Hash Table

Outline

- motivation
- hash functions
- collision handling

Log File vs Search Table

Method	Log File	Search Table
size, isEmpty	O(1)	O(1)
keys, elements	O(n)	O(n)
findElement	O(n)	O(log n)
findAllElements	O(n)	O(log n+s)
insertItem	O(1)	O(n)
removeElement	O(n)	O(n)
removeAllElemts	O(n)	O(n)

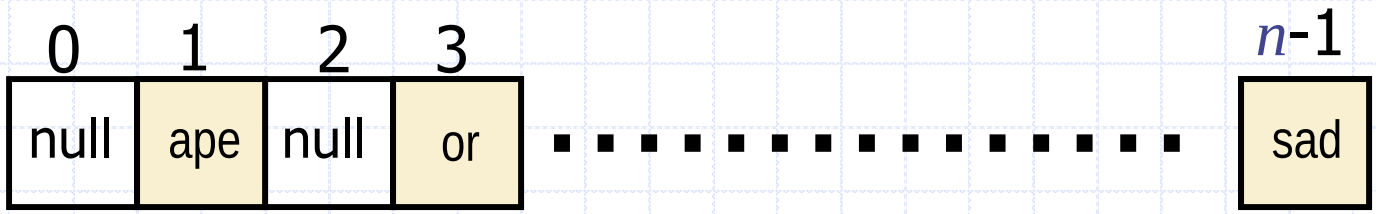
Hash Table Motivation

- Efficient operations, insert, search, delete, to work with a dynamic set of data.

Hash Table Motivation

- Suppose keys ***k*** are *unique* integers between 0 to ***n***-1
- Most efficient implementation (for this example, assume values are characters):

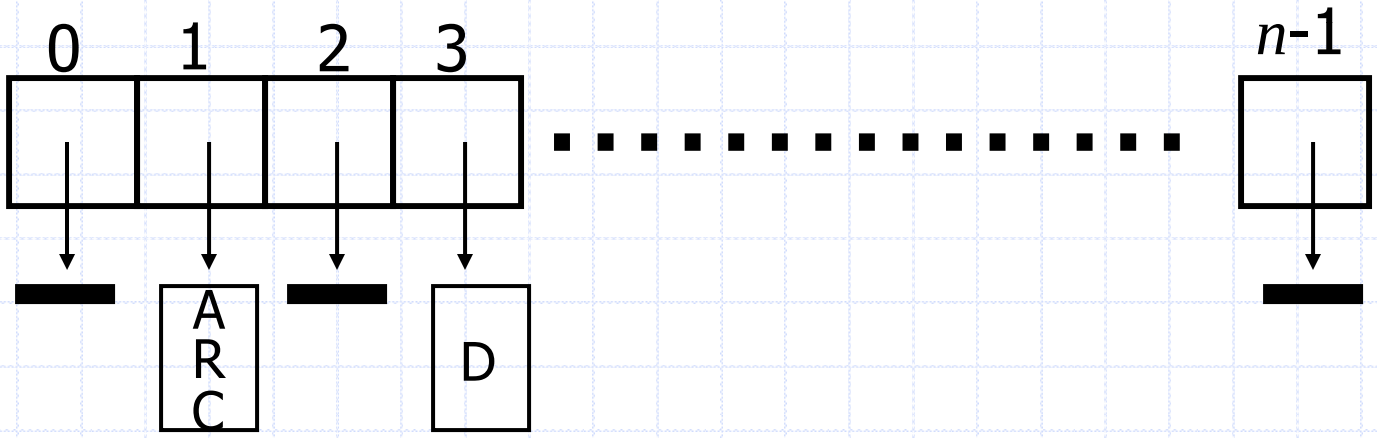
(1,"ape") (3,"or") (n-1,"sad")



- All operations (insert, find,…etc.) are O(1)
- need O(n) space

Hash Table Motivation

- What if keys are not unique?
  (1,A) (1,R) (1,C) (3,D)
- Can use bucket array:



- A bucket can be implemented as a linked list
  - as long as a constant number of entries (say not more than k) fall in the same bucket, all operations are still O(1)

Hash Table Motivation: Why Hashing?

