CSE 350: Advanced Data Structures and Indexes (Spring 2026)

Lecture 13: Hashing Techniques and Hash Indexes

3/10/2026 & 3/12/2026

University at Buffalo
Department of Computer Science
and Engineering
School of Engineering and Applied Sciences

In this lecture

- Hashing basics
- How to design a good hash function?
- External hash index design
- Hashing-based sketches

Hashing basics

- Hash function $h: U \to [M]$
    - U: key domain, $[M] = \{0,1,2, \ldots M-1\}$
    - *Deterministic*
    - Examples:
        - Multiplicative hashing for integers: $h(x) = \lfloor M \cdot frac(x * a) \rfloor$
            - a: a real number with a good mixture of 0s and 1s
            - $frac(y)$: the fractional part of a real number
            - can be efficiently implemented as $h(x) = \left(\frac{ax}{2^q}\right)\ mod\ m$ for appropriately chosen integers a, q, M
        - String hashing: SHA-1, MD5
            - often available off the shelf
            - can combine a *salt* to create different hash functions
                - e.g., SHA-1(concat(a, s)) for some randomly chosen string a
            - not that secure, but works well due to its efficiency

Hashing basics

- (In-memory) hash table
  - With a hash function $h\colon U \to [M]$

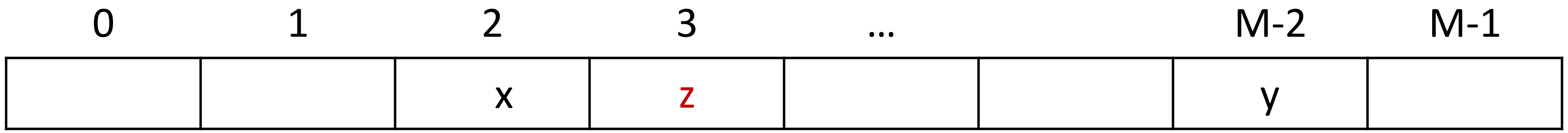
0	1	2	3	…		M-2	M-1
		x				y	

h(x) = 2
h(y) = m-2

Hashing basics

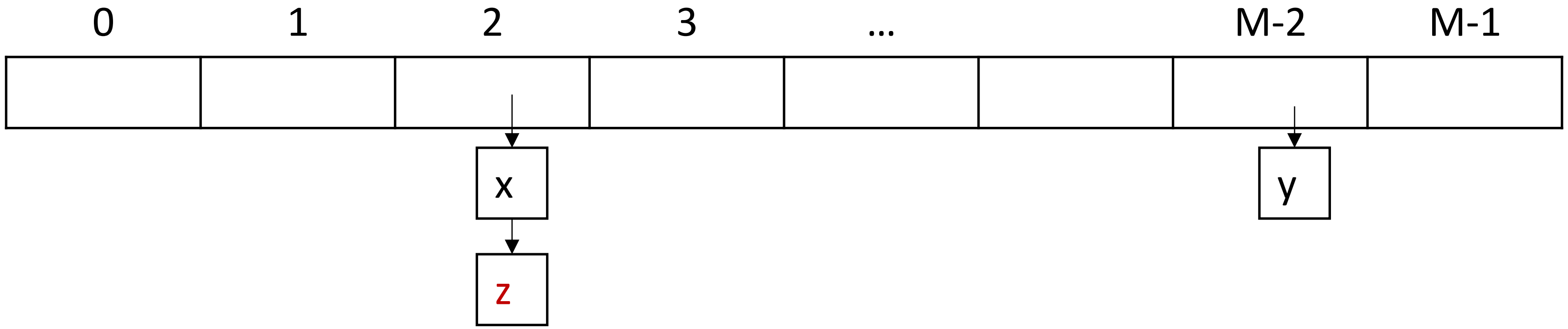
- (In-memory) hash table
  - With a hash function $h: U \rightarrow [M]$
  - How to handle collision?
    - Closed hashing vs open hashing
    - Sometimes also called open addressing vs closed addressing

closed hashing with linear probing



h(x) = 2
h(y) = M-2
h(z) = 2

open hashing with linked list



What might go wrong with hashing?

- Too many items with the same hash value
  - Any hash table design will fail in this case
- Why can that happen?
  1. Too many entries with the same key?
     - Not much that we can do, but we can try to incorporate other fields to make the keys distinct if it’s possible for the user to provide the entire key during lookups
     - Alternatively, consider using other types of index
  2. Hash collision
     - Some hash functions are prone to too many hash collisions
       - For instance, you’re hashing pointers of int64_t,
         - using modular hashing $h(x) = x \; mod \; m$ with $m = 2^d$ for some d is going to leave many buckets completely empty

Designing Good Hash Functions

- Formal set up: let U=[N] denote the numbers {0, 1, 2, . . . , N − 1}. For any set S ⊆ U, where |S|=n, we want to support these ***efficiently***:
  - add(x): add the key x to S
  - query(x): is the key q ∈ S?
  - delete(x): remove the key x from S

We consider a specific set S. Note that even though S is fixed, we don't know S ahead of time. Imagine it's chosen by an adversary from $\binom{N}{n}$ *possible choices.*

Our hash function needs to work well for any such a (fixed) set S.

Static vs Dynamic

- Static: Given a set S of items, we want to store them so that we can do lookups quickly. (e.g., a fixed dictionary).

- Dynamic: here we have a sequence of insert, lookup, and perhaps delete requests.
  We want to do all these efficiently.

Hash Function Basics

- We will perform inserts and lookups by
  an array A of M buckets, and a hash function h : U → {0,... ,M − 1} (i.e., h : U → [M]).
  Given an element x, the idea of hashing is we want to store it in A[h(x)].
  - If N=|U| is small, this problem is trivial. But in practice, N is often big.

- Collision happens when $x \neq y \land h(x) = h(y)$
  - Open hashing with linked list/overflow pages
  - Extendible/linear hashing can be used to alleviate the problem but can’t handle it well if there is skewness in hash values

Desirable Properties

- Small probability of distinct keys colliding
  - If x ≠ y, then the probability of h(x) = h(y) is “small”.
  - We will see a formal definition of this shortly.

- Small range: we want M to be small.
  - At odds with first desired property
  - ideally M=O(n) but it takes too much space.

- Small number of bits to store a hash function h.

- h should be easy to compute

- Given this, the time to lookup an item x is O(length of list A[h(x)])

Bad News

- One way to spread elements out nicely is to spread them randomly.
  - Unfortunately, we can't just use a random number generator to decide where the next element goes
    - won't be able to find it again.
  - `h` should be something "pseudorandom".

- (Bad news) For any deterministic hash function h (i.e., $|H|=1$), if $|U| \geq (n-1)M+1$, there exists a set S of n elements that all hash to the same location.
  - simple argument via pigeon hole principle
    - exercise in homework assignment

Randomness to Rescue

- Introduce a family of hash functions, H with |H|>1, that h will be randomly chosen from for each key (but use the same choice for the same key).

