CSCE 3110
Data Structures & Algorithms

- Hashing
- Reading: Weiss, chap. 5

Content

- Dictionary ADT
  - Storage of unrelated/unordered data
- Hash table
  - Hash functions
  - Table resizing
  - Collision
    - Chaining
    - Open addressing
- Applications

Dictionary

A dictionary ADT manages a collection of items, each associated with a unique key.

- **Unordered Data**: Unlike arrays or lists, dictionaries don't maintain a specific order.
- **Key-Value Pairing**: Each item in a dictionary has a unique key associated with a value, which makes it efficient for retrieving data based on keys.

Operations:

- insert(item): add item to collection.
- delete(item): remove item from collection.
- search(key): return item with key if it exists

We assume: Each item has a unique key, or if a new item with an existing key is inserted, it replaces the old one.

Implementations So Far

	Unsorted List	Sorted Array	Trees AVL – worst case Splay – amortized
insert	find+θ(1)	θ(n)	θ(log n)
find	θ(n)	θ(log n)	θ(log n)
delete	find+θ(1)	θ(n)	θ(log n)

Dictionary

Abstract Data Type (ADT) — maintain a set of items, each with a key, subject to

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists

Each item has a distinct key. If a new item with an existing key is inserted, it replaces the old one.

Balanced BSTs solve in O(lg n) time per op.

Goal: O(1) time per operation.

Item	Price ($)
Apple	0.50
Banana	0.30
Milk	1.20
Bread	2.00
Eggs	3.00
Cheese	4.50
Chicken	5.00
Orange	0.60

Motivation

Dictionaries are perhaps the most popular data structure in CS

- built into most modern programming languages (Python, Perl, Ruby, JavaScript, Java, C++, C#. . .)
- implement databases
  - **Keyword Mapping**: For example, mapping a word to all webpages that contain it.
  - **User Accounts**: Associating usernames with their corresponding account objects, making data retrieval fast and easy.
- compilers & interpreters: names → variables

  - In programming language compilers and interpreters, dictionaries are used to map variable names to actual values or memory locations, enabling quick access during code execution.
- network routers: IP address → wire

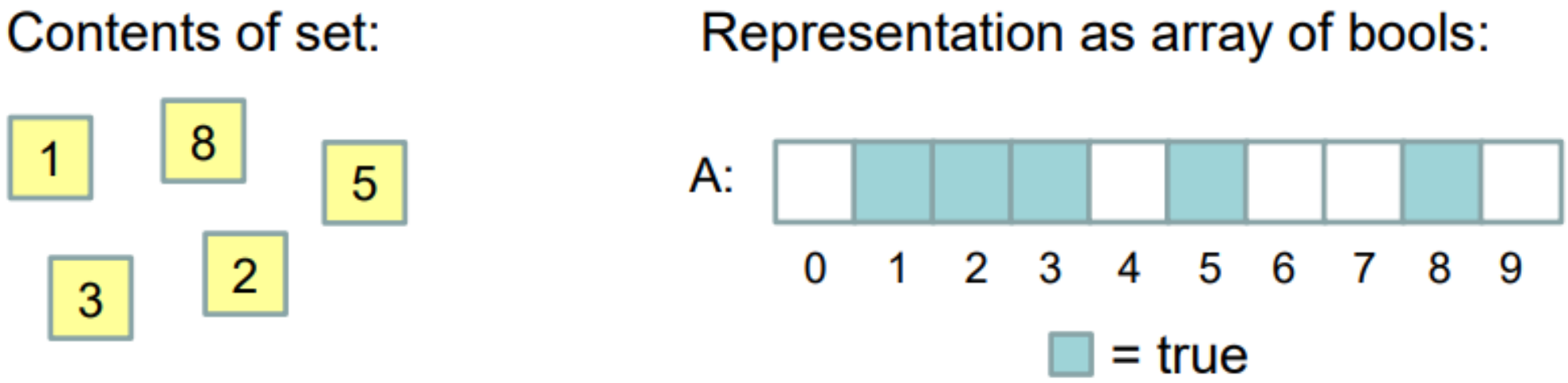
  - dictionaries can map IP addresses to specific physical wires or ports, facilitating efficient data transmission.
- network server: port number → socket/app.

  - Servers use dictionaries to map port numbers to applications or sockets, directing incoming data to the correct application.
- virtual memory: virtual address → physical

  - In operating systems, dictionaries can be used to map virtual memory addresses to physical memory addresses, helping manage memory effectively.

First Attempt: A Direct Access Table

- Maintain a large array of bools
- Presence of key k in structure means $A[k]$ is true.
- Insert, remove, and find all run in O(1) time!
- But what are the drawbacks of this approach?



First Attempt: A Direct Access Table

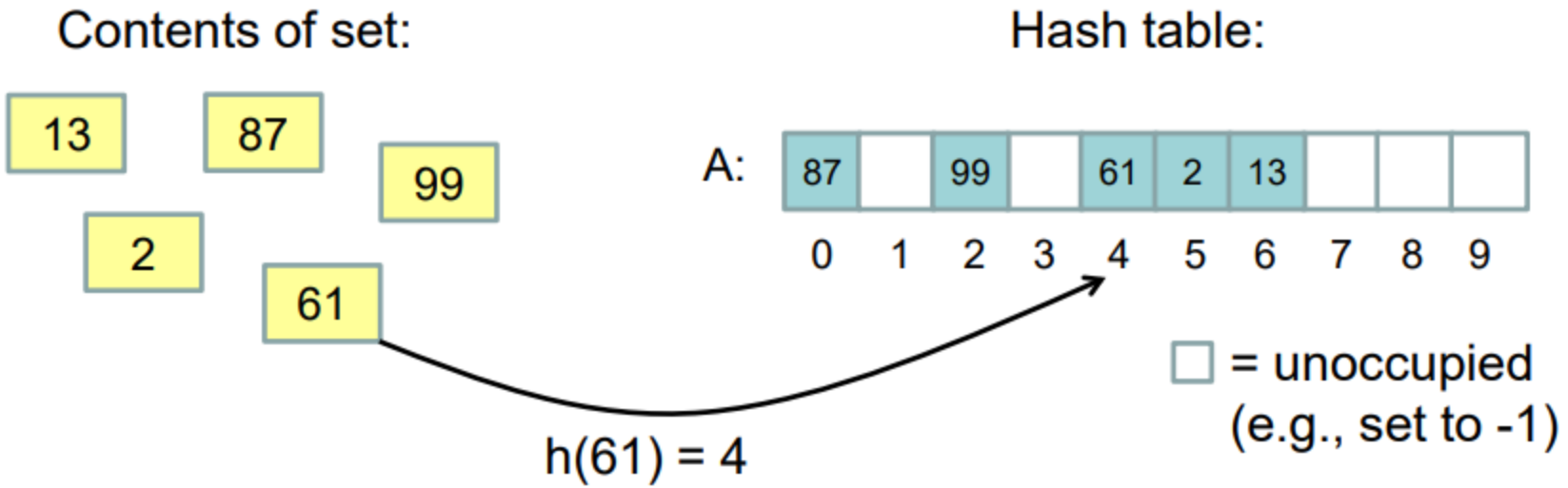
- Maintain a large array of bools
- Presence of key k in structure means A[k] is true.
- Insert, remove, and find all run in O(1) time!
- But what are the drawbacks of this approach?

