Algorithms & Data Structures

Lecture 07
Hash Tables

Giovanni Bacci
giovbacci@cs.aau.dk

Outline

- Direct-address tables
- Hash-tables
- Hash functions
- Open addressing

Intended Learning Goals

KNOWLEDGE

- Mathematical reasoning on concepts such as recursion, induction, concrete and abstract computational complexity
- Data structures, algorithm principles e.g., search trees, hash tables, dynamic programming, divide-and-conquer
- Graphs and graph algorithms e.g., graph exploration, shortest path, strongly connected components.

SKILLS

- Determine abstract complexity for specific algorithms
- Perform complexity and correctness analysis for simple algorithms
- Select and apply appropriate algorithms for standard tasks

COMPETENCES

- Ability to face a non-standard programming assignment
- Develop algorithms and data structures for solving specific tasks
- Analyse developed algorithms

Why Hash Tables

- Many applications require dynamic sets that support only the dictionary operations INSERT, SEARCH, and DELETE

- Elements in the dictionary are assumed to have a key attribute and possibly other satellite data

- A hash table is an effective data structure for implementing dictionaries
  - Under reasonable assumptions, the average time to search for an element in $O(1)$.

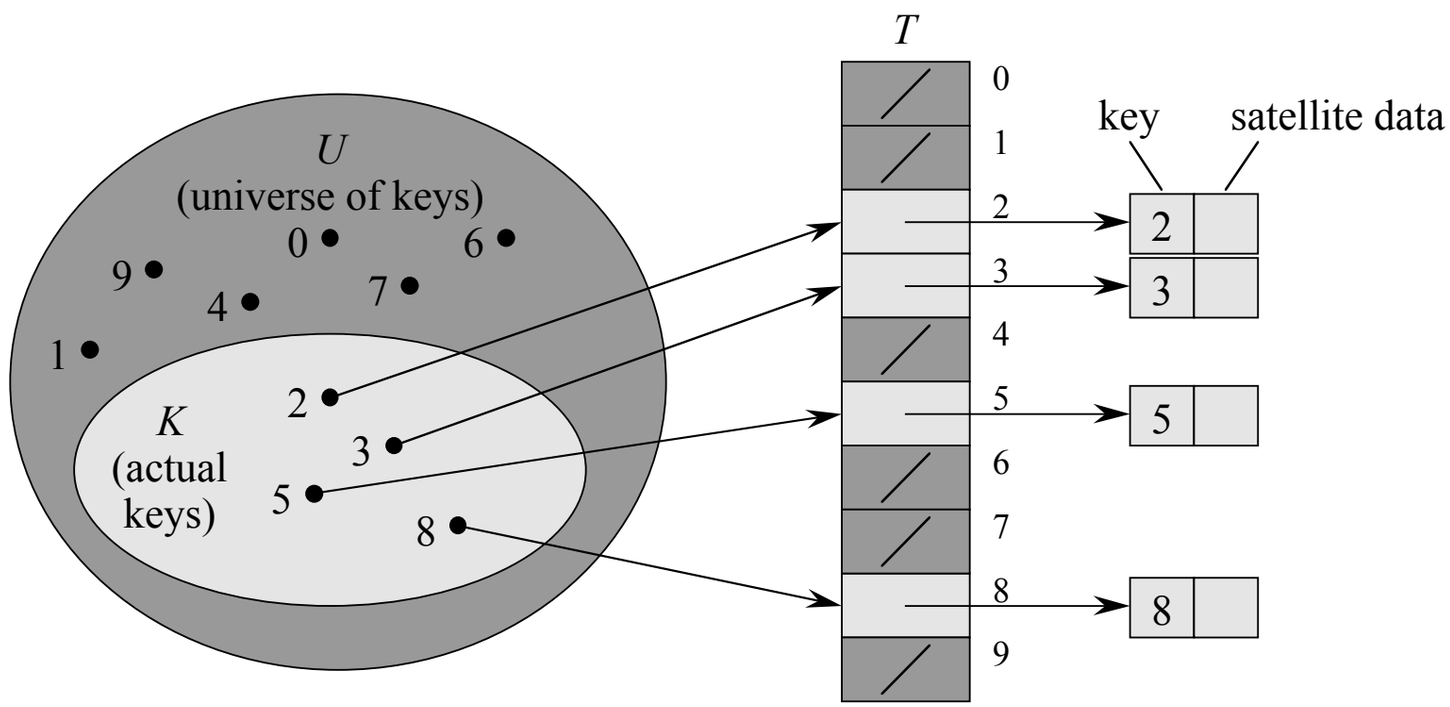
Direct-address tables

Direct-address tables

Assumptions

1. Suppose that an application needs a dynamic set in which each element has a key drawn from the **universe** $U = \{0,1,2,\ldots,m-1\}$, where m is not too large.
2. We shall assume that no two elements have the same key

- We use an array $T[0..m-1]$, called **direct-address table**, in which each position, called **slot**, corresponds to a key in the universe U.

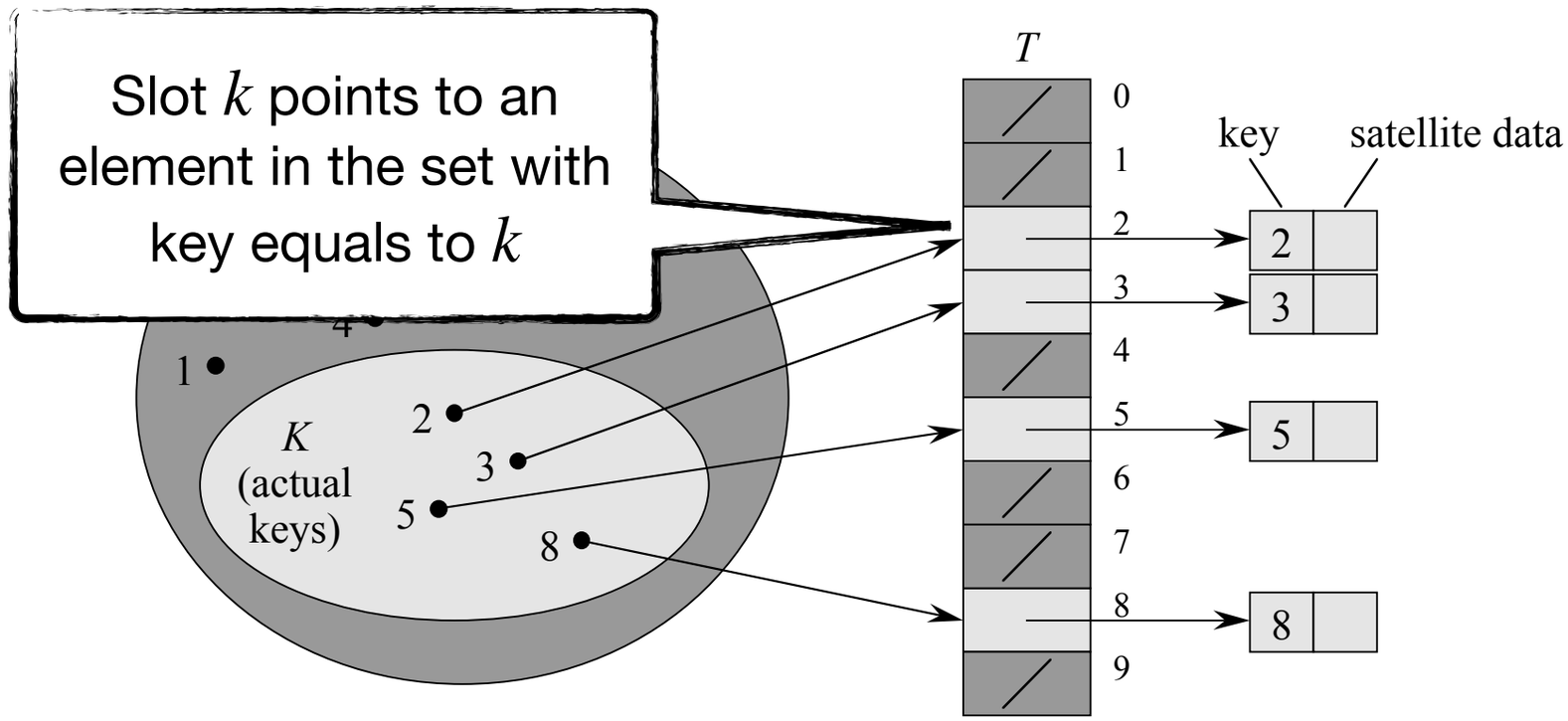


Direct-address tables

Assumptions

1. Suppose that an application needs a dynamic set in which each element has a key drawn from the **universe** $U = \{0,1,2,\ldots,m-1\}$, where m is not too large.
2. We shall assume that no two elements have the same key

