EE2331 Data Structures and Algorithms

Hashing

Outline

- Hash Functions
  - Perfect Hash
  - Minimal Hash
- Collisions Resolution
  - Chaining buckets
  - Linear probing
  - Quadratic probing
  - Double hashing
- Design of Hash Function

Indexing

- What is the purpose of the index in a book?
  - To help you to search the pages (that contain the keyword) quicker
- What happens if there is no index?
  - Probably you have to search the entire book page by page, line by line and word by word (sequential search!)

A Practical Problem

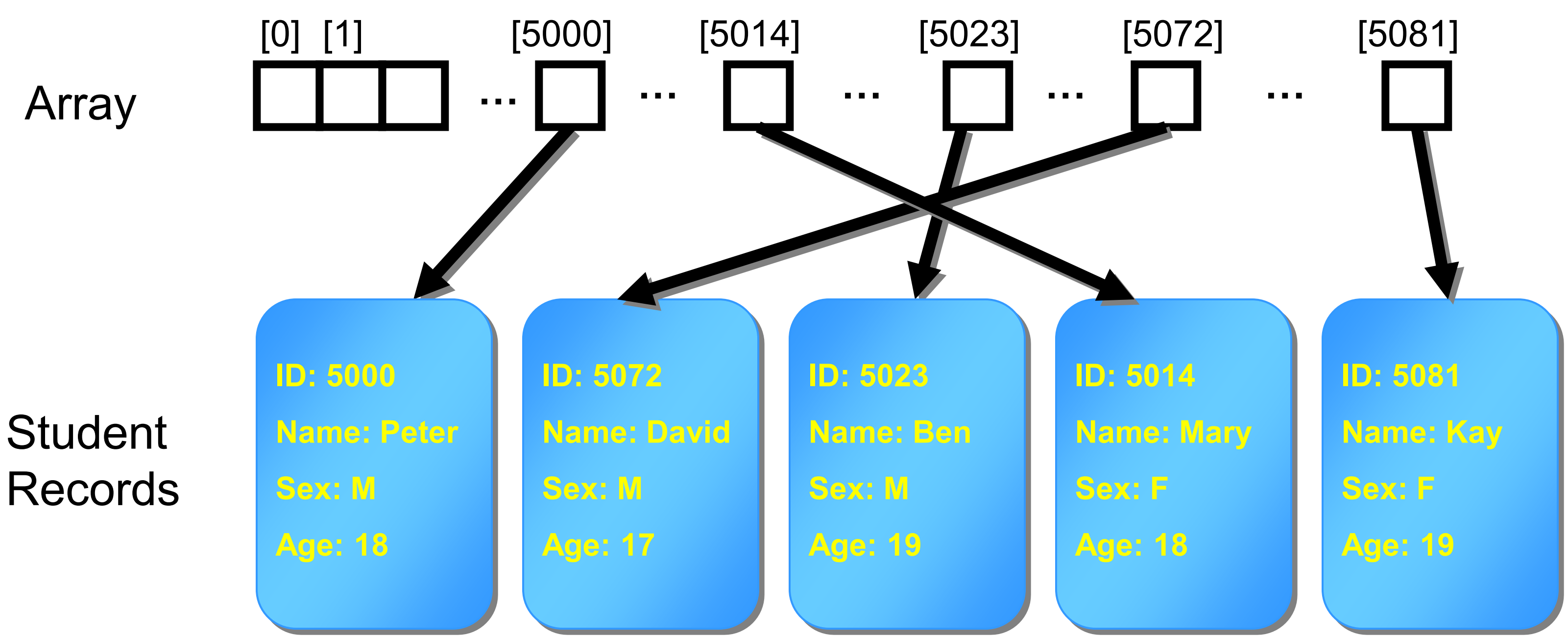
- Given a set of data/records, how can you locate a record by the Student ID?
  - How do you sort?
    - quick sort O(*nlogn*): n is the total number of records
  - How do you search?
    - Binary search O(*log n*)

Student Records

ID: 5000	ID: 5072	ID: 5023	ID: 5014	ID: 5081
Name: Peter	Name: David	Name: Ben	Name: Mary	Name: Kay
Sex: M	Sex: M	Sex: M	Sex: F	Sex: F
Age: 18	Age: 17	Age: 19	Age: 18	Age: 19

Indexing Data Record

- Using an array to hold pointers to the records
  - Use **Student ID** to index the records (ID = positions)
  - What is the time complexity now?
  - But waste too much space…



Hashing

- The term “hash” means to chop and mix!
- The objectives
  - Build an index for a set of elements/records
  - To allow fast **search** (also **insert**, **delete**) operations
  - How fast? Constant time (independent of the element size!)
  - Common operations:
    - search, insert, delete and hash

