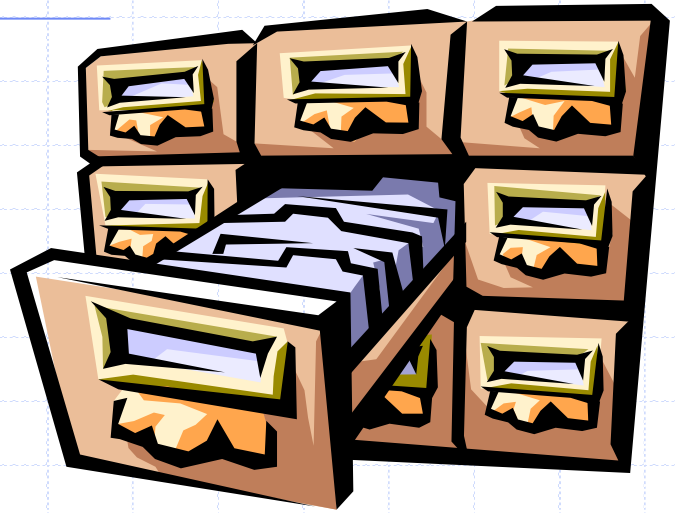
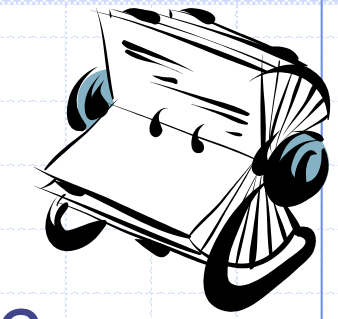
Maps



Maps

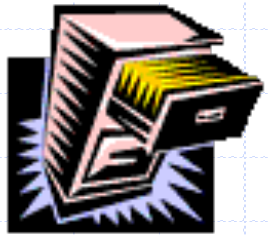


- A **map** is a searchable collection of items that are key-value pairs
- The main operations of a map are for searching, inserting, and deleting items
- Multiple items with the same key are not allowed
- Applications:
  - address book
  - student-record database

Dictionaries

- Python's **dict** class is arguably the most significant data structure in the language.
  - It represents an abstraction known as a **dictionary** in which unique **keys** are mapped to associated **values**.
- Here, we use the term "dictionary" when specifically discussing Python's dict class, and the term "map" when discussing the more general notion of the abstract data type.

The Map ADT (Using **dict** Syntax)



M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError. In Python, this is implemented with the special method `__getitem__`.

M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. In Python, this is implemented with the special method `__setitem__`.

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError. In Python, this is implemented with the special method `__delitem__`.

len(M): Return the number of items in map M. In Python, this is implemented with the special method `__len__`.

iter(M): The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method `__iter__`, and it allows loops of the form, **for** k **in** M.

More Map Operations

k in M:	Return True if the map contains an item with key k. In Python, this is implemented with the special __contains__ method.
M.get(k, d=None):	Return M[k] if key k exists in the map; otherwise return default value d. This provides a form to query M[k] without risk of a KeyError.
M.setdefault(k, d):	If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.
M.pop(k, d=None):	Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d (or raise KeyError if parameter d is None).

