Home

DATA STRUCTURES Algorithms

WEEK 04: LISTS AND HASH TABLES (LECTURE)

Michael T. Gastner

2026-01-26

Home

INTRO

Home

LEARNING OBJECTIVES

By the end of this lecture, you should be able to:

1. Define key concepts: linked list, set, dictionary, direct-address table, and hash table.
2. Explain search, insert, and delete operations for doubly-linked lists and hash tables.
3. State the running times of these operations in Θ-notation.
4. Determine the state of a hash table given a set of keys and a specific hash function.
5. Understand the causes of collisions in hash tables and their implications.
6. Compare and contrast collision resolution techniques: chaining and open addressing.

Home

MOTIVATION

In the random-access machine model, arrays provide constant-time access to elements at any given index. However, checking whether a specific value exists in an **unsorted array** of size n requires scanning the entire array. This process has a worst-case running time of $\Theta(n)$, which occurs when the value is absent.

Additionally, **inserting** and **deleting** elements in arrays is inefficient if you want to maintain as much of the previous sequence as possible. For instance, inserting an element at the beginning of an array requires shifting all subsequent elements one position to the right, resulting in a worst-case running time of $\Theta(n)$.

In this lecture, we will explore **linked lists** and **hash tables**, which potentially accelerate these operations.

Home

LINKED LISTS

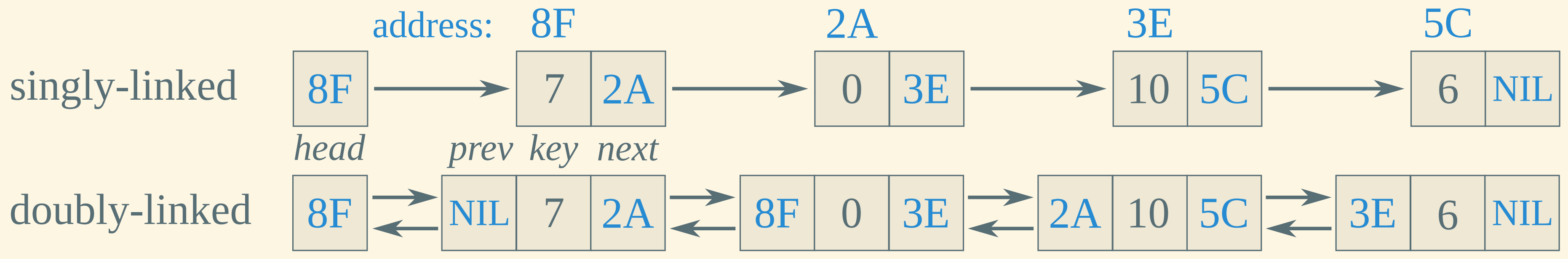
Home

LINKED LISTS

Definition

A **singly linked list** is a data structure where each element contains both a **key** and a **pointer to the next element** in the list. The end of the list is marked by a special pointer, NIL. Additionally, the list possesses a $head$ attribute, which points to the first element in the list.

A **doubly linked list** extends this structure by adding either a **pointer to the previous element** or NIL for the first element.



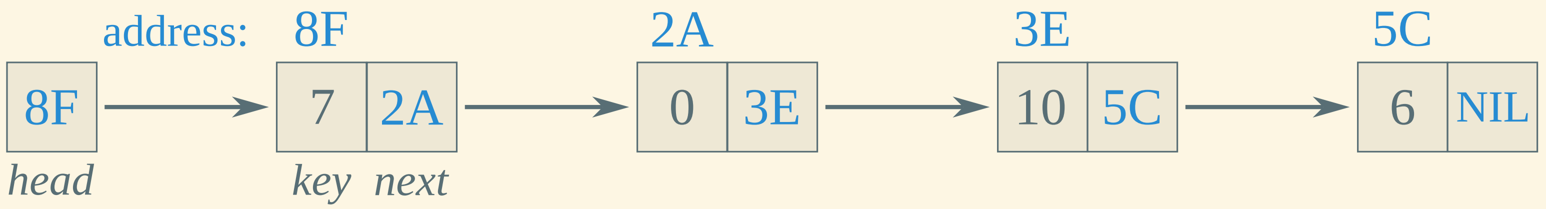
Home

SEARCHING A LINKED LIST

To search a linked list, start at the **head** and traverse the list by following the **pointers** from one element to the next. The traversal continues until either the desired **key** is found or the end of the list is reached. If the key is found, a pointer to the key is returned; otherwise, the procedure returns NIL.

```
procedure LIST-SEARCH($L$, $k$)
    $x = L.head$
    while $x \neq$ NIL and $x.key \neq k$ do
        $x = x.next$
    return $x$
```

In the example below, SEARCH$(L, 10)$ returns 3E.



Home

INSERTING INTO A LINKED LIST

When inserting a new key into a linked list, two cases must be distinguished:

1. List-Prepend(L, x):

   The new key, pointed to by x, is inserted at the **beginning** of the list.
2. List-Insert(x, y):

   The new key is to be inserted **after an existing key**. Here, x is assumed to be a pointer to the new key and y a pointer to the existing key. The list L is not a parameter of List-Insert because only the existing list element y is required as input, not the entire list.

