Lesson 8
Data Structure Review

: *Fully Developing the Container of Knowledge*

Wholeness of the Lesson

An analysis of the average-case and worst-case running times of many familiar data structures (for instance, array lists, linked lists, stacks, queues, hashtables) highlights their strengths and potential weaknesses; clarifies which data structures should be used for different purposes; and points to aspects of their performance that could potentially be improved. Likewise, finer levels of intelligence are more expanded but at the same time more discriminating. For this reason, action that arises from a higher level of consciousness spontaneously computes the best path for success and fulfillment.

Dynamic Sets

- Now we will focus on data structures rather than straight algorithms
- In particular, structures for *dynamic sets*
  - Elements have a *key* and *satellite data*
  - Dynamic sets support *queries* such as:
    - ***Search(S, k), Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x)***
  - They may also support *modifying operations* like:
    - ***Insert(S, x), Delete(S, x)***

Dynamic Sets

- What is Dynamic Set?
  - Mathematical sets are unchanging
  - BUT sets used in computers can grow or shrink i.e. they are DYNAMIC.
  - Will talk about algorithms to manipulate such Dynamic sets.
- Most common operations will be INSERT, DELETE and SEARCH – Sets supporting these are usually Called DICTIONARY.

Stacks

- Stack - LIFO (Last in First Out)
- Queue – FIFO (First in First Out)

