Lecture 7: Dictionaries

Sequential Unfoldment of Knowledge

Wholeness Statement

The Dictionary ADT stores a searchable collection of *key-element* items that represents either an unordered or an ordered collection. Hashing solves the problem of item-lookup by providing a table whose size is not unreasonably large, yet it can store a large range of keys such that the element associated with each key can be accessed quickly ($O(1)$). *Science of Consciousness* provides systematic techniques for accessing and experiencing total knowledge of the Universe to enhance individual life.

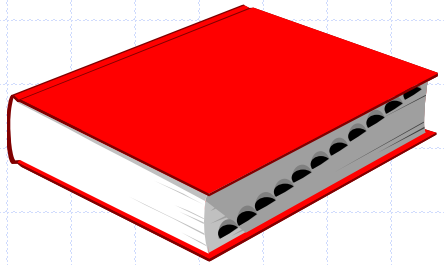
The Dictionary ADT

Two Types of Dictionaries

1. Unordered
2. Ordered

- Both use a key to identify a specific element
- Stores items, i.e., key-element pairs
- For the sake of generality, multiple items can have the same key

Unordered Dictionary ADT



- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)

- Dictionary ADT methods:
  - findElement(k): if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - size(), isEmpty()
  - keys(), elements()

Log Files

- A log file (or audit trail) is a dictionary implemented by means of an unsorted sequence
  - Items are stored in the dictionary in a sequence in arbitrary order
  - Based on doubly-linked lists or a circular array
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - findElement and removeElement take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

Log File

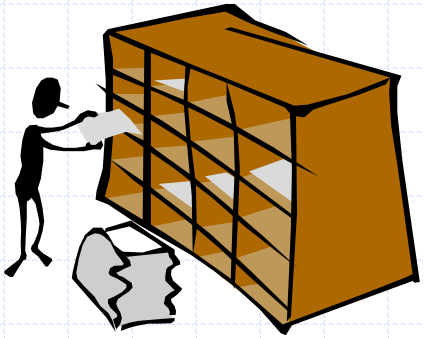
- Effective only for dictionaries of small size or
- For dictionaries on which insertions are the most common operations, while searches and removals are rarely performed

  (e.g., historical record of logins to a workstation)

- What do we do if we need to do frequent searches and removals in a large dictionary?

Hash Tables

Hash Tables and Hash Functions



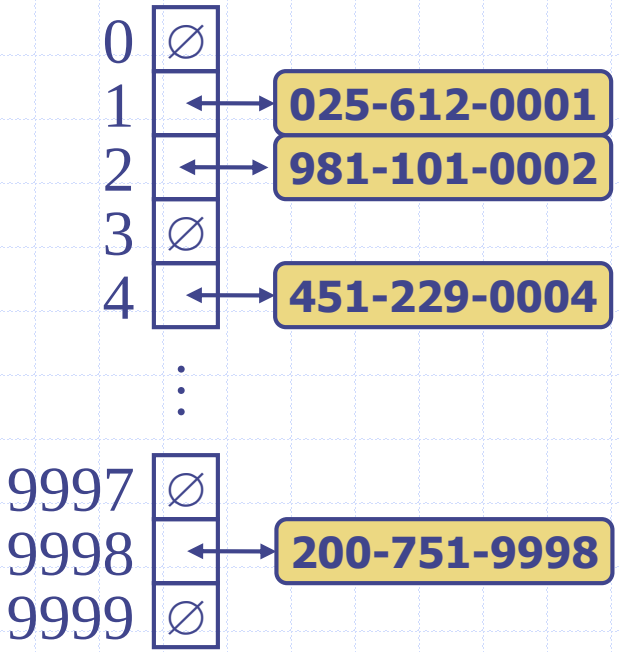
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:
  $h(k) = k \bmod N$
  is a hash function for integer keys
- The integer $h(k)$ is called the hash value of key k