- Universal Hashing: if x ≠ y ∈ S then $\Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M$.

- Examples of universal hashing
  - $H = \{h(x) = \big((ax + b)\ mod\ p\big)\ mod\ M \mid a, b \in [p]\}$
    - where $p \geq M$ and is a prime number
  - Faster alternative
    - $H = \{h(x) = (ax + b\ mod\ 2^{w+m})\ div\ 2^{w}\}$
      - where $M = 2^{m}$, $a \in \{2k + 1 | k \in [0,2^{w+m-1})\}$, $b \in \{2^{w/2}k \mid k \in [0, 2^{w/2})\}$ (see [1], Thm. 5)

[1] Woelfel, P. Efficient Strongly Universal and Optimally Universal Hashing. In MFCS ’99. https://doi.org/10.1007/3-540-48340-3_24.

Property of Universal Hashing

- Property of universal hashing:
  - for any set S ⊆ U of size n,
    - if we choose h at random in a universal hash family H
      - for any x ∈ U (e.g., that we might want to lookup, x may not come from S)
        - the expected number of collisions between x and other elements in S is at most n/M.

- Proof:
  - Each y ∈ S (y ≠ x) has at most a 1/M chance of colliding with x by the definition of “universal”. So
  - Let C_{xy}= 1 if x and y collide and 0 otherwise.
  - Let C_x denote the total number of collisions for x. So, $C_x = \sum_{y \in S \land y \neq x} C_{xy}$.
  - We know E[C_{xy}] = Pr(x and y collide) ≤ 1/M.
  - So, by linearity of expectation, E[C_x] = $\sum_{y \in S \land y \neq x} E\left[C_{xy}\right] \leq$ n/M.

Perfect Hashing (for static case)

- We say a hash function is perfect for S if all lookups involve O(1) work.
- Naïve method: an $O(n^2)$ space solution
- Let H be universal and $M = n^2$. Then just pick a random h from H and try it out!

- Claim: If H is universal and $M = n^2$, then $\Pr_{h \sim H}(no\ collisions\ in\ S) \geq 1/2$

Naïve method: $O(n^2)$ space

- Proof:
  - Randomly pick a pair of different x and y from S
  - How many pairs (x,y) in S are there?
    - $n(n-1)/2$
  - For each pair, the chance they collide is ≤ 1/M by definition of “universal”
  - So, $\Pr_{h \leftarrow H}(exists\ a\ collision) \leq n(n-1)/2M = n(n-1)/2n^2 < 1/2.$ (by union bound)

An O(n) space solution (for static S)

- First, hash all values in S into a table of size n using universal hashing.
  - This will produce some collisions (unless we are extraordinarily lucky)
- Then, for each bin
  - rehash using the naïve method until having zero collisions.

Formally:

- a first-level hash function h and first-level table A,
- n second-level hash functions h1,... ,hn and n second-level tables A1,... ,An
- To lookup an element x, we first compute i = h(x) and then find the element in A_i $[h_i(x)]$.
- We omit the analysis of this method.

Dynamic S?

- Cuckoo hashing
  - Linear space
  - Constant lookup time

  - Pagh, Rasmus; Rodler, Flemming Friche (2001). "Cuckoo Hashing". *Algorithms — ESA 2001*

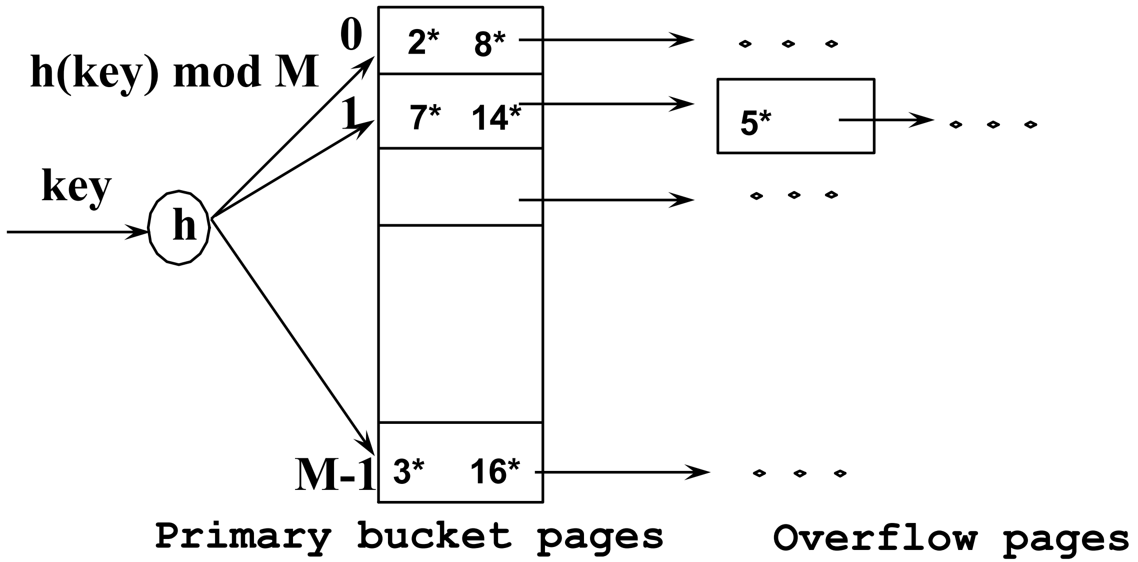
Hash-based index

- Hash-based index are best for equality searches
  - Does not support range searches
- Difference from in-memory hash table
  - Page oriented: multiple data entries per hash bucket
  - (Usually) open hashing
    - Probing needed to physically delete a data entry with closed hashing
  - Rehashing is very expensive! (reading + writing all the pages)

- Static vs dynamic hashing techniques
  - Static: fixed hash value domain $[M]$
    - Easy to implement, no rehashing overhead
    - Inefficient if number of records is large
  - Dynamic: grow hash value domain over time
    - Sometimes needs to rehash
      - How to amortize cost?
    - Scales gracefully with number of records if the choice of hash function is good