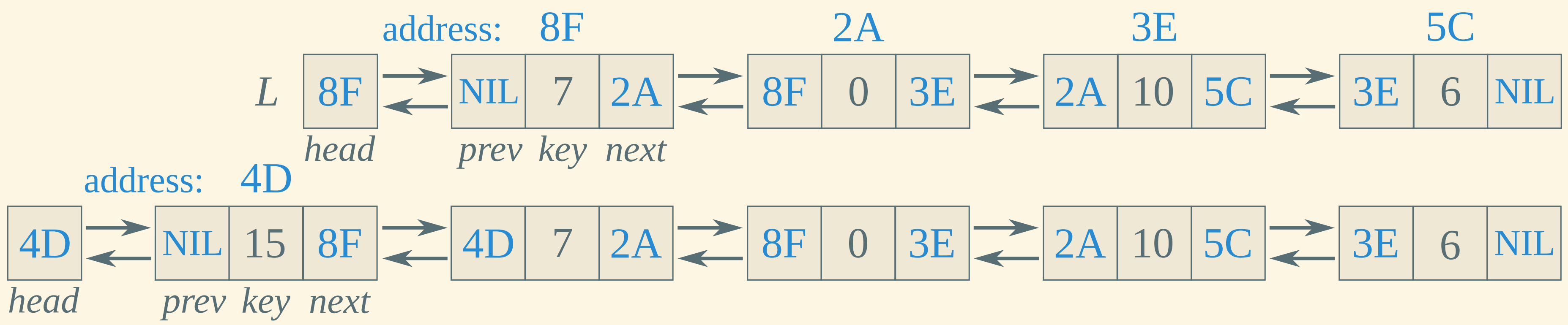
The procedures for both cases are detailed on the following slides, where the list is assumed to be **doubly-linked.**

Home

CASE 1: PREPENDING AN ELEMENT TO A LINKED LIST

```
procedure LIST-PREPEND($L$, $x$)
    $x.next = L.head$
    $x.prev =$ NIL
    if $L.head \neq$ NIL then
        $L.head.prev = x$
    $L.head = x$
```

The example below presents the result of calling LIST-PREPEND(L, x), where x points to the address 4D and $x.key = 15$:

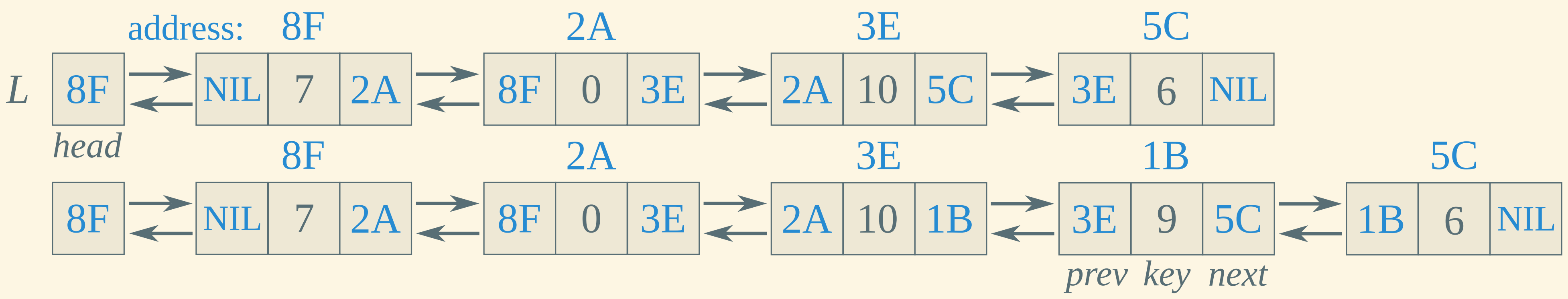


Home

CASE 2: INSERTING AN ELEMENT AFTER AN EXISTING ONE

```
1: procedure LIST-INSERT($x, y$)
2:     $x.next = y.next$
3:     $x.prev = y$
4:     if $y.next \neq$ NIL then
5:         $y.next.prev = x$
6:     $y.next = x$
```

The example below illustrates the result of calling LIST-INSERT(x, y), where x points to the address 1B, $x.\,key = 9$, and y is the element at the address 3E:



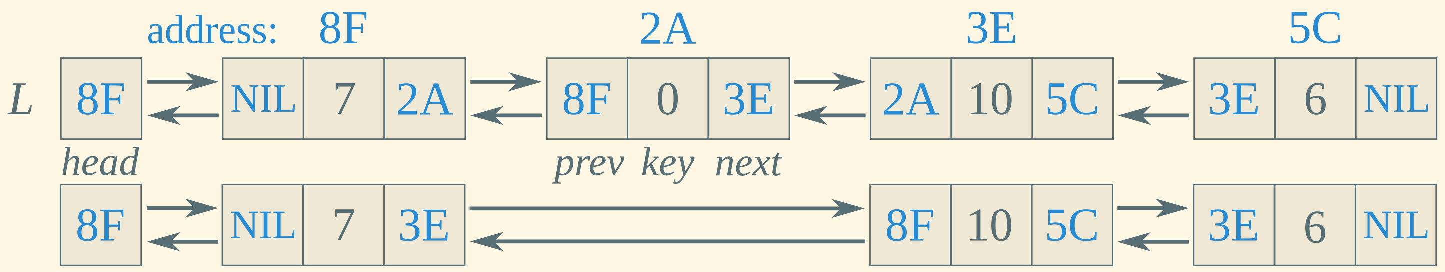
Home

DELETING FROM A DOUBLY-LINKED LIST

The following procedure removes the element pointed to by x from the list L:

```
procedure LIST-DELETE($L$, $x$)
    if $x.prev \neq$ NIL then
        $x.prev.next = x.next$
    else
        $L.head = x.next$
    if $x.next \neq$ NIL then
        $x.next.prev = x.prev$
```

The example below demonstrates the result of calling LIST-DELETE(L, x), where x points to the address 2A:



Home

RUNNING TIMES OF DOUBLY-LINKED LIST OPERATIONS

Operation	Worst-Case Running Time	Reason
List-Search	$\Theta(n)$	Must examine all elements if key is not in the list.
List-Prepend List-Insert List-Delete	$\Theta(1)$	Involves only pointer updates. Note that the pointer to the element to be inserted or deleted must be provided as an argument. If only the key is known, a $\Theta(n)$ search must be performed first.

Compared to an unsorted array, a doubly-linked list does not reduce the asymptotic growth rate of the search operation in the worst case. However, insertions and deletions are faster than the $\Theta(n)$ time required for an array.