- Space! (and therefore, also initialization time)
  - pneumonoultramicroscopicsilicovolcanoconiosis
- How would we represent a set of doubles or strings?

This approach only works well with integer keys within a manageable range. It doesn't work effectively for more complex data types, such as **doubles (floating-point numbers)** or **strings**, which can't be easily mapped to a fixed position in a boolean array without additional transformations.

Hash Tables

- Store elements in an array.
- Key k stored in position $h(k)$
- $h(\quad)$ is known as a "hash function" – it maps keys down to the range of indices in our array. Key --> Index
- Example: $h(k) = (2971k + 101923)\ \%\ 10$.

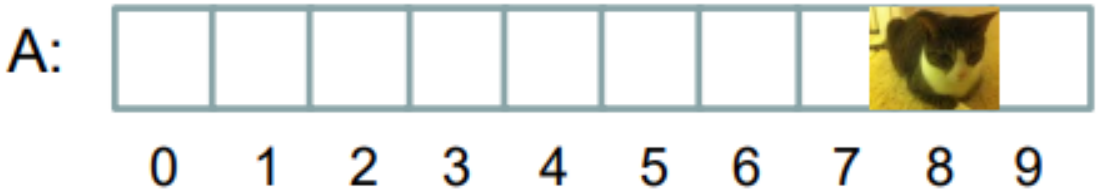


Hash Tables: Positive Features

- Can store doubles, strings, structs, or even more complex or exotic types of data, as long as we can hash it.

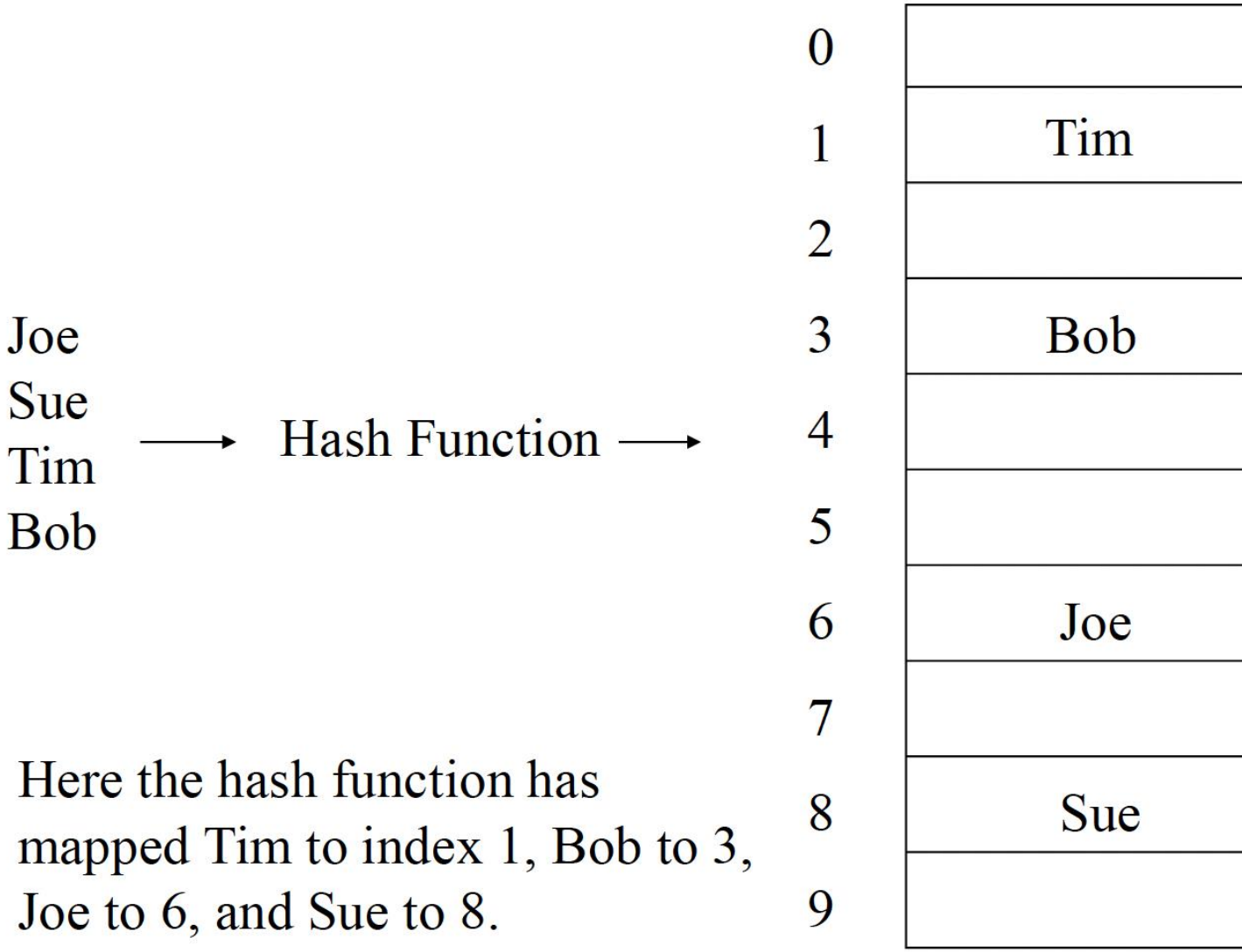
E.g., h() = 8

- Space required for hash table is quite small; ideally, we use only $O(N)$ space for storing N elements.



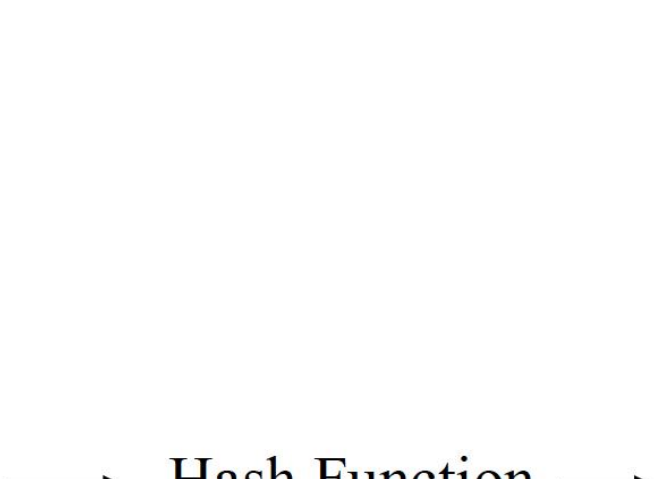
Hash Functions

- Division Method
- Multiplication Method
- Universal Hashing



Hash Functions

Joe
Sue
Tim
Bob



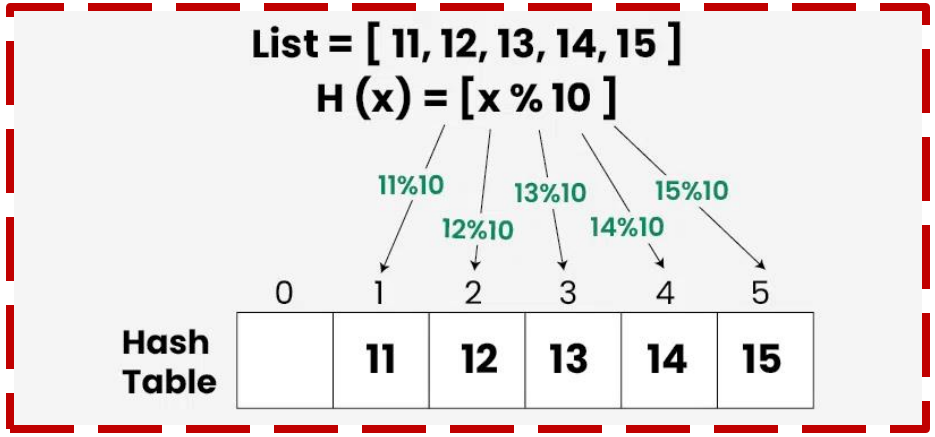
0	
1	Tim
2	
3	Bob
4	
5	
6	Joe
7	
8	Sue
9	

Here the hash function has mapped Tim to index 1, Bob to 3, Joe to 6, and Sue to 8.