- We use an array $T[0..m-1]$, called **direct-address table**, in which each position, called **slot**, corresponds to a key in the universe U.

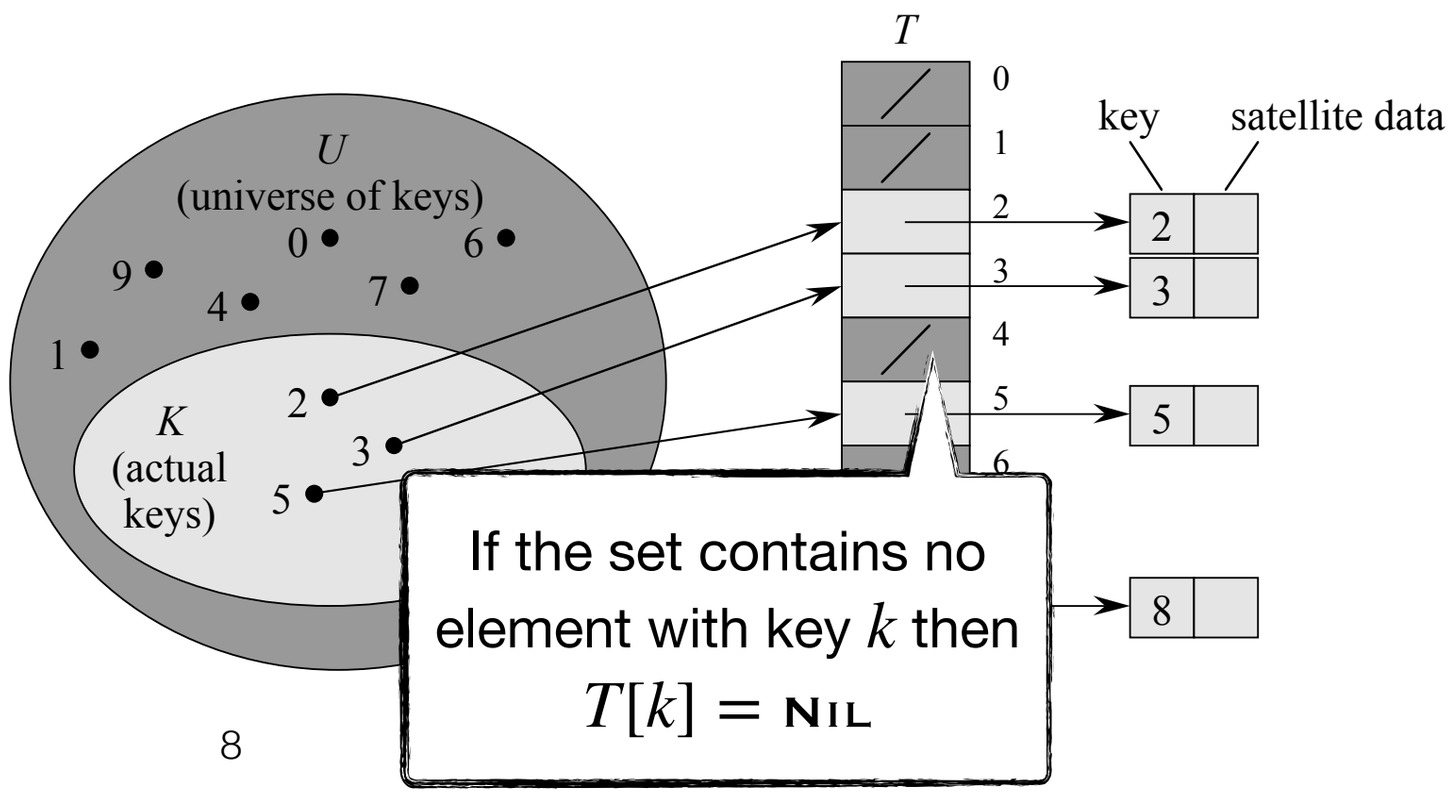


Direct-address tables

Assumptions

1. Suppose that an application needs a dynamic set in which each element has a key drawn from the **universe** $U = \{0,1,2,\ldots,m-1\}$, where m is not too large.
2. We shall assume that no two elements have the same key

- We use an array $T[0..m-1]$, called **direct-address table**, in which each position, called **slot**, corresponds to a key in the universe U.



Operations on Direct-address tables

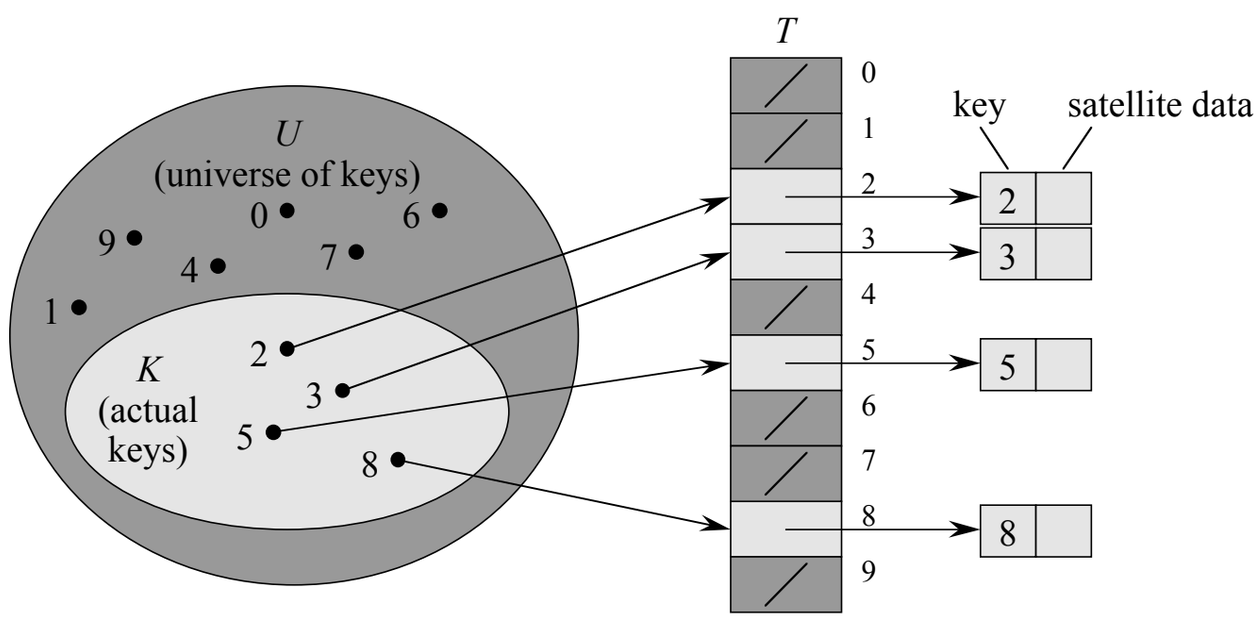
The dictionary operations are trivial and take only $\Theta(1)$ time

