



**Maynooth  
University**

National University  
of Ireland Maynooth

# **CS210**

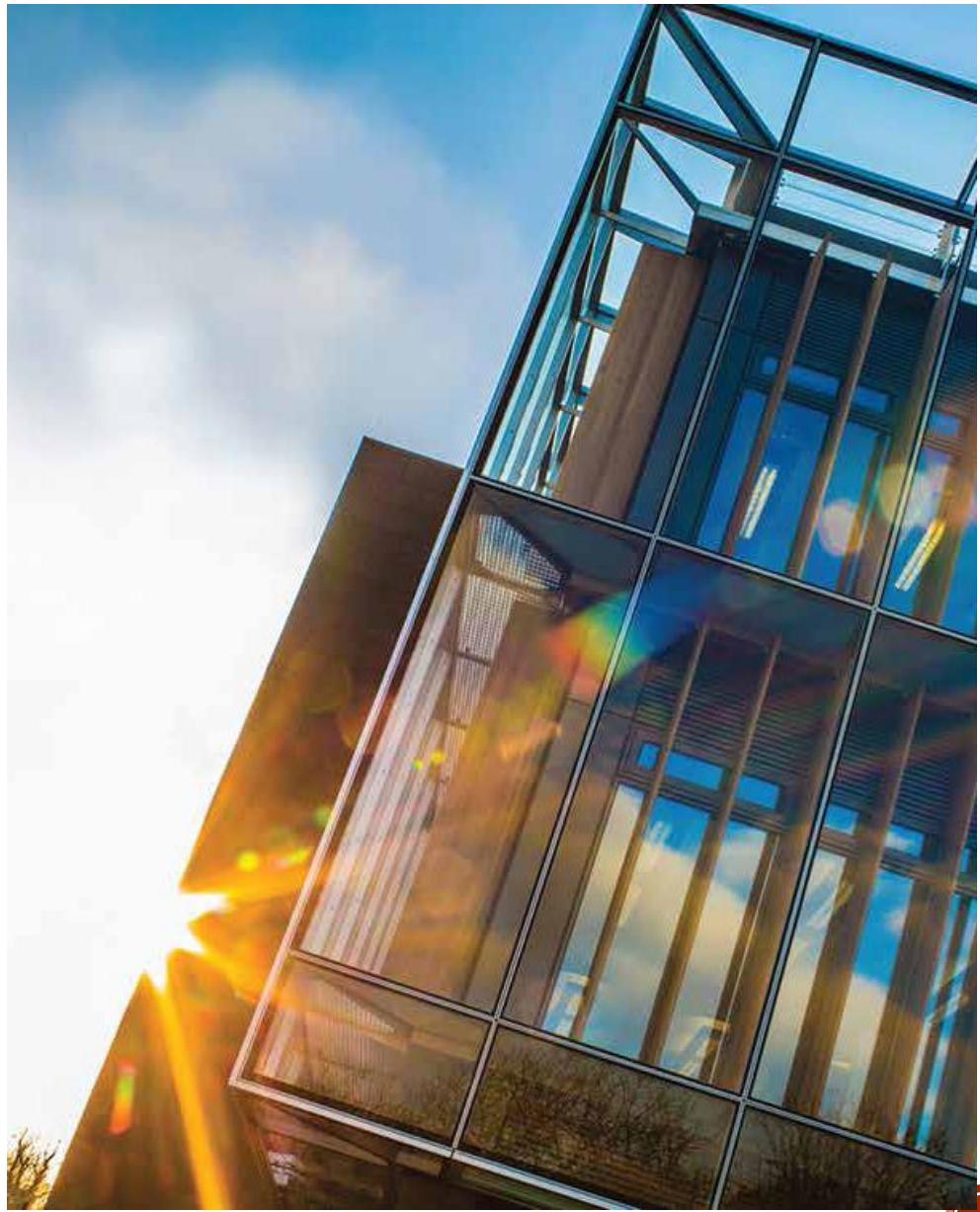
## **Algorithms and Data Structures**

**Hash Tables**

**Week 8**

**Semester 1 2025**

**Dr. Pierpaolo Dondio**



# Week 7

- *Hash Function, Index and Tables: definition and functioning*
- *Closed Addressing Hash Table*
- *Open address Hash Table*
- *Dynamic Hashing*
- *Built-in Java classes*
- *Examples in Java*

# What is an index

- *An additional data structure used to access data in a more efficient way*
- *It contains “keys” that are the location to access the associated value*
- *The index is small and kept sorted*
- *The values can be in a large unsorted data structure*
- *In order to access a value, first we access the index, we get the key, and we access the value/data we were looking for*
- *Therefore, we need to perform two accesses, one for the index, one for the actual data we are searching*

## INDEX

### A

abbreviations  
dates, 50–60  
time, 61  
uses of, 139–142  
absolute phrases, 81  
acronyms, 140, 141–142  
action verbs, 95–96  
active voice, 91–92  
adverbs  
comparative, 159–161  
compound, 162  
defined, 37, 158  
demonstrative, 151–152  
indirect, 162–163  
non-experiencer, 165  
order of, 158  
possessive, 147–148  
proper, 161  
superlative, 159–161  
adverbs  
clustering, 204–205  
clauses, 165  
conjunctive, 166  
defined, 37, 165  
phrases, 165  
sentence, 167  
affixes, 49–52

a.m. (*ante meridiem*), 61  
apostrophe  
contractions, 53–55  
in dates, 60  
and plural suffixes, 128, 131,  
141–142  
possessives, 55–56, 116–117  
appositive phrases, 80  
articles, 37, 121–123  
aspect, 86  
attributive nouns, 114, 117  
audience, 22, 42–43  
auxiliary verbs, 97–99  
*See also modal verbs*

### B

body, paragraph, 26–27  
bracketed ellipsis, 200  
British English, 42–44, 104, 211,  
214–215

### C

capitalization, 132–135, 161, 198,  
206–207  
cardinal numbers, 65–66  
clauses  
adverb, 165

230



# What is an index /2

- *An index can be implemented as a 2D array*
- *For each key, it stores the location of a chunk of data*
- *Index and Data can be two different files, or two tables in a database. It is an abstract data structure*

