

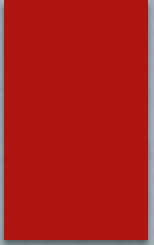
Maps, Hash tables, Dictionaries

Week 8 - LGT

Dr Lin Gui

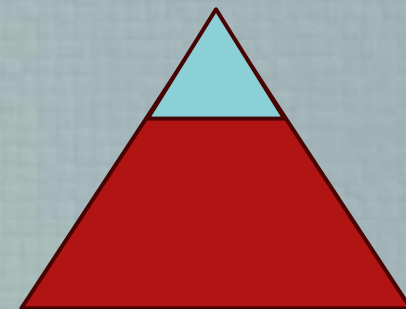
Lin.1.gui@kcl.ac.uk



- 
- ▶ Lin Gui,
 - ▶ Email: Lin.1.gui@kcl.ac.uk
 - ▶ Lecturer in Natural Language processing
 - ▶ Office hour:
 - ▶ Week 9 (10th – 14th march): Monday, 11:00 – 13:00; Friday, 14:00-16:00
 - ▶ Week 10 (17th – 23th march): Monday, 11:00 – 13:00; Tuesday, 14:00-16:00
 - ▶ Week 11 (24th – 30th march): Monday: 11:00 – 13:00
 - ▶ The content of my LGT includes:
 - ▶ 1) The content based on pre-recorded videos
 - ▶ 2) # Important concepts
 - ▶ 3) * extend your knowledge (some commonly seen questions in job interview)

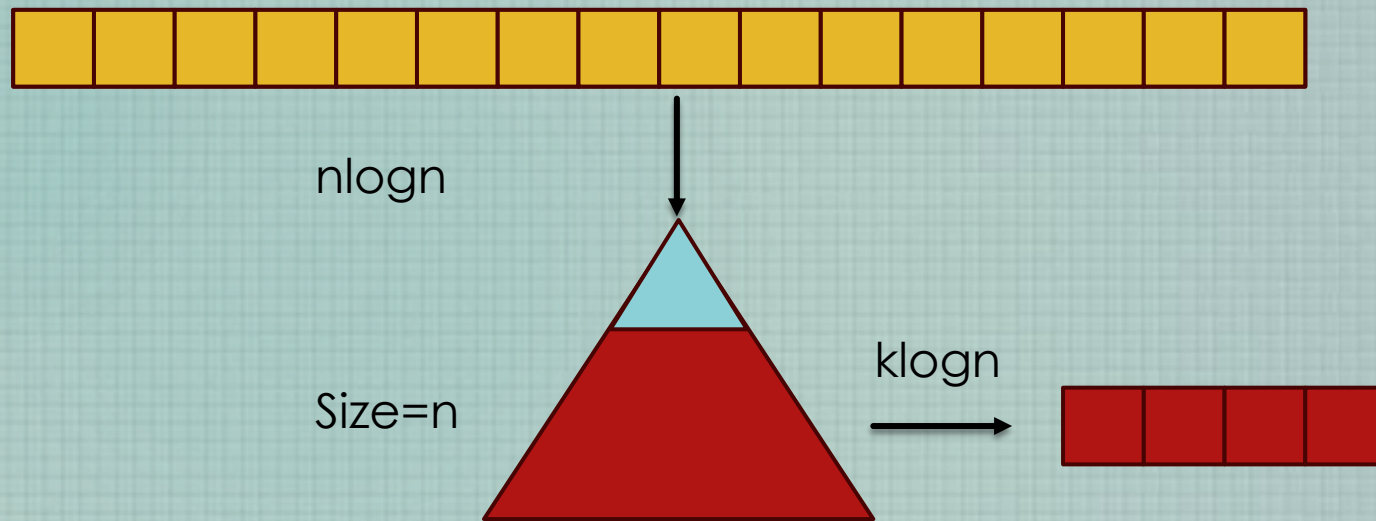
Before we start...

- ▶ Some So, questions after the LGT last week
- ▶ Why using min-heap to find top k largest elements, but max-heap to find top k smallest elements
- ▶ Max-heap: the element in root is larger than all other nodes
- ▶ Min-heap: the element in root is smaller than all other nodes.
- ▶ To find the top k largest element, we need to find the #k-th element in the array first. Then all the elements larger than this element is to top k element.
- ▶ So we choose Min-heap. If the new element is larger than the root, then put it into the heap and replace the root.
- ▶ After one pass of scanning of the array, we could get a k-size min-heap, where:
 - ▶ There are k element in the heap.
 - ▶ All the elements in the heap are no smaller than the root.
 - ▶ All the elements not in the heap are smaller than the root.



Before we start...

- ▶ Some So, questions after the LGT last week
- ▶ Why using min-heap to find top k largest elements, but max-heap to find top k smallest elements
- ▶ Using max-heap



Before we start...

- ▶ Some So, questions after the LGT last week
- ▶ Why using min-heap to find top k largest elements, but max-heap to find top k smallest elements
- ▶ Using min-heap

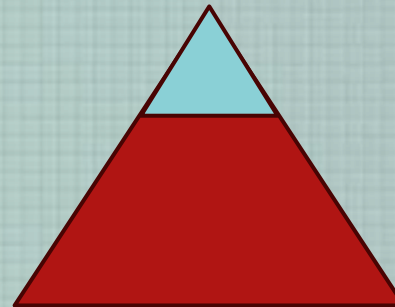


$n \log k$



Compare with **root** first.
Only the element **larger** than **root**
can be put into the heap and
replace the root.

Size=k



Before we start...

- ▶ Question about the final exam:
- ▶ Do we have coding questions in the exam?
- ▶ Yes, about 50% marks are based on coding questions

Before we start...

- ▶ Question about the final exam:
- ▶ How does the coding questions look like?
- ▶ The exam questions are very similar to the mock test and quiz questions. Please make sure you can solve them perfectly before the exam. Even if you've passed the test with full marks, please remember to submit at least one wrong answer to the system — this way, you can receive feedback with the correct solution, which might be more efficient than your own approach.
- ▶ Since there are several coding questions, and all of them have simple solutions, please make sure you can complete them within two hours.

Before we start...

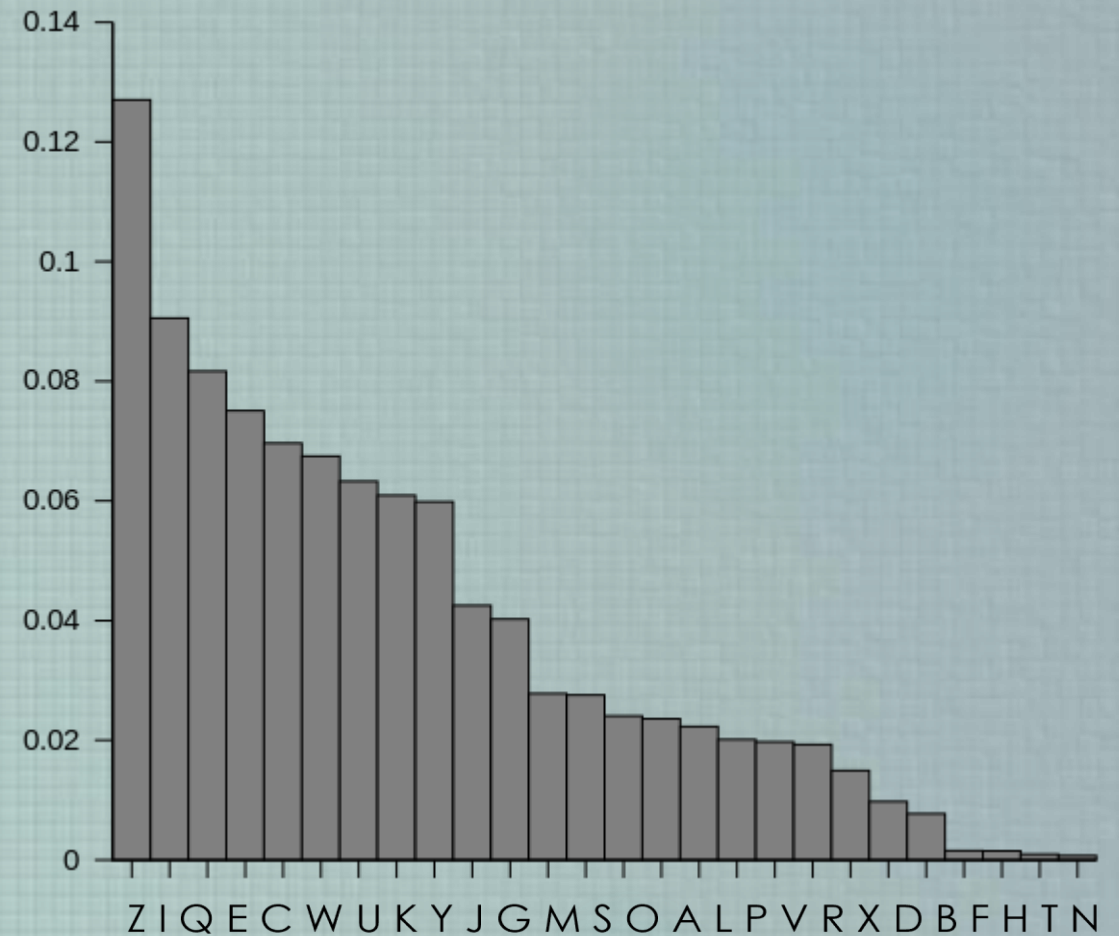
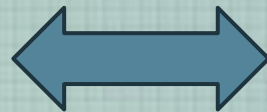
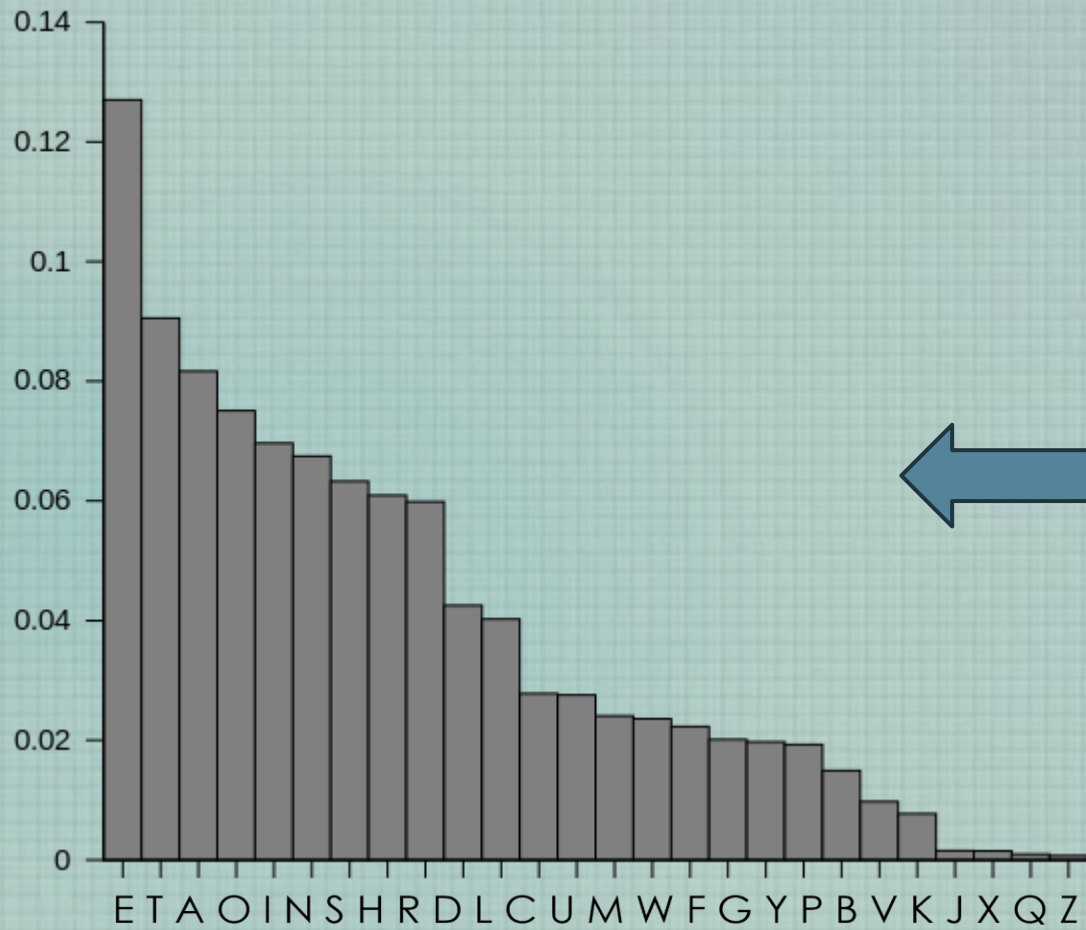
- ▶ Question during the lab session:
- ▶ Any recommended programming environment on local PC for practicing?
- ▶ Yes, try the online platform
 - ▶ <https://www.jdoodle.com/online-java-compiler>
- ▶ You can even practice your coding skills on your mobile phone.

Before we start...now, a puzzle

- ▶ A puzzle before today's LGT.
- ▶ Imagine you are a British Commander in WWII. One day, you receive a message from your allies:
- ▶ **CI'U Q IYQG. ZWZV MEWMZQGZJ – KEGJ VESY PYESWJ EY MKQWPZ COOZJCQIZGV. IYSUI WM DCUCXGZ AZQBWZUU.**
- ▶ Along with the message, you also receive a **strange book** written in this unfamiliar language.
- ▶ Additionally, you have several weeks' worth of **local newspapers** written in English.
- ▶ How would you go about deciphering this mysterious message?
- ▶ It might be a code-switched message!

Before we start...now, a puzzle

- ▶ The idea: No matter how you switch the code, the overall letter distribution should remain the same!



Before we start...now, a puzzle

- ▶ The idea: No matter how you switch the code, the overall letter distribution should remain the same!
- ▶ Find the letter distributions from **code switched book** and **newspaper**. By comparing their sorted frequencies, we can deduce the code-switching rule.

Create two arrays

A for switched book

B for newspaper

The new letter from switched book coming, the corresponding $A[i]++$

For example, **if** read letter "C", **then** $A[2]++$



The new letter from newspaper coming, the corresponding $B[i]++$



Then, sorting A, B, and matching

Before we start...now, a puzzle

- ▶ The idea: No matter how you switch the code, the overall letter distribution should remain the same!
- ▶ Find the letter distributions from **code switched book** and **newspaper**. By comparing their sorted frequencies, we can deduce the code-switching rule.

