Linked lists & Hash tables

Pontus Ekberg

Uppsala University

(Based on previous material by Mohamed Faouzi Atig and Parosh Aziz Abdulla)

Abstract data types & Data structures

- An abstract data type defines a type of object that can be used in an algorithm/program.

Abstract data types & Data structures

- An abstract data type defines a type of object that can be used in an algorithm/program.

- An abstract data type is defined by a set of operations that can be performed on it, such as: Insert, Delete, Access, Search

Abstract data types & Data structures

- An abstract data type defines a type of object that can be used in an algorithm/program.

- An abstract data type is defined by a set of operations that can be performed on it, such as: INSERT, DELETE, ACCESS, SEARCH

- A data structure is a particular way to store and organize data in order to facilitate the operations definied for an abstract data type.

Abstract data types & Data structures

- An abstract data type defines a type of object that can be used in an algorithm/program.
- An abstract data type is defined by a set of operations that can be performed on it, such as: INSERT, DELETE, ACCESS, SEARCH
- A data structure is a particular way to store and organize data in order to facilitate the operations definied for an abstract data type.
- An abstract data type is implemented by a data structure and its corresponding functions. For example, priority queues can be implemented by heaps.

Abstract data types & Data structures

- An abstract data type defines a type of object that can be used in an algorithm/program.
- An abstract data type is defined by a set of operations that can be performed on it, such as: INSERT, DELETE, ACCESS, SEARCH
- A data structure is a particular way to store and organize data in order to facilitate the operations definied for an abstract data type.
- An abstract data type is implemented by a data structure and its corresponding functions. For example, priority queues can be implemented by heaps.
- The complexity of the operations depend on their implementations.

Abstract data types & Data structures

- An abstract data type defines a type of object that can be used in an algorithm/program.

- An abstract data type is defined by a set of operations that can be performed on it, such as: INSERT, DELETE, ACCESS, SEARCH

- A data structure is a particular way to store and organize data in order to facilitate the operations definied for an abstract data type.

- An abstract data type is implemented by a data structure and its corresponding functions. For example, priority queues can be implemented by heaps.

- The complexity of the operations depend on their implementations.

- A data structure can use other data structures.

Lists

A List is an abstract data type that stores a dynamic number of elements in order, where the size of the list can grow and shrink. It should support (for example) the following operations.

- SEARCH
- INSERT (at the head of the list)
- INSERT (at the end of the list)
- INSERT (at a given position in the list)
- DELETE
- HEAD

The elements sometimes have an identifying key in addition to its value.

Lists

A List is an abstract data type that stores a dynamic number of elements in order, where the size of the list can grow and shrink. It should support (for example) the following operations.

- SEARCH
- INSERT (at the head of the list)
- INSERT (at the end of the list)
- INSERT (at a given position in the list)
- DELETE
- HEAD

The elements sometimes have an identifying key in addition to its value.

Typical data structures used to implement Lists are Dynamic arrays and Linked lists.

Singly Linked Lists

- A singly linked list L is a data structure in which the objects are arranged in a linear order.
- Each element x of a linked list has a pointer attribute $x.next$
- The singly linked list L has an attribute $L.head$ pointing to the first element of the list. If $L.head = NIL$, the list is empty.
- The last element x of the linked list L has $x.next = NIL$

Here NIL is some predefined value that indicates the lack of another value.

Singly Linked Lists


```
LIST-SEARCH(L, k)
x ← L.head
while x ≠ NIL and x.key ≠ k
      do x ← x.next
return x
```

```
LIST-INSERT(L, x)
x.next ← L.head
L.head ← x
```

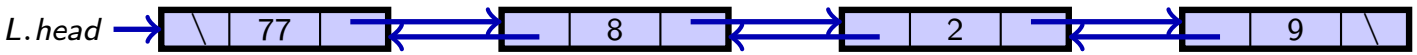
```
LIST-DELETE(L, x)
y ← L.head
if y = x
   then L.head ← x.next
if y ≠ NIL and y ≠ x
   then while y.next ≠ x and y.next ≠ NIL
              do y ← y.next
        if y.next ≠ NIL
           then y.next ← x.next
```

- Worst-Case Complexity:
    - Insertion (at the head of the list): $O(1)$
    - Deletion: $O(n)$ where n is the size of the list.
    - Searching: $O(n)$ where n is the size of the list.

Doubly Linked Lists

- A doubly linked list L is a data structure in which the objects are arranged in a linear order.
- Each element x of a linked list has two pointer attributes $x.\,next$ and $x.\,prev$
- The doubly linked list L has an attribute $L.\,head$ pointing to the first element of the list. If $L.\,head = NIL$, the list is empty.
- The last element x of the linked list L has $x.\,next = NIL$
- The first element x of the linked list L has $x.\,prev = NIL$

Doubly Linked Lists



LIST-SEARCH(L, k)

```
x ← L.head
while x ≠ NIL and x.key ≠ k
      do x ← x.next
return x
```

LIST-INSERT(L, x)

```
x.next ← L.head
if L.head ≠ NIL
   then L.head.prev ← x
L.head ← x
x.prev ← NIL
```

LIST-DELETE(L, x)

```
if x.prev ≠ NIL
   then x.prev.next ← x.next
   else L.head ← x.next
if x.next ≠ NIL
   then x.next.prev ← x.prev
```

- Worst-Case Complexity:
  - Insertion (at the head of the list): $O(1)$
  - Deletion: $O(1)$
  - Searching: $O(n)$ where n is the size of the list.

Dictionary

A dictionary is an abstract data type for elements with keys that supports the following operations:

- $\textsc{Insert}(S, x)$: Given an element x, add it to the dictionary S
- $\textsc{Delete}(S, x)$: Given an element x, remove it from the dictionary S in the case that x is in S.
- $\textsc{Search}(S, k)$: Given a search key k, return the element x in the dictionary S whose key value is k (i.e., $x.key = k$) if one exists, otherwise return NIL.

Dictionary

A dictionary is an abstract data type for elements with keys that supports the following operations:

- $\textsc{Insert}(S, x)$: Given an element x, add it to the dictionary S
- $\textsc{Delete}(S, x)$: Given an element x, remove it from the dictionary S in the case that x is in S.
- $\textsc{Search}(S, k)$: Given a search key k, return the element x in the dictionary S whose key value is k (i.e., $x.key = k$) if one exists, otherwise return NIL.

We assume that keys are unique.

Dictionary

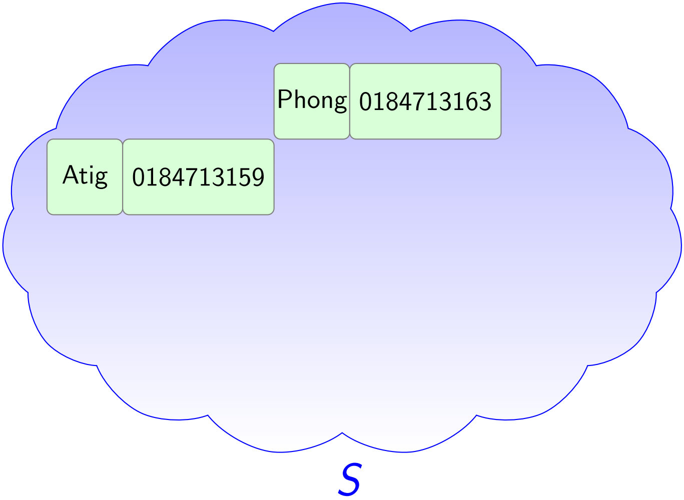
A dictionary is an abstract data type for elements with keys that supports the following operations:

- $\textsc{Insert}(S, x)$: Given an element x, add it to the dictionary S
- $\textsc{Delete}(S, x)$: Given an element x, remove it from the dictionary S in the case that x is in S.
- $\textsc{Search}(S, k)$: Given a search key k, return the element x in the dictionary S whose key value is k (i.e., $x.key = k$) if one exists, otherwise return NIL.

We assume that keys are unique.

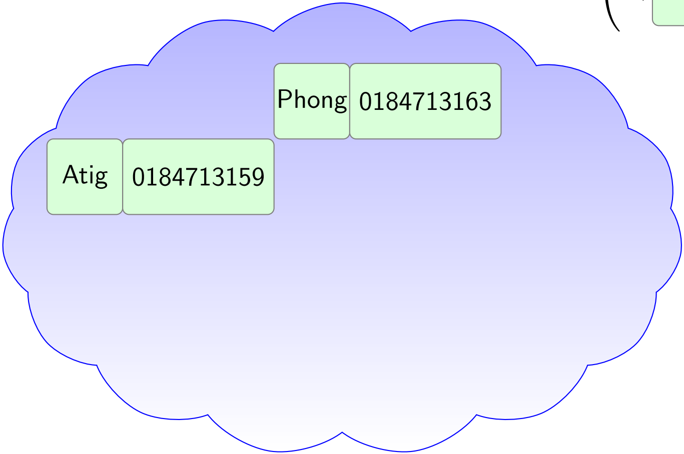
Dictionaries are also called Associative arrays, Maps, or Symbol tables.