Home

SETS AND DICTIONARIES

Home

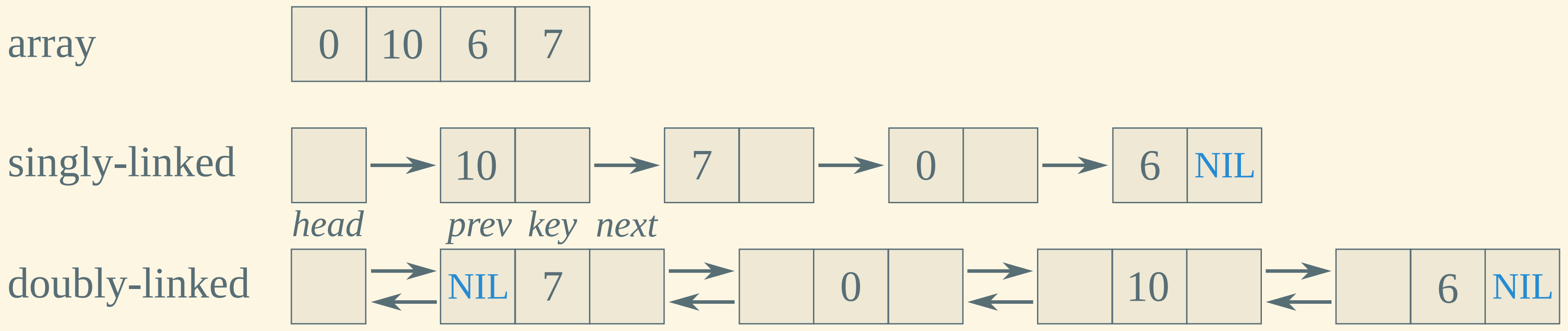
SETS

Both arrays and linked lists can be used to implement **sets:**

Definition

A **set** is an **unordered** collection of **unique elements.**

For instance, the arrays and linked lists shown below represent the same set, $\{0, 6, 7, 10\}$. The order of the elements is **arbitrary** and does not affect the representation of the set.



Home

DICTIONARIES

Definition

A **dictionary** is a data structure that stores a **set** of elements, referred to as **keys**, and supports the following operations:

- **Search:** Check whether a given key exists in the set.
- **Insert:** Add a new key to the set.
- **Delete:** Given a pointer to the storage location of a key, remove it from the set.

Because keys constitute a set, they must be **unique.**

SATELLITE DATA

Definition

Satellite data are objects associated with keys in a dictionary. These associations remain unchanged during any dictionary operation.

Examples:

- A phone book is a dictionary where the keys are names, and the satellite data are phone numbers.
- Health records can be stored in a dictionary where the keys are numeric patient identifiers and the satellite data include names, birthdays, and medical histories.

Home

SEARCHING UNSORTED ARRAYS IS SLOW

In principle, an unsorted array can be used as a dictionary:

- **Search:** Scan the array for the key. This operation requires $\Theta(n)$ time in the worst case, where n is the array size, because the key may not be in the array.
- **Insert:** Append the new key to the end of the array. This operation takes $\Theta(1)$ time, assuming there is sufficient space at the end of the array.
- **Delete:** If the array index of the key is provided, remove the key by swapping it with the last element and decrementing the array size. This operation also needs $\Theta(1)$ time.

The $\Theta(n)$ worst-case running time of the search operation renders unsorted arrays impractical for large dictionaries.

Home

SORTED ARRAYS ARE ALSO INEFFICIENT DICTIONARIES

By storing keys in a sorted array and applying binary search, the search operation can be improved to $\Theta(\log n)$ time.

However, insertion and deletion are slower than for unsorted arrays—$\Theta(n)$ in the worst case—because the keys must be shifted to maintain the sorted order. For example, if the key to be added or deleted is the minimum in the set, $\Theta(n)$ shifts are required.

Home

LINKED LISTS ARE SLOW FOR DICTIONARY SEARCHES

Linked lists exhibit the same asymptotic growth rates for worst-case running times as unsorted arrays:

- **Search:** $\Theta(n)$
- **Insert:** $\Theta(1)$
- **Delete:** $\Theta(1)$

Sorting the linked list does not improve the search time because the list must still be traversed sequentially from the head to locate the key.

AVERAGE RUNNING TIMES

Assume the keys to be searched or inserted into a dictionary are drawn from identical probability distributions, with each key equally likely to be deleted. Under these assumptions, the average running times of dictionary operations exhibit the same growth rates as their worst-case counterparts:

Operation	Unsorted Array	Sorted Array	Linked List
Search	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Insert	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Delete	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

In summary, arrays and linked lists are suboptimal for implementing dictionaries because at least one of the three dictionary operations—search, insert, or delete—is slow, with running times growing linearly with the dictionary size.

On the following page, we introduce **direct-address tables,** which allow all dictionary operations to run in $O(1)$ time in the worst case.