Create two arrays
A for switched book
B for newspaper

The new letter **X** from switched book coming
`A['X'-65]++`



The new letter **Y** from newspaper coming, `B['Y'-65]++`



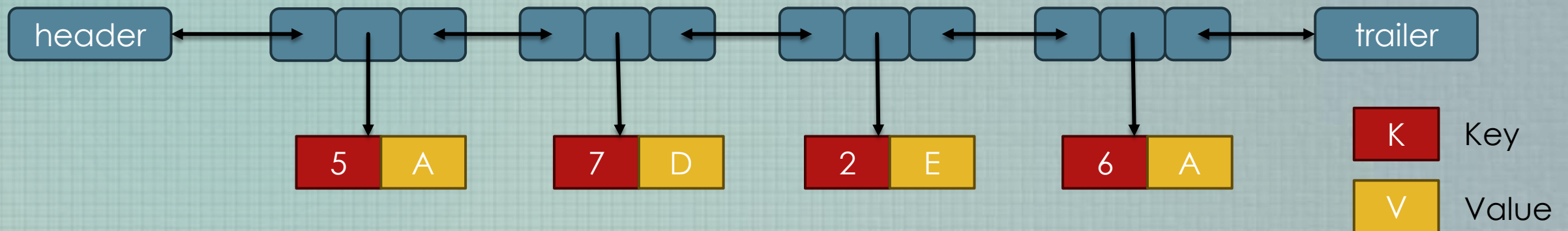
Then, sorting A, B, and matching

Outline

- ▶ Maps
 - ▶ Map ADT
 - ▶ List-Based Map Implementation#
- ▶ Hash Tables
 - ▶ Bucket Array and Hash Functions
 - ▶ Collision Handling#
- ▶ Dictionaries
 - ▶ Dictionary ADT
 - ▶ Dictionary Implementations
- ▶ Applications

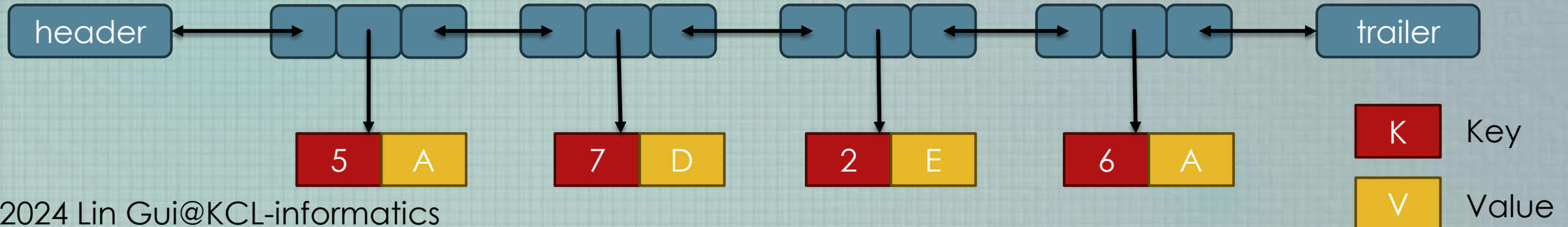
Maps#

- ▶ A map models a searchable collection of key-value entries
- ▶ The main operations of a map are for searching, inserting, and deleting items
- ▶ **Multiple entries** with the **same key** are not allowed#
- ▶ We can easily implement a map using an unsorted list
 - ▶ Duplicated values? YES!
 - ▶ Duplicated keys? NO!



Implementations with a linked list

- ▶ **get(*k*)**: Search the linked list, check if there is an entry with key = *k*
 - ▶ **If yes, return *k*; else return null**
- ▶ **put(*k*, *v*)**: Search the linked list for the key = *k*.
 - ▶ **If yes, replace existing (*k*,*v*) by the new one; else insert a new entry (*k*,*v*) to the list**
- ▶ **remove(*k*)**: Search the linked list for the key = *k*.
 - ▶ **If yes, remove the entry (*k*,*v*) from the list; else return null**
- ▶ **isEmpty()**: check if the list is empty
- ▶ **entrySet()**: scan the list and return the iterable collection of all the entries
- ▶ **keySet()**: scan the list and return the iterable collection of all the keys
- ▶ **values()**: scan the list and return an iterator of all values



Performance of a list-based map

- ▶ Performance
 - ▶ **put**, **get** and **remove** take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key#
- ▶ The unsorted list implementation is effective only for maps of small size

Common Data Structure Operations									
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Stack	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Singly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Doubly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Skip List	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n \log(n))$
Hash Table	N/A	$\theta(1)$	$\theta(1)$	$\theta(1)$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Cartesian Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
B-Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Red-Black Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Splay Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
KD Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$

Performance of a list-based map

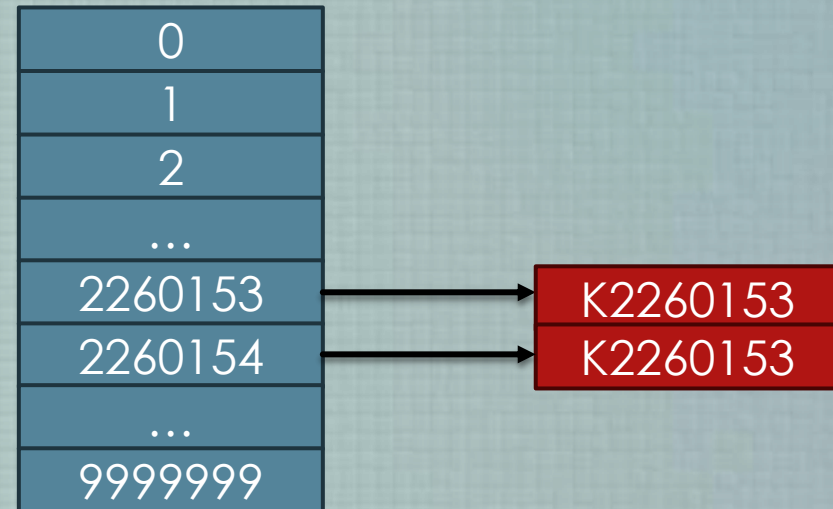
- ▶ Performance
 - ▶ **put**, **get** and **remove** take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key#
- ▶ The unsorted list implementation is effective only for maps of small size
- ▶ Discussion – why the complexity is $O(n)$? Any improvements?
 - ▶ A new data structure
 - ▶ For the given index, the cell can be accessed directly
 - ▶ For the given key, the index should be predictable
 - ▶ Then, the searching, insertion, and remove could be finished within $O(1)$ time

Outline

- ▶ Maps
 - ▶ Map ADT
 - ▶ List-Based Map Implementation#
- ▶ Hash Tables
 - ▶ Bucket Array and Hash Functions
 - ▶ Collision Handling#
- ▶ Dictionaries
 - ▶ Dictionary ADT
 - ▶ Dictionary Implementations
- ▶ Applications

Hash Table#

- ▶ Hash table is designed for map storing entries as (K, V) pair efficiently.
- ▶ For example, use your K-number as key and name as values, we can easily find a student's name with $O(1)$ time
- ▶ The hash table uses an array of size $N=10,000,000$ and the hash function of
- ▶ $h(x)=\text{last 7 digits of } x$



Hash Table#

- ▶ Hash tables are used to implement a map
- ▶ Hash table consists of two main components:
 - ▶ **Bucket array** – is an array A of size N , where each cell of A is thought of as a “bucket” (that is a collection of key-value pair)
 - ▶ **Hash function h** – maps keys of a given type to integers in a fixed interval $[0, N-1]$
 - ▶ Example: Example: $h(x) = x \% N$ is a **hash function** for integer
 - ▶ The integer $h(x)$ is called the hash value of key x
 - ▶ **Question: what if the key is not integer?**
- ▶ When implementing a map with a hash table, the goal is to store item (k,v) at index $i = h(k)$
- ▶ **Question: what if two keys share the same index?**

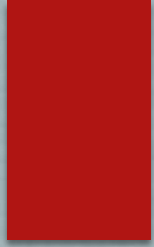
Hash Table#

- ▶ Hash tables are used to implement a map
- ▶ Hash table consists of two main components:
 - ▶ **Bucket array** – is an array A of size N , where each cell of A is thought of as a “bucket” (that is a collection of key-value pair)
 - ▶ **Hash function h** – maps keys of a given type to integers in a fixed interval $[0, N-1]$
 - ▶ Example: Example: $h(x) = x \% N$ is a **hash function** for integer
 - ▶ The integer $h(x)$ is called the hash value of key x
 - ▶ **Question: what if the key is not integer?**
- ▶ When implementing a map with a hash table, the goal is to store item (k,v) at index $i = h(k)$
- ▶ **Question: what if two keys share the same index?**
 - ▶ **Hash collision**

Hash function#

- ▶ A hash function is usually specified as the composition of two functions:
- ▶ Hash code - h_1 : keys \rightarrow integers
- ▶ Compression function – h_2 : integers $\rightarrow [0, N - 1]$
- ▶ The **hash code** is applied first, and the compression function is applied next on the result, i.e.:
- ▶ $h(x) = h_2(h_1(x))$
- ▶ Hash codes assigned to the keys should avoid collisions
- ▶ The goal of the **hash function** is to “disperse” the keys in an apparently random way

Hash Codes*



► Memory address

- We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
- Good in general, except for numeric and string keys – Why?

► Integer cast:

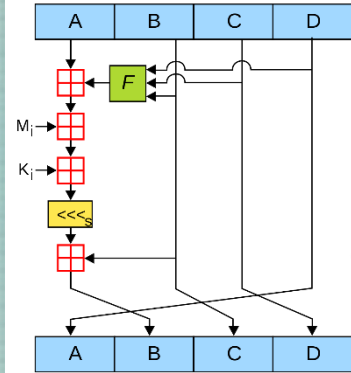
- We reinterpret the bits of the key as an integer
- Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

► Component sum:

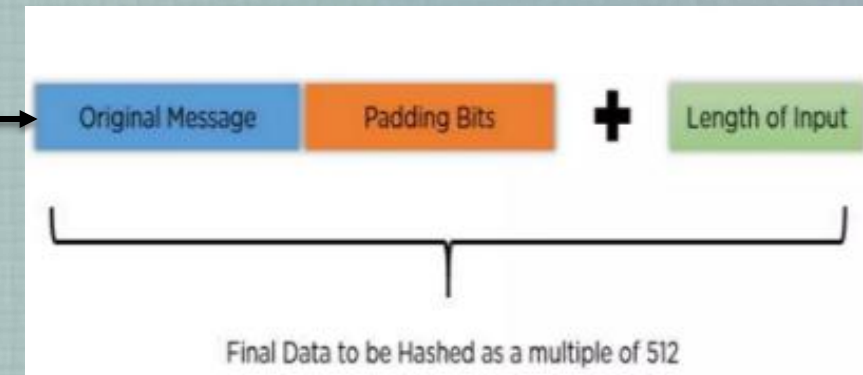
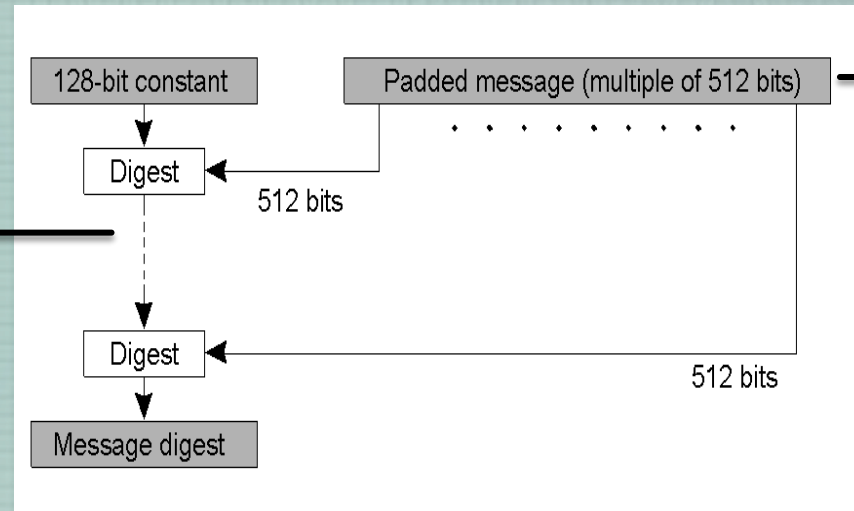
- We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
- Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes – Example: MD5

- ▶ MD5 message-digest algorithm takes as input a message of arbitrary length and produces as output a 128-bit (16 bytes) "fingerprint"
- ▶ For example:
- ▶ `md5("abc") = 900150983cd24fb0d6963f7d28e17f72`
- ▶ `md5("Hello, World!") = fc3ff98e8c6a0d3087d515c0473f8677`
- ▶ `md5("") = d41d8cd98f00b204e9800998ecf8427e`



$$\begin{aligned}F(B, C, D) &= (B \wedge C) \vee (\neg B \wedge D) \\G(B, C, D) &= (B \wedge D) \vee (C \wedge \neg D) \\H(B, C, D) &= B \oplus C \oplus D \\I(B, C, D) &= C \oplus (B \vee \neg D)\end{aligned}$$

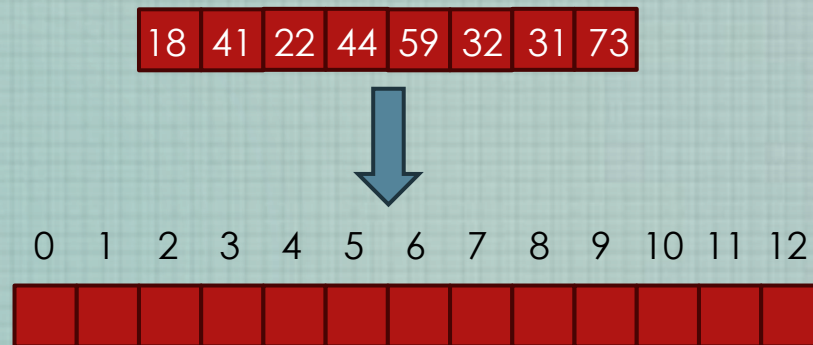


Hash Collision Handling

- ▶ Two ways to handle the hash collision
 - ▶ Separate Chaining
 - ▶ Open addressing
 - ▶ Linear Probing
 - ▶ Double hashing

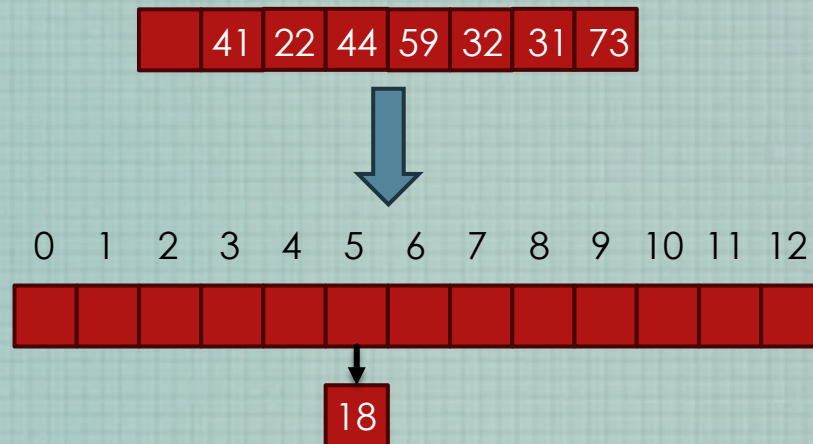
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