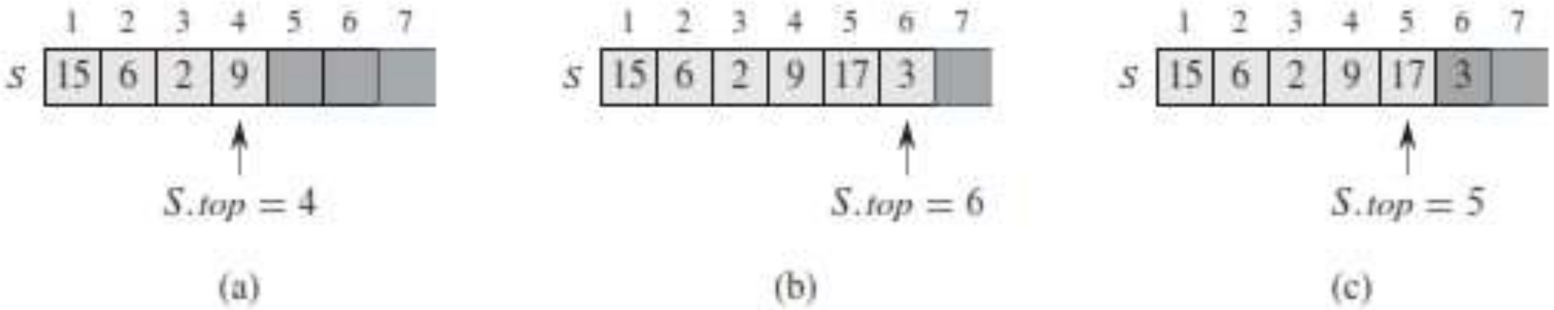

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

Stacks

- Common Operations on a Stack

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE

PUSH(S, x)
S.top = S.top + 1
S[S.top] = x

POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

Queue

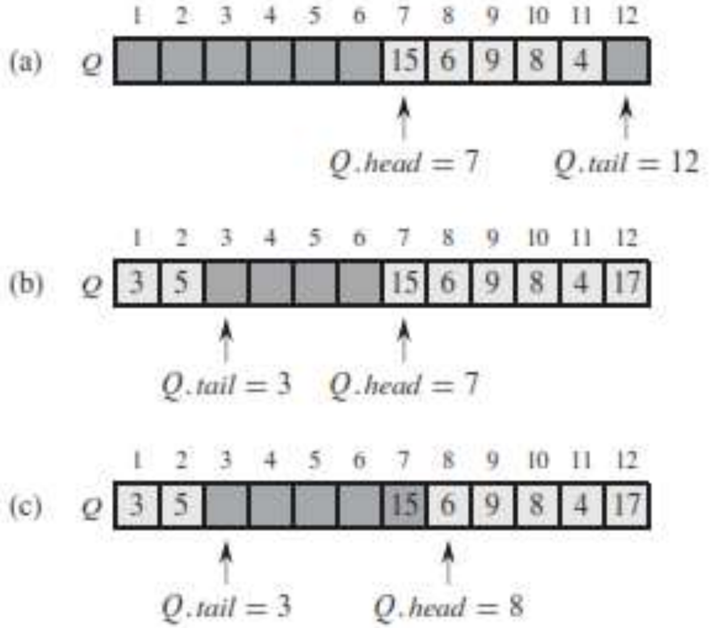

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5). **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Queue

```
ENQUEUE(Q, x)
Q[Q.tail] = x
if Q.tail == Q.length
        Q.tail = 1
else Q.tail = Q.tail + 1

DEQUEUE(Q)
x = Q[Q.head]
if Q.head == Q.length
        Q.head = 1
else Q.head = Q.head + 1
return x
```

Linked List

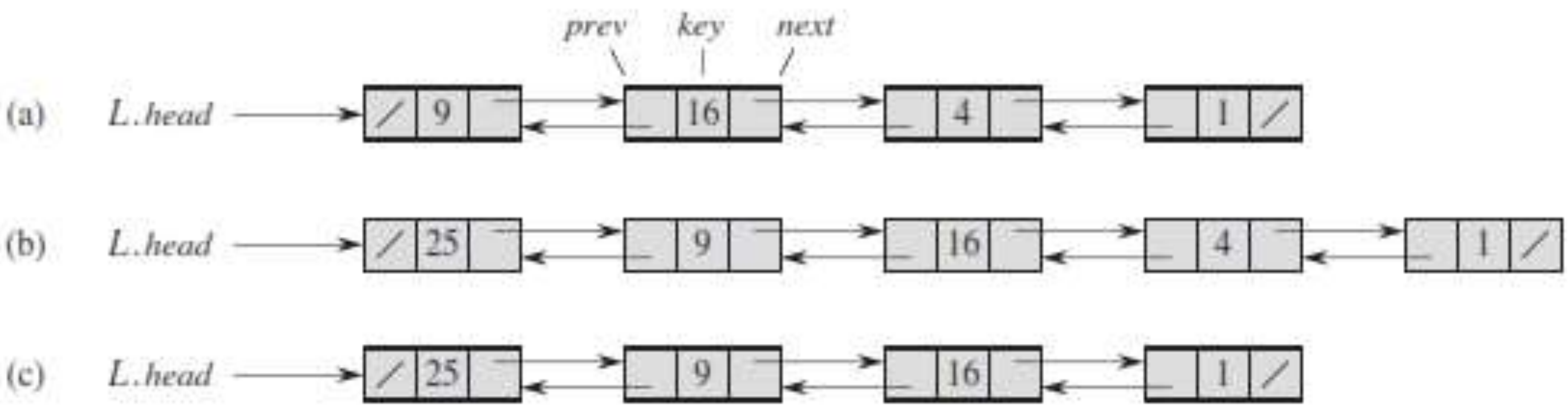

Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Linked List

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

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

Linked List

```
LIST-DELETE(L, x)
  if x.prev ≠ NIL
      x.prev.next = x.next
  else L.head = x.next
  if x.next ≠ NIL
      x.next.prev = x.prev
```

