Hashing

- Hash functions
  - Hash-code maps
  - Compression maps
- Open addressing
  - Linear Probing
  - Double hashing

Hash Functions

- Need to choose a good hash function
  - quick to compute
  - distributes keys uniformly throughout the table
  - good hash functions are very rare – *birthday* paradox
- How to deal with hashing non-integer keys:
  - find some way of turning keys into integers
    - *eg*. remove hyphen in 9635-8904 to get 96358904!
    - for a string, add up ASCII values of the characters of your string (e.g., java.lang.String.hashCode())
  - then use standard hash function on the integers

From Keys to Indices

- The mapping of keys to indices of a hash table is called a ***hash function***
- A hash function is usually the composition of two maps, a ***hash code map*** and a ***compression map.***
  - An essential requirement of the hash function is to map equal keys to equal indices
  - A "good" hash function minimizes the probability of collisions

- ***hash code map***: key → integer
- ***compression map***: integer → $[0, N-1]$

Popular Hash-Code Maps

- Integer cast: for numeric types with 32 bits or less, we can reinterpret the bits of the number as an int
- Component sum: for numeric types with more than 32 bits (e.g., long and double), we can add the 32-bit components.
- Why is the component-sum hash code bad for strings?

Hash-Code Maps (2)

- Polynomial accumulation: for strings of a natural language, combine the character values (ASCII or Unicode) $a_0a_1 \ldots a_{n-1}$ by viewing them as the coefficients of a polynomial:

  $$a_0 + a_1x + \ldots + x^{n-1}a_{n-1}$$

- The polynomial is computed with Horner's rule, ignoring overflows, at a fixed value x:

  $$a_0 + x\,(a_1 + x\,(a_2 + \ldots x\,(a_{n-2} + x\,a_{n-1}) \ldots))$$

- The choice x = 33, 37, 39, or 41 gives at most 6 collisions on a vocabulary of 50,000 English words

Compression Maps

- Use the remainder
  - $h(k) = k \bmod m$, k is the key, m the size of the table
- Need to choose m
- $m = b^e$ **(bad)**
  - if m is a power of 2, $h(k)$ gives the ***e*** least significant bits of k
  - all keys with the same ending go to the same place
- m prime **(good)**
  - helps ensure uniform distribution
  - primes not too close to exact powers of 2

Compression Maps (2)

- Example
  - hash table for n = 2000 character strings
  - we don’t mind examining 3 elements
  - m = 701
    - a prime near 2000/3
    - but not near any power of 2

Compression Maps (3)

- Use
  - $h(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$
  - k is the key, m the size of the table, and A is a constant
    $0 < A < 1$
- The steps involved
  - map $0...k_{max}$ into $0...k_{max}\ A$
  - take the fractional part (mod 1)
  - map it into $0...m$-1

Compression Maps (4)

Choice of *m* and *A*

- value of *m* is not critical, typically use $m = 2^p$
- optimal choice of *A* depends on the characteristics of the data
- Knuth says use $A = \frac{\sqrt{5}-1}{2}$ (conjugate of the *golden ratio*) – *Fibonacci* hashing

Compression Maps (5)

Multiply, Add, and Divide (MAD):

$h(k) = |ak + b| \bmod N$

- eliminates patterns provided a is not a multiple of N
- same formula used in linear congruential (pseudo) random number generators

Universal Hashing

- For any choice of hash function, there exists a bad set of identifiers
- A malicious adversary could choose keys to be hashed such that all go into the same slot (bucket)
- Average retrieval time is $\Theta(n)$
- Solution
  - a random hash function
  - choose hash function independently of keys!
  - create a **set of hash functions *H***, from which *h* can be randomly selected

Universal Hashing (2)

- A collection H of hash functions is ***universal*** if for any randomly chosen f from H (and two keys k and l),

  $\Pr\{f(k) = f(l)\} \leq 1/m$

More on Collisions

- A key is mapped to an already occupied table location
  - what to do?!?
- Use a collision handling technique
- We've seen *Chaining*
- Can also use *Open Addressing*
  - Probing
  - Double Hashing

