

Elementary Data Structures and Hash Tables

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

March 20, 2025

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

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

- 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*** 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***

- 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 $A.heap-size$ to remember how many elements are in the heap

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

- 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

- 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

- *Array-based 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

■ *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)

```
1  if  $S.top == 0$   
2      return TRUE  
3  else return FALSE
```

■ *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)

```
1  if  $S.top == 0$ 
2      return TRUE
3  else return FALSE
```

PUSH(S,x)

```
1   $S.top = S.top + 1$ 
2   $S[S.top] = x$ 
```

POP(S)

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

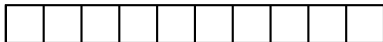
- The ubiquitous “first-in first-out” container (FIFO)

- 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

- 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(Q,x)

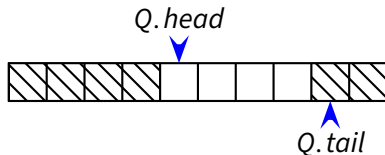
```
1  if Q.queue-full
2      error "overflow"
3  else Q[Q.tail] = x
4      if Q.tail < Q.length
5          Q.tail = Q.tail + 1
6      else Q.tail = 1
7      if Q.tail == Q.head
8          Q.queue-full = TRUE
9      Q.queue-empty = FALSE
```



ENQUEUE(Q,x)

```

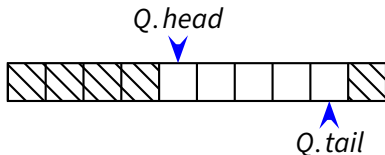
1  if Q.queue-full
2      error "overflow"
3  else  $Q[Q.tail] = x$ 
4      if  $Q.tail < Q.length$ 
5           $Q.tail = Q.tail + 1$ 
6      else  $Q.tail = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-full = TRUE$ 
9       $Q.queue-empty = FALSE$ 
    
```



ENQUEUE(Q,x)

```

1  if Q.queue-full
2      error "overflow"
3  else Q[Q.tail] = x
4      if Q.tail < Q.length
5          Q.tail = Q.tail + 1
6      else Q.tail = 1
7      if Q.tail == Q.head
8          Q.queue-full = TRUE
9      Q.queue-empty = FALSE
    
```

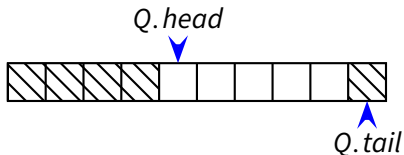


ENQUEUE(Q,x)

```

1  if Q.queue-full
2      error "overflow"
3  else  $Q[Q.tail] = x$ 
4      if  $Q.tail < Q.length$ 
5           $Q.tail = Q.tail + 1$ 
6      else  $Q.tail = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-full = \text{TRUE}$ 
9           $Q.queue-empty = \text{FALSE}$ 

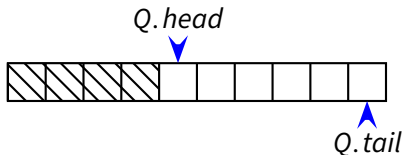
```



ENQUEUE(Q,x)

```

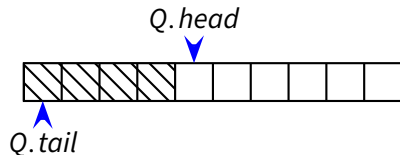
1  if Q.queue-full
2      error "overflow"
3  else Q[Q.tail] = x
4      if Q.tail < Q.length
5          Q.tail = Q.tail + 1
6      else Q.tail = 1
7      if Q.tail == Q.head
8          Q.queue-full = TRUE
9      Q.queue-empty = FALSE
    
```



ENQUEUE(Q,x)

```

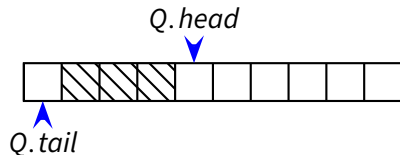
1  if Q.queue-full
2      error "overflow"
3  else  $Q[Q.tail] = x$ 
4      if  $Q.tail < Q.length$ 
5           $Q.tail = Q.tail + 1$ 
6      else  $Q.tail = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-full = TRUE$ 
9       $Q.queue-empty = FALSE$ 
    
```