Hashing

- Direct Address table

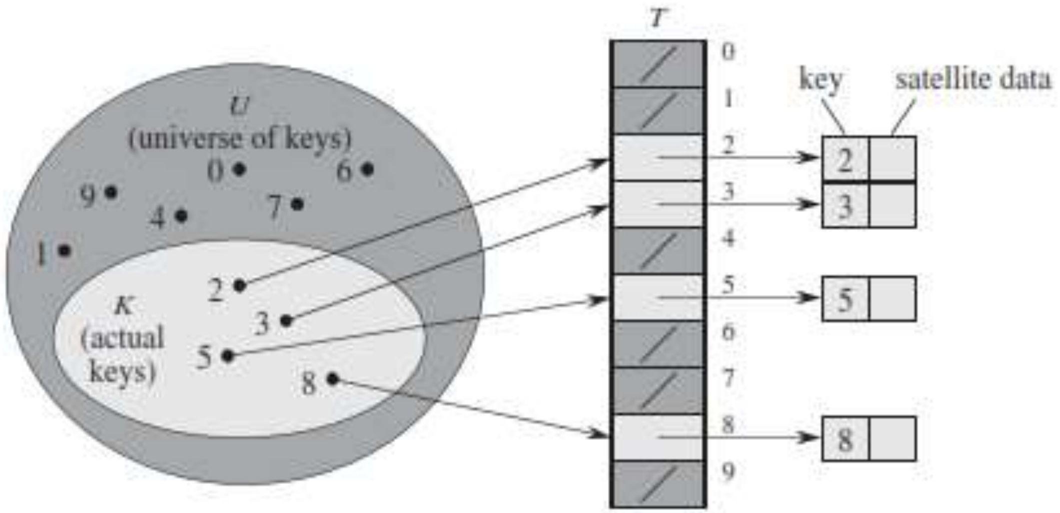

Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Hashing

- Hash table

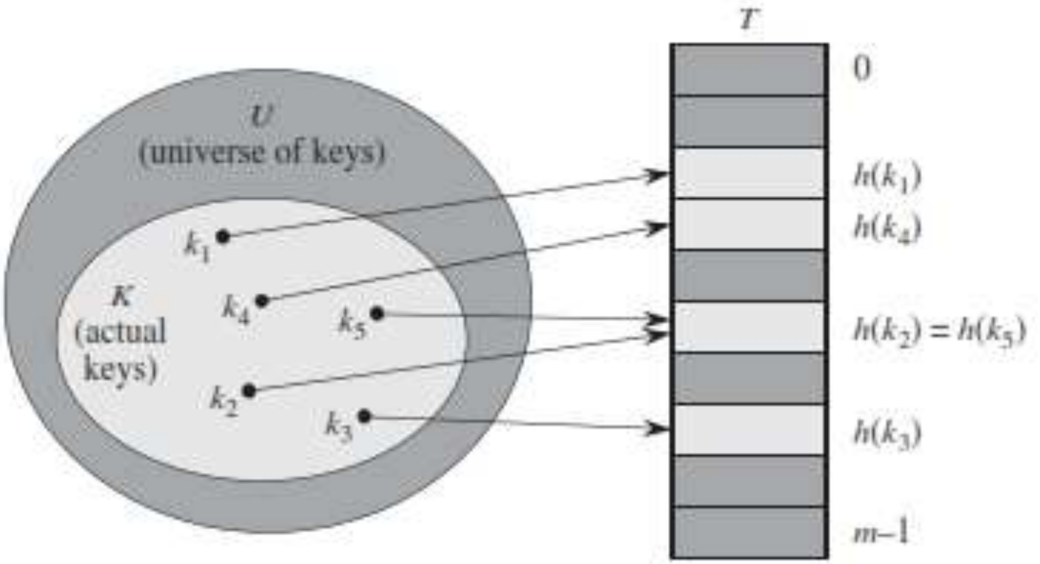

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

Hashing

- Collisions and How to Avoid Them

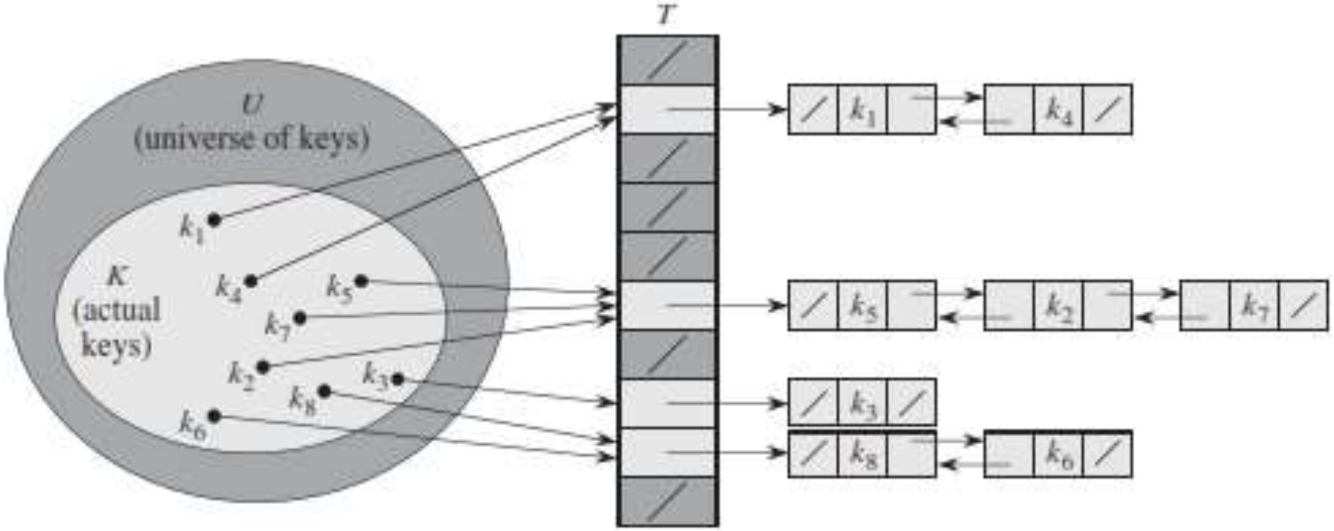

Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

Hash Functions

- Functions that Calculate Hash address using the values of the keys
  - Division Method (m is the number of slots in hash table]

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

Say m=12, k = 100, h(k) = ?

  - Multiplication Method $h(k) = \lfloor m \, (k A \bmod 1) \rfloor \, .$

kA mod 1 means fractional part of kA i.e. kA-floor of kA

  - Universal Hashing

Hash Functions

- More on Multiplication Method
  - A is between 0 and 1. Optimal value is 0.618!

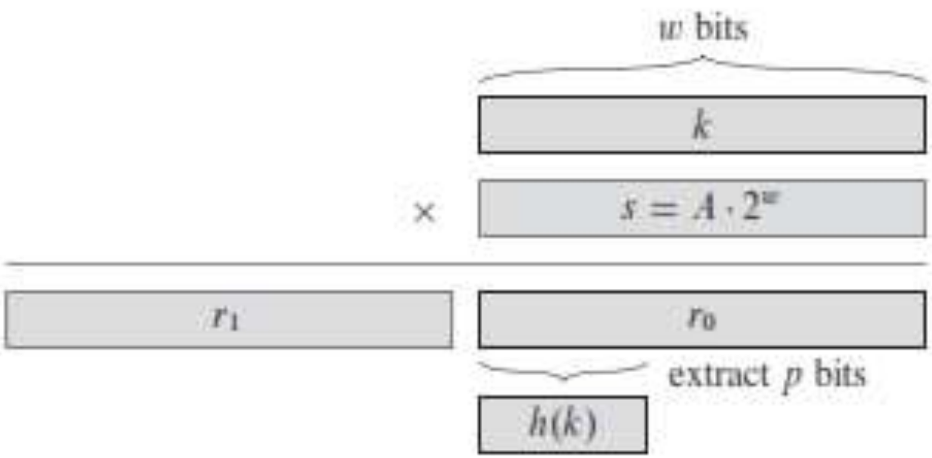

Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

We typically choose it to be a power of 2 ($m = 2^p$ for some integer p).

Multiplication Method - Example

- If $0 < A < 1$, $h(k) = \lfloor m\,(kA \bmod 1) \rfloor = \lfloor m\,(kA - \lfloor kA \rfloor)\rfloor$
where $kA \bmod 1$ means the fractional part of kA, *i.e.*, $kA - \lfloor kA \rfloor$.
- Disadvantage: Slower than the division method.
- Advantage: Value of m is not critical.
  - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes implementation easy.

- Example: $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

$h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1) \rfloor = \lfloor m\,(kA - \lfloor kA \rfloor)\rfloor$

$= \lfloor 1000 \cdot 0.018169\ldots \rfloor = 18.$

Hash Functions – Universal Hashing

- For Division or Multi. Method, the worst case is that all keys map to ONE Slot!
- To avoid so, Universal hashing selects hash functions randomly.
- The key idea is to select Hash Function at random at run time from set of selected functions.
- This provides good average case performance.

