

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 **index** in a book?
- To help you to search the pages (that containing the keyword) quicker
- How about if the book does not have any 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?
 - Radix sort $O(kn)$
 - How do you search?
 - Binary search $O(\log n)$

Student
Records

ID: 5000

Name: Peter

Sex: M

Age: 18

ID: 5072

Name: David

Sex: M

Age: 17

ID: 5023

Name: Ben

Sex: M

Age: 19

ID: 5014

Name: Mary

Sex: F

Age: 18

ID: 5081

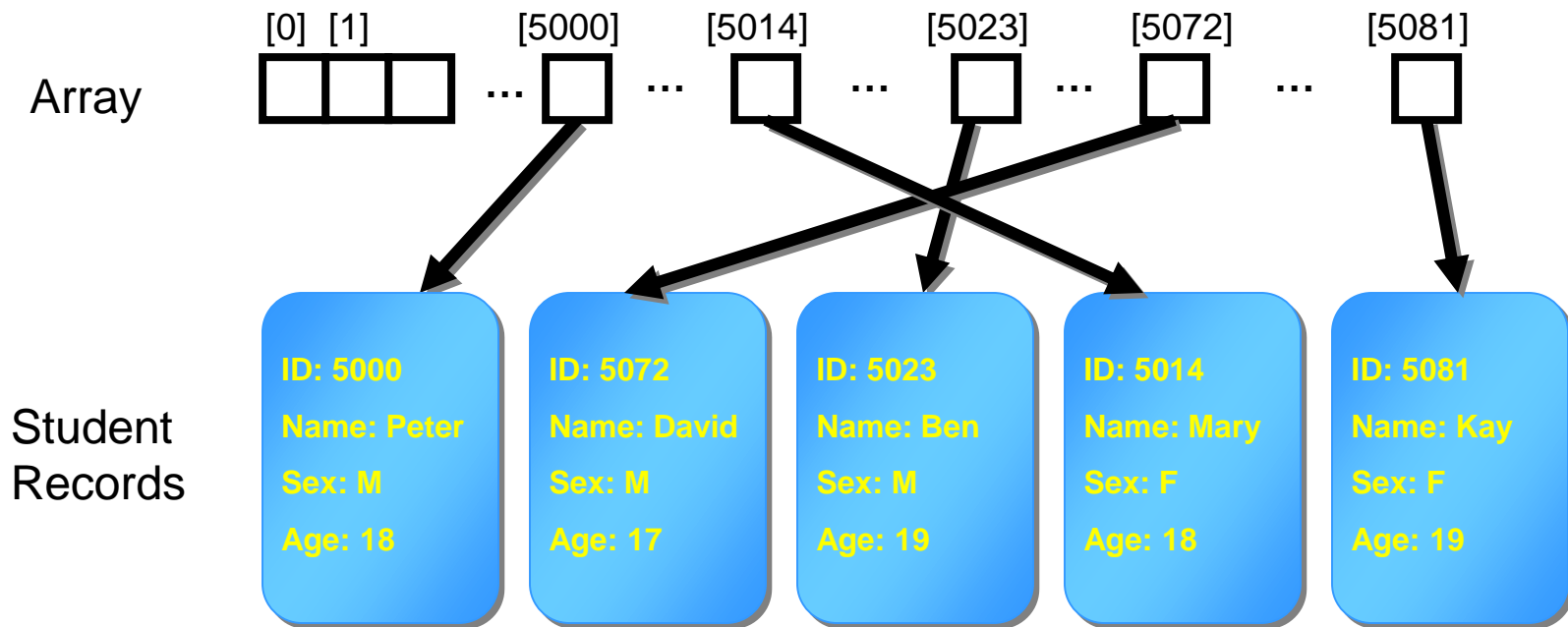
Name: Kay

Sex: F

Age: 19

Indexing Data Record

- Using an array to hold pointers to the records
 - Use Student ID to index the records
 - What is the time complexity now?
 - But waste too much space...



