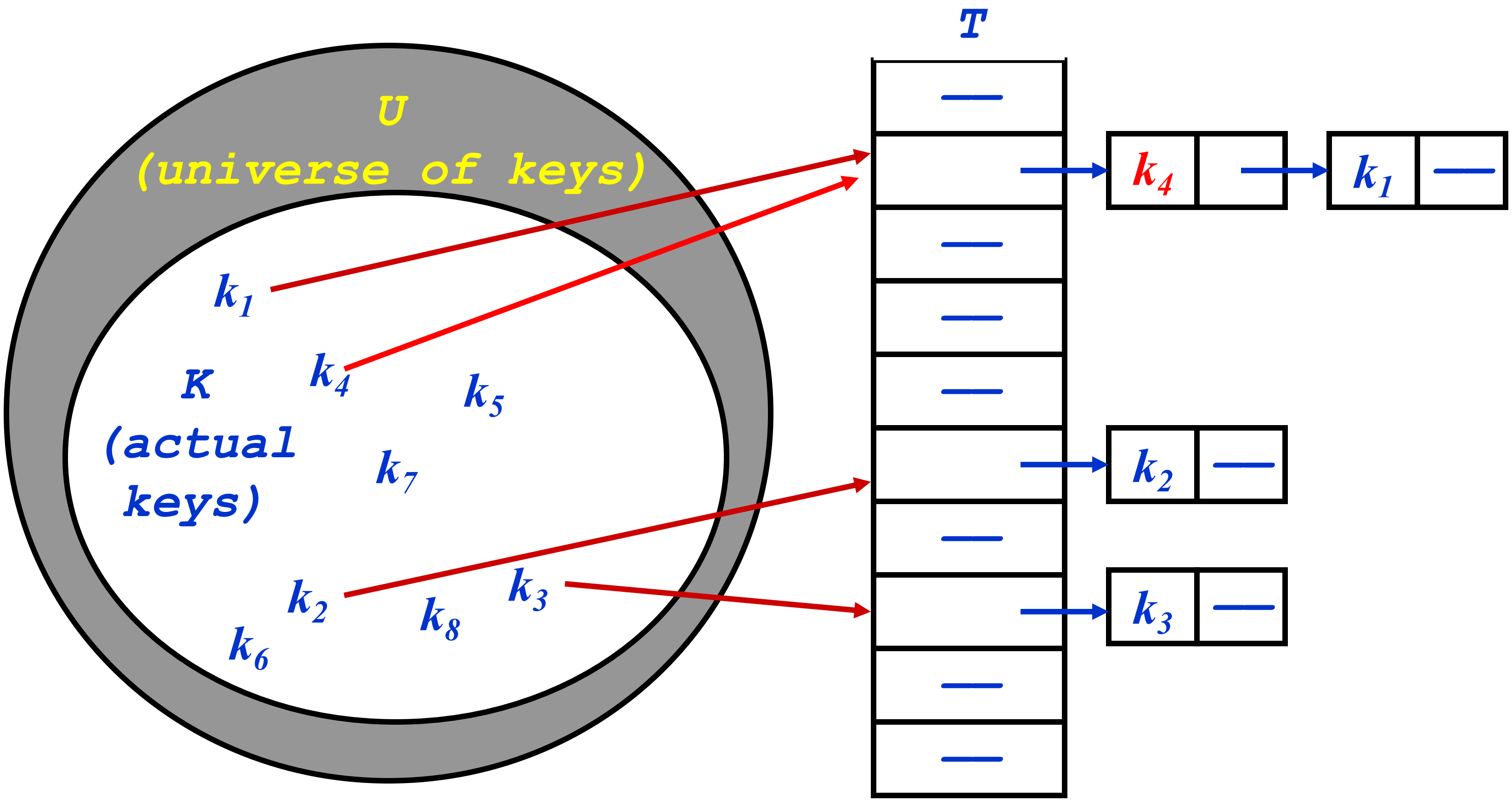
Chaining (insert at the head)