ENQUEUE(Q,x)

```

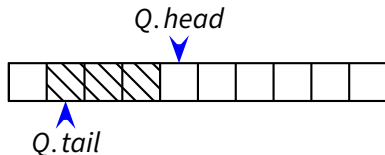
1  if Q.queue-full
2      error "overflow"
3  else Q[Q.tail] = x
4      if Q.tail < Q.length
5          Q.tail = Q.tail + 1
6      else Q.tail = 1
7      if Q.tail == Q.head
8          Q.queue-full = TRUE
9          Q.queue-empty = FALSE
    
```



ENQUEUE(Q,x)

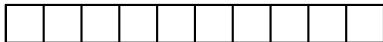
```

1  if Q.queue-full
2      error "overflow"
3  else  $Q[Q.tail] = x$ 
4      if  $Q.tail < Q.length$ 
5           $Q.tail = Q.tail + 1$ 
6      else  $Q.tail = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-full = \text{TRUE}$ 
9       $Q.queue-empty = \text{FALSE}$ 
    
```



DEQUEUE(Q)

```
1  if Q.queue-empty
2      error "underflow"
3  else  $x = Q[Q.head]$ 
4      if  $Q.head < Q.length$ 
5           $Q.head = Q.head + 1$ 
6      else  $Q.head = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-empty = TRUE$ 
9           $Q.queue-full = FALSE$ 
10     return  $x$ 
```

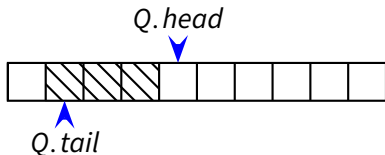


DEQUEUE(Q)

```

1  if Q.queue-empty
2      error "underflow"
3  else  $x = Q[Q.head]$ 
4      if  $Q.head < Q.length$ 
5           $Q.head = Q.head + 1$ 
6      else  $Q.head = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-empty = TRUE$ 
9           $Q.queue-full = FALSE$ 
10     return  $x$ 

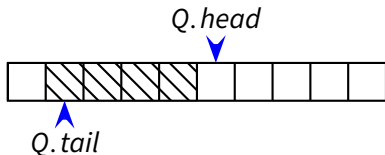
```



DEQUEUE(Q)

```

1  if Q.queue-empty
2      error "underflow"
3  else  $x = Q[Q.head]$ 
4      if  $Q.head < Q.length$ 
5           $Q.head = Q.head + 1$ 
6      else  $Q.head = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-empty = TRUE$ 
9           $Q.queue-full = FALSE$ 
10     return  $x$ 
    
```

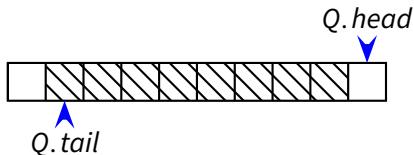


DEQUEUE(Q)

```

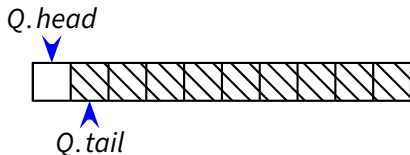
1  if Q.queue-empty
2      error "underflow"
3  else  $x = Q[Q.head]$ 
4      if  $Q.head < Q.length$ 
5           $Q.head = Q.head + 1$ 
6      else  $Q.head = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-empty = \text{TRUE}$ 
9           $Q.queue-full = \text{FALSE}$ 
10     return  $x$ 

```



DEQUEUE(Q)

```
1  if Q.queue-empty
2      error "underflow"
3  else  $x = Q[Q.head]$ 
4      if  $Q.head < Q.length$ 
5           $Q.head = Q.head + 1$ 
6      else  $Q.head = 1$ 
7      if  $Q.tail == Q.head$ 
8           $Q.queue-empty = TRUE$ 
9           $Q.queue-full = FALSE$ 
10     return  $x$ 
```



■ *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

■ *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)

- 1 $L.nil.prev = L.nil$
- 2 $L.nil.next = L.nil$

LIST-INSERT(L, x)

