Elementary Data Structures and Hash Tables

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

April 20, 2021

Outline

- 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 $A.\mathit{heap\text{-}size}$ to remember how many elements are in the heap

- 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

- 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)

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

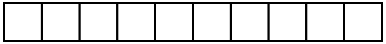
- *Interface*
  - **Enqueue**(Q, x) adds element x at the back of queue Q
  - **Dequeue**(Q) extracts the element at the head of queue Q

- 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

Enqueue(Q,x)

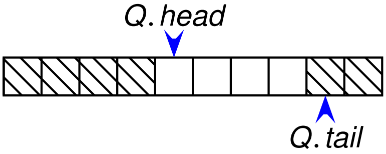
```
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = true
    Q.queue-empty = false
```



Enqueue

Enqueue(Q,x)

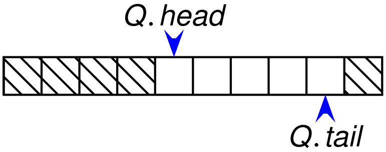
```
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = true
        Q.queue-empty = false
```



Enqueue

Enqueue(Q,x)

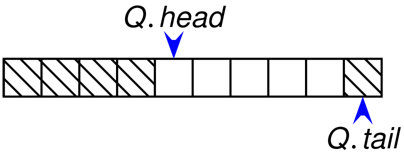
```
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = true
    Q.queue-empty = false
```



Enqueue

Enqueue(Q,x)

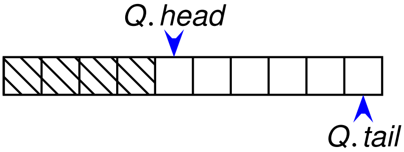
```
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = true
    Q.queue-empty = false
```



Enqueue

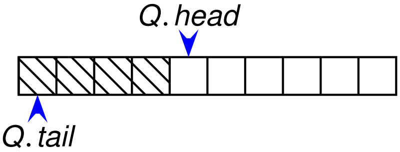
Enqueue(Q,x)

```
if Q.queue-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-full = true
        Q.queue-empty = false
```



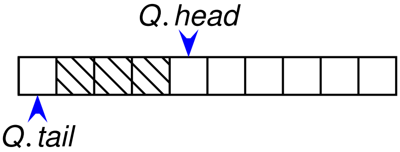
Enqueue

```
Enqueue(Q,x)
if Q.queue-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-full = true
        Q.queue-empty = false
```



Enqueue

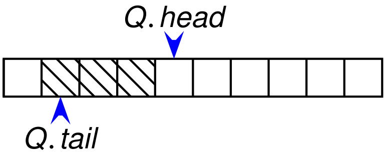
```
Enqueue(Q,x)
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = true
    Q.queue-empty = false
```



Enqueue

Enqueue(Q,x)

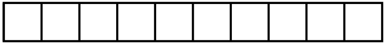
```
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = true
        Q.queue-empty = false
```



Dequeue

Dequeue(Q)

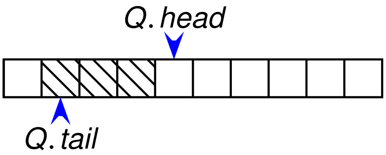
```
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = true
    Q.queue-full = false
    return x
```



Dequeue

Dequeue(Q)

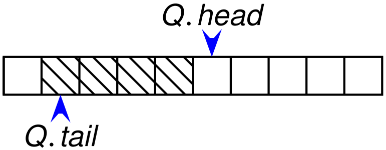
```
if Q.queue-empty
        error "underflow"
else x = Q[Q.head]
        if Q.head < Q.length
                Q.head = Q.head + 1
        else Q.head = 1
        if Q.tail == Q.head
                Q.queue-empty = true
        Q.queue-full = false
        return x
```



Dequeue

Dequeue(Q)

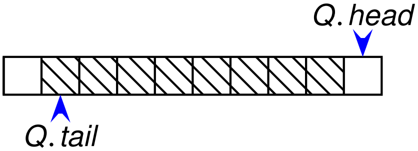
```
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = true
    Q.queue-full = false
    return x
```



Dequeue

Dequeue(Q)

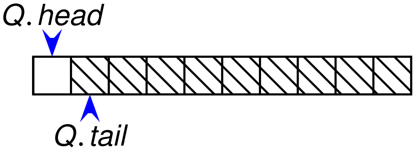
```
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = true
    Q.queue-full = false
    return x
```



Dequeue

Dequeue(Q)

```
if Q.queue-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-empty = true
    Q.queue-full = false
    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)