Index		Data					
ST ID (key)	Location	ST ID	Fname	Lname	age	Address	Tel
101	3	351	John	Fsdfsdf	32	S fsfas f ds fa	1234
221	1	2221	Mary	Dsgds	21	Saf dsaf ds f	35345
351	0	484	Bob	Dfgdfs	22	fdsaf	6445
484	2	101	Carol	Ggdgs	21	gd gsd	2423
510	4	510	Sean	dsgds	19	sd gds	654645

# Complexity of searching by Index

- *If the index is sorted, we can perform a binary search*

$O(\log n)$

- *Once the index is found, we can access the data file directly*

$O(1)$

- *In total, complexity is  $O(\log n)$*
- *Without an index, linear search  $O(n)$*

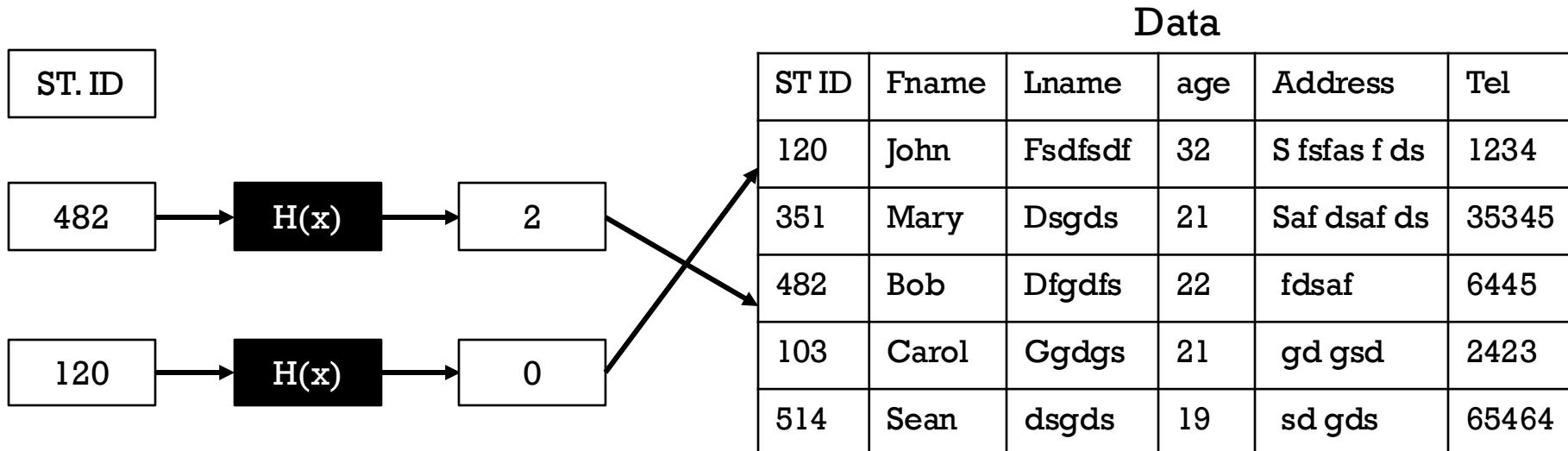
# Hash Table

- A hash table is a data structure that implements an associative array, also called a dictionary or simply map;
- An associative array is an index, it is an abstract data type that maps keys to values.
- A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.
- On average, it can reduce the time to access a data item to  $O(1)$ , while the worst case is  $O(n)$
- The more data are inserted, the higher is the chance that the hash table index will degenerate towards a linear search (worst case)

# Hash Function

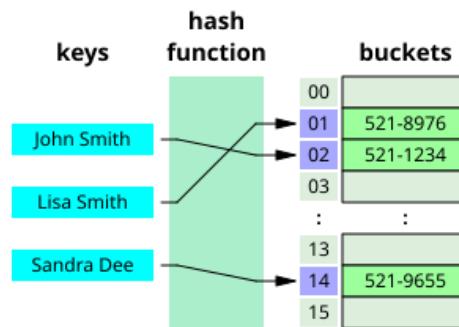
- Hash table are based on the notion of Hash Function  $H(x)$
- $H(x)$  transforms (map) the input value into a pre-defined range of values that are used to build an index to access the data
- Example:

$H(X)$  = “last digit of a number”  $[x \bmod 10]$



# Hash Table Terminology

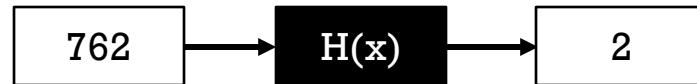
- In a Hash Table the input to the hash function is called a “key”, the hash function will map the key into a “hashed” value that is the location of a “bucket”, the place where the data are stored



- The keys can be anything, numbers , strings, objects... the hash function will map them into indexes to be used to access the right bucket
- Time complexity. After the hash function is computed in constant time  $O(1)$ , we can access the right bucket containing the right data in  $O(1)$
- All fine, until **collisions** start to happen!

# Collisions

- What if the function  $H(x)$  returns the same value for 2 different keys?



- What data is at position 2? The data about the key 482 or 762?
- At the start the Hash Table is empty. If we insert 482 first, 482 goes to the position 2. Then, when we insert 762, it should also go to position 2, but it is already taken. We need to handle this situation...
- This is called a collisions and the Hash Table needs a way to handle collisions
- Load factor = number in the hash table / size of the hash table (% of hash table positions used)

# Some ideas

- First, we should try to avoid collisions or minimize them
- Solution: the hash function  $H(x)$  should only return unique values
- Not possible in practice!
- The input space is much larger than the output of the hash function
- Example: we want to create a hash table to store information about students, each student is represented by a 10 character string
- There are 141 trillions combinations!
- Our data table is much smaller, collisions are inevitable
- Second solution:  $H(x)$  should minimize the collisions by spreading the input values equally

Example:

- Input: a string  $s$  of 5 characters, 128 buckets available
- $H1(x) = \text{ASCII}(s[0]) \bmod 128$
- $H2(x) = \text{for } c \text{ in } s: \{h = (h * 31 + \text{ord}(c)) \% 128\}$