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]
```

Returns the element pointed by the slot k

```
DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x
```

```
DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```



Operations on Direct-address tables

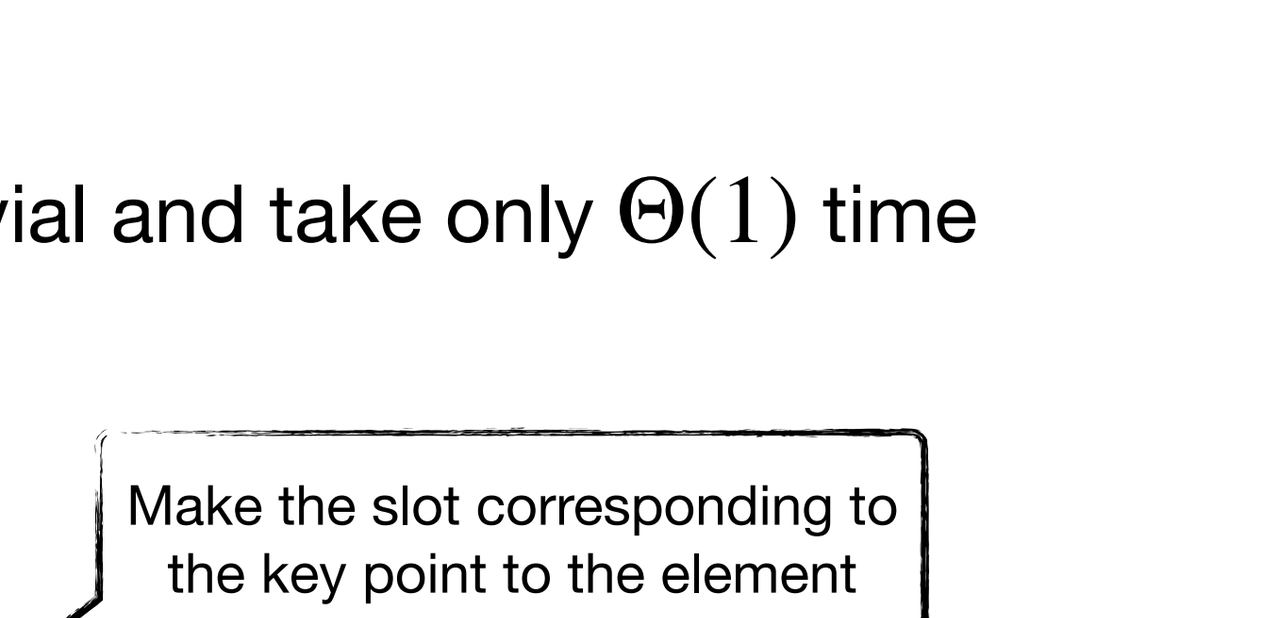
The dictionary operations are trivial and take only $\Theta(1)$ time

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]
```

Make the slot corresponding to the key point to the element

```
DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x
```

```
DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```



Operations on Direct-address tables

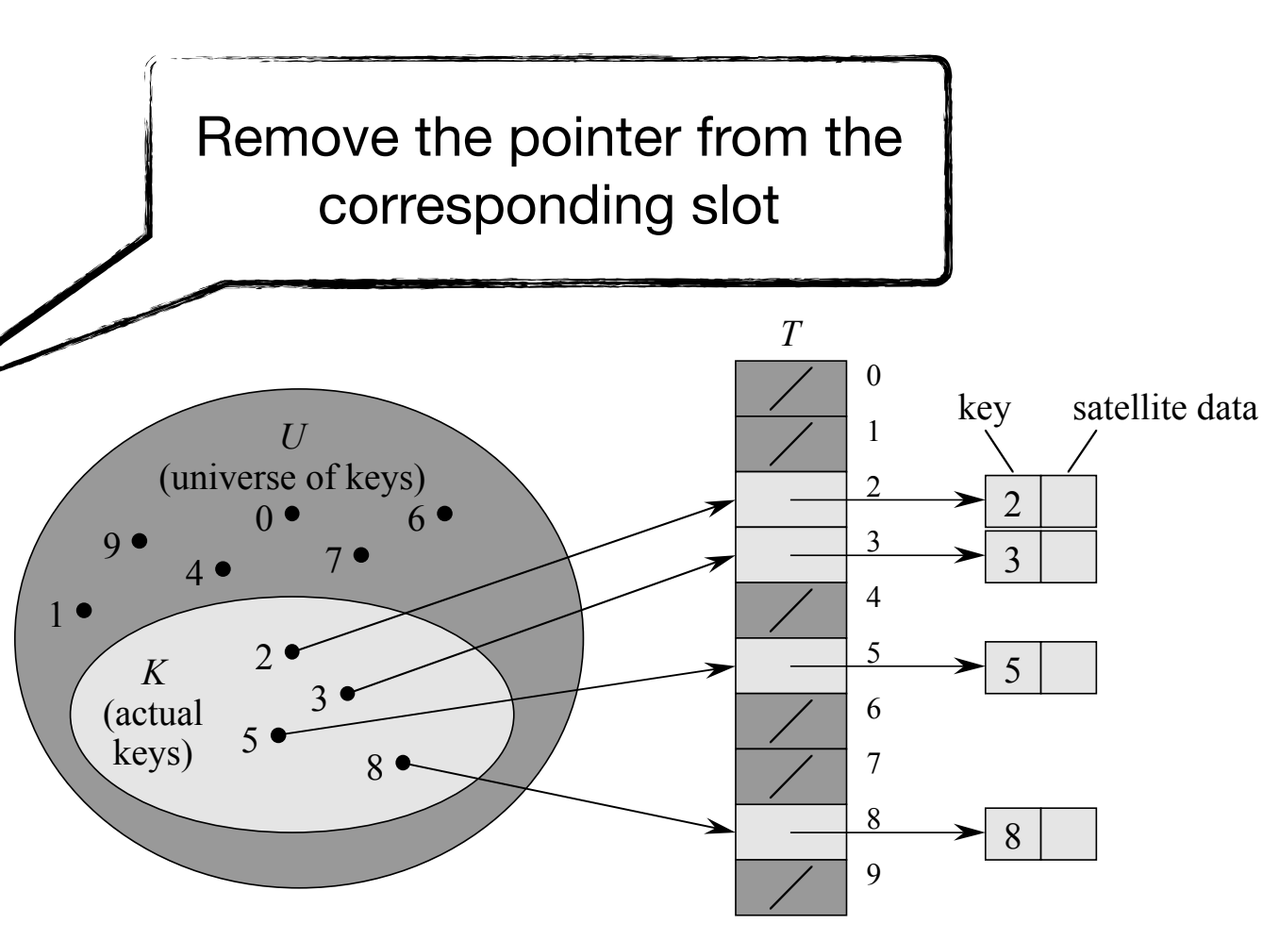
The dictionary operations are trivial and take only $\Theta(1)$ time

```
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] = NIL
```

Remove the pointer from the corresponding slot



Direct-address tables

- Sometimes it is unnecessary even to store the key of the object (it is the index of the slot in the table),
- However we must have some way to tell whether the slot is empty.

Quiz

- Have we already seen an algorithm that use some form of direct-address tables?

Answer

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

The array C used to count the occurrences of a number in A (and final position of the number in B) is in fact a direct-address table!