A Few More Map Operations

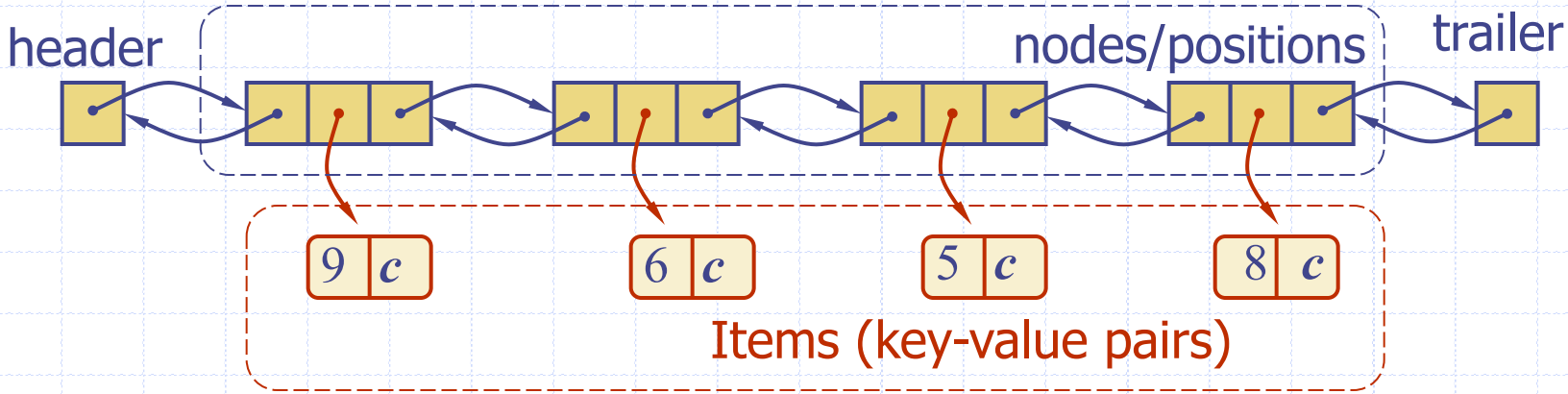
M.popitem( ):	Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.
M.clear( ):	Remove all key-value pairs from the map.
M.keys( ):	Return a set-like view of all keys of M.
M.values( ):	Return a set-like view of all values of M.
M.items( ):	Return a set-like view of (k,v) tuples for all entries of M.
M.update(M2):	Assign M[k] = v for every (k,v) pair in map M2.
M == M2:	Return True if maps M and M2 have identical key-value associations.
M != M2:	Return True if maps M and M2 do not have identical key-value associations.

Example

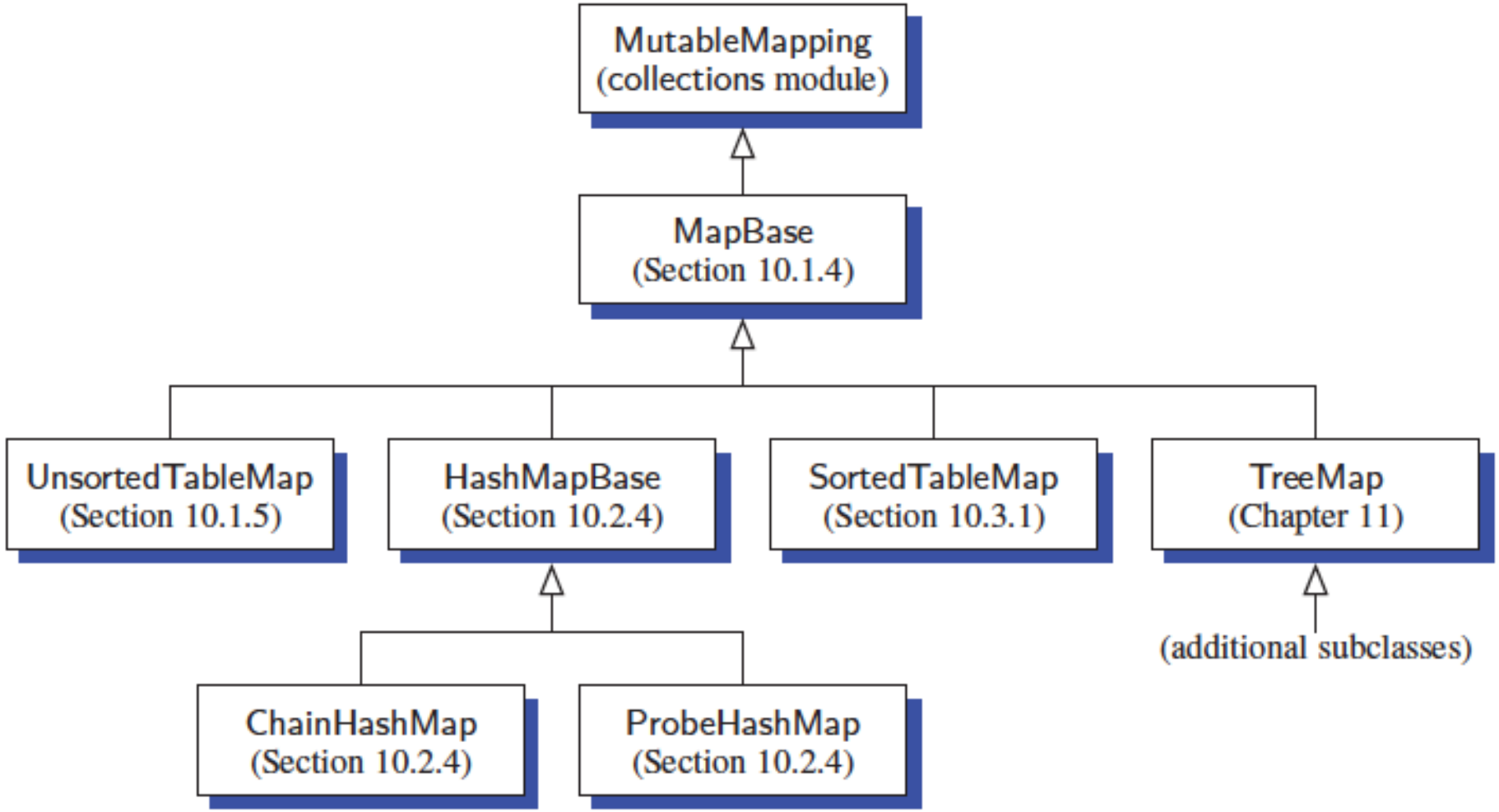
Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Our MapBase Class