Goals of Hash Functions

1. Store item (***k***, ***o***) at index ***i*** = ***h***(***k***) in the table
2. Avoid collisions as much as possible
   - Collisions occur when two keys hash to the same index *i*
   - The average performance of hashing depends on how well the hash function distributes the set of keys (i.e., avoids collisions)

Example

- Design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x) =$ last four digits of x



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code map:

  h_1: keys $\rightarrow$ integers

  Compression map:

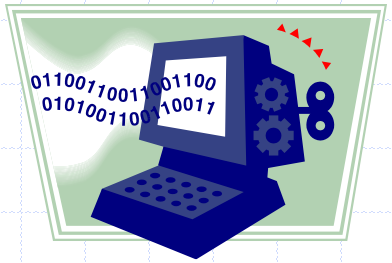
  h_2: integers $\rightarrow [0, N-1]$

- The hash code map is applied first, and the compression map is applied next on the result, i.e.,

  $$h(x) = h_2(h_1(x))$$

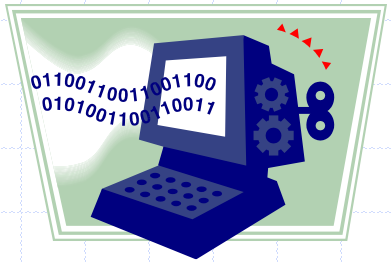
- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Code Maps



- Memory address:
  - We reinterpret the memory address of the key object as an integer
    - (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int, and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits)
  - Then we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Code Maps (cont.)



- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0 a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$

    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

    $$p_0(z) = a_{n-1}$$

    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$

    $$(i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

Compression Maps

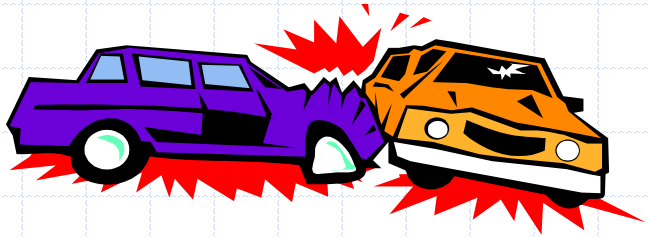
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
    - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Main Point

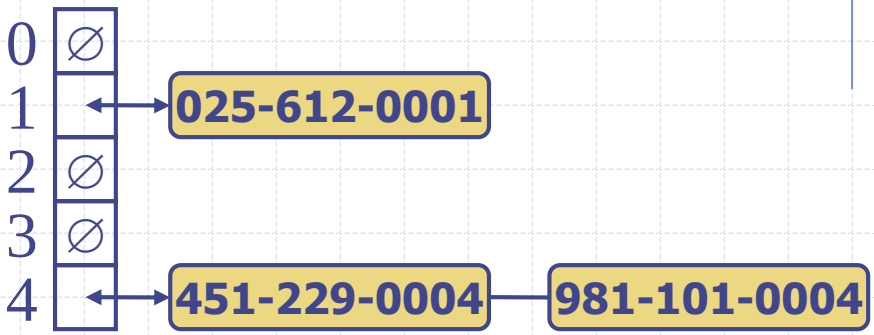
1. The hash function solves the problem of fast table-lookup, i.e., it allows the element associated with each key to be accessed quickly (in *O*(1) time). A hash function is composed of a hash code function and a compression function that transforms (in constant time) each key into a specific location in the table.

   Science of Consciousness: Through a process of self-referral, the unified field sequentially transforms itself into all the values of creation without making mistakes.