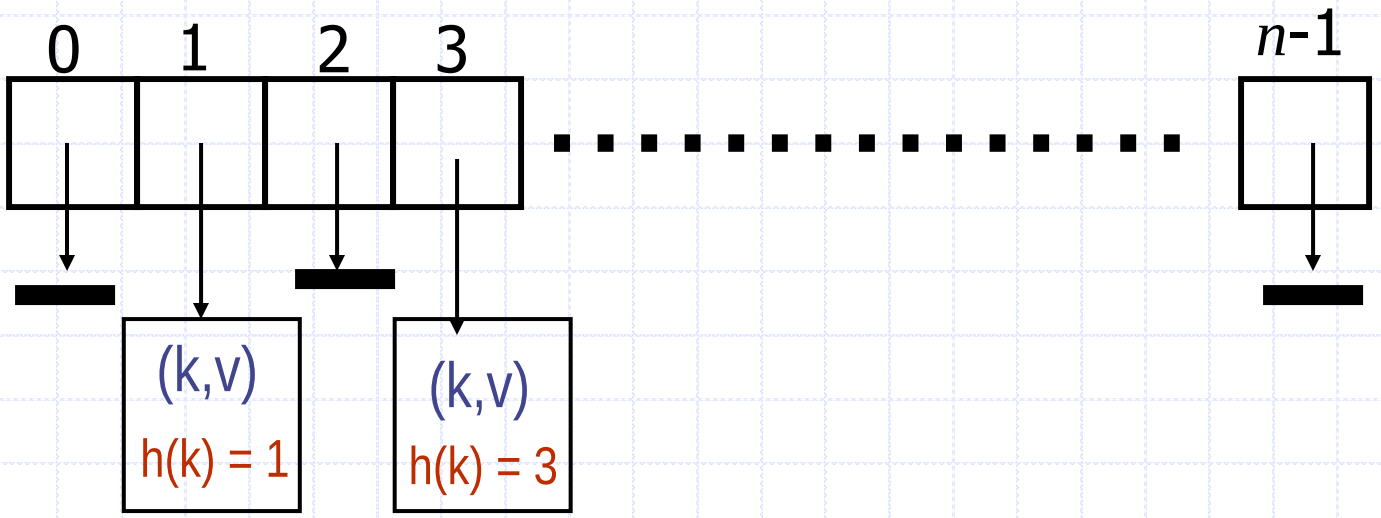
1. What can we do if knowing we'll have only at most 100 entries with integer keys but the keys are in range **0** to **1,000,000,000**?
   - still want O(1) insert, delete, find,
   - but don't want to use **1,000,000,000** memory cells to store only 100 entries
2. What can we do if **keys are not integers**?

- These 2 issues above motivate a **Hash Table** implementation of a dictionary
  - **insert**, **find**, **delete** will be in O(1) expected (average) time
  - **worst-case** time is O(n)

Hash Table: basic idea

- First map a key into a small integer range:

- Then put the entry with key K into the h(K) slot of the bucket array:



Hash Functions and Hash Tables

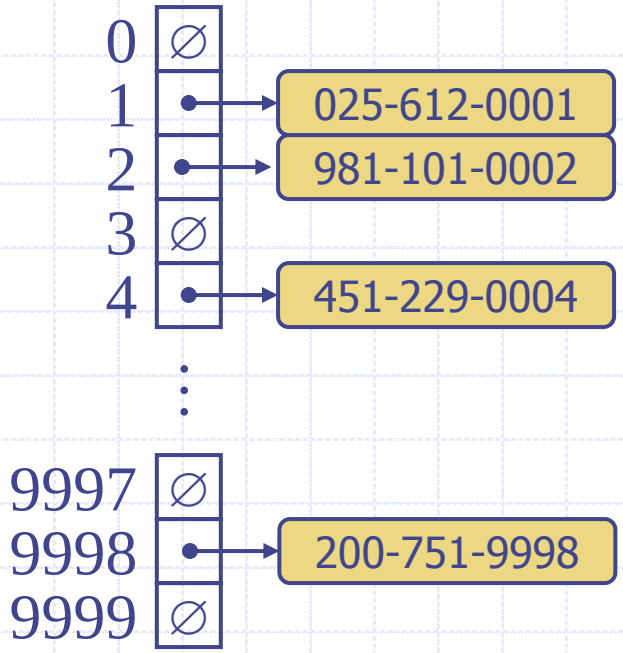
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