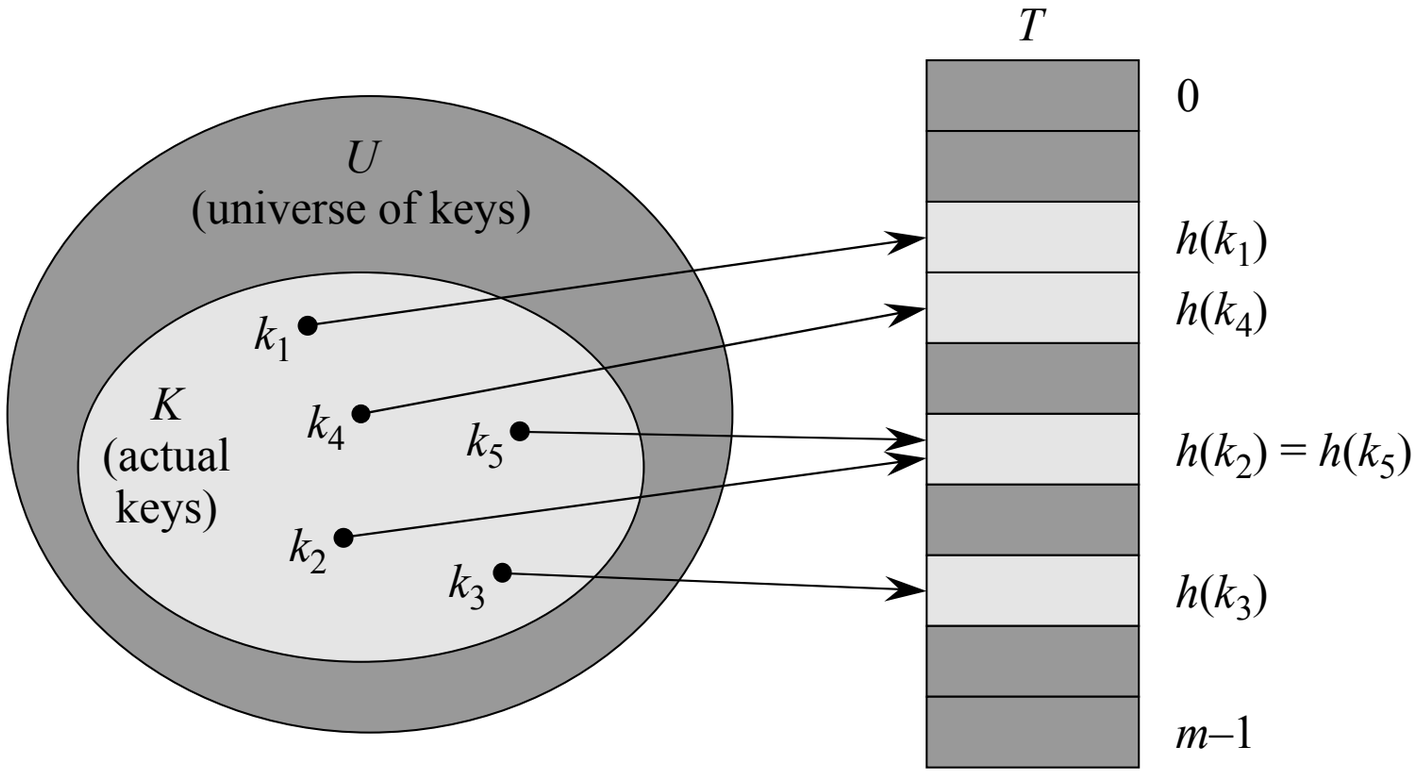
Hash Tables

Hash Tables

- Direct-address tables are not a good solution when the universe U of keys is large.
  - Storing a table T of size $|U|$ may be impractical.
  - The set K of **keys *actually stored*** may be so small relative to U (denoted $|K| \ll |U|$) that most space allocated for T would be just wasted.
- When $|K| \ll |U|$ we can reduce the storage requirement to $\Theta(|K|)$ while maintaining the benefit that searching cost (on average) only $O(1)$ time
- By mapping a key k to its slot using a **hash function** h

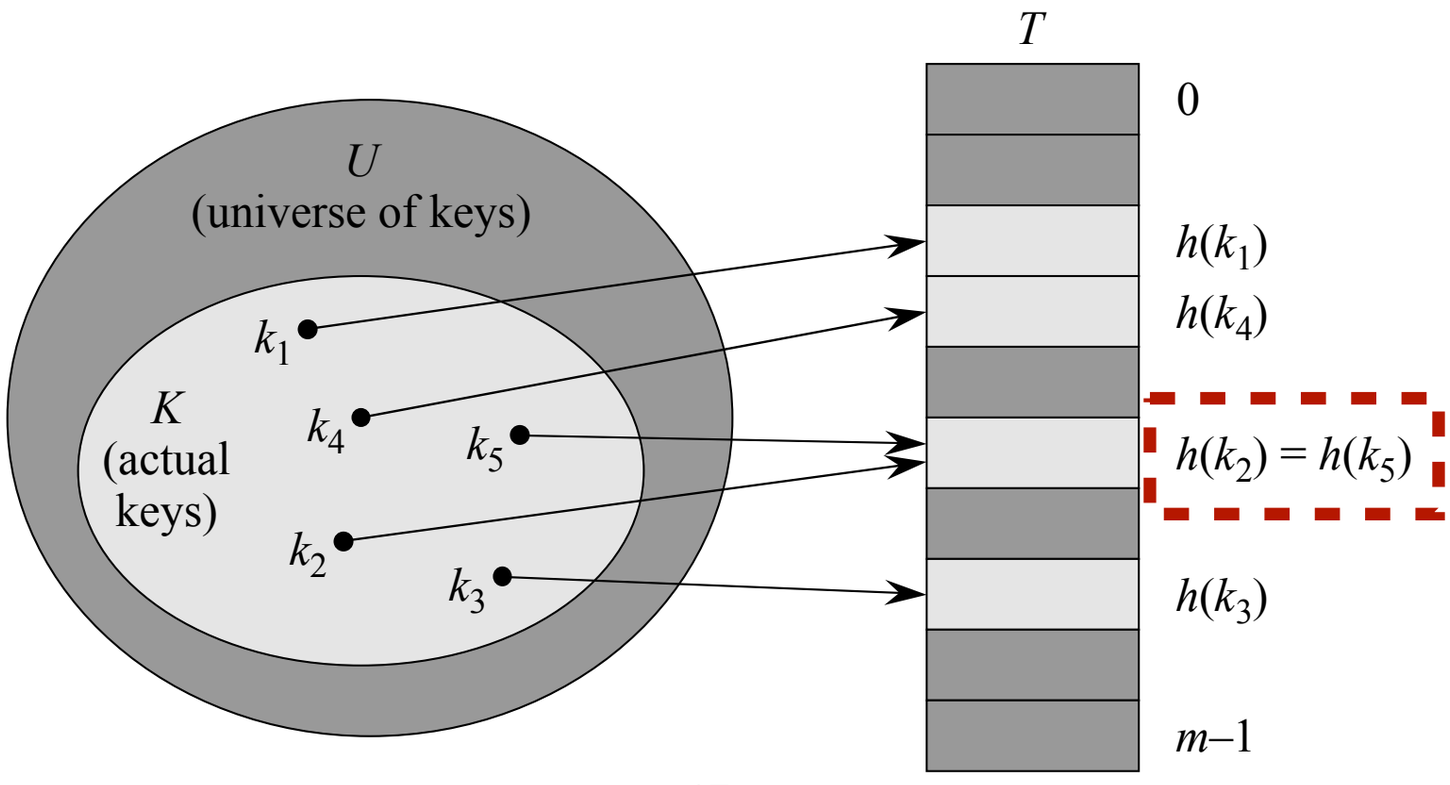
Hash Tables: the idea

- We use an array $T[0..m-1]$, called **hash table**, to store the slots that point to elements in the dictionary.
- We use a **hash function** $h: U \to \{0,1,\ldots,m-1\}$ to map a key $k \in U$ to a slot in T.
- This allows us to have $m < |U|$