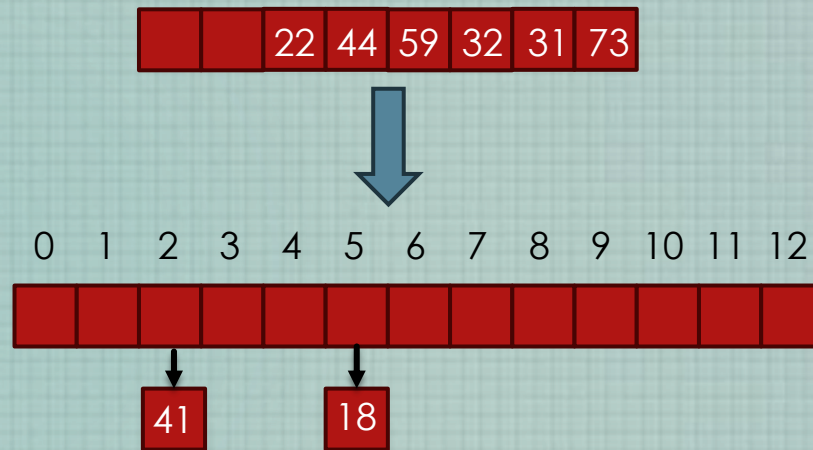
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



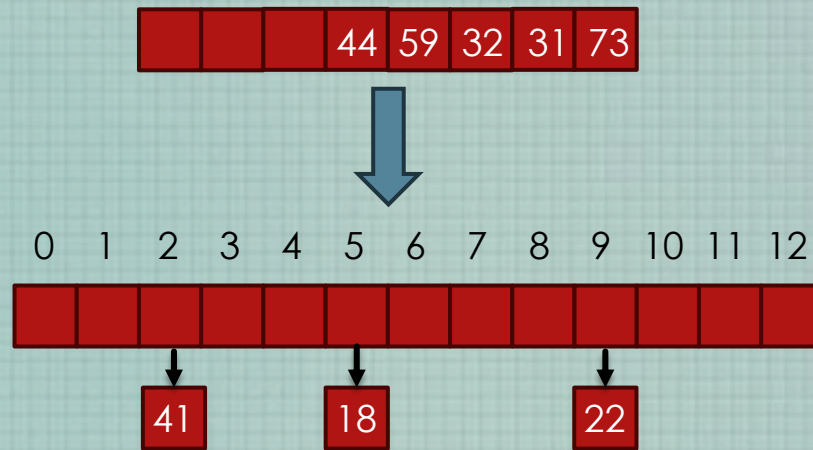
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



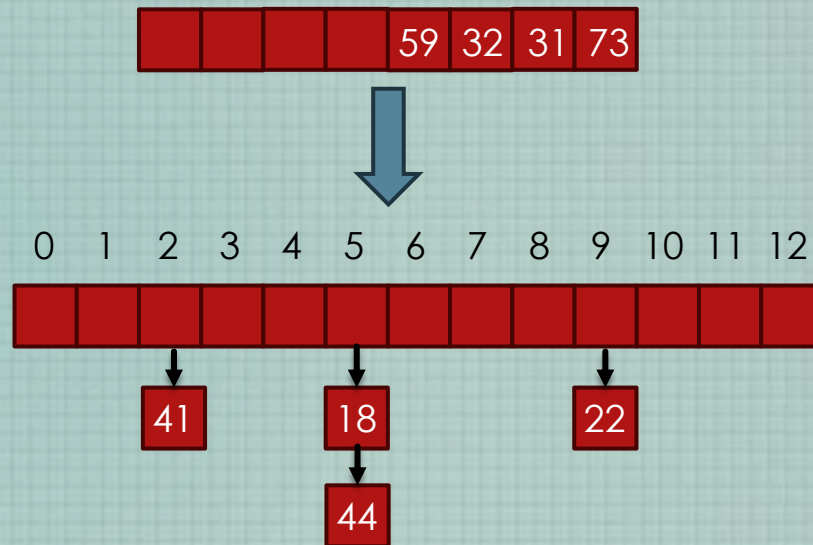
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



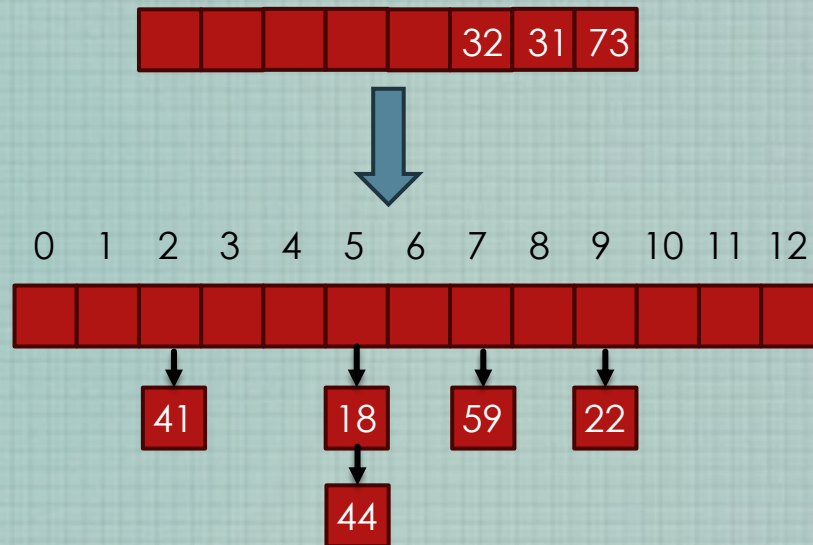
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



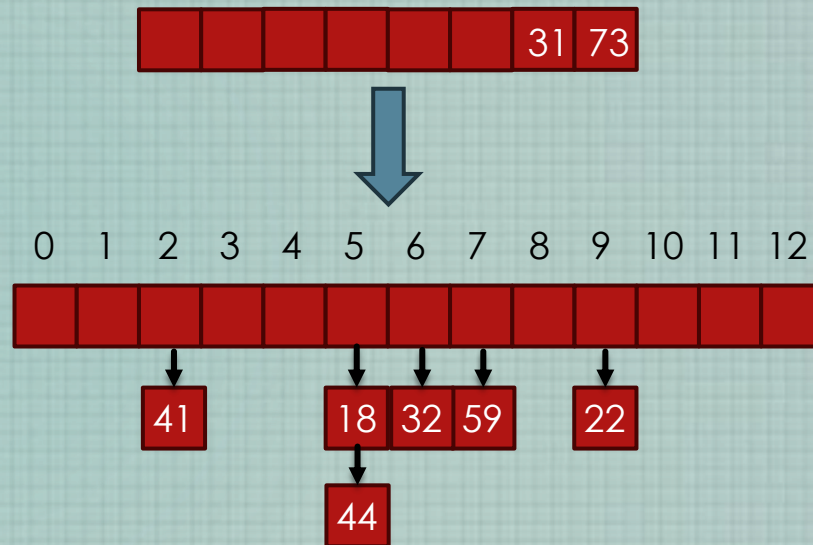
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



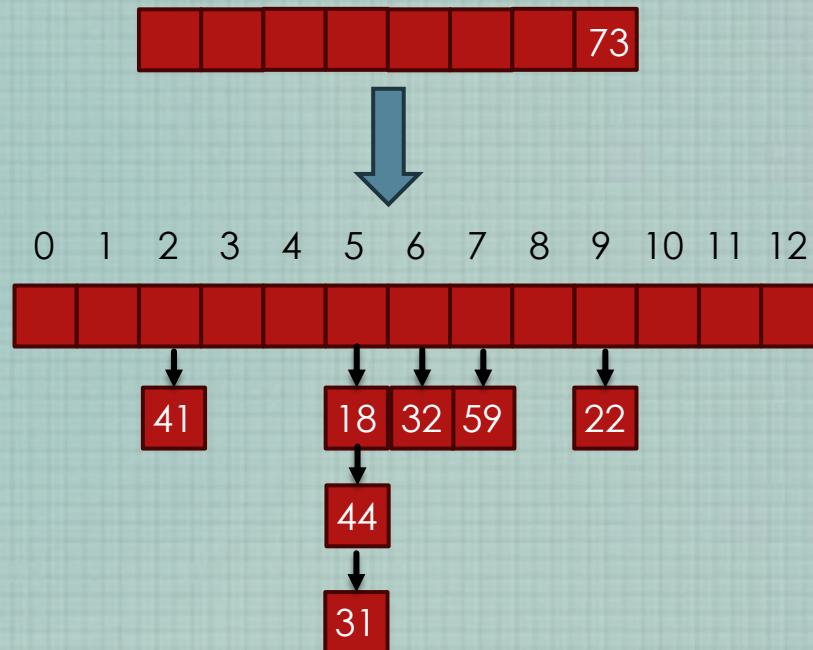
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



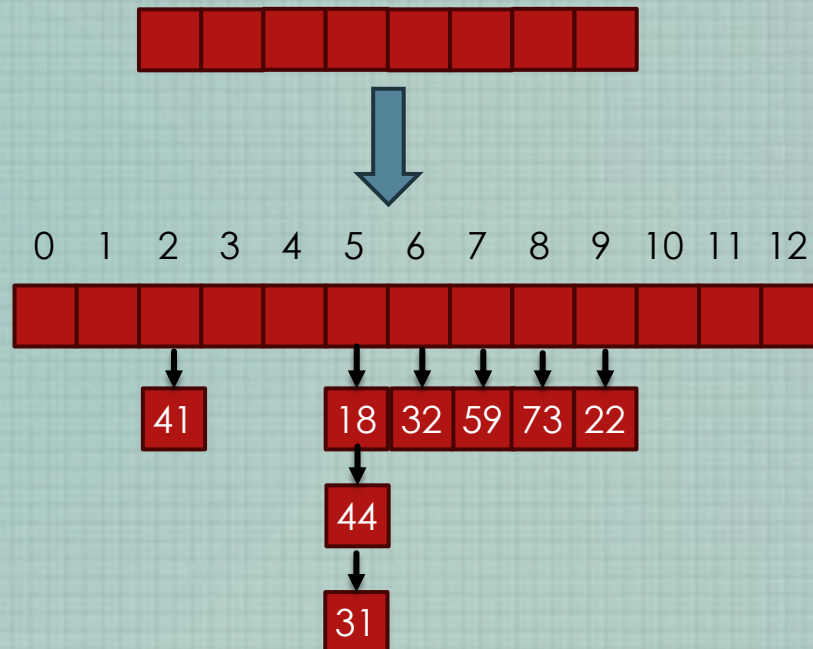
Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order

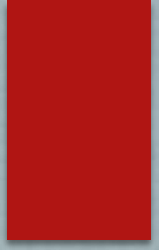


Separate Chaining

- ▶ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ▶ Separate chaining is simple, but requires additional memory outside the table
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



Map with Separate Chaining



- ▶ Delegate operations to a list-based map at each cell:

```
Algorithm get(k):  
return A[h(k)].get(k)
```

```
Algorithm put(k,v):  
t = A[h(k)].put(k,v)  
if t == null then                                {k is a new key}  
    n = n + 1  
return t
```

```
Algorithm remove(k):  
t = A[h(k)].remove(k)  
if t ≠ null then                                {k was found}  
    n = n - 1  
return t
```

Hash Collision Handling

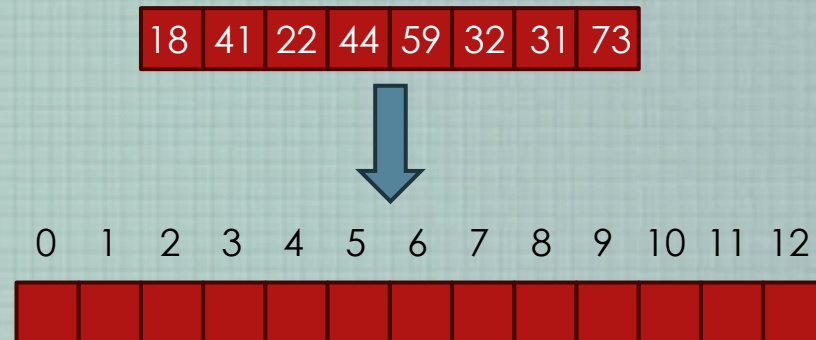
- ▶ Two ways to handle the hash collision
 - ▶ Separate Chaining
 - ▶ Open addressing
 - ▶ Linear Probing
 - ▶ Double hashing

Open addressing#

- ▶ **Open addressing:** the colliding item is placed in a **different** cell of the table:
 - ▶ Linear Probing
 - ▶ Double hashing
- ▶ Each table cell inspected is referred to as a “**probe**”
- ▶ Colliding items lump together, causing future collisions to cause a longer sequence of probes

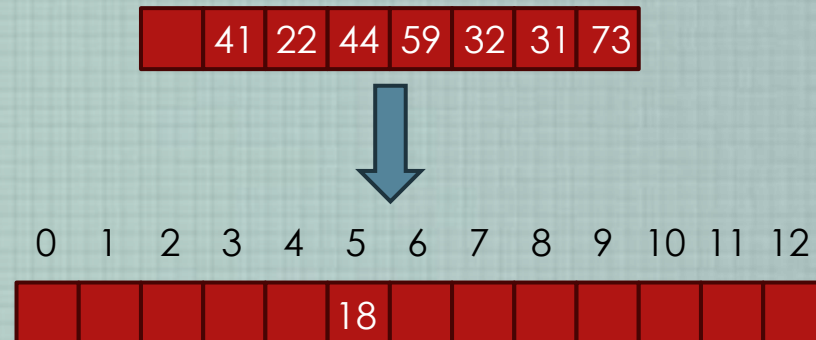
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i=h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



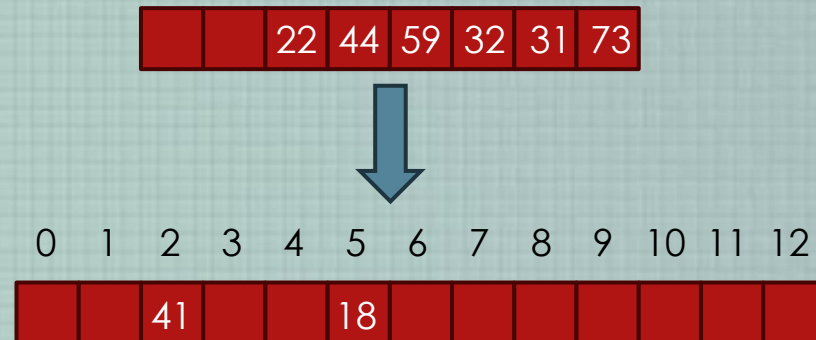
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i = h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