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

  is a hash function for integer keys x
- 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 dictionary with a hash table, the goal is to store an entry (k, v) at index $i = h(k)$

Example

- Suppose a company has 5,000 employees, want to store their information with key = SIN
  - Could store info under full 10 digit SIN, need array of size 9,999,999,999
- We design a hash table for a dictionary storing entries as (SIN, Name), where SIN (social insurance 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:
  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.,
  $h(x) = h_2(h_1(x))$
- In order to avoid **collisions** (2 keys assigned the same hash code), the goal of the hash function is to "disperse" the keys in an apparently random way
  - both **hash code** and **compression function** should be designed so as to avoid collisions as much as possible

Hash Codes

- Hash code maps a key **k** to an integer
  - Not necessarily in the desired range **[0, …, N-1]**
  - May be even negative
- We will assume a hash code is a 32-bit integer
- Hash code should be designed to **avoid collisions** as much as possible
  - If hash code causes collision, then compression function will not resolve this collision

Hash Code Example 1: Memory Address

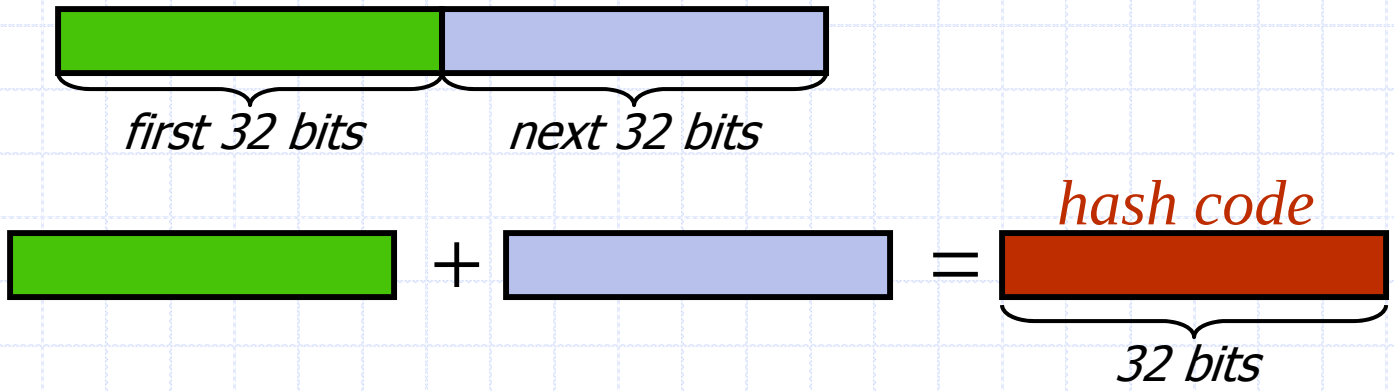
- We can reinterpret the **memory address** of the key object as an integer
- This is what is usually done in Java implementations
  - any object inherits hashCode() method
- May be sufficient, but we usually want "equal " objects to have the same hash code, this will not happen with inherited hashCode(), for example:
  - Want equal strings ("Hello" and "Hello") to have the same hash code, however they will be stored at different memory locations, and will **not** have the same hashCode()
  - Want Integers with the same intValue() map to the same hash code, but since they are stored at different memory locations, they will not have the same hashCode()

Hash Code Example 2: Integer Interpretation

- 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
- For **byte**, **short**, **int**, **char,** cast them into type **int**
- For **float**, use Float.floatToIntBits()
- For 64-bit types **long** or **double**, if we cast them into **int,** half of the information is not used
  - many collisions possible if the big difference between keys is in those lost bits