Open Addressing

- All elements are stored in the hash table (can fill up!), i.e., $n \leq m$
- Each table entry contains either an element or null
- When searching for an element, systematically probe table slots

Open Addressing (2)

- Modify hash function to take the probe number *i* as the second parameter

$$h : U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$$

- Hash function, *h*, determines the sequence of slots examined for a given key
- Probe sequence for a given key *k* given by

$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ - a permutation of $\langle 0,1,\ldots,m-1 \rangle$

Linear Probing

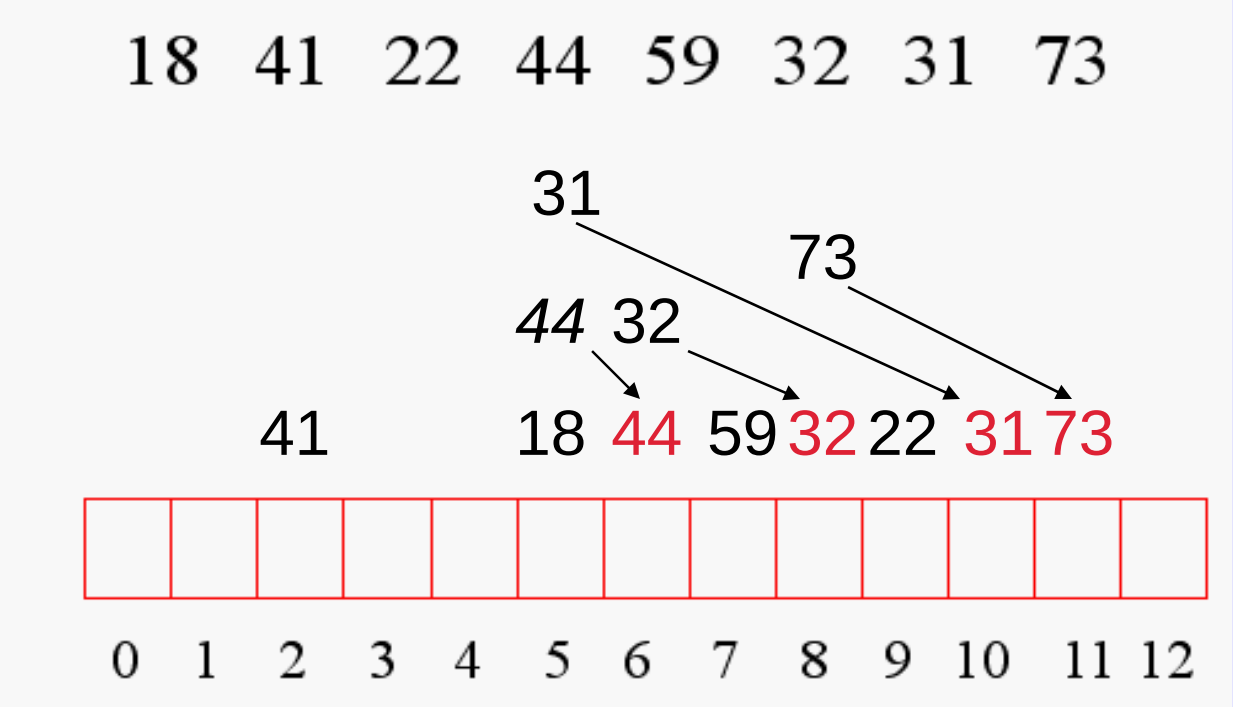
- If the current location is used, try the next table location

```
LinearProbingInsert(k)
 if (table is full) error
 probe = h(k)
 while (table[probe] occupied)
          probe = (probe+1) mod m
 table[probe] = k
```