Hashing

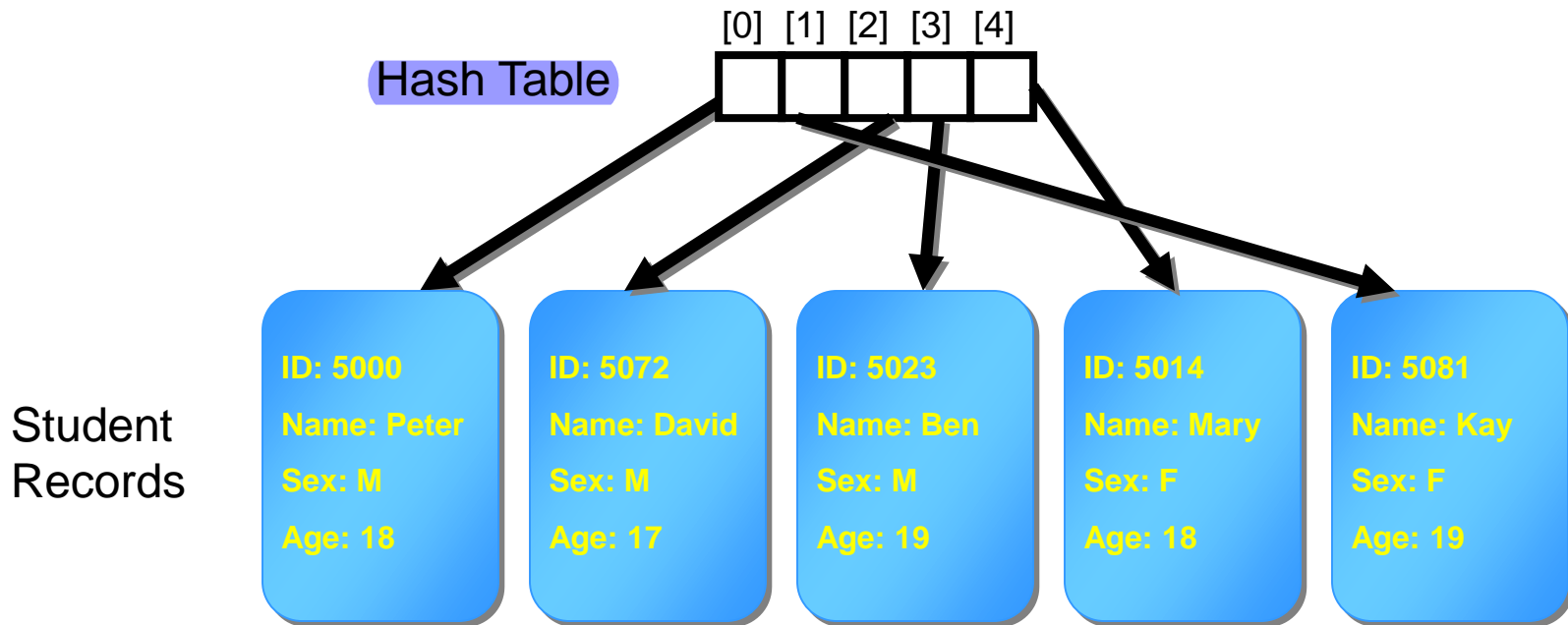
- The term “hash” means to chop and mix!
- The objectives
 - Build an index for a set of elements/records
 - To allow **fast search** (also **insert**, **delete**) operations
 - How fast? **Constant time** (independent of the element size!)
 - Common operations:
 - search, insert, delete and **hash**

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 datum**
- 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

