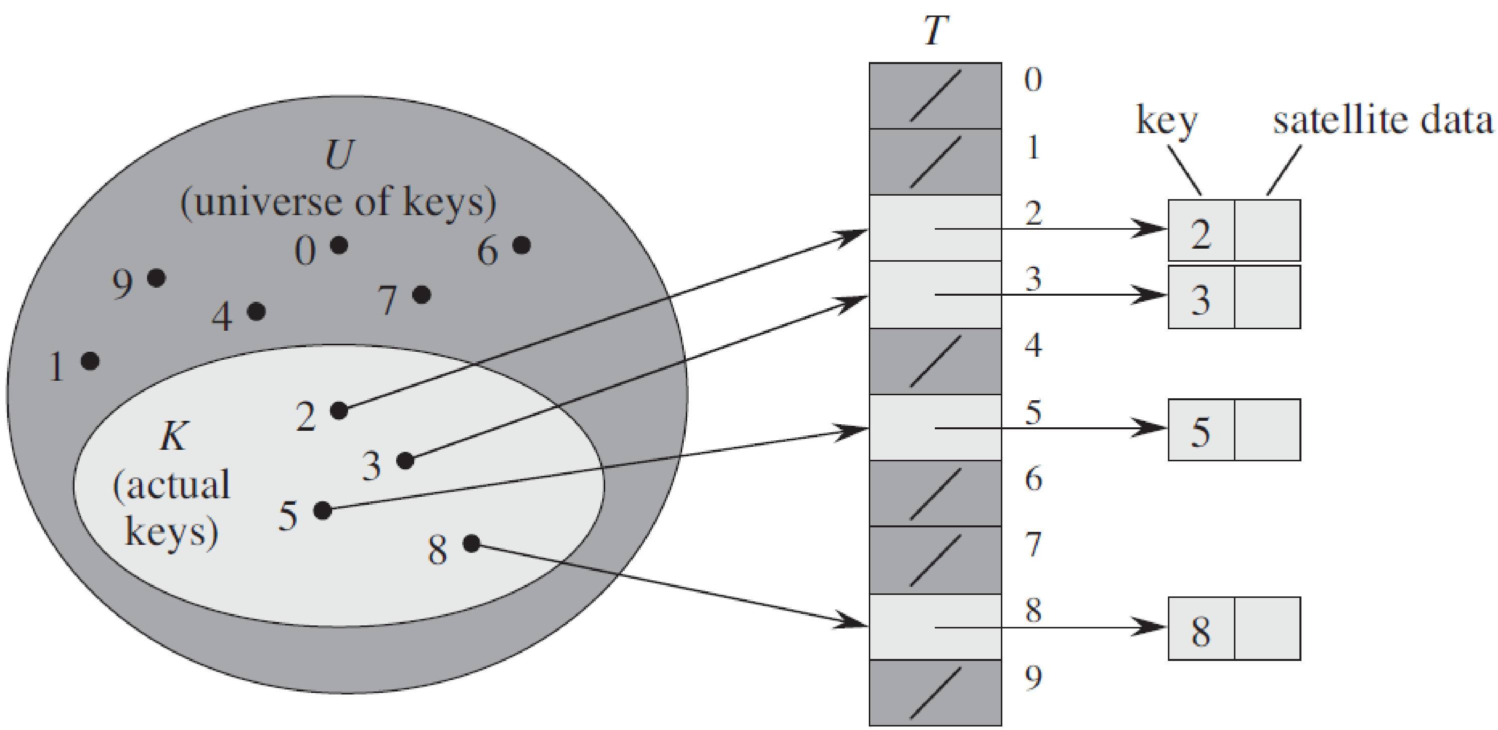
# EECE 2560:
## Fundamentals of Engineering Algorithms

Hash Tables

# Direct-Address Tables

- Direct addressing is a simple technique that works well when the universe *U* of keys is reasonably small assuming that no two elements have the same key.



# Direct-Address Table Operations

- Each of the following operations takes only $O(1)$ time.