- 1 $x.next = L.nil.next$
- 2 $L.nil.next.prev = x$
- 3 $L.nil.next = x$
- 4 $x.prev = L.nil$

LIST-SEARCH(L, k)

- 1 $x = L.nil.next$
- 2 **while** $x \neq L.nil \wedge x.key \neq k$
- 3 $x = x.next$
- 4 **return** x

<i>Algorithm</i>	<i>Complexity</i>
------------------	-------------------

<i>Algorithm</i>	<i>Complexity</i>
------------------	-------------------

STACK-EMPTY	
--------------------	--

<i>Algorithm</i>	<i>Complexity</i>
------------------	-------------------

STACK-EMPTY	$O(1)$
--------------------	--------

PUSH	
-------------	--

<i>Algorithm</i>	<i>Complexity</i>
STACK-EMPTY	$O(1)$
PUSH	$O(1)$
POP	$O(1)$
ENQUEUE	$O(1)$
DEQUEUE	$O(1)$
LIST-INSERT	

<i>Algorithm</i>	<i>Complexity</i>
STACK-EMPTY	$O(1)$
PUSH	$O(1)$
POP	$O(1)$
ENQUEUE	$O(1)$
DEQUEUE	$O(1)$
LIST-INSERT	$O(1)$
LIST-DELETE	

<i>Algorithm</i>	<i>Complexity</i>
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	

<i>Algorithm</i>	<i>Complexity</i>
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	$\Theta(n)$

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

- 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

- 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

- 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**

- A *direct-address table* implements a dictionary

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

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

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

DIRECT-ADDRESS-INSERT(T, k)

```
1  $T[k] = \text{TRUE}$ 
```

DIRECT-ADDRESS-DELETE(T, k)

```
1  $T[k] = \text{FALSE}$ 
```

DIRECT-ADDRESS-SEARCH(T, k)

```
1 return  $T[k]$ 
```

- Complexity

- Complexity

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

- Complexity

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

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

- 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

- 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?*

■ *Idea*

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

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

■ Idea

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

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

HASH-INSERT(T, k)

1 $T[h(k)] = \text{TRUE}$

HASH-DELETE(T, k)

1 $T[h(k)] = \text{FALSE}$

HASH-SEARCH(T, k)

1 **return** $T[h(k)]$

■ Idea

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

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

HASH-INSERT(T, k)

1 $T[h(k)] = \text{TRUE}$

HASH-DELETE(T, k)

1 $T[h(k)] = \text{FALSE}$

HASH-SEARCH(T, k)

1 **return** $T[h(k)]$

Are these algorithms correct?

■ Idea

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

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

HASH-INSERT(T, k)

1 $T[h(k)] = \text{TRUE}$

HASH-DELETE(T, k)

1 $T[h(k)] = \text{FALSE}$

HASH-SEARCH(T, k)

1 **return** $T[h(k)]$

Are these algorithms correct? No!

■ Idea

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

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

HASH-INSERT(T, k)

1 $T[h(k)] = \text{TRUE}$

HASH-DELETE(T, k)

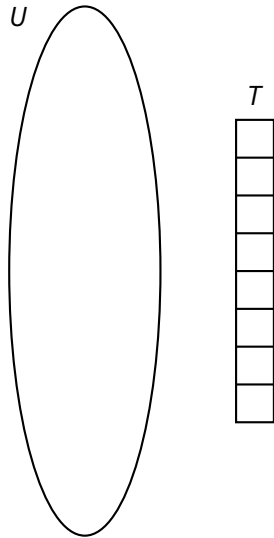
1 $T[h(k)] = \text{FALSE}$

HASH-SEARCH(T, k)

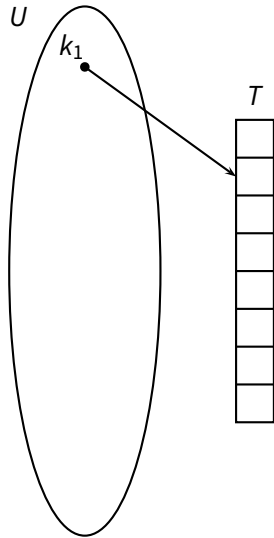
1 **return** $T[h(k)]$

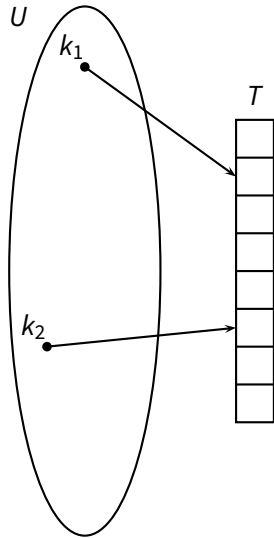
Are these algorithms correct? No!