Hashing: decide the positions

- E.g. A set of keys: 1, 9, 2, 3, 10

Index:	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
T:	1, 2, 3 9, 10

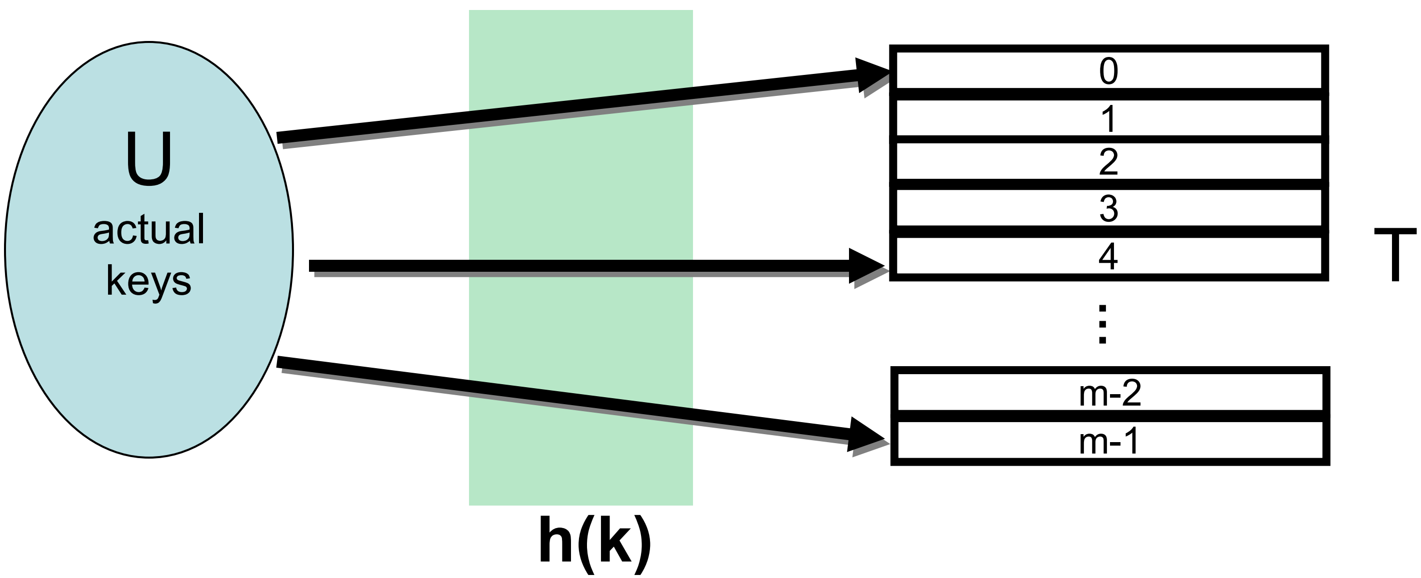
Direct-addressing table: an element with key k is stored at location k

Search(T, k) = O(1)
Insertion(T, k) = O(1)
Deletion(T, k) = O(1)

Problem: The range of keys can be large…2^{64}

Hashing

- Store an element with key k in slot h(k)
- **h(k)**: maps the universe U of keys into the slots of a hash table.



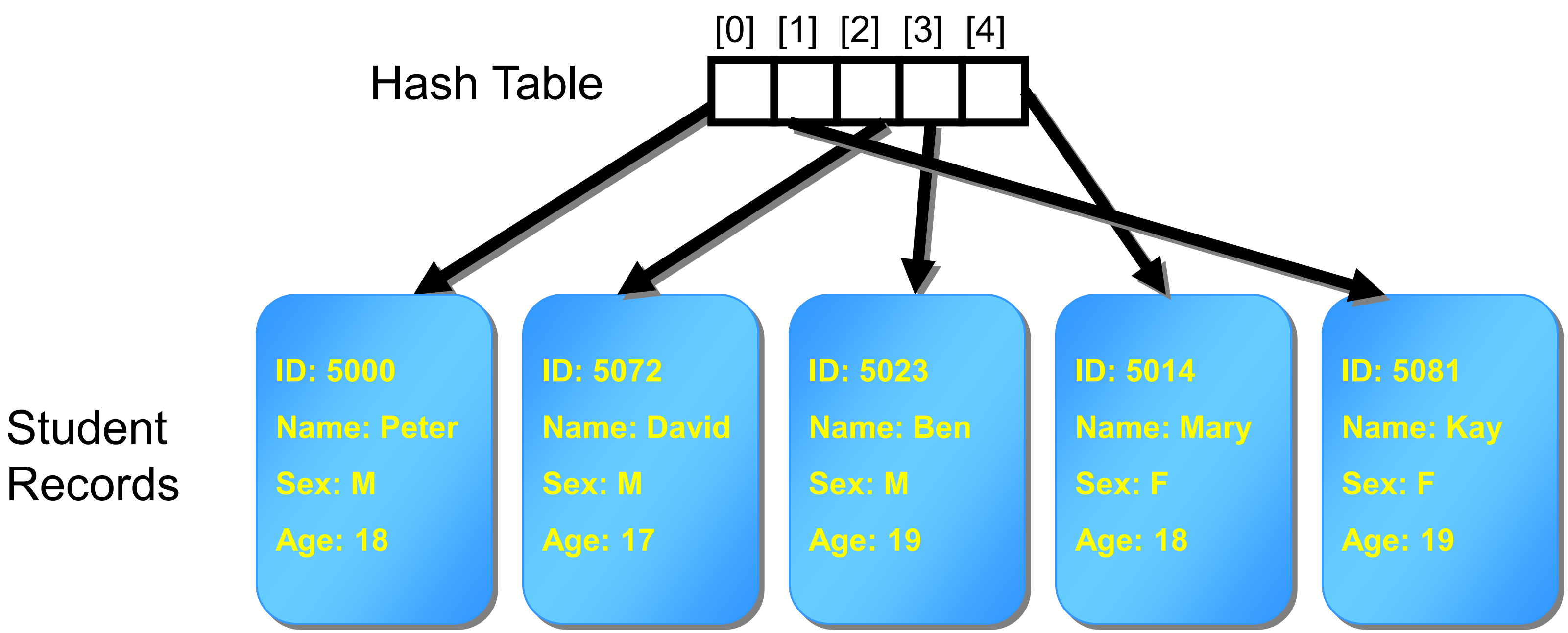
*: **h(k)** is called the hash function.

Hash Function

- A hash function is a well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small range of index (positions)
- The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes
- The hash value is usually a single integer that may serve as an index to an array

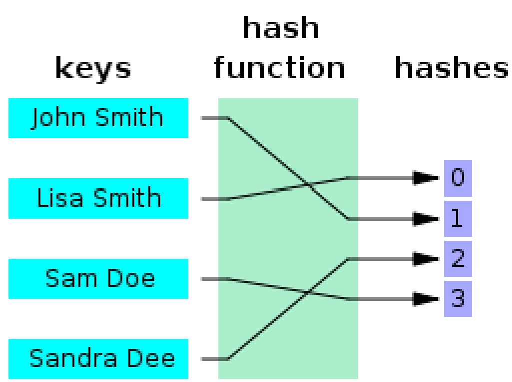
A Simple Hash Function

- To enhance the memory utilization of the previous example, we can apply the following hash function to the key (Student ID) of the records:
  - $h(k) = k \% 5$