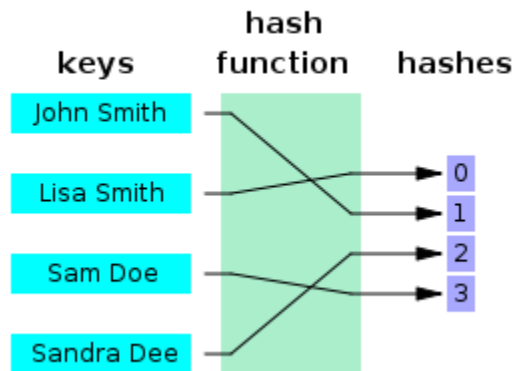
A Simple Hash Function

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

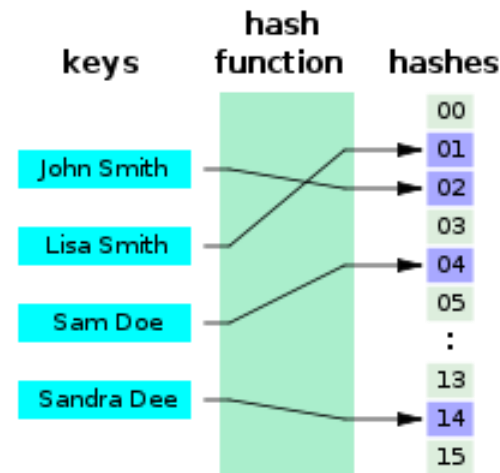


The Hashing Approach

- 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

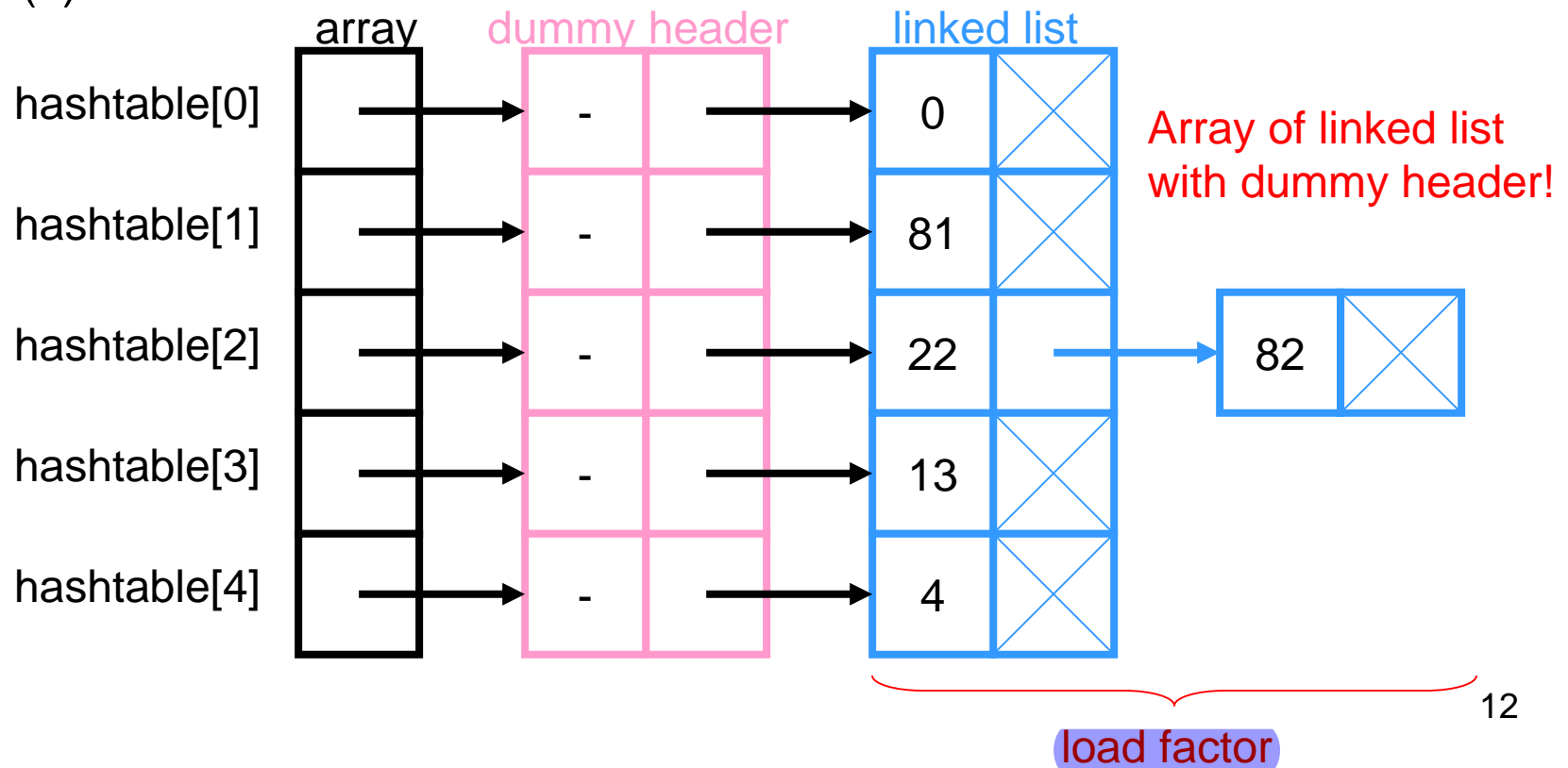
- A collision is a situation that occurs when two distinct pieces of data have the same hash value
- Collisions are unavoidable whenever members of a very large set (such as all possible person names, or all possible computer files) are mapped to a relatively short bit string
- Two types of collision 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
- $h(k) = k \% 5$