Collision Handling



- Collisions occur when different elements are mapped to the same cell
- **Chaining**: let each cell in the table point to a linked list of elements that map there

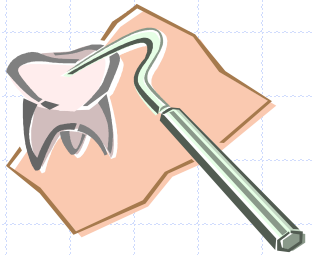


- Chaining is simple, but requires additional memory outside the table

Load Factors and Rehashing

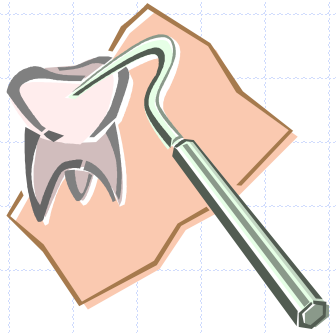
- Load factor is n/N where n is the number items in the table and N is the table size
- When the load factor goes above .75, the table is resized and the items are rehashed

Linear Probing

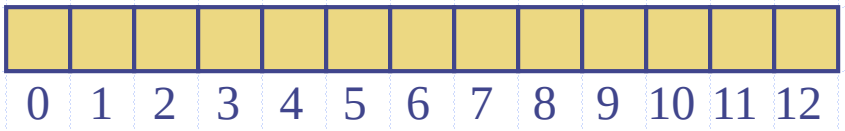


- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a “probe”
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

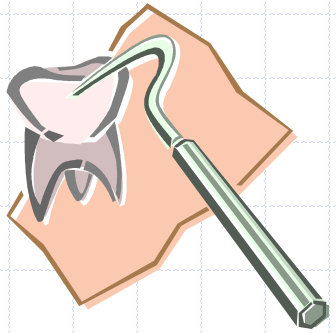
Linear Probing (§2.5.5)



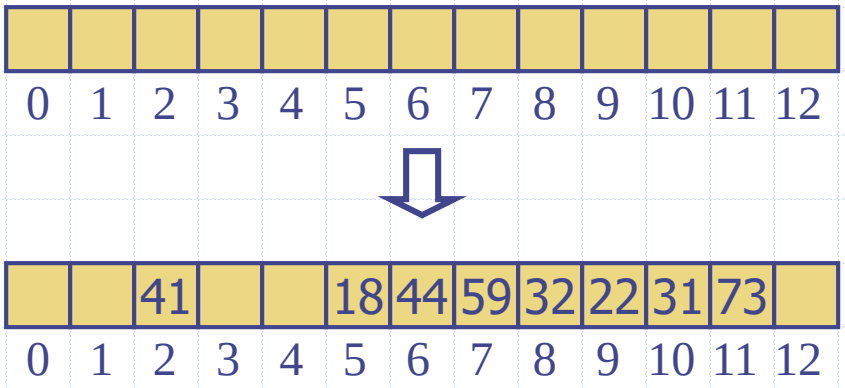
- Exercise:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



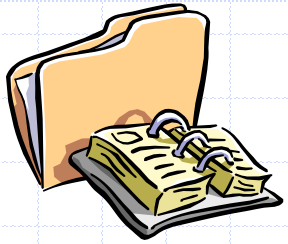
Linear Probing



- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with Linear Probing



- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm findElement(k)
i ← h(k)
p ← 0
do
    x ← (i + p) mod N
    c ← A[x]
    if c = ∅ then
        return NO_SUCH_KEY
    else if c.key () = k then
        return c.element()
    else
        p ← p + 1
while      p < N