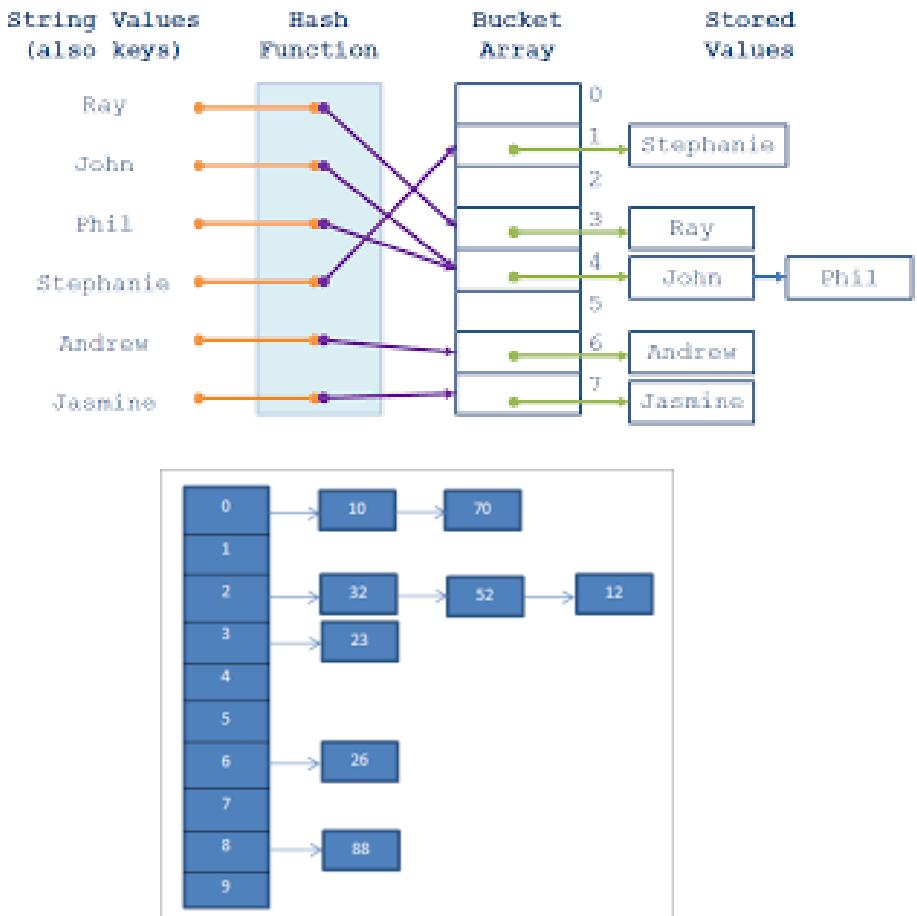
# Some ideas

- $H1(x) = \text{ASCII}(s[0]) \bmod 128$
- $H2(x) = \text{for } c \text{ in } s: \{h = (h * 31 + \text{ord}(c)) \bmod 128\}$
- All the names starting with the same letter go to the same bucket.
- Like “apple” or “ant” or “all” goes to the same bucket
- Since the initial letter is not uniformly distributed, some buckets will have a lot of collisions, some other will be under used.
- We want  $H(x)$  to distribute the values in the most uniform way we can
- $H2(x)$  is more complex but it is more random and names are spread more uniformly over buckets.



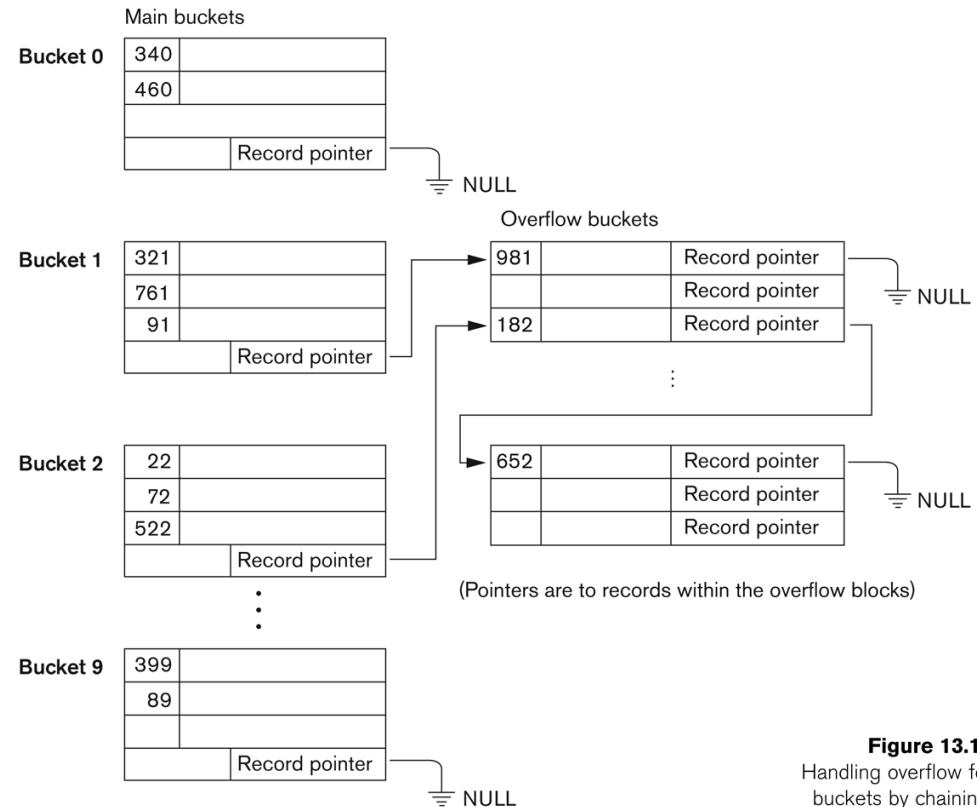
# How to Handle Collisions / Chaining

- Bucket can store more than 1 data value
- A bucket is usually a dynamic array or a list. (in memory) or a block of data (on disk)
- When there is a collision, the element is added to the list of elements for that bucket
- Using the Hash function, I can get the position of a bucket and then I do linear search inside each bucket list, that should be small enough to keep good performance
- If a bucket is full due to too many collisions and they have fixed size, we could use overflow buckets