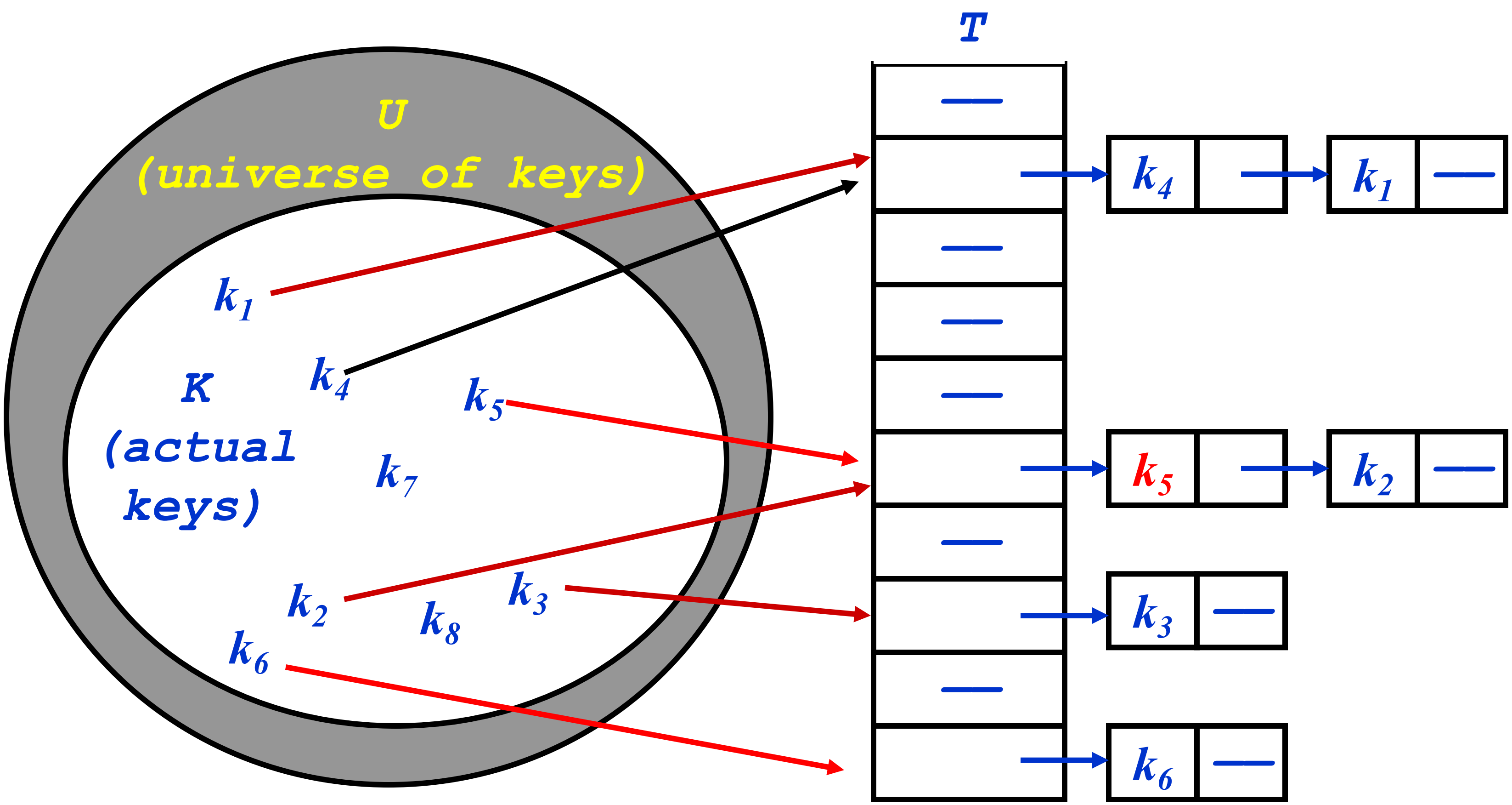
Chaining (insert at the head)