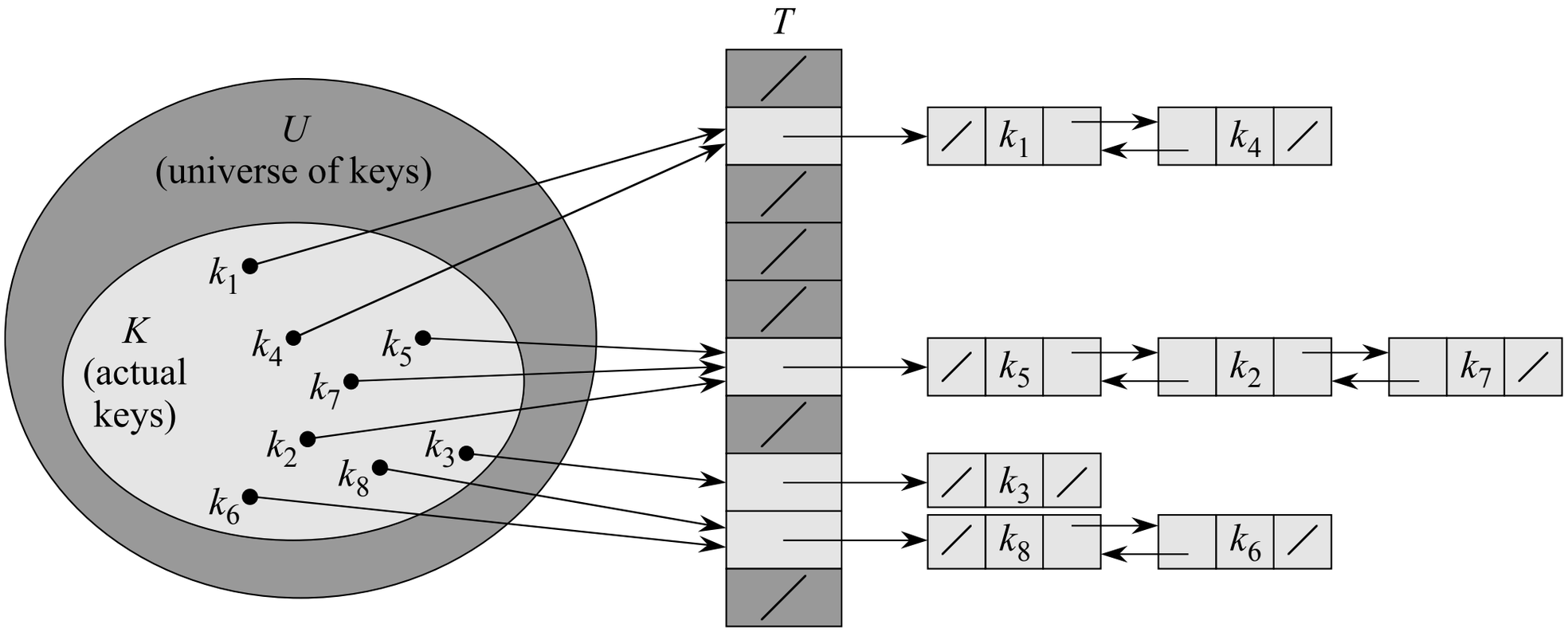
Hash Tables: collisions

- Two keys may hash to the same slot, i.e., $h(k_1) = h(k_2)$.
- We call this situation **collision**.
- The ideal situation would be to avoid collisions as much as possible.
- When $m < |U|$ its impossible to avoid all collisions



Collision resolution by chaining

- In **chaining**, we place all the elements that hash to the same slot into the same (doubly) **liked list**.
- Hence, collisions are resolved by adding the element to the list pointed by the corresponding slot



Hash Tables: operations

- Dictionary operations are easy to implement when collisions are resolved by chaining

CHAINED-HASH-INSERT(T, x)

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

Insertion takes $\Theta(1)$ time if we assume that the element x is not in the table. Otherwise, one should first check this (at additional cost) by searching for an element with key $x.key$

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 Tables: operations

- Dictionary operations 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 $T[h(k)]$

We simply use the linear search procedure for linked lists

CHAINED-HASH-DELETE(T, x)

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

Hash Tables: operations

- Dictionary operations 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 $T[h(x.key)]$

We can delete the element in $\Theta(1)$ time if we use **doubly linked lists**.
With singly linked lists deletion can take liner time in the size of the list

Worst-case Run-time (using chaining)

How long does it take to search for a key k in a hash table T with m slots and n elements?

Worst-case scenario: all n keys hash to the same slot creating a list of length n, hence

$$\Theta(n) + \Theta(1) = \Theta(n)$$

Worst-case runtime for searching a the linked list of length n

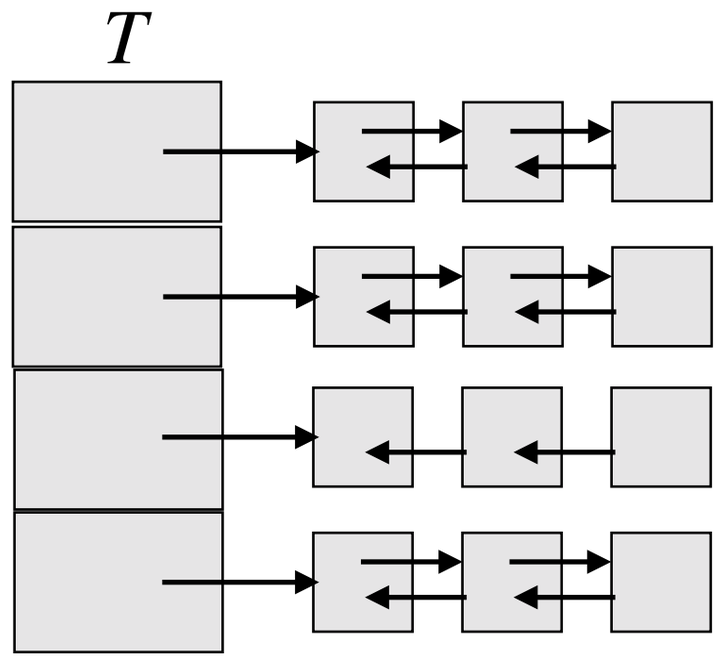
We assume that it takes constant time to compute the hash value $h(k)$

- Clearly we do not use hash tables for their worst-case performance! …otherwise, better using liked lists.

Average-case Run-time (using chaining)

Definition: we define the **load factor** α of T as n/m, that is the average number of elements stored in a chain.

- The **average-case run-time depends on how well the hash function h distributes the set of keys among the m slots**, on average.
- **Assumption**: any given given key is equally likely to hash into any of the m slots. We call this assumption **simple uniform hashing.**
  - Formally, it says that for all slot $0 \leq j \leq m, E[n_j] = \alpha,$
  - where n_j is the length of the list $T[j]$.



Average-case Run-time (using chaining)

- Let us consider the expected number of elements examined for searching the key k
- That is, the expected number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k
- We shall consider two cases:

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Theorem 11.2

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Average-case Run-time (using chaining)

What can conclude from these Theorems?

