Hash Tables

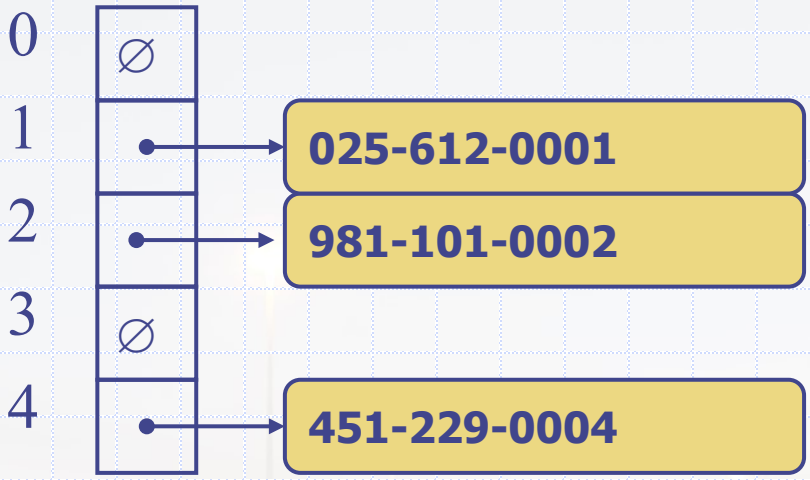
Dr. Aiman Hanna

Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

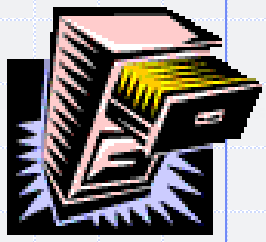
These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Coverage

- Section 9.2: Hash Tables

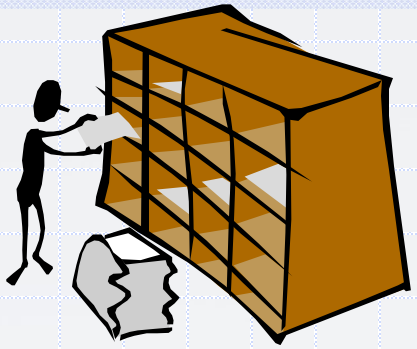


Recall the Map ADT



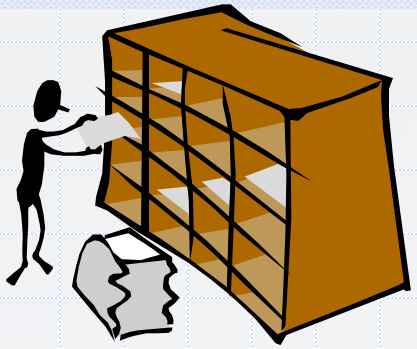
- A map supports the following methods:
  - get(k): if the map M has an entry with key k, return its associated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, replace the value associated with k with v and return that old value
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - entrySet(): return an iterable collection containing all the key-value entries entries in M
  - keySet(): return an iterable collection of all the keys in M
  - values(): return an iterable collection of all the values in M
  - size(): return the number of entries in M
  - isEmpty(): test whether M is empty

Hash Functions and Hash Tables



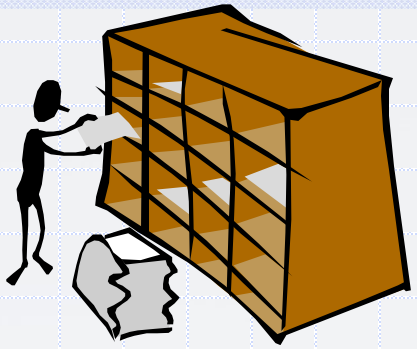
- To speed up searching for an entry, we can create an array of holders/buckets of these entries.
- The resulting data structure is hence a table.
- If insertion of entries is made such that each entry can be placed at only one possible location (bucket) in this data structure, then searching for the entry is consequently restricted to searching that particular bucket instead of searching the entire data structure.