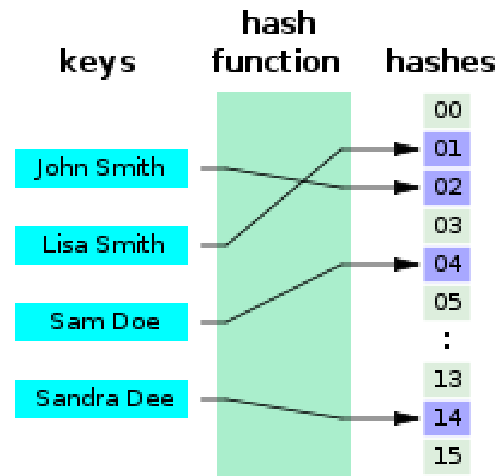


The Hashing Approach

- By using the new hash values to index the records, we can reduce the array size to 5. A hash function maps each valid input to a different hash value is said to be **perfect**
  - With such a perfect hash function one can directly locate the desired entry in a hash table, without any additional searching
- A hash function for *n* keys is said to be **minimal** if it outputs *n* consecutive hash values



Minimal Perfect Hashing



Perfect Hashing

Hash Collision

- E.g. A set of keys: 1, 7, 6, 4, 5, 9
  - $h(k) = k \% 10$

0	1	2	3	4	5	6	7	8	9
	1			4	5	6	7		9

  - h(1) = 1 % 10 = 1, h(4) = 4 % 10 = 4, h(5) = 5 % 10 = 5
    h(6) = 6 % 10 = 6, h(7) = 7 % 10 = 7, h(9) = 9 % 10 = 9
  - If we have a key = 44: h(44) = 44 % 10 = 4
  - Then we have h(44)==h(4), and this is a hash collision.

Hash Collision

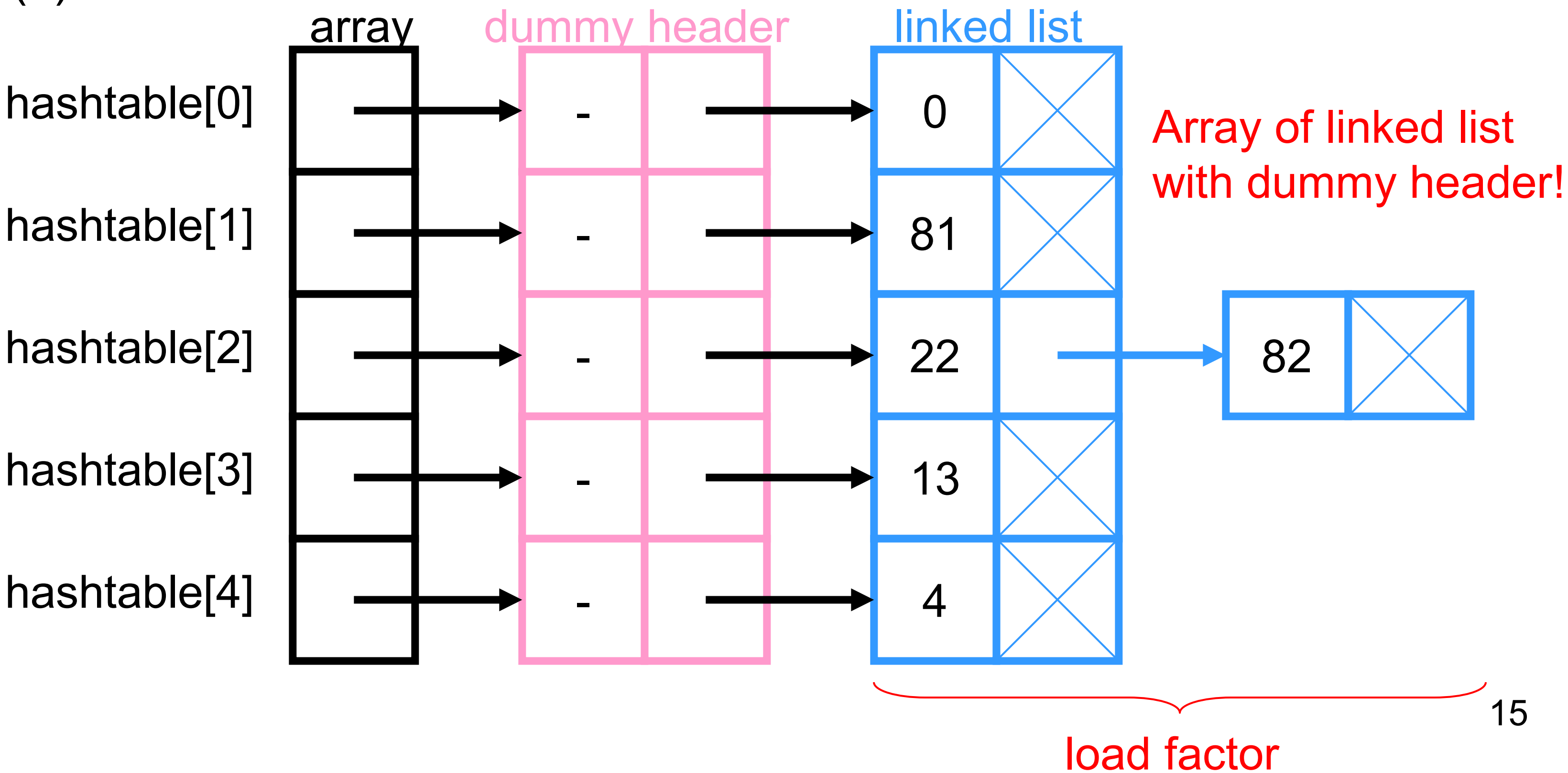
- A collision is a situation that occurs when two distinct pieces of data have the same hash value
- Collisions are unavoidable whenever members of a very large set (such as all possible person names, or all possible computer files) are mapped to a relatively short bit string
- Two types of collision resolution:
  - Closing addressing
    - Chaining buckets
  - Opening addressing
    - Linear probing, Quadratic probing, Double hashing
- Any collision in a hash table increases the average cost of lookup operations

Closing Addressing

Place the key in the same slot
even when collisions has occurred

Chaining Buckets

- Example: store 0, 4, 13, 22, 81 and 82 into the hash table using the chaining buckets
- For simplicity, let the key be the same as the element
- $h(k) = k \% 5$

