Elementary Data Structures and Hash Tables

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

March 28, 2024

Outline

- Common concepts and notation
- Stacks
- Queues
- Linked lists
- Trees
- Direct-access tables
- Hash tables

Concepts

- A ***data structure*** is a way to organize and store information
    - to facilitate access, or for other purposes

Concepts

- A ***data structure*** is a way to organize and store information
  - to facilitate access, or for other purposes

- A data structure has an ***interface*** consisting of procedures for adding, deleting, accessing, reorganizing, etc.

Concepts

- A ***data structure*** is a way to organize and store information
  - to facilitate access, or for other purposes
- A data structure has an ***interface*** consisting of procedures for adding, deleting, accessing, reorganizing, etc.
- A data structure stores ***data*** and possibly ***meta-data***

Concepts

- A ***data structure*** is a way to organize and store information
  - to facilitate access, or for other purposes

- A data structure has an ***interface*** consisting of procedures for adding, deleting, accessing, reorganizing, etc.

- A data structure stores ***data*** and possibly ***meta-data***
  - e.g., a *heap* needs an array A to store the keys, plus a variable $A.\mathit{heap\text{-}size}$ to remember how many elements are in the heap

Stack

- The ubiquitous “last-in first-out” container (LIFO)

Stack

- The ubiquitous “last-in first-out” container (LIFO)

- *Interface*
  - **STACK-EMPTY**(S) returns TRUE if and only if S is empty
  - **PUSH**(S, x) pushes the value x onto the stack S
  - **POP**(S) extracts and returns the value on the top of the stack S

Stack

- The ubiquitous “last-in first-out” container (LIFO)

- *Interface*
  - **Stack-Empty**(S) returns TRUE if and only if S is empty
  - **Push**(S, x) pushes the value x onto the stack S
  - **Pop**(S) extracts and returns the value on the top of the stack S

- *Implementation*
  - using an array
  - using a linked list
  - …

A Stack Implementation

A Stack Implementation

- *Array-based implementation*

A Stack Implementation

- *Array-based implementation*
  - S is an array that holds the elements of the stack
  - $S.top$ is the current position of the top element of S

A Stack Implementation

- *Array-based implementation*
    - S is an array that holds the elements of the stack
    - $S.top$ is the current position of the top element of S

```
STACK-EMPTY(S)
if $S.top == 0$
    return TRUE
else return FALSE
```

A Stack Implementation

- *Array-based implementation*
  - *S* is an array that holds the elements of the stack
  - *S.top* is the current position of the top element of *S*

```
STACK-EMPTY(S)
if $S.top == 0$
        return TRUE
else return FALSE
```

```
PUSH(S,x)
$S.top = S.top + 1$
$S[S.top] = x$
```

```
POP(S)
if STACK-EMPTY($S$)
        error "underflow"
else $S.top = S.top - 1$
        return $S[S.top + 1]$
```

Queue

- The ubiquitous “first-in first-out” container (FIFO)

Queue

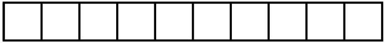
- The ubiquitous “first-in first-out” container (FIFO)
- *Interface*
  - **Enqueue**(Q, x) adds element *x* at the back of queue *Q*
  - **Dequeue**(Q) extracts the element at the head of queue *Q*

Queue

- The ubiquitous “first-in first-out” container (FIFO)
- *Interface*
  - **ENQUEUE**(Q, x) adds element x at the back of queue Q
  - **DEQUEUE**(Q) extracts the element at the head of queue Q
- *Implementation*
  - Q is an array of fixed length $Q.length$
    - i.e., Q holds at most $Q.length$ elements
    - enqueueing more than Q elements causes an “overflow” error
  - $Q.head$ is the position of the “head” of the queue
  - $Q.tail$ is the first empty position at the tail of the queue