Hash Functions and Hash Tables



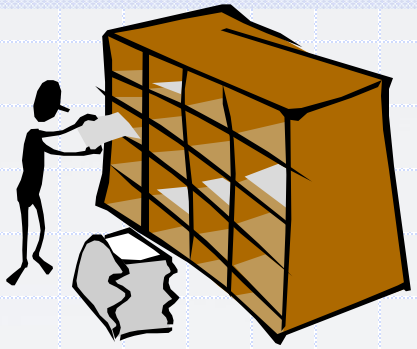
- In a best scenario, where each bucket can contain only one entry, searching for any entry can be done in $O(1)$ time.
- That can be achieved if the keys are unique integers in the range $[0 .. N-1]$, where N is the size of the array.
- This however have two drawbacks:
  - If N is actually much larger than the actual number of stored elements, then a lot of space is wasted
  - Keys must be integers, which may not be the case all the time

Hash Functions and Hash Tables



- Instead, have an array of limited size N, then somehow convert the keys to a value between 0 and $N-1$.
- The method/function performing the conversion is referred to as "hash function".
- The entire data structure (that is the array and the holders/buckets) are referred to as "hash table".
- Notice that now multiple keys may be converted to the same index value, which may result in multiple entries being placed in the same bucket. This is referred to as "collision".

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 resulting integer $h(x)$ is called the hash value of key x.

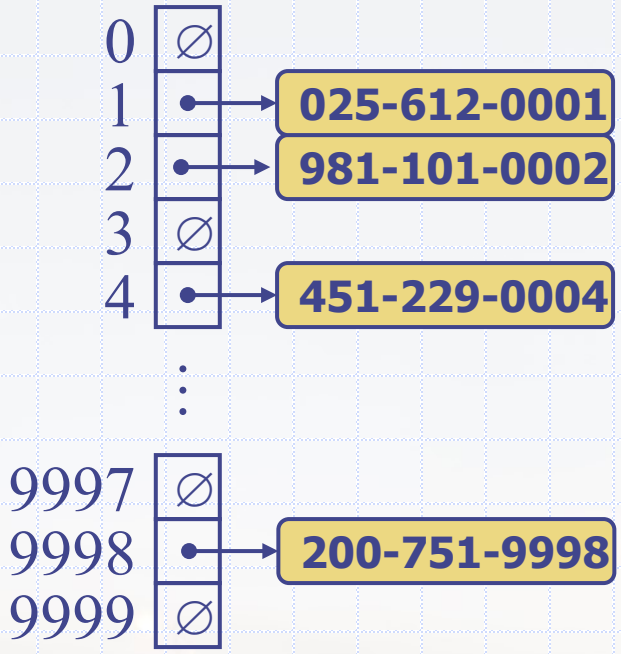
Hash Functions and Hash Tables

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N of holders/buckets of the entries
- When implementing a map with a hash table, the goal is to store entry (k, v) at index $i = h(k)$.
- In other words, entry (k, v) is stored at A[$h(k)$].

Example

- We can design a hash table for a map storing entries as (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
  $h(x) = \text{last four digits of } x.$



Hash Functions

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

  Hash code: mapping keys to integers

  h_1: keys $\rightarrow$ integers

  Compression function:

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

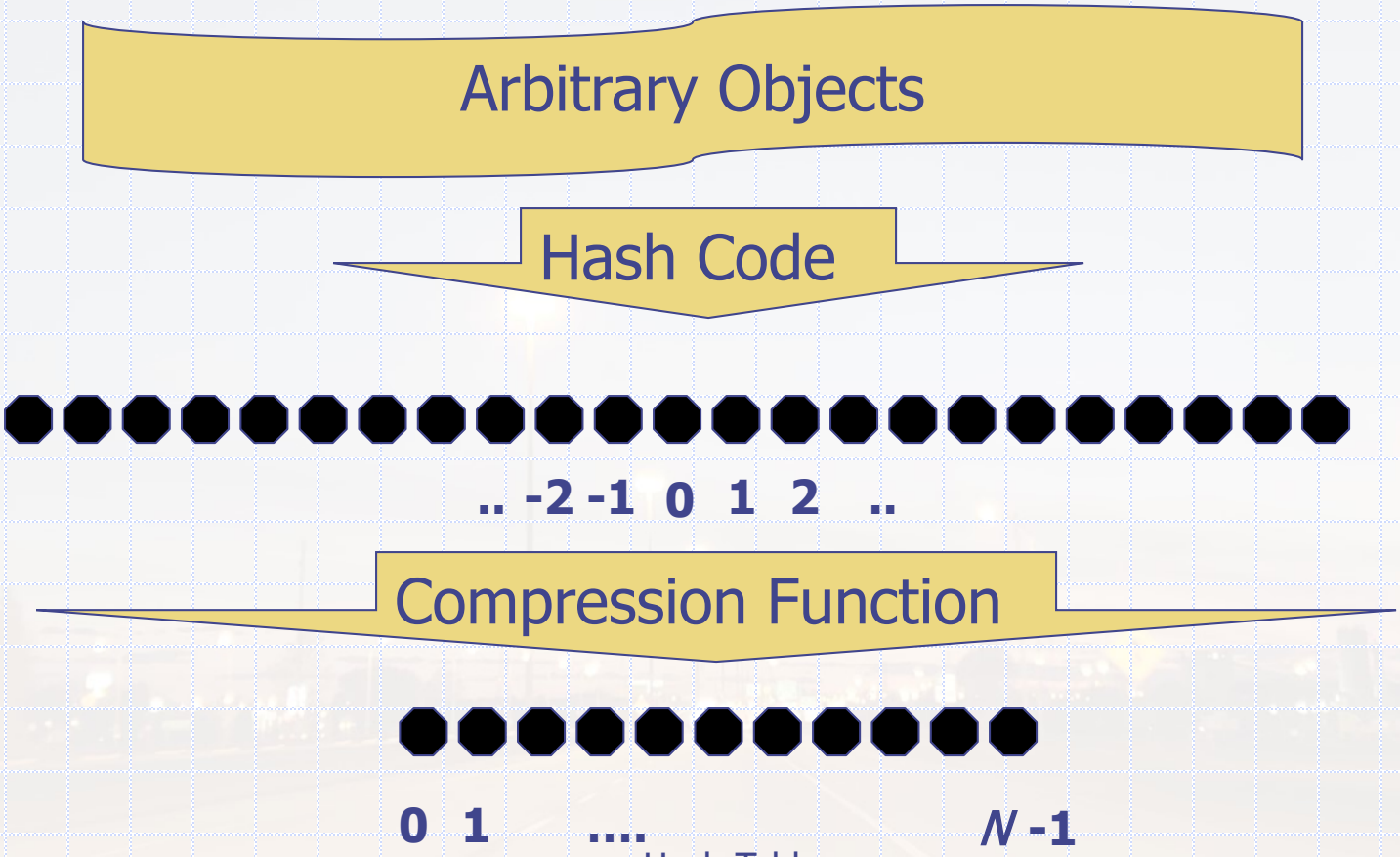
- The hash code is applied first, and the compression function is applied next on the result, i.e.,

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

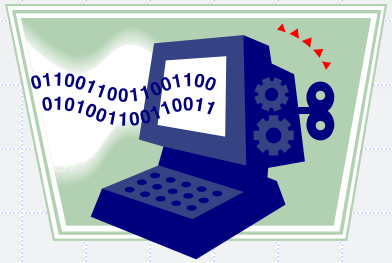
- The goal of the hash function is to "disperse" the keys in an apparently random way.

Hash Functions

- The two parts of a hash function: a hash code and a compression function to a hash value, can be illustrated as follows:

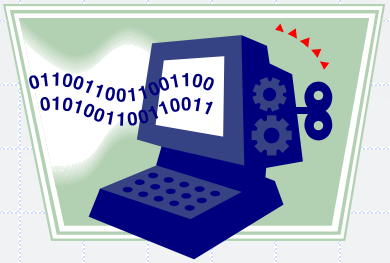


Hash Codes



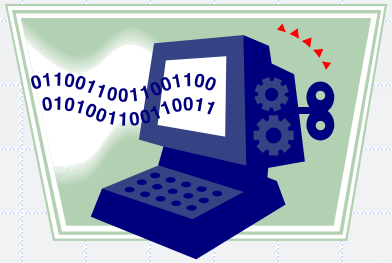
- The hash code set should avoid collision as much as possible.
- Additionally, to be consistent, similar keys should result in the same hash code.
- Hash codes can be obtained in different ways. Some of such are as follows:
  - Use of Memory Address (Hash Codes in Java)
  - Integer Casting
  - Summing Components
  - Polynomial accumulation

Hash Codes (cont.)



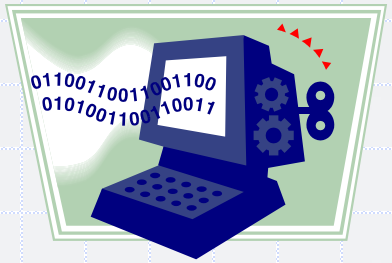
- Memory address:
  - We reinterpret the memory address of the key object as an integer.
  - Java utilizes such approach; the Object class has a method called hashCode(), which returns a 32-bit integer (the address), which is the default hash code of all Java objects.
  - Good in general, except for numeric and string keys.
  - Two similar strings (at two addresses) may not have the same hash code.

Hash Codes (cont.)