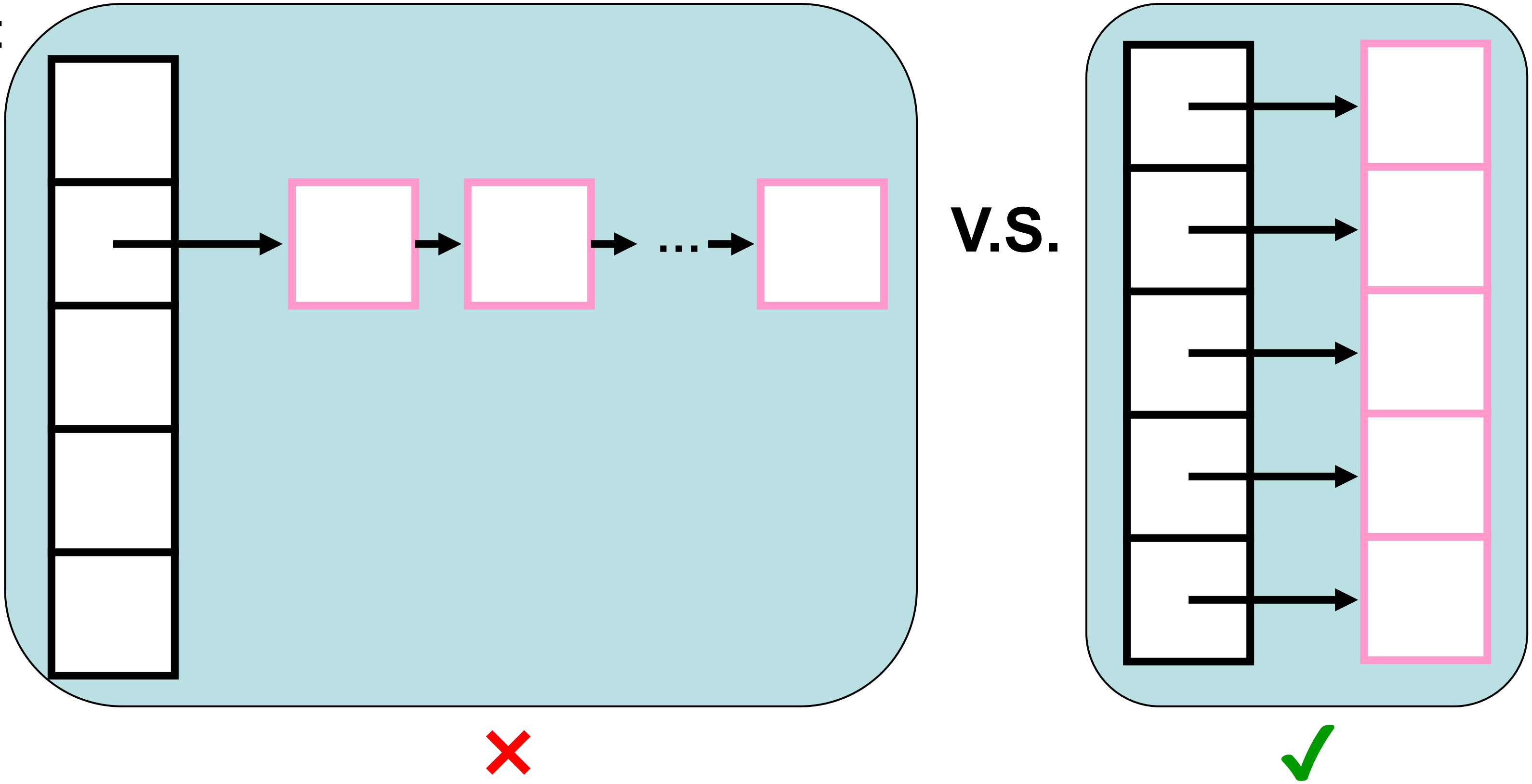


Chaining Buckets

- CHAINED-HASH-INSERT (T, x) // x: key, T: hash table
  - Insert x into the head of list T(h(x)):
    O(1)
- CHAINED-HASH-SEARCH (T, x)
  - Search for an element with key x in list T(h(x)):
    O(|T(hx)|) // size of the chain at h(x)
- CHAINED-HASH-DELETION (T, x)
  - Delete an element with key x in list T(h(x)):
    O(|T(hx)|)

Chaining Buckets

- T:



- Average case analysis of chaining
  - Assumption: Simple uniform hashing
  - Each key k ∈ U is equally likely to be hashed to any slot of table T, independent of where other keys are hashed
  - Probability h(k1) = h(k2): 1/m (m: the size of T)

Chaining Buckets

- Let n be the number of keys, and m be the size of T, define the load factor of T to be

$$\alpha = \frac{n}{m} = \textbf{ average number of keys per slot}$$

- Search cost
  - The expected time for an unsuccessful search is $\underbrace{\theta(1}_{\text{Hash function}}+\underbrace{\alpha)}_{\text{Search for the list}}$
  - Expected search time = $\theta(1)$ if $\alpha = O(1)$

Opening Addressing

Place the key in other free slot
when collisions has occurred

Opening Addressing (V.S. chaining)

- No storage is used outside of the hash table itself
- Insertion systematically probes the table until an empty slot is found
- The hash function depends on both the **key** and the **probe number**

$$h: U \times \{0, 1, \ldots, m-1\} \longrightarrow \{0, 1, 2, \ldots, m-1\} \quad \text{// m is the hash table size}$$

- The probe sequence $\langle h(k, 0), h(k, 1), h(k, 2), \ldots, h(k, m-1) \rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$

Linear Probing

- Place the key in the next free slot when collisions has occurred (i.e. sequentially search the hash table for a free location)
  - $h(k, i) = (h(k) + i) \% n$
  - where i is the step size, n is the table size and $h(k)$ is the original hash function
- If the slot of $h(k)$ mod n has been used, try
  - $(h(k) + 1) \% n$
- If unlucky that the new slot has also been used, try
  - $(h(k) + 2) \% n$
- And so on until a free slot has been found