Opening Addressing

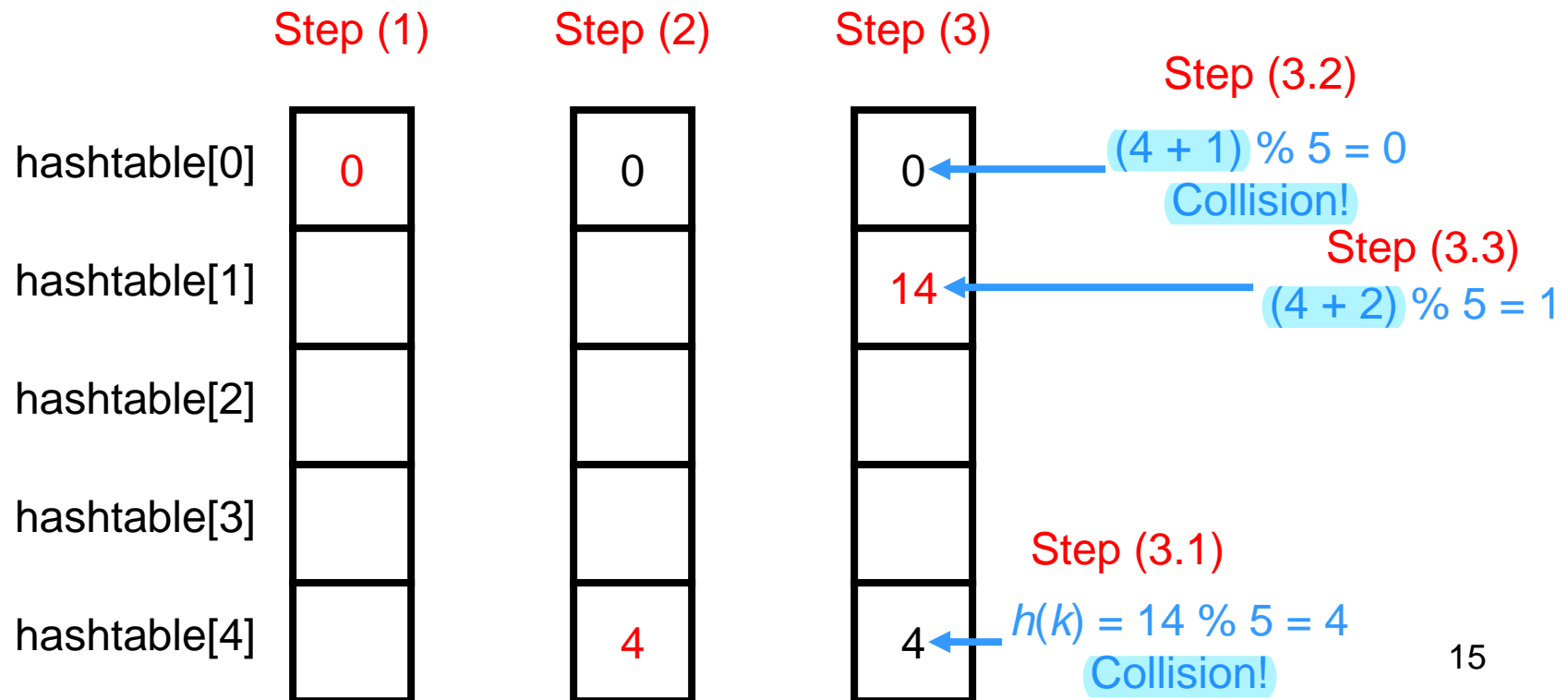
Place the key in other free slot
when collisions has occurred

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 $h(k) \bmod n$ 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