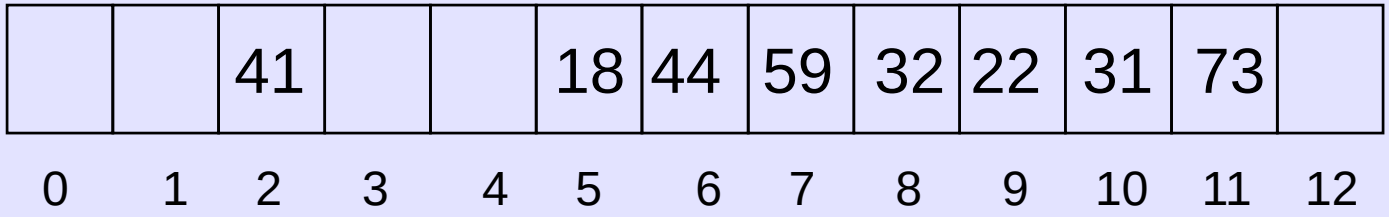
- Uses less memory than chaining as one does not have to store all those links
- Slower than chaining since one might have to walk along the table for a long time

Linear Probing Example

- $h(k) = k \bmod 13$
- insert keys: 18 41 22 44 59 32 31 73

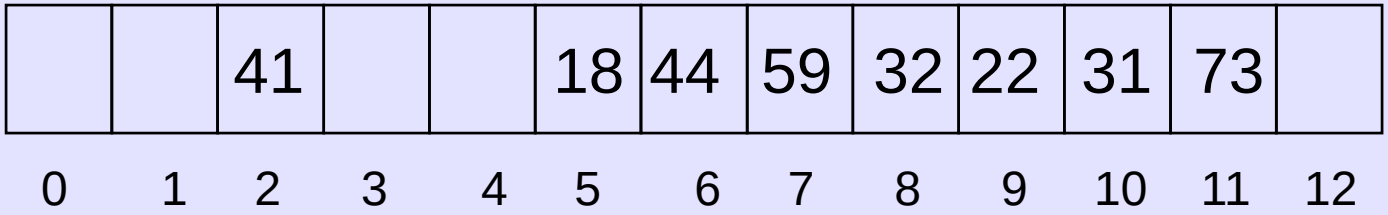


Lookup in linear probing



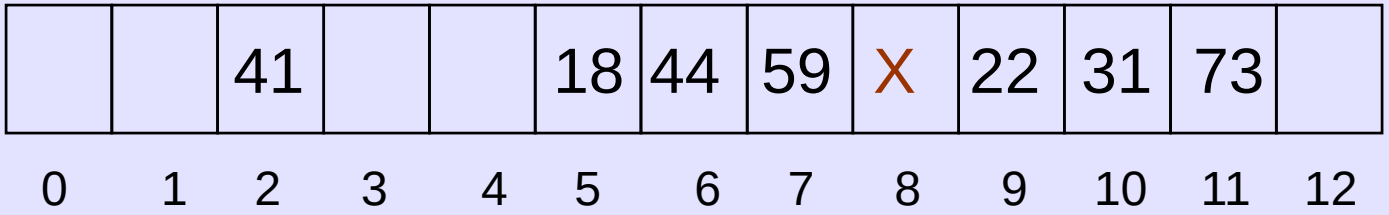
- To search for a key k we go to (k mod 13) and continue looking at successive locations till we find k or encounter an empty location.
- Successful search: To search for 31 we go to (31 mod 13) = 5 and continue onto 6,7,8… till we find 31 at location 10
- Unsuccessful search: To search for 33 we go to (33 mod 5 = 7) and continue till we encounter an empty location (12)

Deletion in Linear Probing



- To delete key 32 we first search for 32.
- 32 is found in location 8. Suppose we set this location to null.
- Now if we search for 31 we will encounter a null location before seeing 31.
- Lookup procedure would declare that 31 is not present.

Deletion (2)



- Instead of setting location 8 to null place a tombstone (a marker) there.
- When lookup encounters a tombstone it ignores it and continues with next location.
- If Insert comes across a tombstone it puts the element at that location and removes the tombstone.
- Too many tombstones degrades lookup performance.
- Rehash if there are too many tombstones.

Double Hashing

Uses two hash functions, h1, h2

- h1(k) is the position in the table where we first check for key k
- h2(k) determines the offset we use when searching for k
- In linear probing h2(k) is always 1.