Example: Phone Book



Example: Phone Book

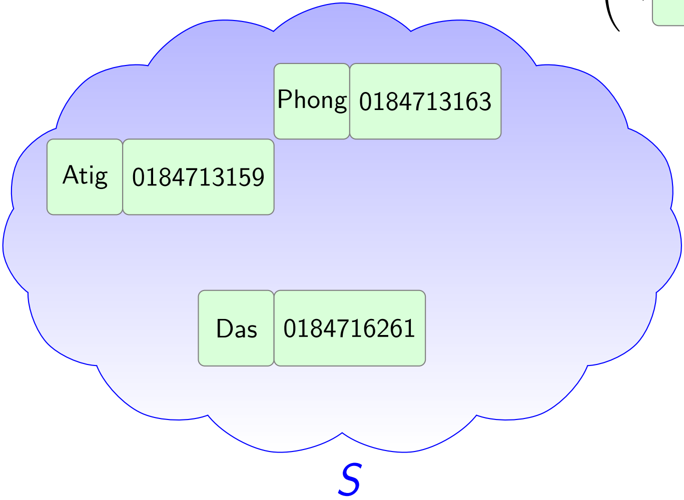
$\textsc{Insert}\left(S, \boxed{\text{Das}}\boxed{\text{0184716261}}\right)$



S

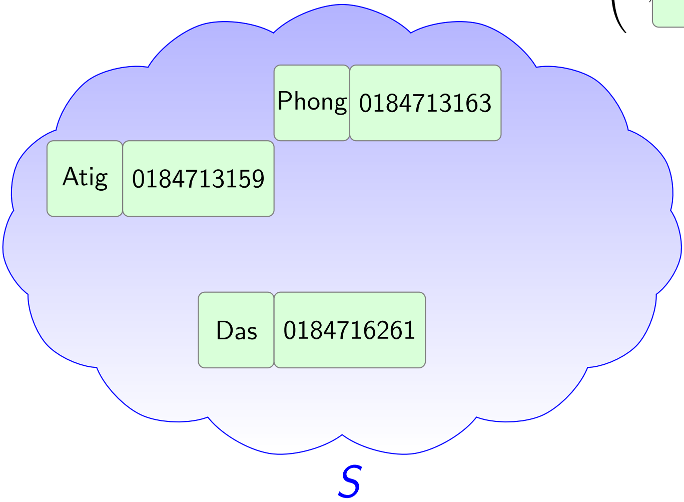
Example: Phone Book

$\textsc{Insert}\left(S, \boxed{\text{Das}}\boxed{0184716261}\right)$



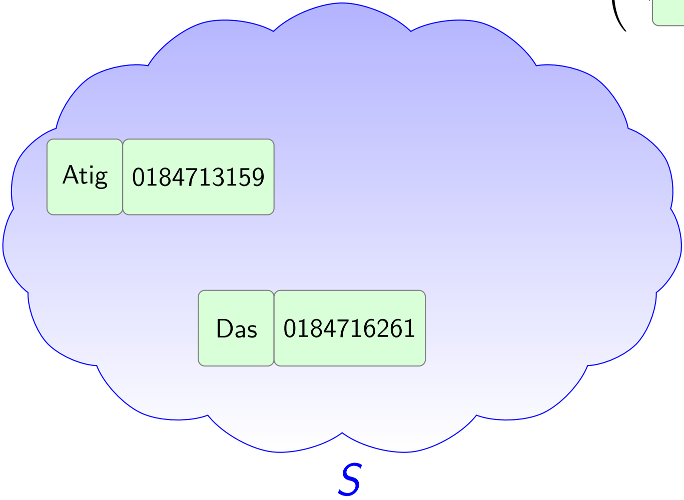
Example: Phone Book

$$\textsc{Delete}\left(S, \boxed{\text{Phong}}\boxed{0184713163}\right)$$



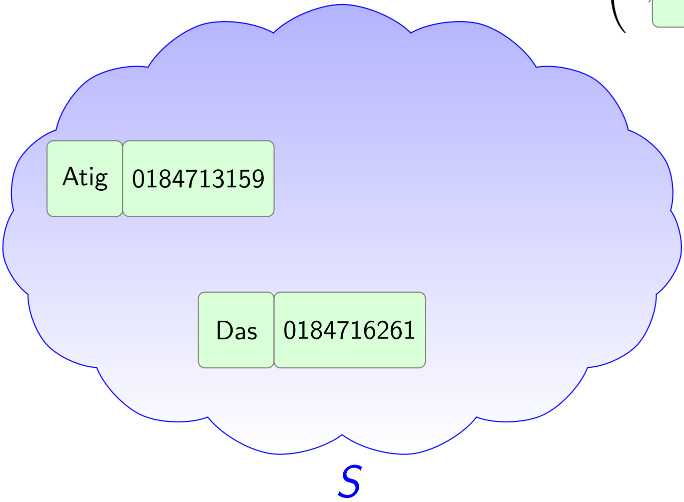
Example: Phone Book

$$\textsc{Delete}\left(S, \boxed{\text{Phong}}\boxed{0184713163}\right)$$



Example: Phone Book

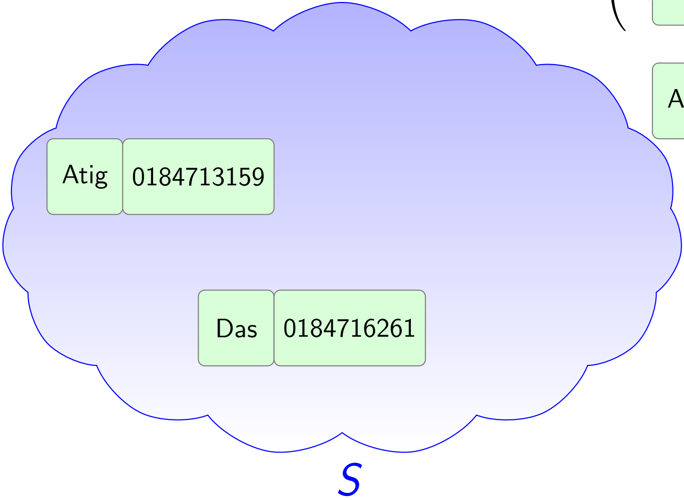
$$\textsc{Search}\left(S, \boxed{\text{Atig}}\right)$$



Example: Phone Book

$\textsc{Search}\left(S, \boxed{\text{Atig}}\right)$

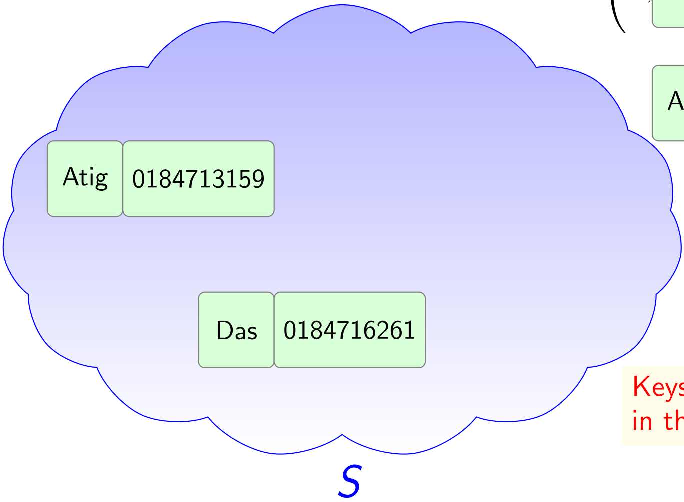
Atig	0184713159



Example: Phone Book

$\textsc{Search}\left(S, \boxed{\text{Atig}}\right)$

Atig	0184713159



Keys are strings in this example!

Strings Are Numbers!

Strings Are Numbers!

- Each character in a string can take one of 256 different values (in extended ASCII encoding).
  - The character A has the value 65.
  - The character t has the value 116.
  - The character i has the value 105.
  - The character g has the value 103.
- Strings can be seen as number in base 256
  - Atig can be represented $65 \cdot 256^3 + 116 \cdot 256^2 + 105 \cdot 256^1 + 103 \cdot 256^0$
  - Atig can be represented $1,098,148,199$ in base 10

Dictionary: Task

A dictionary is an abstract data type for elements with keys that supports the following operations:

- $\text{INSERT}(S, x)$: Given an element x, add it to the dictionary S
- $\text{DELETE}(S, x)$: Given an element x, remove it from the dictionary S in the case that x is in S.
- $\text{SEARCH}(S, k)$: Given a search key k, return the element x in the dictionary S whose key value is k (i.e., $x.key = k$) if one exists, otherwise return NIL.

Dictionary: Task

A dictionary is an abstract data type for elements with keys that supports the following operations:

- $\textsc{Insert}(S, x)$: Given an element x, add it to the dictionary S
- $\textsc{Delete}(S, x)$: Given an element x, remove it from the dictionary S in the case that x is in S.
- $\textsc{Search}(S, k)$: Given a search key k, return the element x in the dictionary S whose key value is k (i.e., $x.key = k$) if one exists, otherwise return NIL.

The universe U of keys is the set $\{1, \ldots, m\}$

Dictionary: Task

A dictionary is an abstract data type for elements with keys that supports the following operations:

- $\textsc{Insert}(S, x)$: Given an element x, add it to the dictionary S
- $\textsc{Delete}(S, x)$: Given an element x, remove it from the dictionary S in the case that x is in S.
- $\textsc{Search}(S, k)$: Given a search key k, return the element x in the dictionary S whose key value is k (i.e., $x.key = k$) if one exists, otherwise return NIL.

The universe U of keys is the set $\{1, \ldots, m\}$

Task: Design an effective data structure for implementing dictionaries.

Dictionary: Challenges

- Two goals:
  - Minimize the runtime of the dictionary operations.
  - Minimize the memory space used to store the data.
- Examples of Applications:
  - A symbol table of a compiler
  - DNS routing table
  - ...
- Many possible implementations

Direct Addressing

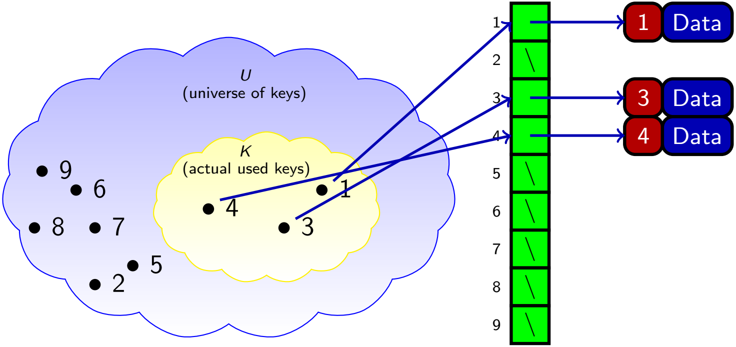
- We assume that:
  - The universe U of keys is the set $\{1, \ldots, m\}$, where m is not too large
  - Each element has its own unique key.
- To implement a dictionary we use an array T[1..m]:
  - Each position (or slot) corresponds to a key in the universe U
  - If there is an object x with the key k, then $T[k]$ contains x
  - If the set contains no element with key k, then $T[k] \leftarrow NIL$

Direct-Address Table

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

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.\ key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.\ key] \leftarrow NIL$

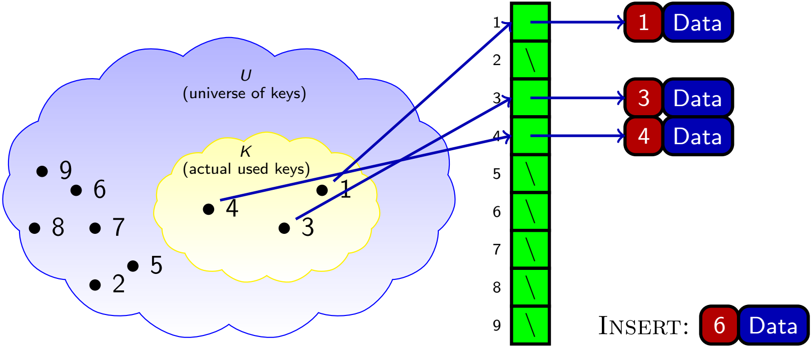


Direct-Address Table

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

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.key] \leftarrow NIL$



Direct-Address Table

DIRECT-ADDRESS-SEARCH(T, k)