Linear Probing – Eg-1

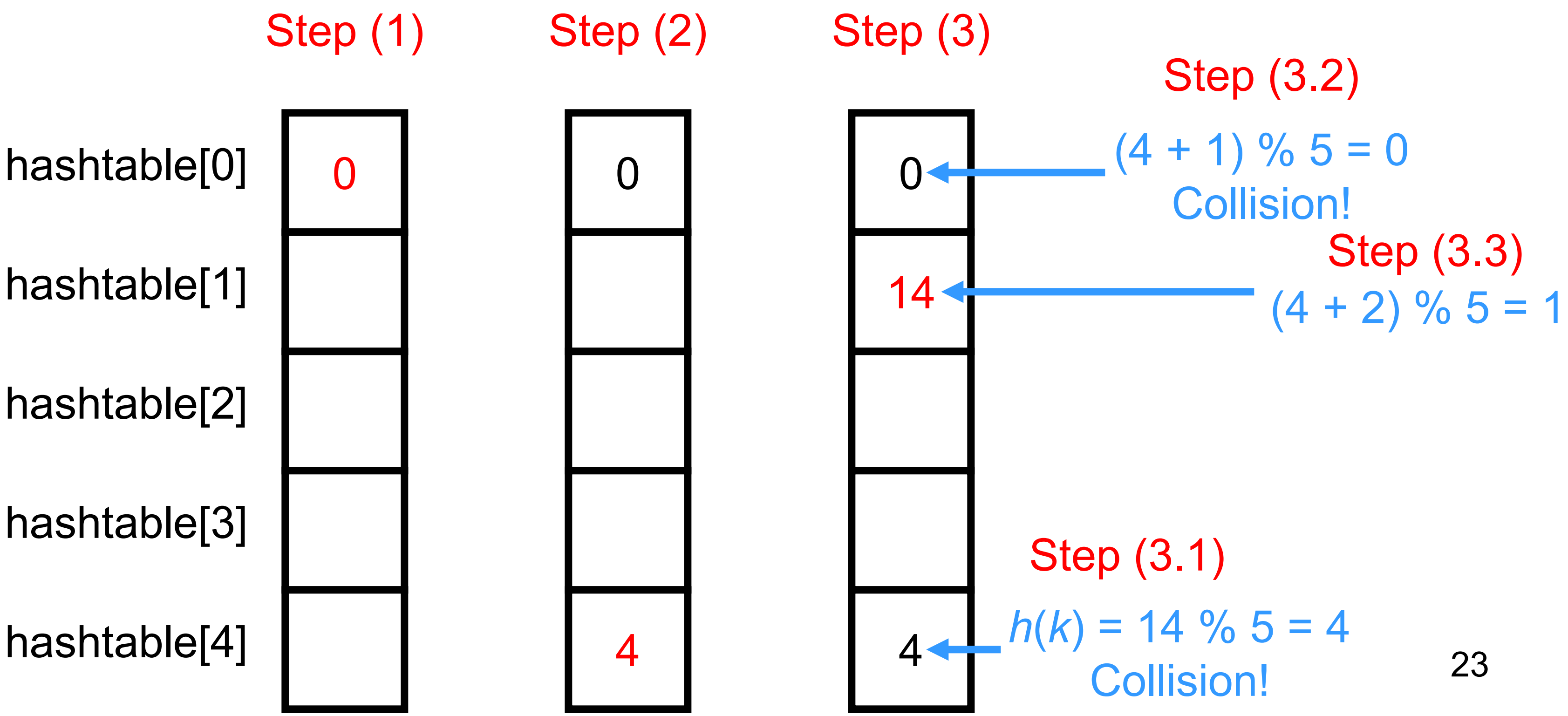
- Given an ordinary hash function $h'(k)$, linear probing was the hash function
  - $h(k, i) = (h'(k) + i) \% n$

- If $h'(k) = k \% 11$, $h(k, i) = ((k \% 11) + i) \% 11$
  - Insert 15
    - $k=15$, $h'(15) = 15 \% 11 = 4$, $h(15, 0) = 4$
  - Insert 4
    - $k=4$, $h'(4) = 4 \% 11 = 4$, $h(k, i) = h(4, 0) = 4$
    - $h(k, i+1) = h(4, 1) = (4+1) \% 11 = 5$
  - Insert 16
    - $k=16$, $h'(16) = 16 \% 11 = 5$, $h(16, 0) = (5+0) \% 11 = 5$
    - $h(16, 1) = (5+1) \% 11 = 6$

