Lecture 11: Dictionaries

Sequential Unfoldment of Knowledge

Wholeness Statement

The Dictionary ADT stores a searchable collection of *key-value* 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 value associated with each key can be found quickly ($O(1)$). ***Science of Consciousness*** provides systematic techniques for accessing and experiencing total knowledge of the Universe to enhance individual life. We experience this each day when we experience the silent, unbounded state of consciousness during our daily TM.

The Dictionary ADT

Our main focus in this part of the course is searching algorithms and how to organize data so it can be searched efficiently.

Review

- Recall that a Priority Queue contains key-value items
  - So does a Dictionary
- Priority Queue items were organized in three different ways
  - Unordered Sequence
  - Sorted Sequence
  - Binary Tree (Heap)
- Dictionaries can be organized in similar ways
  - Log file or Hashtable (unordered sequence)
  - Lookup Table (ordered sequence)
  - Binary Search Tree (ordered tree)

Two Types of Dictionaries

1. Unordered
2. Ordered

- Stores items, i.e., key-value pairs
- Both ordered and unordered Dictionaries search for a key to identify/locate the specific value(s) associated with that key
- For the sake of generality, multiple items could have the same key (e.g., log files), **but generally we will require each item to have a unique key**

Unordered Dictionary ADT

- The dictionary ADT models a searchable collection of key-value items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed in the log-file.
- 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:
  - findValue(k): if the dictionary has an item with key k, then returns that item's value, else, returns the special value NO_SUCH_KEY
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeItem(k): if the dictionary has an item with key k, removes item from the dictionary and returns its value, else returns the special element NO_SUCH_KEY
  - size(), isEmpty()
  - keys(), values(), items()

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(random) order
  - Based on doubly-linked lists or a circular array
- Performance(DLL):
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - findValue and removeItem take $O(n)$ time since in the worst case (the item is not found), we have to traverse the entire sequence to look for an item(s) 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, every time a user logs into a workstation, an entry is made in a log file. This entry might include the user's identification, the time of login, etc.,)

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

Hash Tables

Hash Tables and Hash Functions

- A hash table for a given key type consists of
  - Hash function $\boldsymbol{h}$
  - Array (called table) of size $\boldsymbol{N}$
- A hash function $\boldsymbol{h}$ maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

  $$\boldsymbol{h}(\boldsymbol{k}) = \boldsymbol{k} \bmod \boldsymbol{N}$$

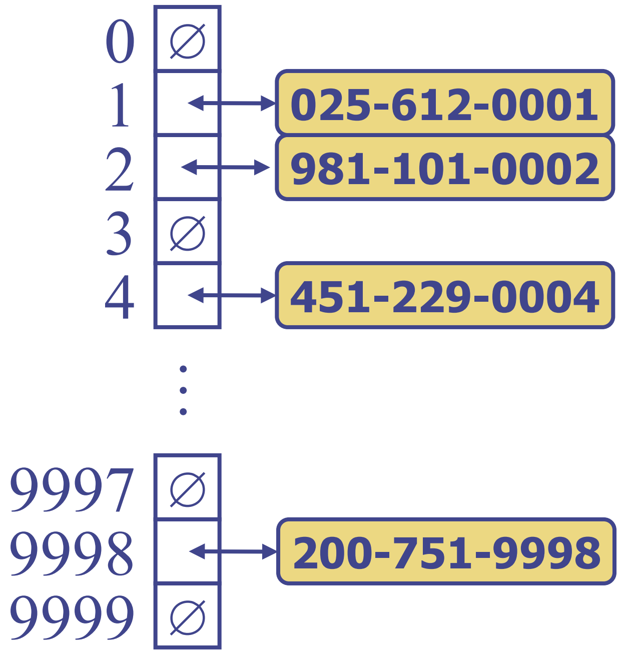
  is a hash function for integer keys
- The integer $\boldsymbol{h}(\boldsymbol{k})$ is called the hash value of key $\boldsymbol{k}$

Goals of Hash Functions

- Store item (***k***, ***o***) at index ***i*** = ***h***(***k***) in the table
- Avoid collisions as much as possible
  - Collisions occur when two different 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
  $h(x) = \text{last four digits of } x$



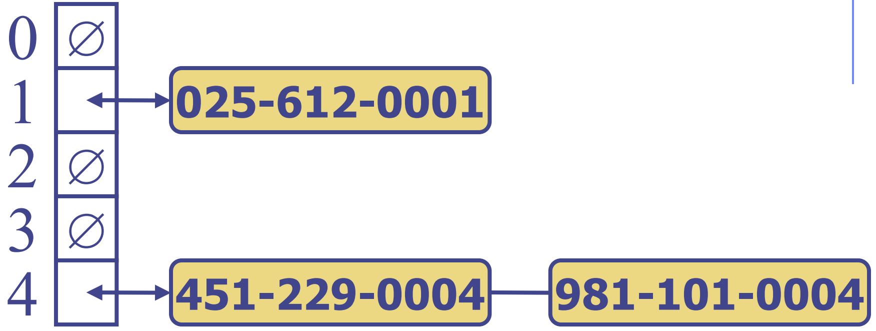
Main Point

1. The hash function solves the problem of fast table-lookup, i.e., it allows the value associated with each key to be accessed quickly (in $O(1)$ expected 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