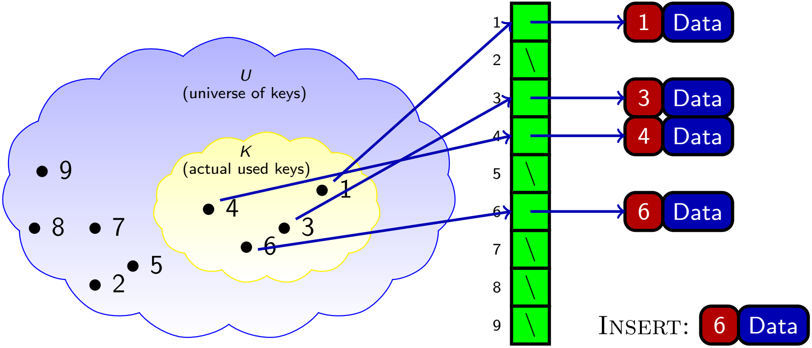
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[x.key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)

$T[x.key] \leftarrow NIL$



Direct-Address Table

DIRECT-ADDRESS-SEARCH(T, k)

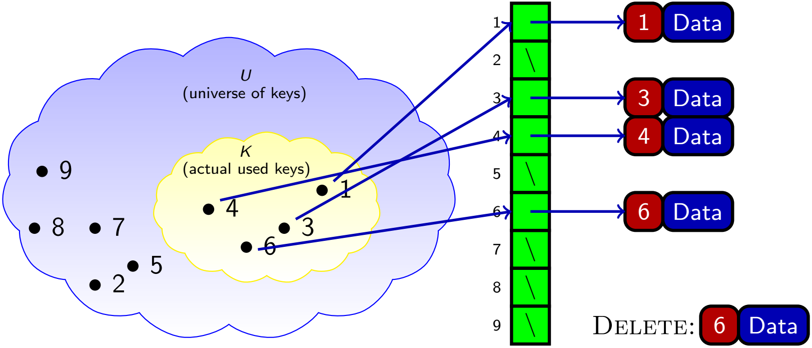
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[x.\,key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)

$T[x.\,key] \leftarrow NIL$

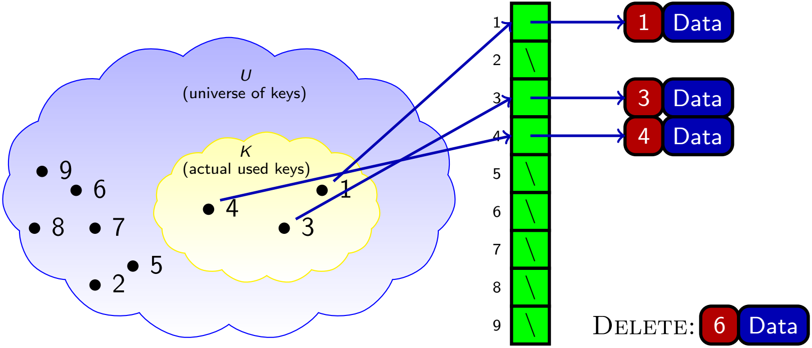


Direct-Address Table

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

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.key] \leftarrow NIL$

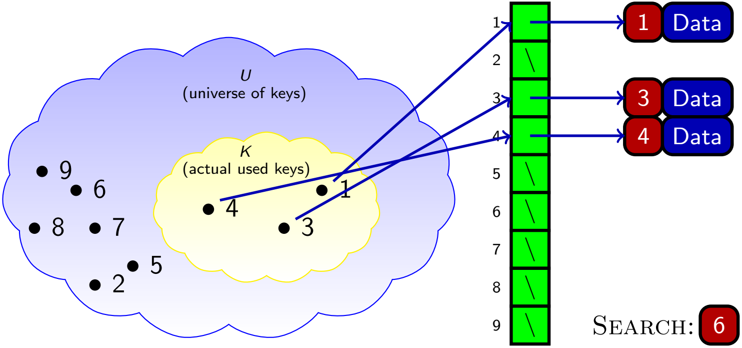


Direct-Address Table

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

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.\ key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.\ key] \leftarrow NIL$

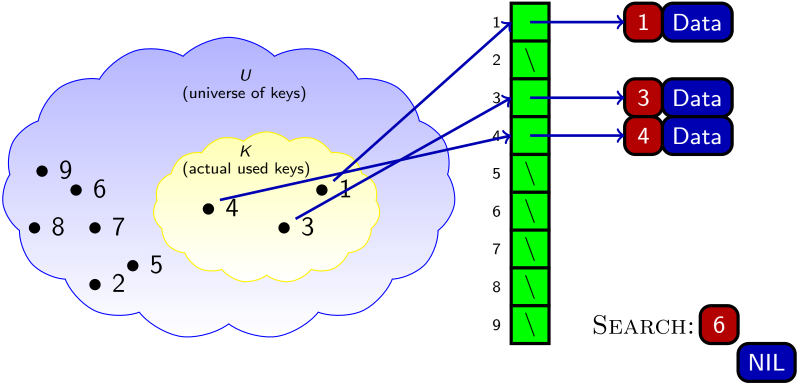


Direct-Address Table

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

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.key] \leftarrow NIL$



Direct-Address Table: Complexity

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

DIRECT-ADDRESS-INSERT(T, x)
$T[x.key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
$T[x.key] \leftarrow NIL$

- Each operation requires $O(1)$ (in all the cases)
- Problem:
  - Space complexity: $\Theta(|U|)$ where $|U|$ denotes the size of U
  - If U is large, then storing an array of size $|U|$ may be impractical, or even impossible.
- Often the set K of keys actually stored may be so small relative to U (Most of space allocated to the array would be wasted)
  - Phone book example: We need an array of size at least $1,089,148,199$ so that we could index into it the string Atig

Can We Do Better?

- Goals:
  - Fast running time for the dictionary operations
  - Minimize the memory space used to store data.

Can We Do Better?

- Goals:
  - Fast running time for the dictionary operations
  - Minimize the memory space used to store data.
- Hashing is a technique for storing elements for a subset $K \subseteq U$ with:
  - Space complexity $\Theta(|K|)$
  - Runtime for search, insertion and deletion in $\Theta(1)$ (in average)

Hash Table

- Developed by Luhn in 1953
- Idea:
  - Use an array (hash table) of size $m << |U|$
  - Store an object x in the position $h(x.key)$, where h is a hash function

$$h : U \to \{1, \ldots, m\}$$

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

HASH-INSERT(T, x)
1 $T[h(x.key)] \leftarrow x$

HASH-DELETE(T, x)
1 $T[h(x.key)] \leftarrow NIL$

Hash Table

- Developed by Luhn in 1953
- Idea:
  - Use an array (hash table) of size $m << |U|$
  - Store an object x in the position $h(x.key)$, where h is a hash function

$$h : U \rightarrow \{1, \ldots, m\}$$

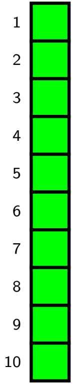
HASH-SEARCH(T, k)
1 **return** $T[h(k)]$

HASH-INSERT(T, x)
1 $T[h(x.key)] \leftarrow x$

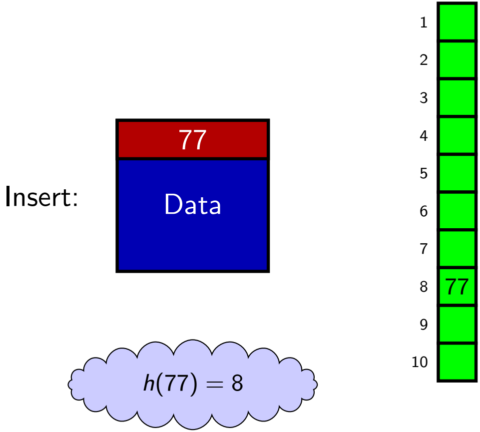
HASH-DELETE(T, x)
1 $T[h(x.key)] \leftarrow NIL$

Are these algorithms correct?