- Casting to an Integer:
  - If the key can be casted to an integer, then we can obtain the hash code by simply casting the key.
  - This is generally suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and char in Java).
  - Variables of type float can also be converted to integers if a conversion method, such as floatToIntBits() in Java, exists.

Hash Codes (cont.)



- Summing Components:
  - Integer casting is not suitable if the variables are of types that are longer than integer representation, such as long and double.
  - An alternative in such cases is to partition the bits of the key into components of fixed length (e.g., 16 or 32 bits), then sum these components (ignoring overflows) to obtain an integer representation.
  - That is, view the representation as a *k-tuple* $(x_0, x_1, \ldots, x_{k-1})$, then find the sum of these components.
  - This technique can also be extended for any object that has binary representation. In that case, we just need to break these representations into smaller parts of integer lengths, then obtain the code by summing these parts.

Hash Codes (cont.)

- Polynomial accumulation:
  - The Summing Components technique is not suitable for character strings or other variable-length objects, where the order of the components in the *k-tuple* $(x_0, x_1, \ldots, x_{k-1})$ is significant.
  - In such cases, strings such as: "temp01" and "temp10", or "stop" and "pots" will result in collision since the sum is the same.
  - A better hash code that somehow takes the position of the components into consideration is needed.
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits): $a_0 a_1 \ldots a_{k-1}$

Hash Codes (cont.)

- Polynomial accumulation:
  - We then evaluate the polynomial
    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{k-1} z^{k-1}$
    for some fixed value z.
  - This would effectively take care of the order of the bits.
  - *Horner's rule*: the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{k-1} z^{k-1}$$
    can actually be written as:
    $$p(z) = a_0 + z(a_1 + z(a_2 + \ldots + z(a_{k-3} + z(a_{k-2} + z a_{k-1})) \ldots))$$

Hash Codes (cont.)

- Polynomial accumulation:
  - Studies suggested that a good value of z (to reduce collisions) would be 33, 37, 39 or 41.
  - For instance, 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 (n successive computation, each from the previous one in $O(1)$ time).
  - Many Java implementations use polynomial accumulation as the default hash function for strings.
  - Some Java implementations only apply polynomial accumulation to a fraction of the characters in long strings.

Compression Functions

- Hash codes may be out of the indices of the hash table's array, so compression is needed to convert these codes to integers within the $[0 .. N-1]$ range.
- A good compression function is the one that minimizes the possible number of collisions in a gives set of hash codes.

Compression Functions

- A simple compression function is the ***division method***.
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
    - This helps spread out the distribution of hashed values.
    - For instance, assume hash codes {200, 205, 210, 215, 220, …, 600}. If N is chosen as 100, then each hash code will collide with three others. If however N is 101, then there will be no collisions.
    - The specifics have to do with number theory and belongs to the scope of other courses

Compression Functions

- A more sophisticated compression function is multiply-add-and-divide (MAD).

- Multiply, Add and Divide (MAD):
  - This method maps an integer y as follows:

$$h_2(y) = (ay + b) \bmod N$$

    - a and b are nonnegative integers such that

$$a \bmod N \neq 0$$

    - Otherwise, every integer would map to the same value b

Compression Functions

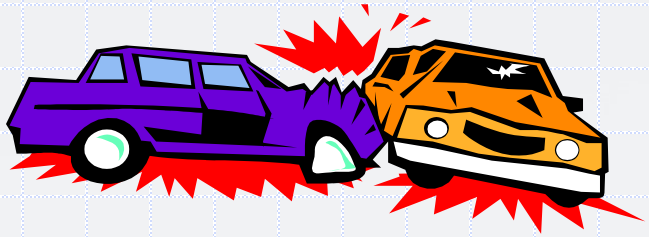
- Actually, a better MAD compression would be as follows:

$$h_2(y) = [(ay + b) \mod p] \mod N$$

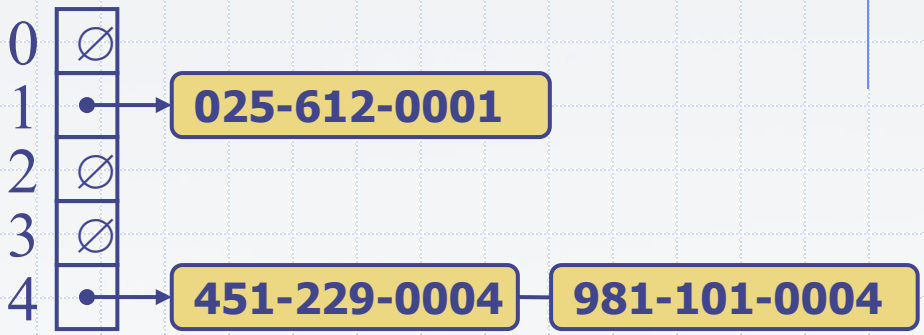
  - p is a prime number larger than N
  - a and b are nonnegative integers randomly chosen from $[0 .. p-1]$ with $a > 0$

  - The above MAD compression gets us closer to having a good hash function.
  - A "good" hash function should ensure that the probability of two different keys getting hashed to the same bucket is $1/N$. In other words, the entries will be "thrown" into the array uniformly.