https://yongdanielliang.github.io/animation/web/SeparateChaining.html

Load Factors and Rehashing

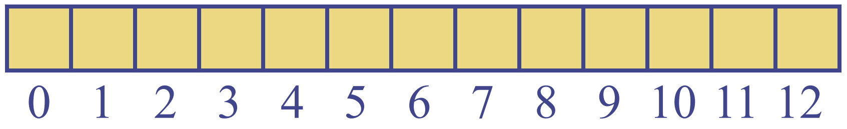
- Load factor(α) is n/N where n is the number items in the table and N is the table size
- When the load factor goes above .75, the table is resized and the items are rehashed

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
- https://yongdanielliang.github.io/animation/web/LinearProbing.html

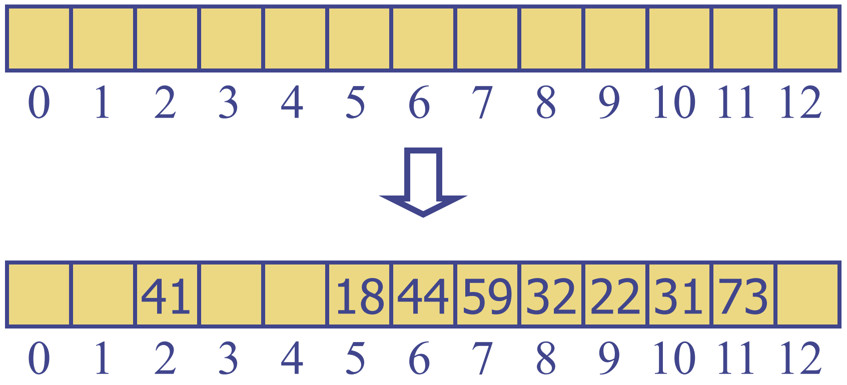
Linear Probing

- Exercise: Table size is 13
  - $h(x) = x \bmod 13$
  - 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
- findValue(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. [*NO_SUCH_KEY*]

```
Algorithm findValue(k)//Pass key and get value
    item ← findItem(k) // Both Key and Value
    return item.value() // Only value mapped with the key

Algorithm findItem(k)
i ← h(k) // Hashed index
p ← 0
while p < N  do // N – Table size
    x ← (i + p) mod N
    item ← A[x] // Retrieve from the x index
    if item = ∅ then
        return NO_SUCH_KEY
    else if item.key () = k then // Found key
        return item
    else
        p ← p + 1//Linear probing to find next Index
return NO_SUCH_KEY
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called *AVAILABLE*, which replaces deleted elements
- removeItem(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 (You have a free space to store)
    - N cells have been unsuccessfully probed. (Array is full)
  - We store item (k, o) in cell i

Quadratic Probing

[Refer class notes for detailed step by step execution]

- 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
- https://yongdanielliang.github.io/animation/web/QuadraticProbing.html

Double Hashing

- Double hashing uses a secondary hash function for key k is $d(k)$ 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
- The secondary hash function:

  $$d(k) = q - (k \bmod q)$$
- where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ are

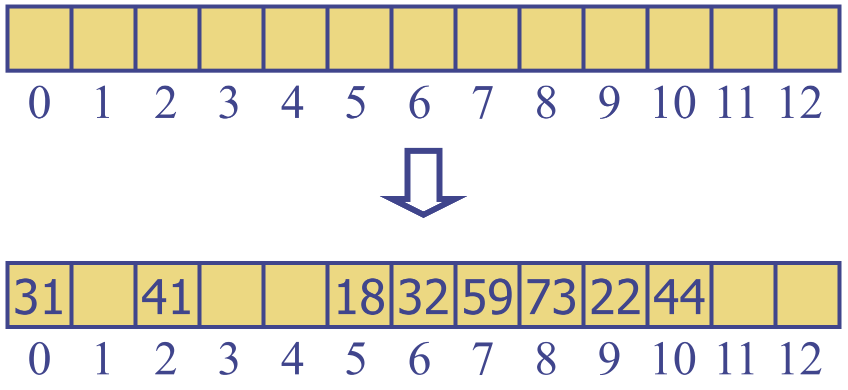
  $$1, 2, \ldots, q$$
- ***https://liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html***

Example of Double Hashing

[Refer class notes for detailed step by step execution]

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$, q = 7
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - (k \bmod 7)$
- 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		



Linear Probing