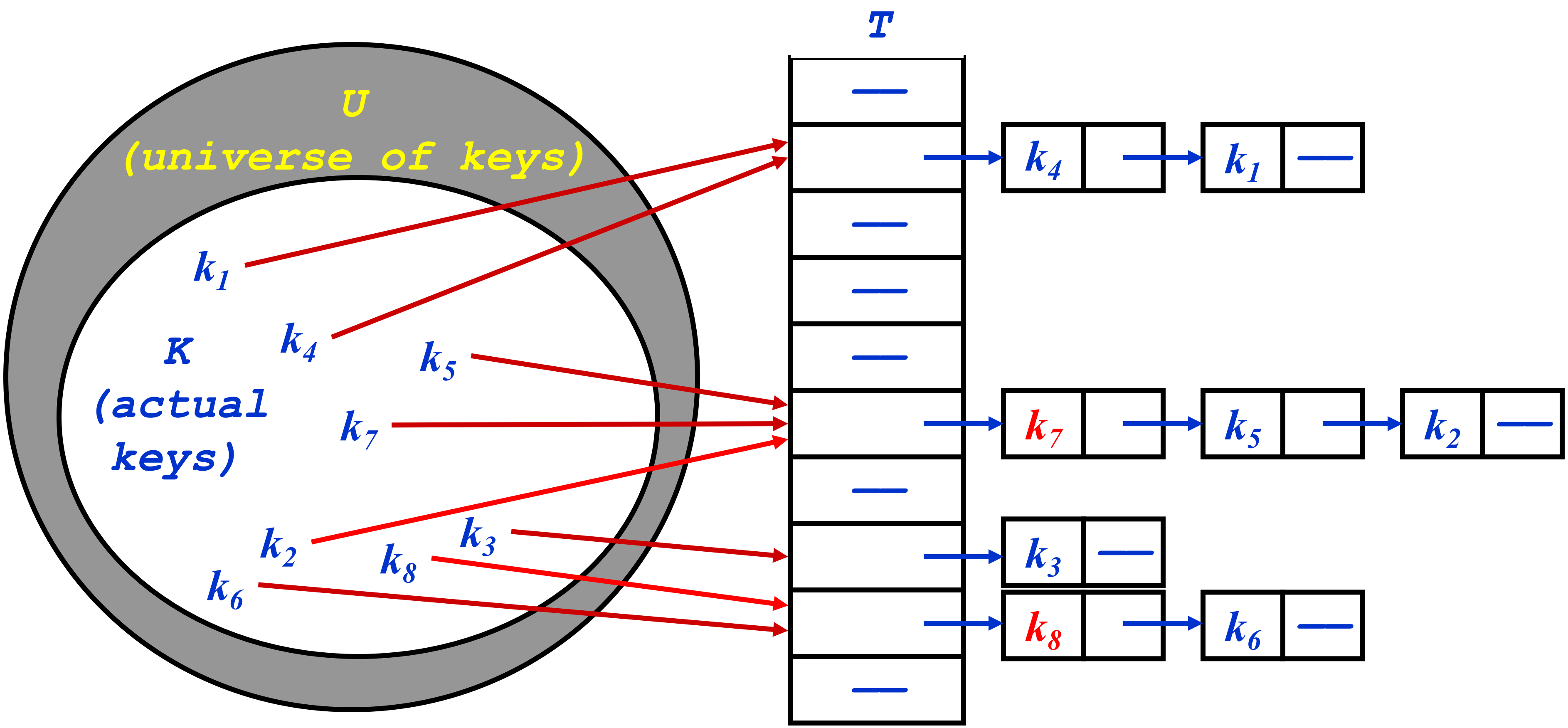
Chaining (insert at the head)



Chaining (insert at the head)



Chaining (insert at the head)



Operations

Direct-Hash-Search(T,k)

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

(running time is proportional to length of the list)

Direct-Hash-Insert(T,x) (worst case $O(1)$)

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

Direct-Hash-Delete(T,x)

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

(For singly linked list we might need to find the predecessor first. So the complexity is just like that of search)

Analysis of hashing with chaining

- Given a hash table with m slots and n elements
- The **load factor** $\alpha = n/m$
- The worst case behavior is when all n elements hash into the same location ($\theta(n)$ for searching)
- The average performance depends on how well the hash function distributes elements
- Assumption: **simple uniform hashing**: Any element is equally likely to hash into any of the m slot
- For any key $h(k)$ can be computed in $O(1)$
- Two cases for a search:
  - The search is unsuccessful
  - The search is successful

Unsuccessful search

Theorem 11.1 : In a hash table in which collisions are resolved by chaining, an unsuccessful search takes $\theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

Proof:

- Simple uniform hashing $\Rightarrow$ any key k is equally likely to hash into any of the m slots.
- The average time to search for a given key k is the time it takes to search a given slot.
- The average length of each slot is $\alpha = n/m$: the load factor.
- The time it takes to compute $h(k)$ is $O(1)$.
- $\Rightarrow$ Total time is $\theta(1+\alpha)$.

Successful Search

Theorem 11.2 : In a hash table in which collisions are resolved by chaining, a successful search takes $\theta(1+ \alpha/2)$, under the assumption of simple uniform hashing.

Proof:

- Simple uniform hashing $\Rightarrow$ any key k is equally likely to hash into any of the m slots.
- Note Chained-Hash-Insert inserts a new element in the front of the list
- The expected number of elements visited during the search is 1 more than the number of elements of the list after the element is inserted

Successful Search

- Take the average over the n elements

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

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

$$=1+\frac{\alpha}{2}-\frac{1}{2m} \qquad \textit{(3)}$$

- $(i-1)/m$ is the expected length of the list to which i was added. The expected length of each list increases as more elements are added.

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m =$ average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* $O(1+\alpha)$
- *What will be the average cost of a successful search?* $O(1 + \alpha/2) = O(1 + \alpha)$

Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- Three popular methods:
  - Division method
  - Multiplication method
  - Universal hashing

The 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
- Elements with adjacent keys hashed to different slots: good
- If keys bear relation to m: bad
- **In Practice: pick table size m = prime number not too close to a power of 2 (or 10)**

The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m\,(\underbrace{kA - \lfloor kA \rfloor}_{\textit{Fractional part of kA}})\rfloor$
- In practice:
  - Choose $m = 2^P$
  - Choose A not too close to 0 or 1
  - Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Universal Hashing

- When attempting to foil an malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly when the algorithm begins
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Need a family of hash functions to choose from
  - Think of quick-sort

Universal Hashing

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

Universal Hashing

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