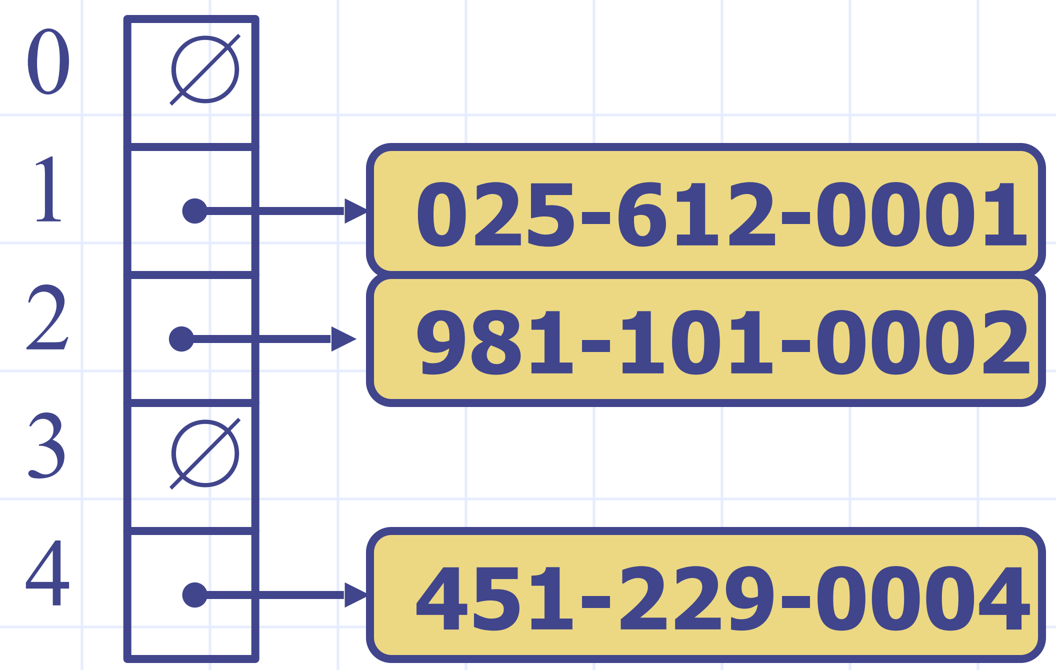
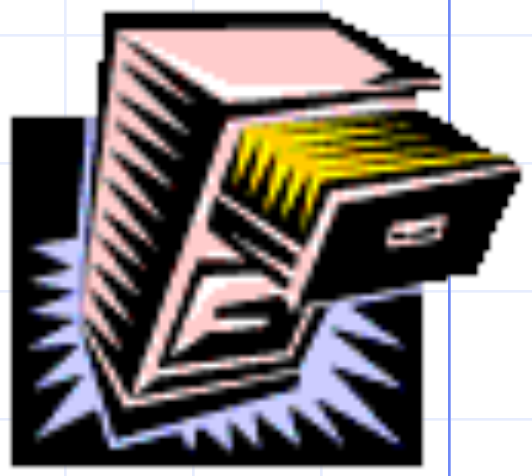
Hash Tables