Performance of a List-Based Map

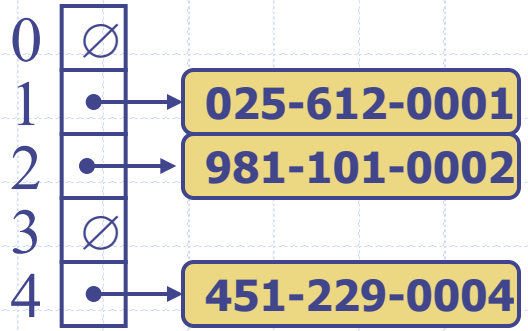
- Performance:
  - Inserting an item takes $O(1)$ time since we can insert the new item at the beginning or at the end of the unsorted list
  - Searching for or removing an item takes $O(n)$ time, since in the worst case (the item is not found) we traverse the entire list to look for an item with the given key

- The unsorted list implementation is effective only for maps of small size or for maps in which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Different Data Structures to Implement Map ADT

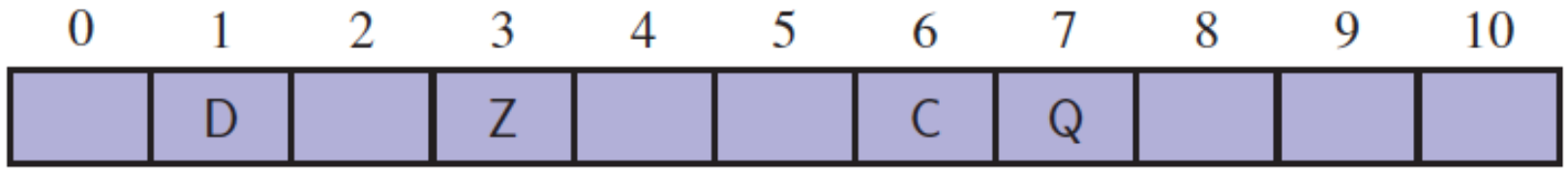
- arrays, linked lists (inefficient)
- Binary Trees
- **Hash Tables**
- 2-4 Trees
- AVL Trees
- B-Trees

Hash Tables



Intuitive Notion of a Map

- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].
- As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to $N - 1$, for some $N \geq n$.