DIRECT-ADDRESS TABLES

Definition

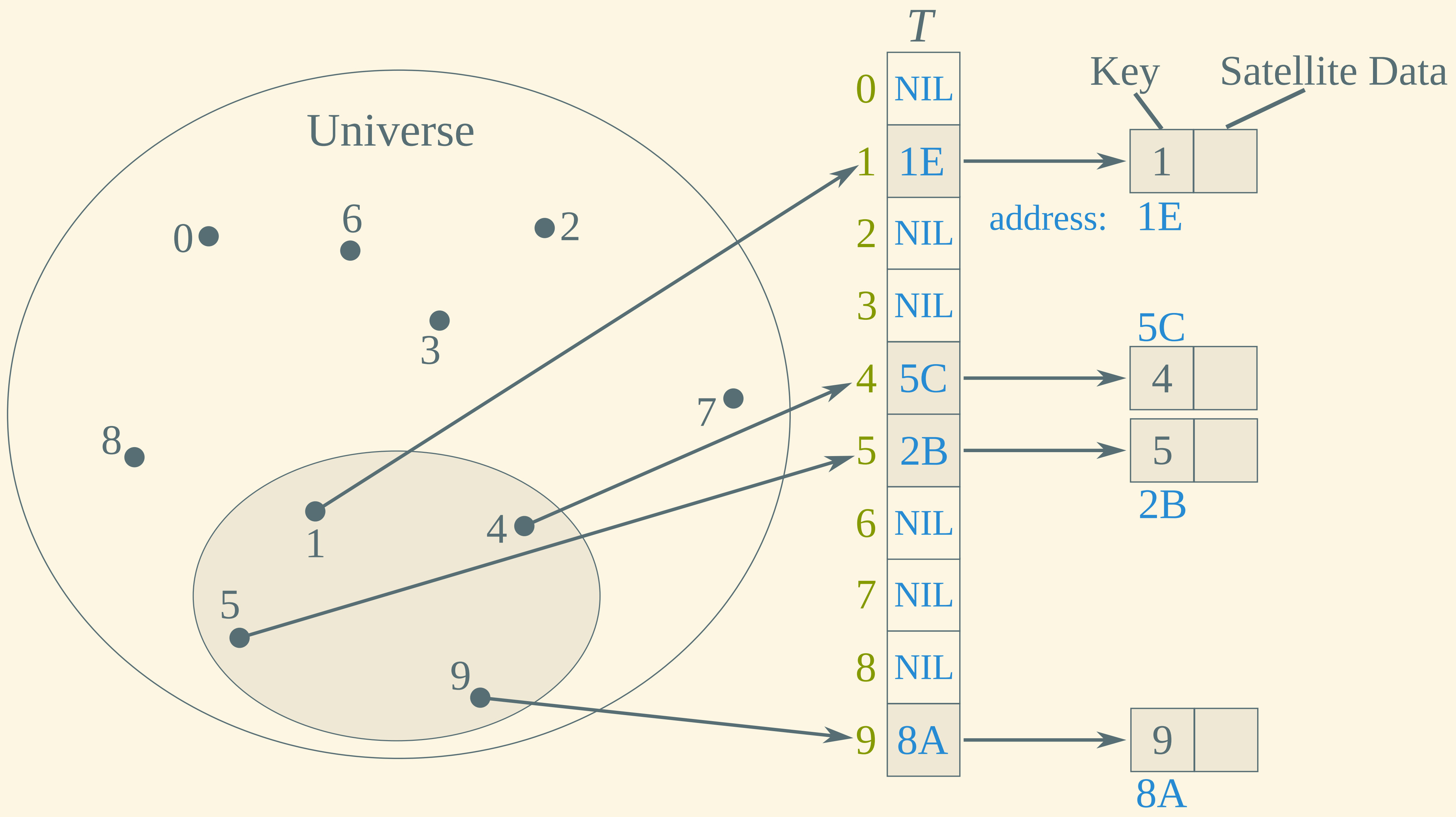
A **direct-address table** T is an array-based data structure that stores a set of keys, each of which is an integer in the range from 0 to $m-1$, where m is the table size. The set of all possible keys $\{0, 1, \ldots, m-1\}$ is called the **universe.**

For each key k in the universe, the table contains a slot $T[k]$ that can store the key and any associated satellite data. If k is not in the set to be stored, $T[k]$ is assigned the value NIL.

The illustration on the following page depicts a direct-address table with a universe of size $m = 10$ and keys in $\{1, 4, 5, 9\}$.

Home

ILLUSTRATION OF DIRECT ADDRESSING



Home

DIRECT-ADDRESS DICTIONARY OPERATIONS

Each of the three dictionary operations can be implemented in $O(1)$ time using direct-address tables:

```
1: procedure DIRECT-ADDRESS-SEARCH($T$, $k$)
2:     return $T[k]$
3: procedure DIRECT-ADDRESS-INSERT($T$, $x$)
4:     $T[x.key] = x$
5: procedure DIRECT-ADDRESS-DELETE($T$, $x$)
6:     $T[x.key] = $ NIL
```

Home

DISADVANTAGES OF DIRECT-ADDRESS TABLES

Although fast, direct-address tables have significant drawbacks:

- If the universe contains a large number of keys, it may be infeasible to store the entire table in memory.
- Typically, the set of keys K that are actually stored is only a small subset of the universe. In such cases, a direct-address table wastes memory.

To address these limitations, we will now introduce **hash tables,** which generally reduce memory requirements to $\Theta(|K|)$ at the expense of an $O(|K|)$ *worst-case* running time for dictionary operations.

However, the *average* running time for hash tables is $O(1)$ under realistic assumptions. Moreover, with well-designed hash functions, it is highly unlikely to experience the worst-case scenario.

HASH TABLES

Home

HASH TABLES

Definition

A **hash table** T is an array-based data structure that stores a set of keys from a universe U by mapping them to array indices using a **hash function** $h : U \to \{0, 1, \ldots, m-1\}$, where m is the array size. Specifically, the key k is stored in the slot $T[h(k)]$.

We say that "the key k **hashes** to slot $h(k)$" and that "$h(k)$ is the **hash value** of key k."

