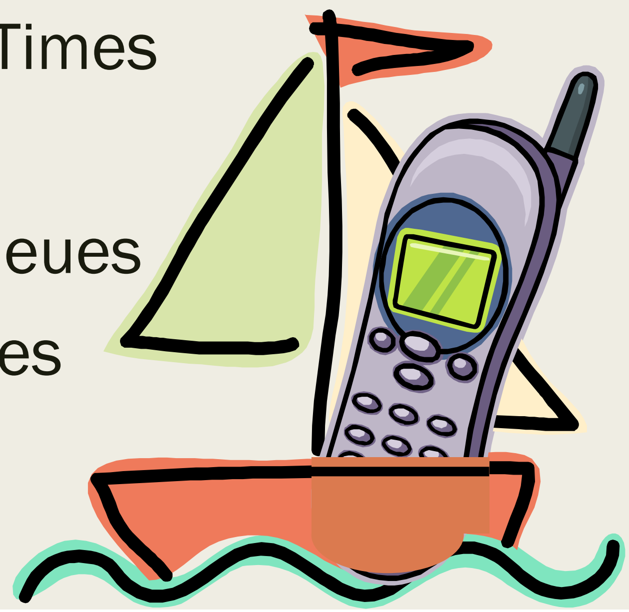
Hash Tables & Priority Queue

ISTE-222

Computer Problem Solving in the Information Domain III

Outline

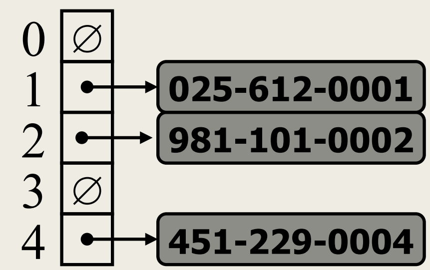
- Hash Tables
- Hash Implementation
- Map/Dictionary Running Times
- Priority Queues
- Applications of Priority Queues
- Sorting with Priority Queues



Hash Tables

Another way to implement

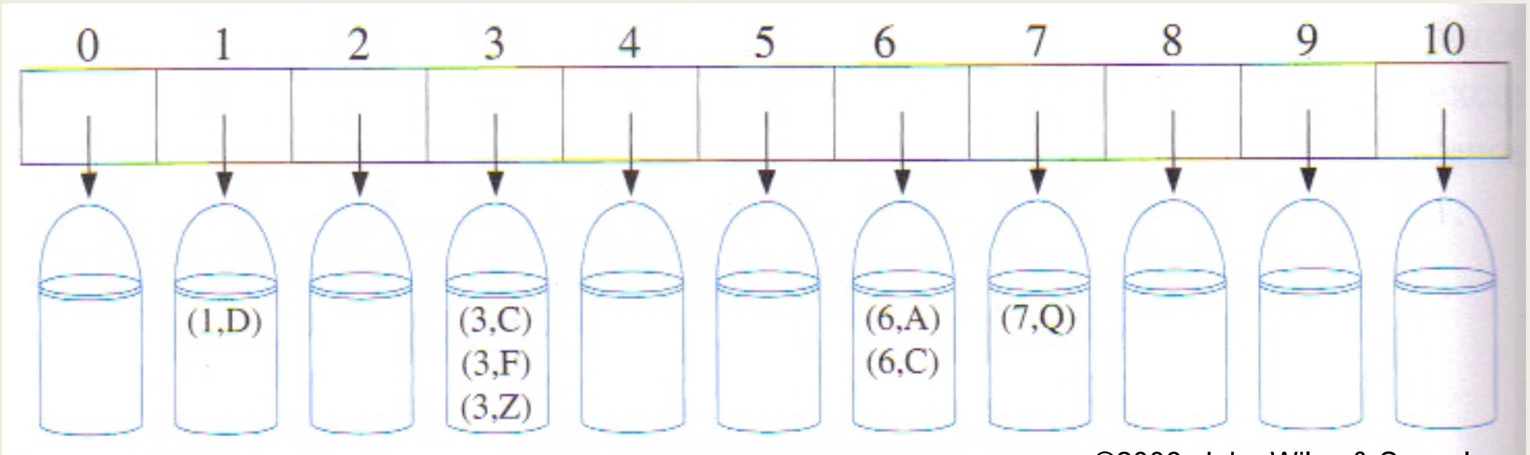
Maps and Dictionaries



hash (noun):
A hash (a.k.a. “hash value” or “message digest”) is a number generated from a text string (or from some other large piece of data). The hash is substantially smaller than the text itself, and is generated by a formula in such a way that it is unlikely that some other text will produce the same hash value. Hashes cannot be reversed back to the original information.