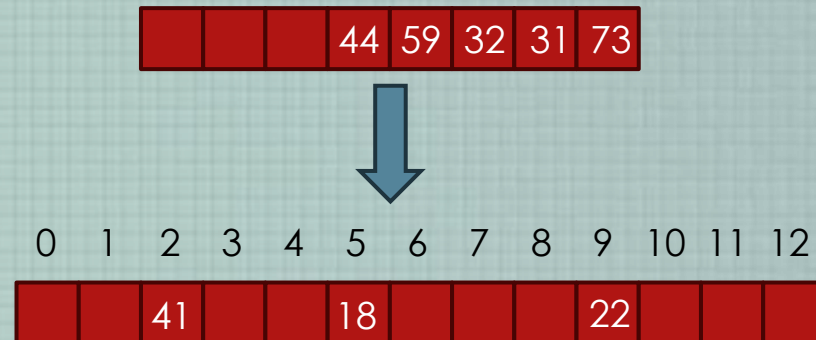
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i=h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



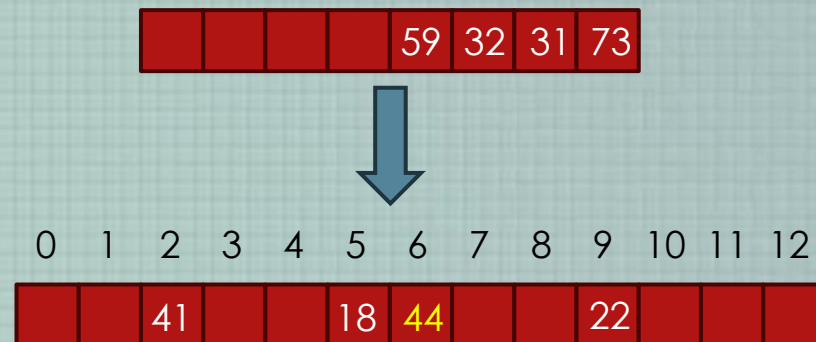
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i=h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



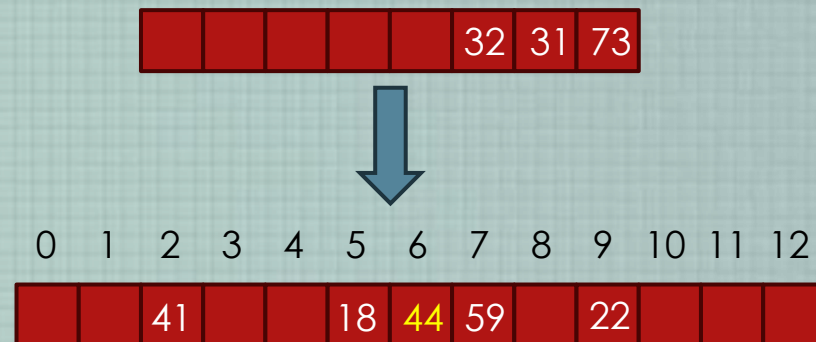
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i=h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



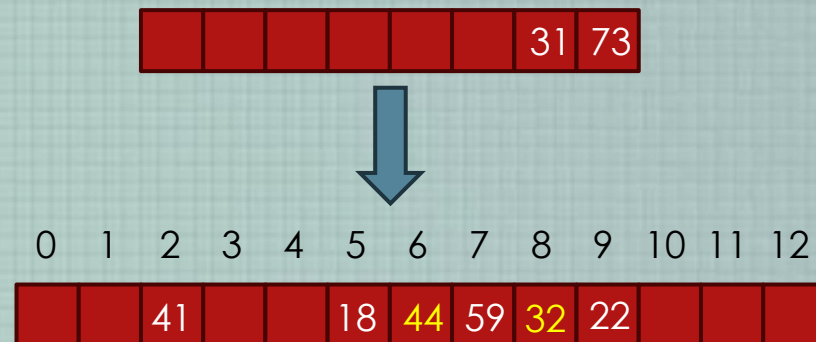
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i=h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



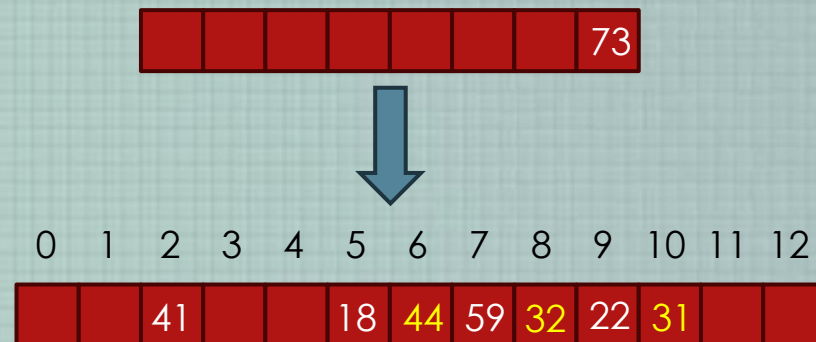
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i = h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



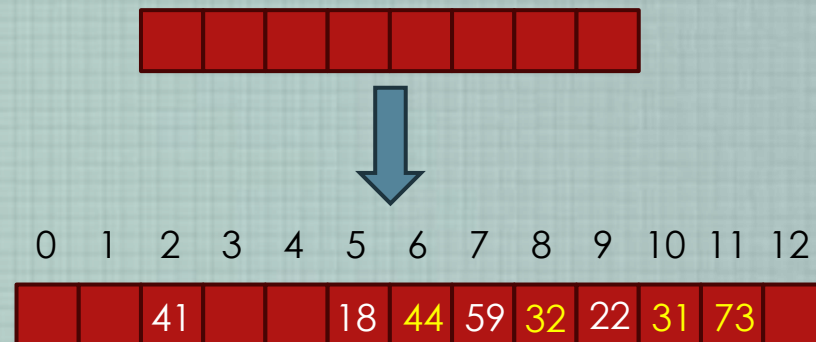
Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i = h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order



Linear Probing#

- ▶ **Linear probing**: handles collisions by placing the colliding item in the next (**circularly**) available table cell:
 - ▶ If we try to insert an entry **(k,v)** into a cell **A[i]** that is already occupied, where **$i=h(k)$** , then we try next at **$A[(i+1) \% N]$** .
 - ▶ If **$A[(i+1) \% N]$** is taken then we try **$A[(i+2) \% N]$** , and so on, until we find an empty cell.
 - ▶ In general, **keep searching the next available cell** after the first attempt of insertion
- ▶ Example:
 - ▶ $h(k) = k \% 13$ ($k \bmod 13$)
 - ▶ **Insert keys** 18, 41, 22, 44, 59, 32, 31, 73, in this order

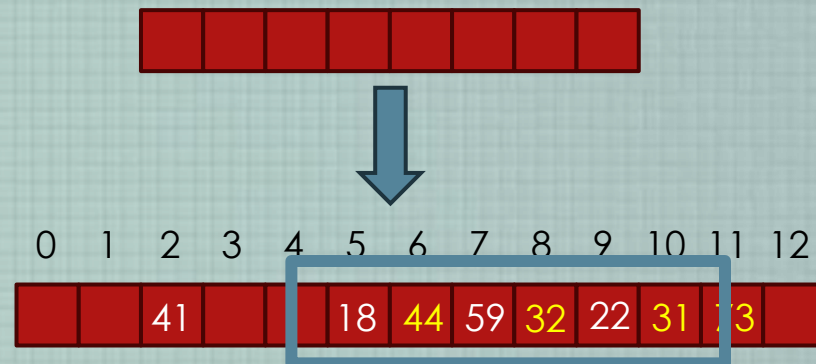


Method in Linear Probing#

- ▶ Three basic operations in hash table:
 - ▶ Search: `get(k)`
 - ▶ Start at cell $h(k)$ and probe consecutive locations until find the item with **key k**. If no **key k** is found but an empty cell is found, return **null**
 - ▶ Delete: `remove(k)`
 - ▶ Search for an entry with **key k**. If (k,v) pair is found, replace it with a special item **available** and return **value v**, otherwise return **null**
 - ▶ Insert: `put(k)`
 - ▶ Search for an entry with **key k**. If an empty cell or **available** is found, store the (k,v) in the cell. Otherwise, all cells have been unsuccessfully probed, return **null**
- ▶ One most commonly used operation in hash table:
 - ▶ For the given key k , modifying the associated value v in the table.
 - ▶ Search k first, if (k, v) pair is found, create a new pair with the value, remove old pair and insert new pair. If it is not found, insert new pair directly.

Problem in Linear Probing#

- ▶ What's the problem in linear probing?
- ▶ A dense subsequence [18, 44, 59, 22, 31] when we want to insert 73 into the hash table.
- ▶ Discussion: why?
 - ▶ only trying to find the next available cell when the collision happened.
 - ▶ If there is already a dense subsequence in the hash table, the next inserted element enlarge the size of dense subsequence.
- ▶ Solution:
 - ▶ **Is it possible for different elements to use different step sizes to skip cells when searching for the next available slot after a collision occurs?**



Hash Collision Handling

- ▶ Two ways to handle the hash collision
 - ▶ Separate Chaining
 - ▶ Open addressing
 - ▶ Linear Probing
 - ▶ Double hashing

Double Hashing#

- ▶ Double hashing uses a secondary hash function $h'(k)$
- ▶ If h maps some key k to a cell $A[i]$, with $i=h(k)$, that is already occupied, then we iteratively try the buckets:

$$A[(i + f(j)) \% N] \text{ for } j = 1, 2, \dots, N-1$$

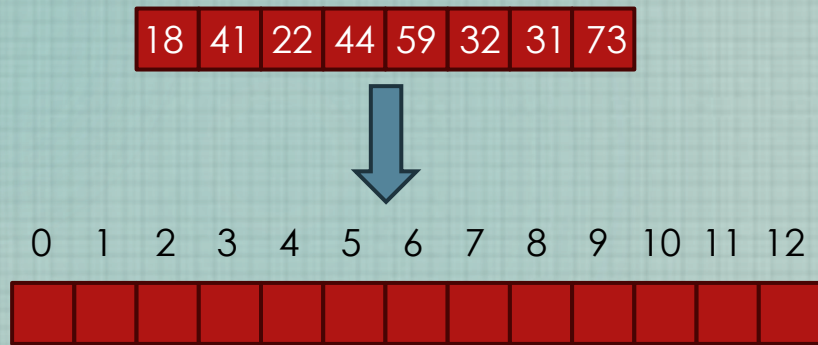
- ▶ where $f(j) = j \cdot h'(k)$
- ▶ The secondary hash function $h'(k)$ cannot have zero values
- ▶ The table size N must be a prime to allow probing of all the cells
- ▶ Common choice of compression function for the secondary
- ▶ hash function:

$$h'(k) = q - k \% q$$

- ▶ where: $q < N$ and q is a prime

Example of Double Hashing#

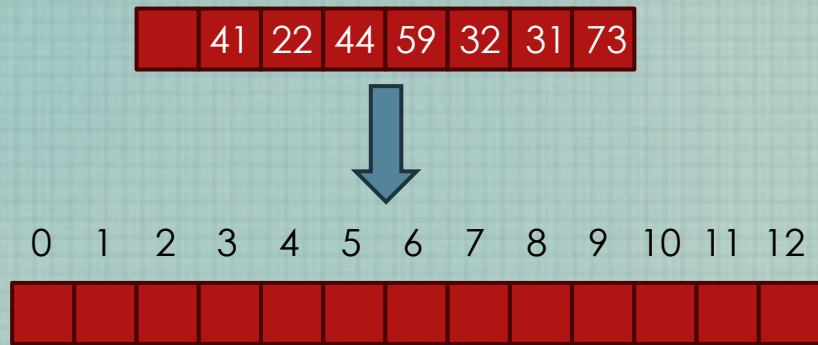
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes

Example of Double Hashing#

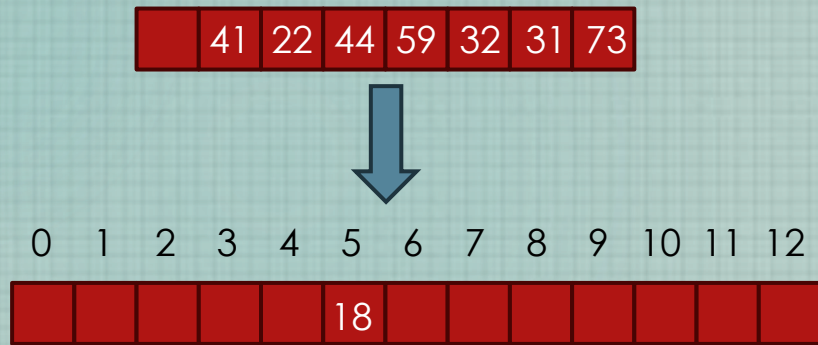
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5

Example of Double Hashing#

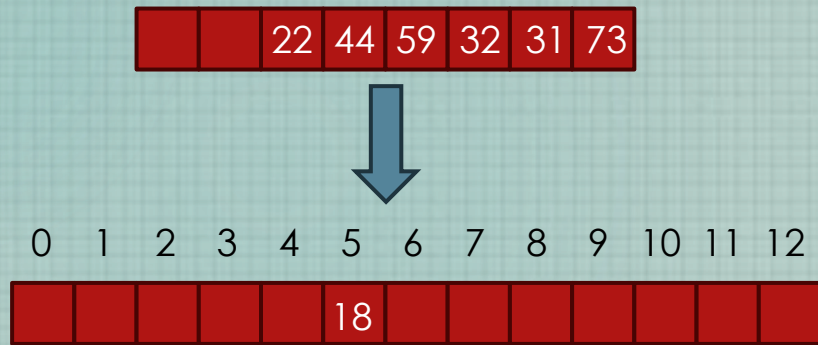
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5

Example of Double Hashing#

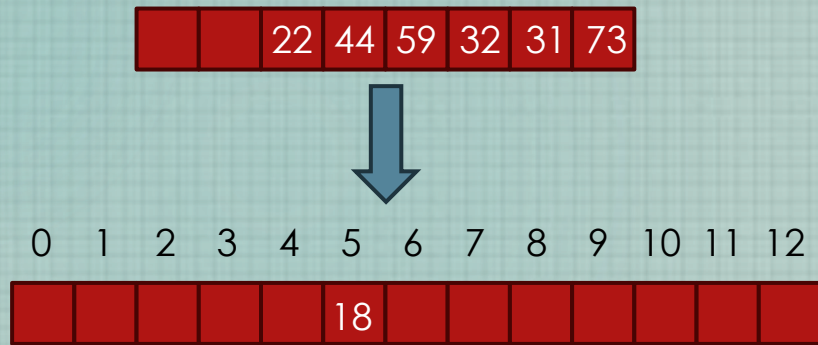
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5

