DATA STRUCTURE: HASHING

HASH TABLE

The implementation of hash tables is frequently called hashing.

Hashing is a technique used for performing insertions, deletions, finds in constant average time.

Operations such as findMin, findMax, and the printing of the entire table in sorted order in linear time are not supported.

More for retrieving information than for analysis

GENERAL IDEA

You can consider a hash table as an Array of items

Items are referenced by a key—i.e. student ID to pull up their records.

Each key is mapped into some number in the range 0 to TableSize-1 and placed in the appropriate cell.

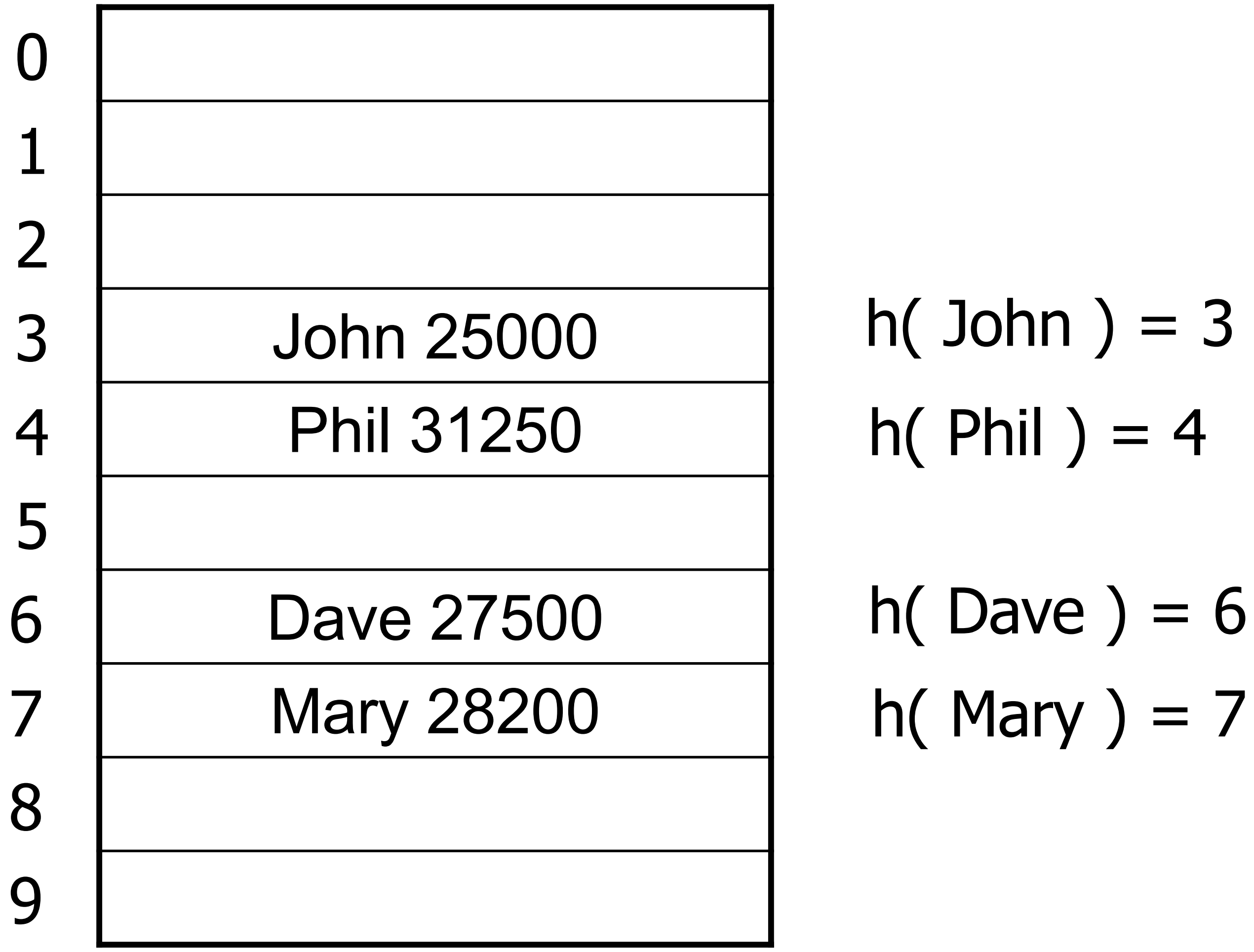
This mapping is called a hash function.