DIRECT-ADDRESS-SEARCH$(T, k)$

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT$(T, x)$

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE$(T, x)$

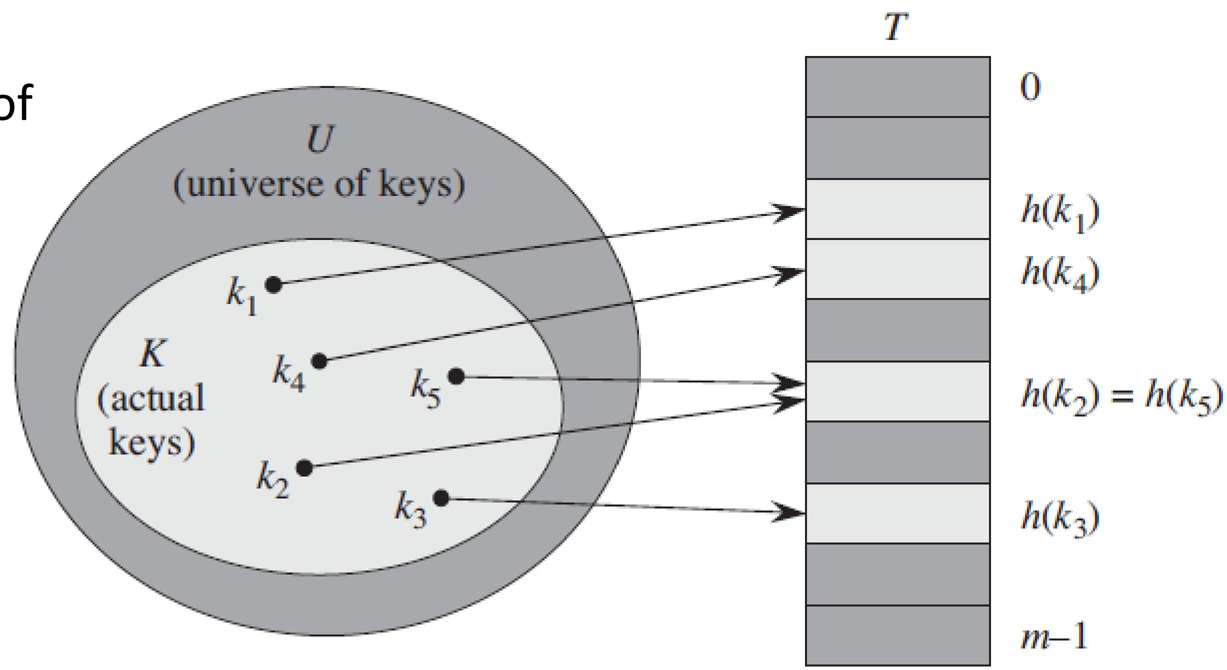
1 $T[x.key] = \text{NIL}$

# Hash Tables

- The downside of direct addressing is obvious: if the universe $U$ is large, storing a table $T$ of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer.
- Furthermore, the set $K$ of keys *actually stored* may be so small relative to $U$ that most of the space allocated for $T$ would be wasted.
- When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array.
- A hash table is an effective data structure for implementing ***dictionaries*** operations INSERT, SEARCH, and DELETE.
- Although searching for an element in a hash table can take as long as searching for an element in a linked list – O($n$) time in the worst case - in practice, hashing performs extremely well.

# Hash Functions

- With direct addressing, an element with key *k* is stored in slot *k*.
- With hashing, this element is stored in slot *h*(*k*); that is, we use a ***hash function*** *h* to compute the slot from the key *k*.
- A hash function *h* must be deterministic in that a given input *k* should always produce the same output *h*(*k*).
- In the shown example, *h* maps the universe *U* of keys into the slots of a ***hash table*** *T* [0 .. *m* - 1] where the size *m* of the hash table is typically much less than |*U*|