Example of Double Hashing#

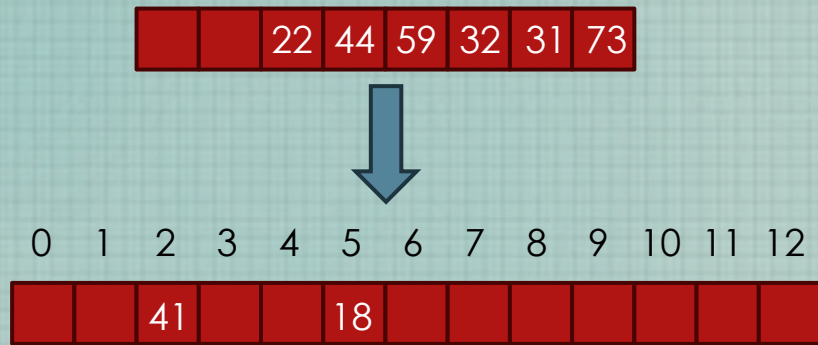
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2

Example of Double Hashing#

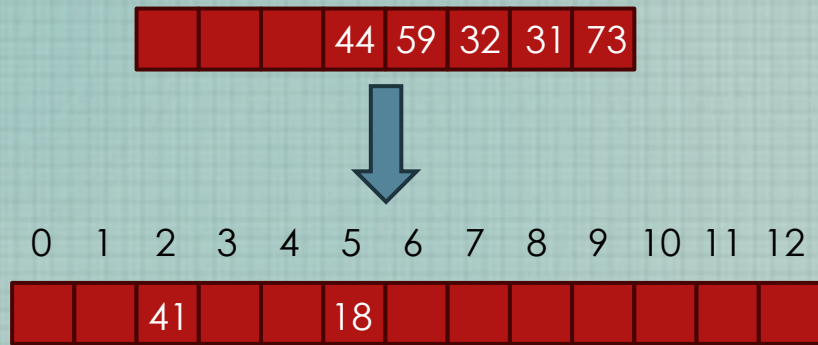
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2

Example of Double Hashing#

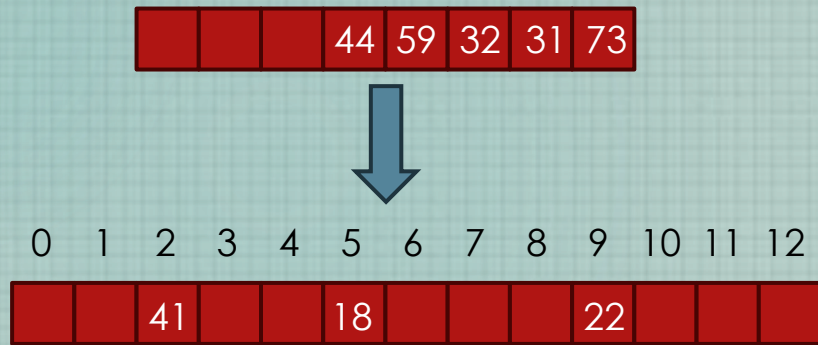
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9

Example of Double Hashing#

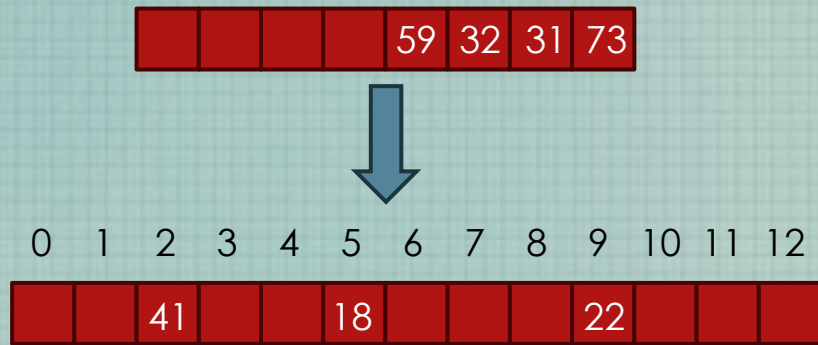
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9

Example of Double Hashing#

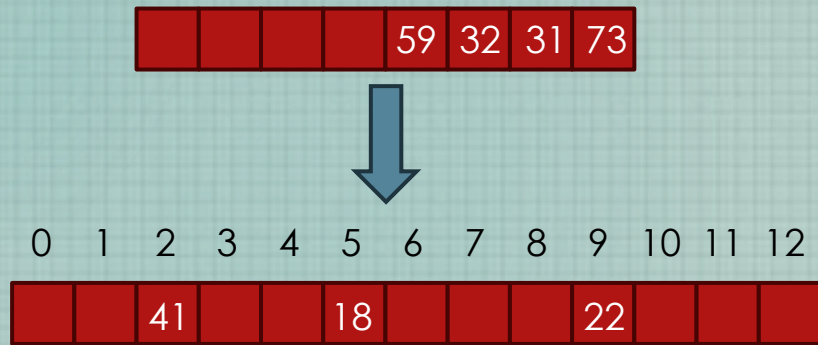
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9

Example of Double Hashing#

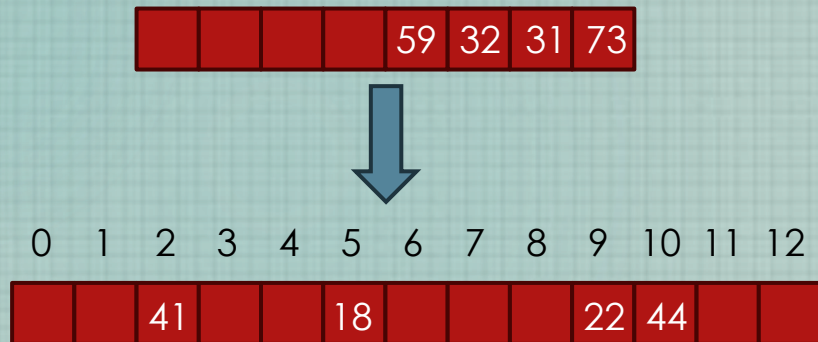
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10

Example of Double Hashing#

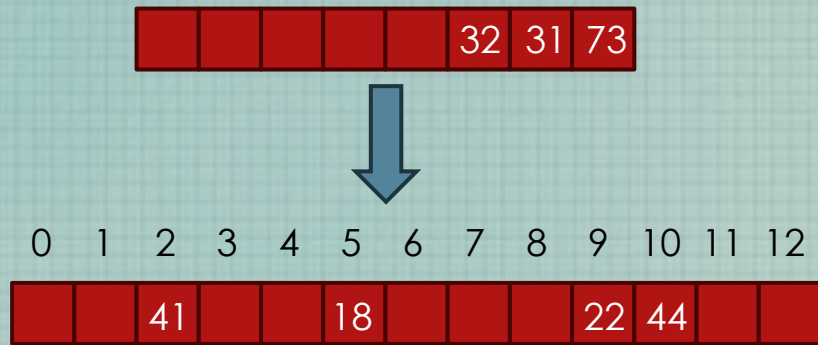
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10

Example of Double Hashing#

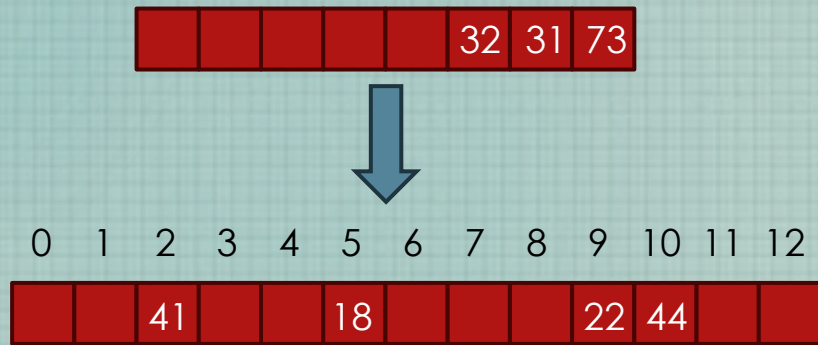
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10

Example of Double Hashing#

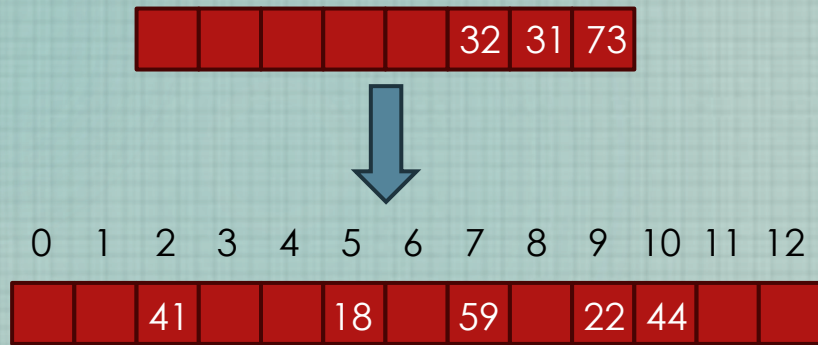
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7

Example of Double Hashing#

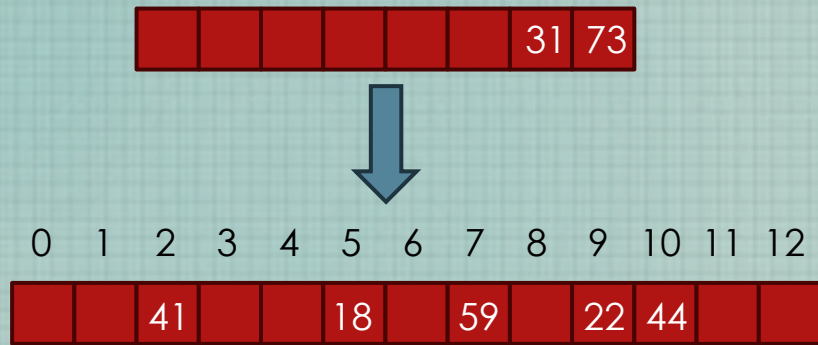
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7

Example of Double Hashing#

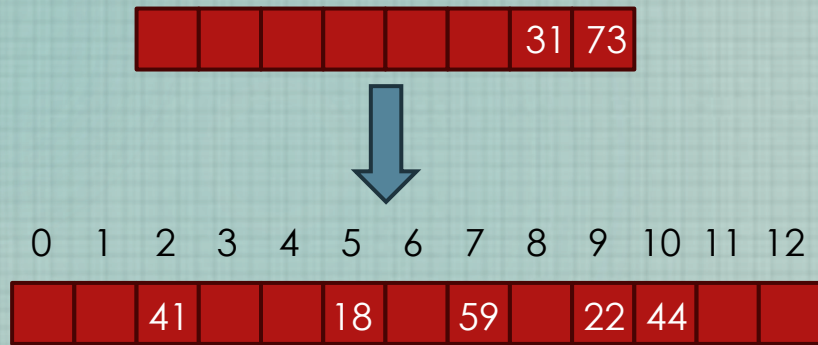
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7

Example of Double Hashing#

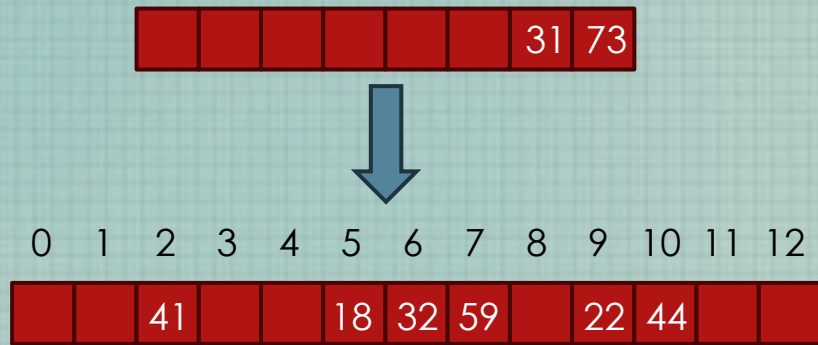
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6

Example of Double Hashing#

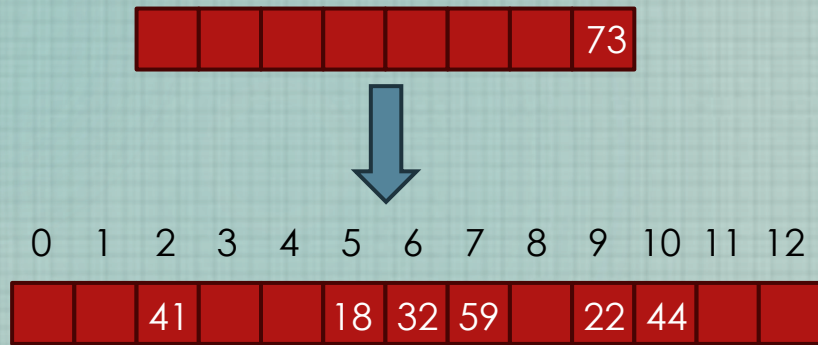
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6

Example of Double Hashing#

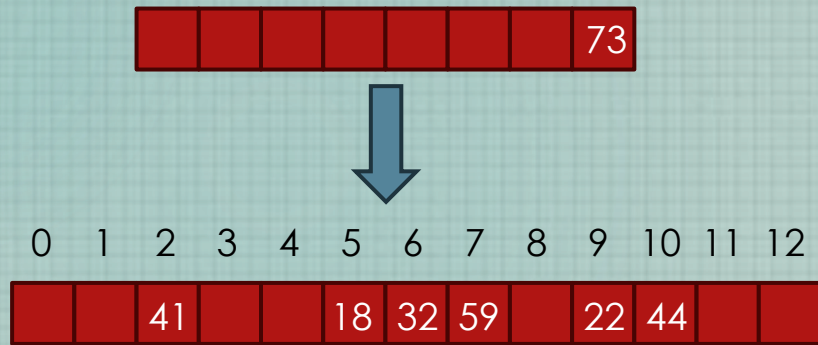
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6

Example of Double Hashing#

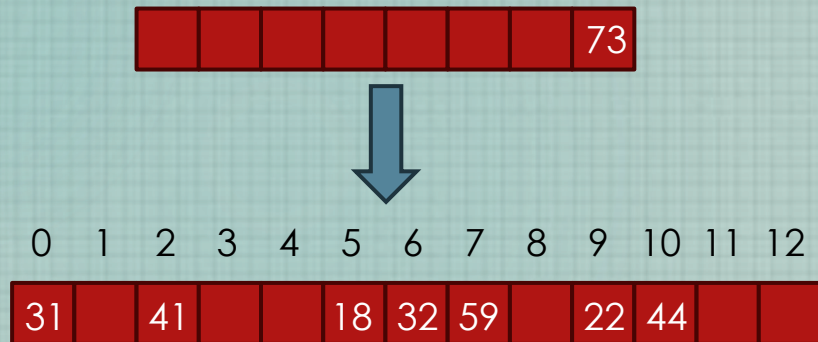
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6
31	5	4	9 0