- h(*k*) = *index* where 0 ≤ *index* ≤ TableSize-1

<u>Ideally</u>

- If element e has key *k* and *h* is hash function, then e is stored in position *h(k)* of table.
- To search for e, compute *h(k)* to locate position. If no element, dictionary does not contain e.

AN IDEAL HASH TABLE



HASH FUNCTIONS

An ideal hash function should:

- be simple to compute.
- ensure that any two distinct keys get different cells.

Assumptions:

- `K`: an unsigned 32-bit integer
- `M`: the number of buckets (the number of entries in a ***hash table***)

Goal:

- We seek a hash function that distributes the keys evenly among the cells

MAIN ISSUES

What is the appropriate table size?

How to choose a good hash function?

Collision: What to do when two keys hash to the same value (cell)?

- There are a finite number of cells and a virtually inexhaustible supply of keys, how can you ensure that any two distinct keys get different cells?

WHAT IS THE APPROPRIATE TABLE SIZE? HASH FUNCTION

If the input keys are integers, then simply returning (key mod TableSize) is generally a reasonable strategy.

- `Hash(K) == K % M`

Suppose

- `M = 10,`
- `K = 2, 20, 34, 42,76`

What is wrong?

- Then `K % M = 2, 0, 4, 2, 6,...`
  - Since `10` is even, all even `K` are hashed to even numbers ...
  - Values of K may not be evenly distributed.
  - But Hash(K) needs to be evenly distributed.

WHAT IS THE APPROPRIATE TABLE SIZE?

To avoid this, it is usually a good idea to ensure that the table size is prime.

When the input keys are random integers, then this function (*key mod TableSize*) is not only very simple to compute but also distributes the keys evenly.

A SIMPLE HASH FUNCTION

What if the keys are strings?

the keys ➔ the cell position

- E.g., `Hash("test") = 98157`

Design Principles

- Use the entire key
- Use the ordering information
- Simple Hash Function: Add the ascii values of the keys

PROBLEMS IN THIS SIMPLE HASH FUNCTION

Not an equitable distribution!

Suppose:
TableSize = 10,007

All the keys are 8 or fewer characters long.

Then, the hash function typically can only assume values between 0 and 1,016 (127*8).

Order of chars in string has no effect.

- **hash("ab") = hash ("ba")**

ANOTHER POSSIBLE HASH FUNCTION

```
public static int hash( String key, int tableSize)
{
 return ( key.charAt(0) + 27 * key.charAt(1) +
 729 * key.charAt(2) ) % tableSize;
}
```

Only 28% of the table can be actually be hashed to.

TableSize = 10,007

possible combination of three characters:
$26^3 = 17,576$

Unfortunately, English is not random. A check of a reasonably large on-line dictionary reveals that the number of different combination is actually only 2,851.

BETTER HASH FUNCTION

```
Public static int hash(String key, int
 tableSize)
{int hashVal = 0;
 for( int i=key.length()-1; i>=0; i-- )
          hashVal = 27*hashVal +
 key.charAT(i);
hashVal %= tableSize;
 if ( hashVal < 0 )
          hashVal += tableSize;
 return hashVal;}
```

$h_k = k_0 + 27k_1 + 27^2k_2 + \ldots + 27^n k_n$

Horner's Rule:

$h_k = (\ldots((k_n)*27+k_{n-1})*27 + \ldots +k_1)*27+k_0$

reduces the number of necessary multiplications

- It is not the best with respect to table distribution, but is very simple and is reasonably fast.

WHAT TO DO WHEN TWO KEYS HASH TO THE SAME VALUE (CELL)?

Collision Resolution

If, when an element is inserted, it hashed to the same value as an already inserted element, then we have a collision and need to resolve it.