Collisions

It is possible for a hash function to hash two keys to the same location.

Division Method



- $h(k) = k\%m$,
- remainder $h(k)$ serves as the index in the hash table.

- However, this function only provides an even or "random" distribution if the input values (keys) themselves are random.
- If the keys have a pattern or are not random, the hash values may be biased, leading to uneven distribution and clustering (i.e., many values going to the same positions).

- Examples:
  - Prices: If we hash prices using modulo 10 (i.e., m=10), we may find that certain hash values, like 9, occur more frequently if prices end in 9 often (e.g., 9.99, 19.99, etc.). This clustering can make the hash table less efficient.
  - With e-mail addresses — If we use a hash function that considers only the last few characters of an email (for example, in ASCII representation), we may frequently get hash values for popular domain endings like .com, leading to clustering in those positions.

Division Method with prime number

$$h(k) = k\%m$$

This is practical when m is prime but not too close to power of 2 or 10, since highly frequent number won't have prime number as its multiple.

Choosing m as a prime number for the Division Method in hashing helps spread values more evenly and reduces collisions:

1. **Avoiding Data Patterns:** because prime numbers do not have divisors other than 1 and themselves, making them less likely to line up with patterns in data that may have common factors, reducing the chance of multiple keys hashing to the same index.
2. **Uniform Distribution:** avoiding clusters that arise with non-prime values. because prime numbers minimize systematic overlap in the remainders produced by the division operation.
3. **Fewer Collisions:** Since prime numbers reduce patterns and encourage a more even spread

Challenges with Division Method

- **Finding an Appropriate Prime Number**: Identifying a prime number that meets the criteria can be inconvenient, especially for very large hash tables.
- **Division Speed**: Division operations are relatively slower than other arithmetic operations, especially with non-standard divisors.
- **Example**:
  - Calculating 7315mod 100 is straightforward (the answer is 15) because it's easy to handle multiples of 10.
  - However, calculating 7315mod 87 requires more steps and is less intuitive, which can slow down the hashing process.

- Suppose we have two hash function $h(k) = k\%7$ and $h'(k) = 5 - (k\%5)$. Given the ordered input $\{35,40,21,33,26\}$ and an empty hash table with size 7.
- Calculate the hashed key for every input:

Keys	$h(k)$	$h'(k)$
35	0	
40	5	
21	0	
33	5	
26	5	

Collisions

It is possible for a hash function to hash two keys to the same location.

- Suppose we have two hash function $h(k) = k\%7$ and $h'(k) = 5 - (k\%5)$. Given the ordered input $\{35,40,21,33,26\}$ and an empty hash table with size 7.
- Calculate the hashed key for every input:

Keys	$h(k)$	$h'(k)$
35	0	5
40	5	5
21	0	4
33	5	2
26	5	4

Avalanche effect

- **Small Input Change, Major Output Change**:

  A slight change in input should cause a drastically different hash output. This makes related inputs look unrelated, improving randomness.

- **Goal**:

  hashes of similar inputs appear completely different, reducing the chance of patterns in hash values.

- **Design Challenge**:

  Achieving a true avalanche effect is hard without sacrificing speed and efficiency, as it requires complex algorithms to ensure drastic changes in output for minor input adjustments.

- **Practicality**:

  A moderate avalanche effect is often enough, where hash values are well-distributed even for similar inputs, achieving a good balance of efficiency and distribution.

Multiplication Method - Bit-Shift

- Multiplication by a number (preferably prime and large enough to produce overflow) to create a hash. This helps spread the values across the hash table.

- $h(k) = \left((a \times k)\%2^w\right) \gg (w - r)$
  - where a is random,
  - a and k are w −bits values
  - hash table size $m = 2^r$

  This formula shifts the bits right after multiplication to get the final hash value.

- This is practical when a is odd & $2^{w-1} < a < 2^w$ & not too close to 2^{w-1} or 2^w, to ensure a good distribution of hash values.

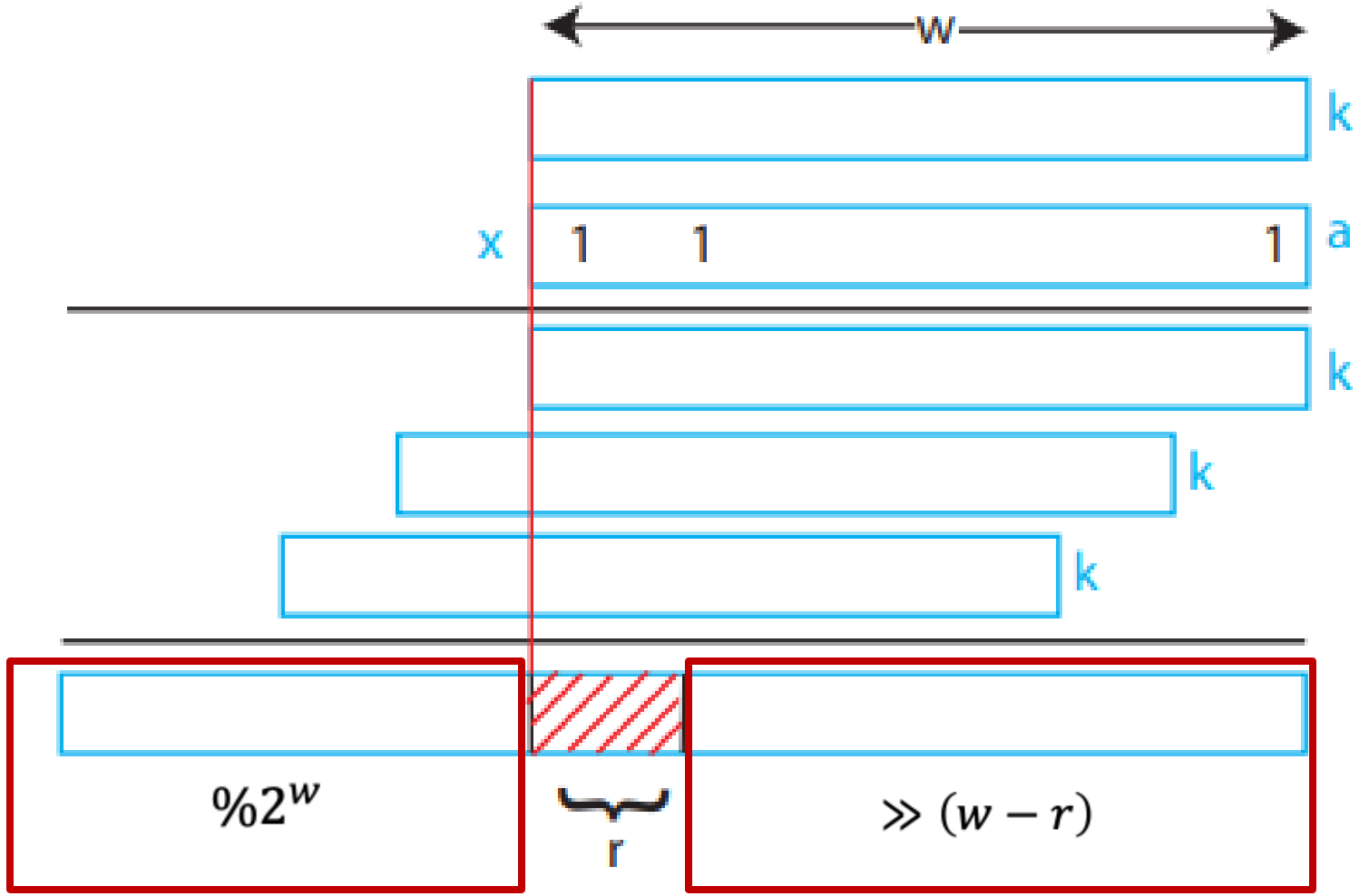
- Multiplication and bit extraction are faster than division