Hash Code Example 3: Component Sum

- We partition the bits of the key into components of fixed length of 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)

Hash Code for Strings

- Suppose we apply the above idea to **strings**
- Sum up ASCII (or Unicode) values for characters in string **s**
- Component Sum Hash code for string "abracadabra" is ASCII("a")+ ASCII("b")+…+ ASCII("a")
- Lots of **collisions** possible
  - "stop", "tops", "pots", "spot" all get mapped to the same hash code
- Problem:

  position of individual characters is important, but not taken into account by the component sum hash code

Hash Code Example 4: Polynomial Accumulation

- Suppose we have variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{k-1})$ and the order of x_i's is significant
- We choose non-zero constant a (not equal to 1)
- **Polynomial hash code** is:

$$p(a) = x_0 + x_1 a + x_2 a^2 + \ldots + x_{n-1} a^{n-1}$$

  - overflows are ignored
- Intuitively, multiplication by a makes room for each component in a tuple of values, while also preserving a characterization of the previous components

Hash Code Example 4: Polynomial Accumulation

- Especially suitable for strings. Experimentally, good choices $a = 33, 37, 39, 41$
- Choice $a = 33$ gives at most 6 collisions on a set of 50,000 English words
- Polynomial $p(a)$ can be evaluated in $O(n)$ time using **Horner's rule**:

$$p(a) = x_0 + x_1 a + x_2 a^2 + \ldots + x_{n-1} a^{n-1}$$
$$= x_0 + a(x_1 + a(x_2 + \ldots + a(x_{n-2} + x_{n-1} a)\ldots)$$

- Use:

$$p_1 = x_{n-1}$$
$$p_2 = a p_1 + x_{n-2}$$
$$\ldots\ldots\ldots\ldots$$
$$p_i = a p_{i-1} + x_{n-i}$$

- $p(a) = p_n$

Hash Code Example 4: Polynomial Accumulation

- **Horner's rule:**

  $p(a) = x_0 + a(x_1 + a(x_2 + \ldots + a\,(x_{n-2} + x_{n-1}a)\ldots)$

- Let a = 33
  - "top" = "t"+"o"*33+"p"*33²=116+111*33+112*33²=125747
  - "pot" = "p"+"o"*33+"t"*33²=112+111*33+116*33²=130099

- Algorithm:

  $p = x[n-1]$
  $i = n - 2$
  while $i \geq 0$ do
  $p = p*a+x[i]$
  $i = i - 1$

Compression Functions

- Now know how to map objects to integers using a suitable hash code
- The hash code for key k will typically not in the legal range [0, ..., N-1]
- Need **compression function** to map the hash code into the legal range [0, ..., N-1]
- Good compression function will minimize the possible number of collisions

Division Compression Function

- $h_2(y) = |y| \bmod N$
- If N is a **prime** number, then this compression function helps to "spread out" the hashed values
  - Thus size N is usually chosen to be a **prime**
- The reason has to do with number theory and is beyond the scope of this course
- but consider an example:

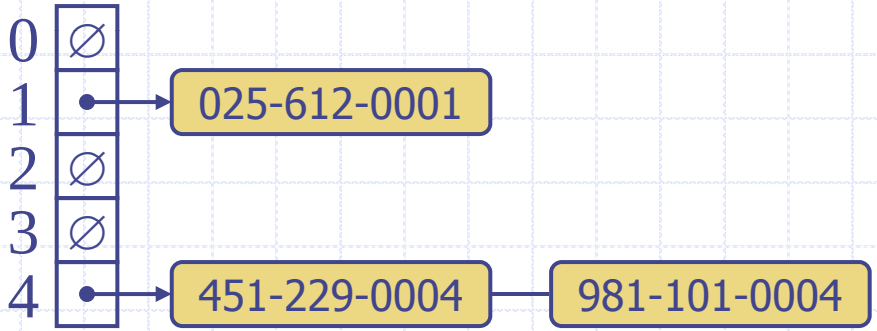
  $\{200, 205, 210, 300, 305, 310, 400, 405, 410\}$
  - N=100, hashed values $\{0, 5, 10, 0, 5, 10, 0, 5, 10\}=\{0, 5, 10\}$
  - N=101, hashed values $\{99, 3, 8, 98, 2, 7, 97, 1, 6\}$
- "**mod N**" compression does not work well if there is repeated pattern of hash codes of form $pN+q$ for different p's