0 ⋮	
4	15
5	4
6	16
	⋮

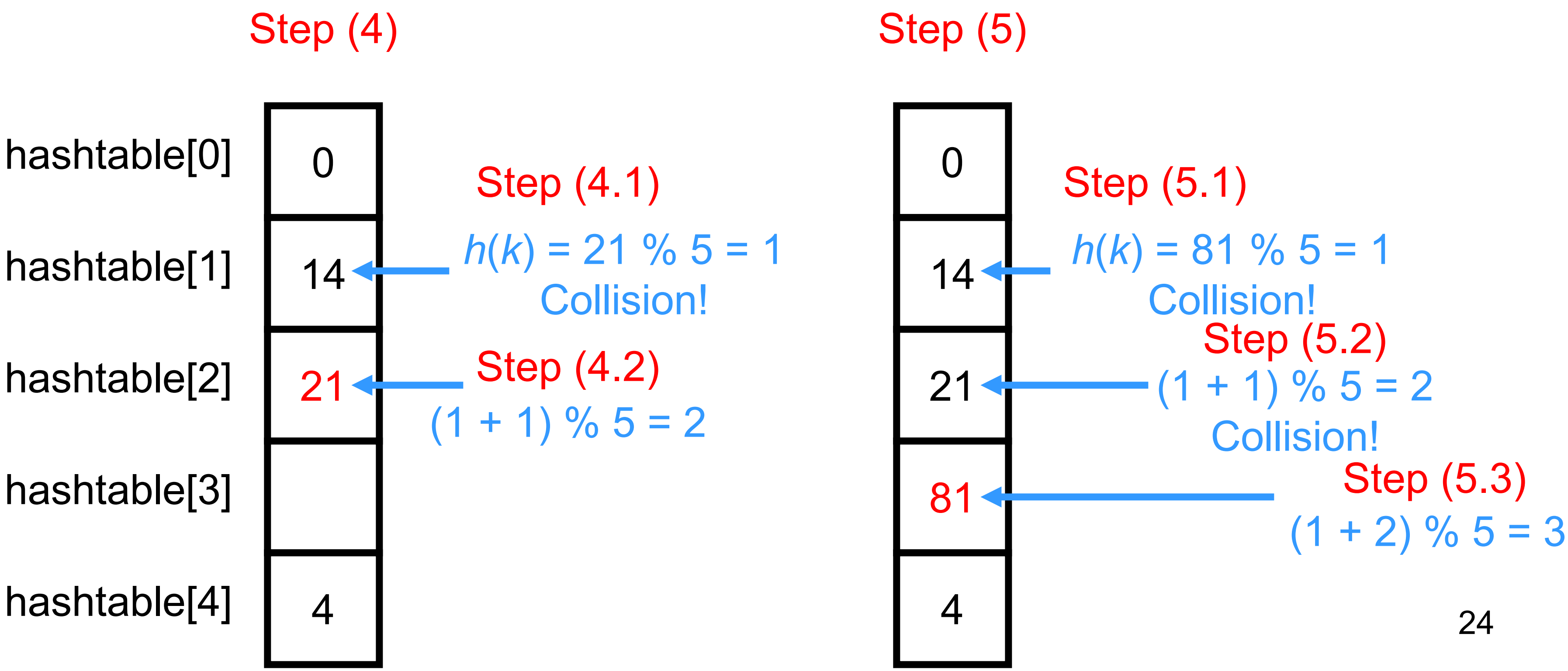
Linear Probing – Eg-2

- Store 0, 4, 14, 21 and 81 into hash table using linear probing. Let $h(k) = k \% 5$



Linear Probing – Eg-2

- Store 0, 4, 14, 21 and 81 into hash table using linear probing. Let $h(k) = k \% 5$



Exercise

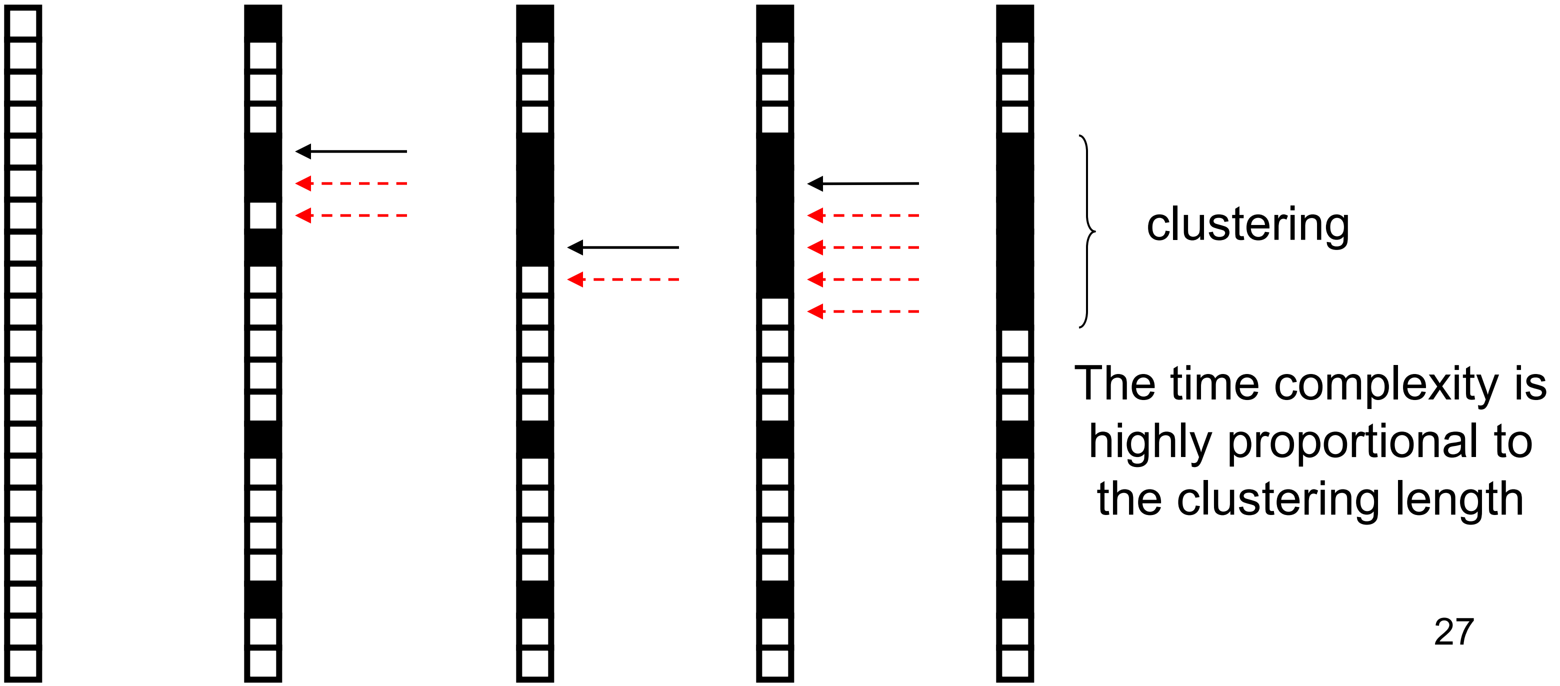
- Show the final hash table for input 100, 20, 28, 31, 33, 80, 98, 42 using chaining and linear probing, respectively. h(i)=i % 10. The hash table size is 10.

