Algorithms and Data Structures (II)

Gabriel Istrate

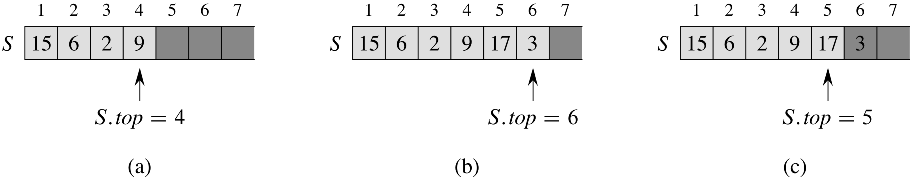
March 4, 2020

First of all ...

Last time: Stacks

- A Stack is a sequential organization of items in which the last element inserted is the first element removed. They are often referred to as LIFO, which stands for “last in first out.”
- Examples: letter basket, stack of trays, stack of plates.
- Only element that may be accessed: the one that was most recently inserted.
- There are only two basic operations on stacks, the push (insert), and the pop (read and delete).

Stacks: Implementation



- (a). Stack representing set $S = \{2, 6, 9, 15\}$.
- (b). After PUSH(S,3).
- (c). After POP(S).

Operator Precedence Parsing

- We can use the stack class we just defined to parse and evaluate mathematical expressions like:

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

- First, we transform it to postfix notation:

$$5\ 9\ 8\ +\ 4\ 6\ *\ *\ 7\ +\ *$$

- Usual form for arithmetic expressions: infix. term1 op term2.
- Postfix notation: term1 term2 op.
- How to convert infix to postfix: later !

Evaluating Postfix expressions

Then, the following C++ routine uses a stack to perform this evaluation:

```
char c;
Stack acc(50);
int x;
while (cin.get(c))
{
x = 0;
while (c == ’ ’ ) cin.get(c);
if (c == ’+’) x = acc.pop() + acc.pop();
if (c == ’*’) x = acc.pop() * acc.pop();
while (c≥ ’0’ && c ≤ ’9’)
x = 10*x + (c-’0’); cin.get(c);
acc.push(x);
}
cout << acc.pop();
```

Explanation of code

- We read one character at a time in c.
- In x we compute the value of the currently evaluated expression.
- After computing it we push the value on the stack - we will need it later.
- When reading an op we take the last two value off the stack and apply the op on them and assign this to x.
- When reading a digit we update value of x by making the last read digit the least significant one.

Stacks: Applications

- Algorithms (later).
- Recursion removal.
- Reversing things.
- Procedure call and procedure return is similar to matching symbols:
  - When a procedure returns, it returns to the most recently active procedure.
  - When a procedure call is made, save current state on the stack. On return, restore the state by popping the stack.
  - Formal languages: pushdown automata.

Queues

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

Queues

- 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

Queues

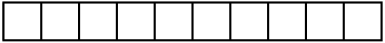
- The ubiquitous “first-in first-out” container (FIFO)

- 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

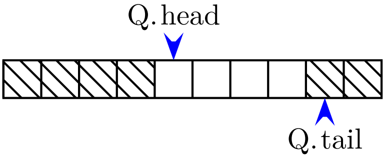
Enqueue

```
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
```



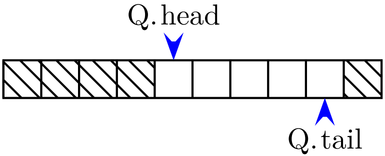
Enqueue

```
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
```



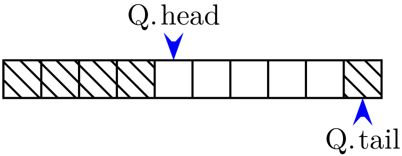
Enqueue

```
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 $Q.tail == Q.head$
        $Q.queue\text{-}full = true$
    $Q.queue\text{-}empty = false$
```



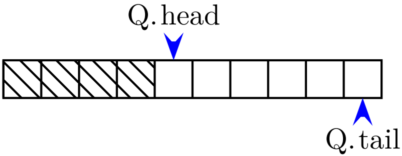
Enqueue

```
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
```