Multiplication Method - Bit-Shift

$$h(k) = \left((a \times k)\%2^w\right) \gg (w - r)$$

a and k are w −bits values
hash table size $m = 2^r$

This formula shifts the bits right after multiplication to get the final hash value.



Multiplication Method - Fractional Part Extraction

$$\text{hash}(key) = \text{floor}(m \times (A \times key \mod 1))$$

Multiply the key by a constant A (a value between 0 and 1).Take the fractional part of A × key (discard the integer part).Multiply by the table size m and take the floor value to get the final hash.

Advantages:

- Flexible: Works for any hash table size, not just powers of 2.
- Simple: Easier to implement without bit manipulation.

Best Use Case:

- Suitable when the hash table size is arbitrary, or when simplicity is prioritized over raw speed.

Disadvantage:

- Slightly Slower: Uses floating-point operations, which can be marginally slower than bit operations but is generally efficient on modern hardware.

Given:

- **Table size** $m = 10$
- **Constant** $A = 0.6180339887$
- **Key** $k = 37$

Hash Value: The hash for key 37 is **8**.

Universal Hashing

A hashing technique that uses randomly chosen hash functions from a family of hash functions (Many methods) to ensure low collision probability for any two different keys.

One example:

- $h(k) = \big((a \times k + b)\%p\big)\%m$
- where a and b are random $\in \{0,1,\cdots,p-1\}$, p is a large prime ($> |U|$).

- A family of hash functions is universal if for any x≠ y, the number of hash functions h in H such that h(x) = h(y) is at most |H|/ M.
- This implies if a hash function is chosen from the family randomly, the probability of a collision is 1/M.

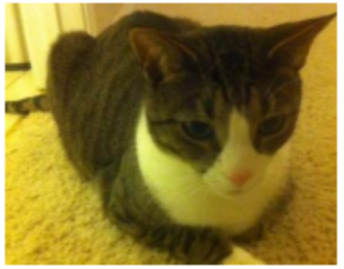
  - for worst case keys $k_1 \neq k_2$
  - $Pr_{a,b}\big(h(k_1) = h(k_2)\big) = \frac{1}{m}$, so, $E_{a,b}(\#collisions\ with\ k_1) = \frac{n}{m} = \alpha$

Hashing / Storing Large Objects

- So far, we have stored integers using a hash function like $h(k) = (ak + b)\%M$. (M = table size).
- However, we can store any type of object as long as we can invent a good hash function… include arrays, images, strings, files, and complex structures.
- (and we often store just a location pointer to the object in the table, instead of the object itself)

h(A[0], A[1], A[2], A[3], …,) = ?

h(main.cpp) = ?



h( ) = ?

h(CSCE 3110 is awesome) = ?

h(
```
struct {
   string name;
   int age;
   int id_number;
};
```
) = ?

Hashing Arrays

- Any complicated object can be “serialized” and represented by an array of small integers.
- E.g., “CSCE” → ‘C’, ‘S’, ‘C’, ‘E’ → 67, 83, 67, 69.

- So how do we hash an array $A[0 \ldots N-1]$ to get an output value in the range $0 \ldots M-1$?

Hashing Arrays

- Any complicated object can be “serialized” and represented by an array of small integers.
- E.g., “CSCE” → ‘C’, ‘S’, ‘C’, ‘E’ → 67, 83, 67, 69.

- So how do we hash an array $A[0 \ldots N-1]$ to get an output value in the range $0 \ldots M-1$?

- How about something like this:

h(A[]) = (A[0] + A[1] + … + A[N – 1]) mod M

Polynomial Hash Functions

- Think of A[] as the coefficients of a polynomial:

$p(x) = A[0] + A[1]\,x + A[2]\,x^2 + A[3]\,x^3 + \ldots + A[N-1]\,x^{(N-1)}$

- To hash A[], evaluate p(x) mod M at a randomly chosen point x.

- How do we do this quickly?

Polynomial Hash Functions

- Think of A[] as the coefficients of a polynomial:

p(x) = A[0] + A[1] x + A[2] x^2 + A[3] x^3 + … + A[N-1] x^(N-1)

- To hash A[], evaluate p(x) mod M at a randomly chosen point x.
- How do we do this quickly?

```
x = 17; /* pick your favorite number… */
hash = 0;
for (i=N-1; i>=0; i--)
      hash = (hash * x + A[i]) % M;
```

Table Resizing

- Want $m = \Theta(n)$ at all times
- Don't know how large n will get at creation
- m too small ⇒ slow; m too big ⇒ wasteful

Idea:

- Start small (constant) and grow (or shrink) as necessary.

Rehashing

⇒ must rebuild hash table from scratch

- for item in old table: → for each slot, for item in slot
- insert into new table

- To grow or shrink table, (m → m’)
- Hash values must change

⇒ $\Theta(n + m + m')$ time $= \Theta(n)$ if $m = \Theta(n)$ & $\mathrm{m}' = \Theta(n)$

How fast to grow?

When n reaches m,

$m \mathrel{+}= 1$?

$m \mathrel{*}= 2$?

How fast to grow?

When n reaches m,

$m \mathrel{+}= 1$?

- $\Rightarrow$ rebuild every step
- $\Rightarrow n$ inserts cost $\Theta(1 + 2 + \cdots + n) = \Theta(n^2)$

$m \mathrel{*}= 2$?

- $\Rightarrow$ rebuild at insertion 2^i
- $\Rightarrow n$ inserts cost $\Theta(1 + 2 + 4 + 8 + \cdots + n)$ where n is really the next power of 2
- = $\Theta(n)$

A few inserts cost linear time, but $\Theta(1)$ “on average”

Deletion

$n \to \frac{m}{2}$ then shrink to $\frac{m}{2}$?

$n \to \frac{m}{4}$ then shrink to $\frac{m}{2}$?

Deletion

$n \to \frac{m}{2}$ then shrink to $\frac{m}{2}$?

- SLOW, $2^k \leftrightarrow 2^k + 1$, $\Theta(n)$ per op.

$n \to \frac{m}{4}$ then shrink to $\frac{m}{2}$?