- If the number of slots in T is at least proportional to the number of elements in T, we have $n = O(m)$
- Then, $\alpha = n/m = O(m)/m = O(1)$

Overall performance Hash Tables (with chaining):

- Searching for a key: worst-case $\Theta(n)$, average-case $O(1)$
- Inserting an element: worst-case $\Theta(1)$
- Deleting an element: worst-case (doubly linked lists) $\Theta(1)$

Hash functions

What makes a good h?

- A good hash function **satisfies (approximately) the simple uniform hashing** assumption
- Unfortunately we have **no general way to check this condition**, since we rarely know the probability distribution from which the keys are drawn
- In practice **we can often employ heuristic techniques** to create hash functions that work well in practice
  - Qualitative information about the distribution of keys may be useful in this design process
  - E.g., symbol table use alpha-numeric keys, and similar names occurs often in the same program
- A good approach derives the **hash value** in a way that it is **independent of any pattern** that may exists in the set of keys

Hash functions: domain

- Recall $h: U \to \{0,\ldots,m-1\}$
- Most hash functions assume the universe of keys $U = \mathbb{N}$
- If the keys are not natural numbers we find a way to interpret them as such (e.g., we can encode characters using their ASCII code and strings expressed as radix-128 integer)

The division method

- In the division method for creating hash functions, we map a key k into one of m slots using

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

  - It often works well when m is a prime not too close to an exact power of 2 (see Exercise 11.3-3 in CLRS-3)

Example:

- Suppose we want to allocate a hash table to hold roughly $n = 2000$ character strings, where a character has 8 bits.
- We don't mind examining on average 3 elements in an unsuccessful search, an so we use $m = 701$ because 701 is a prime near $2000/3 \cong 666.67$ but it is not near any power of 2, indeed $2^9 = 512 < 701 < 1024 = 2^{10}$

The multiplication method

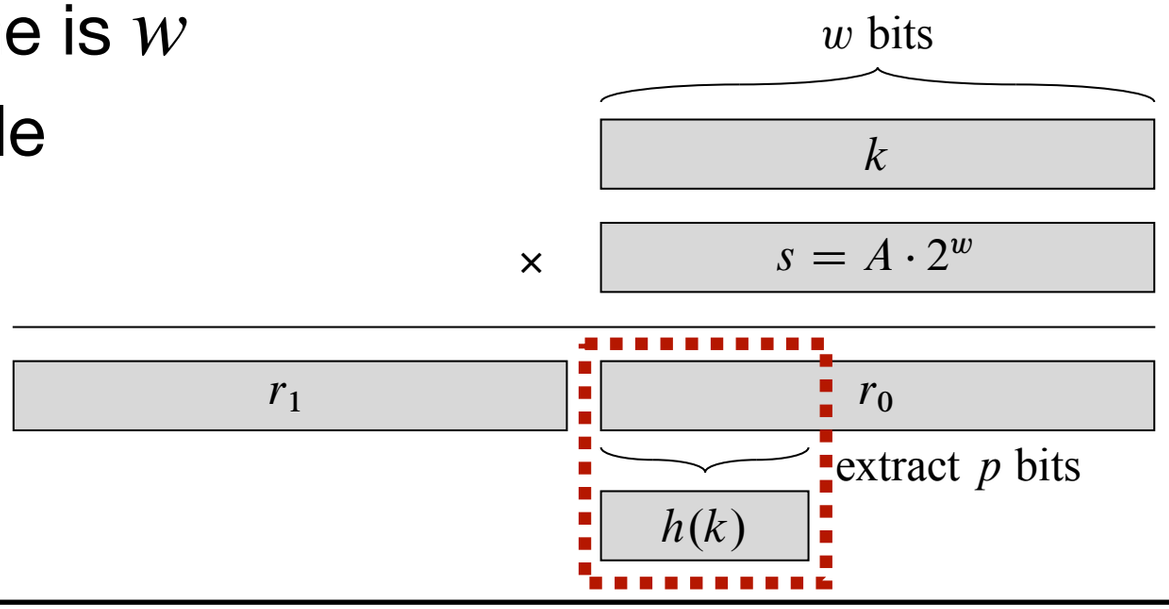
- In the multiplication method for creating hash functions, we map a key k into one of m slots using a constant $A \in (0,1)$

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

- Advantage: **the value of m is not critical.** Typically one chooses it to be a power of 2 ($m = 2^p$ for some $p \in \mathbb{N}$)

The hash function is easily computed:

- Suppose the word size of the machine is w bits and that the key k fits into a single word.
- Using $m = 2^p$ for some $p \in \mathbb{N}$, and
- $A = s/2^w$ for some $s \in \mathbb{N} \cap (0,2^w)$



Open addressing

Open Addressing

- In open addressing all elements occupy the hash table itself, that is each table slot contains either an element of the dynamic set or Nil
- Unlike chaining, no list and no elements are stored outside the table
- The hash table can “fill up” so that no further insertions can be made
- The load factor α can never exceed 1 (because $m \geq n$)
- It avoids pointers altogether $\Longrightarrow$ more slots for the same amount of memory

Open addressing: collisions

- 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
- Instead of following the fixed order $0,1,2\ldots,m-1$, the order depends on the key
- We extend the hash function h to **include the probe number** as a second input:

$$h\colon U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$$

- The **probe sequence** $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ must be a permutation of $\langle 0,1,\ldots,m-1\rangle$, so that every slot is eventually considered as candidate spot for k

Open addressing: insertion

- **Idea:** follow the probing order $\{h(k,i)\}_{i=1..m-1}$ until the first *free* spot is found

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

Initialise probe number

Current slot

If the slot is free insert the key, otherwise move to the next probe number

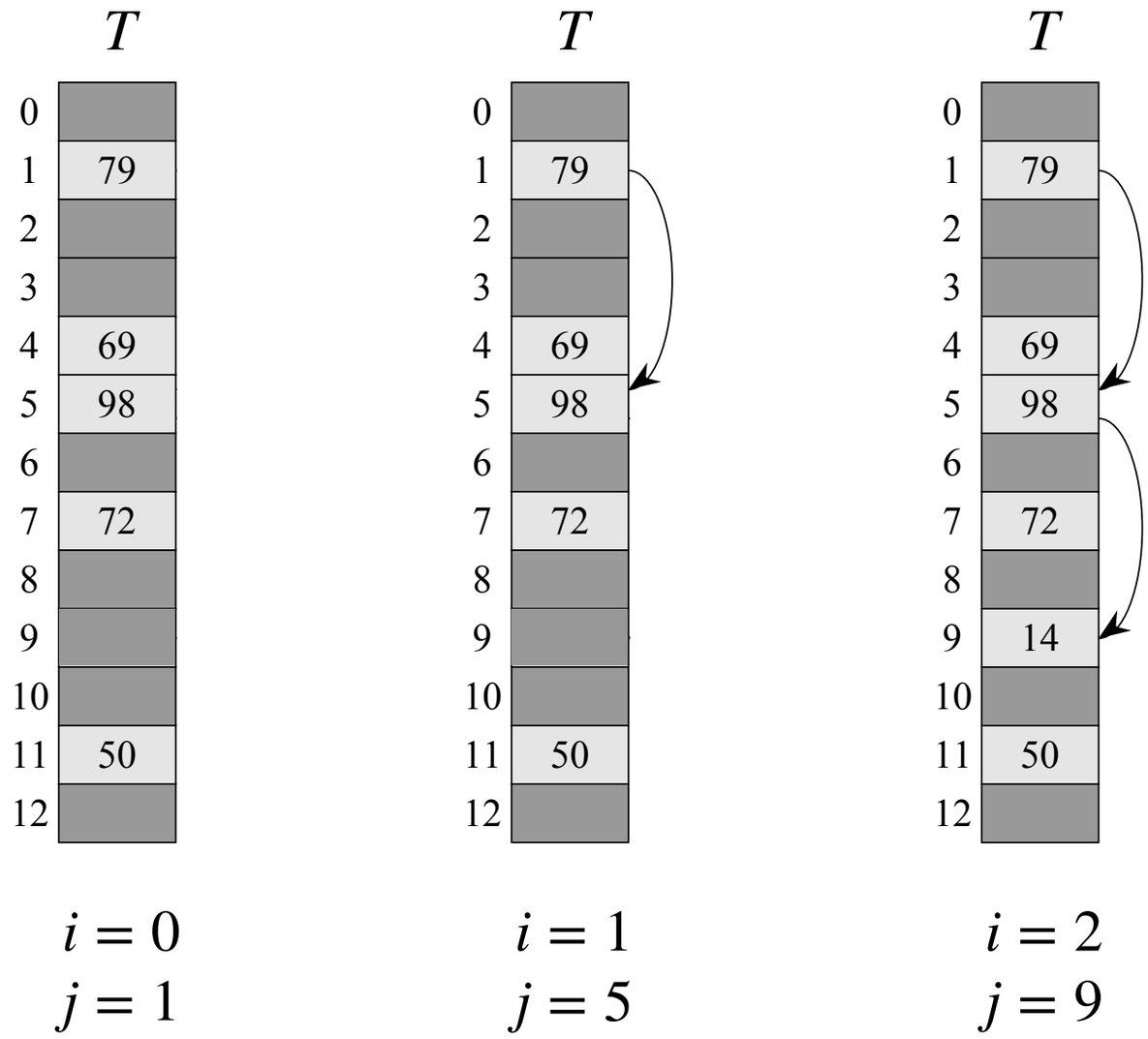
We have probed the whole table and no spot is available.