MAD Compression Function (**M**ultiply, **A**dd and **D**ivide)

- $h_2(y) = |ay + b| \bmod N$
- N is a **prime** number
- $a > 0$ and $b \geq 0$ are integers such that
  - $a \bmod N \neq 0$
  - otherwise, every integer would map to the same value, that is to $(b \bmod N)$
  - a and b are usually chosen randomly at the time when a MAD compression function is chosen
- This compression function spreads hash codes fairly evenly in the range $[0, \ldots, N\text{-}1]$
- MAD compression is the best one to use

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



Load Factor

- Suppose that
  - the bucket array is of capacity N
  - there are n entries in the hash table
- The load factor is defined as $\lambda = n/N$
- If hash function is good (that is spreads keys evenly in the array $A[0,\ldots N]$), then n/N is the expected number of items in each bucket
- Find, insert, remove, take $O(n/N)$ expected time
- Ideally, each bucket should have at most 1 item
- Thus should keep the load factor $\lambda < 1$
  - for separate chaining, recommended $\lambda < 0.9$

Dictionary Methods with Separate Chaining for Resolving Collisions

Implement each bucket as list-based dictionary:

Algorithm insert(k,v):
Input: A key k and value v
Output: Entry (k,v) is added to dictionary D

if $(n+1)/N > \lambda$ **then** // Load factor became too large
double the size of A (again a prime) and rehash all existing entries

e = A[h(k)].insert(k,v) // A[h(k)] is a linked list
$n = n+1$ // n is number of entries in hash table
return e

Algorithm findAll(k):
Input: A key k
Output: An iterator of entries with key equal to k

return A[h(k)].findAll(k)

Dictionary Methods with Separate Chaining for Resolving Collisions

Algorithm remove(*e*):
Input: An entry e
Output: The (removed) entry *e* or **null** if *e* was not in dictionary *D*

```
t = A[h(k)].remove(e)      // delegate the remove to
                           // dictionary at A[h(k)]}
if t ≠ null then           // e was found
   n = n - 1               // update number of entries in
                           // hash table
return t
```

Open Addressing for Handling Collisions

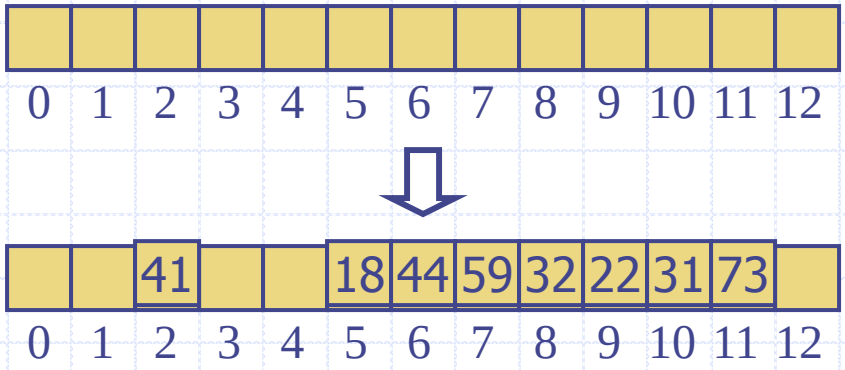
- Separate chaining
  - **Advantage**: simple implementation
  - **Disadvantage**: additional storage requirements, that is auxiliary data structure (list) to hold entries with colliding keys; may not have this additional space available (for example, if coding for a small handheld device)
- Can handle collisions without additional data structure, that is always store entries in the array ***A*** itself, this is called **open addressing**
  - ***A*** does not need to be a bucket array in this case
  - Load factor has to be always at most 1, since all entries are stored in the array ***A*** itself

Open Addressing: Linear Probing

- **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**"
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys in order of

x = (18, 41, 22, 44, 59, 32, 31, 73)

h(x)= (5, 2, 9, 5, 7, 6, 5, 8)