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

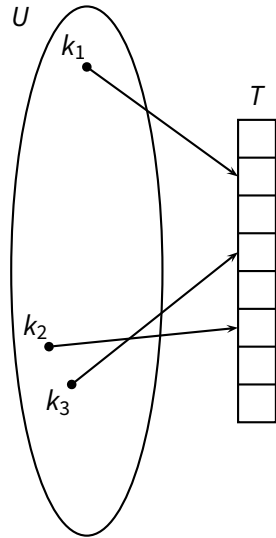


Hash Table

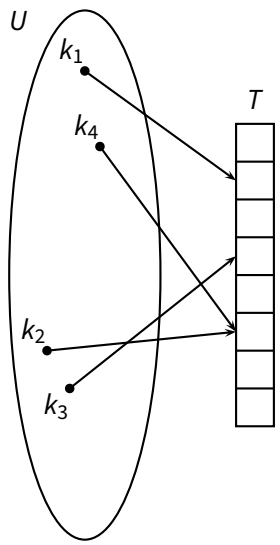


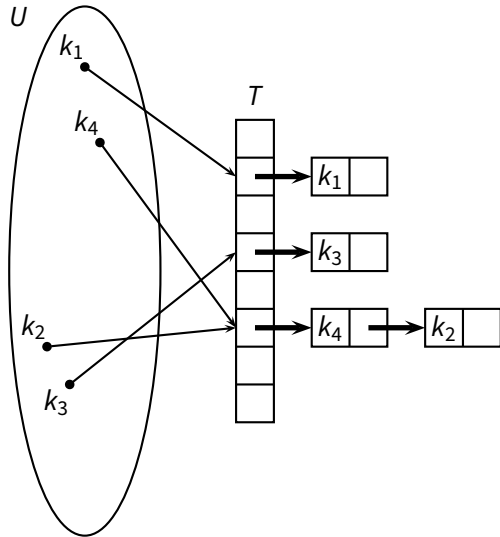


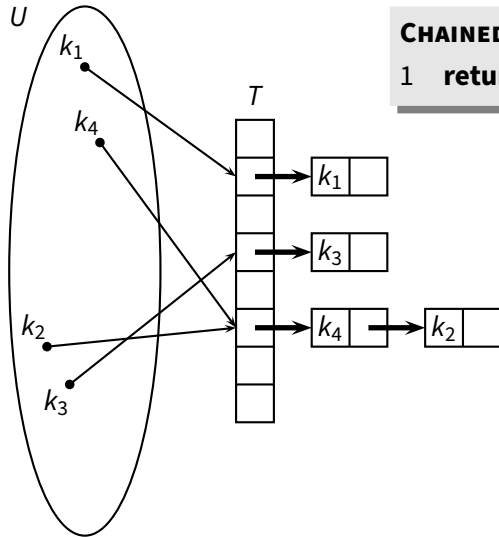
Hash Table



Hash Table

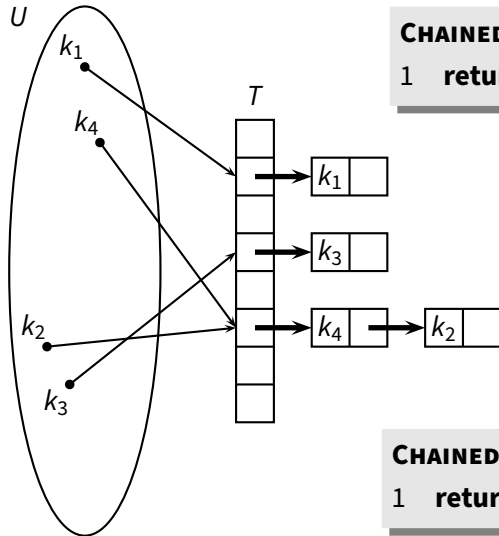






CHAINED-HASH-INSERT(T, k)