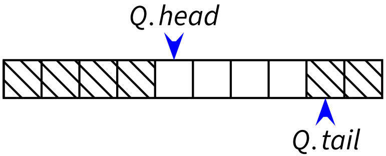
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
    error "overflow"
else $Q[Q.tail] = x$
    if $Q.tail < Q.length$
        $Q.tail = Q.tail + 1$
    else $Q.tail = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}full = $ TRUE
    $Q.queue\text{-}empty = $ FALSE
```



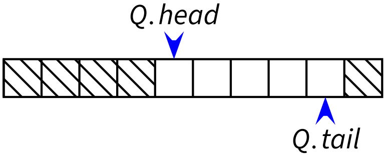
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
        error "overflow"
else $Q[Q.tail] = x$
        if $Q.tail < Q.length$
                $Q.tail = Q.tail + 1$
        else $Q.tail = 1$
        if $Q.tail == Q.head$
                $Q.queue\text{-}full =$ TRUE
        $Q.queue\text{-}empty =$ FALSE
```



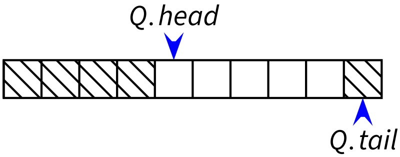
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
        error "overflow"
else $Q[Q.tail] = x$
        if $Q.tail < Q.length$
                $Q.tail = Q.tail + 1$
        else $Q.tail = 1$
        if $Q.tail == Q.head$
                $Q.queue\text{-}full = \text{TRUE}$
        $Q.queue\text{-}empty = \text{FALSE}$
```



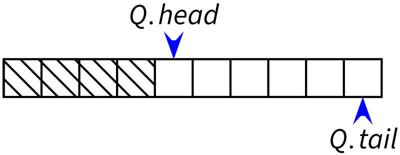
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
    error "overflow"
else $Q[Q.tail] = x$
    if $Q.tail < Q.length$
        $Q.tail = Q.tail + 1$
    else $Q.tail = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}full = \text{TRUE}$
    $Q.queue\text{-}empty = \text{FALSE}$
```



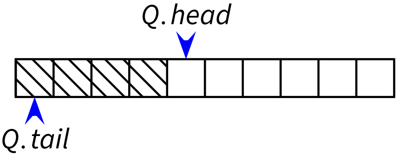
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
        error "overflow"
else $Q[Q.tail] = x$
        if $Q.tail < Q.length$
                $Q.tail = Q.tail + 1$
        else $Q.tail = 1$
        if $Q.tail == Q.head$
                $Q.queue\text{-}full = \text{TRUE}$
        $Q.queue\text{-}empty = \text{FALSE}$
```



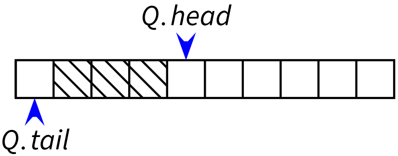
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
    error "overflow"
else $Q[Q.tail] = x$
    if $Q.tail < Q.length$
        $Q.tail = Q.tail + 1$
    else $Q.tail = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}full = $ TRUE
    $Q.queue\text{-}empty = $ FALSE
```



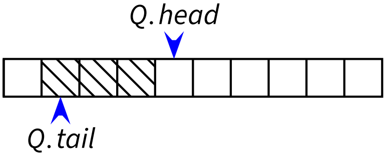
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
        error "overflow"
else $Q[Q.tail] = x$
        if $Q.tail < Q.length$
                $Q.tail = Q.tail + 1$
        else $Q.tail = 1$
        if $Q.tail == Q.head$
                $Q.queue\text{-}full = \text{TRUE}$
        $Q.queue\text{-}empty = \text{FALSE}$
```



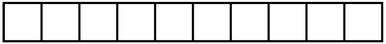
Enqueue

```
ENQUEUE(Q,x)
if $Q.queue\text{-}full$
        error "overflow"
else $Q[Q.tail] = x$
        if $Q.tail < Q.length$
                $Q.tail = Q.tail + 1$
        else $Q.tail = 1$
        if $Q.tail == Q.head$
                $Q.queue\text{-}full =$ TRUE
        $Q.queue\text{-}empty =$ FALSE
```



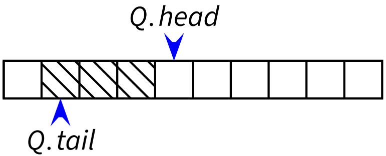
Dequeue