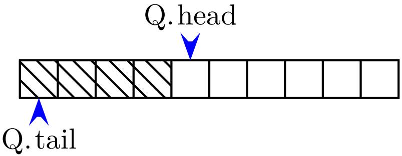
Enqueue

```
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 $Q.tail == Q.head$
        $Q.queue\text{-}full = true$
    $Q.queue\text{-}empty = false$
```



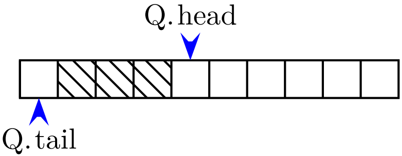
Enqueue

```
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
```



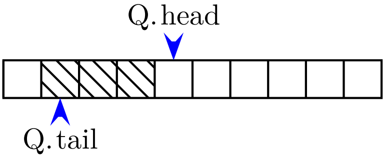
Enqueue

```
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
```



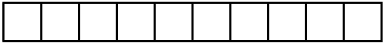
Enqueue

```
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
```



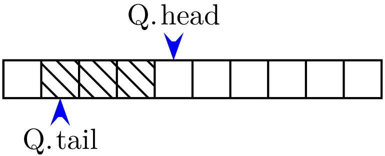
Dequeue

```
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
```



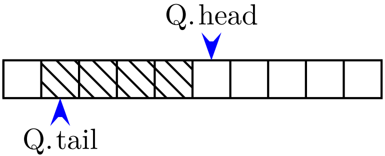
Dequeue

```
Deque(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
```



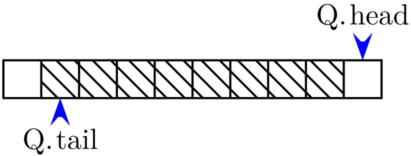
Dequeue

```
Deqeueue(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
```



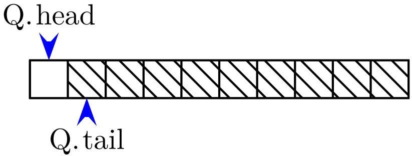
Dequeue

```
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

```
Deqeue(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
```



Applications of Queues

- Scheduling (disk, CPU)
- Used by operating systems to handle congestion.
- Algorithms (we'll see): breadth-first search.

Stacks,Queues: Scorecard

Algorithm	Complexity
Stack-Empty	O(1) ✓
Push	O(1) ✓
Pop	O(1) ✓
Enqueue	O(1) ✓
Dequeue	O(1) ✓
Restrictions:	LIFO/FIFO orders only. ×

Deques

- Like queues but can enqueue/dequeue at both ends.
- Can modify the code for queues, add two more procedure.
- do it !
- Complexity scorecard: similar to queues.

Dynamic sets

Major problem this semester:

Represent a set S whose elements may vary through time. May want to perform some of:

- INSERT(S,x)
- DELETE(S,x)
- SEARCH(S,x). Result YES/NO. Better: handle for x, if found.
- MIN(S)
- MAX(S)
- SUCC(S,x), PRED(S,x)

Example: stacks/queues

- Stacks: dynamic sets with LIFO order.
- Queues: dynamic sets with FIFO order.

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)
    - $\mathrm{Insert(D, k)}$ adds a key k to the dictionary D
    - $\mathrm{Delete(D, k)}$ removes key k from D
    - $\mathrm{Search(D, k)}$ 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)
    - $\mathrm{Insert}(D, k)$ adds a key k to the dictionary D
    - $\mathrm{Delete}(D, k)$ removes key k from D
    - $\mathrm{Search}(D, k)$ 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)
    - $\text{Insert}(D, k)$ adds a key k to the dictionary D
    - $\text{Delete}(D, k)$ removes key k from D
    - $\text{Search}(D, k)$ tells whether D contains a key k
- Implementation
    - many (concrete) data structures
    - we'll see: 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 $\mathrm{U} = \{1, 2, \ldots, \mathrm{M}\}$
- 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?

- 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?

- 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

- Want: the benefits of a direct-address table but with a table of reasonable size.

Direct Access Tables: Scorecard

Algorithm	Complexity
INSERT	O(1)✓
DELETE	O(1)✓
SEARCH	O(1)✓
MEMORY:	θ(M)×

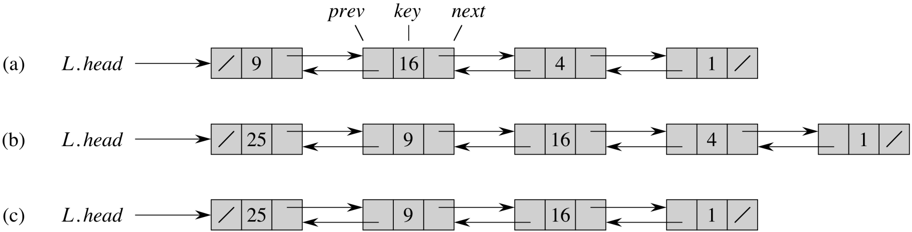
Linked Lists

- 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 Lists

- 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: two "links" $x.prev$ and $x.next$ to the previous and next elements, respectively
  - each element x: key $x.key$

Linked List: Implementation



- (a). Linked list representing set $S = \{1, 4, 9, 16\}$.
- (b). After LIST-INSERT(S,25).
- (c). After LIST-DELETE(S,4).

Linked List: Implementation