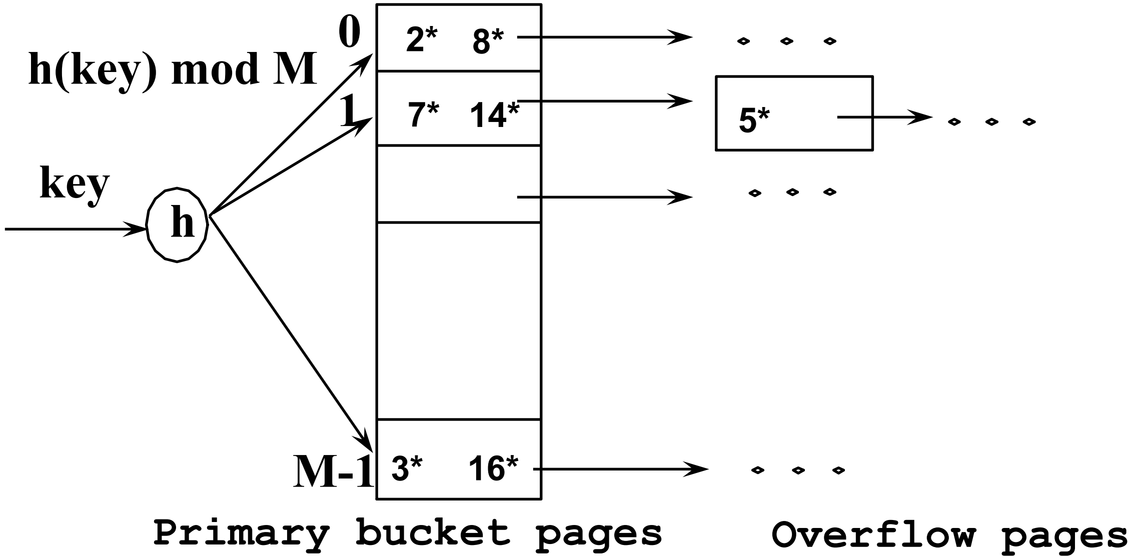
Static hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- $h(k)\ mod\ M$ = bucket to which data entry with key k belongs. (M = # of buckets)



Static hashing

- Buckets contain *data entries*.
- Hash function works on *search key* field(s) of record *r.*
  - Use its value MOD N to distribute values over range 0 ... N-1.
- Long overflow chains can develop over time and degrade performance.
  - ***Extendible*** and ***Linear Hashing***: dynamic techniques to fix this problem.



Extendible hashing

- When the primary page of a bucket gets full,
  - why not doubling the number of buckets and rehash?
  - reading and writing all the pages are very expensive!

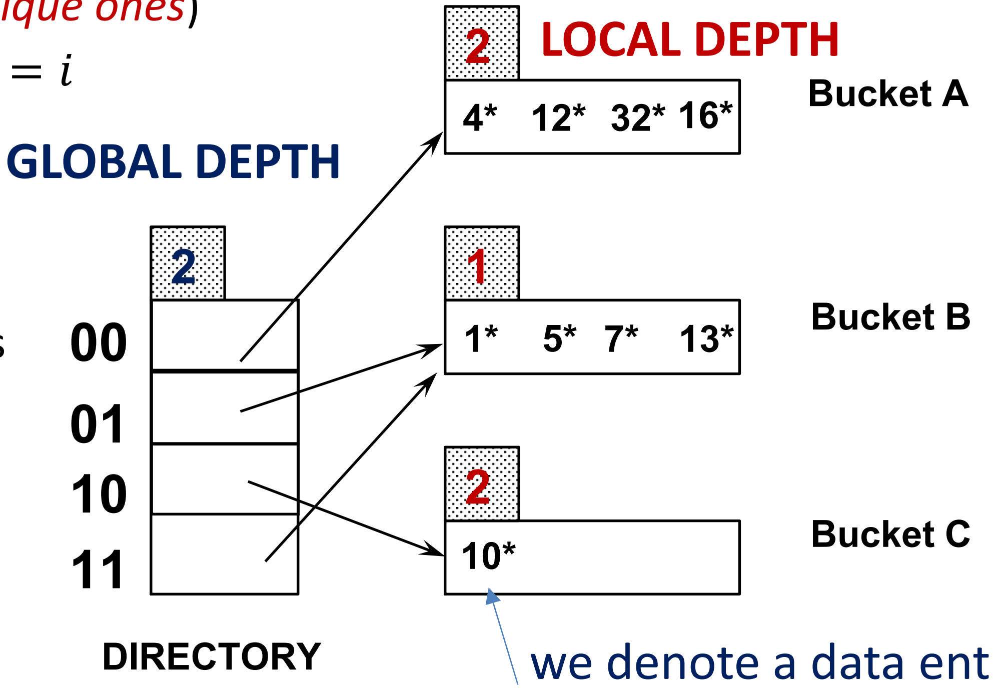
- Idea: use *directory of pointers* to buckets (these pointers are page numbers)
  - To double number of buckets, only need to double the directory size
  - Only split the bucket about to overflow. *No overflow page!*
- Why this works?
  - Directory is much smaller than the data files
  - Uses a family of hash functions $h_D: U \to [2^D]$
    - Trick is how to switch from h_D to h_{D+1} without rehashing for doubling number of buckets

Extendible hashing example

- Hash function $h: U \rightarrow [2^{32}]$ (or $[2^{64}]$ depending on the type of hash value)
  - Define $h_D(k) = h(k)\ mod\ 2^D$ -- therefore $h_D: U \rightarrow [2^D]$
  - Essentially taking *the lowest i bits of the key hash* as the hash value

- Directory: an array of pointers (page numbers) of size 2^D
  - D: global depth
  - Each points to a bucket p_i (*not necessarily unique ones*)
    - A data entry with key k is in p_i iff $h_D(k) = i$

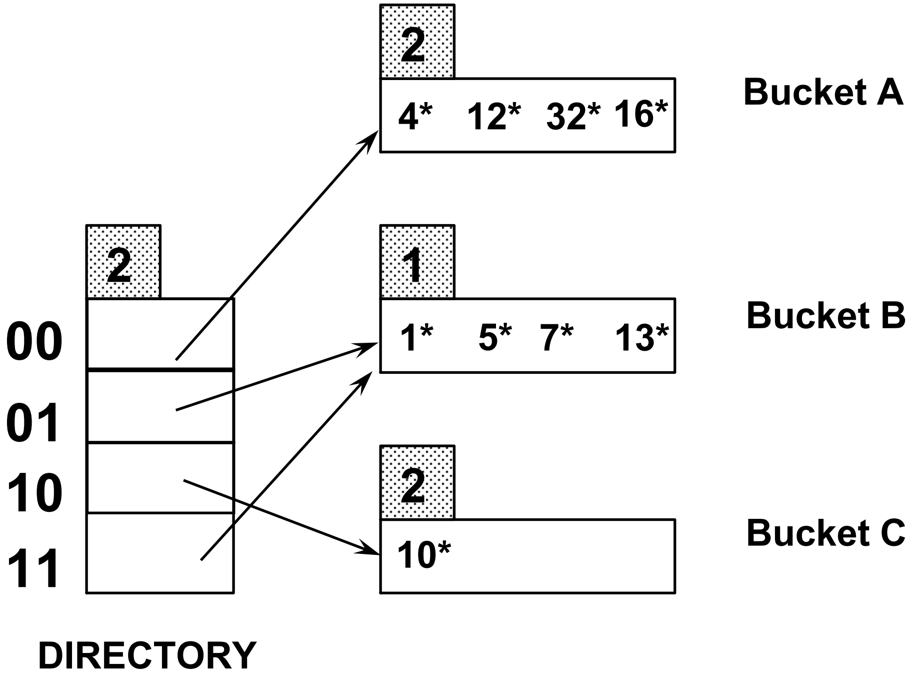
- Each bucket has a local depth d_i
  - Can be used to determine whether this bucket is currently shared by two hash values
    A bucket is not shared iff D == d_i
  - $h_D(k)$ may be different in a bucket
    - Question: what's in common?
    - *$h_{d_i}(k)$ are always the same*



we denote a data entry its hash value by h(r).