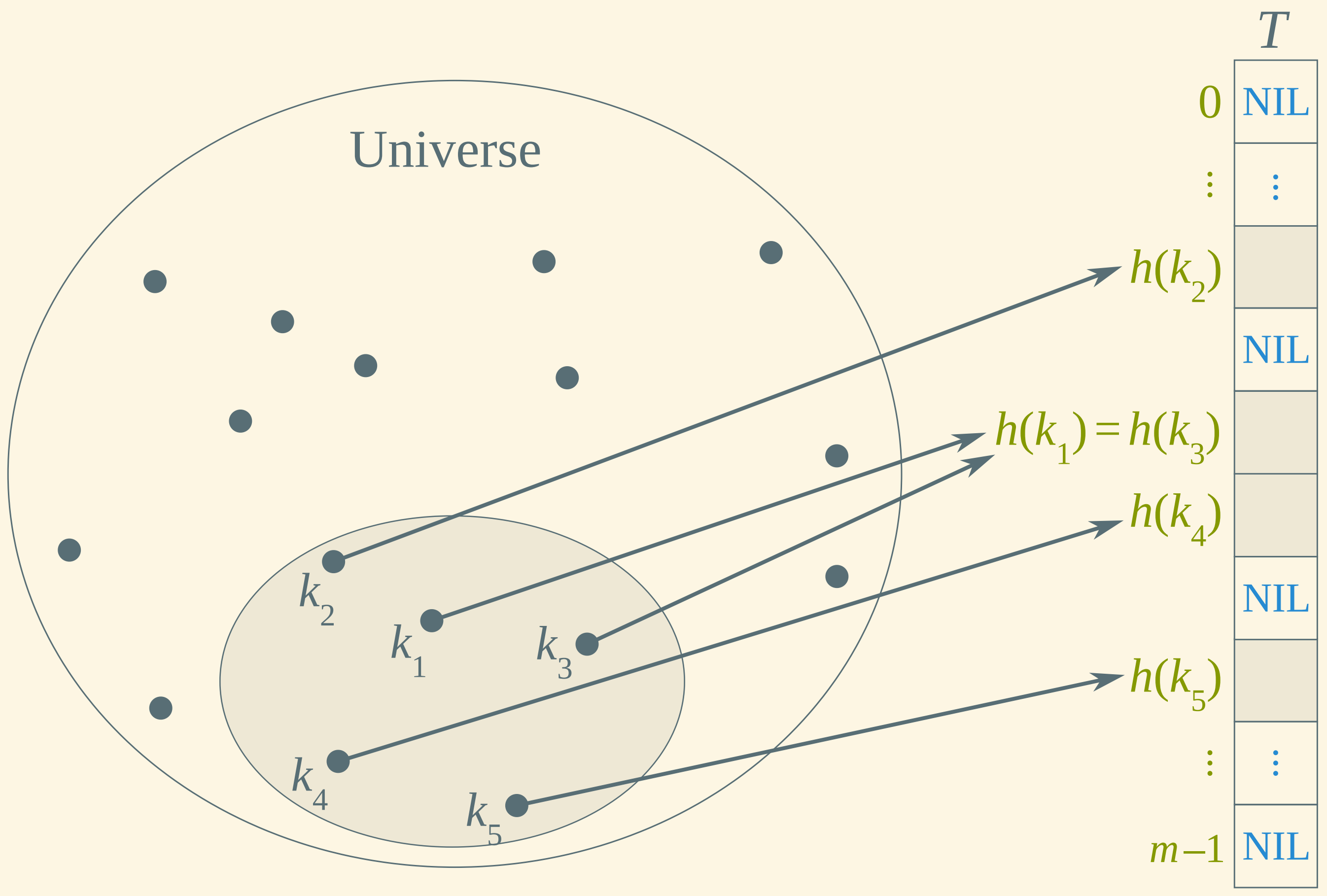
If two distinct keys k_i and k_j hash to the same slot, we encounter a **collision.**

We will assume that h can be computed in $O(1)$ time. A direct-address table is a special case of a hash table where h is the identity function.

Home

ILLUSTRATION OF HASHING

The hash function h maps the keys to the slots in the hash table. In the plot below, k_1 and k_2 hash to the same slot, causing a collision.



INDEPENDENT AND UNIFORM HASHING

Definition

A **hash function** h is said to be **independent and uniform** if it satisfies the following properties:

- For any two distinct keys k_i and k_j, the hash values $h(k_i)$ and $h(k_j)$ are independent.
- For any key k, the probability that $h(k) = i$ is $1/m$ for each i in the range of array indices, $\{0, 1, \ldots, m-1\}$.

Uniform hashing is an idealized model that helps us analyze the expected behavior and key properties of hash tables, such as collision frequency and average running times.

Home

HASH FUNCTIONS USED IN PRACTICE

In practice, hash functions are not independent and uniform. Instead, they are designed to be **computationally efficient** and **deterministic**. However, with careful hash-function design, we can approximate the ideal behavior in practice.

Here are common techniques to generate hash functions:

- Division method
- Multiplication method
- Universal hashing

These techniques will be discussed in the following slides.

Home

DIVISION METHOD

Assume that every key is a nonnegative integer. If necessary, a **surrogate key** can be created by mapping each input key to a unique nonnegative integer (e.g., $\mathrm{A} \to 0$, $\mathrm{B} \to 1$, etc.).

The **division method** generates hash values by applying simple arithmetic to the nonnegative integer key k:

$$h(k) = k \bmod m,$$

where:

- m is the number of slots in the hash table.
- k is the key.
- $\bmod$ represents the modulo operation, which returns the remainder when k is divided by m.

Home

HOW TO CHOOSE THE DIVISOR

To help the division method spread keys more evenly, choose m to be a **prime number.** Using a prime breaks many simple patterns in the keys (for example, when lots of keys share the same last digit or are multiples of a fixed number), which otherwise can cause many keys to land in the same slots. Also avoid choosing m close to a power of 2 (2, 4, 8, 16, ...), since patterns in the low-order bits can then show up directly in $k \bmod m$.

EXAMPLE

Using the division method with $m = 11$, the hash values for the keys 56, 29, 90, 40, 82, 30, and 4 are computed as follows:

$h(56) = 1,\quad h(29) = 7,\quad h(90) = 2,\quad h(40) = 7,\quad h(82) = 5,\quad h(30) = 8,$ and $h(4) = 4$.

Here, the keys 29 and 40 collide because they hash to the same slot: $h(29) = h(40) = 7$.