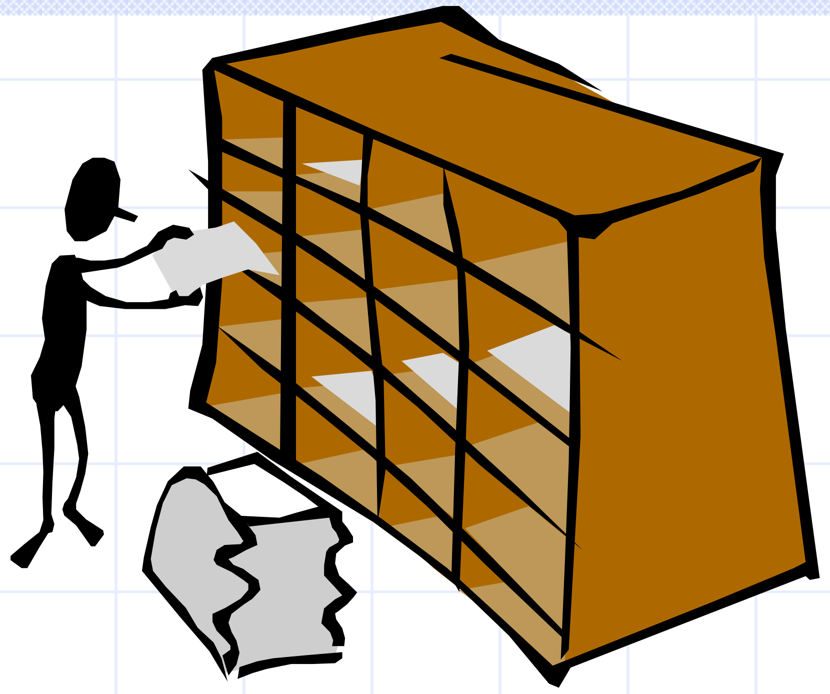


Hash Tables



- find(k): if the table has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the table; if key k is not already in table, then return null; else, return old value associated with k and replace the value with v
- erase(k): if the table has an entry with key k, remove it and return its associated value; else, return null
- size(), empty() : (obvious)
- entrySet(): return a list of the (k,v) entries in table
- keySet(): return a list of the keys in table
- values(): return a list of the values in table

Hash Functions and Hash Tables



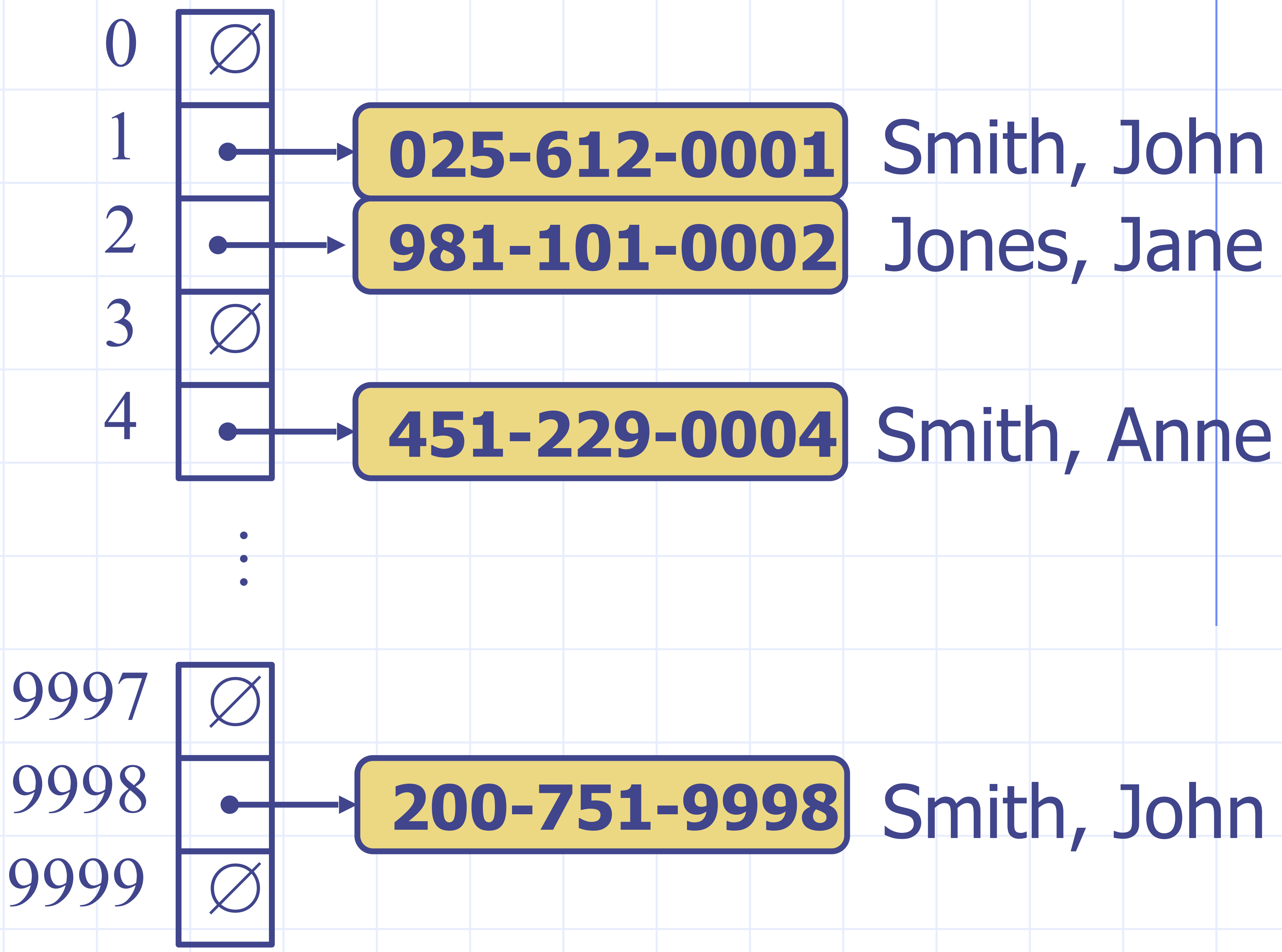
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:

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

  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 hash table, the goal is to store item (k, v) at index $i = h(k)$

Example

- A hash table storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- This hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) =$ last four digits of x



Notes:

- Often, a pointer to an object is stored in the table instead of actual objects.
- The value in the table may be duplicated. Here, there's more than one John Smith.

Hash Functions

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

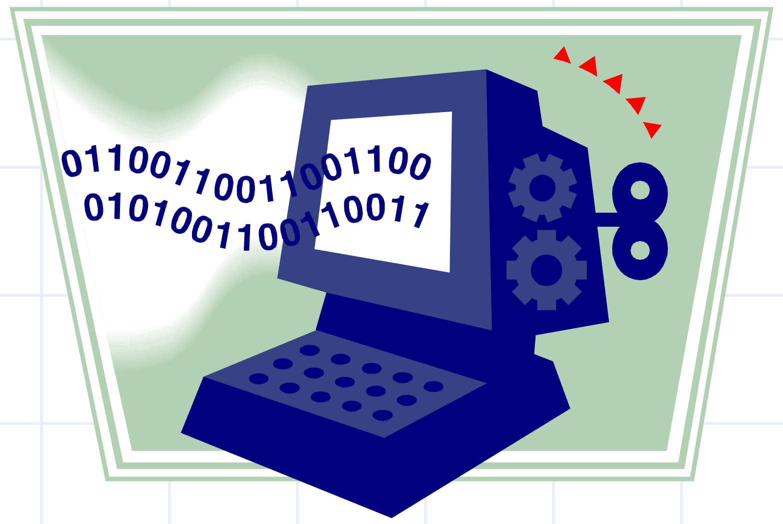
  h_2: integers $\rightarrow [0, N-1]$

- 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

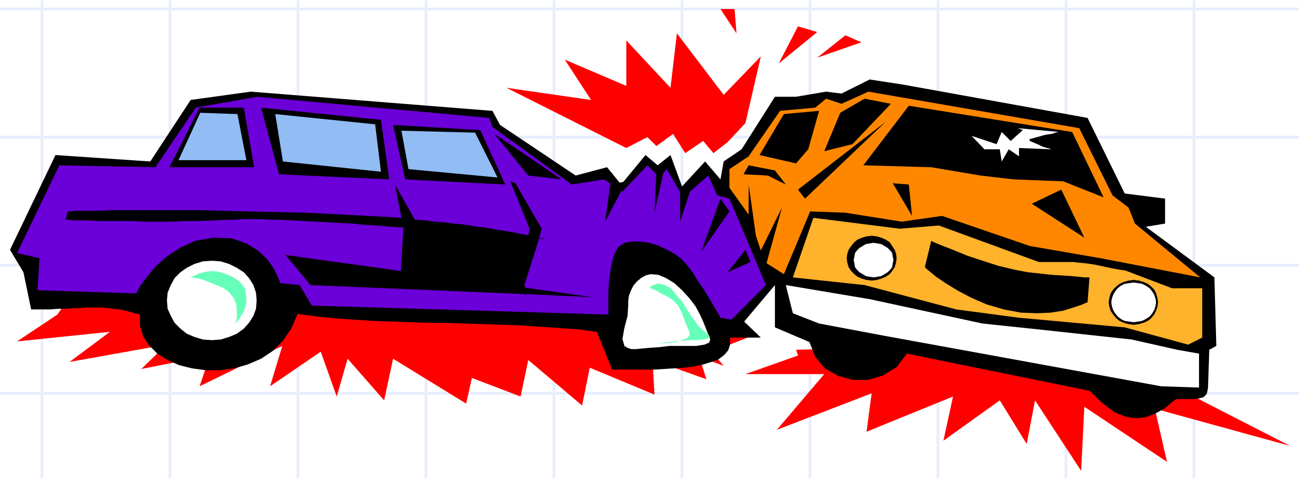


- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 8, 16 or 32 bits) and we sum the components (ignoring overflows)
    - example: sum the values of a character array (each character has an ASCII integer equivalent)
  - Suitable for keys of fixed length greater than the number of bits in an integer