```
DEQUEUE(Q)
if $Q.queue\text{-}empty$
        error "underflow"
else $x = Q[Q.head]$
        if $Q.head < Q.length$
                $Q.head = Q.head + 1$
        else $Q.head = 1$
        if $Q.tail == Q.head$
                $Q.queue\text{-}empty$ = TRUE
        $Q.queue\text{-}full$ = FALSE
        return $x$
```



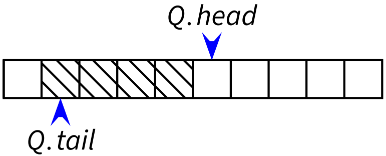
Dequeue

```
DEQUEUE(Q)
if $Q.queue\text{-}empty$
    error "underflow"
else $x = Q[Q.head]$
    if $Q.head < Q.length$
        $Q.head = Q.head + 1$
    else $Q.head = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}empty = \text{TRUE}$
    $Q.queue\text{-}full = \text{FALSE}$
    return $x$
```



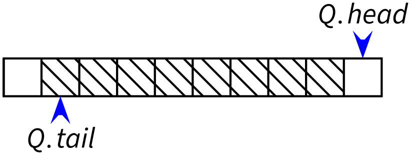
Dequeue

```
DEQUEUE(Q)
if $Q.queue\text{-}empty$
    error "underflow"
else $x = Q[Q.head]$
    if $Q.head < Q.length$
        $Q.head = Q.head + 1$
    else $Q.head = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}empty =$ TRUE
    $Q.queue\text{-}full =$ FALSE
    return $x$
```



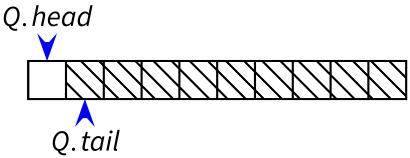
Dequeue

```
DEQUEUE(Q)
if $Q.queue\text{-}empty$
    error "underflow"
else $x = Q[Q.head]$
    if $Q.head < Q.length$
        $Q.head = Q.head + 1$
    else $Q.head = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}empty = $ TRUE
    $Q.queue\text{-}full = $ FALSE
    return $x$
```



Dequeue

```
DEQUEUE(Q)
if $Q.queue\text{-}empty$
    error "underflow"
else $x = Q[Q.head]$
    if $Q.head < Q.length$
        $Q.head = Q.head + 1$
    else $Q.head = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}empty =$ TRUE
    $Q.queue\text{-}full =$ FALSE
    return $x$
```



Linked List

- *Interface*
  - **LIST-INSERT**(L, x) adds element x at beginning of a list L
  - **LIST-DELETE**(L, x) removes element x from a list L
  - **LIST-SEARCH**(L, k) finds an element whose key is k in a list L

Linked List

- *Interface*
  - **LIST-INSERT**(L, x) adds element x at beginning of a list L
  - **LIST-DELETE**(L, x) removes element x from a list L
  - **LIST-SEARCH**(L, k) finds an element whose key is k in a list L

- *Implementation*
  - a *doubly-linked* list
  - each element x has two “links” $x.prev$ and $x.next$ to the previous and next elements, respectively
  - each element x holds a key $x.key$
  - it is convenient to have a dummy “sentinel” element $L.nil$

Linked List With a "Sentinel"

```
List-Init($L$)
    $L.nil.prev = L.nil$
    $L.nil.next = L.nil$
```

```
List-Insert($L, x$)
    $x.next = L.nil.next$
    $L.nil.next.prev = x$
    $L.nil.next = x$
    $x.prev = L.nil$
```