# Collisions / Chaining and Overflow

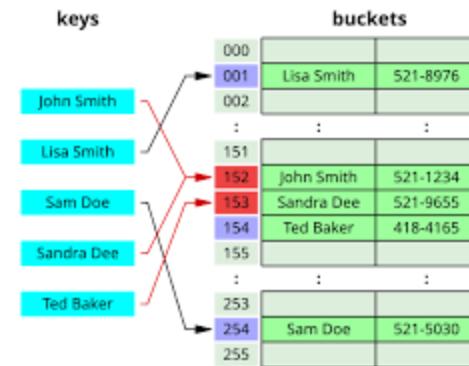
- If a bucket is full due to too many collisions and they have fixed size, we could use overflow buckets, common to all the buckets



**Figure 13.10**  
Handling overflow for buckets by chaining.

# How to Handle Collisions / Open Addressing

- Buckets store exactly 1 value
- If there is a collision
- If a collision happens, the algorithm **probes** (searches) for another empty bucket according to a rule.
- Let's assume the size of the hash table is **size**
- Main three strategies:
  1. Linear probing: check next slots →  $(index + n) \% \text{size}$  for  $n=1 \dots \text{size}$
  2. Quadratic probing: check  $(index + n^2)$ ,  $(index + n^2)$ , for  $n=1 \dots \text{size}$
  3. Double hashing: use a second hash function for the step size.



# Linear Probing

Linear probing is a method to resolve collisions in a hash table using open addressing.

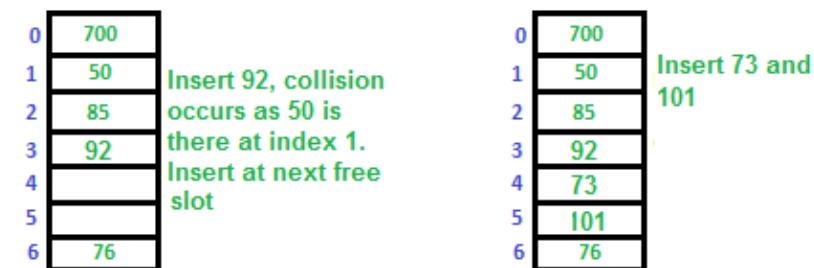
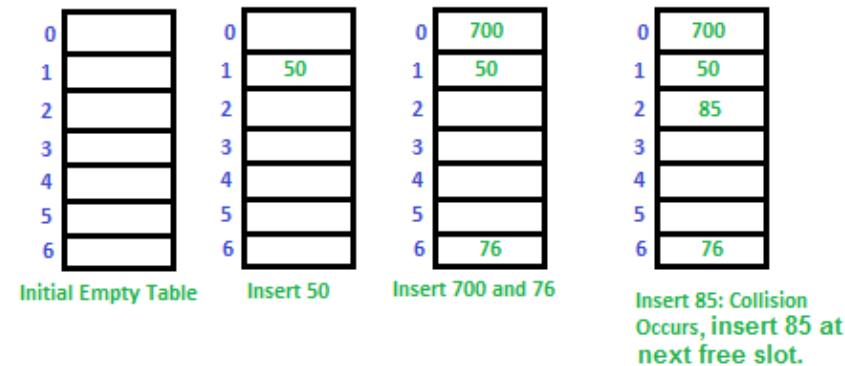
When a key hashes to a bucket that is already occupied, linear probing searches for the next available bucket in a sequential manner.

Typically, the next bucket is checked by adding 1 each time.

Formula:  $\text{Index} = (h+i) \bmod \text{capacity}$

- Pros: simple, memory-efficient
- Cons: primary clustering of consecutive slots, sensitive to high load factor
- Complexity: O(1) with no collisions, gradually becoming O(n) with high number of collisions

$$\text{Index} = (h+i) \bmod \text{capacity}, \\ i=0,1,\dots$$



# Quadratic Probing

**Quadratic probing** is an **open addressing** collision resolution **technique**.

When a collision occurs, instead of checking the **next sequential slot** (like linear probing), it checks slots at a **quadratic distance** from the original hash.

Formula: **Index =  $(h+i^2) \bmod \text{capacity}$**

- Pros: still simple, avoid big primary sequential clusters
- Cons: clusters are still possible (they are called secondary clusters), no guarantee to find a place in the table
- Complexity: O(1) with no collisions, gradually becoming O(n) with high number of collisions

$$\text{index} = (h + i^2) \bmod \text{capacity}, i = 0, 1, \dots$$

Index	Value
0	17
1	8
2	-
3	3
4	-
5	-
6	-

Index	Value
0	17
1	8
2	-
3	3
4	<b>14</b>
5	-
6	-

Add 14

- $14 \bmod 7 = 0$  (Bucket 0 occupied)
- Try bucket  $(0+1^2) = 1$  (Bucket 1 occupied)
- Try bucket  $(0+2^2) = 4$  Bucket 4 Free)
- Insert 14 into bucket 4

# Double Hashing

$$\text{index} = (h_1(x) + i * h_2(x)) \bmod \text{size}, \quad i = 0, 1, \dots$$

Double hashing uses two hash functions instead of one.

- The first hash function  $h_1(\text{key})$  determines the initial bucket.
- The second hash function  $h_2(\text{key})$  determines the step size for probing if a collision occurs.

Even keys with same initial hash go to different sequences of buckets since  $h_1$  and  $h_2$  are different → reduces clustering significantly.