return NO_SUCH_KEY
```

Updates with Linear Probing

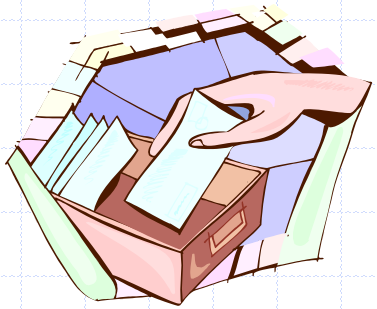
- To handle insertions and deletions, we introduce a special object, called $AVAILABLE$, which replaces deleted elements
- removeElement(k)
  - We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item $AVAILABLE$ and we return element o
  - Else, we return NO_SUCH_KEY
- insert Item(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores $AVAILABLE$, or
    - N cells have been unsuccessfully probed
  - We store item (k, o) in cell i

Quadratic Probing

- Start with the hash value i=h(k),
- Then search A[(i + j^2) mod N]

  for j = 0, 1, 2, … until an empty slot is found
- Disadvantages
  - Complicates removal even more
  - Secondary clustering

Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + j*d(k)) \bmod N$$

for $j = 0, 1, \ldots, N-1$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression map for the secondary hash function:

$$d_2(k) = q - (k \bmod q)$$

where

  - $q < N$
  - q is a prime

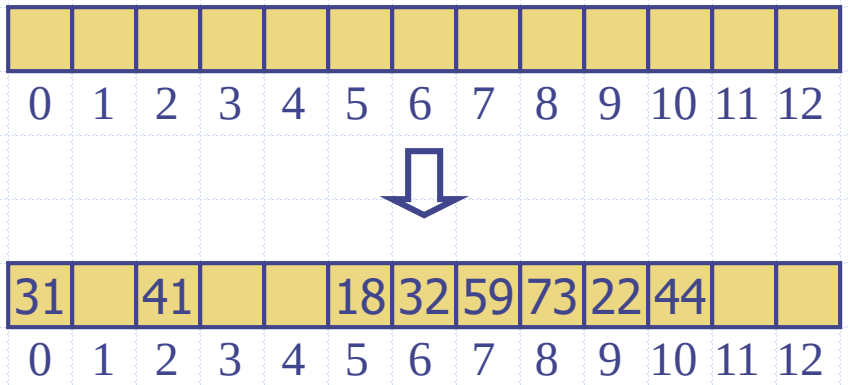
- The possible values for $d_2(k)$ are

$$1, 2, \ldots, q$$

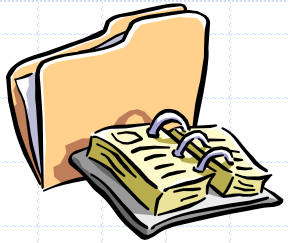
Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - (k \bmod 7)$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Search



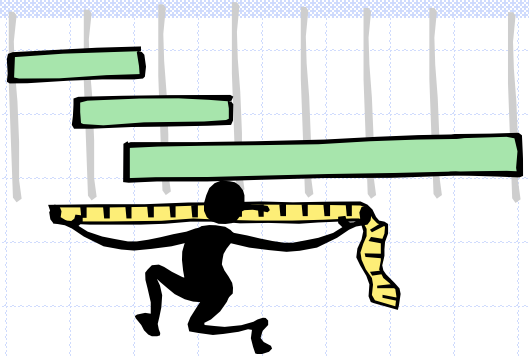
Quadratic Probing

```
Algorithm findElement(k)
i ← h(k)
p ← 0
do
    x← (i + p^2) mod N
    c ← A[x]
    if c = ∅ then
        return NO_SUCH_KEY
    else if c.key () = k then
        return c.element()
    else
        p ← p + 1
while    p < N
return NO_SUCH_KEY
```

Double Hashing