- Colliding items lump together, thus future collisions to cause a longer sequence of probes

Open Addressing: Search with Linear Probing

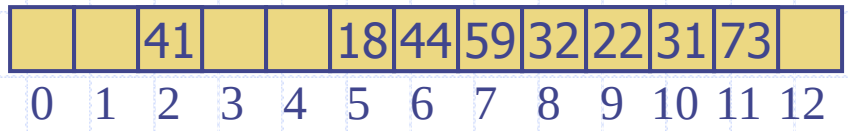
- Consider a hash table A that uses linear probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Algorithm *find*(k)

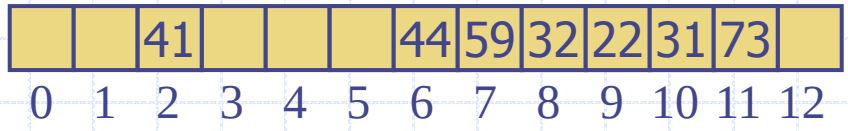
$i \leftarrow h(k)$
$p \leftarrow 0$
repeat
  $c \leftarrow A[i]$
  if c == *null*
    return *null*
  else if c.*key* () == k
    return c.*element*()
  else
    $i \leftarrow (i + 1) \bmod N$
    $p \leftarrow p + 1$
until $p == N$
return *null*

Linear Probing: what about remove(e)?

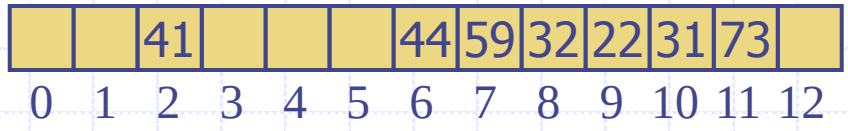
- $h(x) = x \bmod 13$



remove(18) ⇨



find(31) ⇨

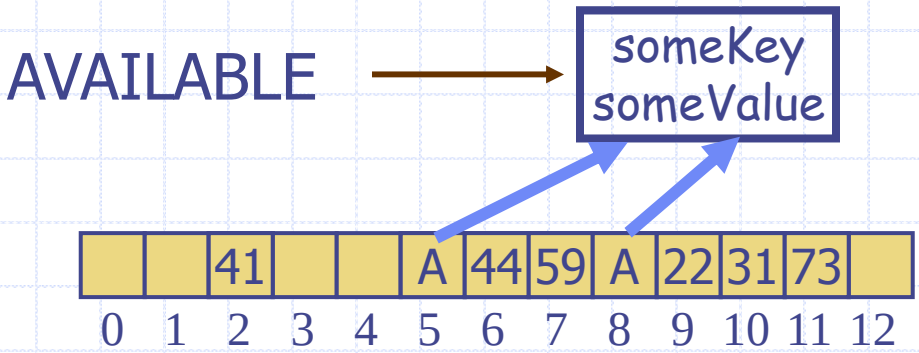


- Oops, 31 is not found now!

Linear Probing: solution for remove(e)

- To handle deletions, replace deleted entry with a special "marker" to signal that an entry was deleted from that index
  - **null** is a special marker, but we already use it!
  - Create special Entry object called ***AVAILABLE***