Two of the simplest methods:

- Separate chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Probing

SEPARATE CHAINING

Separate chaining is to keep a list of all elements that hash to the same value.

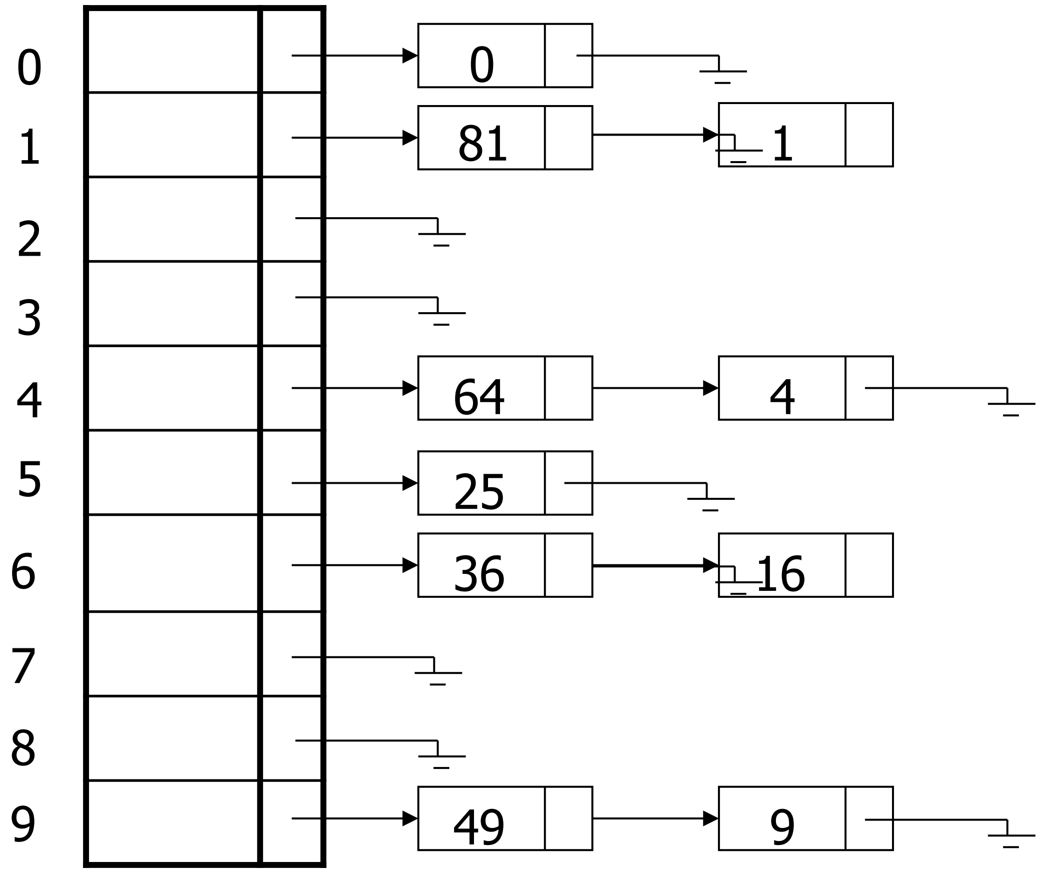
For example:

- assume that the keys are the first 10 perfect squares.
- hash(x) = x mod 10   (for simplicity)

A SEPARATE CHAINING HASH TABLE

hash(X) = x mod 10

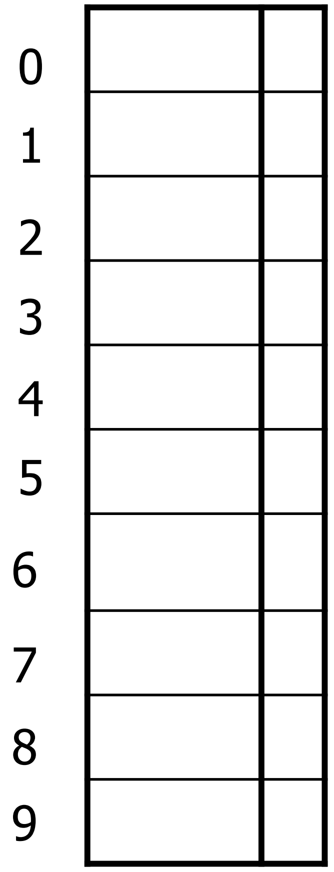
- Insert 1
- Insert 16



EXERCISES 5.1

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

- Separate chaining hash table



NOT ONLY LINKED LISTS

Any scheme could be used besides linked lists to resolve the collisions

A binary search tree or even another hash table would work.