Home

QUIZ

What is the hash value of the key 100 when using the division method with the divisor $m = 13$?

a. 0

b. 7

c. 9

d. 12

MULTIPLICATION METHOD

The **multiplication method** applies the hash function

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

where:

- m is the number of slots. An integer power of 2 (e.g., $m = 2^{14} = 16,384$) is recommended so that the multiplication with m can be implemented efficiently as a bit shift.
- k is the key.
- A is a constant in the open interval $(0, 1)$. A commonly recommended value is the inverse of the golden ratio: $A = \frac{\sqrt{5}-1}{2}$.
- The $\bmod 1$ operation returns the fractional part of its operand; that is, $x \bmod 1 = x - \lfloor x \rfloor$.

Home

EXAMPLE OF THE MULTIPLICATION METHOD

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

SOLUTION

The hash values are 700, 318, 936, 554 and 172, respectively.

Because m is not an integer power of 2, we cannot use bit shifting to compute the multiplication. Therefore, this hash function is not efficient.

Moreover, all hash values are even numbers, indicating that this hash function does not achieve uniform hashing. Thus, it is unsuitable for practical use.

Home

UNIVERSAL HASHING

Definition

A hashing algorithm is called **universal** if it satisfies the following two conditions:

- The hash function is selected **at random** from a family of hash functions.
- For any two distinct keys k_i and k_j, the probability that $h(k_i) = h(k_j)$ is at most $1/m$.

Universal hashing mitigates worst-case scenarios where many keys hash to the same slot during each execution of the program.

For examples, refer to Section 11.3.4 in Cormen *et al.* (2022).

COLLISION RESOLUTION

Even the best hash functions cannot completely eliminate collisions between keys. There are two common approaches to resolving collisions in hash tables:

Chaining
: Each slot in the hash table contains a pointer to a linked list of all keys that hash to that slot.

Open addressing
: When a collision occurs, the algorithm searches for an alternative slot in the hash table to store the key. The search is guided by a **probe sequence** that determines the order in which slots are examined.

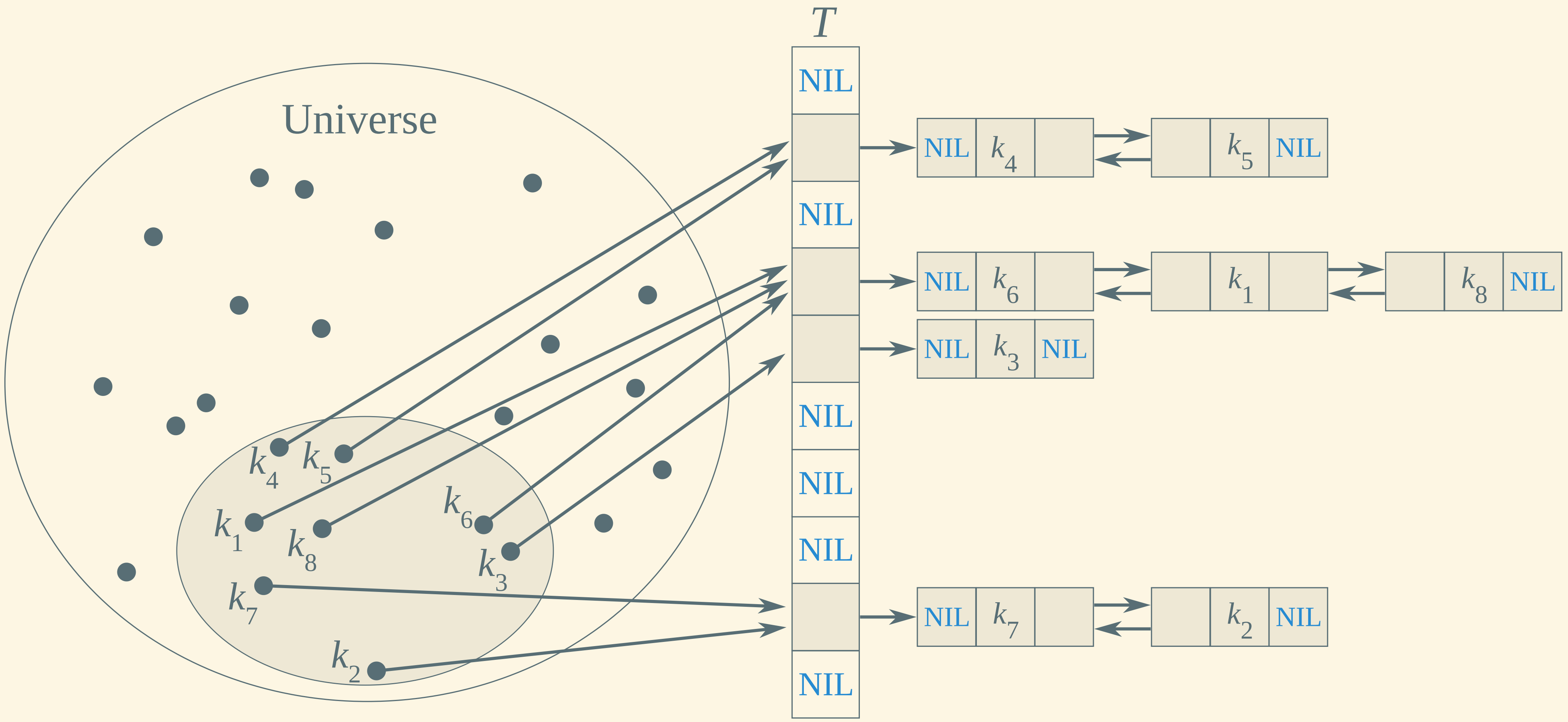
Home

CHAINING

Home

CHAINING

In the example depicted below, the keys k_4 and k_5 collide. Thus, they are stored together with their values in a linked list:



Home

LOAD FACTOR

The following definition is useful for expressing the average running time of dictionary operations when using chaining:

Definition

The **load factor** α is the ratio of the number of keys n stored in the hash table to the number of slots m in the table:

$$\alpha = \frac{n}{m}$$

The load factor α can be interpreted as the average length of a linked list associated with a randomly chosen slot.