```
List-Search($L, k$)
    $x = L.nil.next$
    while $x \neq L.nil \land x.key \neq k$
        $x = x.next$
    return $x$
```

Complexity

Complexity

Algorithm	*Complexity*

Complexity

Algorithm	*Complexity*
Stack-Empty	

Complexity

Algorithm	*Complexity*
Stack-Empty	$O(1)$
Push	

Complexity

Algorithm	*Complexity*
STACK-EMPTY	$O(1)$
PUSH	$O(1)$
POP	$O(1)$
ENQUEUE	$O(1)$
DEQUEUE	$O(1)$
LIST-INSERT	

Complexity

Algorithm	*Complexity*
Stack-Empty	$O(1)$
Push	$O(1)$
Pop	$O(1)$
Enqueue	$O(1)$
Dequeue	$O(1)$
List-Insert	$O(1)$
List-Delete	

Complexity

Algorithm	*Complexity*
Stack-Empty	$O(1)$
Push	$O(1)$
Pop	$O(1)$
Enqueue	$O(1)$
Dequeue	$O(1)$
List-Insert	$O(1)$
List-Delete	$O(1)$
List-Search	

Complexity

Algorithm	*Complexity*
Stack-Empty	$O(1)$
Push	$O(1)$
Pop	$O(1)$
Enqueue	$O(1)$
Dequeue	$O(1)$
List-Insert	$O(1)$
List-Delete	$O(1)$
List-Search	$\Theta(n)$

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a ***dynamic*** set

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a ***dynamic*** set

- *Interface* (generic interface)
  - **INSERT**(D, k) adds a key k to the dictionary D
  - **DELETE**(D, k) removes key k from D
  - **SEARCH**(D, k) tells whether D contains a key k

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
    - a ***dynamic*** set

- *Interface* (generic interface)
    - **INSERT**(D, k) adds a key k to the dictionary D
    - **DELETE**(D, k) removes key k from D
    - **SEARCH**(D, k) tells whether D contains a key k

- *Implementation*
    - many (concrete) data structures

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a ***dynamic*** set
- *Interface* (generic interface)
  - **INSERT**(D, k) adds a key k to the dictionary D
  - **DELETE**(D, k) removes key k from D
  - **SEARCH**(D, k) tells whether D contains a key k
- *Implementation*
  - many (concrete) data structures
  - ***hash tables***

Direct-Address Table

- A *direct-address table* implements a dictionary

Direct-Address Table

- A *direct-address table* implements a dictionary
- The *universe* of keys is $U = \{1, 2, \ldots, M\}$

Direct-Address Table

- A *direct-address table* implements a dictionary

- The *universe* of keys is $U = \{1, 2, \ldots, M\}$

- *Implementation*
  - an array T of size M
  - each key has its own position in T

Direct-Address Table

- A *direct-address table* implements a dictionary
- The *universe* of keys is $U = \{1, 2, \ldots, M\}$
- *Implementation*
  - an array T of size M
  - each key has its own position in T

```
DIRECT-ADDRESS-INSERT($T, k$)
$T[k] = $ TRUE
```

```
DIRECT-ADDRESS-DELETE($T, k$)
$T[k] = $ FALSE
```

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

Direct-Address Table (2)

- Complexity

Direct-Address Table (2)

- Complexity

All direct-address table operations are $O(1)$*!*

Direct-Address Table (2)

- Complexity

All direct-address table operations are $O(1)$*!*

So why isn't every set implemented with a direct-address table?

Direct-Address Table (2)

- Complexity

  All direct-address table operations are $O(1)$*!*

  So why isn't every set implemented with a direct-address table?

- The ***space complexity*** is $\Theta(|U|)$
  - $|U|$ is typically a very large number—U is the *universe* of keys!
  - the represented set is typically *much smaller* than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

Direct-Address Table (2)

- Complexity

  All direct-address table operations are $O(1)$*!*

  So why isn't every set implemented with a direct-address table?

- The ***space complexity*** is $\Theta(|U|)$
  - $|U|$ is typically a very large number—U is the *universe* of keys!
  - the represented set is typically *much smaller* than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

- *Can we have the benefits of a direct-address table but with a table of reasonable size?*

Hash Table

- *Idea*
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a ***hash function***

$$h : U \longrightarrow \{1, \ldots, |T|\}$$

Hash Table

- *Idea*
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a ***hash function***

$$h : U \longrightarrow \{1, \ldots, |T|\}$$

```
HASH-INSERT$(T, k)$
    $T[h(k)] =$ TRUE
```

```
HASH-DELETE$(T, k)$
    $T[h(k)] =$ FALSE
```

```
HASH-SEARCH$(T, k)$
    return $T[h(k)]$
```

Hash Table

- *Idea*
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a ***hash function***

$$h : U \longrightarrow \{1, \ldots, |T|\}$$