From Lecture 19

Bucket Arrays



Imagine that we had an array that stored a bucket at each position of the array. We could then add some (key,value) pair entries into this bucket array according to the following rule:

*Put entries with key **k** into the bucket at position **k**.*

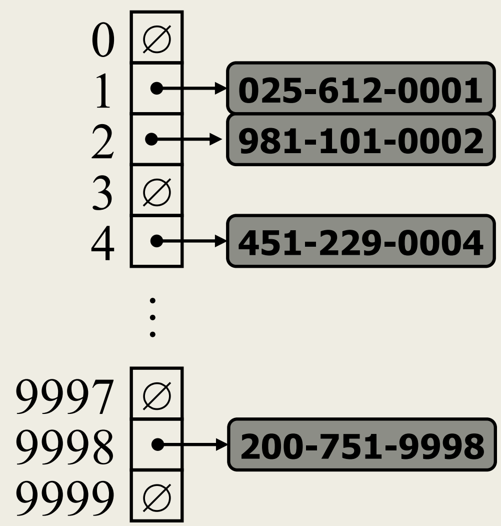
To retrieve an entry, we'd know which bucket to look for it in.
How do we pick buckets if key values aren't all small integers?

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 $i = h(k)$

Hash Table Example

- Let's 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 $\boldsymbol{h}(\boldsymbol{x}) = \text{last four digits of } \boldsymbol{x}$



Ten digits used in the picture to avoid using the real SSN of someone.