Analysis

- The 1st element, key 0, located in its home position
- The 2nd element, key 4, also located in its home position
- The 3rd element, key 14, tried 3 positions before finding an empty slot
- The 4th element, key 21, tried 2 positions
- The last element, key 81, tried 3 positions
- The **total number of comparisons** required to search for all these 5 entities is
  - 1 + 1 + 3 + 2 + 3 = 10
- **Average number of comparisons for a successful search**
  - = 10 / 5 = 2

Another Problem of Linear Probing

- Overflow addresses tends to group in a region of the array
- Called **clustering**

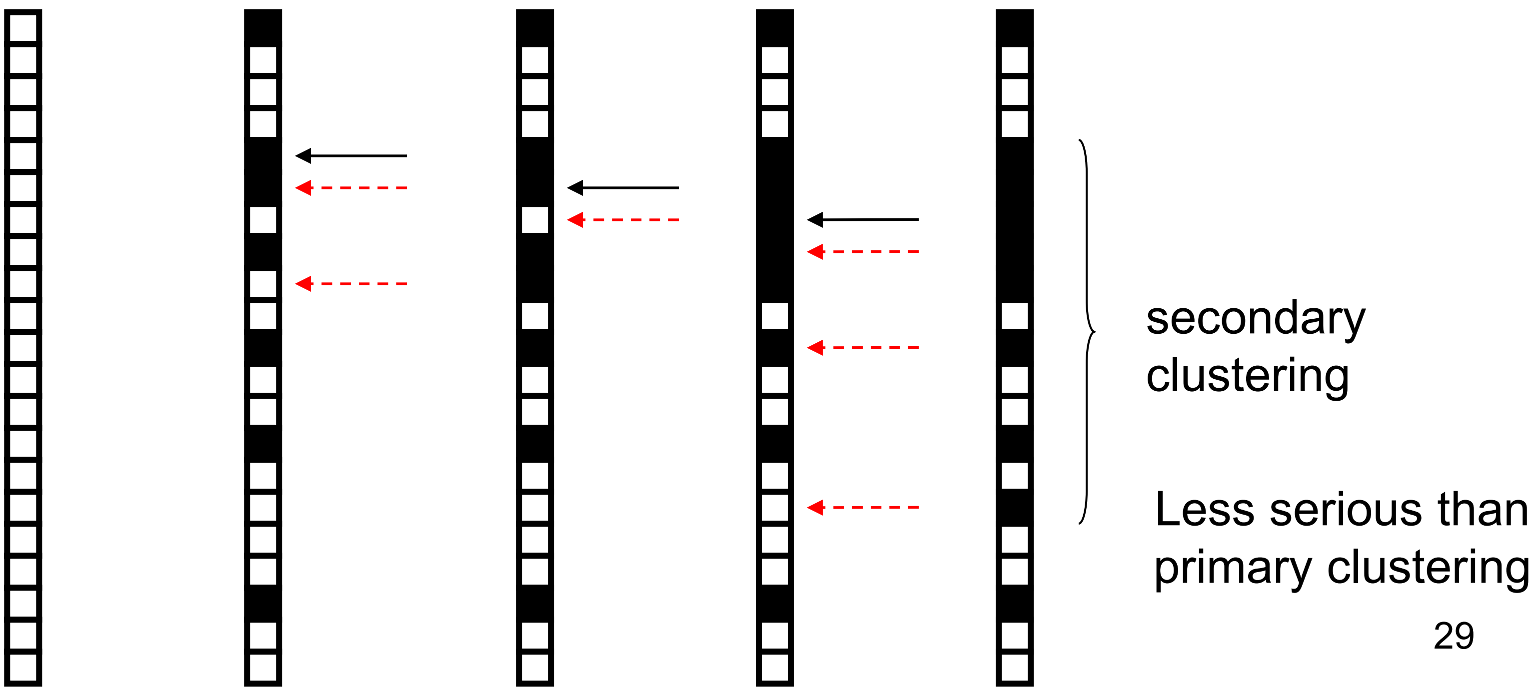


Solution of Clustering

- Instead of using linear probing, try **Quadratic Probing**
  - $h(k, i) = (h(k) + i^2) \% n$
  - where i is the step size, n is the table size and $h(k)$ is the original hash function
- So the try sequence is
  - $h(k) \% n$
  - $(h(k) + 1^2) \% n$ Jump 1 slot
  - $(h(k) + 2^2) \% n$ Jump 3 slots more
  - $(h(k) + 3^2) \% n$ Jump 5 slots more again
  - And so on until a free slot is found
- To "jump" away from clustering

Problem of Quadratic Probing

- Quadratic probing eliminate primary clustering
- But produce secondary clustering



To Avoid Clustering

- **Double hashing**: design 2 independent hash functions h1() and h2()
  - $h(k, i) = (h1(k) + i * h2(k))$ % n
  - where i is the step size and n is the table size
- So the try sequence is
  - $h1(k)$ % n
  - $(h1(k) + h2(k))$ % n
  - $(h1(k) + 2 * h2(k))$ % n
  - $(h1(k) + 3 * h2(k))$ % n
  - And so on until a free slot has been found
- The jump interval is decided using a second, independent hash function. So values mapping to the same location have different jump sequences
- This minimizes repeated collisions and the effects of clustering
- The trade off: cost more time to compute new hash value

Double hashing

- **Double hashing**: Given ordinary hash function, *h1*(*k*) and *h2(k)*, double hash is
  - $h(k, i) = (h1(k) + i * h2(k)) \% m$ // *m is the table size*
  - If $h1(k) = k \% 13$, $h2(k) = 1 + (k \% 11)$
  - Insert 14:
    - $h1(14) = 14 \% 13 = 1$, $h2(14) = 1+ (14 \% 11) = 4$
    - $h(14, 0) = (h1(14) + 0 * h2(14)) \% 13 = 1 \% 13 =1$
    - $h(14, 1) = (h1(14) + 1 * h2(14)) \% 13 = (1 + 4) \% 13 =5$
  - Delete 72:
    - $h(72, 0) = 7$
  - Delete 98:
    - $h(98, 0) = 7, h(98, 1) = ?, \ldots, h(98, 2), \ldots, h(98, 12)$

T(m = 13)

0	
1	79
2	
3	
4	69
5	14
6	
7	~~72~~
8	
9	9
10	50
11	
12	

  - A hash table of size m is used to store n items with n ≤ m/2, opening address is used for collision resolution. Assume uniform hashing, show that for i = 1, 2, …, n, the probability that the ith insertion requires strictly more than k probs is at most 2^{-k}

Exercise

- Store 90 100 89 88 67 30 23 33 in a hash table T of size 10 using double hashing.
  - h1(i) = i % 10, h2(i)=1+i%11
  - $h(k, i) = (h1(k) + i * h2(k)) \% m$