1 **return LIST-INSERT($T[h(k)], k$)**

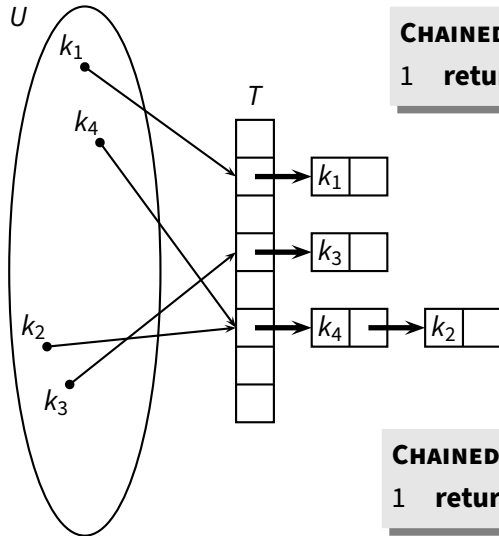


CHAINED-HASH-INSERT(T, k)

1 return **LIST-INSERT**($T[h(k)], k$)

CHAINED-HASH-SEARCH(T, k)

1 return **LIST-SEARCH**($T[h(k)], k$)



CHAINED-HASH-INSERT(T, k)

1 return **LIST-INSERT**($T[h(k)], k$)

load factor

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

CHAINED-HASH-SEARCH(T, k)

1 return **LIST-SEARCH**($T[h(k)], k$)

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

- 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.)

- 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 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

- 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.)

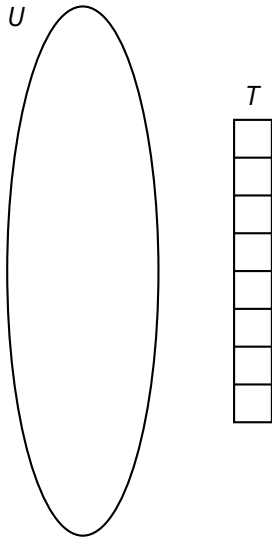
- So, given n distinct keys, the expected length n_i of the linked list at position i is

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

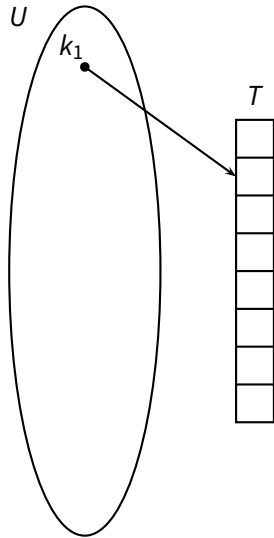
- We further assume that $h(k)$ can be computed in $O(1)$ time
- Therefore, the complexity of **CHAINED-HASH-SEARCH** is