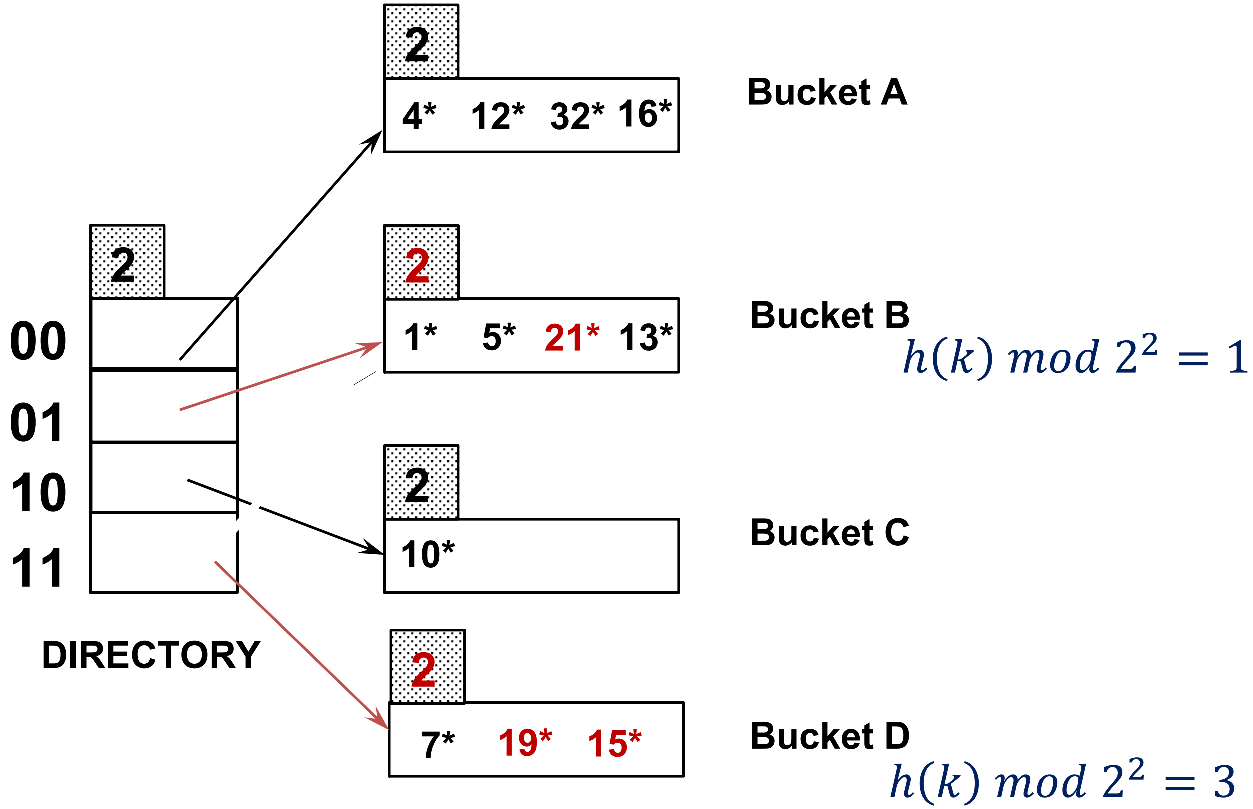
Handling inserts in extendible hashing

- Find the bucket p_i where the insertion belongs
- If there's room, insert it into p_i
- If not, split p_i before insertion
  - increment the local depth d_i
  - allocate a new page with the same (new) local depth
  - redistribute the data entries with the new page
  - double the global depth if local depth is now larger than global depth
    - also duplicate the old directory if global depth is doubled
  - set the pointer for the new page in the directory



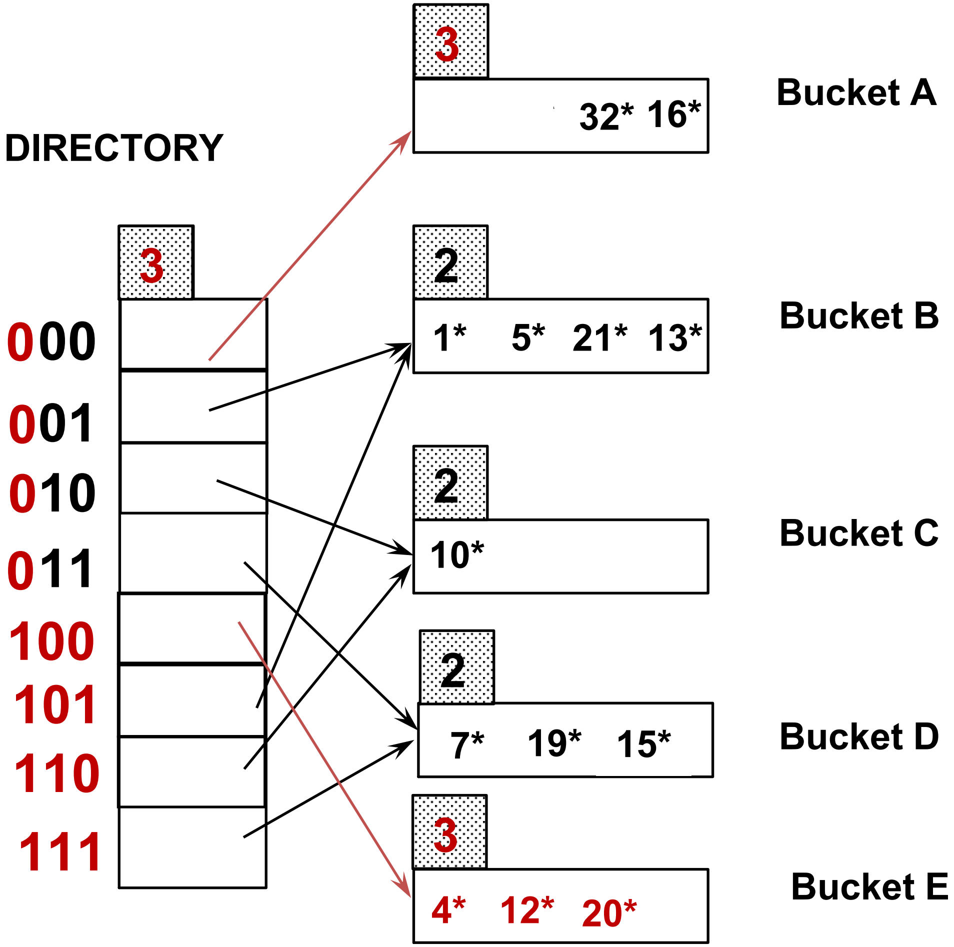
Example: inserting $21^*, 19^*, 15^*$

- $21 = (10101)_2$
- $19 = (10011)_2$
- $15 = (01111)_2$



Example: inserting 20^* (causing doubling)

- $20 = (10100)_2$
- Last **2** bits (00) tell us *20** belongs to A or E
  Last **3** bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page.