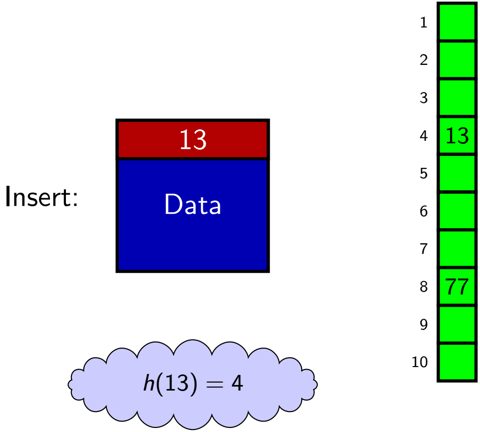
Hash Tables – Collisions



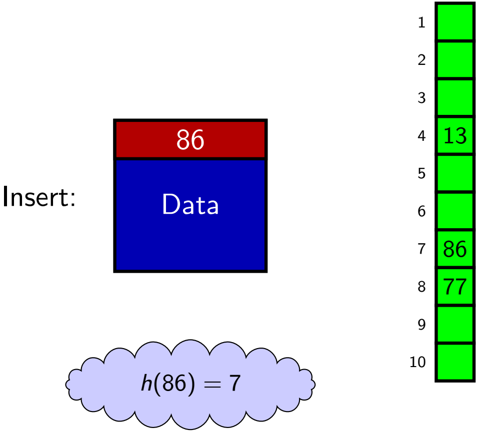
Hash Tables – Collisions



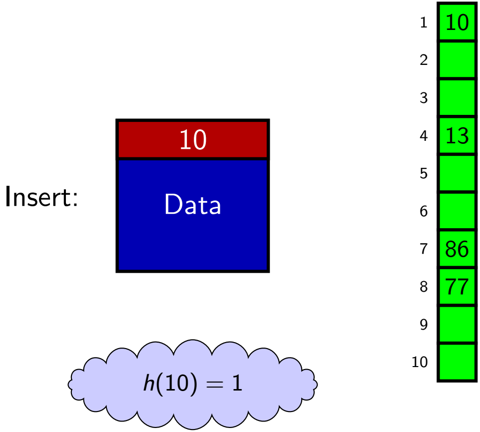
Hash Tables – Collisions



Hash Tables – Collisions

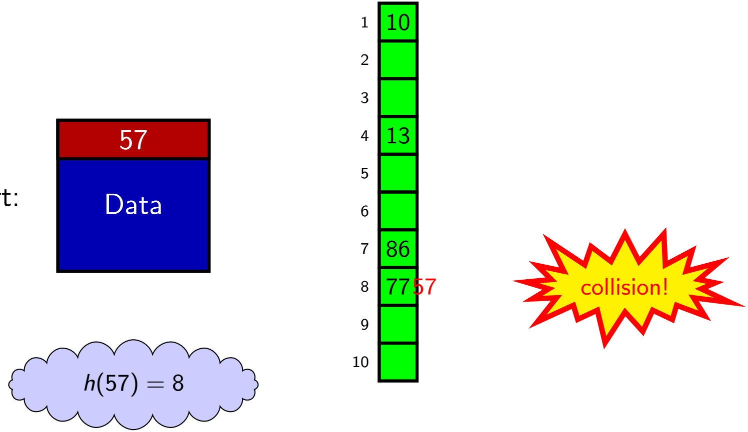


Hash Tables – Collisions

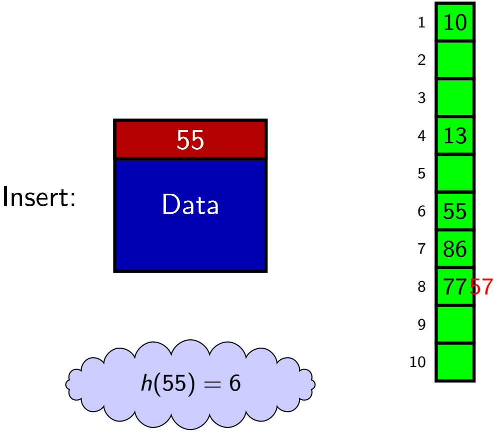


Hash Tables – Collisions

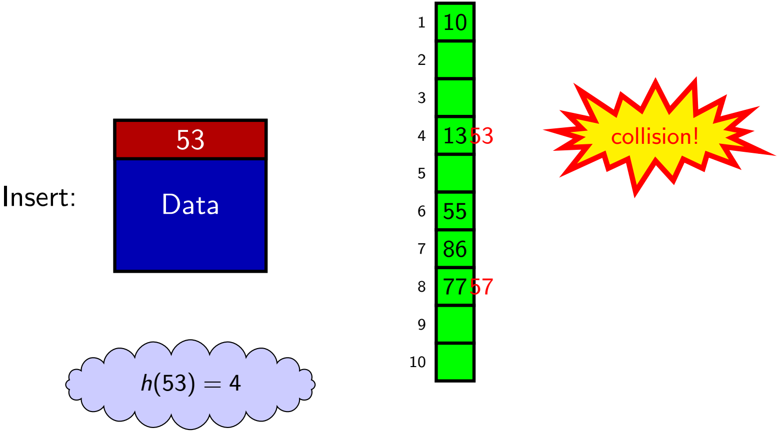
Insert:



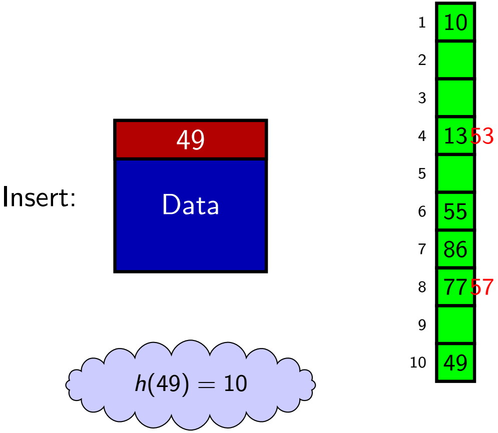
Hash Tables – Collisions



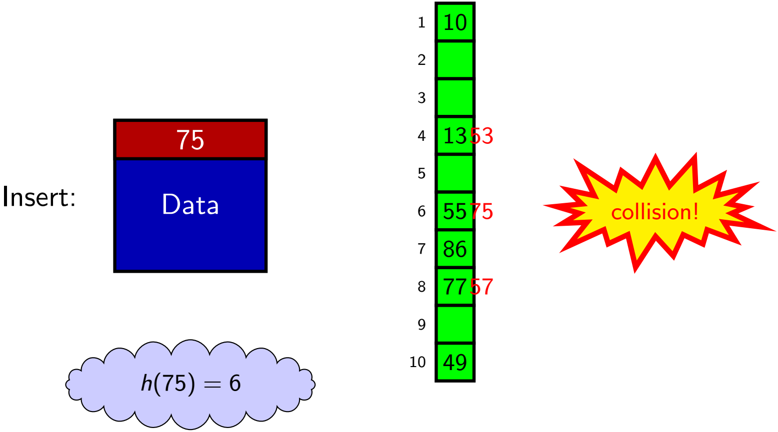
Hash Tables – Collisions



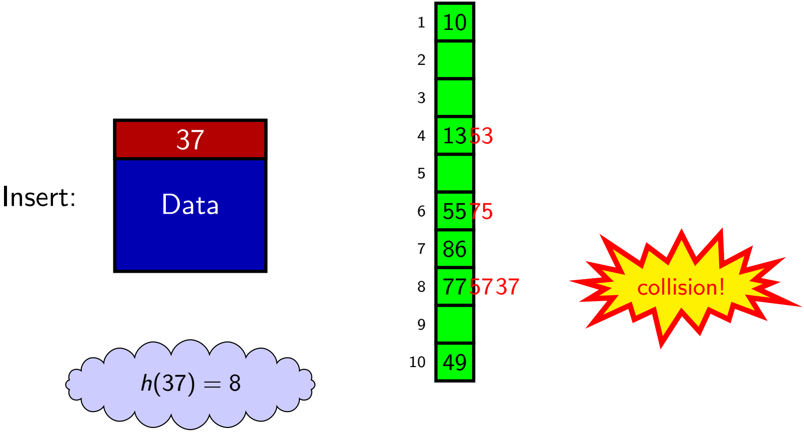
Hash Tables – Collisions



Hash Tables – Collisions



Hash Tables – Collisions



Hash Tables: Collisions

- Collision may occur, i.e., several different keys may be mapped to the same slot.

Hash Tables: Collisions

- Collision may occur, i.e., several different keys may be mapped to the same slot.
- Collisions will always occur when the number K of observed keys is larger than the size of the hash table m (i.e., $|K| > m$)

Hash Tables: Collisions

- Collision may occur, i.e., several different keys may be mapped to the same slot.
- Collisions will always occur when the number K of observed keys is larger than the size of the hash table m (i.e., $|K| > m$)
- How likely are collisions if a relatively small fraction of the hash table is used and if the hash function distributes the keys essentially uniformly?

Birthday Paradox

- Hypothesis:
  - We assume that each year contains 365 days.
  - All birthdays are equally probable.
  - n people.

Birthday Paradox

- Hypothesis:
  - We assume that each year contains 365 days.
  - All birthdays are equally probable.
  - n people.
- If p is the probability of a birthday collision:

$$1 - p = \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{365 - n + 1}{365} = \frac{365!}{(365 - n)! 365^n}$$

Birthday Paradox

- Hypothesis:
  - We assume that each year contains 365 days.
  - All birthdays are equally probable.
  - n people.
- If p is the probability of a birthday collision:

$$1 - p = \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{365 - n + 1}{365} = \frac{365!}{(365 - n)!365^n}$$

- Example:
  - $n = 23 \Rightarrow 0.5 < p < 0.51$

Birthday Paradox

- Hypothesis:
  - We assume that each year contains 365 days.
  - All birthdays are equally probable.
  - n people.
- If p is the probability of a birthday collision:

$$1 - p = \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{365 - n + 1}{365} = \frac{365!}{(365 - n)!365^n}$$

- Example:
  - $n = 23 \Rightarrow 0.5 < p < 0.51$
  - $n = 57 \Rightarrow p \approx 0.99$

Birthday Paradox

- Hypothesis:
  - We assume that each year contains 365 days.
  - All birthdays are equally probable.
  - n people.
- If p is the probability of a birthday collision:

$$1 - p = \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{365 - n + 1}{365} = \frac{365!}{(365 - n)!365^n}$$

- Example:
  - $n = 23 \Rightarrow 0.5 < p < 0.51$
  - $n = 57 \Rightarrow p \approx 0.99$
  - $n = 100 \Rightarrow p \approx 0.9999997$

Birthday Paradox

- Hypothesis:
  - We assume that each year contains 365 days.
  - All birthdays are equally probable.
  - n people.
- If p is the probability of a birthday collision:

$$1 - p = \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{365 - n + 1}{365} = \frac{365!}{(365 - n)! 365^n}$$

- Example:
  - $n = 23 \Rightarrow 0.5 < p < 0.51$
  - $n = 57 \Rightarrow p \approx 0.99$
  - $n = 100 \Rightarrow p \approx 0.9999997$
- For a hash table:
  - $m = 365$ and 57 keys $\Rightarrow$ collision probability is higher than 99%
  - $m = 1000000$ and 2500 keys $\Rightarrow$ collision probability is higher than 95%

Hash Table: Collision

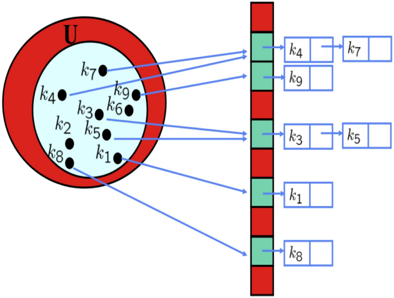
- To avoid many collisions:
  - Ensure simple uniform hashing: Keys should be uniformly disturbed (Each key is equally likely to mapped to any of the m slots, independently of where the other keys are mapped to).
  - Use a sufficient number of slots.

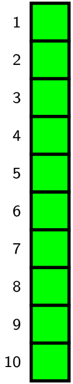
  (Even in this case, the probability of collision is significant, as illustrated by the Birthday Paradox!)