Hash Functions

- 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 Function: Two Parts

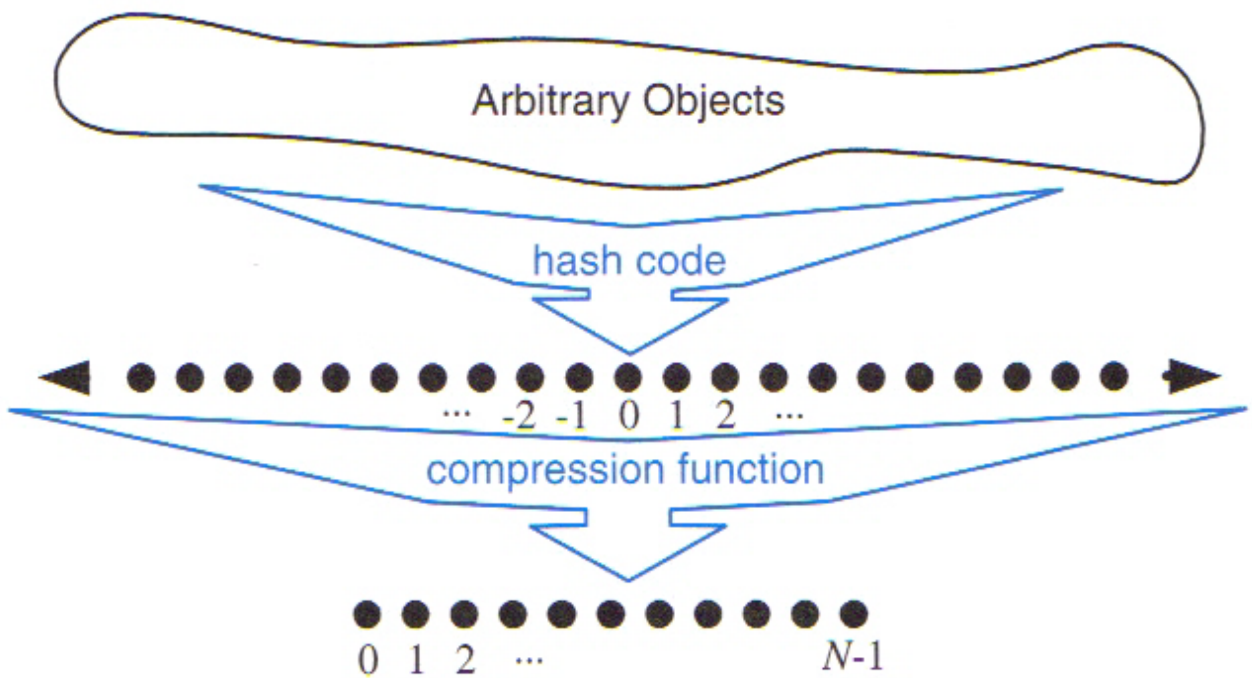


Figure 9.3: The two parts of a hash function: a hash code and a compression function.

Patterns in Data

- Key Goal of Hash Function = eliminate patterns in the set of keys entered into the table.
- Patterns could cause data to clump in the table and therefore have more collisions.
- We'd like to use all of the table's cells.
  - *This will help us to make better use of the space.*
- A good hash function will scatter/disperse.

Some Common Hash Codes (1)

- 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 ≤ '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) 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 in Java)*

Java Code for Hash Code

Component Sum Code:

```
static int hashCode(long i) {
   return (int)((i >> 32) + (int) i);
}
```

Some Common Compression Functions

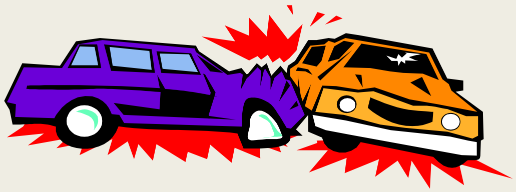
- Division:
  - $h_2(y) = y \bmod N$
  - *The size* N *of the hash table is usually chosen to be a prime*
  - *The reason has to do with number theory and is beyond the scope of this course*

- 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

One downside: Input values that were adjacent produce output values that are adjacent… We might want more scattering than this (to avoid collisions, discussed later).

Collision Handling

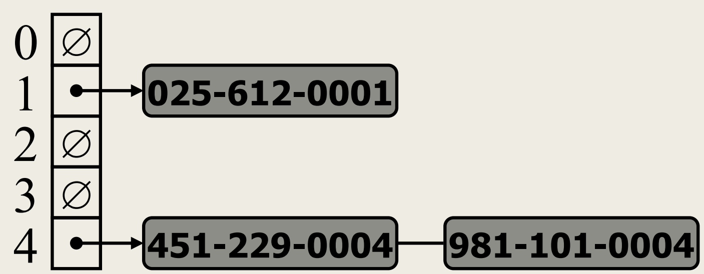


- Collisions occur when different elements are mapped to the same cell: two different keys could have the same hash value.
- There are two ways to handle this situation in a hash table:

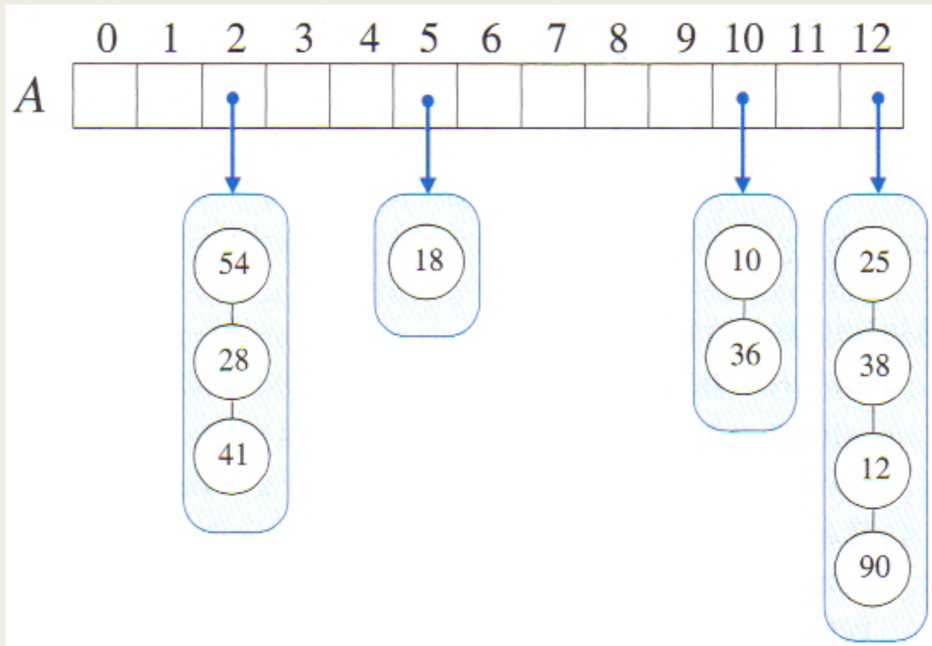
1. **Separate Chaining**: let each cell in the table point to a linked list of entries that map there
2. **Open Addressing:**
   - ***Linear Probing:*** *If the first spot you try to put something is full, put it in the next available.*
   - ***Double Hashing*** *uses another function to see what position to try next*

Separate Chaining

- **Separate Chaining:**
  - *Let each cell in the table point to a linked list of entries that map there*
  - *Separate chaining is simple, but requires additional memory outside the table.*



Separate Chaining Example



A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, the image does not show the values associated with the keys.

Map Methods with Separate Chaining

Algorithm get(k):
Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map
return $A[h(k)]$.get(k) {delegate the get to the list-based map at $A[h(k)]$}

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return **null**
$t = A[h(k)]$.put(k,v) {delegate the put to the list-based map at $A[h(k)]$}
if t = **null then** {k is a new key}
 $n = n + 1$
return t

Delegate operations to a list-based map at each cell; similar idea to a 2D array…

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map
$t = A[h(k)]$.remove(k) {delegate the remove to the list-based map at $A[h(k)]$}
if $t \neq$ **null then** {k was found}
 $n = n - 1$
return t

Linear Probing

- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell. The colliding item is placed in a different cell.
- Each time we inspect a cell (to see if it is the one we're looking for or to see if it's free) is called a "probe."
- Colliding items can lump together; so, after several collisions, we may need to do a long sequence of probes to find an empty cell or to find the key we looking for.

Example:

– ***h*(*x*) = *x* mod 13**

– *Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order*

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

Linear Probing Example

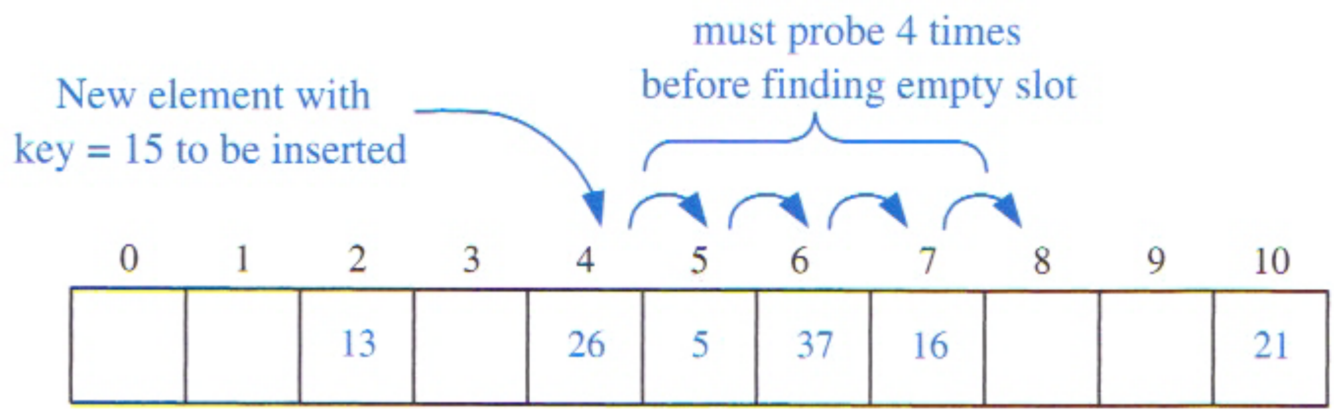


Figure 9.5: Insertion into a hash table with integer keys using linear probing. The hash function is $h(k) = k \bmod 11$. Values associated with keys are not shown.