Notes on extendible hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - If the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!
    - *What if we still don’t have space after split?*
- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image’. If each directory element points to same bucket as its split image, can halve directory.

Linear hashing

- Another dynamic hashing scheme that handles long overflow chains without using a directory.

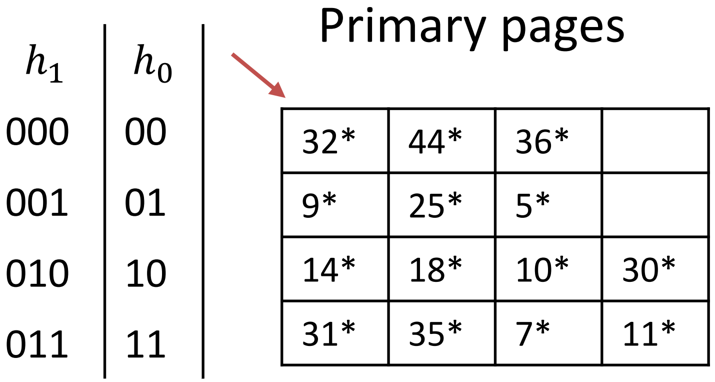
- Page to split is chosen in a *round-robin* fashion, not where it will overflow
  - LH allows using *temporary* overflow pages
  - If the hash values are reasonably uniform -- overflows will be resolved quickly

Handling insertion in linear hashing

- Also uses a family of hash functions
  - $h_i(k) = h(k)\ mod\ (2^i N)$
  - Initial size N does not need to be power of 2
- Proceeds in "rounds".
  Current round number is called level $l\ >= \ 0$
- There are $N_l = N * 2^l$ buckets at the beginning
  - *Next* initially set to 0
  - Invariant:
    - Buckets [0, Next) has been split in this round
    - Buckets [Next, N_l) are to be split in this round
- On insert
  - If the bucket for insertion is full
    - Add an overflow page and insert the data entry
    - Split Next bucket and increment Next
  - Use h_{l+1} to redistribute entries for a split bucket
- Round ends when $Next = N_l$
  - Start a new round, $Next \leftarrow 0, l \leftarrow l + 1$

$N\ = 4, l = 0$

Next = 0



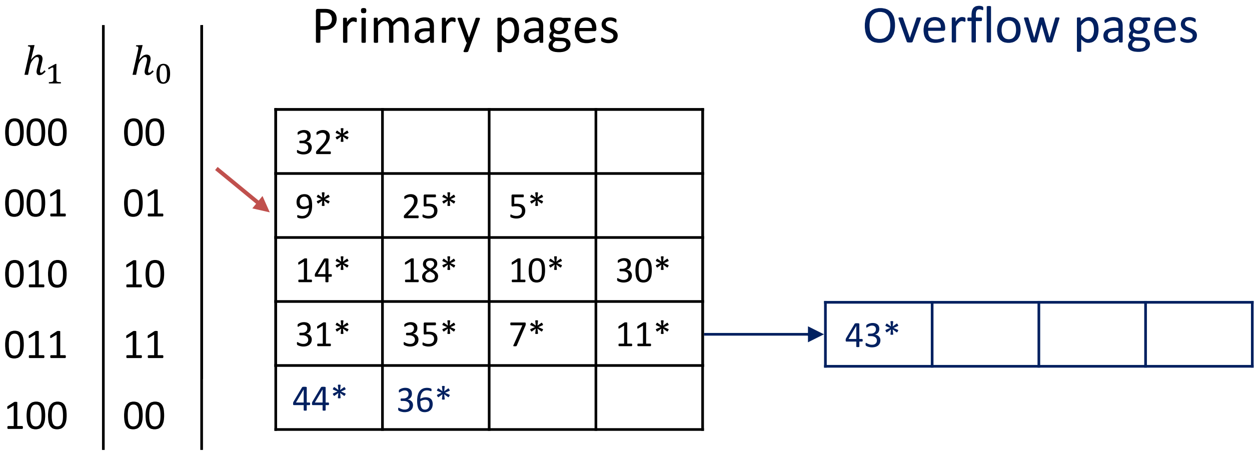
Primary pages

h_1	h_0				
000	00	32*	44*	36*	
001	01	9*	25*	5*	
010	10	14*	18*	10*	30*
011	11	31*	35*	7*	11*

Example: insert $43^* \; (101011)_2$

$N = 4, l = 0$

Next = 1



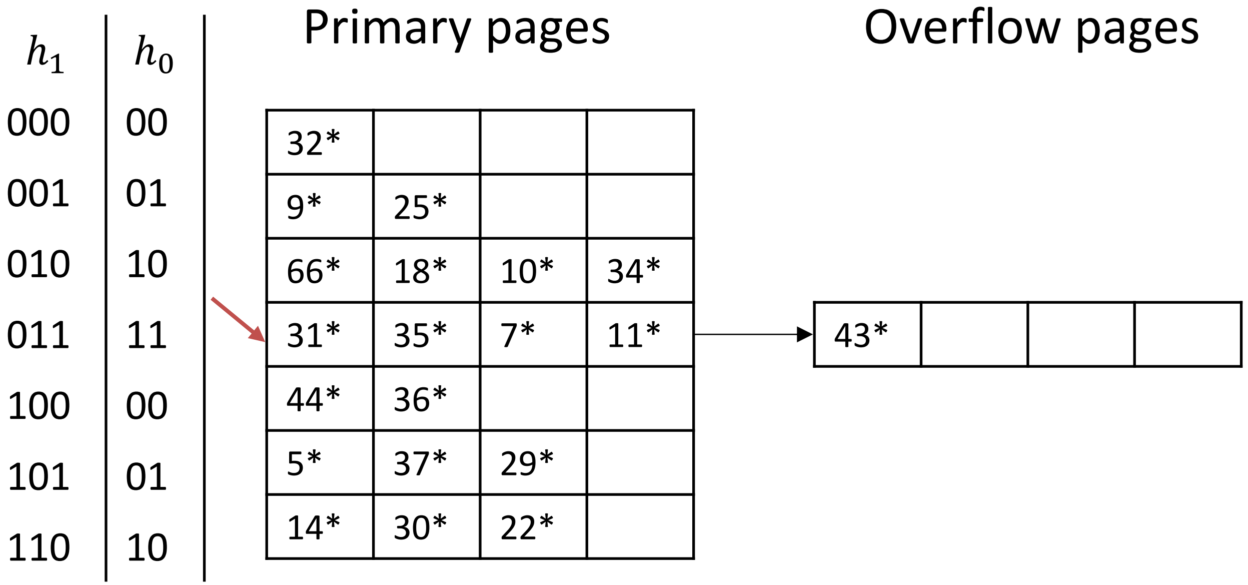
h_1	h_0	Primary pages				Overflow pages			
000	00	32*							
001	01	9*	25*	5*					
010	10	14*	18*	10*	30*				
011	11	31*	35*	7*	11*	43*			
100	00	44*	36*						