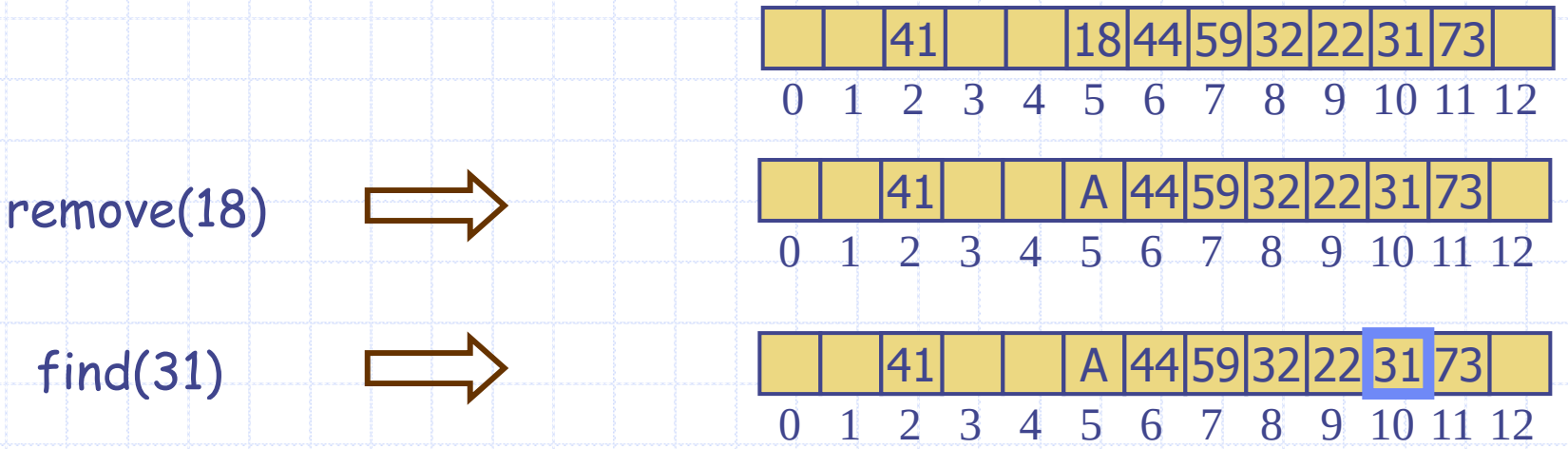
Entry AVAILABLE=new Entry(someKey,someValue);



  - don't care what the someKey and someValue are, we never use them
  - we use only the reference (address) of the newly created object AVAILABLE
    - if A[index] == AVAILABLE

Linear Probing: solution for remove(e)

- remove(*e*)
  - We search for an entry *e*
  - If such an entry *e* is found, we replace it with the special item *AVAILABLE* and we return the entry *e*
  - Else, we return *null*
- Example: $h(x) = x \bmod 13$



Open Addressing: Fixed Search with Linear Probing

- Need to fix find(k)
  - skip over AVAILABLE entries
  - Check if A[i] references the same object as the special marker AVAILABLE
  - Do not access A[i].key or A[i].value before you make sure A[i]≠AVAILABLE
    - AVAILABLE.key and AVAILABLE.value do not make any sense