```
HASH-INSERT$(T, k)$
    $T[h(k)] = \text{TRUE}$
```

```
HASH-DELETE$(T, k)$
    $T[h(k)] = \text{FALSE}$
```

```
HASH-SEARCH$(T, k)$
    return $T[h(k)]$
```

Are these algorithms correct?

Hash Table

- *Idea*
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a ***hash function***

$$h : U \longrightarrow \{1, \ldots, |T|\}$$

```
HASH-INSERT$(T, k)$
    $T[h(k)] = \text{TRUE}$
```

```
HASH-DELETE$(T, k)$
    $T[h(k)] = \text{FALSE}$
```

```
HASH-SEARCH$(T, k)$
    return $T[h(k)]$
```

Are these algorithms correct? No!

Hash Table

- *Idea*
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a ***hash function***

$$h : U \longrightarrow \{1, \dots, |T|\}$$

```
HASH-INSERT(T, k)
    $T[h(k)] = \text{TRUE}$
```

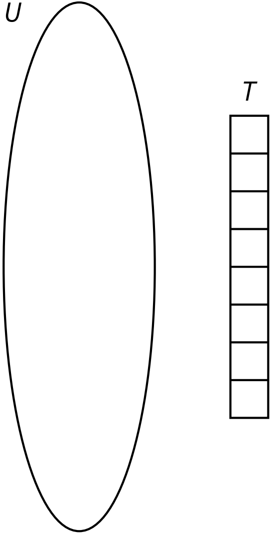
```
HASH-DELETE(T, k)
    $T[h(k)] = \text{FALSE}$
```