$$\Theta(1 + \alpha)$$

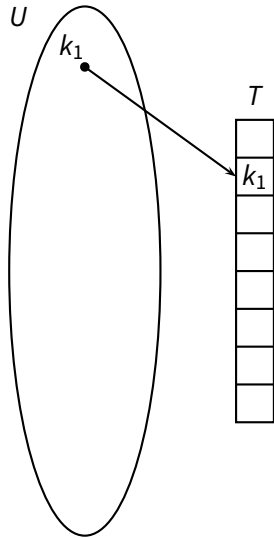
Open-Address Hash Table



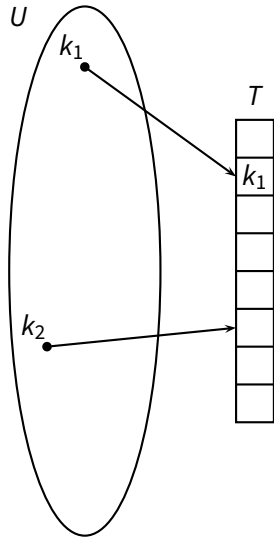
Open-Address Hash Table



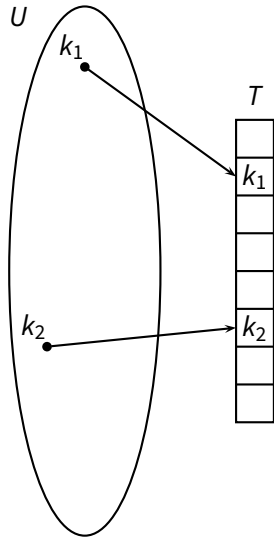
Open-Address Hash Table



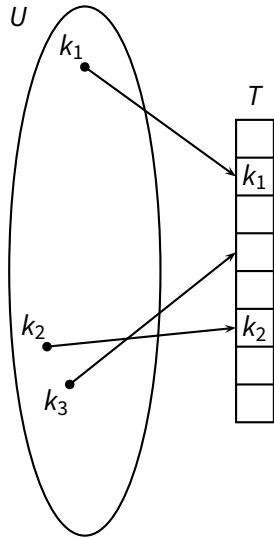
Open-Address Hash Table



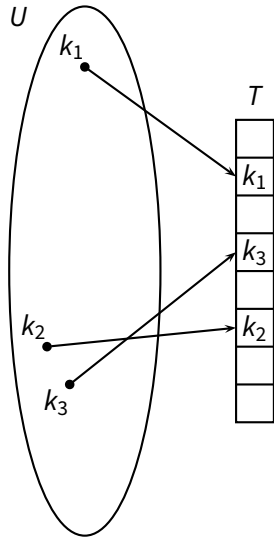
Open-Address Hash Table



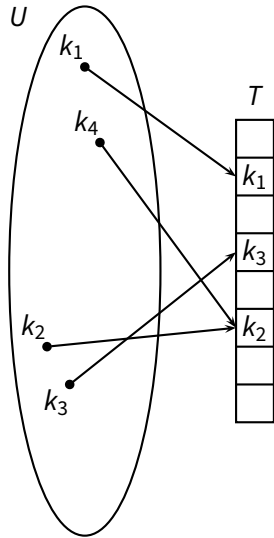
Open-Address Hash Table



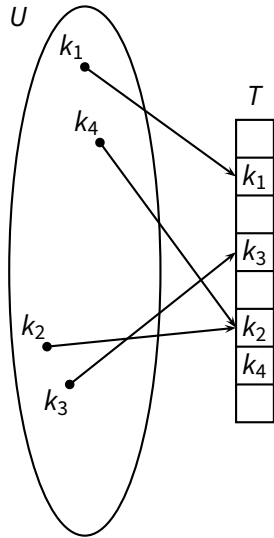
Open-Address Hash Table



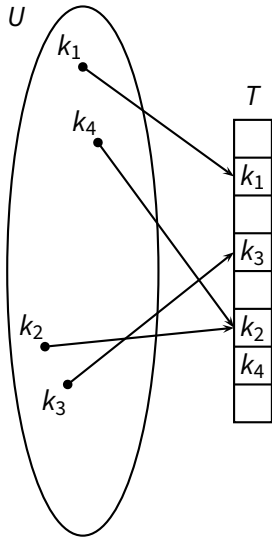
Open-Address Hash Table



Open-Address Hash Table



Open-Address Hash Table



HASH-INSERT(T, k)

```
1  $j = h(k)$ 
2 for  $i = 1$  to  $T.length$ 
3   if  $T[j] == \text{NIL}$ 
4      $T[j] = k$ 
5     return  $j$ 
6   elseif  $j < T.length$ 
7      $j = j + 1$ 
8   else  $j = 1$ 
9   error "overflow"
```

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

- *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?

HASH-INSERT(T, k)

```
1 for  $i = 1$  to  $T.length$ 
2    $j = h(k, i)$ 
3     if  $T[j] == \text{NIL}$ 
4        $T[j] = k$ 
5       return  $j$ 
6 error "overflow"
```

```
HASH-INSERT( $T, k$ )  
1 for  $i = 1$  to  $T.length$   
2    $j = h(k, i)$   
3     if  $T[j] == \text{NIL}$   
4        $T[j] = k$   
5     return  $j$   
6 error “overflow”
```

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