If the table is large and the hash function is good, all the lists should be short.

It is not worthwhile to try anything complicated.

COMPLEXITY ON SEPARATE CHAINING

λ -- the load factor of a hash table.

$$\lambda = \frac{\text{the number of elements in the hash table}}{\text{the table size}}$$

The average length of a list is λ.

The general rule for separate chaining hashing is to make the table size as large as the number of elements expected. ($\lambda \approx 1$)

It is also a good idea to keep the table size prime to ensure a good distribution.

COMPLEXITY ON SEPARATE CHAINING (CONT'D)

The effort required to perform a search is the (constant time required to evaluate the hash function plus the time to traverse the list.)

- Unsuccessful Search: the number of nodes to examine is λ on average.
- Successful Search: $1+(\lambda/2)$ links would be traversed.

What are the disadvantages of Separate Chaining?

HASH TABLES WITHOUT CHAINING

Try to avoid buckets with separate lists

How ➔ use Probing Hash Tables

- If a collision occurs, alternative cells are tried until an empty cell is found.
- More formally, cells $h_0(x)$, $h_1(x)$, $h_2(x)$, … are tried in succession, where

$$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{TableSize} \text{ with } f(0) = 0.$$

The function, f, is the collision function.

LINEAR PROBING

hash(X) = x mod 10

$h_i(x) = (\text{hash}(x) + f(i)) \bmod 10$

Linear Probing: f(i) = i

$h_0(x), h_1(x), h_2(x), \ldots$

Inserting keys:

89, 18, 49, 58, 69, 35

0	49
1	58
2	69
3	
4	
5	35
6	
7	
8	18
9	89

EXERCISES 5.1

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

- Open address hash table using linear probing

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

THE LOAD FACTOR

Because all the data go inside the table, a bigger table is needed for open addressing hashing than for separate chaining hashing.

Generally, the load factor should be below $\lambda=0.5$ for open addressing hashing.

PRIMARY CLUSTERING

Primary clustering

- As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Any key that hashes into cluster will require several attempts to resolve the collision, and then it will be added to the cluster
  - blocks of occupied cells start forming.
- The expected number of probes in an unsuccessful search = the expected number of probes until we find an empty cell.
  - Since the fraction of empty cells is $(1-\lambda)$, the number of cells we expect to probe is $1/(1-\lambda)$.

QUADRATIC PROBING

Quadratic Probing is a collision resolution method that eliminates the primary clustering problem of linear probing.

The collision function: $f(i) = i^2$.

QUADRATIC PROBING

$\text{hash}(X) = x \bmod 10$

$h_i(x) = (\text{hash}(x) + f(i)) \bmod 10$

Quadratic Probing: $f(i) = i^2$

$h_0(x), h_1(x), h_2(x), \ldots$

Inserting keys:

89, 18, 49, 58, 69, 35

0	49
1	
2	58
3	69
4	
5	35
6	
7	
8	18
9	89

EXERCISES 5.1

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

- Open address hash table using quadratic proving

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

PERFORMANCE OF QUADRATIC PROBING

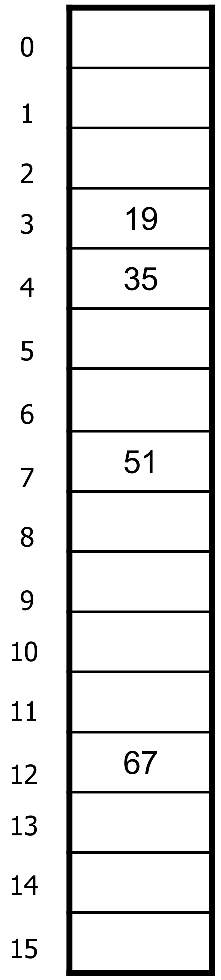
For linear probing it is a bad idea to let the hash table get nearly full because performance degrades.