- Two approaches to deal with collisions:
  - Chaining (Close Addressing)
  - Probing (Open Addressing)

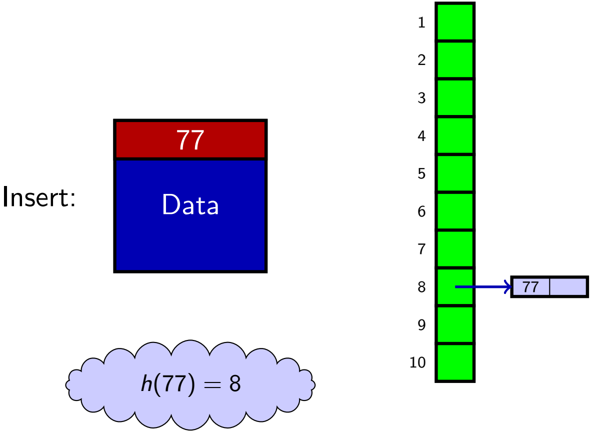
Collision Resolution by Chaining

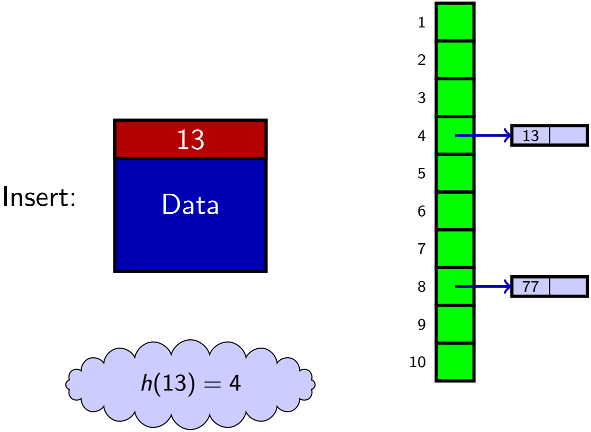
- Put all elements which mapping to the same slot in a linked list.
- Implement the operations as follows:
  - CHAINED-HASH-INSERT(T, x)
    Insert the element x at the head of the list $T(h(x.key))$
  - CHAINED-HASH-SEARCH(T, k)
    Search for element x with a key k in the list $T(h(k))$
  - CHAINED-HASH-DELETE(T, x)
    Delete the element x from the list $T(h(x.key))$



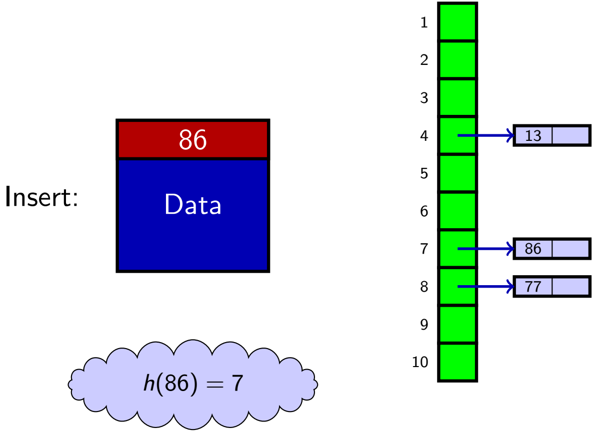

1
2
3
4
5
6
7
8
9
10

Chaining

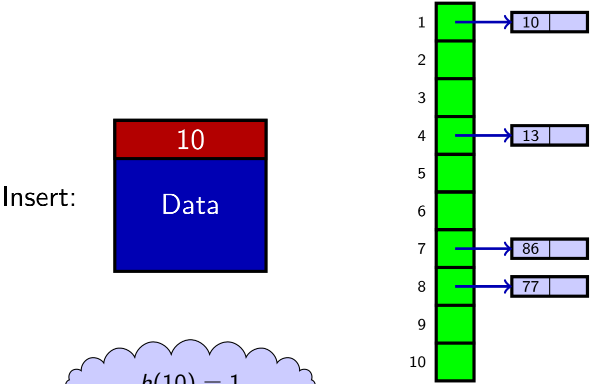



Insert:
13
Data
$h(13) = 4$
1
2
3
4
5
6
7
8
9
10
13
77

Chaining

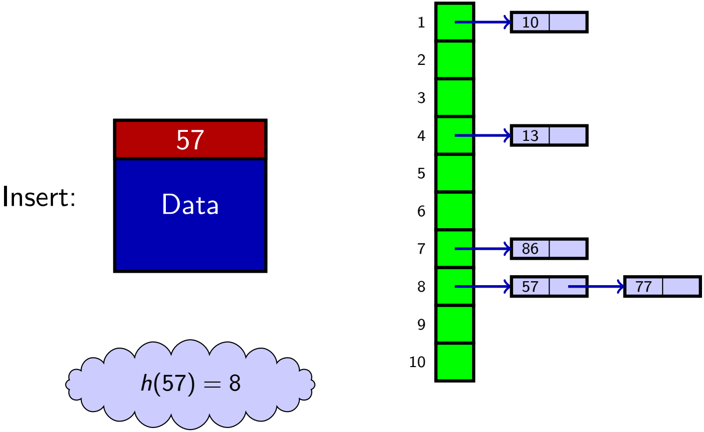


Chaining

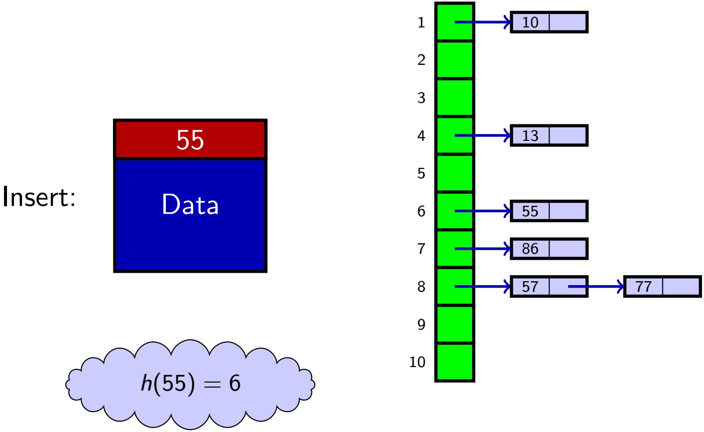


$h(10) = 1$

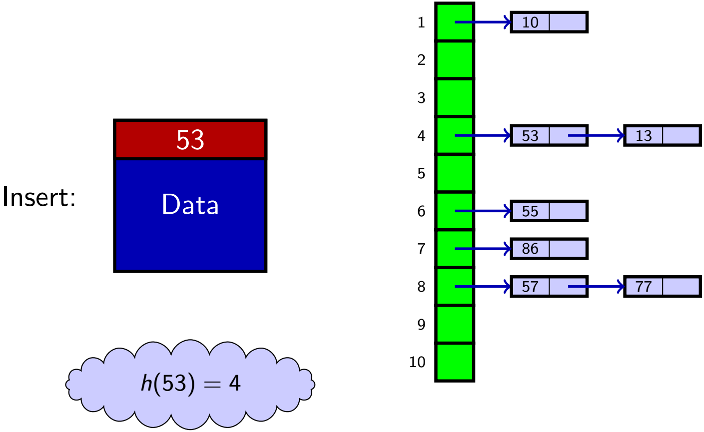
Chaining



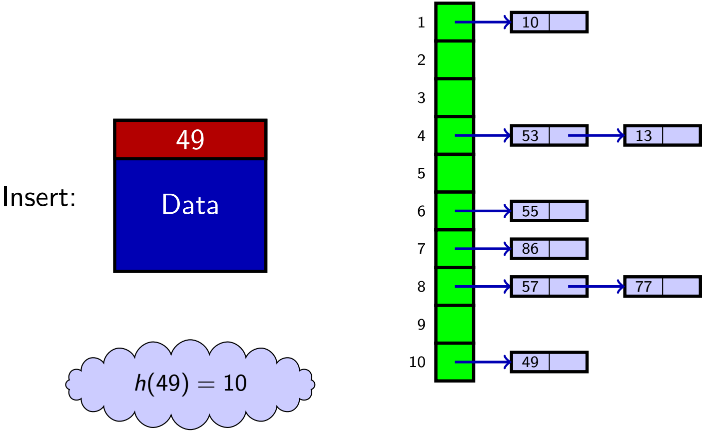
Chaining



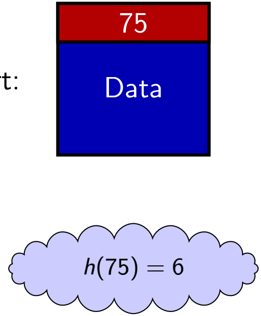
Chaining

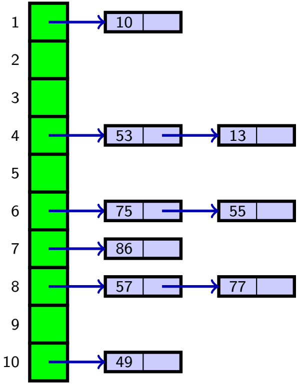


Chaining

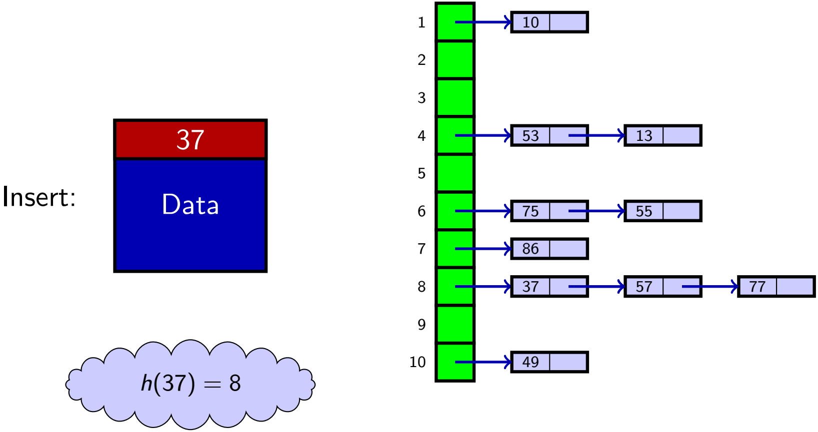


Insert:





Chaining



Review: Singly Linked Lists

LIST-SEARCH(L, k)

```
x ← L.head
while x ≠ NIL and x.key ≠ k
      do x ← x.next
return x
```

LIST-INSERT(L, x)

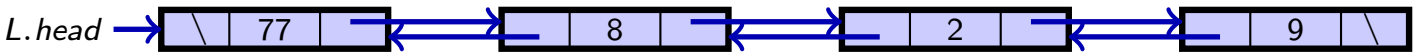
```
x.next ← L.head
L.head ← x
```

LIST-DELETE(L, x)

```
y ← L.head
if y = x
   then L.head ← x.next
if y ≠ NIL and y ≠ x
   then while y.next ≠ x and y.next ≠ NIL
                 do y ← y.next
            if y.next ≠ NIL
               then y.next ← x.next
```

- Worst-Case Complexity:
    - Insertion: $O(1)$
    - Deletion: $O(n)$ where n is the size of the list.
    - Searching: $O(n)$ where n is the size of the list.