Home

DICTIONARY OPERATIONS WHEN USING CHAINING

```
1: procedure CHAINED-HASH-SEARCH($T$, $k$)
2:     return LIST-SEARCH($T[h(k)]$, $k$)
3: procedure CHAINED-HASH-INSERT($T$, $x$)
4:     LIST-PREPEND($T[h(x.key)]$, $x$)
5: procedure CHAINED-HASH-DELETE($T$, $x$)
6:     LIST-DELETE($T[h(x.key)]$, $x$)
```

Operation	Average Running Time	Comments
Chained-Hash-Search	$O(1+\alpha)$	Derived in Section 11.2 of Cormen et al. (2022).
Chained-Hash-Insert	$O(1)$	
Chained-Hash-Delete	$O(1)$	Assuming the list is doubly linked.

Home

OPEN ADDRESSING

Home

OPEN ADDRESSING

While chaining resolves collisions by storing linked lists outside the hash table, **open addressing** stores all keys directly in the slots of the hash table. Each slot contains either a key or NIL.

Unlike chaining, open addressing allows at most one key per slot. Consequently, the load factor α can never exceed 1. If a user attempts to insert a key into a full hash table, an error message must be displayed.

When searching for a key, the algorithm systematically examines table slots until it either finds the desired key or determines that the key is not in the table.

Home

PROBING

To perform insertion using open addressing, we **probe** the hash table until an empty slot is found for the key. The sequence of probes **depends on the key being inserted.**

To determine which slots to probe, the hash function is extended to include the probe number as a second input:

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

The **probe sequence** $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ must be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, ensuring that every hash-table position is eventually considered as a slot for a new key as the table fills up.

INSERTING A KEY IF THERE HAVE BEEN NO DELETIONS

For simplicity, assume that no keys have been deleted from the hash table so far. The HASH-INSERT-WITHOUT-DELETED procedure takes as input a hash table T and a key k.

The procedure either returns the slot number where k is stored or flags an error if the hash table is already full:

```
procedure HASH-INSERT-WITHOUT-DELETED($T$, $k$)
    $i = 0$
    repeat
        $q = h(k, i)$
        if $T[q]$ == NIL then
            $T[q] = k$
            return $q$
        else
            $i = i + 1$
    until $i$ == $m$
    error "hash table overflow"
```

OPEN-ADDRESSING SEARCH FOR A KEY

The algorithm for searching for a key k probes the same sequence of slots that was examined when key k was inserted.

```
procedure HASH-SEARCH$(T, k)$
    $i = 0$
    repeat
        $q = h(k, i)$
        if $T[q]$==$k$ then
            return $q$
        $i = i + 1$
    until $T[q]$==NIL or $i$==$m$
    return NIL
```

DELETING A KEY

Deletion from an open-address hash table is challenging. When a key is deleted from slot q, we cannot simply mark that slot as empty by storing NIL in it. Doing so might prevent the retrieval of any key whose insertion involved probing slot q and finding it occupied.

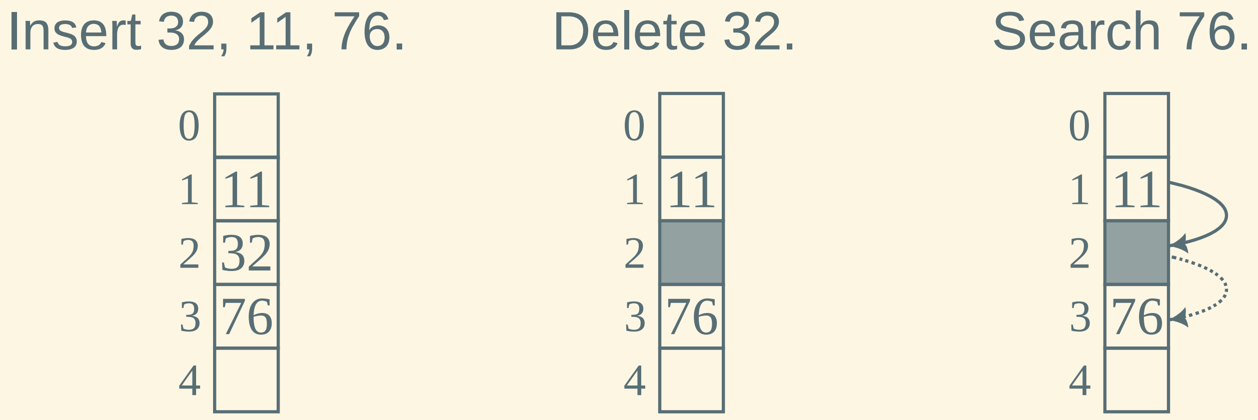
We can solve this problem by marking the slot as DELETED instead of NIL.

ILLUSTRATING THE DELETION OF A KEY

In the figure below, the hash function is assumed to be $h(k, i) = (k + i) \bmod 5$.

If slot 2 is marked as NIL after deleting 32, $\textsc{Hash-Search}(T, 76)$ would return NIL, incorrectly indicating that key 76 is not in the hash table.

However, if slot 2 is marked as DELETED, $\textsc{Hash-Search}(T, 76)$ finds key 76 in slot 3.



INSERTING A KEY IF THERE HAS BEEN A DELETION

If keys have been deleted from the hash table, the insert procedure must also check for slots marked DELETED. Such slots can be reused for new keys:

```
1: procedure HASH-INSERT($T$, $k$)
2:     $i = 0$
3:     repeat
4:         $q = h(k, i)$
5:         if $T[q]$== NIL or $T[q]$== DELETED then
6:             $T[q] = k$
7:             return $q$
8:         else
9:             $i = i + 1$
10:    until $i$ == $m$
11:    error "hash table overflow"
```

The only difference from HASH-INSERT-WITHOUT-DELETED is the additional check for DELETED in line 5.