```
Algorithm findItem(k)
$i \leftarrow h(k)$
$p \leftarrow 0$
while $p < N$ do
    $x \leftarrow (i + p) \bmod N$
    $item \leftarrow A[x]$
    if $item = \varnothing$ then
        return NO_SUCH_KEY
    else if $item.key() = k$ then
        return item
    else
        $p \leftarrow p + 1$

return NO_SUCH_KEY
```

Probing Algorithms

Quadratic Probing

```
Algorithm findItem(k)
$i \leftarrow h(k)$
$p \leftarrow 0$
while $p < N$ do
    $x \leftarrow (i + p^2) \bmod N$
    $item \leftarrow A[x]$
    if $item = \varnothing$ then
        return NO_SUCH_KEY
    else if $item.key() = k$ then
        return item
    else
        $p \leftarrow p + 1$
return NO_SUCH_KEY
```

Double Hashing

```
Algorithm findItem(k)
$i \leftarrow h(k)$
$p \leftarrow 0$
while $p < N$ do
    // $q - (k \bmod q)$
    $x \leftarrow (i + p*d(k)) \bmod N$
    $item \leftarrow A[x]$
    if $item = \varnothing$ then
        return NO_SUCH_KEY
    else if $item.key() = k$ then
        return item
    else
        $p \leftarrow p + 1$
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(collision)
- 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

Main Point

2. A hash table is an example of a highly efficient implementation of an unordered Dictionary ADT (its operations have expected 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.
- Positions in a Dictionary now have two methods instead of element(), i.e., we now have
  - key()
  - value()
- Iterators return objects in order by associated key:
  - keys()
    - the iterator iterates through the keys in sorted order
  - values()
    - the iterator iterates through the values in the sorted order of the associated key

Lookup Tables

Analogous to the Priority Queue implemented as a sorted Sequence

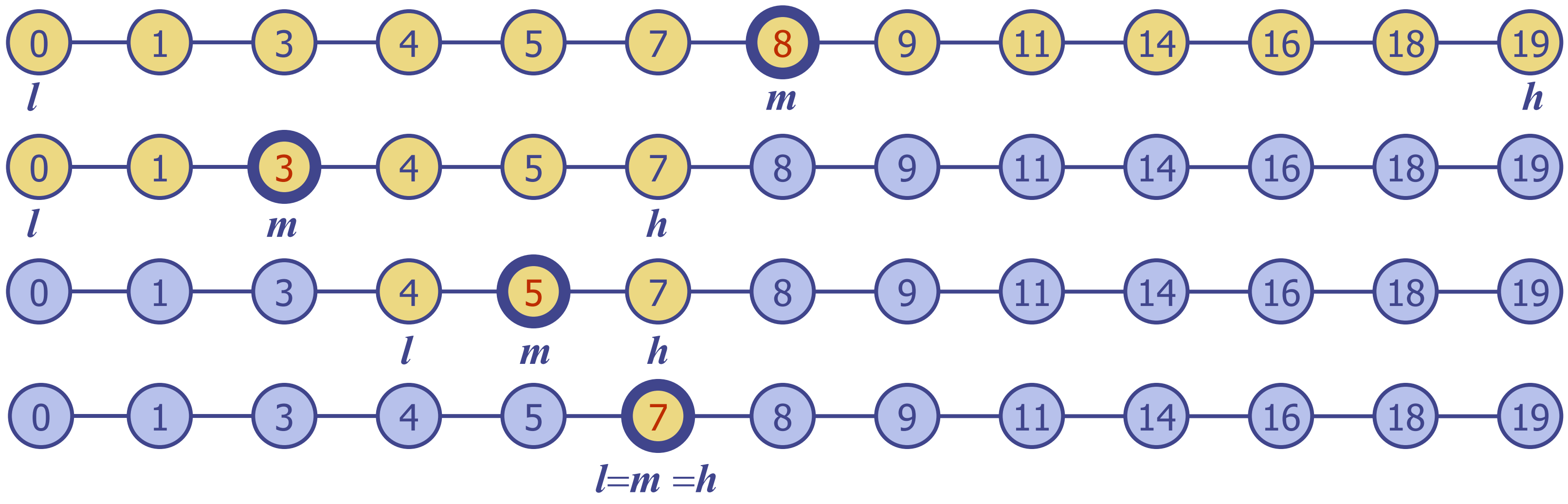
Lookup Table

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

- How should we lookup and find the value associated with a given key?
  - Hint: my favorite Dictionary because it uses my favorite algorithm

Binary Search

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



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  // exit once the key is found
          return value(mid)
      else
          low ← mid + 1
  return NO_SUCH_KEY
```

Binary Search Algorithm (can be done recursively)

```
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 value(mid)
     else
        return BinarySearch(S, k, mid + 1, high)
```

Binary Search Algorithm (improved, fewer key compares)

```
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    // always does log n iterations
      mid ← floor((low + high)/2)
      if k < key(mid) then  // one key comparison per iteration
          high ← mid – 1
      else
          low ← mid   // note that mid has not been eliminated yet

  if k = key(mid) then   // check for equality after the loop
          return value(mid)
  else 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:
  - findValue 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
  - removeItem takes $O(n)$ 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)

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.