```
HASH-SEARCH(T, k)
    return $T[h(k)]$
```

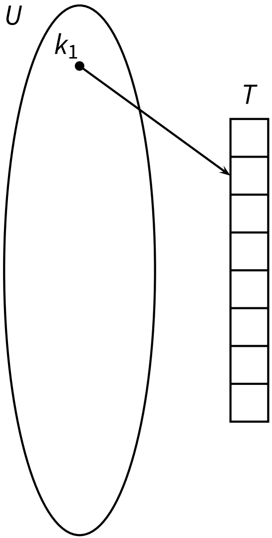
Are these algorithms correct? No!

What if two distinct keys $k_1 \neq k_2$ *collide?* (I.e., $h(k_1) = h(k_2)$)

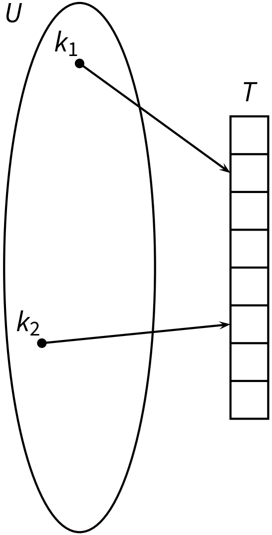
Hash Table



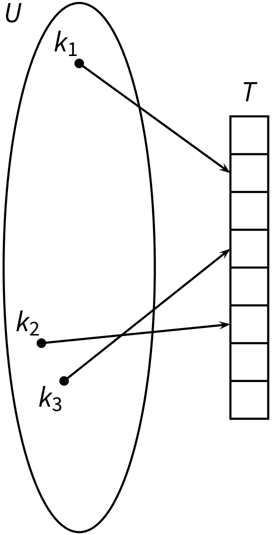
Hash Table



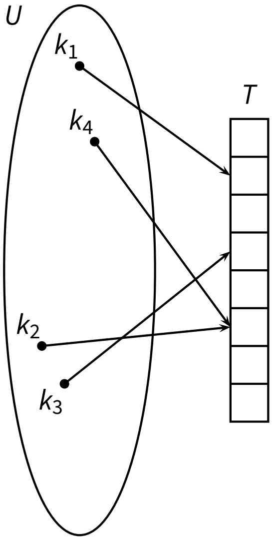
Hash Table



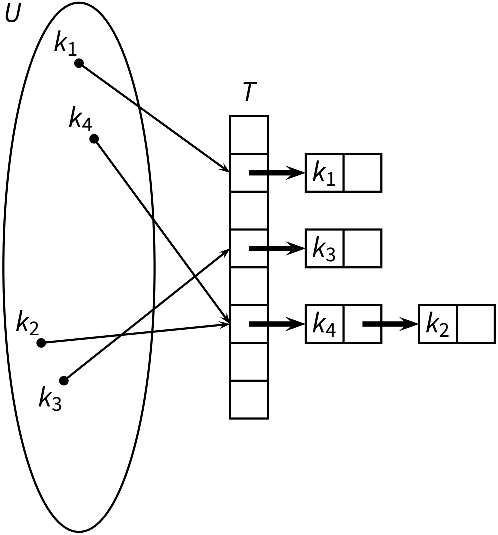
Hash Table



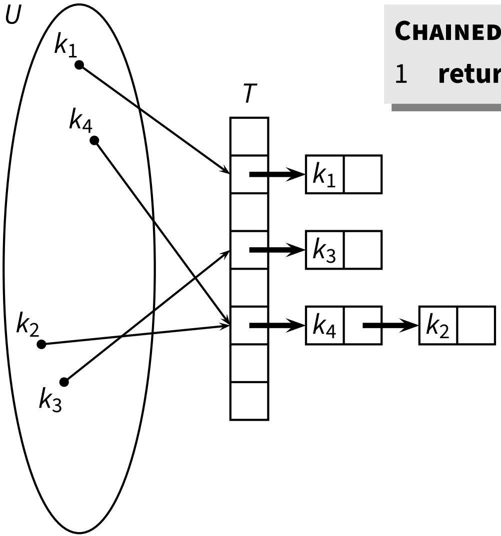
Hash Table



Hash Table

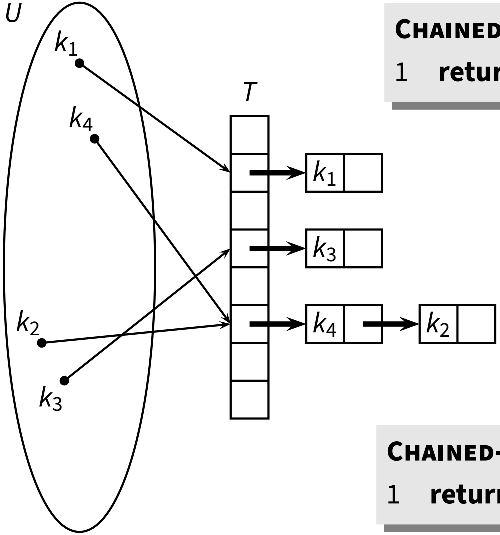


Hash Table



```
CHAINED-HASH-INSERT$(T, k)$
    return LIST-INSERT$(T[h(k)], k)$
```

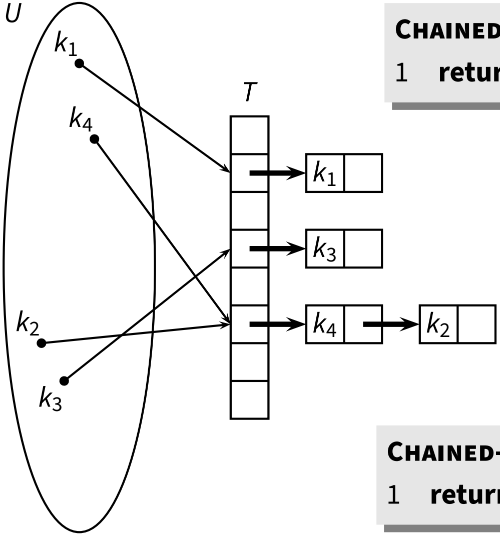
Hash Table



```
CHAINED-HASH-INSERT$(T, k)$
    return LIST-INSERT$(T[h(k)], k)$
```

```
CHAINED-HASH-SEARCH$(T, k)$
    return LIST-SEARCH$(T[h(k)], k)$
```

Hash Table



```
CHAINED-HASH-INSERT$(T, k)$
    return LIST-INSERT$(T[h(k)], k)$
```

load factor

$$\alpha = \frac{n}{|T|}$$