For quadratic probing, there is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime.

Theorem: If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

QUADRATIC PROBING

- If the table is even one more than half full, the insertion could fail (although this is extremely unlikely).
- It is also crucial that the table size be prime. If the table size is not prime, the number of alternative location can be severely reduced.
- As an example, if the table size were 16, then the only alternative locations would be 1, 4, or 9 away. (insert 83)



DELETION IN OPEN ADDRESSING HASH TABLE

Standard deletion cannot be performed in an open addressing hash table, because the cell might have caused a collision to go past it.

What would the best (easiest) strategy for deletion open/closed be?

DELETION EXAMPLES

$hash(X) = x \bmod 10$

$h_i(x) = (hash(x) + f(i)) \bmod 10$

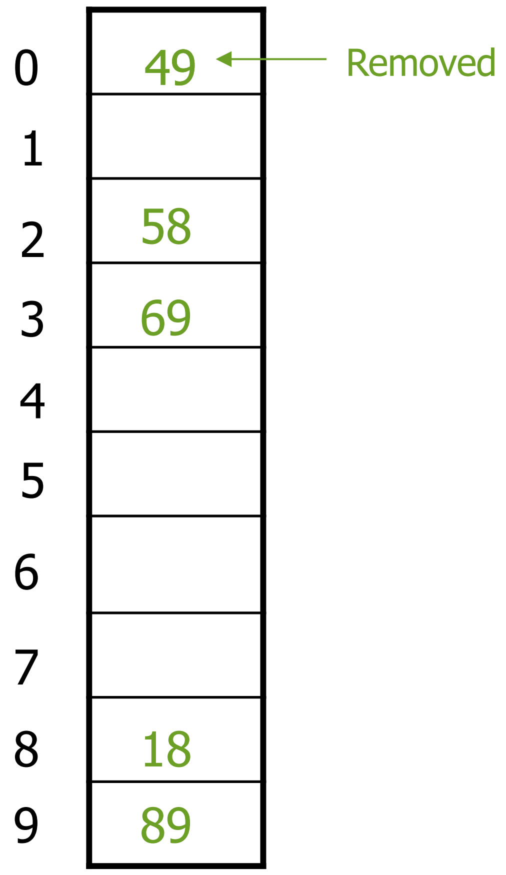
Quadratic Probing: $f(i) = i^2$

$h_0(x), h_1(x), h_2(x), \ldots$

Step 1: Insert 69

Step 2: Remove 49

Step 3: Find 69



0	49
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89

DISCUSSION: QUADRATIC PROBING

Although quadratic probing eliminates primary clustering, elements that hash to the same position will probe the alternative cells.

- This is known as secondary clustering.
- Double hashing eliminates this.

DOUBLE HASHING

One popular choice is $f(i) = i*hash_2(x)$

Apply a second hash function to x and probe at a distance $hash_2(x)$, $2hash_2(x)$, …, and so on.

A poor choice of $hash_2(x)$ would be disastrous. (e.g. $hash_2(x) = x \bmod 9$ and we want insert 99.)

CHOOSE A SECOND HASH FUNCTION

The function must never evaluate to zero.

It is also important to make sure all cells can be probed.

$hash_2(X) = R - (x \bmod R)$, with R a prime smaller than TableSize.

OPEN ADDRESSING HASH TABLE WITH DOUBLE HASHING

hash(X) = x mod 10

$h_i(x) = (hash(x) + i{*}hash_2(x)) \bmod 10$

Double Hashing:

$hash_2(x) = 7-(x \bmod 7)$

Inserting keys:

89, 18, 49, 58, 69, 35

0	69
1	
2	
3	58
4	
5	35
6	49
7	
8	18
9	89

EXERCISES 5.1

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

- Open address hash table with the second hash function h2(x) = 7 – (x mod 7).