Review: Doubly Linked Lists



List-Search(L, k)

$x \leftarrow L.head$
while $x \neq NIL$ and $x.key \neq k$
  do $x \leftarrow x.next$
return x

List-Insert(L, x)

$x.next \leftarrow L.head$
if $L.head \neq NIL$
  then $L.head.prev \leftarrow x$
$L.head \leftarrow x$
$x.prev \leftarrow NIL$

List-Delete(L, x)

if $x.prev \neq NIL$
  then $x.prev.next \leftarrow x.next$
  else $L.head \leftarrow x.next$
if $x.next \neq NIL$
  then $x.next.prev \leftarrow x.prev$

- Worst-Case Complexity:
  - Insertion: $O(1)$
  - Deletion: $O(1)$
  - Searching: $O(n)$ where n is the size of the list.

Chaining: Implementing the Operations

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

CHAINED-HASH-INSERT(T, x)
1 LIST-INSERT($T[h(x.key)]$, x)

CHAINED-HASH-DELETE(T, x)
1 LIST-DELETE($T[h(x.key)]$, x)

- We assume that $O(1)$ time suffices to compute the hash value $h(k)$ for any key k

Chaining: Implementing the Operations

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

CHAINED-HASH-INSERT(T, x)
```
LIST-INSERT(T[h(x.key)], x)
```

CHAINED-HASH-DELETE(T, x)
```
LIST-DELETE(T[h(x.key)], x)
```

- We assume that $O(1)$ time suffices to compute the hash value $h(k)$ for any key k
- Worst-Case Complexity for n keys: All the keys are mapped to the same slot
  - Insertion: $O(1)$ if we assume that the key k is not already in the hash table.
  - Deletion: $O(1)$ in the case of doubly linked list and $O(n)$ in the case of singly linked list.
  - Searching: $O(n)$.

Chaining: Implementing the Operations

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

CHAINED-HASH-INSERT(T, x)
LIST-INSERT($T[h(x.key)]$, x)

CHAINED-HASH-DELETE(T, x)
LIST-DELETE($T[h(x.key)]$, x)

- We assume that $O(1)$ time suffices to compute the hash value $h(k)$ for any key k
- Worst-Case Complexity for n keys: All the keys are mapped to the same slot
  - Insertion: $O(1)$ if we assume that the key k is not already in the hash table.
  - Deletion: $O(1)$ in the case of doubly linked list and $O(n)$ in the case of singly linked list.
  - Searching: $O(n)$.
- Best-Case Complexity is always $O(1)$.

Chaining: Average-Case Complexity (1)

- For searching for a key k is the hash table, we have two cases:
  - Successful search: The hash table does contain an element with key k
  - Unsuccessful search: The hash table contains no element with key k
- Analysis is in terms of the load factor $\alpha = \frac{n}{m}$:
  - n is the number of keys stored in the hash table
  - m is the number slots in the hash table
- Load factor is the average number of elements per linked list

Chaining: Average-Case Complexity (2)

- Assumptions
  - Simple uniform hashing: any key is equally likely to hash (i.e., be mapped) to any of the m slots. Formally,
    - For any given key $k \in U$, the probability of $h(k)$ is equal to i is $\frac{1}{m}$, for any slot $i \in \{1, \ldots, m\}$
  - $O(1)$ time suffices to compute $h(k)$
- $\Rightarrow$ Average-case complexity
  - Successful search: $\Theta(1 + \alpha)$
  - Unsuccessful search: $\Theta(1 + \alpha)$
- If $n = O(m)$ then searching takes $O(1)$

Chaining: Average-Case Complexity (3)

Unsuccessful Search: The hash table contains no element with key k

- To search unsuccessfully, we need to traverse the whole list $T[h(k)]$
- The expected (i.e.,average) length of the list $T[h(k)]$ is α
- $\Rightarrow$ The expected number of elements examined is α

Chaining: Average-Case Complexity (3)

Unsuccessful Search: The hash table contains no element with key k

- To search unsuccessfully, we need to traverse the whole list $T[h(k)]$
- The expected (i.e.,average) length of the list $T[h(k)]$ is α
- $\Rightarrow$ The expected number of elements examined is α

Successful Search: The hash table contains an element with key k

- On average, we will find the key half-way through the list $T[h(k)]$
- $\Rightarrow$ The expected number of elements examined is $\approx \alpha/2$

Chaining: Average-Case Complexity (3)

Unsuccessful Search: The hash table contains no element with key k

- To search unsuccessfully, we need to traverse the whole list $T[h(k)]$
- The expected (i.e.,average) length of the list $T[h(k)]$ is α
- $\Rightarrow$ The expected number of elements examined is α

Successful Search: The hash table contains an element with key k

- On average, we will find the key half-way through the list $T[h(k)]$
- $\Rightarrow$ The expected number of elements examined is $\approx \alpha/2$

Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$ in both cases.

How to Choose Hash Functions?

- Ideally, the hash function should:
  - Be easy to compute $O(1)$
  - Satisfy the simple uniform hashing assumption
- The second assumption is very hard to achieve:
  - The probability distribution of keys is unknown
  - The keys may be not be drawn independently
- In practice, we use heuristics, based on the domain of the keys, to create a hash function that performs well.

Keys are Natural Numbers

- Hash functions assume that the keys are natural numbers.
- If it is not the case, we need to interpret them as natural numbers

Hash Functions: The Division Method

- The hash function maps a key k into one of the m slots by taking the remainder of k divided by m. That is, the hash function is:

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

If $m = 12$ and $k = 100$, then $h(k) = 4$

Hash Functions: The Division Method

- The hash function maps a key k into one of the m slots by taking the remainder of k divided by m. That is, the hash function is:

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

  If $m = 12$ and $k = 100$, then $h(k) = 4$

- Advantage: Simple!

Hash Functions: The Division Method

- The hash function maps a key k into one of the m slots by taking the remainder of k divided by m. That is, the hash function is:

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

  If $m = 12$ and $k = 100$, then $h(k) = 4$

- Advantage: Simple!

- Disadvantage: Have to avoid certain values of m

Hash Functions: The Division Method

- The hash function maps a key k into one of the m slots by taking the remainder of k divided by m. That is, the hash function is:

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

  If $m = 12$ and $k = 100$, then $h(k) = 4$

- Advantage: Simple!
- Disadvantage: Have to avoid certain values of m
  - Bad choice: If $m = 2^p$ for some natural number p, then $h(k)$ is the p lowest-order bits of k, and thus not dependent on the whole key.

Hash Functions: The Division Method

- The hash function maps a key k into one of the m slots by taking the remainder of k divided by m. That is, the hash function is:

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

  If $m = 12$ and $k = 100$, then $h(k) = 4$

- Advantage: Simple!

- Disadvantage: Have to avoid certain values of m
  - Bad choice: If $m = 2^p$ for some natural number p, then $h(k)$ is the p lowest-order bits of k, and thus not dependent on the whole key.
  - Good choice: A prime not close to 2^p. Example $m = 1511$ since $2^{10} = 1024 < 1511 < 2^{11} = 2048$

Hash Functions: The Multiplication Method

- The multiplication method for creating hash functions operates as follows:
  - Choose constant A in the range $0 < A < 1$.
  - Multiply the key k by A.
  - Multipy the fractional part by m.
  - Take the floor of the result.
- In short, the hash function is: $h(k) = \lfloor m(kA \, mod \, 1) \rfloor$

Hash Functions: The Multiplication Method

- The multiplication method for creating hash functions operates as follows:
  - Choose constant A in the range $0 < A < 1$.
  - Multiply the key k by A.
  - Multipy the fractional part by m.
  - Take the floor of the result.
- In short, the hash function is: $h(k) = \lfloor m(kA \, mod \, 1) \rfloor$
- Advantage: Value of m is not critical for the distribution of keys (but the value of A is).

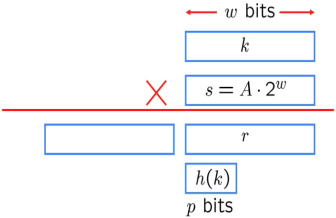
Hash Functions: The Multiplication Method

- The multiplication method for creating hash functions operates as follows:
  - Choose constant A in the range $0 < A < 1$.
  - Multiply the key k by A.
  - Multipy the fractional part by m.
  - Take the floor of the result.
- In short, the hash function is: $h(k) = \lfloor m(kA \, mod \, 1) \rfloor$
- Advantage: Value of m is not critical for the distribution of keys (but the value of A is).
- $A = \frac{\sqrt{5}-1}{2} = 0.61803\ldots$ is often a good choice.

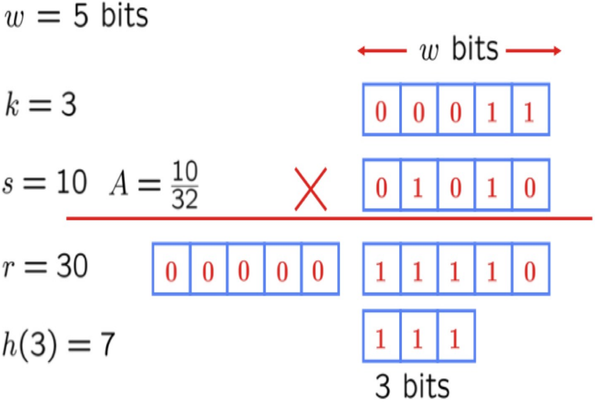
Hash Functions: The Multiplication Method

- The multiplication method for creating hash functions operates as follows:
  - Choose constant A in the range $0 < A < 1$.
  - Multiply the key k by A.
  - Multipy the fractional part by m.
  - Take the floor of the result.
- In short, the hash function is: $h(k) = \lfloor m(kA \, mod \, 1) \rfloor$
- Advantage: Value of m is not critical for the distribution of keys (but the value of A is).
- $A = \frac{\sqrt{5}-1}{2} = 0.61803\ldots$ is often a good choice.
- Advantage: Faster than the division method if m is a power of 2 (and may be for other m as well).

- Choose $m = 2^p$
- Assume that the encoding of the keys takes w bits
- A is of the form $\frac{s}{2^w}$ with $0 < s < 2^w$ (s takes w bits)
- Compute $t = k \cdot s = k \cdot A \cdot 2^w$
- Let r be the first w bits of t
- $h(k)$ is the last p bits of r



The Multiplication Method: Example 1



Collision Resolution by Open Addressing

- An alternative to chaining for handling collisions.
- All the n-elements are stored inside the m-slot hash table itself (rather than in linked lists outside the table)
- It works only when the load factor $\alpha = \frac{n}{m}$ satisfies $\alpha \leq 1$
- To perform insertion of an element, we systematically examine (or probe), the hash table until we find an empty slot to put this element.
- The sequence of slots to be probed depends upon the key being inserted.