- $O(1)$ amortized cost for both insert and delete (proof. Introduction to algorithm, 17.4, dynamic tables)

Collisions

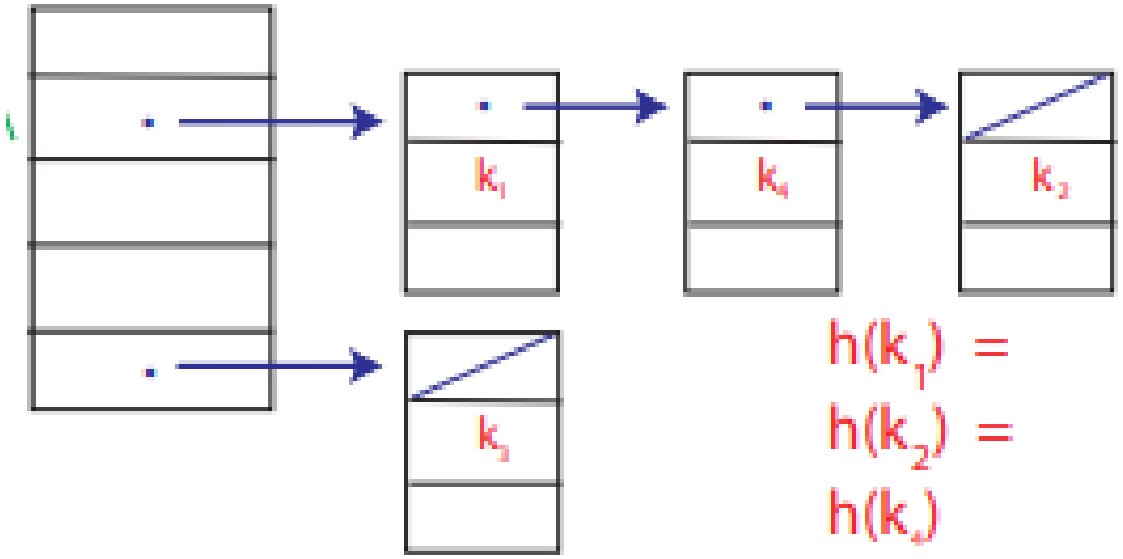
- Two keys $k_i, k_j \in K, k_i \neq k_j$ collide if $h(k_i) = h(k_j)$

How do we deal with collisions?

- Chaining
- Open addressing

Chaining

- Linked list of colliding elements in each slot of table



- Search must go through whole list
- Worst case: all n keys hash to same slot $\Rightarrow \Theta(n)$ per operation

Running time with Chaining

Simple Uniform Hashing:

- Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.

Running time

- Let n = # keys stored in table
- m = # slots in table
- Load factor $\alpha = n/m$ = expected # keys per slot = expected length of a chain
- Expected running time for search is $\Theta(1 + \alpha)$
  - 1 comes from applying the hash function and random access to the slot
  - α comes from searching the list
  - $\Theta(1)$ if $\alpha = \Theta(1)$, i.e., $m = \Theta(n)$.

- Suppose we have two hash function $h(k) = k\%7$ and $h'(k) = 5 - (k\%5)$. Given the ordered input $\{35,40,21,33,26\}$ and an empty hash table with size 7.

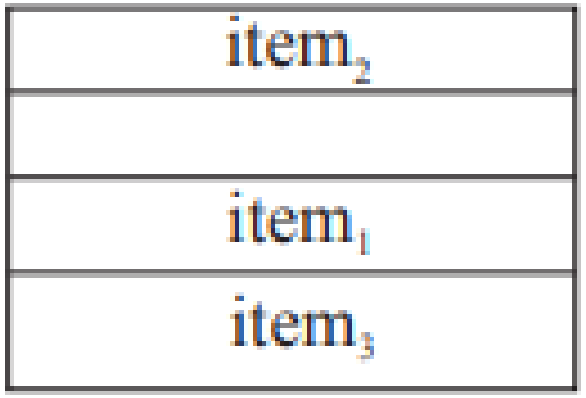
- Show the hash table using chaining and $h(k)$ after insert all the inputs

Keys	$h(k)$	$h'(k)$
35	0	5
40	5	5
21	0	4
33	5	2
26	5	4

Open Addressing

- Another approach to collisions
- No chaining; instead, all items stored in table

- One item per slot $\Rightarrow m \geq n$



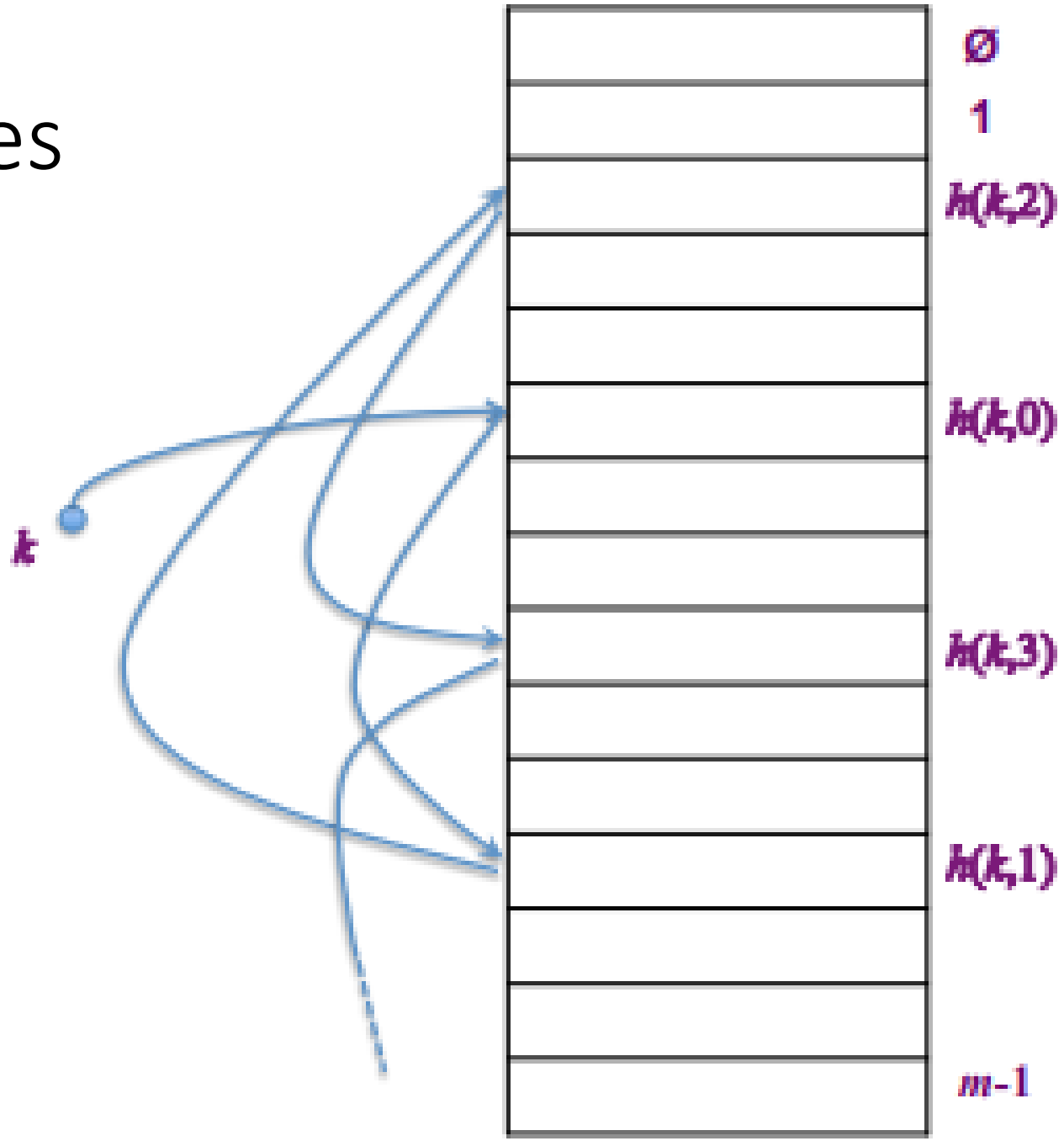
Probing

- Hash function specifies order of slots to probe (try) for a key (for insert/search/delete), not just one slot;
- We want to design a function h, with the property that for all $k \in U$:
- $< h(k,0), h(k,1), \ldots, h(k,m-1) >$ is a permutation of $0,1,\ldots,m-1$.