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

ABOUT DOUBLE HASHING

It is important to make sure that the table size is prime.

If the table size is not prime, it is possible to run out of alternative locations prematurely.

If double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy. ($1/(1-\lambda)$)

This makes double hashing theoretically interesting.

Quadratic probing, however, does not require the use of a second hash function and is thus likely to be simpler and faster in practice.

REHASHING

If the table gets too full,

- the running time for the operations will start taking too long;
- Insertion might fail for open addressing hashing with quadratic resolution.

This can happen if there are too many removals intermixed with insertions.

A solution:

- to build another table that is about twice as big
- to scan down the entire original hash table, computing the new hash value for each (nondeleted) element, and inserting it in the new table.

AN EXAMPLE OF REHASHING

Open addressing hash table with linear probing

Old hash function: h(x) = x mod 7

Delete 24, Insert 23

0	6
1	15
2	23
3	24 *
4	
5	
6	13

New hash function: h(x) = x mod 17

0	
1	
2	
3	
4	
5	
6	6
7	23
8	
9	
10	
11	
12	
13	13
14	
15	15
16	

The size of a new table is 17, because this is the first prime that is twice as large as the old table size.

* Indicates deleted elements

WHEN TO REHASHING

To rehash as soon as the table is half full.

To rehash only when an insertion fails.

To rehash when the table reaches a certain load factor.

The running time of rehashing: O(N)

Actually it happens very infrequently: about one rehashing O(N) for every N/2 insertions.

Since performance does degrade as the load factor increases, the third strategy (with a good cut-off) could be best.

BLOOM FILTER

Suppose we have a simpler problem where we only need to find whether an entry exists

Use Bloom Filter!

Bloom Filter is formed of

- An array of size m-1 with elements initially set to zero
- K has functions where the hash function maps from 0 to m-1

Insertion

We had the entry k times, to get preferably k unique numbers from 0 to m-1

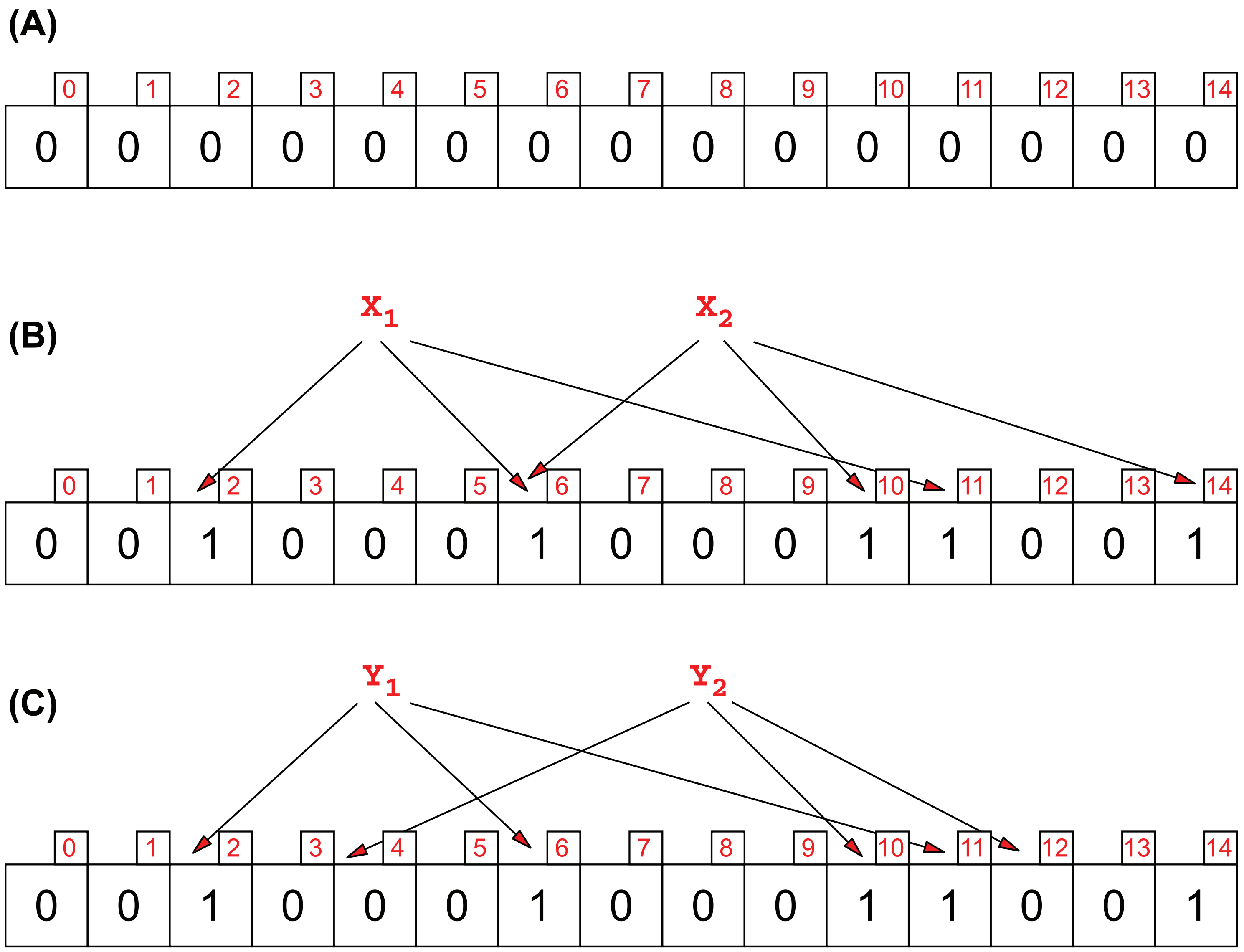
Each of these numbers is set to 1 in the array