Example:

Suppose we have:

Hash table capacity = 7

Keys = 10, 3, 17

Define two hash functions:

$h_1(\text{key}) = \text{key \% 7}$  → initial bucket

$h_2(\text{key}) = \text{key \% 5}$  → step size (must be non-zero and less than capacity)

# Double Hashing

$$H1(x) = x \bmod 9$$

$$H2(x) = x \bmod 5$$

$$H1 + i * H2$$

Index	Value
0	17
1	8
2	-
3	3
4	-
5	34
6	-
7	
8	



Index	Value
0	17
1	8
2	-
3	3
4	-
5	34
6	-
7	
8	

Add 14

- $14 \bmod 9 + 0 * 14 \bmod 5 = 5$  (bucket 5 occupied)
- Try bucket  $= (14 \bmod 9 + 1 * 14 \bmod 5) \bmod 9 = 14 \bmod 9 = 5$  (bucket 5 occupied)
- Try bucket  $= (14 \bmod 9 + 2 * 14 \bmod 5) \bmod 9 = 8 \bmod 9 = 8$  (bucket 8 free)

# Double Hashing

$$H1(x) = x \bmod 9$$

$$H2(x) = x \bmod 5$$

$$H1 + i * H2$$

Index	Value
0	17
1	8
2	-
3	3
4	-
5	34
6	-
7	
8	



Index	Value
0	17
1	8
2	-
3	3
4	-
5	34
6	-
7	
8	14

Add 14

- $14 \bmod 9 + 0 * 14 \bmod 5 = 5$  (bucket 5 occupied)
- Try bucket  $= (14 \bmod 9 + 1 * 14 \bmod 5) \bmod 9 = 14 \bmod 9 = 5$  (bucket 5 occupied)
- Try bucket  $= (14 \bmod 9 + 2 * 14 \bmod 5) \bmod 9 = 8 \bmod 9 = 8$  (bucket 8 free)

# Comparison

- All the techniques start with a search complexity of  $O(1)$  and they degenerate to linear search  $O(n)$  when the load factor starts to grow
- However, they degenerate “at different speed”. Linear probing is the easier to implement, but the first to degenerate.. Chaining is more complex (it requires also more memory to build the index and access the data) but it is more robust to high load factor

Load Factor ( $\alpha$ )	Linear Probing	Quadratic Probing	Double Hashing	Chaining
0.1 – 0.5	Excellent $O(1)$	Excellent $O(1)$	Excellent $O(1)$	Excellent $O(1)$
0.6 – 0.8	Degrades (clustering)	Good	Very Good	Still Good
0.8 – 1.0	Poor	Fair	Still OK	Good (slower but stable)
> 1.0	Not possible (open addressing must resize)	Not possible	Not possible	Still works (chains just grow)



# Dynamic And Extendible Hashing

- Dynamic and Extendible Hashing Techniques
  - Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records.
- Dynamic hashing use the binary representation of the hash value  $h(K)$  in order to access a directory.
  - In extendible hashing the directory is an array of size  $2^d$  where  $d$  is called the global depth.

# Dynamic And Extendible Hashing

- The directories can be stored on disk, and they expand or shrink dynamically.
  - Directory entries point to the disk blocks that contain the stored records.
- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks.
  - The directory is updated appropriately.
- The directories can be stored on disk, and they expand or shrink dynamically.
  - Directory entries point to the disk blocks that contain the stored records.
- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks.
  - The directory is updated appropriately.

# Extendible Hashing

## Key structure:

A **directory**: array of pointers to buckets.

Each **bucket**: holds a few key-value pairs and a local depth.

**Global depth**: number of hash bits used to index the directory.

## Expansion rule:

When a bucket overflows, it splits.

If the bucket's local depth becomes larger than the global depth → **double the directory size (expand)**.

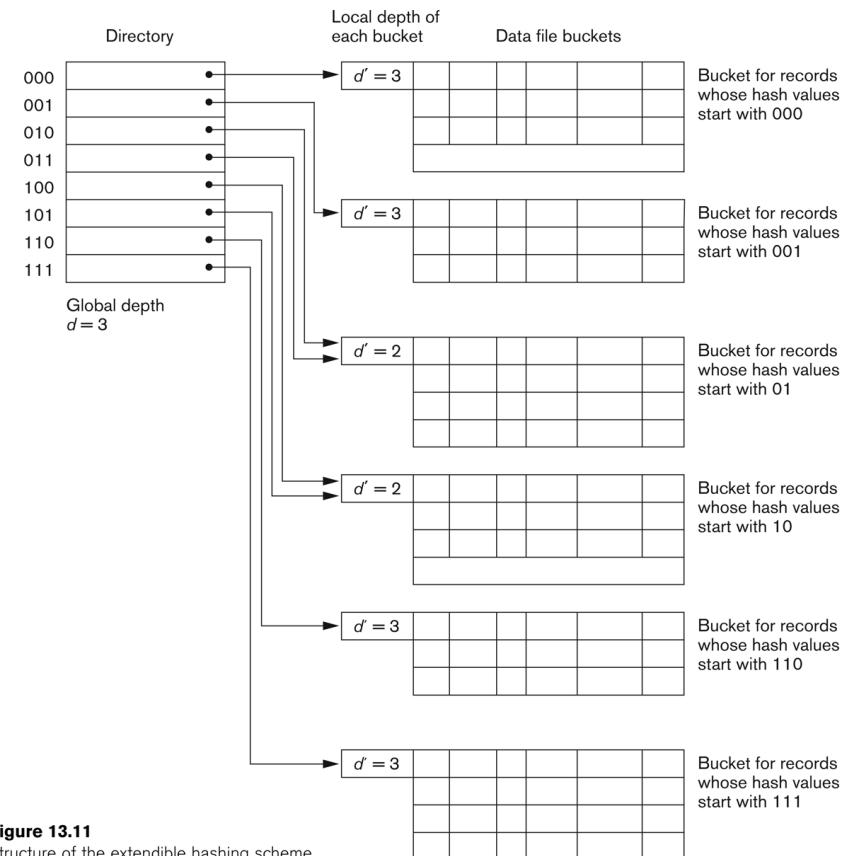
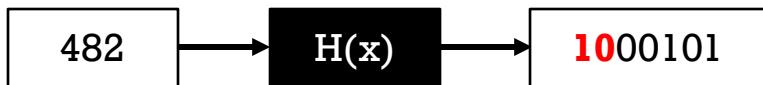


Figure 13.11  
Structure of the extendible hashing scheme.

# Dynamic Hashing



Let's assume

- Bucket size = 2
- At the beginning, only the first 2 bits are used to compute the bucket location. Therefore 4 buckets are indexed at the start ( $2^2$ )
- The hash function returns 7 bits (global depth =  $2^7$  = 128), a max of 128 buckets can be supported. In real systems, it can go up to 32 bits or more.