Example: end of a round

Insert $50^{*}\ (110010)_2$

$N\ = 4, l = 0$

Next = 3



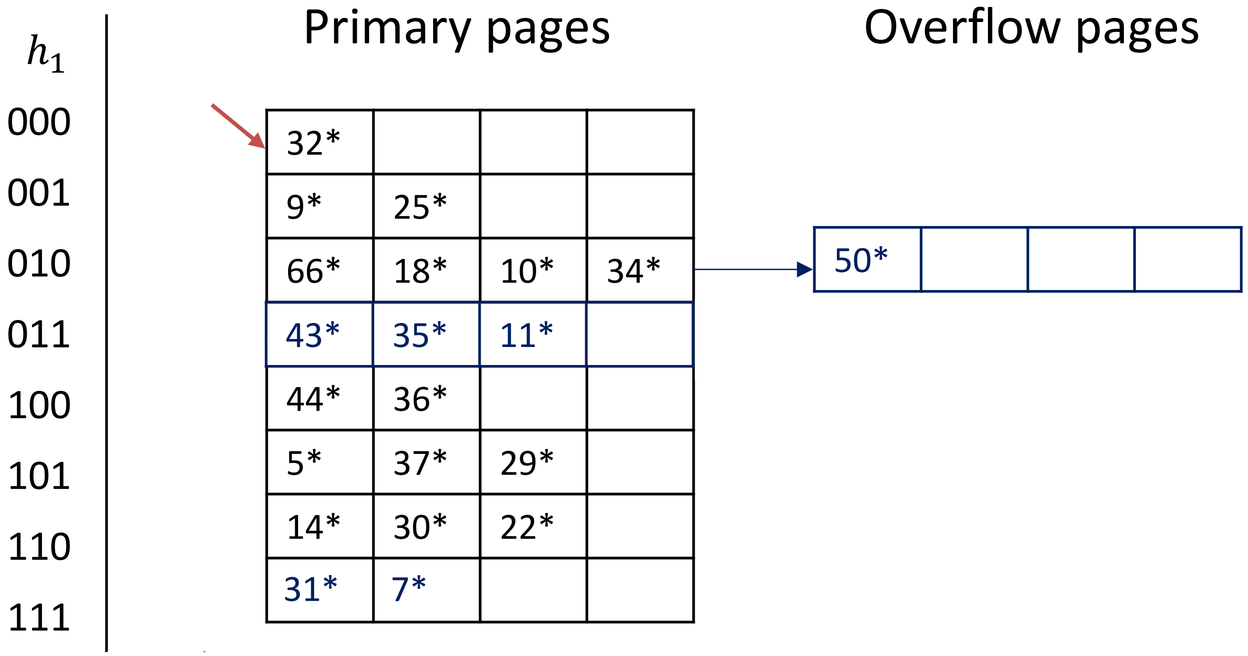
h_1	h_0	Primary pages				Overflow pages			
000	00	32*							
001	01	9*	25*						
010	10	66*	18*	10*	34*				
011	11	31*	35*	7*	11*	43*			
100	00	44*	36*						
101	01	5*	37*	29*					
110	10	14*	30*	22*					

Example: end of a round (cont’d)

Insert $50^* \ (110010)_2$

$N\ = 4, l = 1$

Next = 0



h_1	Primary pages				Overflow pages			
000	32*							
001	9*	25*						
010	66*	18*	10*	34*	50*			
011	43*	35*	11*					
100	44*	36*						
101	5*	37*	29*					
110	14*	30*	22*					
111	31*	7*						

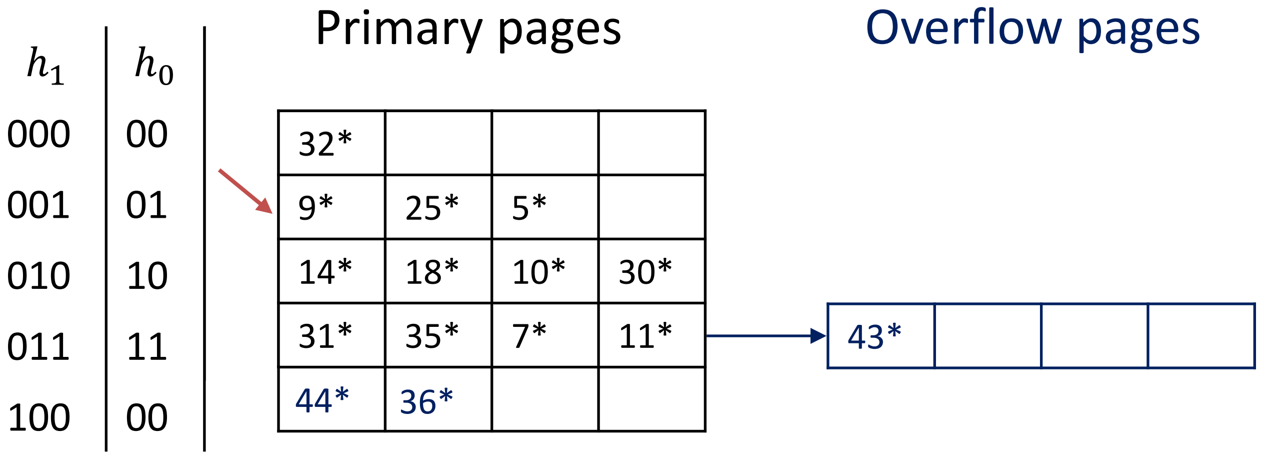
Linear Hashing Search Algorithm

- To find the bucket for a data entry k^*
  - Compute $h_l(k) = h(k)\ mod\ (2^l N)$
    - If $h_l(k) \geq Next$
      - Bucket $h_l(k)$ is the bucket for k^* (because it hasn't been split in this round)
    - Otherwise,
      - k^* could belong to either bucket $h_l(k)$ or bucket $h_l(k) + 2^l N$
      - Compute $h_{l+1}(k)$ to find out

$N = 4, l = 0$

Next = 1

Find 32^*?
Find 14^*?
Find 43^*?



h_1	h_0	Primary pages				Overflow pages			
000	00	32*							
001	01	9*	25*	5*					
010	10	14*	18*	10*	30*				
011	11	31*	35*	7*	11*	43*			
100	00	44*	36*						