```
CHAINED-HASH-SEARCH$(T, k)$
    return LIST-SEARCH$(T[h(k)], k)$
```

Analysis

- We assume ***uniform hashing*** for our hash function $h : U \longrightarrow \{1 \ldots |T|\}$ (where $|T| = T.length$)

Analysis

- We assume ***uniform hashing*** for our hash function $h : U \longrightarrow \{1 \ldots |T|\}$ (where $|T| = T.length$)

  $$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \ldots |T|\}$$

  (The formalism is actually a bit more complicated.)

Analysis

- We assume ***uniform hashing*** for our hash function $h : U \longrightarrow \{1 \ldots |T|\}$ (where $|T| = T.length$)

  $$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \ldots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

  $$E[n_i] = \frac{n}{|T|} = \alpha$$

Analysis

- We assume ***uniform hashing*** for our hash function $h : U \longrightarrow \{1 \ldots |T|\}$ (where $|T| = T.\mathit{length}$)

  $$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \ldots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

  $$\mathrm{E}[n_i] = \frac{n}{|T|} = \alpha$$

- We further assume that $h(k)$ can be computed in $O(1)$ time

Analysis

- We assume ***uniform hashing*** for our hash function $h : U \rightarrow \{1 \dots |T|\}$ (where $|T| = T.length$)

  $$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

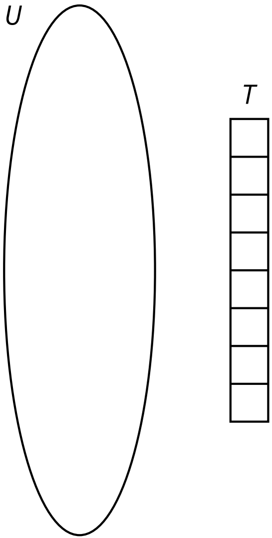
  $$E[n_i] = \frac{n}{|T|} = \alpha$$

- We further assume that $h(k)$ can be computed in $O(1)$ time

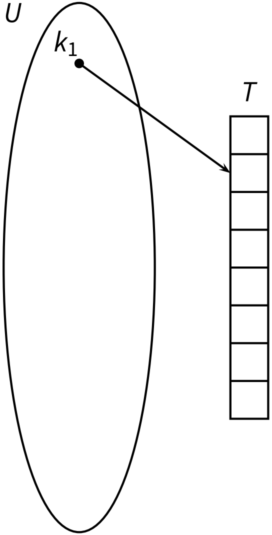
- Therefore, the complexity of **CHAINED-HASH-SEARCH** is

  $$\boxed{\Theta(1 + \alpha)}$$

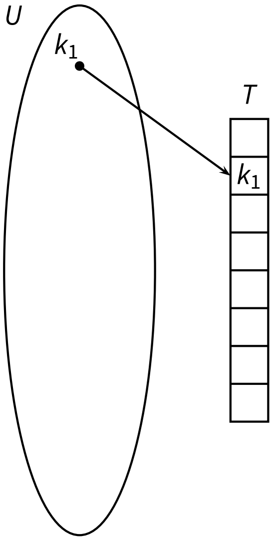
Open-Address Hash Table



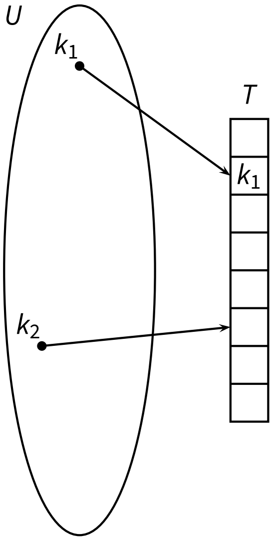
Open-Address Hash Table



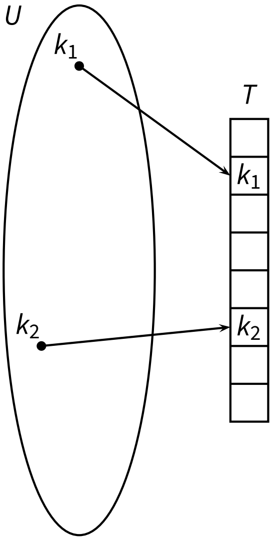
Open-Address Hash Table



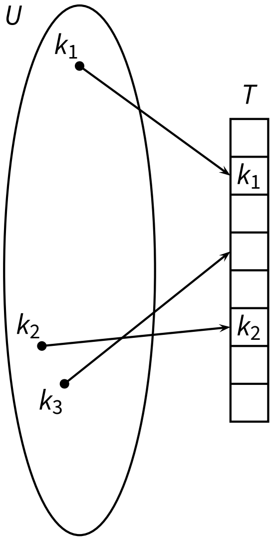
Open-Address Hash Table



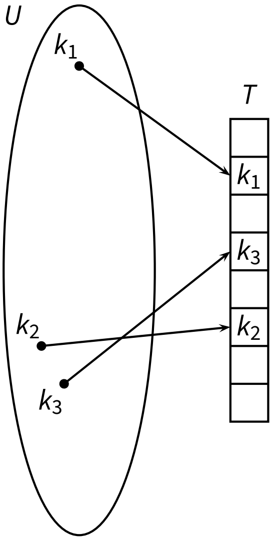
Open-Address Hash Table



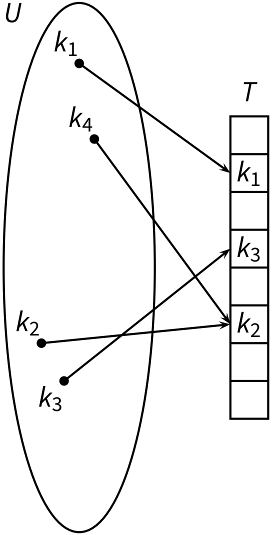
Open-Address Hash Table



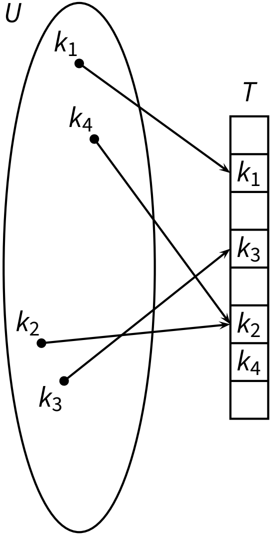
Open-Address Hash Table



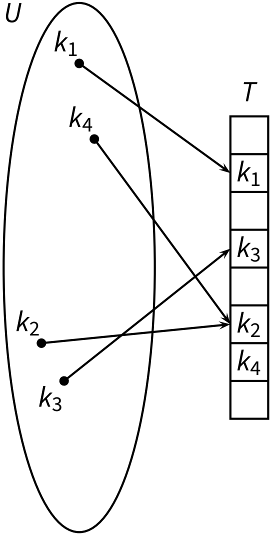
Open-Address Hash Table



Open-Address Hash Table



Open-Address Hash Table