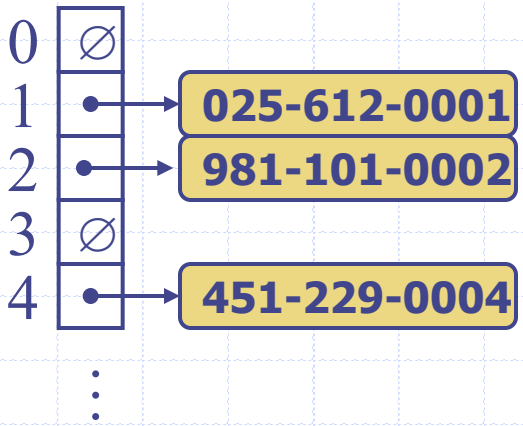
Hash Table Solution

- O(1) - expected time
- O(n+m) - space, where m is size of the table
- Instead of a one-to-one map between the key values and array locations, find a function to map the large range into one which we can manage
  - e.g., key value modulo size of array, and use that as an index
  - Insert (12345678, C) into a hashed array of size 5, - 12345678 mod 5 = 3

			C	
XXXX-XXXX	XXXX-XXXX	XXXX-XXXX	1234-5678	XXXX-XXXX

More General Kinds of Keys

- But what should we do if our keys are not integers in the range from 0 to N – 1?
  - Use a **hash function** to map general keys to corresponding indices in a table.
  - For instance, the last four digits of a Identification number.

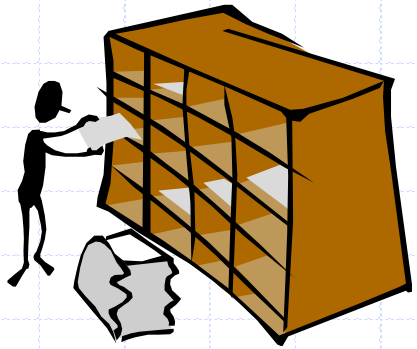


Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

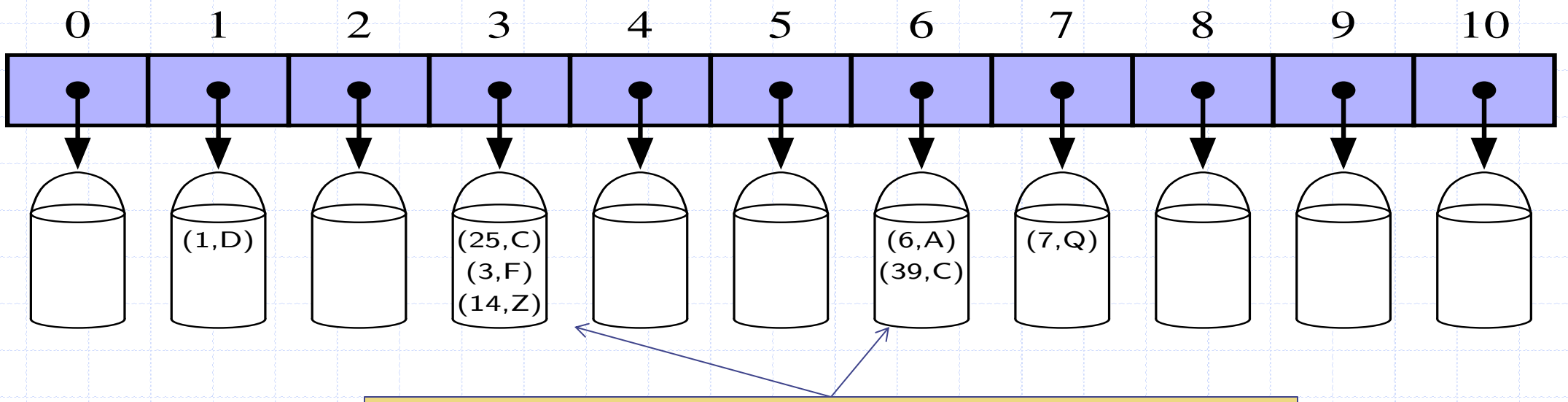
  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x



- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example



Collision – Two different keys resulting in the same hash value

Hash Functions

- Need to choose a good hash function
  - quick to compute
  - uniform distribution of keys throughout the table
  - good hash functions are very rare.
- How to deal with hashing non-integer keys
  - find some way to turn keys into integers
  - use standard hash functions on these integers

Hash Functions (2)

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

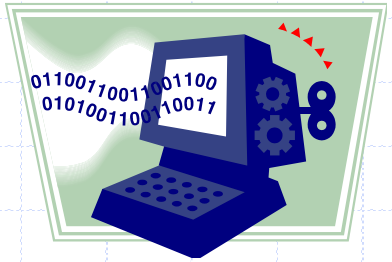
  h_2: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function 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 Codes



- 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)
  - For keys of length greater than the number of bits of the integer type, ignore the exceeding bits
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and 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)
  - Not a good choice for strings

Hash Codes (2)

- 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

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

    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)

Compression Functions

- Division:
  - Use the remainder
  - $h_2(y) = y \bmod N$
    - ***y is the key***
    - ***N is size of the table***
  - how to choose N?