Collision Handling



- Collisions occur when different elements are mapped to the same cell.



- Separate Chaining: let each cell in the table point to a map implemented as a linked list of entries.
- Separate chaining is simple, but requires additional memory outside the table.

Separate Chaining

- The array indices point to maps implemented using linked list. The operations of these individual maps are defined as usual:
  - get(k): if the map M has an entry with key k, return its associated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, replace the value associated with k with v and return that old value
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null

Map with Separate Chaining

The operations of map ADT using has tables implementations can be defined as follows (Notice that we 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
```

Performance of Separate Chaining

- Assume n entries are inserted into the hash with Array of size N.
- If the hash function is good, then the entries are uniformly distributed; that is the size of each bucket is n/N.
- This value n/N is called the ***load factor*** of the hash table. This load factor should generally be bounded by some small constant, preferably below 1.
- Thus, with a good hash function, the expected time of get(), put() and remove() is ***O[n/N].***

  ➔ **provided that *n* is *O(N)*, the operations can run in *O(1)* time.**

Open Addressing

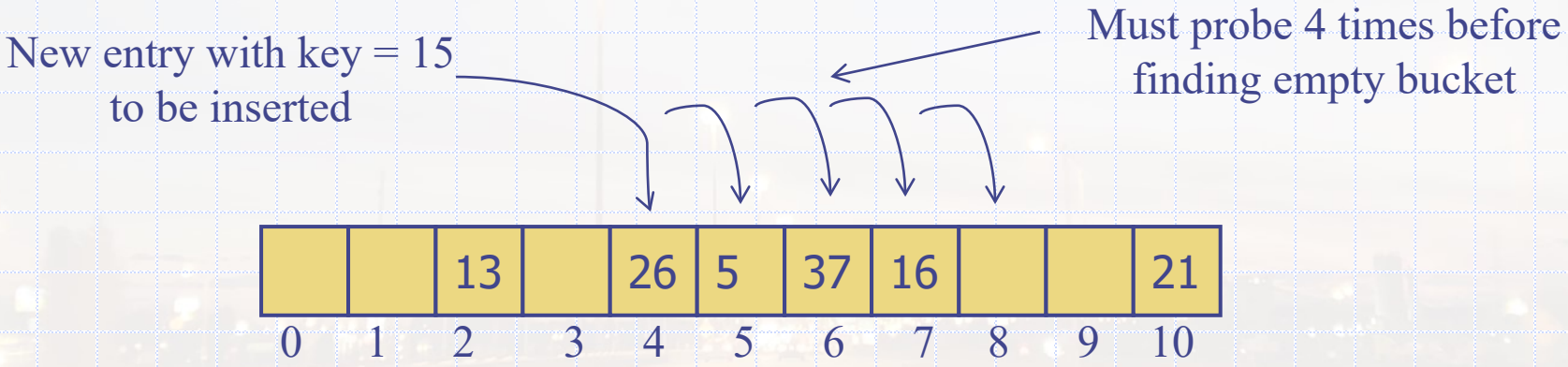
- *Separate Chaining* has many nice properties, however it has one slight disadvantage:
  - It requires the utilization of an auxiliary data structure (the lists) to hold entries with colliding keys.
  - This places space requirements, which could be significant if space is limited; i.e. for handheld devices.
- Alternatively, we can store the entries directly in the bucket (one entry per bucket).