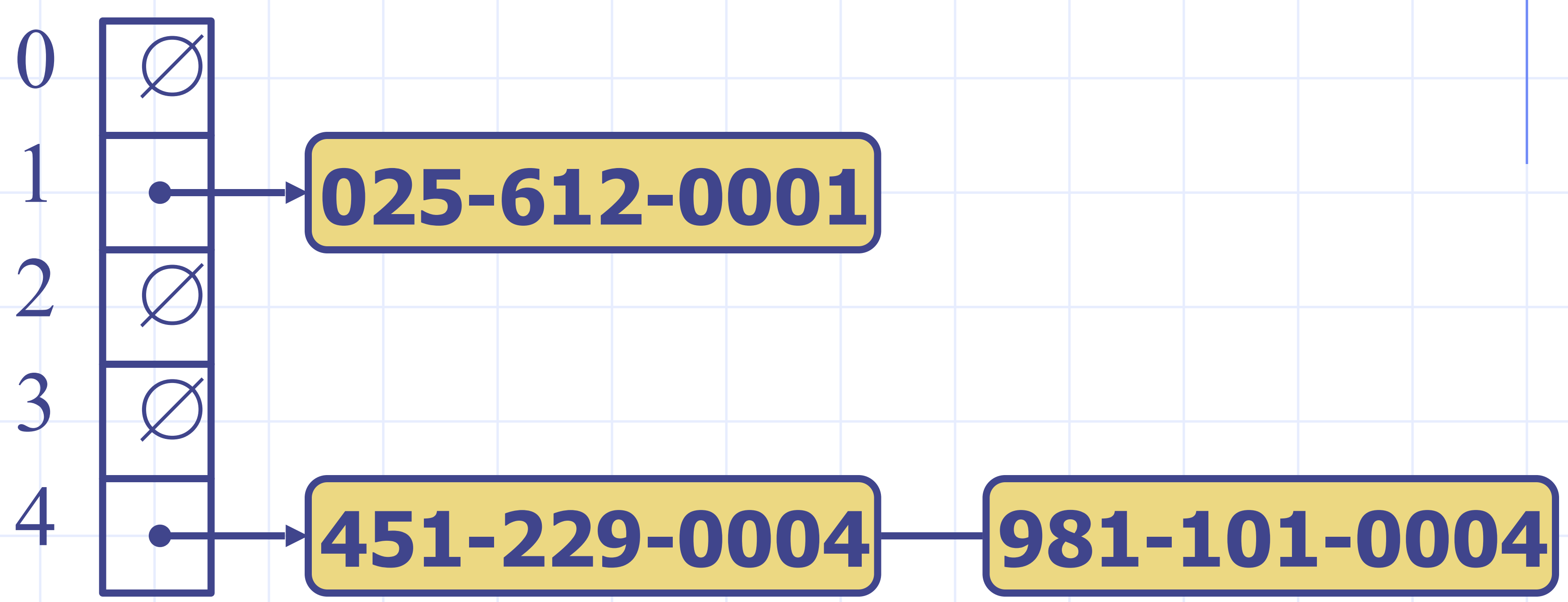
Compression Functions

- 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

Collision Handling



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table store a linked list of entries that map to the cell



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

Psuedocode for Separate Chaining

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

```
Algorithm find(k):
return A[h(k)].find(k)          //use set/list's find

Algorithm put(k,v):             // update node with key = k
t = A[h(k)].add(k,v)            // use set's add, returning old value
if t = null then                // if there was no old value
    n = n + 1                   // n is count of # values in table
return t

Algorithm erase(k):
t = A[h(k)].erase(k)            //use set's erase, returning value erased
if t ≠ null then                //if we erase something
    n = n - 1                   //… then there's one fewer item in table
return t
```