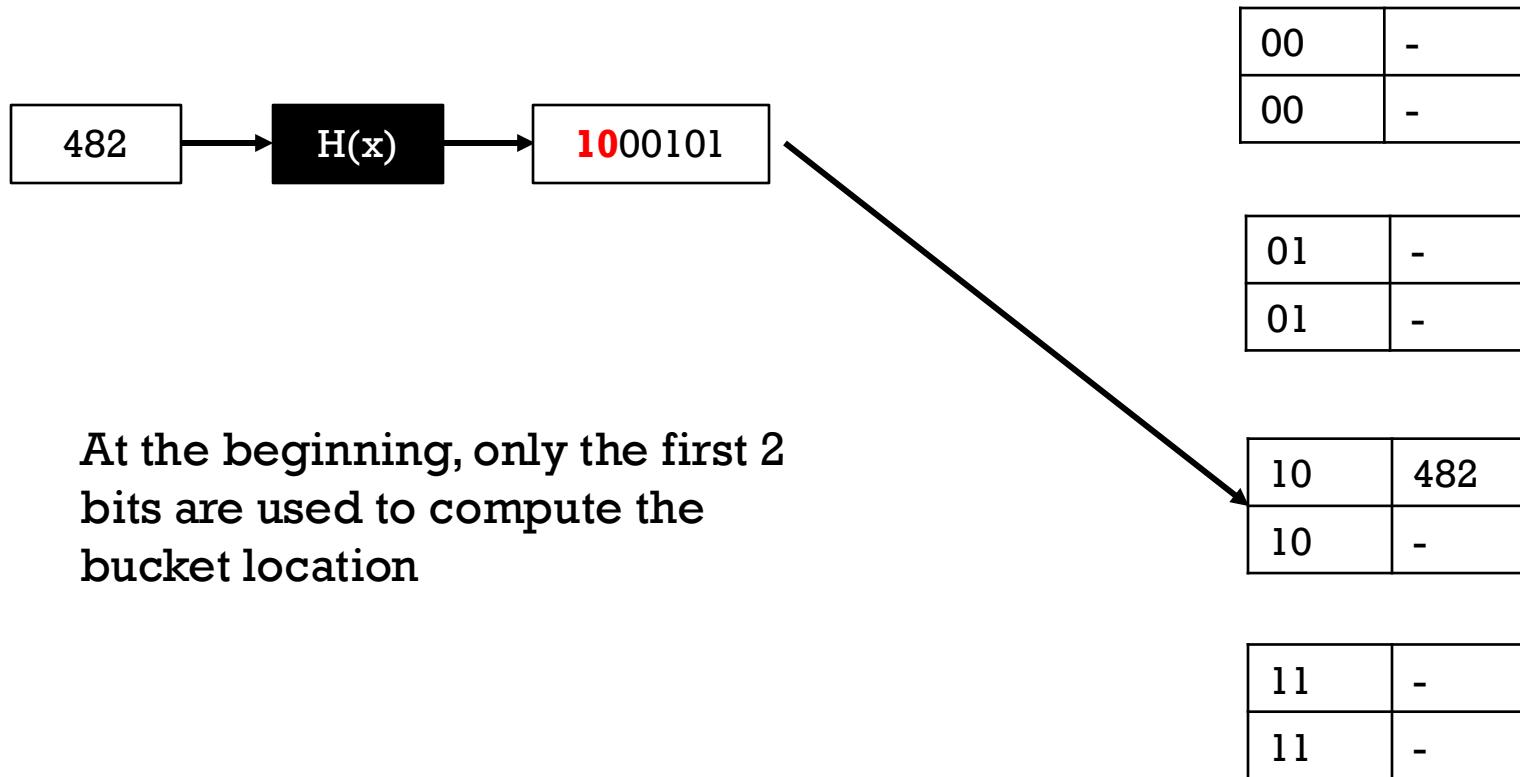
00	-
00	-

01	-
01	-

10	-
10	-

11	-
11	-

# Dynamic Hashing



# Dynamic Hashing



00	-
00	-

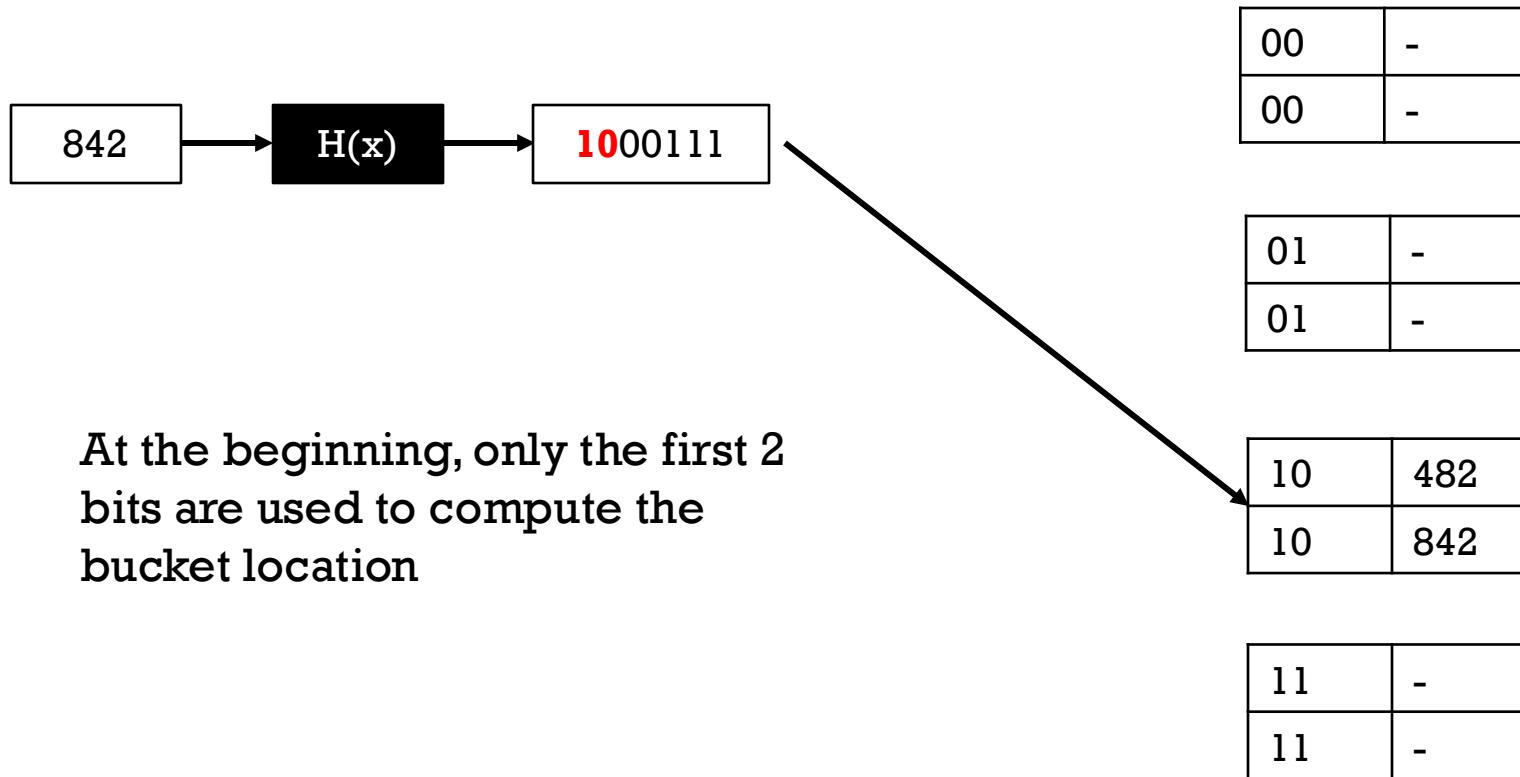
01	-
01	-

At the beginning, only the first 2 bits are used to compute the bucket location

10	482
10	

11	-
11	-

# Dynamic Hashing



# Dynamic Hashing



00	-
00	-

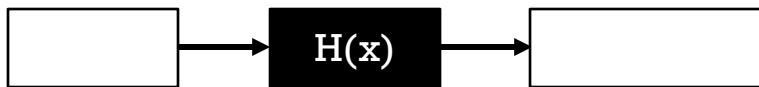
01	-
01	-

Bucket 10 is full, double the index and use 1 extra bit from the hash function to identify the bucket

10	482
10	842

11	-
11	-

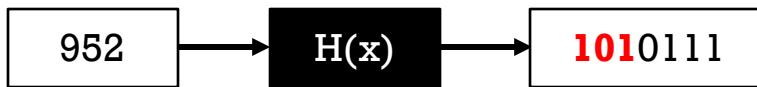
# Dynamic Hashing



Now there are  $2^3=8$  buckets,  
and we use 3 bits from the hash  
function value

000	-
000	-
001	-
001	-
010	-
010	-
011	-
011	-
100	482
100	842
101	-
101	-
110	-
110	-
111	-
111	-