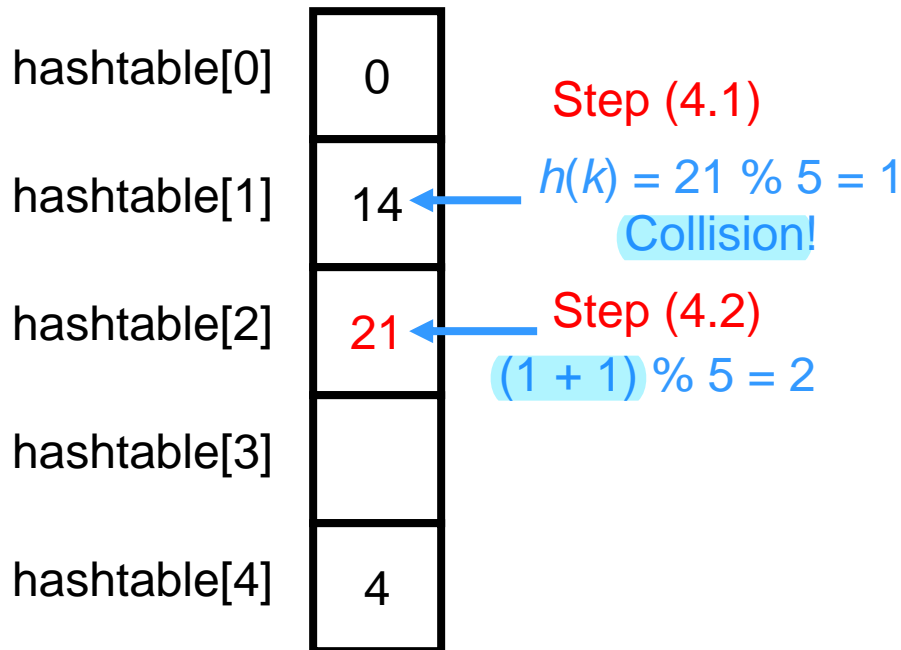
- Store 0, 4, 14, 21 and 81 into hash table using linear probing. Let $h(k) = k \% 5$



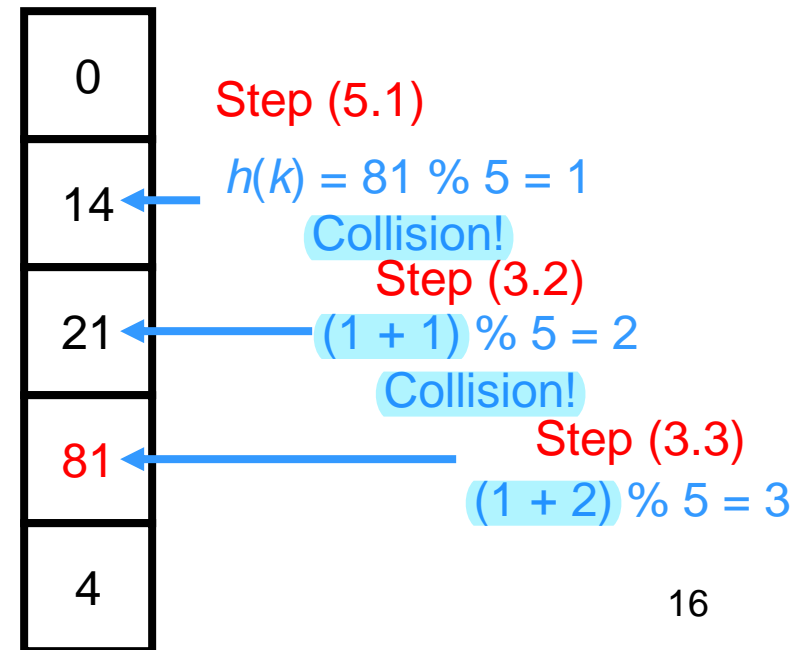
Linear Probing

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

Step (4)



Step (5)