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

universe of keys trial count slot in table

Order of Probes

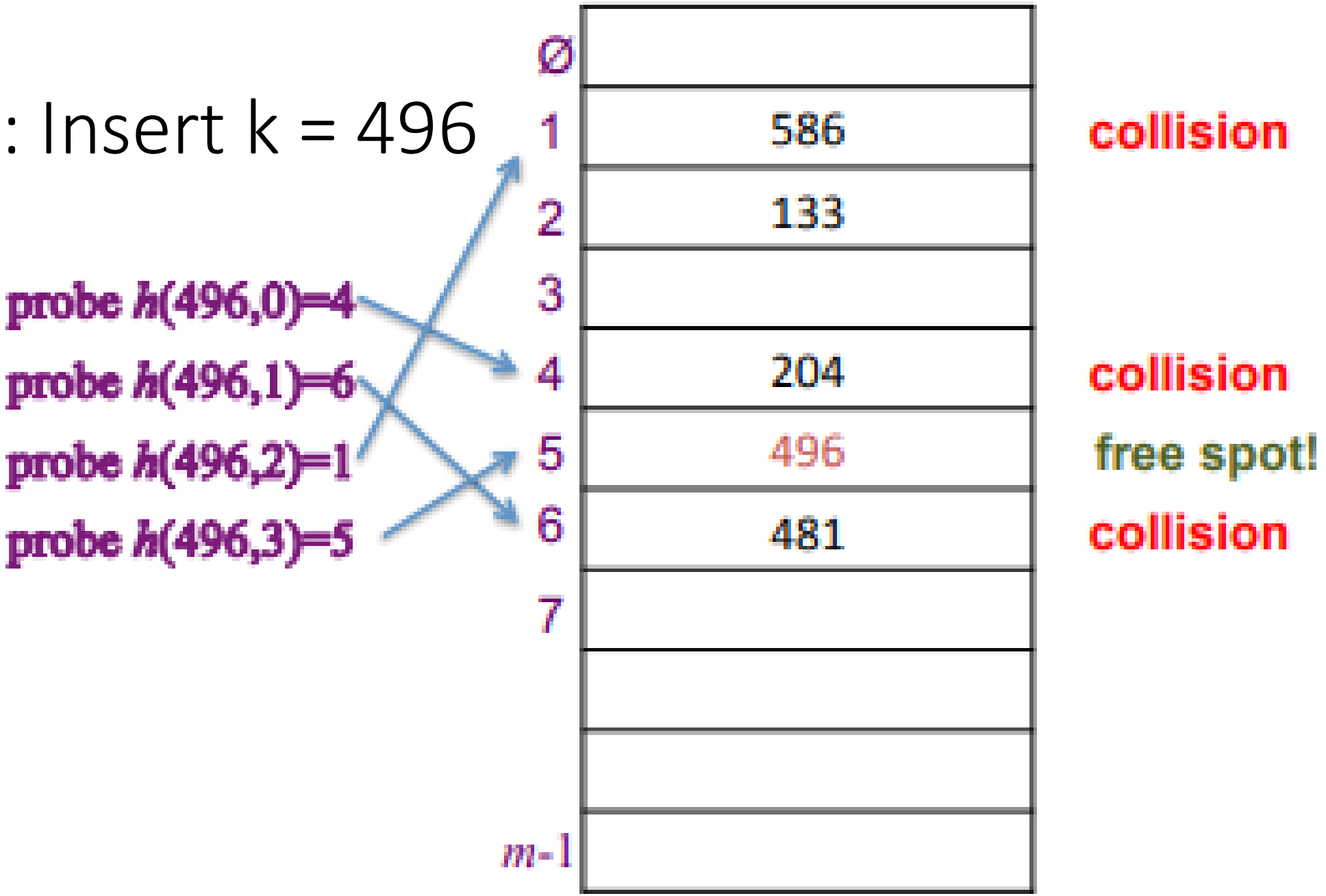


Insert(k, v)

Keep probing until an empty slot is found. Insert item into that slot.

```
for i in range(m):
     if A[h(k, i)] is None: # empty slot
          A[h(k, i)] = (k, v) # store item
          return
raise ‘full’
```

Example: Insert k = 496



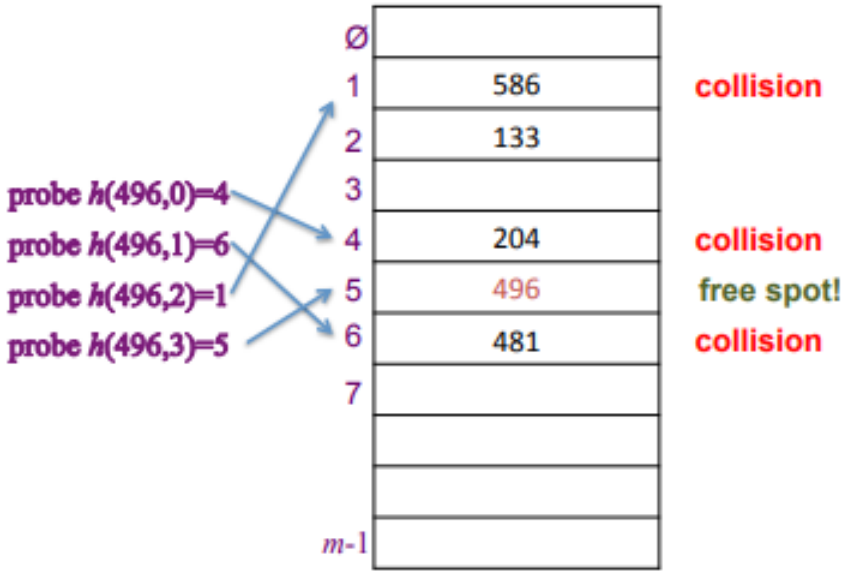
Search(k)

As long as the slots you encounter by probing are occupied by keys ≠ k, keep probing until you either encounter k or find an empty slot—return success or failure respectively.

```
for i in range(m):
     if A[h(k, i)] is None:          # empty slot?
          return None                # end of “chain”
     elif T[h(k, i)][0] == k:        # matching key
          return T[h(k, i)]          # return item
return None                         # exhausted table
```

Delete(k)

- Can’t just find item and remove it from its slot (i.e., set A[h(k, i)] = None)
- Example: delete(586) ⇒ search(496) fails
- Replace item with special flag: “DeleteMe”, which Insert treats as none, but Search doesn’t