```
Algorithm findElement(k)
i ← h(k)
p ← 0
do
    x ← (i + p*d(k)) mod N
    c ← A[x]
    if c = ∅ then
        return NO_SUCH_KEY
    else if c.key () = k then
        return c.element()
    else
        p ← p + 1
while    p < N
return NO_SUCH_KEY
```

Performance of Hashing



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

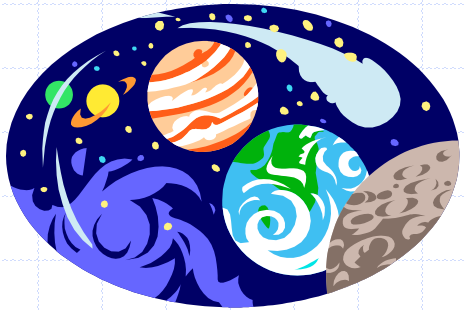
  $1 / (1 - \alpha)$

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Universal Hashing

- If allowed to pick the keys to be hashed, then a malicious adversary can choose n keys that all hash to the same slot
  - Any fixed hash function is vulnerable to this sort of worst-case behavior
- The only effective way to improve the situation
  - Choose the hash function randomly in a way that is independent of the keys to be stored
- This approach is called universal hashing
  - The hash function is chosen randomly at beginning of execution
- Yields good performance no matter what keys are chosen by an adversary

Universal Hashing



- A family of hash functions is **universal**

  if, for any $0 \le j,k \le M-1$,
  $Pr(h(j)=h(k)) \le 1/N$.

- Keys are in the range [0, M-1]
- A hash function maps to the range [0, N-1]

- Theorem: The set of all functions, h, as defined here, is universal.

- Choose p as a prime between M and 2M.
- Randomly select $0<a<p$ and $0 \le b<p$, and define

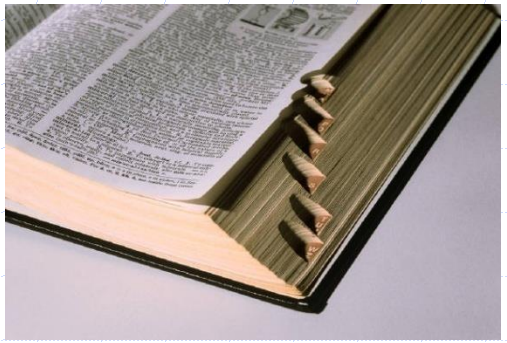
  $h(k)=(ak+b \bmod p) \bmod N$

Main Point

2. A hash table is an example of a highly efficient implementation of an unordered Dictionary ADT (its operations have complexity *O*(1)).
   Science of Consciousness: Access to Pure Consciousness is simple, effortless, and spontaneous through the introduction of the proper techniques.

Ordered Dictionaries

Ordered Dictionaries



- Keys are assumed to come from a total order.
- New operations:
  - closestKeyBefore(k)

    Return the key of the item with largest key less than or equal to *k*
  - closestElemBefore(k)

    Return the element of the item with largest key less than or equal to *k*
  - closestKeyAfter(k)

    Return the key of the item with smallest key greater than or equal to *k*
  - closestElemAfter(k)

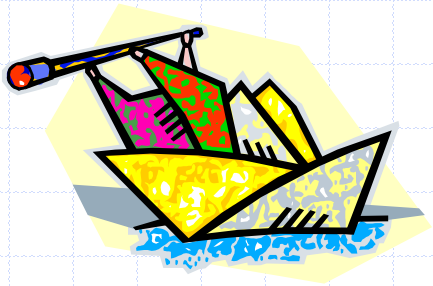
    Return the element of the item with smallest key greater than or equal to *k*

Lookup Tables

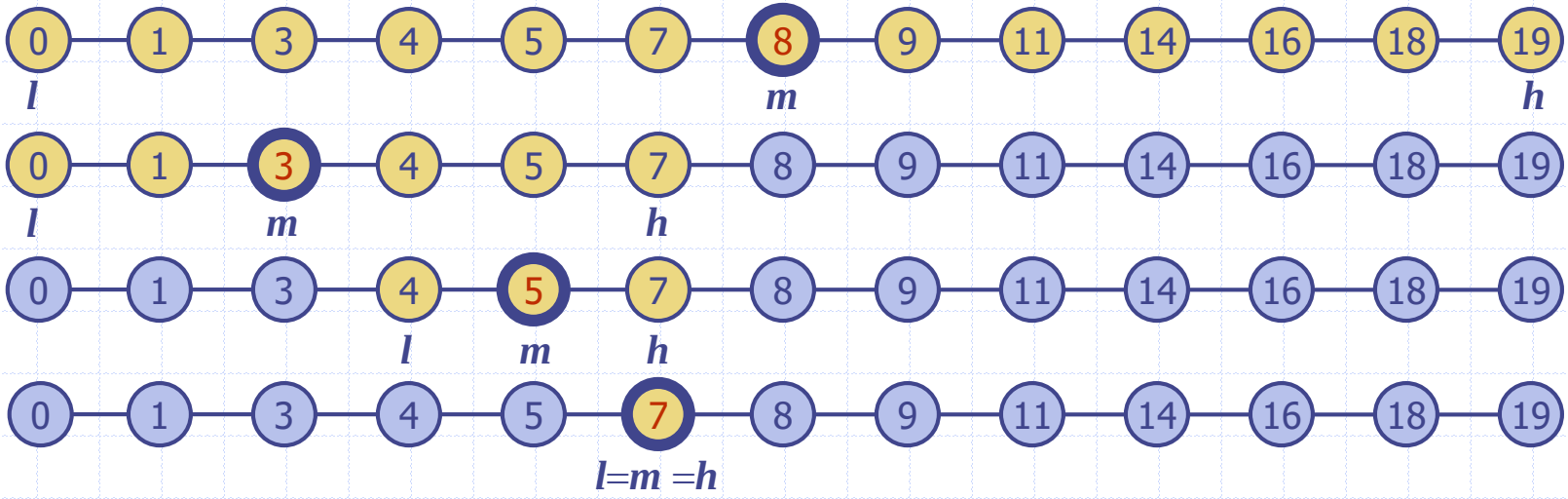
Lookup Table

- A dictionary implemented by means of a sorted sequence
  - store the items of the dictionary in an array-based sequence, sorted by key
  - use an external comparator for the keys

Binary Search



- Binary search performs operation findElement(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: findElement(7)



Binary Search Algorithm (recursive)

Algorithm BinarySearch(*S, k, low, high*):

Input: *An ordered vector S storing n items, accessed by keys()*

Output: *An element of S with key k and rank between low & high.*