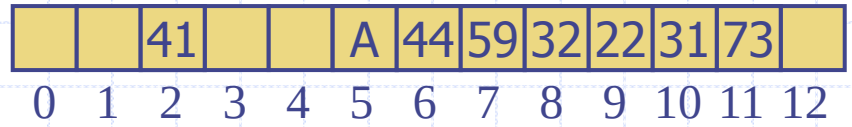
```
Algorithm find(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c == null
            return null
        else if c != AVAILABLE
            if c.key () == k
                return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until  p == N
    return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

Open Addressing: insert() with Linear Probing

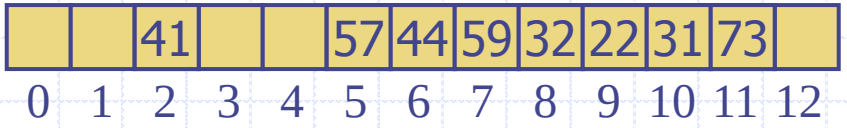
- insert(***k***,***v***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until cell ***i*** is found that is
    - either empty (= **null**)
    - or ***AVAILABLE***
  - We store entry (***k***, ***v***) in cell ***i***

- Example:

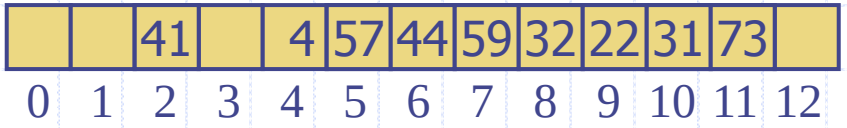
$h(x) = x \bmod 13$



insert(57)



insert(4)



Open Addressing: Updates with Linear Probing

- **Disadvantages** of Linear Probing
  - code is more complicated
  - entries tend to cluster into contiguous runs, which slows down the **find** and **insert** operations significantly

Where are we now?

- What is hashing? Why hashing?
- How to do hashing?
  - hash function + (bucket) array (size is a prime number)
  - **hash function**: hash code + compression function
  - **hash codes**: memory address, integer interpretation, component sum, polynomial accumulation
  - **compression function**: division, MAD
  - What hash functions are good?
- The big issue with hash tables: **collision**
- Collision handling:
  - Separate chaining, bucket array, load factor $\lambda = n/N < 0.9$
  - Opening addressing, array, load factor $\lambda = n/N < 0.5$
    - Linear probing, problem of clustering
    - **Double hashing, more costly by using two hashes**

Open Addressing: Double Hashing

- Suppose $i = h(k)$, and $A[i]$ is already occupied
- **Double hashing** uses a secondary hash function $h'(k)$ and handles collisions by placing an item in the first available cell of the series. N is the size of the table.

$$[\, i + j\, h'(k)\,] \bmod N, \quad \text{for } j = 0, 1, \ldots, N - 1$$

- The secondary hash function $h'(k)$ cannot have zero values (the number of cells in each leap)
- N must be a **prime** to allow probing of all the cells
- **Linear Probing** is a special case of double hashing with $h'(k) = 1$ for any k (only one cell in each leap)
  - $[i + j\, h'(k)] \bmod N = [i + j] \bmod N$ for $j = 0, 1, \ldots, N - 1$

Open Addressing: Double Hashing

- Common choice of compression function for the 2nd hash function:

  $h'(k) = q - k \bmod q$, where

  - $q < N$
  - q is a **prime**
  - Here we assumed k is an integer. If not, use
    - $h'(k) = q - |\text{HashCode}(k)| \bmod q$

- The possible values for $h'(k)$ are

  $1, 2, \ldots, q$

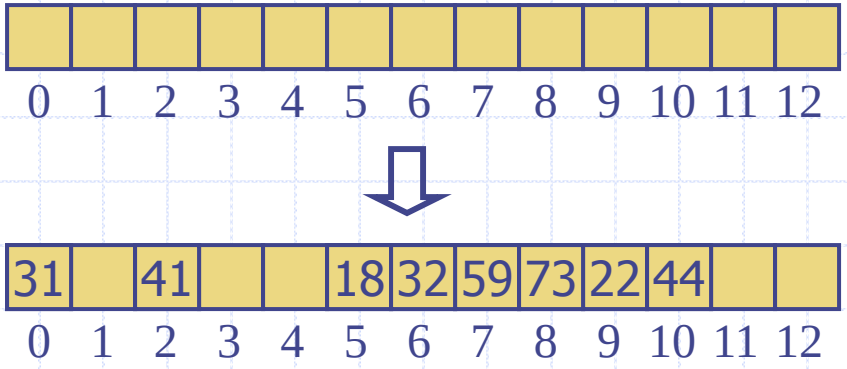
Open Addressing: Example of Double Hashing

- Double hashing strategy:

  $[\, h(k) + jh'(k) \,] \bmod N$, for $j = 0, 1, \ldots, N - 1$

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $h'(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$h'(k)$	Probes (0,1,2,.
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



Performance of Hashing with Open Addressing

- 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 dictionary collide
- The **load factor** $\lambda = n / N$ 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

  $1 / (1 - \lambda)$

  - It is recommended to keep $\lambda < 0.5$

Separate Chaining vs. Open Addressing

- Open addressing **saves space** over separate chaining
- Separate chaining is usually **faster** (depending on **load factor** of the bucket array) than open addressing, both theoretically and experimentally
- Thus, if memory space is not a major issue, use separate chaining, otherwise use open addressing

Performance of Hashing

- 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 1**
- **Rehash** with bigger array if load factor becomes bad
  - Rule of thumb: new array **twice** the size of old array (still a prime)
- Applications of hash tables:
  - small databases
  - compilers
  - web browser caches