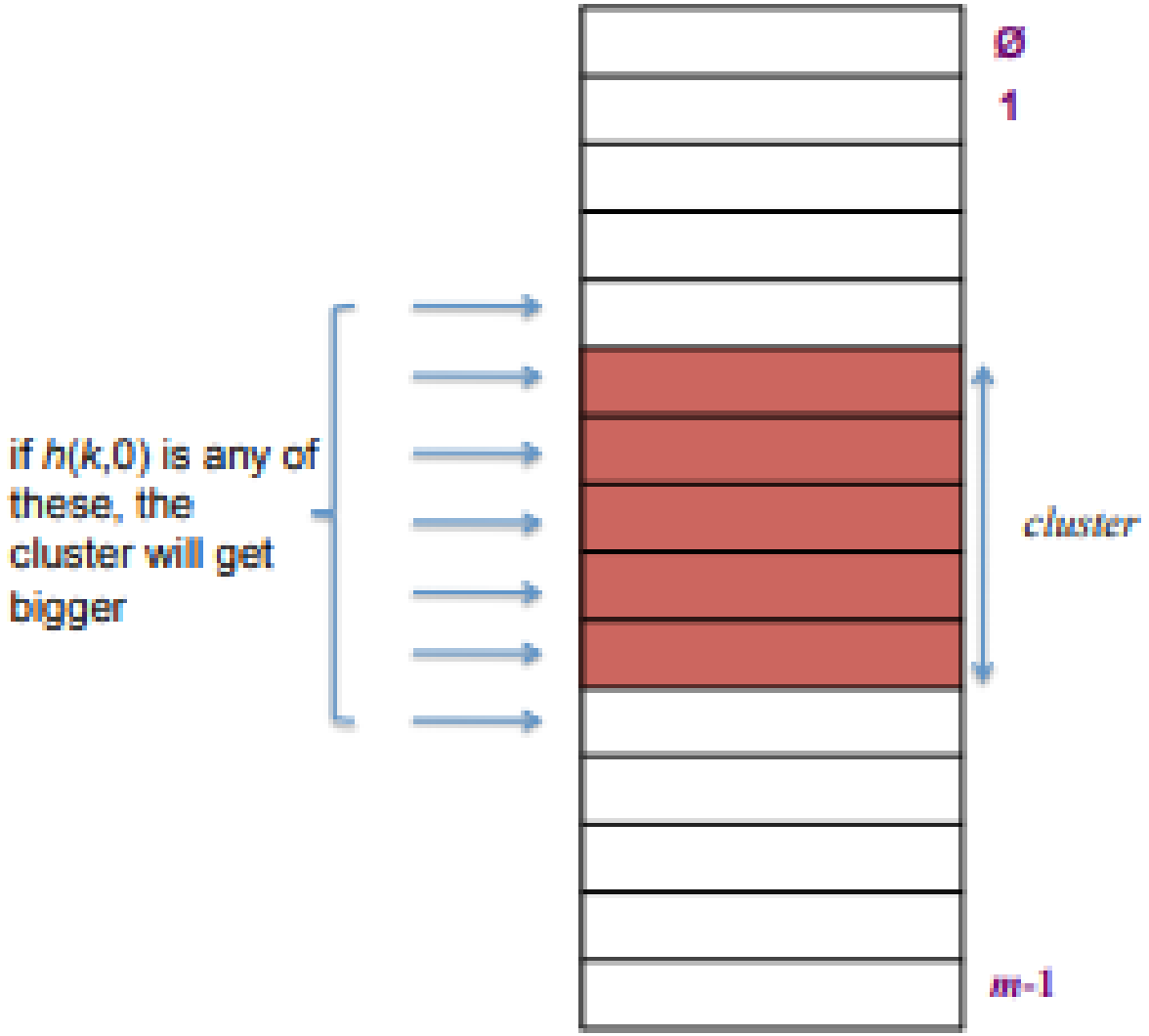
Probing Strategies

- Linear Probing
- Double Hashing

Linear Probing

$h(k,i) = (h'(k) + i)\%m$ where
$h'(k)$ is ordinary hash function

- Like street parking
- Problem? clustering—cluster: consecutive group of occupied slots as clusters become longer, it gets more likely to grow further
- Can be shown that for 0.01 < α < 0.99 say, clusters of size $\Theta(\log n)$



Linear Probing

- One idea that could come to mind is to do linear probing using a jump size p
- It turns out that if the jump size is fixed, this does not make the slightest difference with respect to our "standard" linear probing (i.e., with jump size $p = 1$)

- So... What if we could choose a different jump size for each insertion?
  - For example, the first insertion uses jump size 1, second insertion jump size 2, and so on... Would this work, and avoid the issue of clustering?

Double Hashing

What if the jump size was a function of the value being inserted?

$h(k,i) = \big(h_1(k) + i \cdot h_2(k)\big) mod\ m$

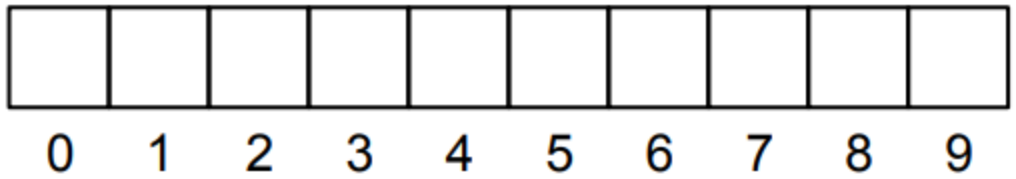
- where $h_1(k)$ and $h_2(k)$ are two ordinary hash functions.

- Example: What about taking advantage of the computation already done, like multiplication method, and choose a different block of bits for the second hash function?

Double Hashing - Example

- The hash table uses size 10
- For the hash function, multiply the value times 117 and keep the right-most digit
- For the second hash function (jump size), just use the same result, and take the second digit

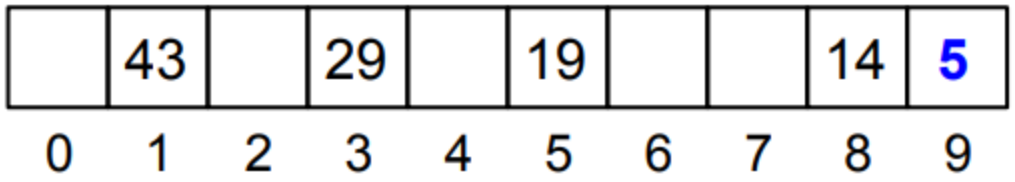
- We'll insert values 14, 29, 43, 19, and 5 into the initially empty hash table:



Double Hashing - Example

insert values 14, 29, 43, 19, and 5

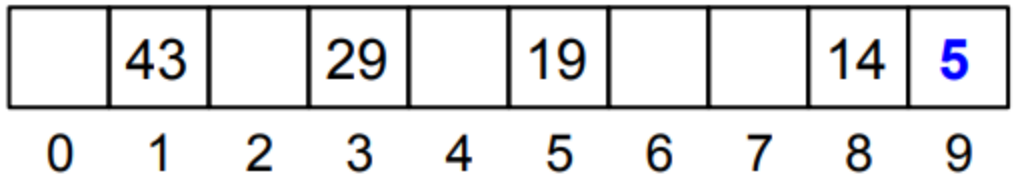
- 14×117 = 1638 -> bin 8
- 29×117 = 3393 -> bin 3
- 43×117 = 5031 -> bin 1
- 19×117 = 2223 -> bin 3 (jump size 2) -> bin 5
- 5×117 = 585 -> bin 5 (jump size 8) -> bin 3 -> bin 9