```
List-Init(L)
L.head= NIL
```

```
List-Insert(L, x)
x.next = L.head
if $L.head \neq NIL$
    L.head.prev = x
    L.head = x
    x.prev = NIL
```

```
List-Search(L, k)
x = L.head.next
while $x \neq NIL \wedge x.key \neq k$
    x = x.next
return x
```

Linked List: Implementation (II)

```
List-Delete(L, x)
if x.prev $\neq$ NIL
        x.prev.next = x.next
else L.head = x.next
if x.next $\neq$ NIL
        x.next.prev = x.prev
```

Linked List With a “Sentinel”

- instead of NIL sometimes convenient to have a dummy “sentinel” element L. nil
- Simplifies LIST-DELETE .
- Adds more memory ×.

Linked List With a “Sentinel”

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

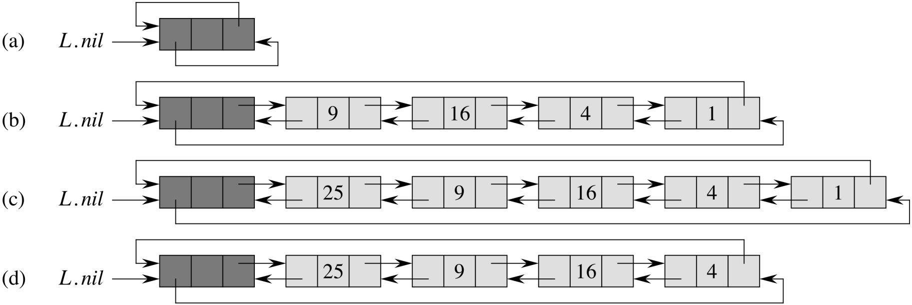
```
List-Insert(L, x)
    $x.next = L.nil.next$
    $L.nil.next.prev = x$
    $L.nil.next = x$
    $x.prev = L.nil$
```

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

Linked Lists: Observations on Implementation

- Insert: at the head of the list.
- Possible: insert arbitrary position.

Circular Linked Lists



- Can use nil sentinel as head of the list.
- (a): empty circular list.
- (b): Linked list representing set $S = \{1, 4, 9, 16\}$.
- (c): After LIST-INSERT(S,25).
- (d): After LIST-DELETE(S,4).

Linked Lists: Scorecard

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	O(1) ✓
List-Delete (with pointer)	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	O(1) ✓
List-Delete (with pointer)	O(1) ✓
List-Search	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	O(1) ✓
List-Delete (with pointer)	O(1) ✓
List-Search	Θ(n) ×

Linked Lists: to conclude

- Can reimplement Stacks/Queues using Linked Lists.
- Implementation with pointers: will not pass the class if you don't know it !

Hash Tables

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

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

Hash Tables

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

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

```
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 Tables

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

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

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

```
Hash-Delete(T, k)
    $T[h(k)] = false$
```

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

Are these algorithms always correct?