Notes on linear hashing

- If hash values are skewed
  - because of key skew or bad hash function
  - *then will still have long overflow chains*
- Delete: the reverse of insertion algorithm

Composite keys in hash index

- Composite key: multiple fields as the key (f1, f2, …, fk)

- How to handle composite keys in hash index?
  - Combine the hash values of each field together
  - Many libs available, e.g., boost::hash_combine, absl::Hash::combine(), etc …

    e.g., in boost:

    ```
    seed ^= hash_value + 0x9e3779b9 + (seed<<6) + (seed>>2);
    ```

- Search with composite keys
  - Must specify *all the fields*, equality search only
  - Can't perform partial key search
    - e.g., hash index on (sid, login)
      - may be used for predicate `sid = 12345 AND login = 'alice'`
      - but not `sid = 12345`
      - nor `login = 'Alice'`

Sketches

- Sketches - compact data structures for approximate query answering
  - many relies on hashing

- Examples
  - Frequency estimation: Count-Min, AMS sketch
  - Set membership testing: bloom filter
  - Distinct counting: HyperLogLog

- We will examine bloom filter today as an example

Set membership testing

Assumption: $|U| \gg |S|$

Exact set membership testing

Given a finite Universe U, and a set of distinct items $S = \{x_1, x_2, \ldots, x_N\} \subset U$, for any $x \in U$, we want to answer whether $x \in \mathrm{S}$.

Approximate membership testing

Given a finite Universe U, and a set of distinct items $S = \{x_1, x_2, \ldots, x_N\} \subset U$,

for any $x \in U$, we want to answer whether $x \in S$ with some errors as follows:

(1) If $x \in S$, we always answer yes.

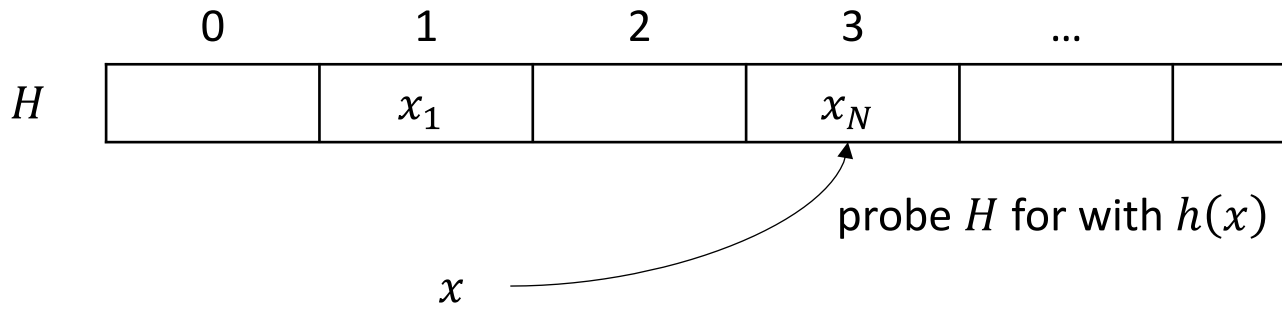
(2) If $x \notin S$, we may answer yes or no.

We want the false positive rate $\Pr(x \notin S \wedge answers\ yes)$ to be small.

Exact set membership testing

- Hash table H
  - Insertion: for each $x_i \in S$, add it to H
  - Query: for any $x \in U$, probe H with $h(x)$

- Space: $O(|S| \log|U|)$
- Query: $O(1)$ in expectation (or $O(1)$ in worst case if using Cuckoo hashing)

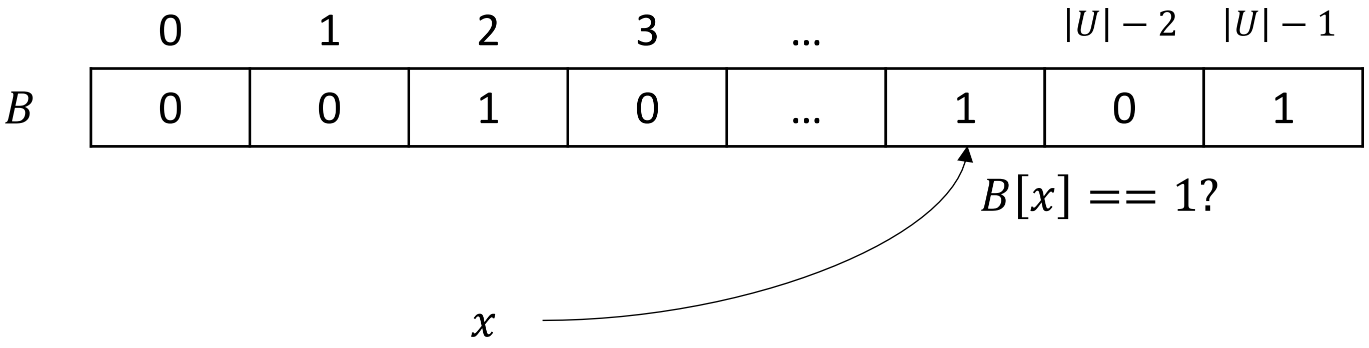


Exact set membership testing

- If $U = \{x \in \boldsymbol{Z} \mid x \in [0, |U|)\}$
  - can also use a bitmap B
    - Insertion: for each $x_i \in S$, set $B[x_i] = 1$
    - Query: for any $x \in U$, check $B[x] == 1$?

$\gg O(|S| \log|U|)$ if $|U| \gg |S|$

  - Space: $O(|U|)$
  - Query: $O(1)$ in worst case



	0	1	2	3	…		$\lvert U\rvert - 2$	$\lvert U\rvert - 1$
B	0	0	1	0	…	1	0	1

$B[x] == 1?$

x

Approximate set membership testing

- Bloom filter
  - Idea: maintain a bitmap over a hash function $h: U \rightarrow [M]$
  - Correctness?
    - $x \in S \Rightarrow B[h(x)] = 1$
  - FP rate?
    - Depends on the choice of the hash function

Analysis: suppose $h \in \mathcal{H}$ where $\mathcal{H}$ has a *uniformity* property: $\forall x \in U, y \in [M], Pr\{h(x) = y\} = \frac{1}{M}$

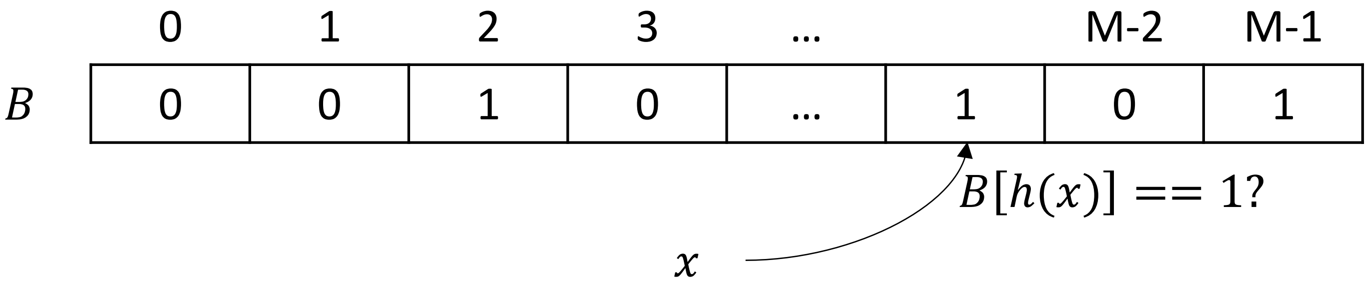
=> $\forall i \in [|U|], \Pr\{B[i] == 0\} = \left(1 - \frac{1}{M}\right)^{|S|} \approx e^{-\frac{|S|}{M}}$

=> $\forall x \in U - S$, FP rate = $1 - \Pr\{h(x) == 0\} = 1 - e^{-\frac{|S|}{M}}$

Problem?