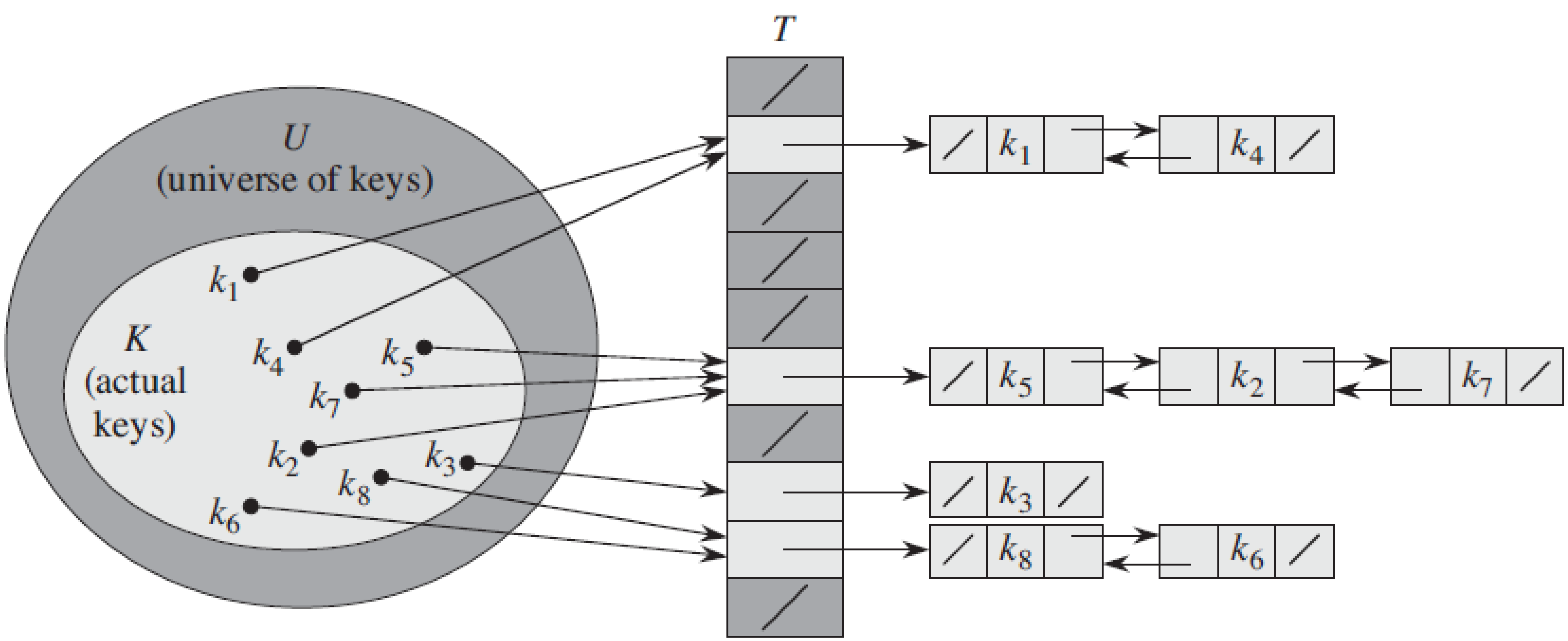


# Hash Collision

- Two keys may hash to the same slot. We call this situation a ***collision***.
- The ideal solution would be to avoid collisions altogether or at least minimizing their number.
- We might try to achieve this goal by choosing a suitable hash function *h*.
- Because $|U| > m$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible.

# Collision Resolution by Chaining

- In ***chaining***, we place all the elements that hash to the same slot into the same linked list, as shown.



# Hash Table Operations

- The dictionary operations on a hash table *T* are easy to implement when collisions are resolved by chaining:

CHAINED-HASH-INSERT$(T, x)$

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

CHAINED-HASH-SEARCH$(T, k)$

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

CHAINED-HASH-DELETE$(T, x)$

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

# Hash Functions

- A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the *m* slots, independently of where any other key has hashed to.

- Most hash functions assume that the universe of keys is the set $\mathbb{N}$ = {0,1,2,.....} of natural numbers.

- Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers.

# Division Hash Function

- In the ***division method*** for creating hash functions, we map a key $k$ into one of $m$ slots by taking the remainder of $k$ divided by $m$. That is, the hash function is $h(k) = k \bmod m$.
- For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$.
- Example of **bad** choice of $m$ is $2^p$, as $h(k) = p$ lowest-order bits of $k$.
  - We are better off designing the hash function to depend on all the bits of the key.
- A prime not too close to an exact power of 2 is often a good choice for $m$.
  - *Example:* we wish to allocate a hash table, with collisions resolved by chaining, to hold elements with keys range from 0 to 2000. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$ because it is a prime near 2000/3 but not near any power of 2.

# Resolving Collisions by Open Addressing

- In ***open addressing***, all elements occupy the hash table itself.
  - That is, each table entry contains either an element of the set or NIL.
- When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.
- Unlike chaining, no lists and no elements are stored outside the table.
- Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made.
- The advantage of open addressing is that it avoids pointers altogether.
- Instead of following pointers, we *compute* the sequence of slots to be examined.

# Open Addressing (Cont'd)

- To perform insertion using open addressing, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key.
- To determine which slots to probe, we extend the hash function to include the ***probe number*** (starting from 0) as a second input.
- For every key $k$, the probe sequence

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

  should be a permutation of $\{0, 1, \ldots, m-1\}$.
- So that every hash-table position is eventually considered as a slot for a new key as the table fills up.

# Open Addressing - Insert

- The following HASH-INSERT procedure takes as input a hash table *T* and a key *k*. It either returns the slot number where it stores key *k* or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

# Open Addressing - Search

- The algorithm for searching for key *k* probes the same sequence of slots that the insertion algorithm examined when key *k* was inserted.
- Therefore, the search can terminate (unsuccessfully) when it finds an empty slot.
(This argument assumes that keys were not deleted from the hash table.)

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

# Open Addressing - Delete

- To delete a key from slot *i* of an open-address hash table, we cannot simply mark that slot as empty by storing NIL in it. If we do that, we might be unable to retrieve any key *k* that was inserted after slot *i* when slot *i* was probed as occupied.
- We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL.
- We would then modify the procedure HASH-INSERT to treat such a slot as if it were empty so that we can insert a new key there.
- We do not need to modify HASH-SEARCH, since it will pass over DELETED values while searching.

# Linear Probing

- Given an ordinary hash function $h'(k)$, linear probing uses the hash function:

$$h(k,i) = (h'(k) + i) \bmod m \text{ for } i = 0,1, \ldots , m\text{-}1.$$

- Given key *k*, we first probe T[*h'*(*k*)], i.e., the slot given by the auxiliary hash function.
- We next probe slot T[*h'*(*k*) + 1], and so on up to slot T[*m* -1].
- Then we wrap around to slots T[0], T[1], … until we finally probe slot T[*h'*(*k*) - 1]. Because the initial probe determines the entire probe sequence, there are only *m* distinct probe sequences.
- In the shown example T[*h'(k)*] = i