Searching

We hash the entry k times to get the hash values

If all the indices corresponding to the hash values are 1 in the array, then the entry probably exists

If at least one index is zero then the entry definitely does not exist



(A)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(B)
X_1
X_2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
0 0 1 0 0 0 1 0 0 0 1 1 0 0 1
(C)
Y_1
Y_2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
0 0 1 0 0 0 1 0 0 0 1 1 0 0 1

CHOICE OF HASH FUNCTIONS

Use a generator for every hash function

Use a single hash function+random generator

Use double or triple hashing

PROBABILITY OF FINDING FALSE POSITIVE

$$p(n, m, k) = \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k$$

N= number of insertions
K= number of hash functions
M= size of array

PROBABILITY OF FINDING FALSE POSITIVE

Given the array is made on m bits

Then probability that any one of these bits is changed to 1 is $1/m$

Probability that the bit will be set to zero$=1-1/m$

Each k hash function is independent. So the probability that the bit will remain zero after k hash functions is $(1-1/m)^k$

Probability that the bit will remain zero after adding n entries is

- $(1-1/m)^{k*n}$

To obtain a false positive all the k bits for an entry V should be set to 1 independently. Thus the probability of a false positive is;

- $(1-(1-1/m)^{k*n})^k \approx (1-e^{-k*n/m})^k$

OPTIMAL VALUE OF K

If m (size of array) and n (number of entries) are constant, then how many hash functions are required to minimize the probability of false positives.

This is equivalent to minimizing $(1\text{-}e^{-k*n/m})^k$ with respect to k

$$f(k) = \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k = e^{k \cdot \ln\left(1 - e^{-\frac{k \cdot n}{m}}\right)}$$

Since e is constant we aim to minimize the exponent $g(k) = k \cdot \ln\left(1 - e^{-\frac{k \cdot n}{m}}\right)$

The derivative wrt k is $\frac{\partial g}{\partial k} = \ln\left(1 - e^{-\frac{k \cdot n}{m}}\right) + \frac{k \cdot n}{m} \cdot \frac{-e^{-\frac{k \cdot n}{m}}}{1 - e^{-\frac{k \cdot n}{m}}}$

This value becomes zero when $k = \ln(2)*m/n$

Plugging in the value of k for probability of false positives we get

$$f = \left(\frac{1}{2}\right)^k = (0.6185)^{\frac{m}{n}}$$