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 (the array must be full!)

```
Algorithm get(k)
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅
      return null
     else if c.key () = k
      return c.element()
    else
      i ← (i + 1) mod N
      p ← p + 1
  until   p = N
  return null
```

Problem with deletes and "get" (1)

- If we were to delete items from our hash table (that uses linear probing), consider case #2 of "get":

During a "get" operation after some delete, we might stop looking for an item even though it is actually in the heap – it might just be after this newly created gap.

We start at cell ***h***(***k***).
We probe consecutive locations until one of the following occurs:

1. key ***k*** is found, or
2. empty cell is found, or
3. ***N*** cells have been unsuccessfully probed (the array must be full!)

Problem with deletes and “get” (2)

- Consider delete of 18 from our example.
- Now, what if we look for 44?

Example:

– $h(x) = x \bmod 13$

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

First, try looking at index 5 (this is 44 mod 13). Whoops… It’s empty. According to the definition of the “get” operation, we should stop looking. But that means we won’t see that 44 is in our table; it’s just a little later. We had to put it there because index 5 used to be full.

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special value, called ***AVAILABLE***, 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* ***AVAILABLE*** *and we return element* o
  - *Else, we return* ***null***

- 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 ***AVAILABLE***, or
- N cells have been unsuccessfully probed
  - *We store entry* (k, o) *in cell* i

Use of "AVAILABLE"

- Consider remove(18) from our example.
- Now, what if we look for 44?

Example:

- $h(x) = x \bmod 13$

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

First, try looking at index 5 (this is 44 mod 13). It's "AVAILABLE". According to the definition of the "get" operation, we should keep looking. (We only stop if we encounter an empty node… "AVAILABLE" nodes aren't *empty* nodes.) So, now we'll try index 6, and we'll see 44 is there.

Double Hashing

- "Bunching up" can often happen with linear probing.
- Double hashing uses a secondary hash function $\boldsymbol{d(k)}$ and handles collisions by placing an item in first available cell of the series

$$(\ \boldsymbol{h}(\boldsymbol{k}) + \boldsymbol{j} \cdot \boldsymbol{d}(\boldsymbol{k})\ ) \bmod N$$
$$\text{for } \boldsymbol{j} = 0,\ 1,\ \ldots,\ N-1$$

- $\boldsymbol{d}(\boldsymbol{k})$ shouldn't have zero values, and table size $\boldsymbol{N}$ should be prime to allow probing of all the cells.
- Common choice for $\boldsymbol{d}(\boldsymbol{k})$:

$$\boldsymbol{d}\,(\mathbf{k}) = \boldsymbol{q} - (\ \boldsymbol{k} \bmod \boldsymbol{q}\ )$$

where $\boldsymbol{q}$ is a prime less than N

- Possible values for $\boldsymbol{d}\,(\boldsymbol{k})$:

$$1,\ 2,\ \ldots,\ \boldsymbol{q}$$

When j=0, this is just h(k)… In other words, we first try to place it using original h().

You can think of d(k) as the "skip" amount. (Linear probing always skips by 1.)

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 in this order: 18, 41, 22, 44, 59, 32, 31, 73.

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
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:* $1 / (1 - \alpha)$
  - *This is* **average case** *performance.*

"average case"

- 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*

Average Case Running Time

- Hash tables are the first real-world data structure that we've discussed in which the average case performance is what people generally think of as important. Why?
  - *We can use number theory and probabilities to understand the expected running times.*
  - *If we make the hash bigger (and thus reduce the load factor for given set of input objects it must store), then we can reduce the load factor.*
- Thus, we know how to make the $O(1)$ average case running time more likely. (We have some control). O(1) is such a good running time that it's worth using hash tables since we can often get O(1) even if we sometimes (rarely) could have a bad case of O(n).