```
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 ≠ L.nil ∧ x.key ≠ k
      x = x.next
return x
```

- *Structure*
  - fixed branching
  - unbounded branching

- *Structure*
  - fixed branching
  - unbounded branching

- *Implementation*
  - for each node $x \neq T.root$, $x.parent$ is x's parent node
  - fixed branching:
    e.g., $x.left\text{-}child$ and $x.right\text{-}child$ in a *binary tree*
  - unbounded branching:
    $x.left\text{-}child$ is x's first (leftmost) child
    $x.right\text{-}sibling$ is x closest sibling to the right

Algorithm	*Complexity*

Algorithm	*Complexity*
Stack-Empty	

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

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

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

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	

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

- 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

- 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

- 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)
1 $T[k] = \text{true}$

Direct-Address-Delete(T, k)
1 $T[k] = \text{false}$

Direct-Address-Search(T, k)
1 **return** $T[k]$

Direct-Address Table (2)

- Complexity

Direct-Address Table (2)

- 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

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?*

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

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

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

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

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***

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

Hash-Insert(T, k)
1 $T[h(k)] =$ true

Hash-Delete(T, k)
1 $T[h(k)] =$ 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***

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

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***

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

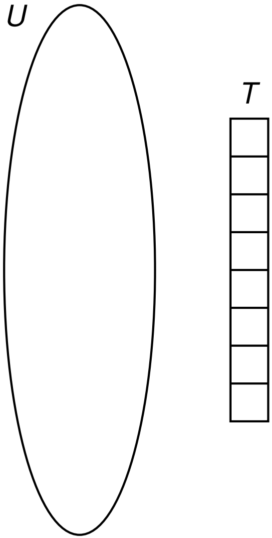
Hash-Insert(T, k)
1 $T[h(k)] = \text{true}$

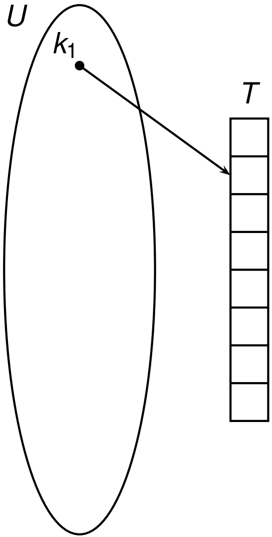
Hash-Delete(T, k)
1 $T[h(k)] = \text{false}$

Hash-Search(T, k)
1 **return** $T[h(k)]$

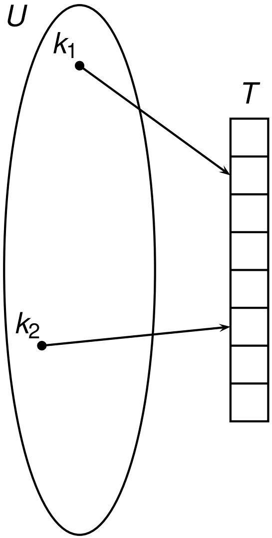
Are these algorithms correct? No!

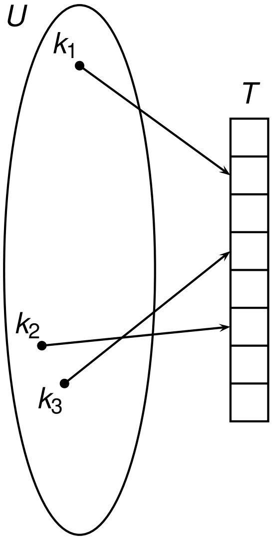
What if two distinct keys $k_1 \neq k_2$ *collide?* (I.e., $h(k_1) = h(k_2)$)


U
T

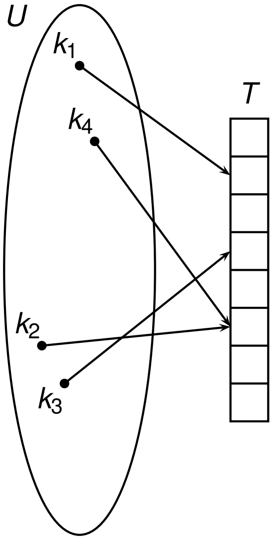


U
k1
T

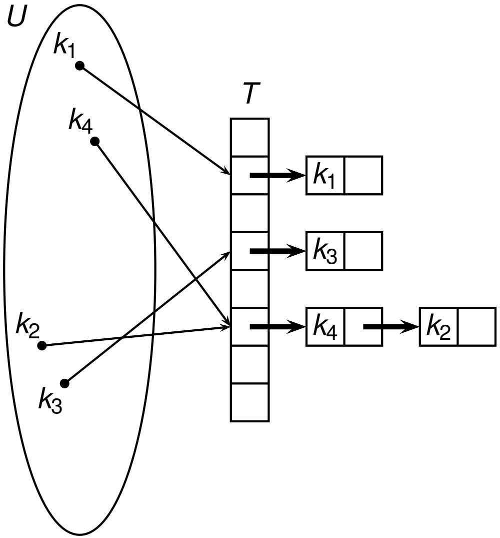

U
k_1
k_2
T

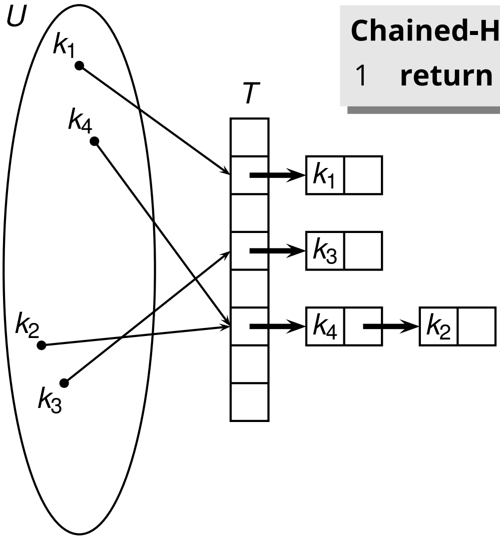


U
k_1
T
k_2
k_3


U
k_1
k_4
k_2
k_3
T

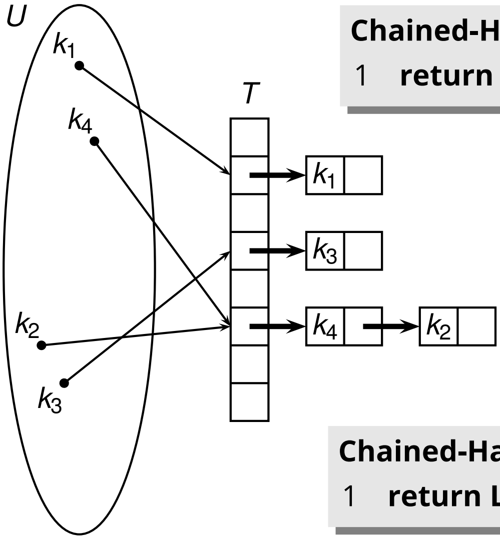
Hash Table





Chained-Hash-Insert(T, k)

1 **return List-Insert**$(T[h(k)], k)$

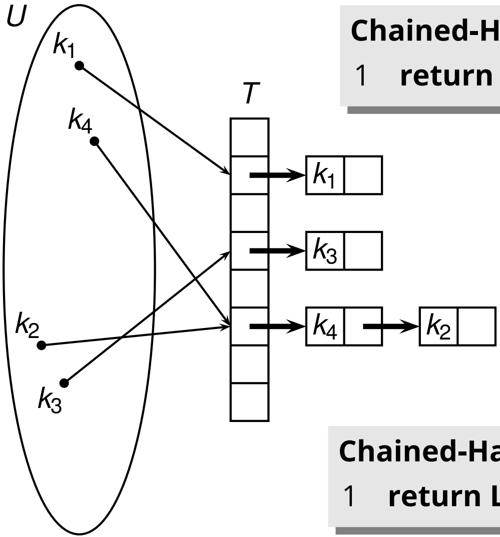


Chained-Hash-Insert(T, k)

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 $h : U \longrightarrow \{1 \ldots |T|\}$ (where $|T| = T.\mathit{length}$)