Example of Double Hashing#

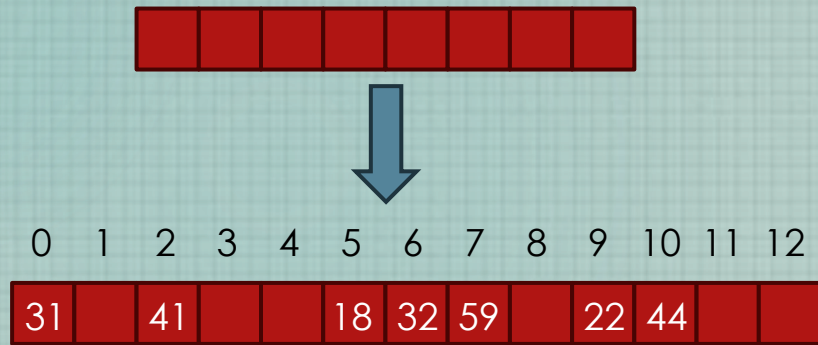
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6
31	5	4	9 0

Example of Double Hashing#

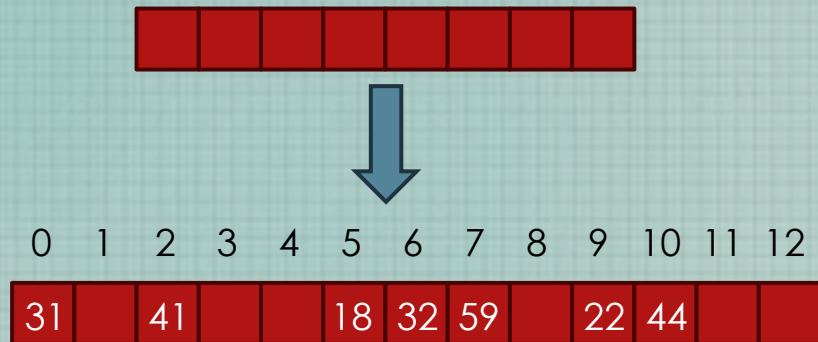
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6
31	5	4	9 0

Example of Double Hashing#

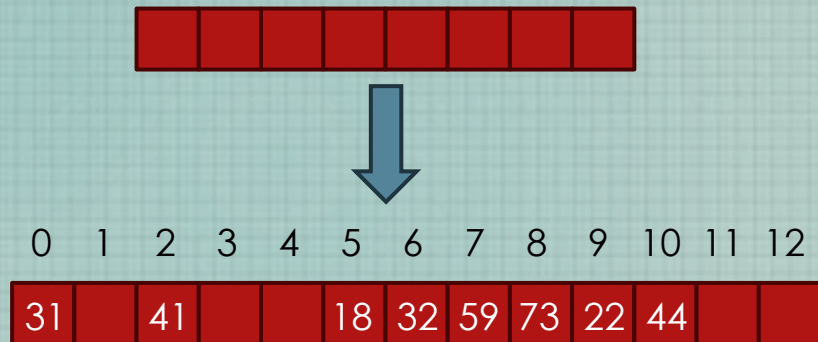
- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6
31	5	4	9 0
73	8	-	8

Example of Double Hashing#

- ▶ Consider a hash table storing integer keys that handles collision with double hashing
 - ▶ $N = 13$
 - ▶ $h(k) = k \% 13$
 - ▶ $h'(k) = 7 - k \% 7$
- ▶ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



k	h(k)	h'(k)	Probes
18	5	-	5
41	2	-	2
22	9	-	9
44	5	5	10
59	7	-	7
32	6	-	6
31	5	4	9 0
73	8	-	8

Conclusion on Double hashing#

- ▶ Calculate the $h(k)$, if the output never occurs in the probes list, using it as probes directly
- ▶ If there is collision, calculate $h'(k)$
- ▶ Using $h'(k)$ as **step size**, and adding on the $h(k)$ multiple times until find the available cell
- ▶ **$h'(k)$ cannot be used to obtain probe directly.**

Performance of Hashing#

- ▶ The worst case occurs when all the keys inserted into the map collide
- ▶ The load factor $\alpha = n/N$ affects the performance of a hash table
- ▶ Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is: $1 / (1 - \alpha)$
- ▶ In practice, hashing is very fast provided the load factor is not close to 100%
- ▶ **To keep hashing performance up, we need to keep the load factor a down**
 - ▶ Whenever load factor becomes too large (e.g., > 0.5)
 - ▶ Create new bucket array approx. double the size of the old one
 - ▶ Iterate all elements in the old bucket array and use put to add them to the new bucket array

Performance of Hashing*

- ▶ The worst case occurs when all the keys inserted into the map collide
- ▶ In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- ▶ (NOT TURE after 28th Feb 2024!)
- ▶ (NOT TURE after 28th Feb 2024!!)
- ▶ (NOT TURE after 28th Feb 2024!!!)
- ▶ IT IS $O((\log n)^2)$ for searching and insertion now (Funnel Hashing)

Performance of Hashing*



- ▶ In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- ▶ Proposed By Andrew Yao, 1985
- ▶ Turing Award winner 2000



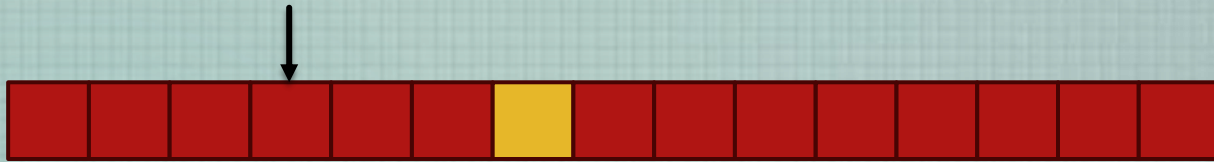
- ▶ No, it should be $O((\log n)^2)$ in the worst case for searching and insertion
- ▶ Proposed By Andrew Krapivin, 2025
- ▶ Undergraduate student from Cambridge

Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ The array A is broken into disjoint arrays $A_1 \oplus A_2 \oplus A_3 \oplus \dots \oplus A_k$, where each subsequent array A_{i+1} is approximately half the size of A_i
- ▶ If A_i is sufficiently empty, the element is placed there.
- ▶ Otherwise, the element is placed in A_{i+1}

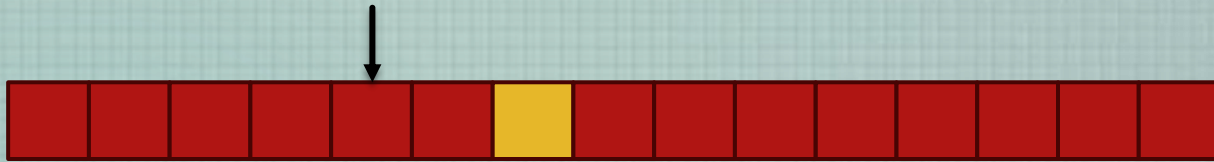
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ How does it work? For open addressing based hashing, when the hash table is almost full, we need to scan the whole table to find the available cell



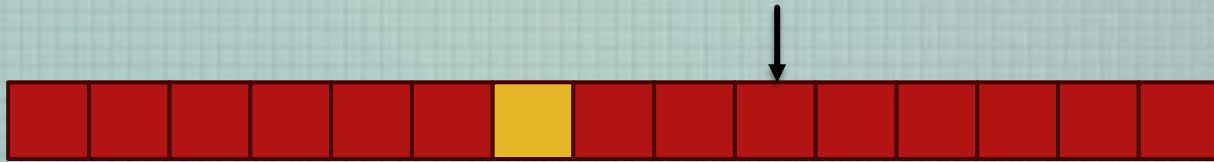
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ How does it work? For open addressing based hashing, when the hash table is almost full, we need to scan the whole table to find the available cell



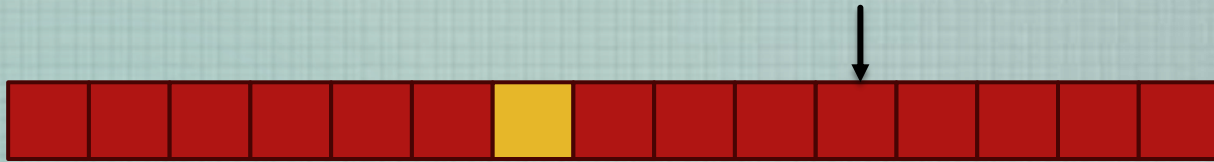
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ How does it work? For open addressing based hashing, when the hash table is almost full, we need to scan the whole table to find the available cell



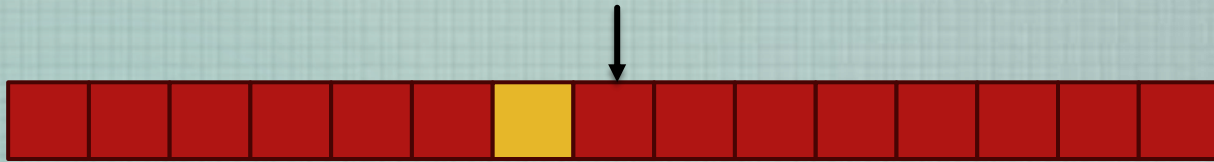
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ How does it work? For open addressing based hashing, when the hash table is almost full, we need to scan the whole table to find the available cell



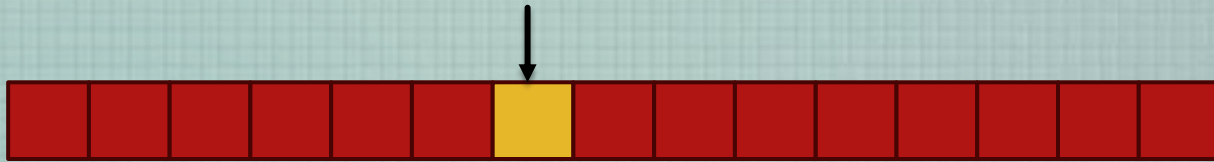
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ How does it work? For open addressing based hashing, when the hash table is almost full, we need to scan the whole table to find the available cell



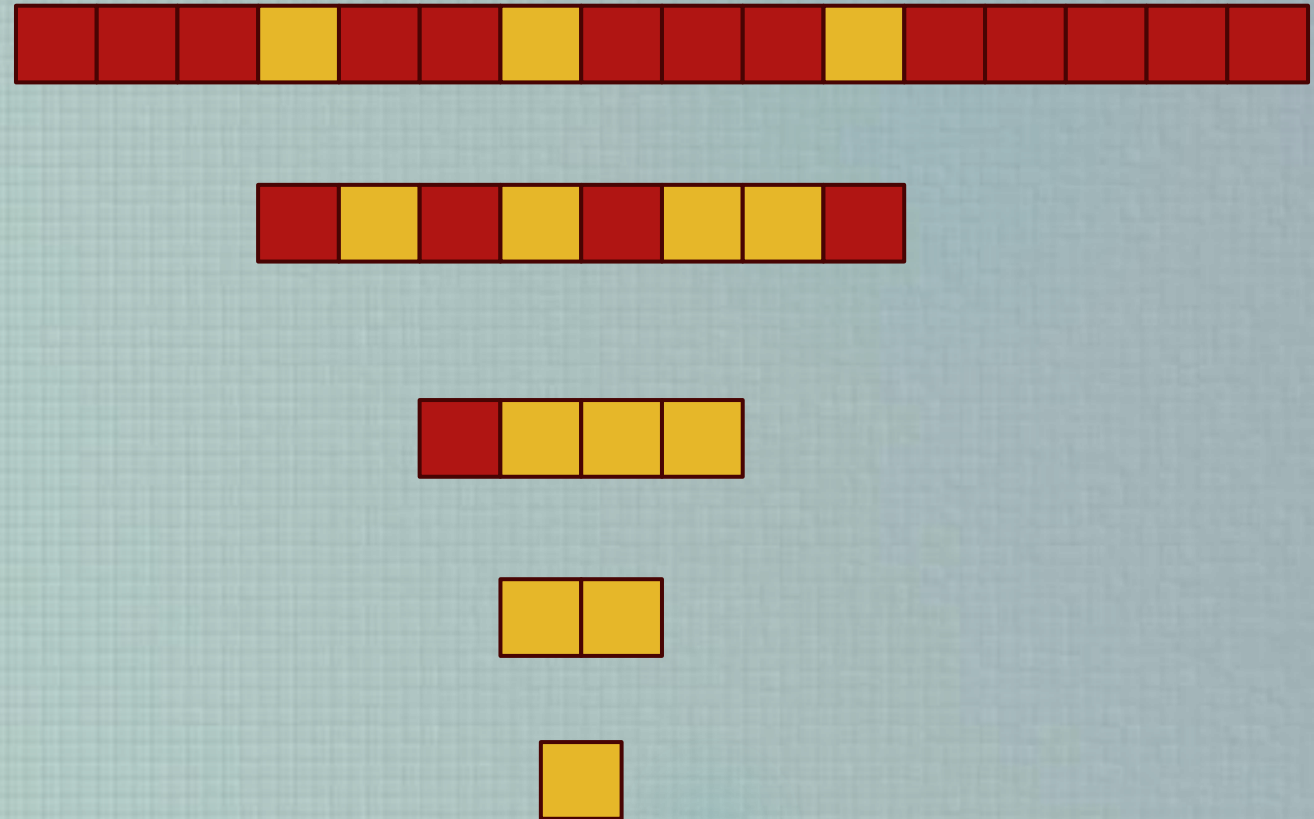
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ How does it work? For open addressing based hashing, when the hash table is almost full, we need to scan the whole table to find the available cell



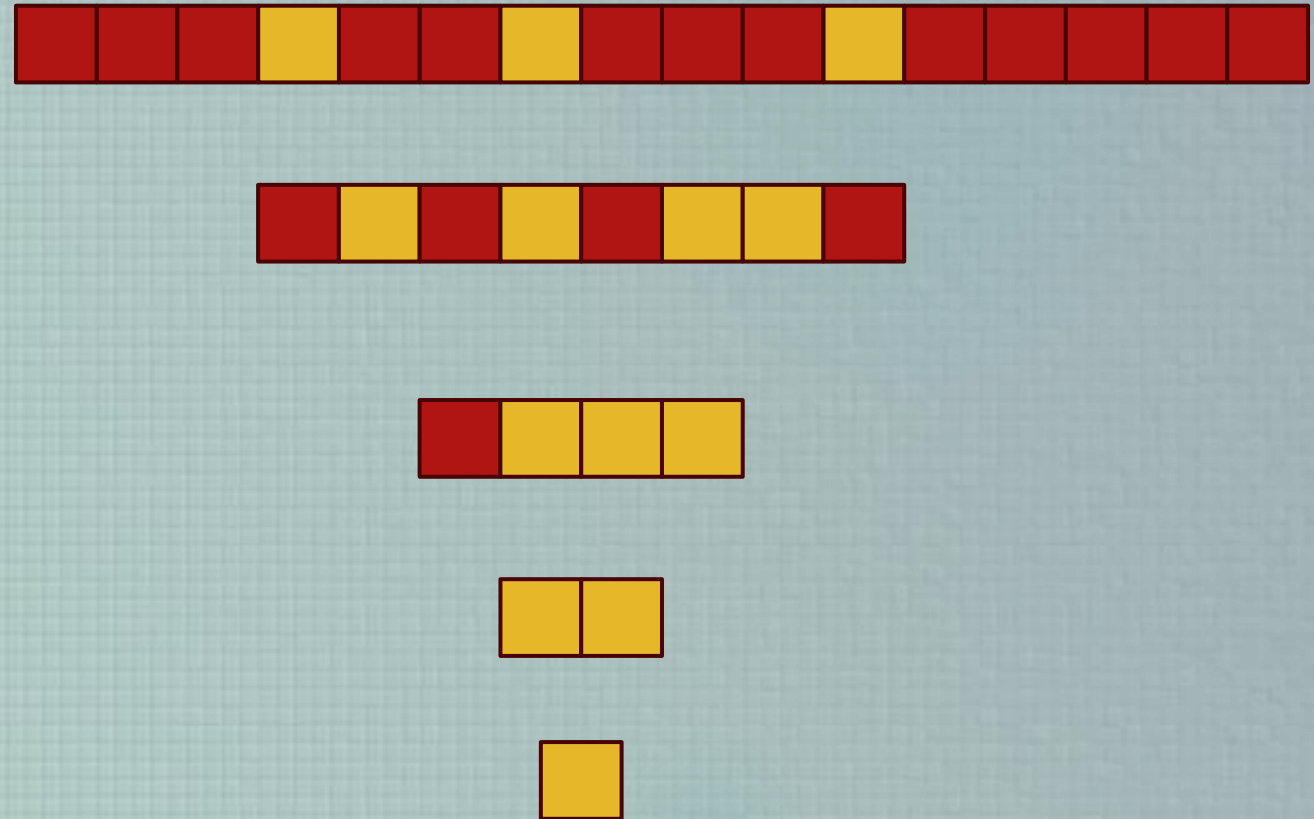
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer



Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer



Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer



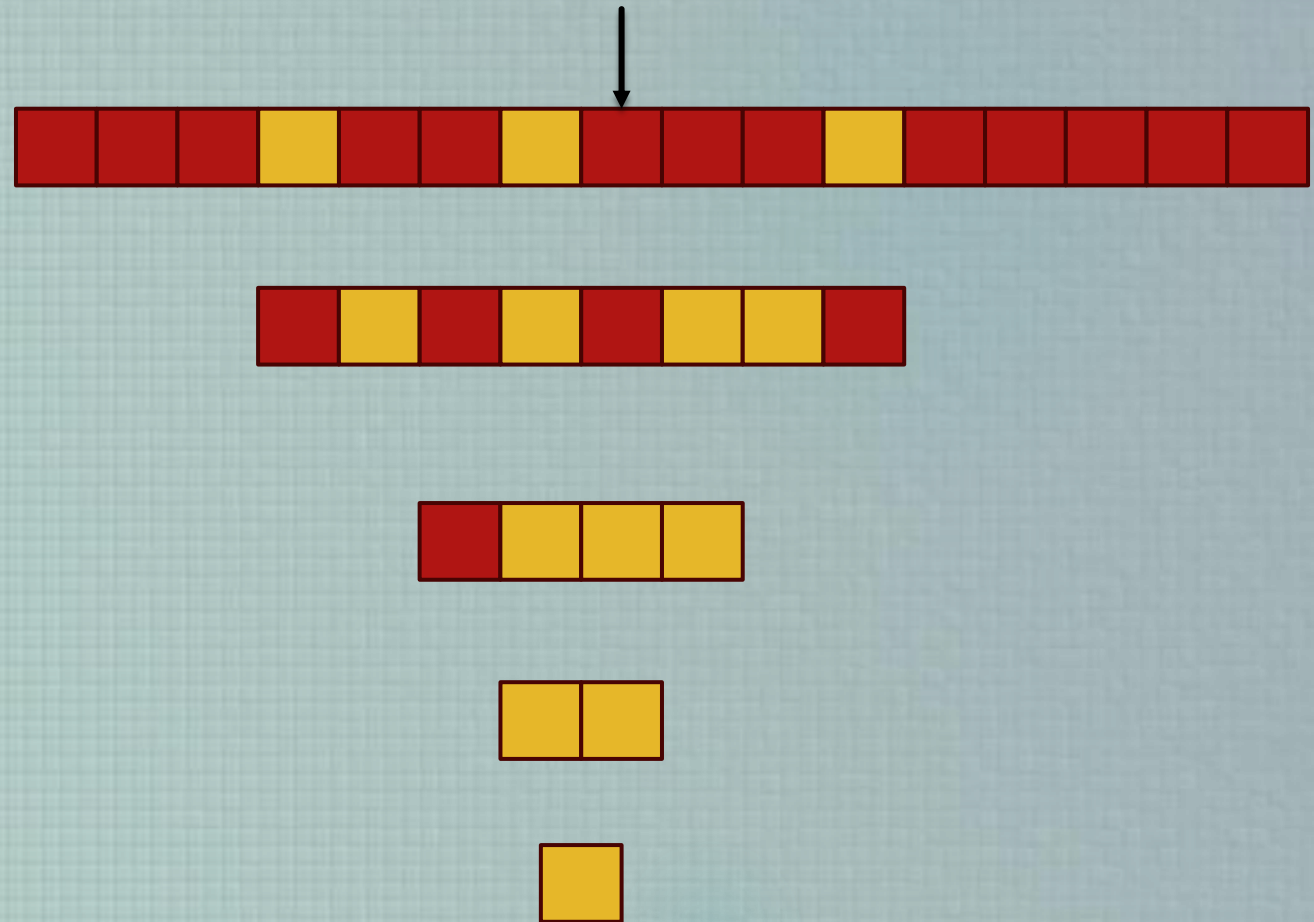
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full



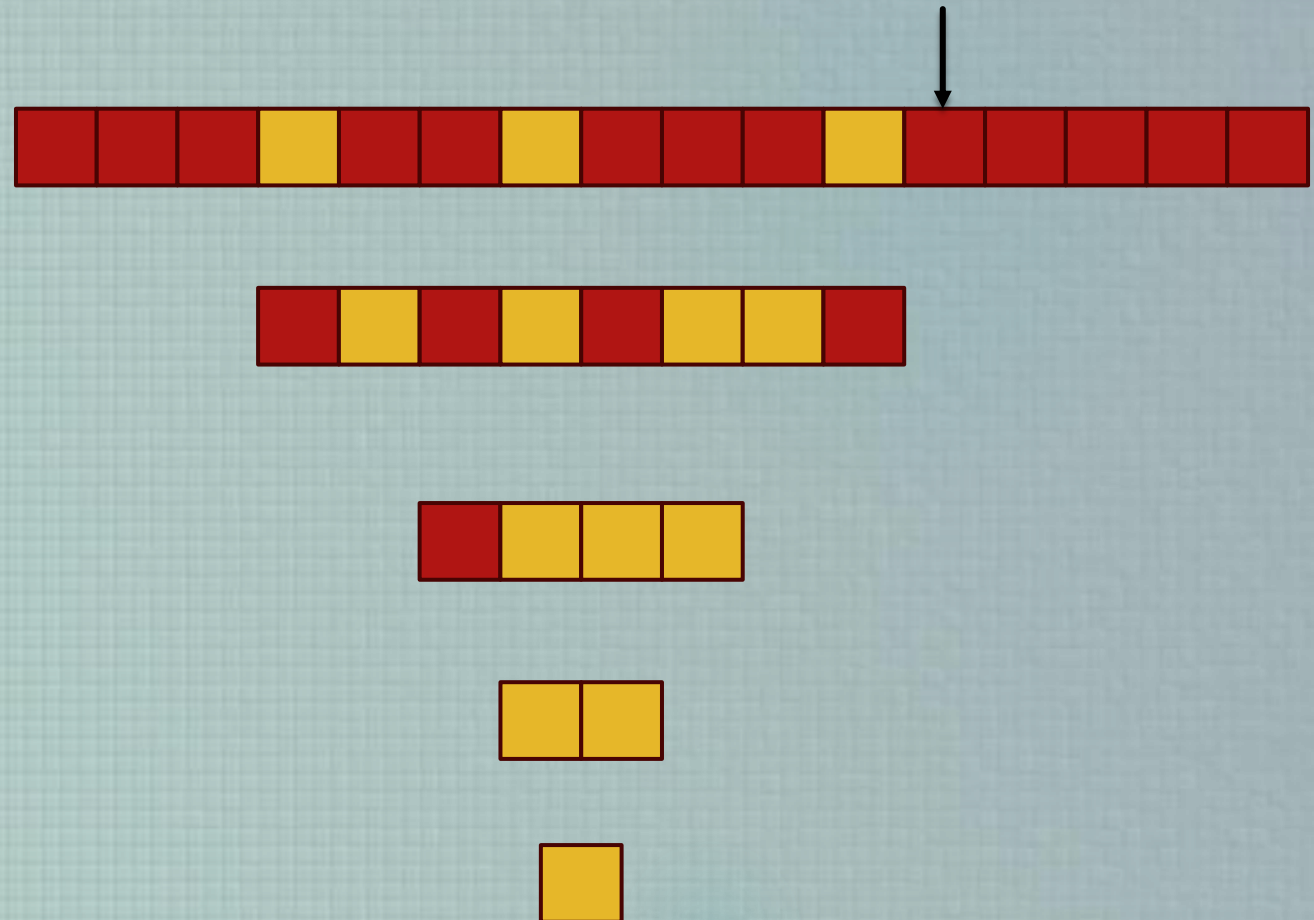
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full



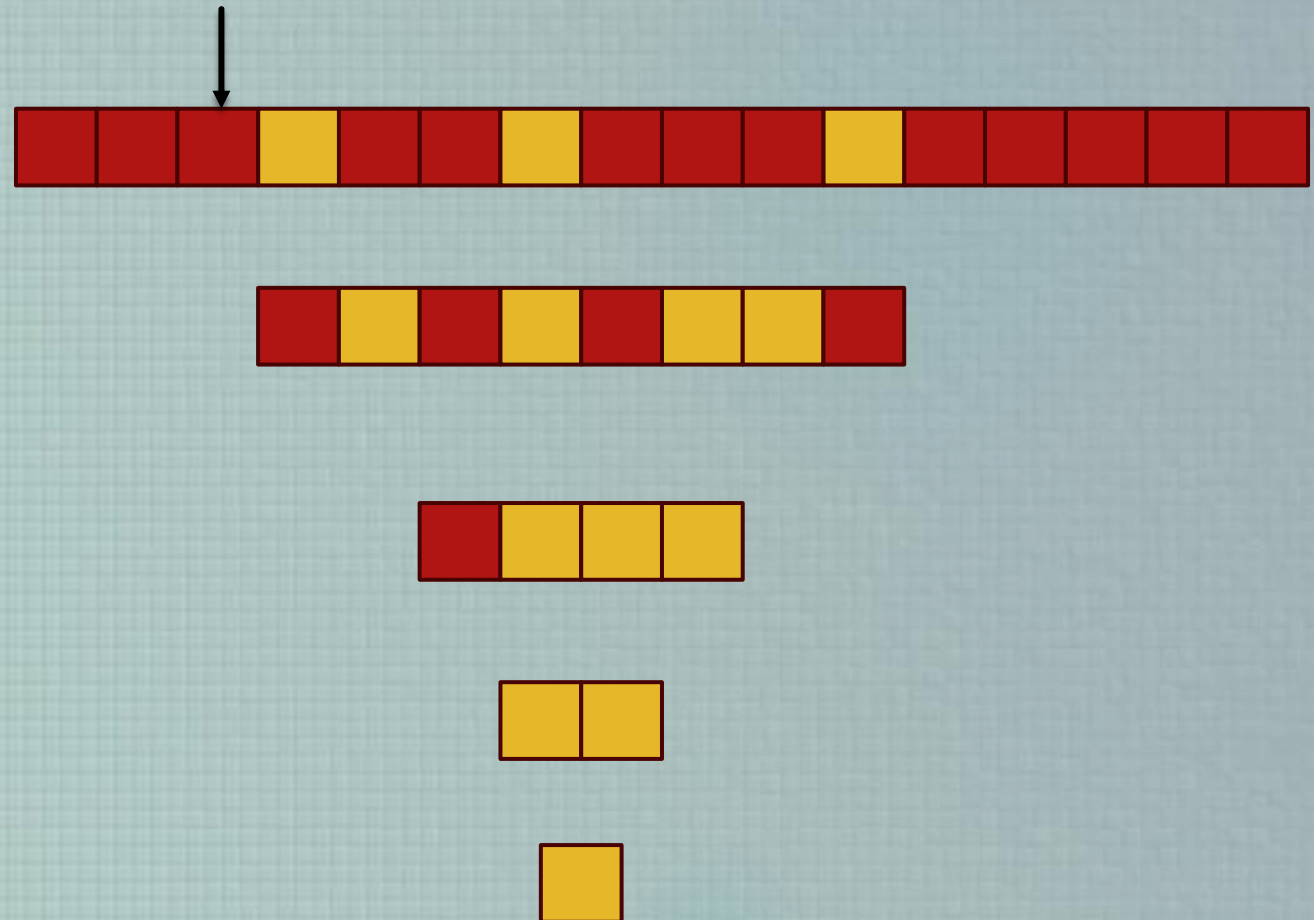
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full



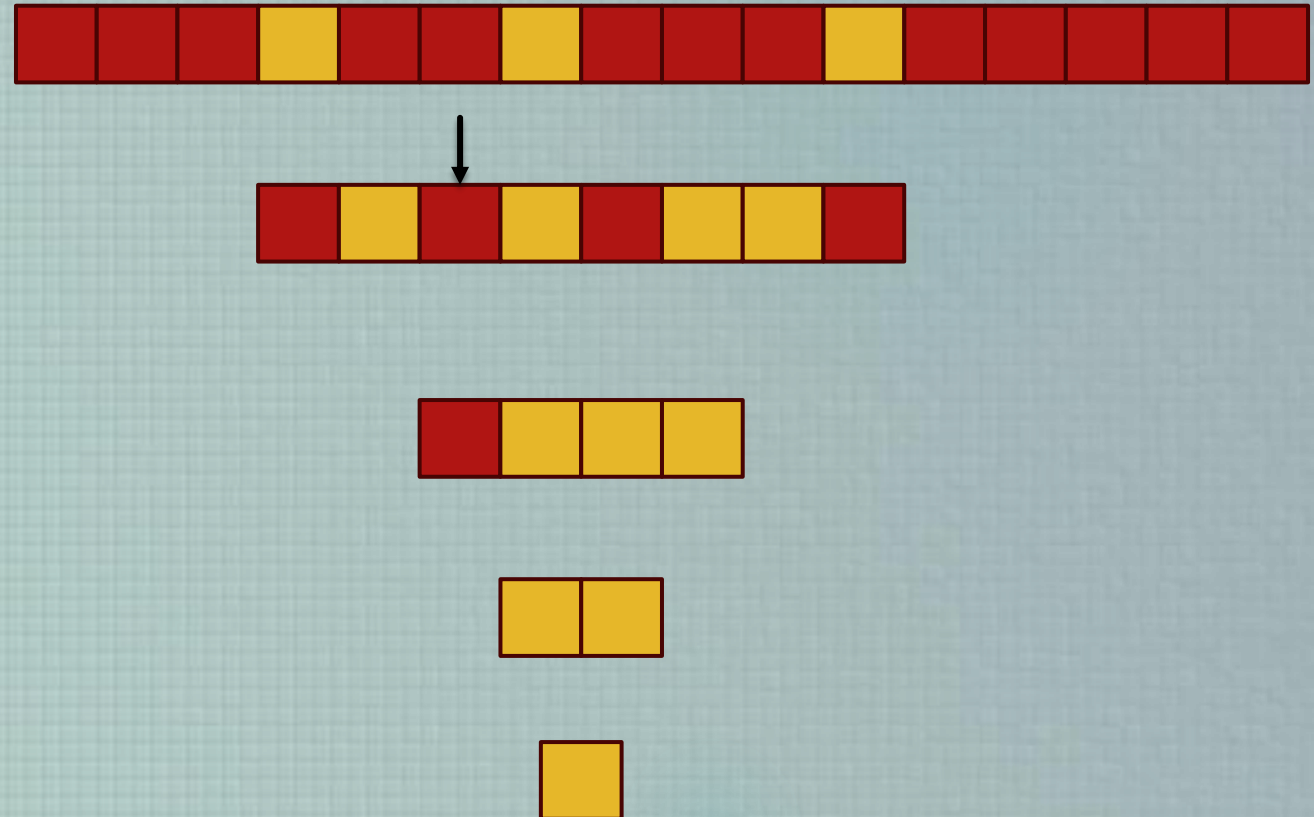
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full