Open Addressing: Probe Sequence

- We extend the hash function to include the probe number (starting from 0) as second input. Thus the hash function becomes:

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

- The number i in $h(k, i)$ is called the probe number.
- For every key k, the sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ is called the probe sequence of k
- The probe sequence should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, (i.e., every slot in the hash table is eventually considered).

Open Addressing: Probing Functions

Given an ordinary hash function $h' : U \to \{0, 1, \dots, m-1\}$, we define

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

for every $i \in \{0, 1, \dots, m-1\}$, and f is called the probing function

- Some examples:
  - Linear probing: $f(i) = i$.
  - Quadratic probing: $f(i) = c_1 \cdot i + c_2 \cdot i^2$ where $c_2 \neq 0$.
  - Double hashing: $f(i) = i \cdot h''(k)$ where h'' is another ordinary hash function.

Open Addressing: Pseudocode for Insertion

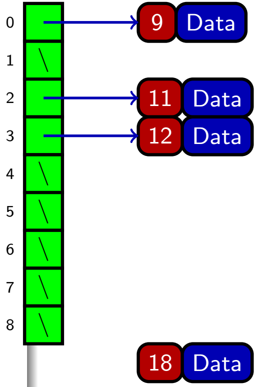
To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k, i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k, i)]$ to x
- If $i = m$ then return "hash table overflow"

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"

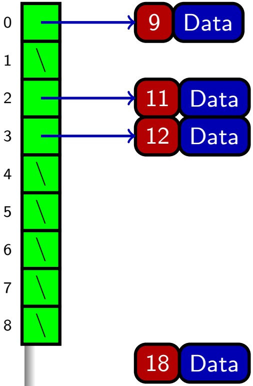


Linear Probing: $\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\, 9}$ $\boxed{h'(k) = k\, mod\, 9}$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing:

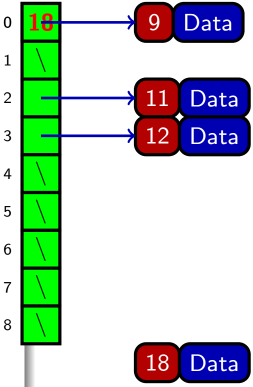
$$\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\, 9}$$

$$\boxed{h'(18) = 18\, mod\, 9 = 0}$$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing:

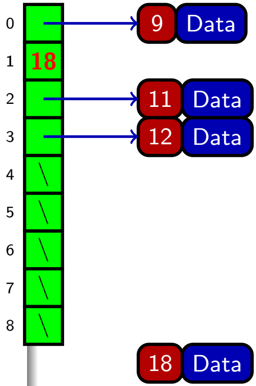
$$\boxed{h(k,i) = \left(h'(k) + i\right) mod\, 9}$$

$$\boxed{h'(18) = 18\; mod\, 9 = 0}$$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



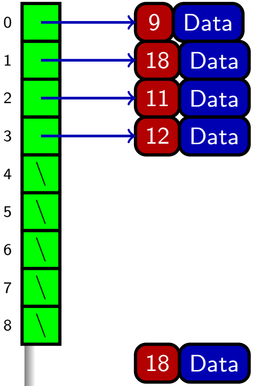
Linear Probing: $\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\; 9}$

$\boxed{h'(18) = 18\; mod\; 9 = 0}$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing:

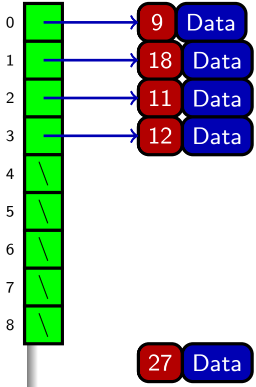
$$\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\ 9}$$

$$\boxed{h'(18) = 18\ mod\ 9 = 0}$$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing:

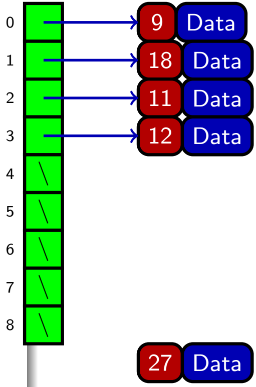
$$\boxed{h(k,i) = \big(h'(k) + i\big) \, mod \, 9}$$

$$\boxed{h'(k) = k \, mod \, 9}$$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
    - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing:

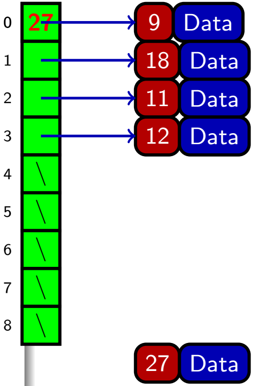
$$\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\, 9}$$

$$\boxed{h'(27) = 27\, mod\, 9 = 0}$$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing:

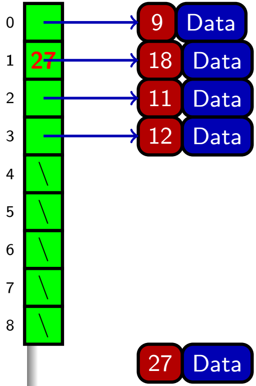
$$\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\, 9}$$

$$\boxed{h'(27) = 27\, mod\, 9 = 0}$$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k, i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k, i)]$ to x
- If $i = m$ then return "hash table overflow"



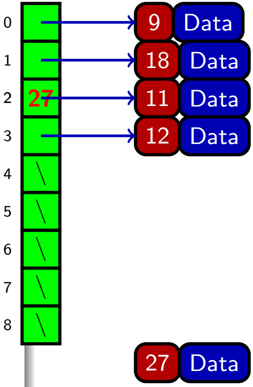
Linear Probing: $\boxed{h(k, i) = \big(h'(k) + i\big)\, mod\, 9}$

$\boxed{h'(27) = 27\, mod\, 9 = 0}$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



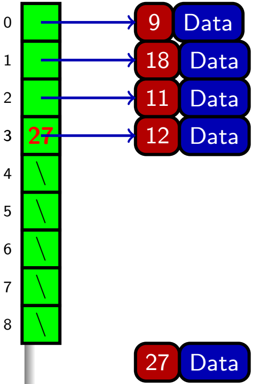
Linear Probing: $\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\, 9}$

$\boxed{h'(27) = 27\, mod\, 9 = 0}$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing:

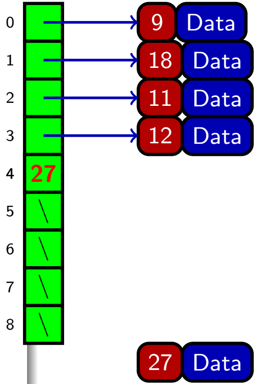
$$\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\; 9}$$

$$\boxed{h'(27) = 27\; mod\; 9 = 0}$$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



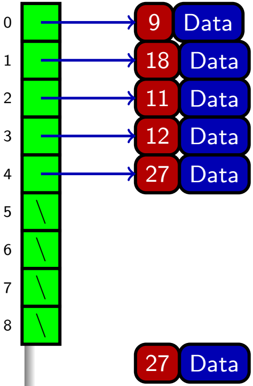
Linear Probing: $\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\, 9}$

$\boxed{h'(27) = 27\, mod\, 9 = 0}$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"



Linear Probing: $\boxed{h(k,i) = \big(h'(k) + i\big)\, mod\, 9}$

$\boxed{h'(27) = 27\ mod\ 9 = 0}$

Open Addressing: Pseudocode for Insertion

To insert an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$ or $T[h(k,i)] = NIL$
  - $i \leftarrow i + 1$
- If $i < m$ then set $T[h(k,i)]$ to x
- If $i = m$ then return "hash table overflow"

```
HASH-INSERT(T, x)
i ← 0
repeat
        j ← h(x.key, i)
        if T[j] = NIL
           then T[j] ← x
                return j
           else i ← i + 1
   until i = m
error "hash table overflow"
```

Open Addressing: Pseudocode for Searching

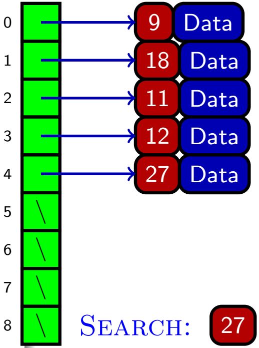
To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k, i)] = NIL$ or $T[h(k, i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k, i)].key = k$ then return $T[h(k, i)]$, otherwise return NIL

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
    - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

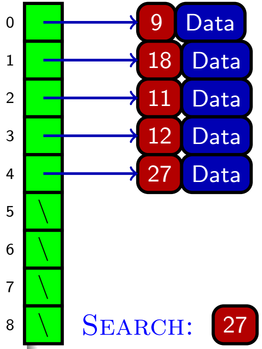
$$h(k,i) = \big(h'(k) + i - 1\big)\, mod\ 9$$

$$h'(k) = k\ mod\ 9$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

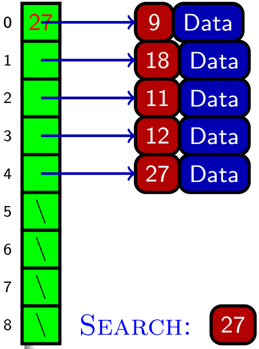
$$h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9$$

$$h'(27) = 27\, mod\, 9 = 0$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].\,key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].\,key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

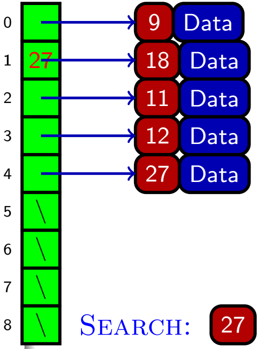
$$\boxed{h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9}$$

$$\boxed{h'(27) = 27\, mod\, 9 = 0}$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

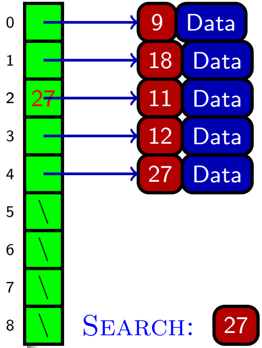
$$h(k,i) = \big(h'(k) + i - 1\big)\, mod\; 9$$

$$h'(27) = 27\; mod\; 9 = 0$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