(*) Assumption: the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k

Open addressing: insertion

- Consider the table T, and key $k = 14$
- Assume $h(14,0) = 1$, $h(14,1) = 5$, and $h(14,2) = 9$

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```



Open addressing: search

- **Idea:** follow the probing order $\{h(k,i)\}_{i=1..m-1}$ until the key k is found

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Initialise probe number

Current slot

If the slot contains the key return, otherwise move to the next probe number

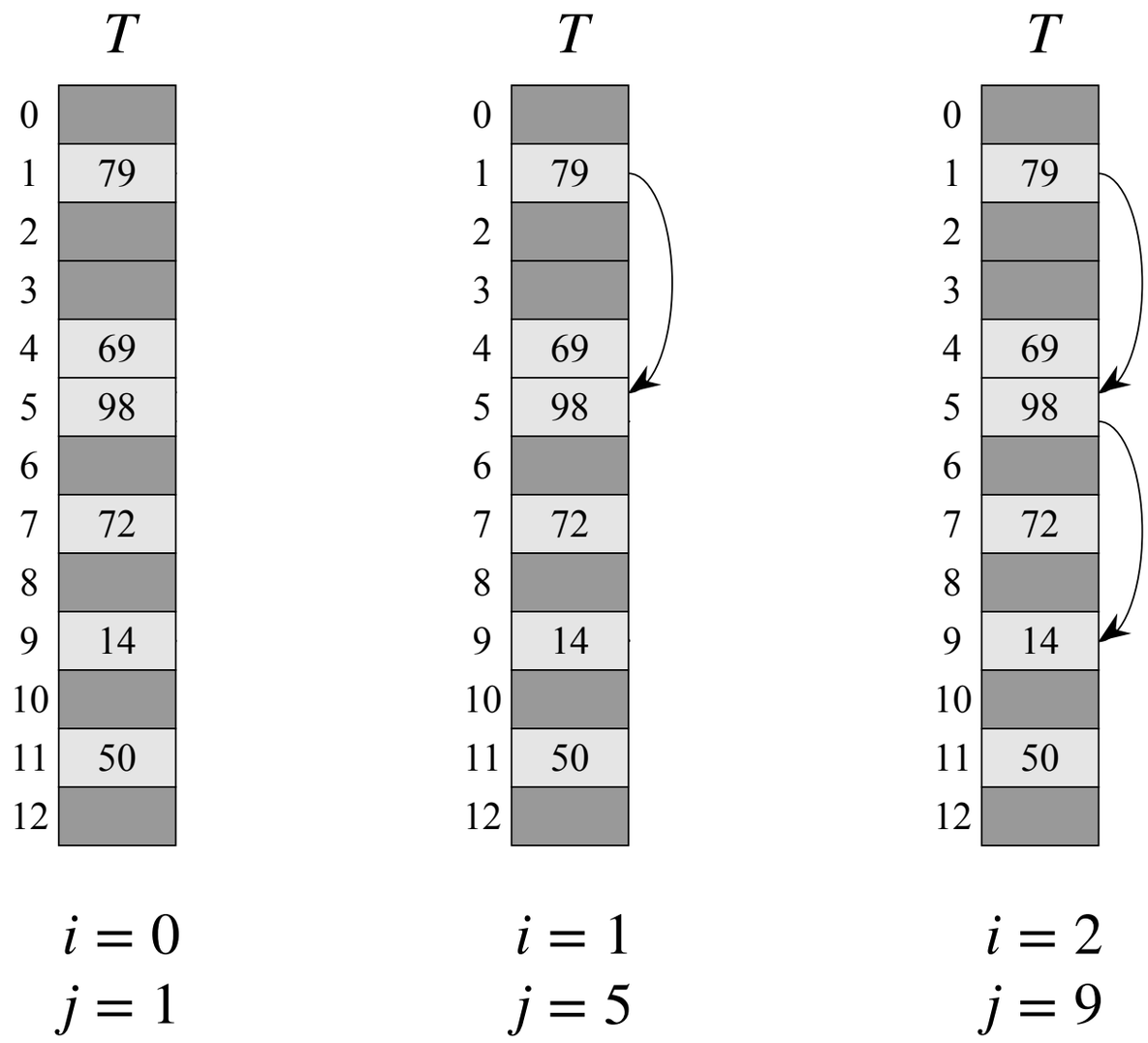
We have probed the whole table and k was not found.

(*) Assumption: the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k

Open addressing: search

- Consider the table T, and key $k = 14$
- Assume $h(14,0) = 1$, $h(14,1) = 5$, and $h(14,2) = 9$

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```



What about deletion?