Consider (200, 205,210,215,220,…600)
N = 100
N = 101

Compression Functions (2)

- Division:
  - Use the remainder
  - $h_2(y) = y \bmod N$
    - ***y is the key***
    - ***N is size of the table***
  - how to choose N?
  - N = b^x (bad)
    - N is a power of 2, $h_2(y)$ gives the x least significant bits of y.
    - all keys with the same ending go to the same place
  - N is prime (good)
    - helps ensure uniform distribution

Consider (200,205,210,215,220,...600)
N = 100
N = 101

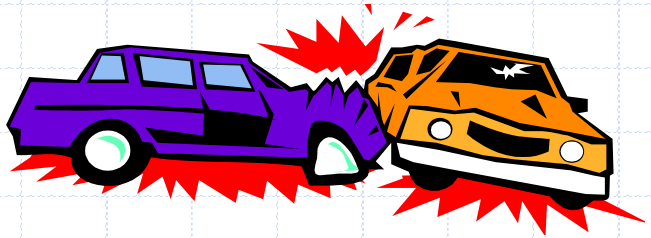
Compression Functions (3)

- Multiply, Add and Divide (MAD):
  - $h_2(y) = [(ay + b) \bmod p] \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b
  - p is a prime number larger than maximum value of y
  - a and b are chosen at random from the interval $[0, p-1]$, with $a > 0$

Compression Functions (4)

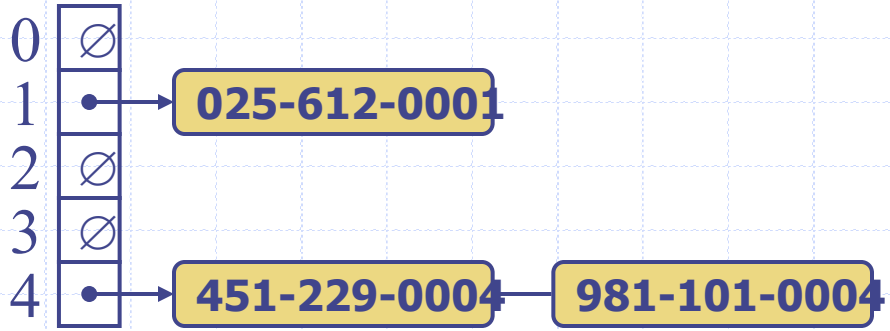
- For any choice of hash function, there always exists a bad set of keys
- You could choose only these keys, resulting in all of them getting mapped to the same slot.
  - reduction in performance.
- Solution
  - collection of hash functions
  - a random hash function
  - choose a hash function that is independent of the keys

Collision Handling



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there
  - complexity depends on load factor

- Separate chaining is simple, but requires additional memory outside the table



Map with Separate Chaining

Delegate operations to a list-based map at each cell:

```
Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = null then                    {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ null then                    {k was found}
    n = n - 1
return t
```

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
  - load factor is at most 1
- 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)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
					18							