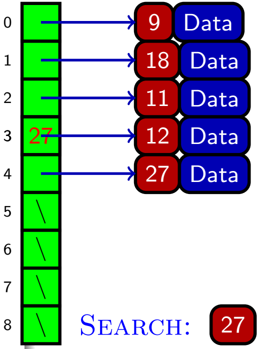
$$h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9$$

$$h'(27) = 27\, mod\, 9 = 0$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k, i)] = NIL$ or $T[h(k, i)].key = k$
    - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k, i)].key = k$ then return $T[h(k, i)]$, otherwise return NIL



Linear Probing:

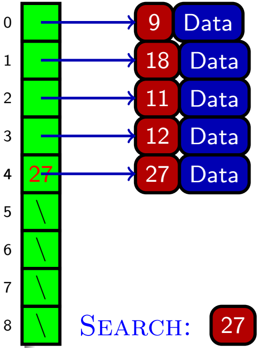
$$\boxed{h(k, i) = \big(h'(k) + i - 1\big)\, mod\, 9}$$

$$\boxed{h'(27) = 27\, mod\, 9 = 0}$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

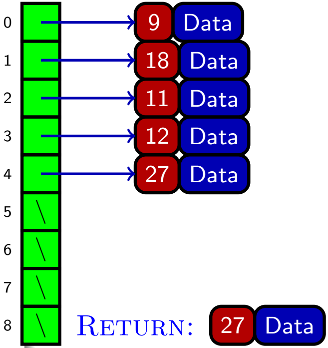
$$\boxed{h(k,i) = \big(h'(k) + i - 1\big) \, mod\, 9}$$

$$\boxed{h'(27) = 27 \, mod\, 9 = 0}$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

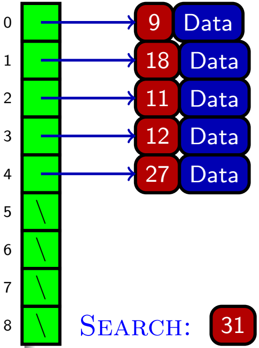
$$h(k,i) = \big(h'(k) + i - 1\big) \, mod \, 9$$

$$h'(27) = 27 \, mod \, 9 = 0$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

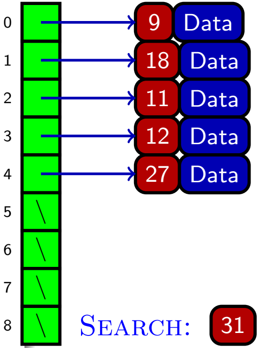
$$h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9$$

$$h'(k) = k\, mod\, 9$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

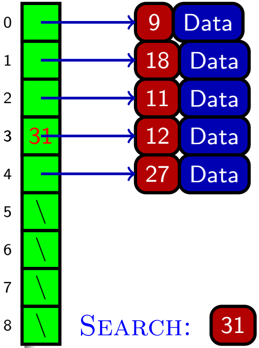
$$\boxed{h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9}$$

$$\boxed{h'(31) = 31\, mod\, 9 = 4}$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

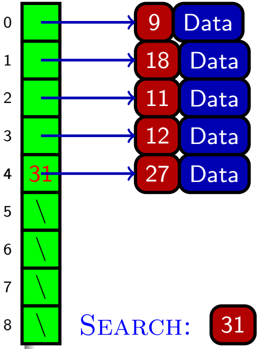
$$\boxed{h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9}$$

$$\boxed{h'(31) = 31\, mod\, 9 = 4}$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

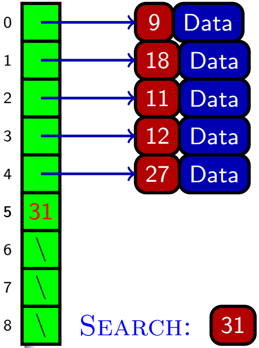
$$\boxed{h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9}$$

$$\boxed{h'(31) = 31\, mod\, 9 = 4}$$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



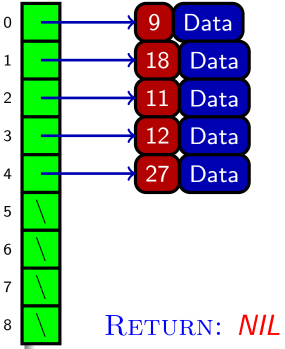
Linear Probing: $h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9$

$h'(31) = 31\, mod\, 9 = 4$

Open Addressing: Pseudocode for Searching

To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL



Linear Probing:

$$\boxed{h(k,i) = \big(h'(k) + i - 1\big)\, mod\, 9}$$

$$\boxed{h'(31) = 31\, mod\, 9 = 4}$$

Open Addressing: Pseudocode for Searching

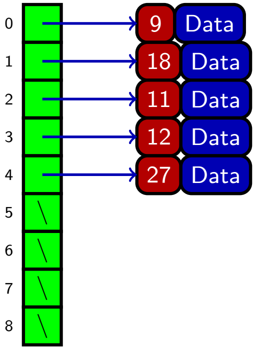
To search for an element x with a key k:

- Initialize $i = 0$
- Repeat until $i = m$, $T[h(k,i)] = NIL$ or $T[h(k,i)].key = k$
  - $i \leftarrow i + 1$
- If $i < m$ and $T[h(k,i)].key = k$ then return $T[h(k,i)]$, otherwise return NIL

```
HASH-SEARCH(T, k)
i ← 0
repeat
        j ← h(k, i)
        if T[j].key = k
          then return T[j]
        i ← i + 1
    until T[j] = NIL or i = m
return NIL
```

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
  - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted



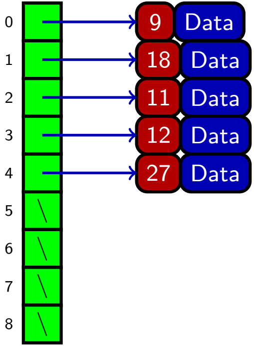
Linear Probing:

$$\boxed{h(k,i) = \left(h'(k) + i\right) \mathit{mod}\ 9}$$

$$\boxed{h'(k) = k\ \mathit{mod}\ 9}$$

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
  - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted
  - Assume that we delete the element at the slot number 2



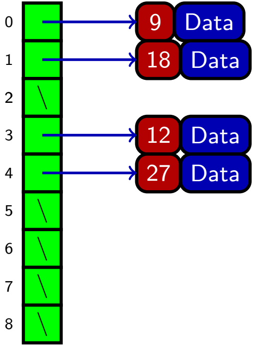
Linear Probing:

$$h(k,i) = \left(h'(k) + i\right) \bmod 9$$

$$h'(k) = k \bmod 9$$

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
    - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted
    - Assume that we delete the element at the slot number 2



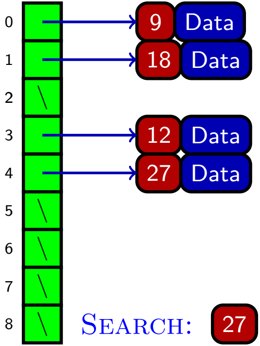
Linear Probing:

$$\boxed{h(k,i) = \left(h'(k) + i\right) \mathit{mod}\ 9}$$

$$\boxed{h'(k) = k\ \mathit{mod}\ 9}$$

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
  - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted
  - Assume that we delete the element at the slot number 2
  - The search of the key 27 would be unsuccessful



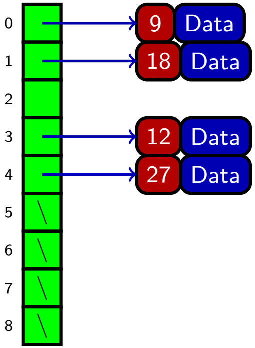
Linear Probing:

$$\boxed{h(k,i) = \left(h'(k) + i\right) \bmod 9}$$

$$\boxed{h'(k) = k \bmod 9}$$

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
  - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted
  - Assume that we delete the element at the slot number 2
  - The search of the key 27 would be unsuccessful
- Solution: Use a special symbol $\perp$ instead of NIL when marking a slot as empty during deletion



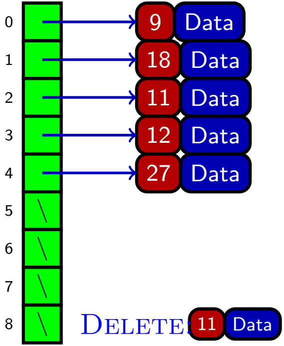
Linear Probing:

$$h(k,i) = \left(h'(k) + i\right) \textit{ mod } 9$$

$$h'(k) = k \textit{ mod } 9$$

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
  - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted
  - Assume that we delete the element at the slot number 2
  - The search of the key 27 would be unsuccessful
- Solution: Use a special symbol $\perp$ instead of NIL when marking a slot as empty during deletion



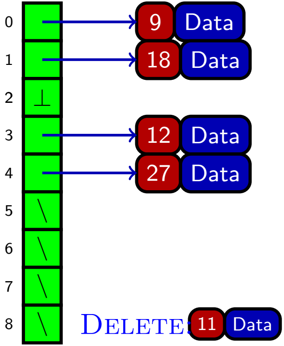
Linear Probing:

$$h(k, i) = (h'(k) + i) \bmod 9$$

$$h'(k) = k \bmod 9$$

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
  - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted
  - Assume that we delete the element at the slot number 2
  - The search of the key 27 would be unsuccessful
- Solution: Use a special symbol $\perp$ instead of NIL when marking a slot as empty during deletion



Linear Probing:

$$h(k,i) = (h'(k) + i) \bmod 9$$

$$h'(k) = k \bmod 9$$

Open Addressing: Deletion

- Deletion from an open-address hash table is more difficult
  - Avoid the use of open addressing when we plan to delete many keys.
- Cannot just put *NIL* into the slot containing the element to be deleted
  - Assume that we delete the element at the slot number 2
  - The search of the key 27 would be unsuccessful
- Solution: Use a special symbol $\perp$ instead of NIL when marking a slot as empty during deletion

HASH-DELETE(T, x)

```
i ← 0
repeat
        j ← h(x.key, i)
        if T[j] = x
           then T[j] ← ⊥
                return j
        i ← i + 1
   until T[j] = NIL or i = m
return NIL
```

Open Addressing: Observations

- Ideally, the hash function satisfies the uniform hashing:
    - Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence.
    - Hard to implement true uniform hashing.
    - In practice, we approximate it with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$:
        - Linear probing
        - Quadratic probing
        - Double hashing
- Open addressing avoids the overheads of linked lists, but is more sensitive to the load factor α (which must always be ≤ 1).