Hash Functions – Universal Hashing

- H – finite collection of hash functions that maps Universe U to keys in the range [0,1,2, m-1].
- If the Number of hash functions h, for which h(x) = h(y) is |H| / m , then

  H is a Universal Hash Function.
- Chance of collision between x and y is 1/m.

Hash Functions - Universal Hashing

- Decompose

key x into $r+1$ bytes (i.e., characters, or fixed-width binary substrings), so that $x = \langle x_0, x_1, \ldots, x_r \rangle$; the only requirement is that the maximum value of a byte should be less than m. Let $a = \langle a_0, a_1, \ldots, a_r \rangle$ denote a sequence of $r+1$ elements chosen randomly from the set $\{0, 1, \ldots, m-1\}$. We define a corresponding hash function $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m . \tag{12.3}$$

With this definition,

$$\mathcal{H} = \bigcup_a \{h_a\} \tag{12.4}$$

has m^{r+1} members.

Universal Hashing – Key Notes

- A hash function is selected at random (from the family of hash functions) at run time.
- This means that one hash function is used for that run time. Next run time, another hash function used for all keys (if a hash function causes too many collision, the program can possibly rerun using another hash function).
- Thus, on the average for several runs, the collision is minimized.
- To find a key, use the same hash function to calculate the address and get the data from that address.

Example

$max^m = 3$

00
01
10
11

- Let $m = 5$, and the size of each string is 2 bits (binary). Note the maximum value of a string is 3 and $m = 5$
- $a = 1,3$, chosen at random from 0,1,2,3,4
- Example for $x = 4 = 01,00$ (note $r = 1$)
- $h_a(4) = 1 \times (01) + 3 \times (00) = 1 \text{ Mod } 5 = 1$

[Note 01 and 00 need to be converted to decimal]

NOTES: - Pick r based on m and the range of keys in U

- *Remember – Total # of Hash Function = m**(r+1)*
- *Probability of Collision = 1/m (why?)*

Proof for Collision pr. 1/m (optional)

Collision happens when 2 keys are hashed to the same location. Thus, we have

$$h(x) = h(y)$$

$$\sum_{i=0}^{r} a_i x_i \bmod m = \sum_{i=0}^{r} a_i y_i \bmod m$$

or, $$a_0(x_0 - y_0) + \sum_{i=1}^{r} a_i(x_i - y_i) \bmod m = 0$$

or, $$a_0(x_0 - y_0) = -\sum_{i=1}^{r} a_i(x_i - y_i) \bmod m$$

or, $$a_0 = -\sum_{i=1}^{r} a_i(x_i - y_i) \cdot (x_0 - y_0)^{-1} \bmod m$$

The non-zero quantity $(x_0 - y_0)$ has (a multiplicative inverse modulo m,) and thus there is a unique soln. for a_0.

$x_0 - y_0$ will not be 0

Note that for a fixed value of $a_1, a_2 \dots a_r$, there is exactly one value of a_0 that satisfies $h(x) = h(y)$.

Now, there are m^{r+1} hash functions and m^r possible collisions. Therefore, the probability of collision $= \frac{m^r}{m^{r+1}} = \frac{1}{m}$ $\square$

About Inverse Modulo

Given two integers ‘x’ and ‘m’, find modular multiplicative inverse of ‘x’ under modulo ‘m’.

The modular multiplicative inverse is an integer ‘x’ such that.
A . X = 1 mod (M)

Example:
Input: X = 3, m = 11, Output: 4. Since (4*3) mod 11 = 1,

4 is modulo inverse of 3(under 11).

Keys as Natural Numbers

- Hash functions assume that the keys are natural numbers.
- When they are not, have to interpret them as natural numbers.
- Example: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: C=67, L=76, R=82, S=83.
  - There are 128 basic ASCII values.
  - So, CLRS = $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141{,}764{,}947$.

Main Point – Hash Table

Hashtables are a generalization of the concept of an array. They support (nearly) random access of table elements by looking up with a (possibly) non-integer key, and therefore their main operations have an average-case running time of O(1). Hashtables illustrate the principle of *Do less and accomplish more* by providing extremely fast implementation of the main List operations.