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
					18							

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18							

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18							

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18				22			

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18				22			

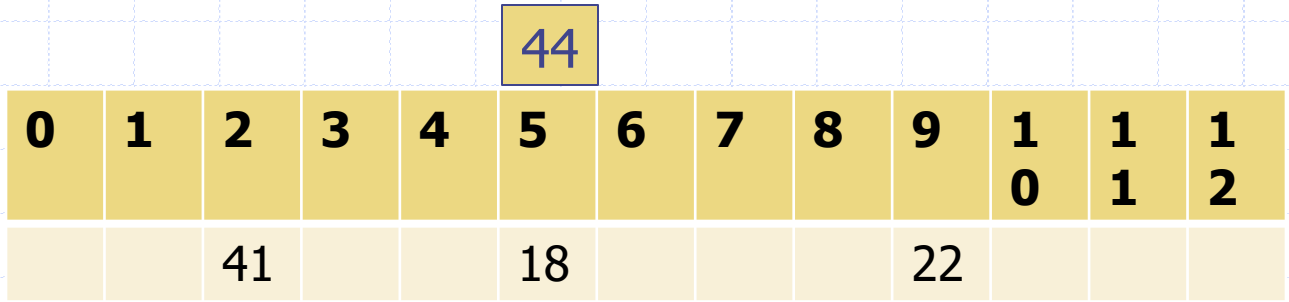
Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

					44							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
		41			18				22			

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13



Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44			22			

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44			22			

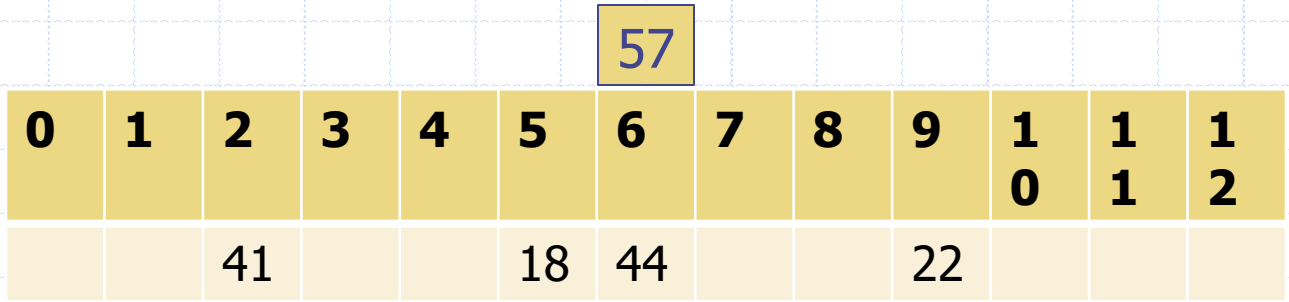
Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order

					57							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
		41			18	44			22			

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13



Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13
    - h(x) = (x+2) mod 13

57

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44			22			

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13
    - h(x) = (x+2) mod 13

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	57		22			

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13
    - h(x) = (x+2) mod 13

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	57	32	22			

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13
    - h(x) = (x+2) mod 13

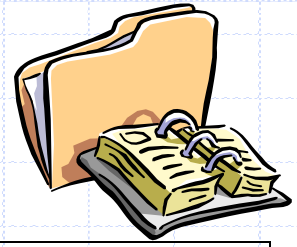
0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	57	32	22	31		

Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
    - h(x) = (x+1) mod 13
    - h(x) = (x+2) mod 13

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	57	32	22	31	73	

Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(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 get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.getKey() = k
            return c.getValue()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

Search with Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
  - Search for 57

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	57	32	22	31	73	

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called $DEFUNCT$, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item $DEFUNCT$ and we return element o
  - Else, we return $null$

Update with Linear Probing (2)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
  - Remove 57
  - Search for 32

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44		32	22	31	73	

Update with Linear Probing (3)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
  - Remove 57
  - Search for 32

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	X	32	22	31	73	

Updates with Linear Probing (4)

- To handle insertions and deletions, we introduce a special object, called $DEFUNCT$, which replaces deleted elements
- put(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 $DEFUNCT$, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

Update with Linear Probing (5)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
  - Remove 57
  - Search for 32
  - Insert 58

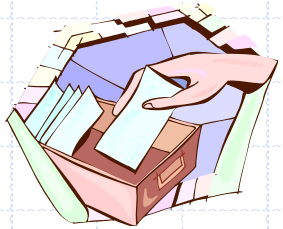
0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	X	32	22	31	73	

Update with Linear Probing (5)

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 57, 32, 31, 73, in this order
  - Remove 57
  - Search for 32
  - Insert 58

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	58	32	22	31	73	

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

$$(i + jd(k)) \bmod N$$

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

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		

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	
31		41			18	32	59	73	22	44		

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 map 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