```
if low > high then
    return NO_SUCH_KEY
else
    mid ← floor((low + high)/2)
    if k < key(mid) then
        return BinarySearch(S, k, low, mid-1)
    else if k = key(mid) then
        return elem(mid)
    else
        return BinarySearch(S, k, mid + 1, high)
```

Binary Search Algorithm (iterative)

Algorithm BinarySearch(*S, k*):

Input: *An ordered vector S storing n items, accessed by keys()*

Output: *An element of S with key k.*

```
low ← 0
high ← S.size() - 1
while low ≤ high do
    mid ← floor((low + high)/2)
    if k < key(mid) then
        high ← mid – 1
    else if k = key(mid) then
        return elem(mid)
    else
        low ← mid + 1
return NO_SUCH_KEY
```

Lookup Table

- A dictionary implemented by means of a sorted sequence
  - store the items of the dictionary in an array-based sequence, sorted by key
  - use an external comparator for the keys
- Performance:
  - findElement
  - insertItem
  - removeElement

Lookup Table

- A dictionary implemented by means of a sorted sequence
  - store the items of the dictionary in an array-based sequence, sorted by key
  - use an external comparator for the keys
- Performance:
  - findElement takes $O(\log n)$ time, using binary search
  - insertItem takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - removeElement take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal

Lookup Table

- Effective only
  - for dictionaries of small size or
  - for dictionaries on which
    - searches are the most common operation, and
    - insertions and removals are rarely performed
    - (e.g., credit card authorizations)

- What do we do if this is not the case?

Main Point

3. A lookup table is an example of an ordered Dictionary ADT allowing elements to be efficiently accessed in order by key. When implemented as an ordered sequence, searching for a key is relatively efficient, O(log n), but insertion and deletion are not, O(n).

   Science of Consciousness: The unified field of natural law always operates with maximum efficiency.

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. A hash table is a very efficient way of implementing an unordered Dictionary ADT; the running time of search, insertion, and deletion is expected O(1) time.

2. To achieve efficient behavior of the hash table operations takes a careful choice of table size, load factor, hash function, and handling of collisions.

3. **Transcendental Consciousness** is the silent field of perfect efficiency and frictionless flow for coordinating all activity in the universe.
4. **Impulses within Transcendental Consciousness**: The dynamic natural laws within this unbounded field create and maintain the order and balance in creation, all spontaneously without effort.
5. **Wholeness moving within itself:** In Unity Consciousness, the diversity of creation is experienced as waves of intelligence, perfectly efficient fluctuations of one's own self-referral consciousness.