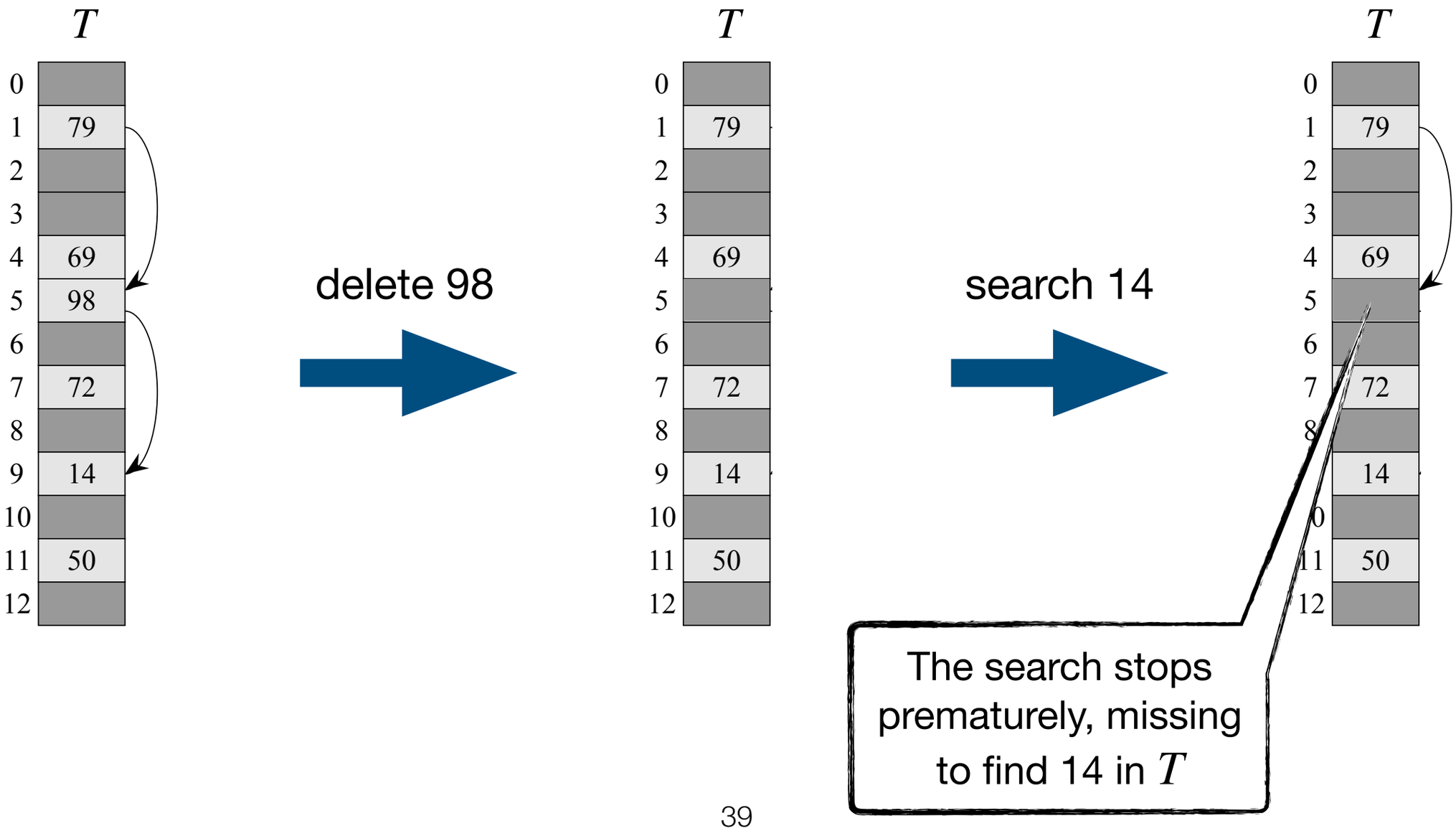
- Deletion in open addressing requires a bit of thoughts
- When we delete a key from slot j we cannot simply mark it as empty by storing $\textsc{Nil}$ in the corresponding slot.

Quiz

- What problems may occur?

Answer

- Operations follow the **probe sequence** $\langle h(k,0), h(k,2), \ldots, h(k, m-1)\rangle$ until they encounter the first empty slot



What about deletion?

- Deletion in open addressing requires a bit of thoughts
- When we delete a key from slot j we cannot simply mark it as empty by storing NIL in it.

- We can solve the problem by making the slot storing the special value DELETED instead of NIL
- We then modify HASH-INSERT to treat such elements as NIL
- While HASH-SEARCH is left unchanged since it will pass the special node while searching.
- However, now search time does not depend any longer on the load factor $\alpha = n/m$. Hence, if deletion is needed, better using chaining

Average-case Run-time (open-address)

- We express our analysis in terms of load factor $\alpha = n/m$
- Recall that with open addressing, at most one element occupies each slot, thus $n \leq m$ and $0 \leq \alpha \leq 1$
- **Assumption:** we use uniform hashing:
  - The probe sequence $\langle h(k,0), h(k,2), \ldots, h(k, m-1) \rangle$ used to insert or search of each key k is equally likely to be any permutation of $\langle 0,1,\ldots, m-1 \rangle$

Average-case Run-time (open-address)

The analysis studies the expected number to probes for hashing with open addressing under uniform hashing.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Intuitive Interpretation:

$$1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

The first probe is always performed

Average-case Run-time (open-address)

The analysis studies the expected number to probes for hashing with open addressing under uniform hashing.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Intuitive Interpretation:

$$1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

With probability α, the first probe doesn't find a free slot, so we need to probe a second time

Average-case Run-time (open-address)

The analysis studies the expected number to probes for hashing with open addressing under uniform hashing.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Intuitive Interpretation:

$$1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

With probability α^2, the first 2 probes don't find a free slot, so we need to probe a third time

Average-case Run-time (open-address)

The analysis studies the expected number to probes for hashing with open addressing under uniform hashing.

Theorem 11.6
Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

- If α is constant **Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time**.

- E.g. if the hash table is $50\,\%$ full, the average number of probes is at most $1/(1-0.5) = 2$
- If the table is $90\,\%$ full, we have $1/(1-0.9) = 10$

Average-case Run-time (open-address)

The analysis studies the expected number to probes for hashing with open addressing under uniform hashing.

Theorem 11.8
Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} ,$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

- If α is constant **Theorem 11.8 predicts that a successful search runs in $O(1)$ time.**
- E.g. if the table is $50\,\%$ full, the average number of probes is at most $1/\alpha \ln 1/(1-\alpha) = 2 \ln 2 \cong 1.387$
- If the table is $90\,\%$ full, we have $1/\alpha \ln 1/(1-\alpha) = 10/9 \ln 10 \cong 2.559$

Hashing in open addressing

- In our analysis we assumed uniform hashing: the probe sequence of each key is equally likely to be any of the $m!$ possible permutations of $\langle 0,1,\ldots,m-1\rangle$
- True uniform hashing is very difficult to implement
- We present three commonly used techniques:
  - Linear probing
  - Quadratic probing
  - Double hashing

Linear Probing

- Given **auxiliary hash function** $h' \colon U \to \{0,\ldots,m-1\}$
- The method of linear probing uses h' to define the hash function $h \colon U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$ as

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

- The initial probe determines the entire sequence, hence we have only m distinct probe sequences (Note: $m \ll m!$)
- Easy to implement
- Suffers from a problem known as **primary clustering**: long runs of occupied slots build up, increasing the average search time.

Quadratic Probing

- Given **auxiliary hash function** $h' \colon U \to \{0,\ldots,m-1\}$
- The method of quadratic probing uses h' to define the hash function $h \colon U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$ as

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

- The initial probe determines the entire sequence, hence we have only m distinct probe sequences (Note: $m \ll m!$)
- Better than linear probing, but c_1, c_2, and m have to be chosen wisely
- Suffers from a milder form of clustering, called **secondary clustering.**

Double Hashing

- Given **auxiliary hash functions** h_1 and h_2
- The method of double hashing defines the hash function $h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$ as

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m$$

- Unlike the previous methods, the probe sequence is not entirely determined by the first probe.
- For suitable h_1 and h_2, the number of possible probe sequences is $\Theta(m^2)$ since each pair $(h_1(k), h_2(k))$ can yield a distinct one.
- The performance gets very close to the “ideal” scheme of uniform hashing

Learned Today

- Direct-address tables
  - Dictionary operations take $\Theta(1)$ time
  - Does not scale with increasing number of keys
- Hash-tables
  - Use hash function to map keys to slots
  - Collisions resolved by chaining elements in (doubly) linked lists
    - Dictionary operations take $\Theta(1)$ time, on average assuming simple uniform hashing
- Hash functions
  - Should be designed to be independent from patterns in the set of keys
  - Heuristics: division method & multiplication method
- Open addressing
  - Insertion and search take $\Theta(1)$ time
  - Collisions resolved by probing
  - Hashing Heuristics: linear probing, quadratic probing, double hashing