Double Hashing - Example

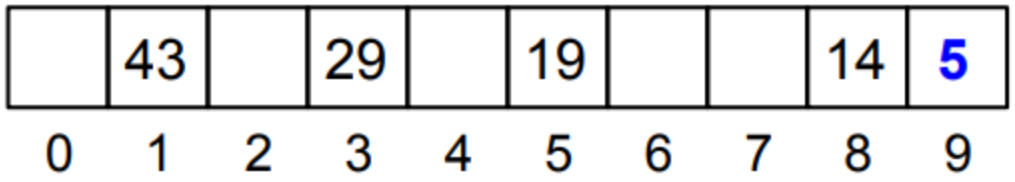
Insert values 14, 29, 43, 19, and 5

- Let's try inserting 59
- 59×117 = 6903 -> bin 3 (jump size 0) Oops! make it 1
- Let's try inserting 74
- 74×117 = 8658 -> bin 8 (jump size 5) Oops!
- Why does this happen? How do we fix it?



Double Hashing - Example

- The problem is that the cycle is given by the least-common-multiple of the two values.
- We'd like it to be the product of the two numbers, but if the numbers are not relatively prime (i.e., share common factors), then the LCM is lower.
- The idea is that the jump size and the table size should be relatively prime to guarantee a permutation



Double Hashing

- The idea is that the jump size and the table size should be relatively prime
  - This guarantees that every bin will be visited before cycling and start repeating bins
  - And this is easy if one of the numbers (e.g., the table size) is prime
- Problems:
  - Modulo operations become expensive
  - Also, dynamically growing the size gets complicated and expensive

Double Hashing

One solution:

- $m = 2^r$
- make $h_2(k)$ always odd, how?

- Suppose we have two hash function $h(k) = k\%7$ and $h'(k) = 5 - (k\%5)$. Given the ordered input {35,40,21,33,26} and an empty hash table with size 7.

- Show the hash table using open addressing and double hashing of $h(k)$ and $h'(k)$ after insert all the inputs:

Keys	$h(k)$	$h'(k)$
35	0	5
40	5	5
21	0	4
33	5	2
26	5	4

Insert the keys 8, 4, 11, 20, 15, 17, 10 in order into a hash table with an initial size of 6 that uses open addressing with linear probing and jump size of 1. The hash function is defined as: $h(k) = k\%SIZE$, where $SIZE$ is the current size of the table (initially, $SIZE = 6$). Resizing should be performed as soon as the table is completely full and will double the size of the table (note that the increased table size is not required to be a prime number). Show how the table looks like immediately before the resizing, as well as the result after all insertions.

Running time with Open addressing

- Uniform Hashing Assumption
- Each key is equally likely to have any one of the $m!$ permutations as its probe sequence, not really true, but double hashing can come close
- Under the uniform hashing assumption, the next operation has expected cost $O\left(\frac{1}{1-\alpha}\right)$, where $\alpha = \frac{n}{m}$.

Running time with Open addressing

- $O\left(\frac{1}{1-\alpha}\right)$
- Proof:

Always make a first probe.
With probability n/m, first slot occupied.
In worst case (e.g. key not in table), go to next.
With probability $\frac{n-1}{m-1}$, second slot occupied.
Then, with probability $\frac{n-2}{m-2}$, third slot full.
Etc. (n possibilities)

$$
\begin{aligned}
\text{So expected cost} &= 1 + \frac{n}{m}(1 + \frac{n-1}{m-1}(1 + \frac{n-2}{m-2}(\cdots) \\
\text{Now } \frac{n-1}{m-1} \leq \frac{n}{m} &= \alpha \\
\text{So expected cost} &\leq 1 + \alpha(1 + \alpha(1 + \alpha(\cdots))) \\
&= 1 + \alpha + \alpha^2 + \alpha^3 + \cdots \\
&= \frac{1}{1-\alpha}
\end{aligned}
$$

Open Addressing vs. Chaining

- Open Addressing: better cache performance (better memory usage, no pointers needed)

- Chaining: less sensitive to hash functions (OA requires extra care to avoid clustering) and the load factor α (OA degrades past 70% or so and, in any event, cannot support values larger than 1)

Hashing in Security

- We can detect tampering in data (a file, or a message) if we also store a hash of the data.
- Particularly if we use a cryptographic hash function (e.g., MD5, SHA1), which has specifically been designed to be hard to invert.
- Difficult to change the data while keeping the same hash: by the birthday paradox, it takes ~2^{64} guesses to find a single colliding pair of keys if our hash output is a 128-bit integer.
- Store passwords on a computer system as hashes instead of “in the clear” – still allows you to log in!

The Birthday Paradox

- How many people do you need to assemble before there is a reasonable chance (say, probability ½) that some pair has a shared birthday?

A=No one share the same birthday with others.
P(A) = (1/365)^n x (365*364*363…)
n = 23
P(A) = 0.49

The Birthday Paradox

- How many people do you need to assemble before there is a reasonable chance (say, probability ½) that some pair has a shared birthday?
- Only 23!
- In general, if there are N possible birthdays, the answer is roughly $\sqrt{\mathrm{N}}$.
- If you want to estimate the number of fish N in a lake, …

The Birthday Paradox

- How many people do you need to assemble before there is a reasonable chance (say, probability ½) that some pair has a shared birthday?
- Only 23!
- In general, if there are N possible birthdays, the answer is roughly $\sqrt{\mathrm{N}}$.
- If you want to estimate the number of fish N in a lake, repeatedly mark them and throw them back; you expect to catch a marked fish after ~ $\sqrt{\mathrm{N}}$ trials.

Substring Pattern Matching

Input: A text string t and a pattern string p.

Problem: Does t contain the pattern p as a substring, and if so, where?

E.g.: Is *Google* in the Bible?

Substring Pattern Matching

Brute Force Search

The simplest algorithm to search for the presence of pattern string p in text t overlays the pattern string at every position in the text and checks whether every pattern character matches the corresponding text character.

This runs in $O(nm)$ time, where $n = |t|$ and $m = |p|$.

Substring Pattern Matching

- Hashing

- Suppose we compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t.
- If these two strings are identical, clearly the resulting hash values will be the same. If the two strings are different, the hash values will almost certainly be different.
- These false positives should be so rare that we can easily spend the $O(m)$ time to explicitly check the identity of two strings whenever the hash values agree.

The Catch

- This reduces string matching to $n - m + 2$ hash value computations (the $n - m + 1$ windows of t, plus one hash of p), plus what should be a very small number of $O(m)$ time verification steps.
- The catch is that it takes $O(m)$ time to compute a hash function on an m-character string, and $O(n)$ such computations seems to leave us with an $O(mn)$ algorithm again.

The Trick

- Look closely at our string hash function, applied to the m characters starting from the j-th position of string S:
- $H(S, j) = S[j] + S[j+1]x + S[j+2]x^2 + \cdots + S[j+m-1]x^{m-1}$

- A little algebra reveals that
- $H(S, j-1) = (H(S, j) - S[j+m-1]x^{m-1})x + S[j-1]$

- Thus, once we know the hash value from the j position, we can find the hash value from the $(j-1)$ position for the cost of two multiplications, one addition, and one subtraction. This can be done in constant time.
- Therefore, the algorithm can be done in $O(n)$.