Design of Hash Function

Division Method
Mid-Square
Folding Method

Design of Hash Function

- An ideal hash function should have the following properties
  - Low Cost
    - Easy and fast to compute
  - Variable Range
    - Able to transform words, symbols into numbers
  - Uniformity
    - Distributes the keys evenly
    - Minimize the chance of collisions

1) Division Method

- Easy to implement and fast to compute
  - **Division**: key % tablesize
  - Use **prime number** as the hash table size to reduce collisions
- How about the key is not an integer?
  - Transform into integer

```
int hash(char key[SIZE]) {
    int i = SIZE, k = 0;
    while (i-- > 0)
        k += key[i];           //sum the ASCII values
        //or k ^= key[i];      //XOR the ASCII values
    return k % tablesize;      //return the hash value
}
```

2) Mid-Square

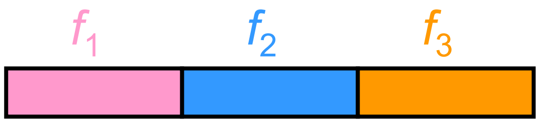
- Step 1. Transform to integer
- Step 2. Square the number
- Step 3. Select some digits from the middle

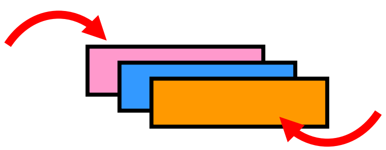
- E.g. Put these keys into a table of size 5

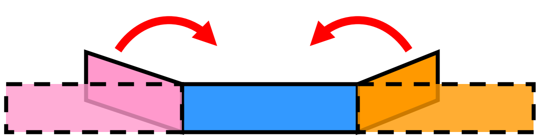
key	**key^2**	**hash(key)**
281	78961	96 mod 5 = 1
99	9801	80 mod 5 = 0
123	15129	12 mod 5 = 2

3) Folding Method

- Step 1. Split the key into several parts
- Step 2. Sum the folded the key
- E.g. key = 245769908
  - Shift-folding
    - Define f_1 = 245, f_2 = 769, f_3 = 908
    - $h(k) = (f_1 + f_2 + f_3)$ mod size
  - Boundary-folding
    - Define f_1 = 542, f_2 = 769, f_3 = 809
    - $h(k) = (f_1 + f_2 + f_3)$ mod size







Design of Hash Function

- Examples: A company with 100 employers. The access key is a nine-digit number (SSN). The hash table is indexed from 0 to 99.
  - $h(d_1 d_2 \ldots d_9) = (d_4 d_5)$ % 100 X
  - $h(d_1 d_2 \ldots d_9) = (d_1 \ldots d_9)$ % 100 X
  - $h(d_1 d_2 \ldots d_9) = \sum_{i=1}^{9} d_i$ % 100 Better

Applications

- Hash functions are mostly used to speed up table lookup or data comparison tasks such as finding items in a database, detecting duplicated or similar records in a large file
- Determine if there are any duplicated numbers from the following sequence of numbers:
  - {52, 61, 18, 70, 39, 48, 28, 57, 61, 39, 43}
  - 61 and 39 repeated twice
- Can you suggest an algorithm to find the duplicated numbers?

Applications

- Pattern matching / string search. For a string sub, test whether it is a substring of another (longer) string S.
  - |sub| = m, |S| = n
  - E.g. sub = “ACGT”, h(ACGT) == h(ACGT)

S: —AGGT——ACGT——ACGT—

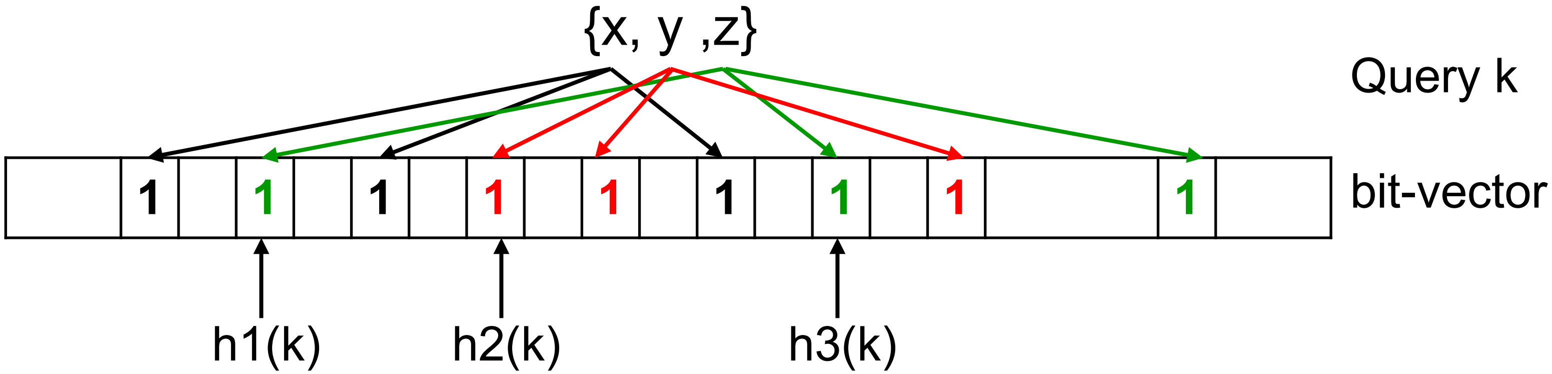
50 150

Algorithm 1 Use of hashing for substring search function RabinKarp(String S[1, . . . , n], String sub[1, . . . , m])

hsub = hash(sub[1, ..., m])
hs = hash(S[1, ..., n])
for $i = 1$ to $n - m + 1$ **do**
  if $hs == hsub$ **then**
    if $S[i, ..., i + m - 1] == sub$ **then** ▷ String comparison
      return i
    hs = hash(S[i+1, ..., i+m)
return not found

Applications

- Bloom-filter "set membership detection"
  - Burton. H. Bloom, 1970
  - A space-efficient data structure that is used to test whether an element is a member of a set



E.g. 3 hash functions. Allow false positives, not false negatives.
Probably "yes", definitely "no"!