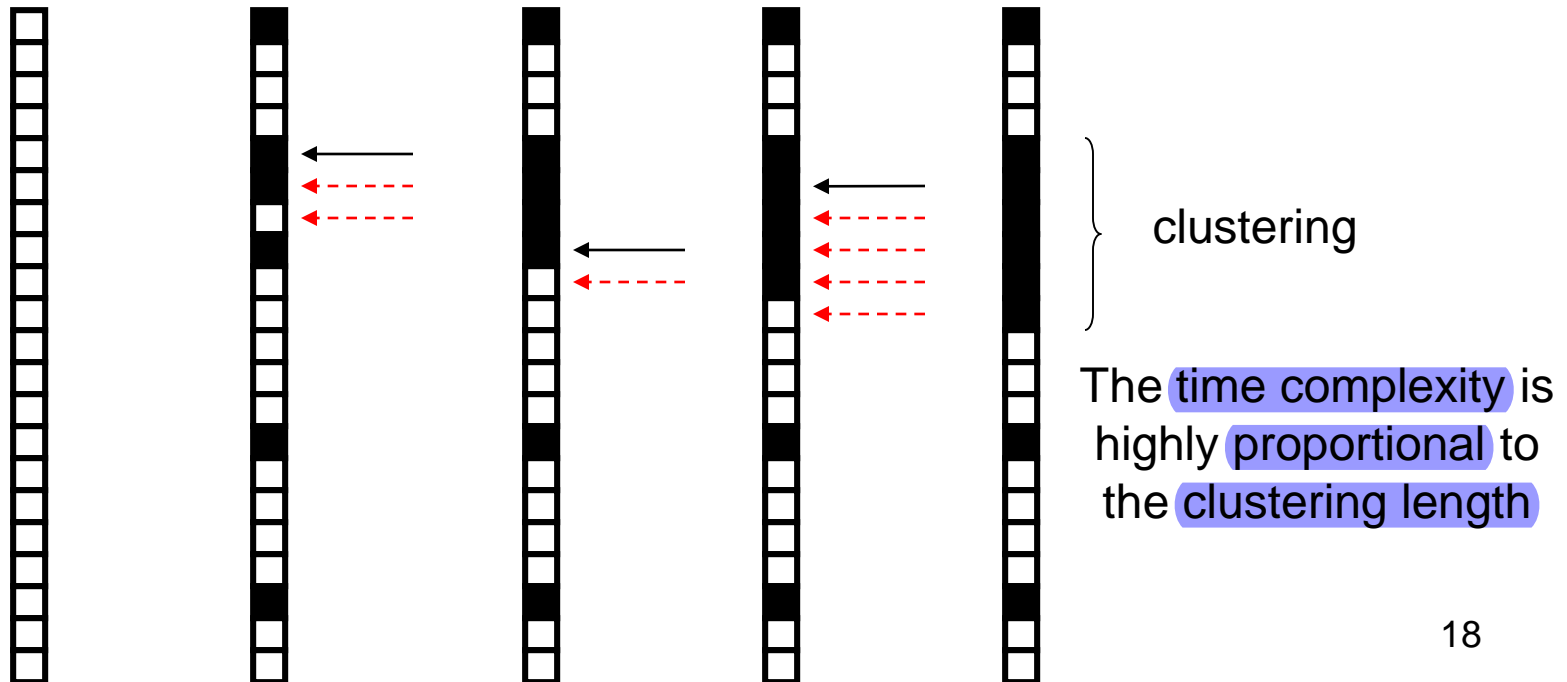


Analysis

- 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**

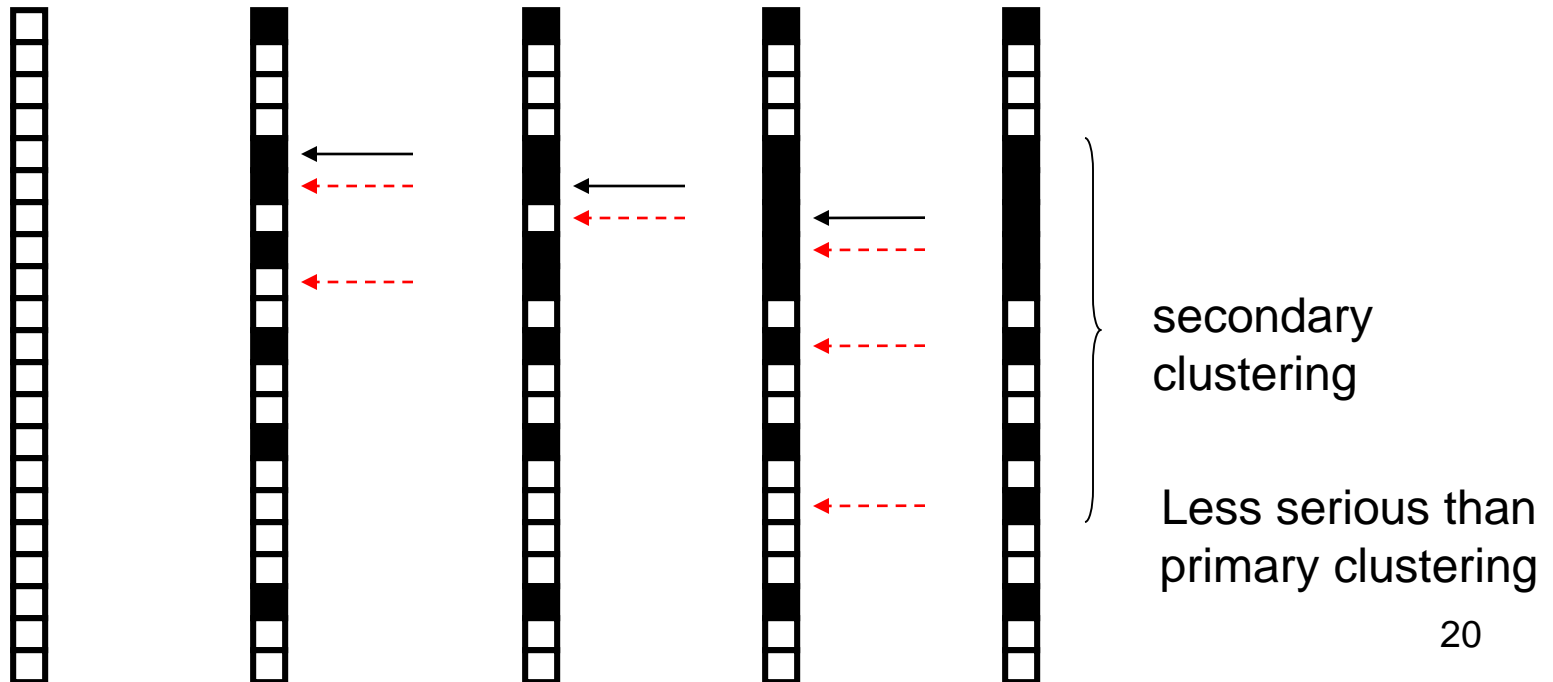


Solution of Clustering

- 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

Design of Hash Function

Division Method
Mid-Square
Folding Method
Radix Transform

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:** $\text{key} \% \text{tablesize}$
 - 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  
}
```

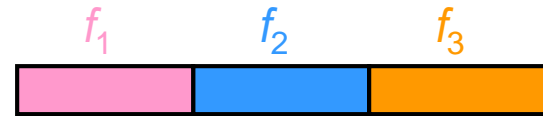

2) Mid-Square

- 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

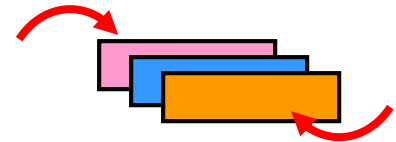
3) Folding Method

- Step 1. Split the key into several parts
- Step 2. Sum the folded the key
- E.g. key = 245769908



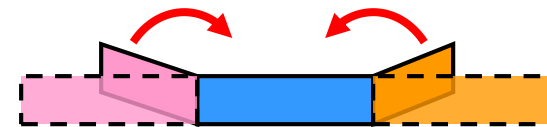
- Shift-folding

- Define $f_1 = 245$, $f_2 = 769$, $f_3 = 908$
- $h(k) = (f_1 + f_2 + f_3) \bmod \text{size}$



- Boundary-folding

- Define $f_1 = 542$, $f_2 = 769$, $f_3 = 809$
- $h(k) = (f_1 + f_2 + f_3) \bmod \text{size}$



4) Radix Transformation

- Generate the hash value by transforming the key using new radix
- e.g. key = $358345_{(10)}$
 - Define $k_2 = 358345_{(9)}$
 - $h(k) = k_2 \bmod \text{size}$
 - $k_2 = 358345_{(9)} = 216068_{(10)}$
 - $h(k) = 216068_{(10)} \bmod \text{size}$

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?