- That will save space, however it requires a bit more complexity to deal with collisions.
- There are several variants to this approach, collectively referred to as *open addressing* schemes. Open addressing requires that the load factor is always at most 1 and that the entries are stored directly into the cells of the bucket array itself.

Linear Probing

- *Linear Probing* is a simple open addressing method for collision handling.
- If the hash code for entry (k, v) indicates that it should be inserted into the bucket $A[i]$ but $A[i]$ is already occupied by another entry, then try the next entry (circularly), which is $A[(i+1) \bmod N]$.
- If $A[(i+1) \bmod N]$ is also occupied, then try $A[(i+2) \bmod N]$, and so on, until you find a empty bucket that can accept the entry.
- Of course, the other methods, such as $\text{get}()$, have to be changed. In particular, in order to find the entry, either find an entry with *key* = k at the expected bucket, or continue to examine the consecutive buckets until an empty bucket is found.

Linear Probing

- Example:
  - Assume $N = 11$ and the hash function uses *division* for compression; that is
    $h(k) = k \bmod 11$
- Note: For easier illustration, the figure shows only the keys (not the entire entries)

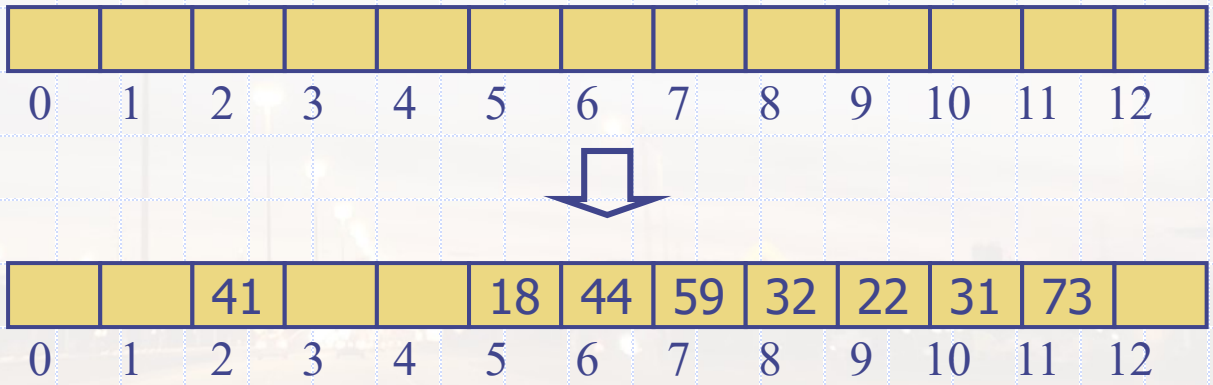


Linear Probing

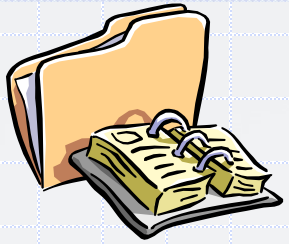
- Each table cell inspected is referred to as a "probe".
- Colliding items lump together (forming a cluster), and causing future collisions to cause a longer sequence of probes (longer clusters).

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

18 % 13 → 5
41 % 13 → 2
22 % 13 → 9
44 % 13 → 5
59 % 13 → 7
32 % 13 → 6
31 % 13 → 5
73 % 13 → 8



Search with Linear Probing



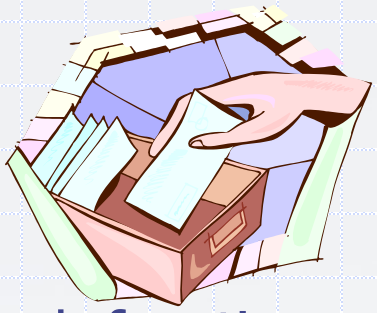
- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - All N cells have been unsuccessfully probed