```
HASH-INSERT(T, k)
j = h(k)
for i = 1 to T.length
        if T[j] == NIL
                T[j] = k
                return j
        elseif j < T.length
                j = j + 1
        else j = 1
error "overflow"
```

Open-Addressing (2)

- *Idea:* instead of using linked lists, we can store all the elements in the table
  - this implies $\alpha \leq 1$

Open-Addressing (2)

- *Idea:* instead of using linked lists, we can store all the elements in the table
  - this implies $\alpha \leq 1$

- When a collision occurs, we simply find another free cell in T

Open-Addressing (2)

- *Idea:* instead of using linked lists, we can store all the elements in the table
    - this implies $\alpha \leq 1$

- When a collision occurs, we simply find another free cell in T

- A sequential “probe” may not be optimal
    - can you figure out why?

Open-Addressing (3)

```
HASH-INSERT$(T, k)$
for $i = 1$ to $T.length$
$j = h(k, i)$
    if $T[j]$ == NIL
        $T[j] = k$
        return $j$
error "overflow"
```

Open-Addressing (3)

```
HASH-INSERT($T, k$)
for $i = 1$ to $T.length$
$j = h(k, i)$
        if $T[j]$ == NIL
                $T[j] = k$
                return $j$
error “overflow”
```

- Notice that $h(k, \cdot)$ must be a ***permutation***
  - i.e., $h(k, 1), h(k, 2), \ldots, h(k, |T|)$ must cover the entire table T