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

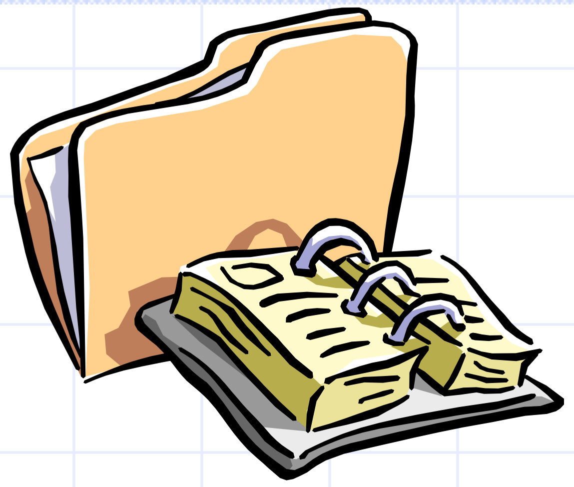
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

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

⇩

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

Search with Linear Probing



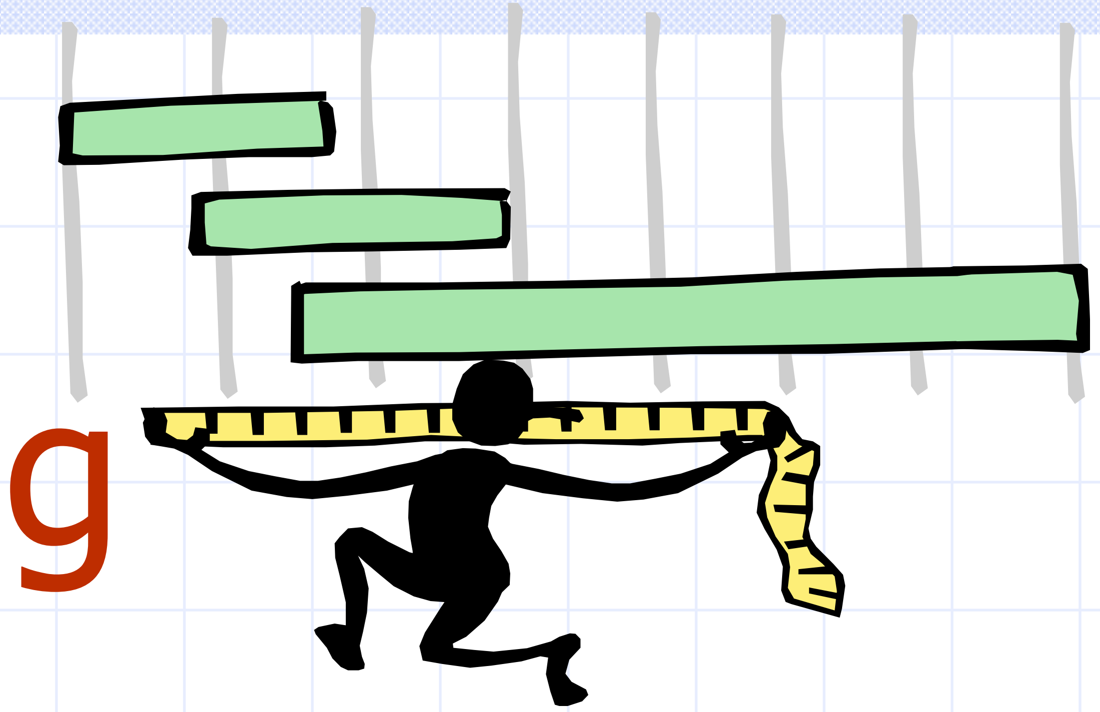
- Consider a hash table A that uses linear probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm find(k)
    i ← h(k)
    p ← 0   // number of probes
    do
        c ← A[i]
        if c = ∅
            return null
         else if c.key () = k
            return c.value()
        else
            // wrap around if needed.
            i ← (i + 1) mod N
        p ← p + 1
  while   p != N
    return null
```

Updates with Linear Probing

- To handle insertions and deletions, use a special object called ***AVAILABLE*** which replaces deleted elements
- erase(k)
  - search for an entry with key k
  - If such an entry (k, v) is found, we replace it with the special item ***AVAILABLE*** and return element v
  - Else, we return ***null***
- put(k, v)
  - throw an exception if the table is full
  - start at cell $h(k)$
  - 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 (k, v) in cell i

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 table 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 is
  $$1 / (1 - \alpha)$$
- The expected running time of all the 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