Hash Tables

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

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

```
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 always correct? No!

Hash Tables

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

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

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

```
Hash-Delete(T, k)
    $T[h(k)] = false$
```

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

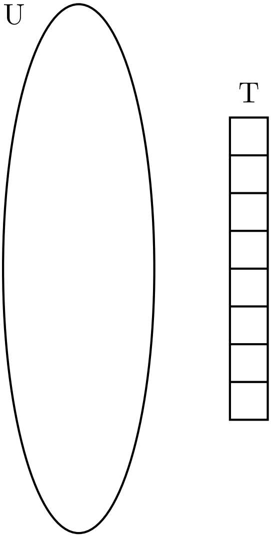
Are these algorithms always correct? No!

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

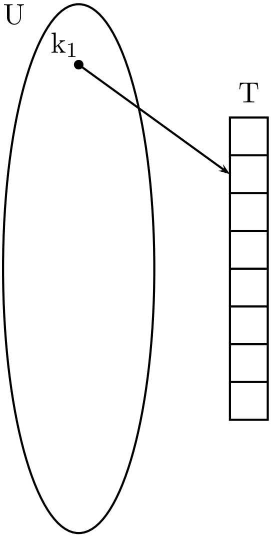
Hash tables

- Hash Tables: work well ”on the average”
- Analogy: throw T balls at random into N bins.
- If T << N (in fact $T = o(\sqrt{N})$) then with high-probability no two balls land in the same bin.
- Want our hash-function to be ”random-like”: elements of U ”thrown out uniformly” by h onto elements of T.

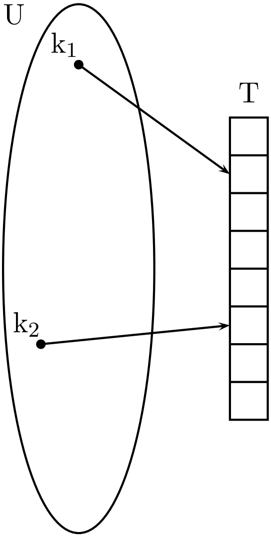
Hash Table: Chaining



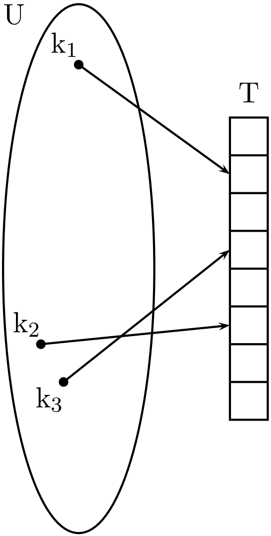
Hash Table: Chaining



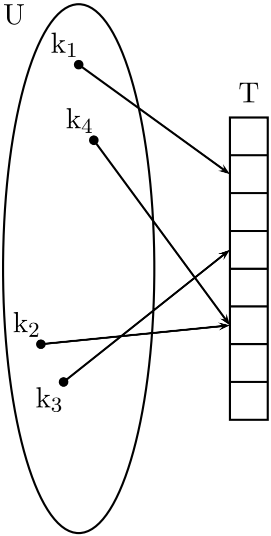
Hash Table: Chaining



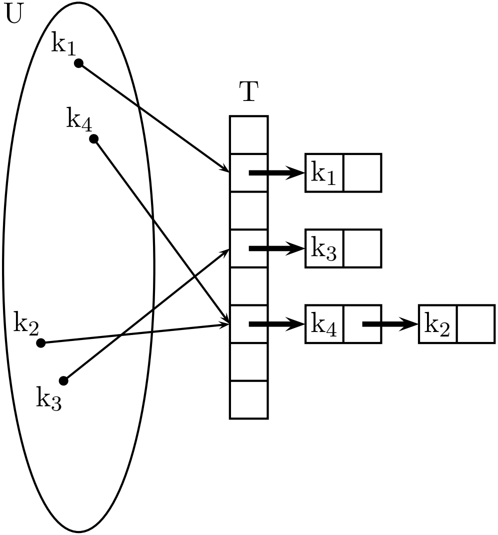
Hash Table: Chaining



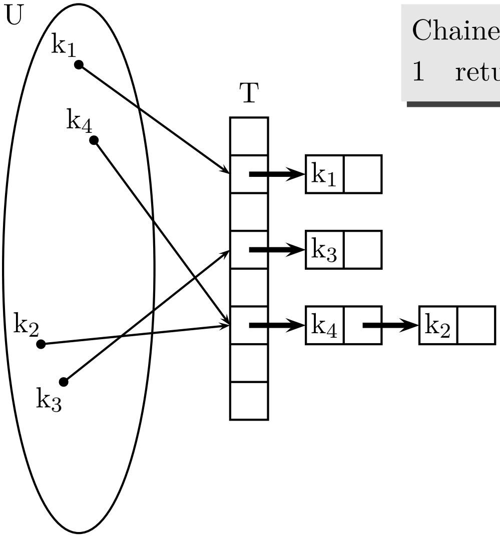
Hash Table: Chaining



Hash Table: Chaining

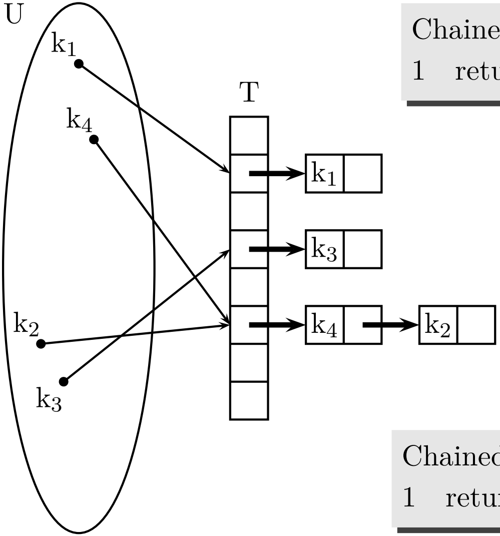


Hash Table: Chaining