```
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
```

Updates with Linear Probing

- Deletion of entries would require considerable amount of shifting to adjust the array; however this can be avoided.
- To handle insertions and deletions, we introduce a special object, called $AVAILABLE$, which replaces deleted elements.
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, v) is found, we replace it with the special item $AVAILABLE$ and we return element v
  - Else, we return *null*
- put(k, v)
  - 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
  - If a cell i is found, we store (k, v) in that cell i

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:

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

  where, $i = h(k)$
  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.
  - For instance, if the key is 31 and N = 13, then $h(k)$ is 5. If $d(k)$ is 4, then inserting 31 will then attempt to insert at the following cell #s in order: 5,9,0,4,8,12,3,7,11,2,6,10, 1, which are all the cells
- The second hash function $d(k)$ is chosen in a way that would minimize clustering as much as possible.
- Common choice of compression function for the secondary hash function:

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

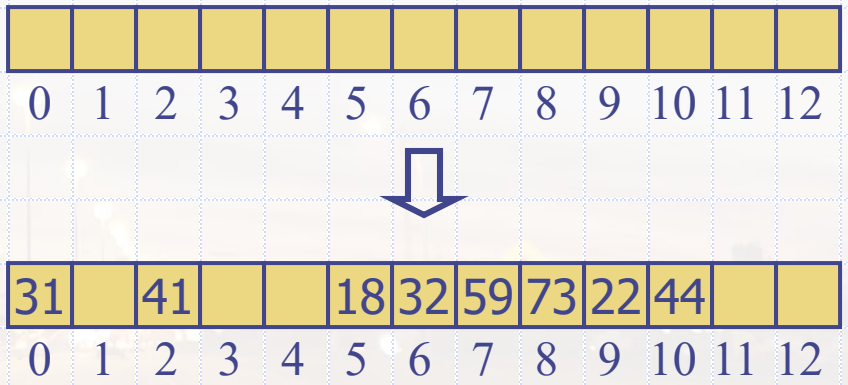
  where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ are

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

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		



Quadratic Probing

- Quadratic probing is another example of Open Addressing schemes.

- With Linear Probing, for a key *k*, the entries are attempted at the following locations in the array until an open location is found:

$$h(k),\ h(k)+1,\ h(k)+2,\ h(k)+3,\ h(k)+4,\ h(k)+5,\ \ldots..$$

- With Quadratic Probing, the insertions are attempted at the following locations:

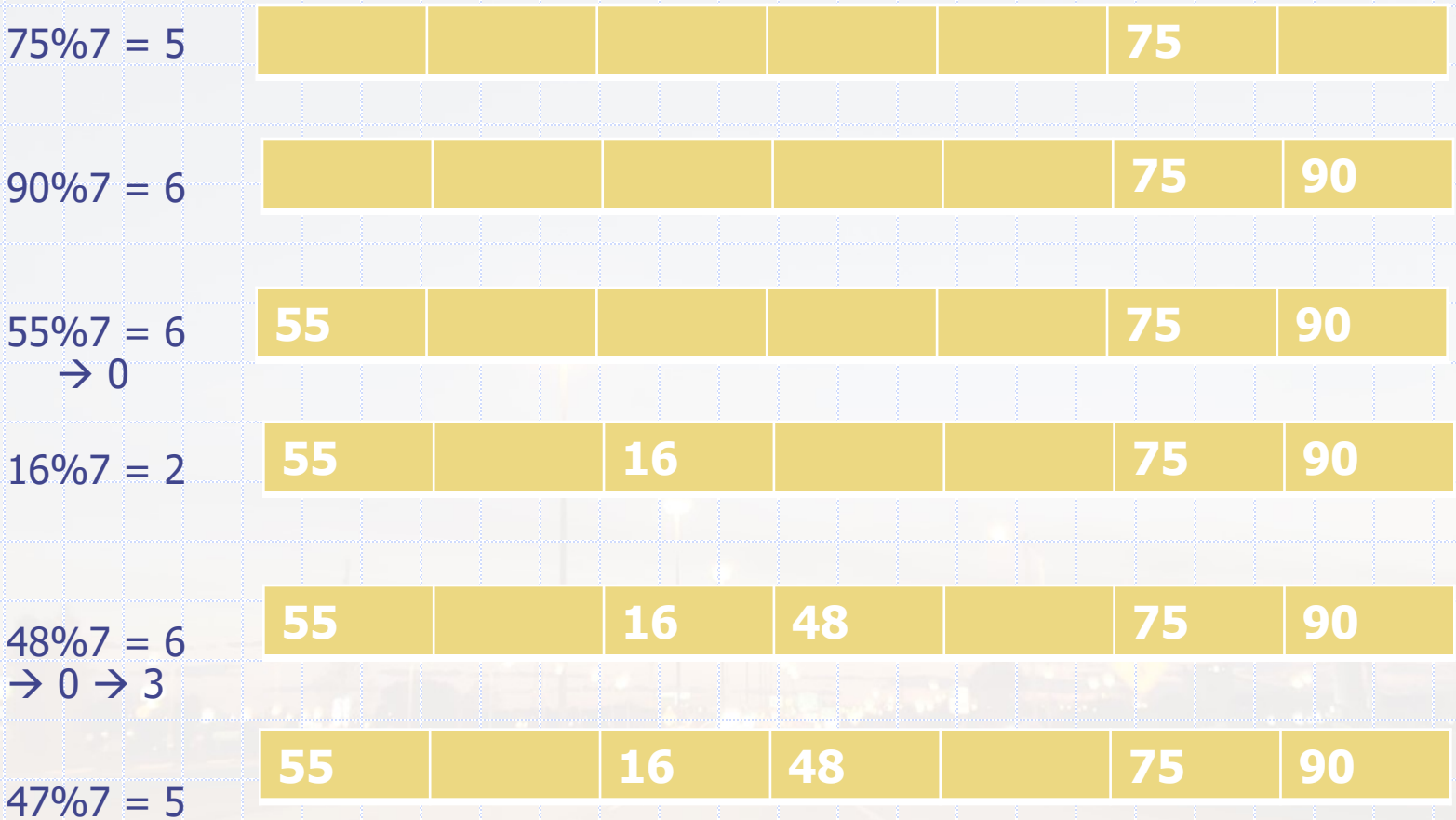
$$h(k),\ h(k)+1^2,\ h(k)+2^2,\ h(k)+3^2,\ h(k)+4^2,\ h(k)+5^2,\ldots..$$

- Since this avoids the clustering problem, Quadratic Probing can be more efficient.

➔ However, are there cases when Quadratic Probing can lead to serious problems?

Quadratic Probing

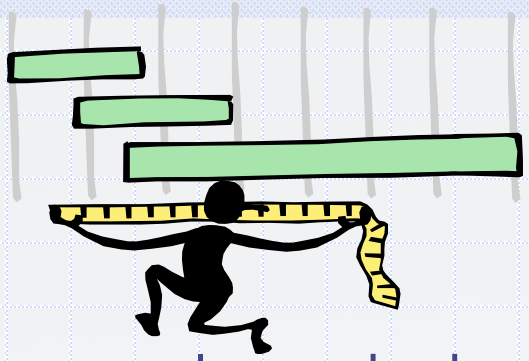
- Quadratic probing may fail: Assume N = 7. Insert the following keys:
  75, 90, 55, 16, 48, 47



Analysis of Open Addressing

- Open addressing schemes save space over separate chaining.
- However, they are not necessarily faster; in experimental and theoretical analysis , separate chaining is either competitive or faster depending on the load factor of the table.
- Additionally, the number of entries that can be inserted in the hash table using separate chaining is not restricted to the size of the array bucket.
- Consequently, if space is not a significant issue, separate chaining is the preferred collision-handling method.

Performance of Hashing

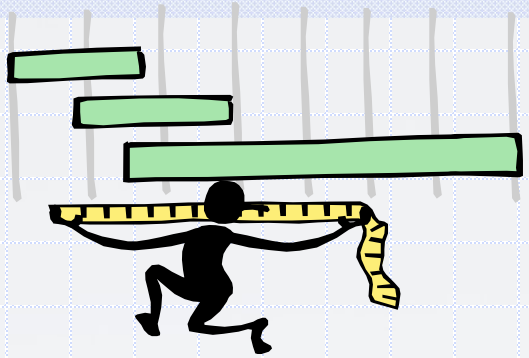


- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time.
- The worst case occurs when all the keys inserted into the map collide.
- The load factor $\alpha = n/N$ affects the performance of a hash table.
- Assuming that the hash values are like random numbers, it can be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

  $$1 / (1 - \alpha)$$

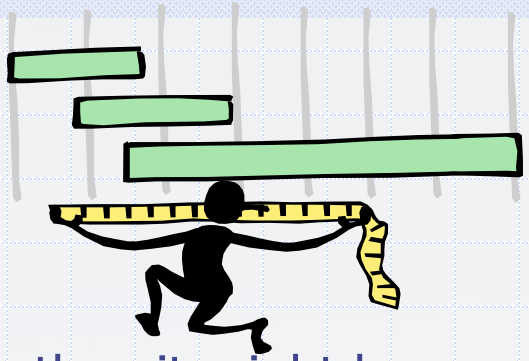
  - In other words, the smaller the load factor is, the smaller the number of entries in a probes will be. Larger load factors will result in larger/longer probes.

Performance of Hashing



- The **expected** running time of all the dictionary ADT (to be discussed next) operations in a hash table is $O(1)$.
- In practice, hashing is very fast provided the load factor is not close to 100%.
- In specific, experiments and average-case analysis suggested that we should maintain the load factor, α, at a value < 0.5 for open addressing and < 0.9 for separate chaining.
  - For instance, Java uses a threshold of 0.75 as the maximum load factor, and rehashes if the load factor exceeds that.

Performance of Hashing



- If this value significantly exceeds such limits, then it might be better to resize the entire table.
- We refer to this as ***rehashing*** to a new table.
- In such case, it is advisable to double the size of the array so we can amortize the cost of rehashing all the entries against the time used to insert them.
- It is also advisable to redefine a new hash function to go with the new array.

➔ Even with periodic rehashing, hast tables are efficient means of implementing maps.