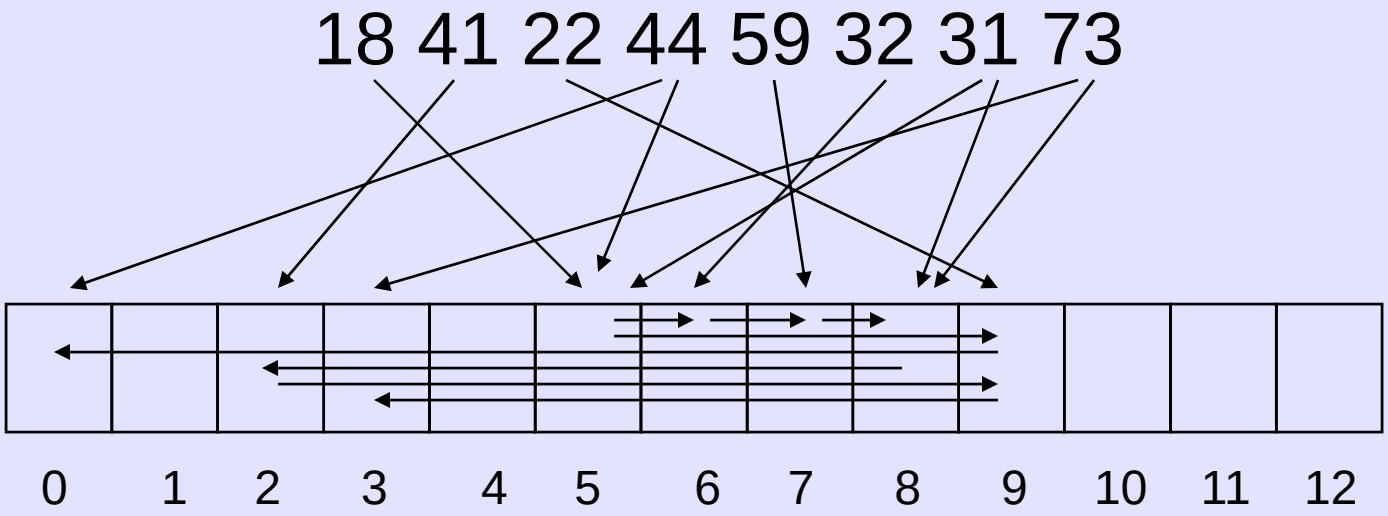
```
DoubleHashingInsert(k)
if (table is full) error
probe = h1(k); offset = h2(k)
while (table[probe] occupied)
    probe = (probe+offset) mod m
table[probe] = k
```

Double Hashing(2)

- If *m* is prime, we will eventually examine every position in the table
- Many of the same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing

Double Hashing Example

- $h1(k) = k \bmod 13$
- $h2(k) = 8 - (k \bmod 8)$
- insert keys: 18 41 22 44 59 32 31 73



Analysis of Double Hashing

- The load factor α is less than 1.
- We assume that every probe looks at a random location in the table.
- 1- α fraction of the table is empty.
- Expected number of probes required to find an empty location (unsuccessful search) is 1/(1- α)

Analysis (2)

- Average no of probes for a successful search = average no of probes required to insert all the elements.
- To insert an element we need to find an empty location.

inserting	Avg no of probes	Total no of probes
First m/2	<= 2	m
Next m/4	<= 4	m
Next m/8	<= 8	m

Analysis(3)

- No of probes required to insert $m/2+m/4+m/8+\ldots+m/2^i$ elements = number of probes required to leave 2^{-i} fraction of the table empty = m x i.
- No of probes required to leave $1-\alpha$ fraction of the table empty = $-\ m \log (1-\alpha)$
- Average no. of probes required to insert n elements is $-\ (m/n) \log (1-\alpha) = -(1/\alpha) \log (1-\alpha)$

Expected Number of Probes

- Load factor $\alpha < 1$ for probing
- Analysis of probing uses *uniform hashing* assumption – any permutation is equally likely
  - What about linear probing and double hashing?

	unsuccessful	successful
chaining	$O(1+\alpha)$	$O(1+\alpha)$
probing	$O\left(\frac{1}{1-\alpha}\right)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$

Expected Number of Probes (2)