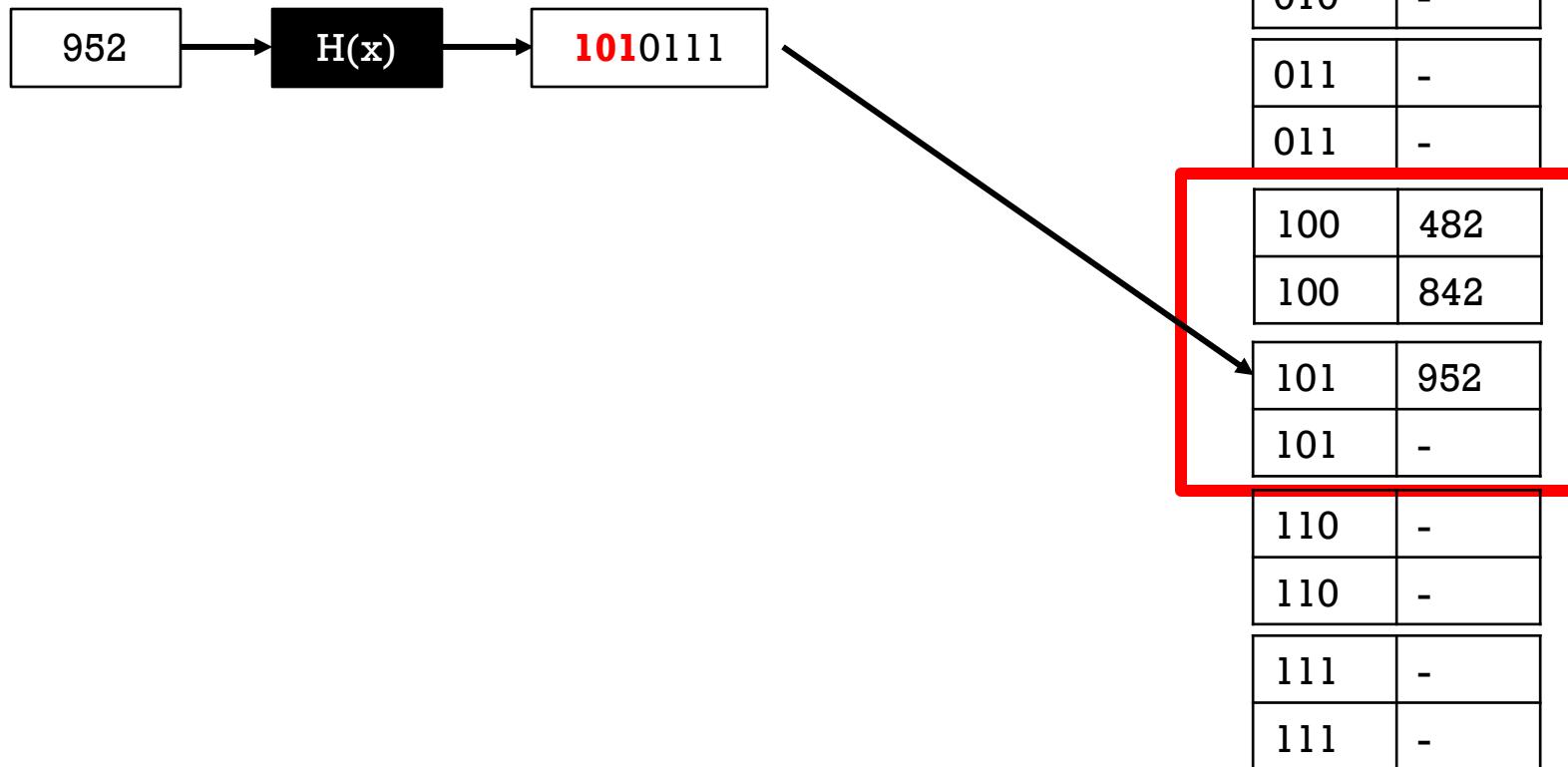
# Dynamic Hashing



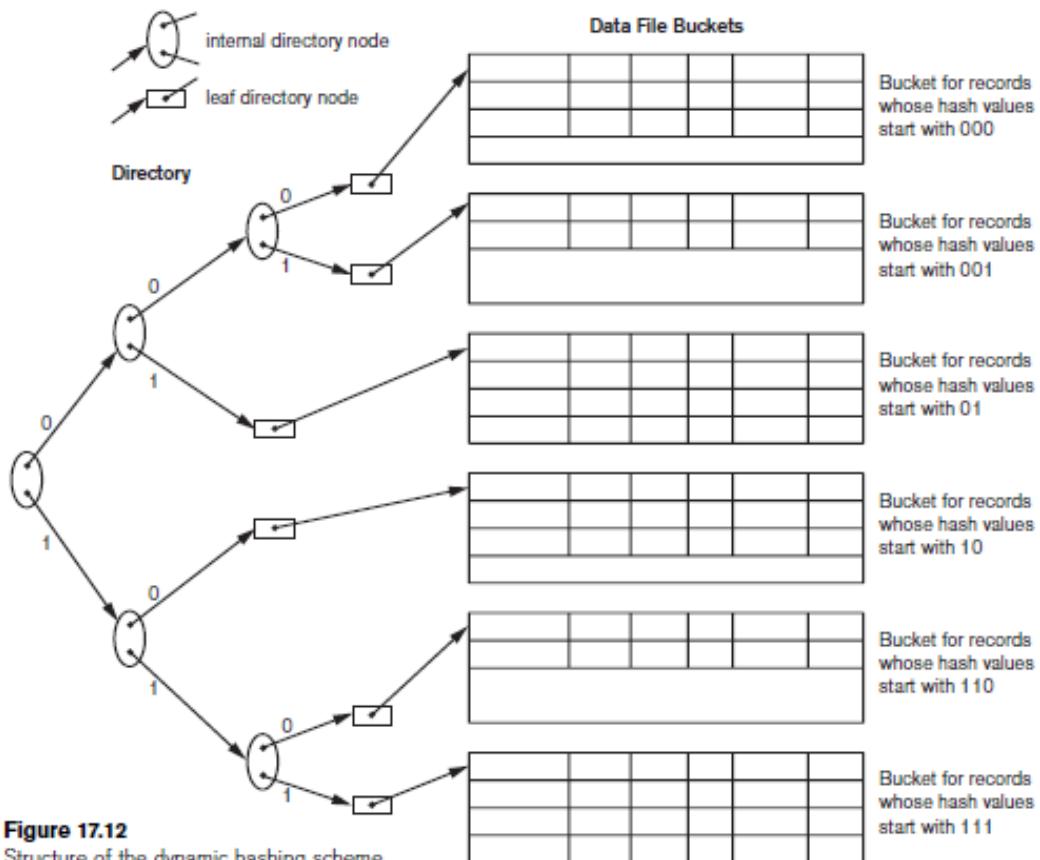
Now 3 bits are used for the hash function, 952 goes to bucket 101

000	-
000	-
001	-
001	-
010	-
010	-
011	-
011	-
100	482
100	842
101	-
101	-
110	-
110	-
111	-
111	-

# Dynamic Hashing



# Dynamic Hashing



**Figure 17.12**  
Structure of the dynamic hashing scheme.



# Exercises

Consider an Open Addressing Hash Table of size 13 (from 0 to 12), where the hash function uses the mod operator.

Show how the numbers 372, 553 and 651 would be inserted in the above table using linear probing

Show how the numbers 372, 553, 651 would be inserted in the above table using quadratic probing

Index	Value
0	—
1	261
2	40
3	-
4	900
5	-
6	29
7	265
8	53
9	-
10	540
11	-
12	638

# Java Classes

- **HashMap class**
- See *HashMapDemo.java* on Moodle for a demo
- **Features**
  - Hash Function is the hashCode() function built in in java
  - Hascode() produces an integer (int) that represents the input object.
  - In a hash table, this integer is used to compute the bucket index
  - Example: “hello”.hashCode() → 99162322
  - Formula for strings:

$$\text{hash}(s) = s[0] \times 31^{n-1} + s[1] \times 31^{n-2} + \dots + s[n - 1]$$

- Starting capacity (number of buckets) = 16. Given the hashCode, the bucket will be found with the formula  $H(x) \bmod \text{capacity}$
- It is an Hash Table with chaining
- Dynamic hashing used if load factor > 0.75. The capacity is doubled.

# Java Classes

- **HashMap class Methods**
- **Constructor:**

```
HashMap<String, Integer> map = new HashMap<>(size);
```

- **put(key,value) = insert a key/value pair in the hash table:** map.put("apple", 5)
- **get(key) = return the value for a specific key:** map.get("apple")
- **map.containsKey(key) = check if a key is in the map** map.containsKey("cherry")
- **map.containsValue(value) = check if a value is in the map** Ex:map.containsKey(10)
- **map.remove("banana") = remove an entry from the Hash table**
- **Loop over the elements:**

```
for (Map.Entry<String, Integer> entry : map.entrySet()) {  
    System.out.println("Key:" + entry.getKey() + ", Value:" + entry.getValue());  
}
```