```
Chained-Hash-Insert(T, k)
1   return List-Insert(T[h(k)], k)
```

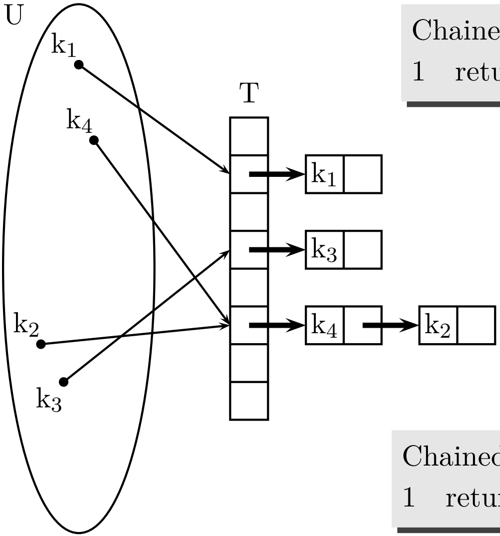
Hash Table: Chaining



```
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)
```

Hash Table: Chaining



```
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)
```

Hashing With Chaining: Analysis

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

Hashing With Chaining: Analysis

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

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

(The formalism is actually a bit more complicated.)

Hashing With Chaining: Analysis

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

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

(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

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

Hashing With Chaining: Analysis

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

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

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length $\mathrm{n_i}$ of the linked list at position i is

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

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