UNIFORM HASHING

Hash functions used for open addressing should ideally perform **uniform hashing**, defined by two criteria:

1. For every fixed probe number i, the hash function $h(k, i)$ should perform simple uniform hashing, meaning that the hash value $h(k, i)$ of any random key k:
    - is equally likely to hash into any slot $0, 1, \ldots, m-1$.
    - is independent of the hash value of any other key.
2. The probe sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ is:
    - equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1\rangle$.
    - independent of any probe sequence $\langle h(k', 0), h(k', 1), \ldots, h(k', m-1)\rangle$ with $k \neq k'$.

Violating either criterion can cause clustering of keys in certain parts of the hash table, which in turn can degrade the performance of the hash table.

Home

UNIFORM HASHING IS DIFFICULT TO IMPLEMENT

The implementation of true uniform hashing is challenging. Most hash functions that are used in practice do not generate all of the $m!$ possible permutations.

In this lesson, we discuss two methods that guarantee that the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ for each key k:

- **Linear probing,** which generates at most m distinct probe sequences
- **Double hashing,** which generates at most m^2 distinct probe sequences

Home

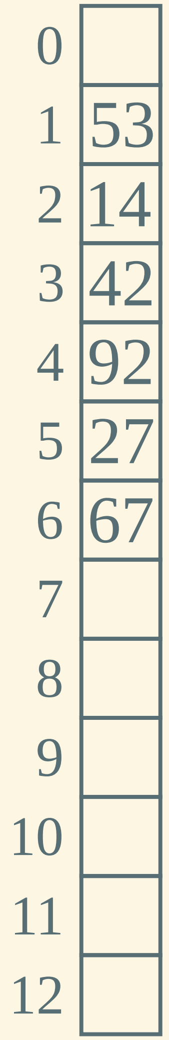
LINEAR PROBING

Given a hash function $h' : U \to \{0, 1, \ldots, m-1\}$, which we refer to as an **auxiliary hash function,** the method of **linear probing** uses the hash function

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

for $i = 0, 1, \ldots, m-1$.

In the example on the right, keys were inserted in the sequence $\langle 42, 53, 14, 92, 27, 67 \rangle$, $h'(k) = k$ and $m = 13$. The result exhibits **clustering,** whereby long sequences of occupied slots are created, leading to an increase in the average search time.



Home

DOUBLE HASHING

Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

where both h_1 and h_2 are auxiliary hash functions.

The value of $h_2(k)$ must be relatively prime to m so that the entire hash table can be searched. This property can be established, for example, in either of the following two ways:

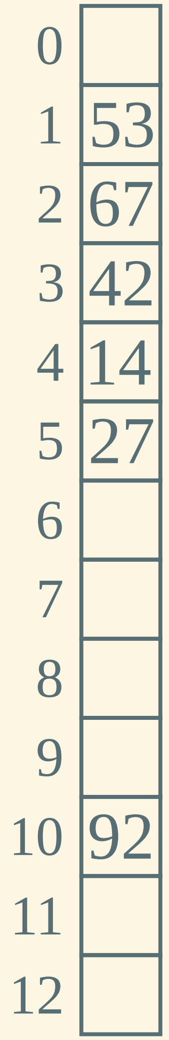
DOUBLE HASHING

In the example on the right,

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

with $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod 12)$. As before, keys were inserted in the sequence $\langle 42, 53, 14, 92, 27, 67 \rangle$. Note that we never had to probe any sequence beyond $i = 1$, which provides evidence that double hashing is less prone to clustering than linear probing.

Although double hashing produces only m^2 out of the $m!$ possible probe sequences, its performance is practically as good as the ideal scheme of uniform hashing.



Home

RUNNING TIME OF SEARCHING

One can show that the average time needed for searching an open-address hash table depends on the load factor $\alpha = n/m$ as follows:

- $O\left(\frac{1}{1-\alpha}\right)$ if the search is unsuccessful.
- $O\left(\frac{1}{\alpha}\log\frac{1}{1-\alpha}\right)$ if the search is successful or a key is inserted,

Note that n includes the number of deleted keys because they still occupy slots in the hash table.

Home

INTERPRETATION OF OPEN ADDRESSING'S ASYMPTOTIC RUNNING TIME

As the hash fills up (i.e. α approaches 1 from below), both of the upper bounds $O\left(\frac{1}{1-\alpha}\right)$ and $O\left(\frac{1}{\alpha}\log\frac{1}{1-\alpha}\right)$ diverge. When α exceeds 1, open-addressing cannot be used for resolving collisions because there is insufficient space in the hash table.

However, if we can guarantee that α never exceeds a constant < 1 (for example, by occasionally resizing the hash table when it becomes too full), then searching only needs $O(1)$ time.

CONCLUSION

Home

SUMMARY OF KEY LEARNING OUTCOMES

1. We defined these data structures:
   a. Linked lists ↪
   b. Sets ↪
   c. Dictionaries ↪
   d. Direct-address tables ↪
   e. Hash tables ↪
2. We explained the following operations:

	Search	**Insert**	**Delete**
Linked list	↪	↪	↪
Hash table (chaining)	↪	↪	↪
Hash table (open addressing)	↪	↪	↪

3. We stated the **running times** of these operations. ↪
4. We determined the state of a hash table given a set of keys and a specific **hash function.** ↪
5. We understood the causes of **collisions** in hash tables and their implications. ↪
6. We compared collision resolution techniques:
   a. Chaining ↪
   b. Open addressing ↪

Home

OUTLOOK

While hash tables are efficient data structures for inserting, searching, and deleting keys, they are not suitable for all applications. For instance, they are not well-suited for finding the smallest or largest key in a set.

Next week, we will discuss **binary search trees,** which are data structures that enable fast retrieval of minimal and maximal keys while still allowing efficient search, insertion, and deletion of keys.

Home

BIBLIOGRAPHY

- Cormen, T.H. *et al.* (2022) *Introduction to algorithms*. 4th ed. MIT Press.