- 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.)

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

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

(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 $h : U \longrightarrow \{1 \dots |T|\}$ (where $|T| = T.\mathit{length}$)

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

  (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

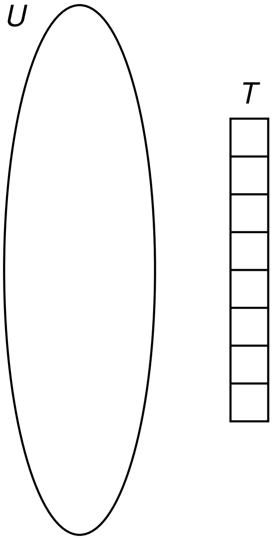
$$\mathrm{E}[n_i] = \frac{n}{|T|} = \alpha$$

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

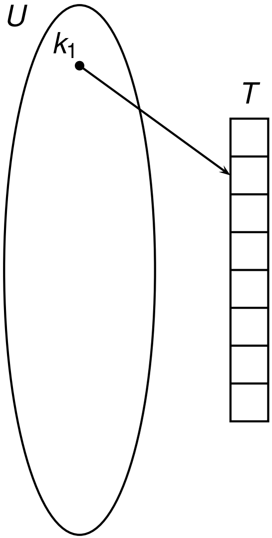
- Therefore, the complexity of **Chained-Hash-Search** is

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

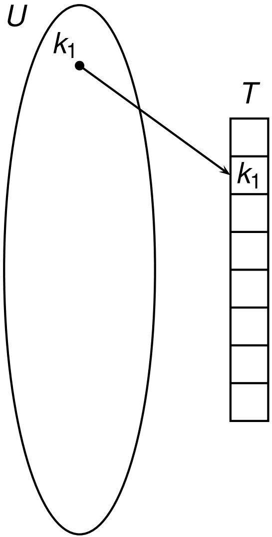
Open-Address Hash Table



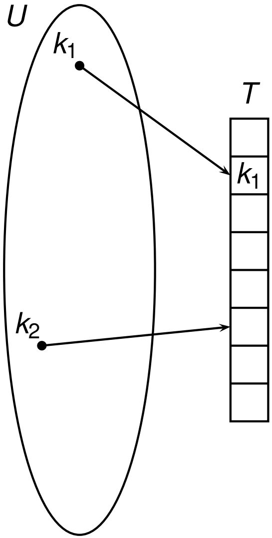
Open-Address Hash Table



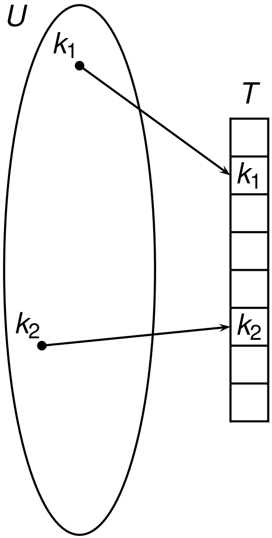
Open-Address Hash Table



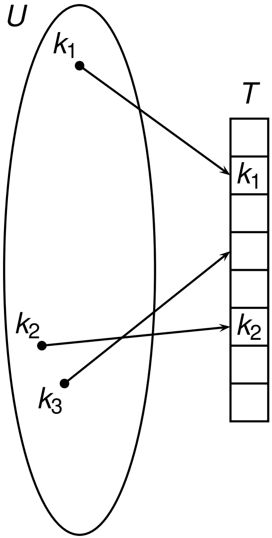
Open-Address Hash Table



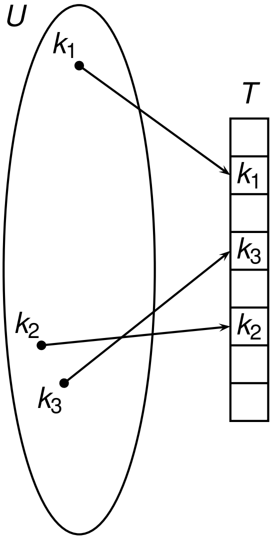
Open-Address Hash Table



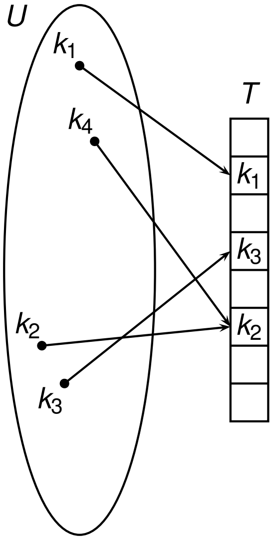
Open-Address Hash Table



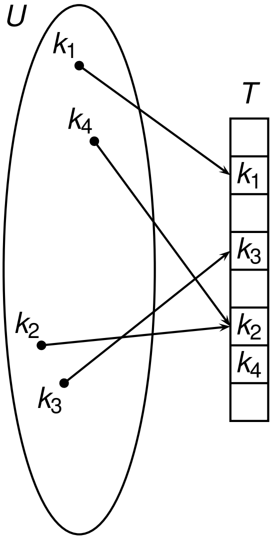
Open-Address Hash Table



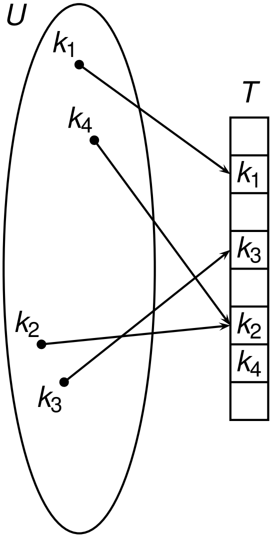
Open-Address Hash Table



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"
```

- *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"
```

```
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