For FP rate = ε, spaced required: $M = -\frac{|S|}{\ln(1-\varepsilon)}$

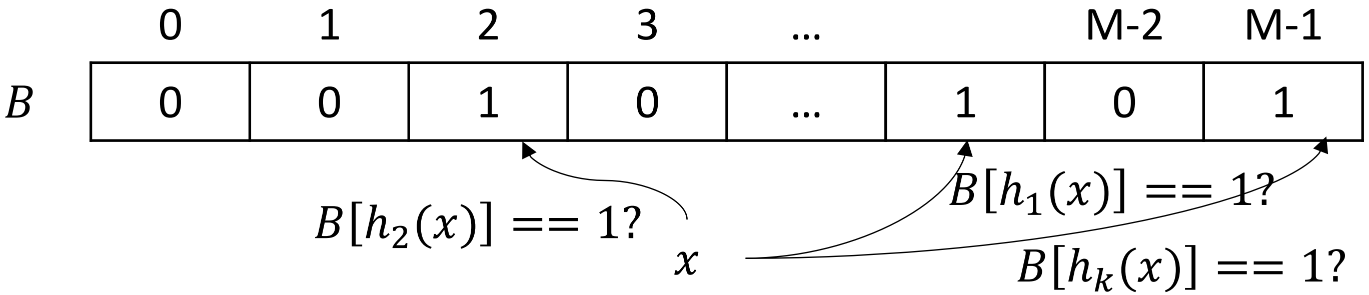
E.g., $\varepsilon = 0.01 \Rightarrow M \approx 99.5|S|$ > $|S| \log|U|$



	0	1	2	3	…		M-2	M-1
B	0	0	1	0	…	1	0	1

Approximate set membership testing

- Bloom filter
  - Idea: maintain a bitmap over k independent hash functions $h_1, h_2, \dots, h_k : U \rightarrow [M]$
    - Insert x: set $B[h_1(x)] = 1, \mathrm{B}[\mathrm{h}_2(\mathrm{x})] = 1, \dots \mathrm{B}[h_k(x)] = 1$
    - Query x: returns $B[h_1(x)] == 1 \wedge B[h_2(x)] == 1 \wedge \cdots \wedge B[h_k(x)] == 1?$
  - Correctness?
    - $x \in S \Rightarrow B[h_1(x)] == 1 \wedge B[h_2(x)] == 1 \wedge \cdots \wedge B[h_k(x)] == 1$



	0	1	2	3	…		M-2	M-1
B	0	0	1	0	…	1	0	1

$B[h_2(x)] == 1?$

x

$B[h_1(x)] == 1?$

$B[h_k(x)] == 1?$

Approximate set membership testing

- Bloom filter
  - Idea: maintain a bitmap over k independent hash functions $h_1, h_2, \ldots, h_k: D \rightarrow [M]$
    - Insert x: set $B[h_1(x)] = 1, \mathrm{B}[\mathrm{h}_2(\mathrm{x})] = 1, \ldots \mathrm{B}[h_k(x)] = 1$
    - Query x: returns $B[h_1(x)] == 1 \wedge B[h_2(x)] == 1 \wedge \cdots \wedge B[h_k(x)] == 1?$
  - Correctness?
    - $x \in S \Rightarrow B[h_1(x)] == 1 \wedge B[h_2(x)] == 1 \wedge \cdots \wedge B[h_k(x)] == 1$
  - FP rate?

Analysis: suppose $h_1, h_2, \ldots, h_k \in \mathcal{H}$ are independent and uniform

$$=> \forall i \in [|U|], \Pr\{B[i] == 0\} = \left(1 - \frac{1}{M}\right)^{k|S|} \approx e^{-\frac{k|S|}{M}}$$

$$=> \forall x \in U - S, \text{FP rate} = 1 - \Pr\{\wedge_{1 \le i \le k} h_i(x) == 0\} \approx \left(1 - e^{-\frac{k|S|}{M}}\right)^k$$

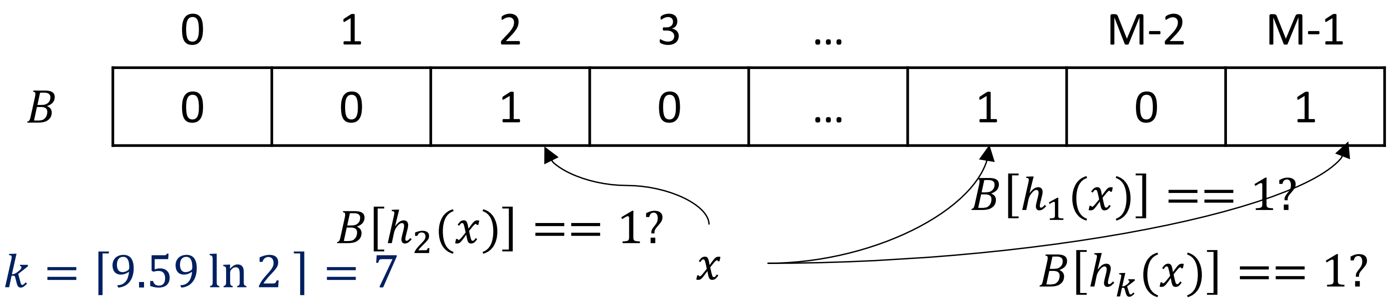
Solving for minimum FP rate:

$$k_{opt} = \frac{M}{|S|} \ln 2, \varepsilon_{opt} = e^{-\frac{M}{|S|}(\ln 2)^2}$$

=> For FP rate = ε, spaced required:

$$M = -\frac{\ln \varepsilon}{(\ln 2)^2} |S|$$

Example: $\varepsilon = 0.01 \Rightarrow M \approx 9.59|S| \ll |S| \log|U|, k = \lceil 9.59 \ln 2 \rceil = 7$



	0	1	2	3	…		M-2	M-1
B	0	0	1	0	…	1	0	1