Hashing With Chaining: Analysis

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

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

  (The formalism is actually a bit more complicated.)

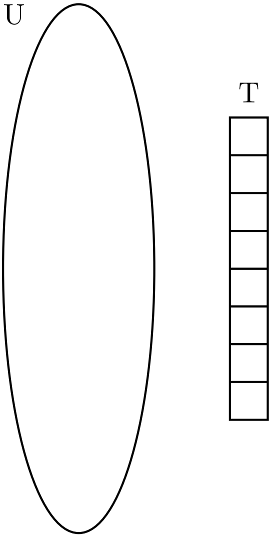
- So, given n distinct keys, the expected length n_i of the linked list at position i is

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

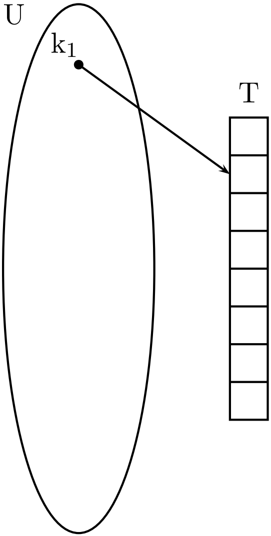
- We further assume that $h(k)$ can be computed in $O(1)$ time
- Therefore, the complexity of Chained-Hash-Search is

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

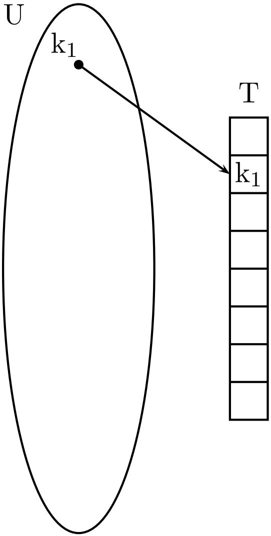
Open-Address Hash Table



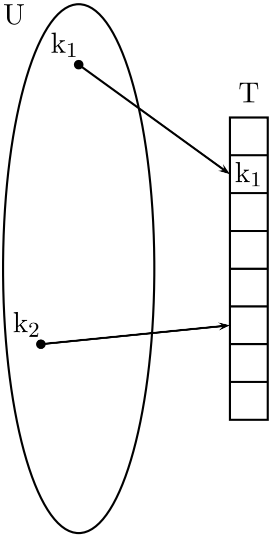
Open-Address Hash Table



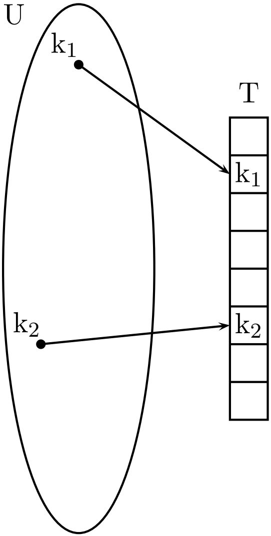
Open-Address Hash Table



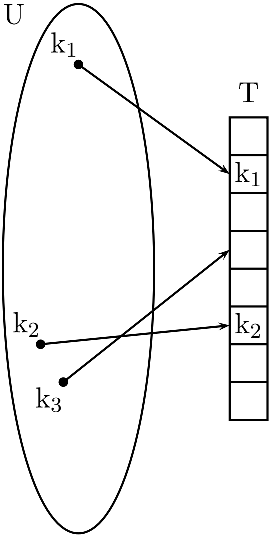
Open-Address Hash Table



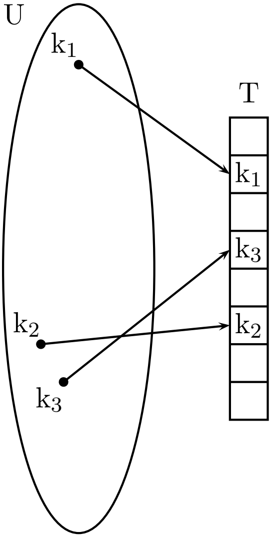
Open-Address Hash Table



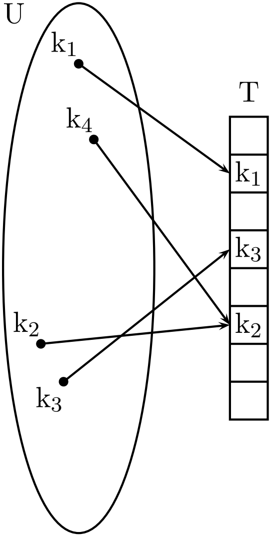
Open-Address Hash Table



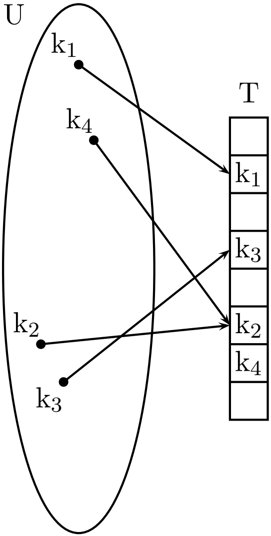
Open-Address Hash Table



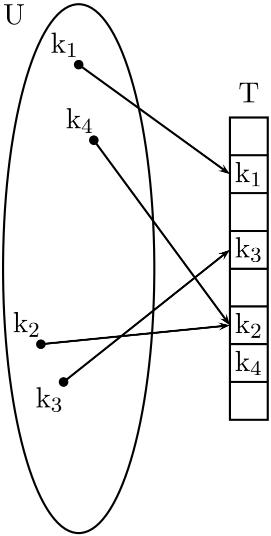
Open-Address Hash Table



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