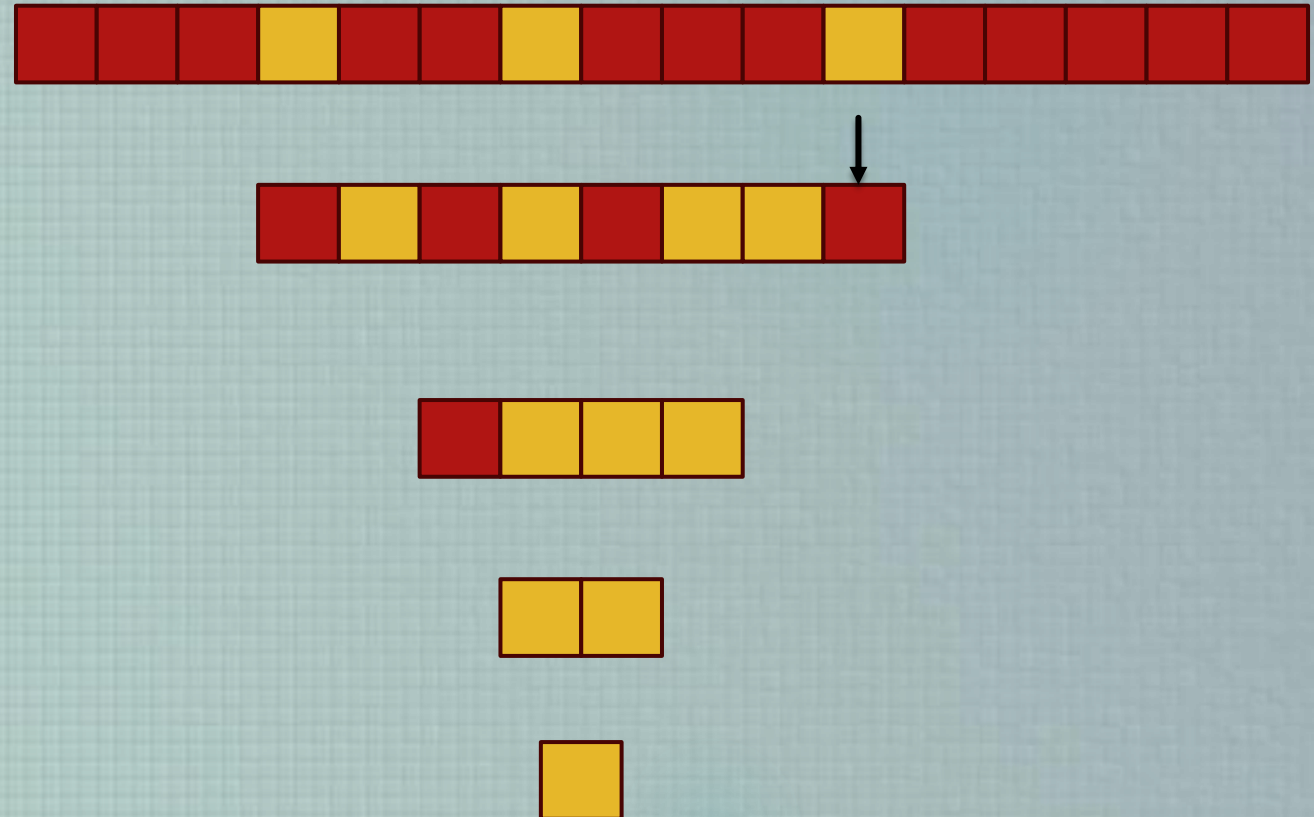
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full



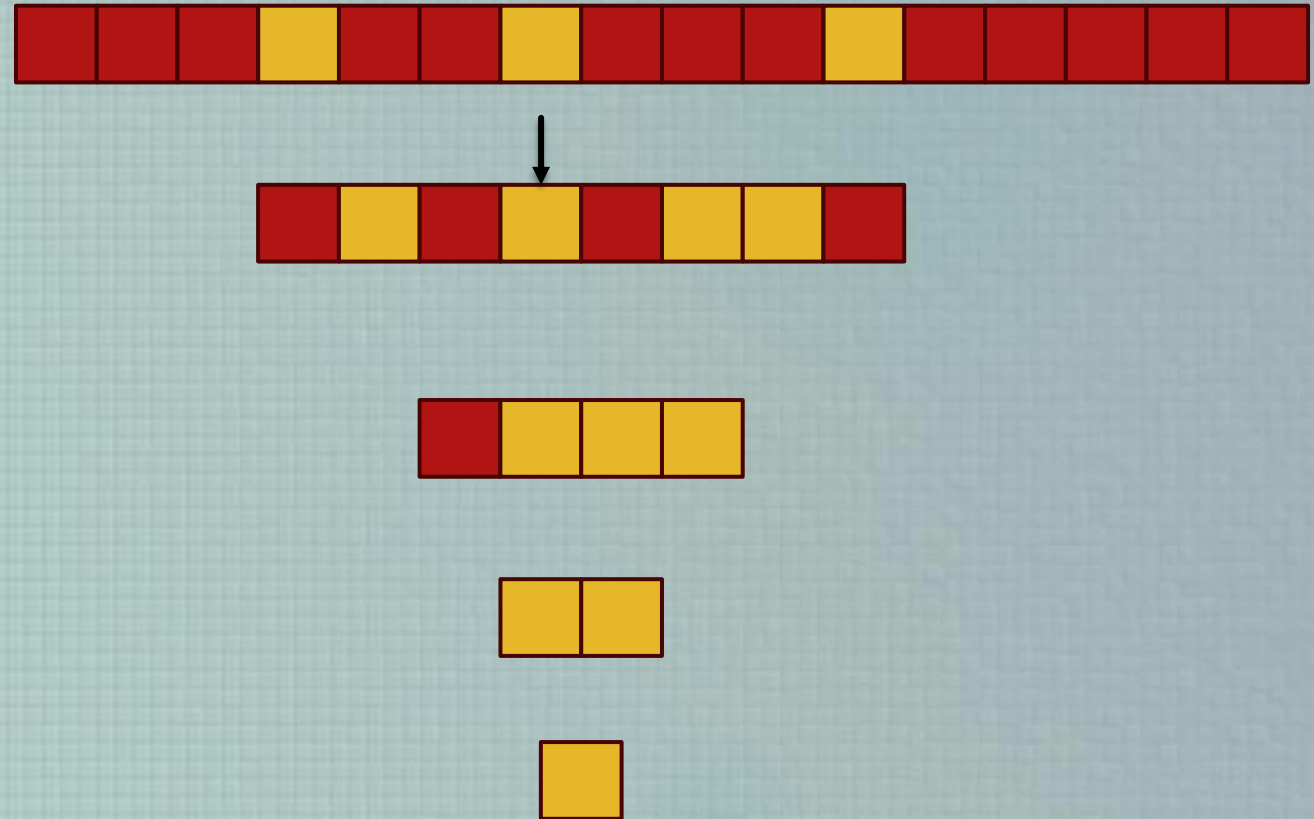
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full



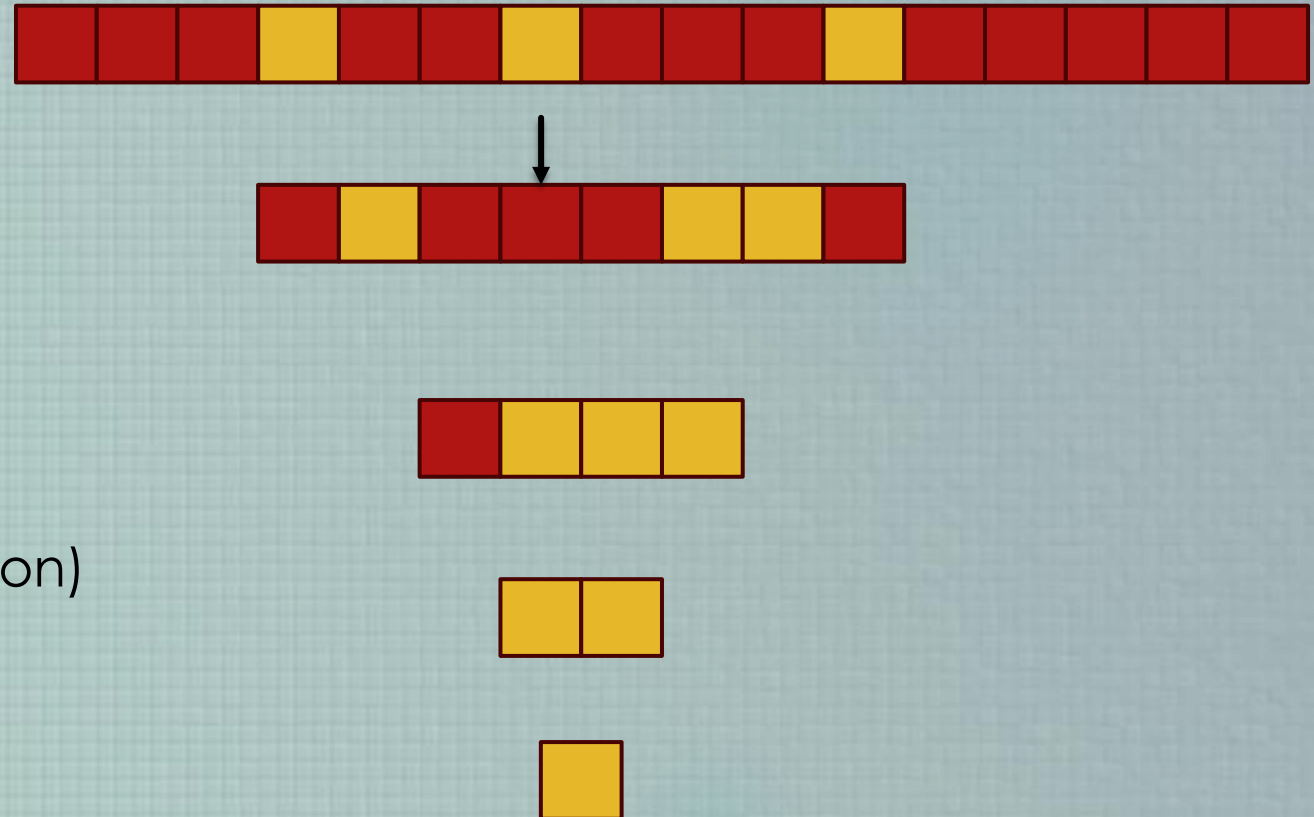
Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full



Performance of Hashing*

- ▶ **IT IS $O((\log n)^2)$ for searching and insertion now (Funnel/Elastic Hashing)**
- ▶ In Andrew's method, the table is separated into K disjoint sub array in k layers. The size of array is about 50% of the array in previous layer
- ▶ The empty cell often appears in the bottom layer
- ▶ Try to insert in the upper layer t times
 - ▶ If it works, insert the element
 - ▶ If all failed, move to the next layer and try again
- ▶ Skip those layers which are already full
- ▶ Find Andrew's talk [here](#)
- ▶ Find the implementation [here](#) (python version)



Outline

- ▶ Maps
 - ▶ Map ADT
 - ▶ List-Based Map Implementation#
- ▶ Hash Tables
 - ▶ Bucket Array and Hash Functions
 - ▶ Collision Handling#
- ▶ Dictionaries
 - ▶ Dictionary ADT
 - ▶ Dictionary Implementations
- ▶ Applications

Dictionary ADT

- ▶ The dictionary ADT models a searchable collection of key-value entries
- ▶ The main operations of a dictionary are searching, inserting, and deleting items
- ▶ **Multiple items with the same key are allowed (different from hash table)**

Dictionary ADT

- ▶ Dictionary ADT methods:
 - ▶ **get(k)**: if the dictionary D has an entry with key k, returns it, else, returns null
 - ▶ **getAll(k)**: returns an iterable collection of all entries with key k
 - ▶ **put(k, v)**: inserts into the dictionary D and returns the entry (k, v)
 - ▶ **remove(e)**: removes the entry e from the dictionary and returns the removed entry; an error occurs if entry is not in the dictionary D
 - ▶ **entrySet()**: returns an iterable collection of the entries in the dictionary
 - ▶ **size()**: returns the current size of the dictionary
 - ▶ **isEmpty()**: returns True/False if the dictionary is empty/non-empty

Example

Operation
put(5,A)

Output
(5,A)

Dictionary
(5,A)

Example

Operation

put(5,A)

put(7,B)

Output

(5,A)

(7,B)

Dictionary

(5,A)

(5,A),(7,B)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

Output

(5,A)

(7,B)

(2,C)

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

Output

(5,A)

(7,B)

(2,C)

(8,D)

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

get(7)

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

(7,B)

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

get(7)

get(4)

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

(7,B)

null

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

get(7)

get(4)

get(2)

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

(7,B)

null

(2,C)

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

get(7)

get(4)

get(2)

getAll(2)

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

(7,B)

null

(2,C)

(2,C),(2,E)

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

get(7)

get(4)

get(2)

getAll(2)

size()

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

(7,B)

null

(2,C)

(2,C),(2,E)

5

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

get(7)

get(4)

get(2)

getAll(2)

size()

remove(get(5))

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

(7,B)

null

(2,C)

(2,C),(2,E)

5

(5,A)

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(7,B),(2,C),(8,D),(2,E)

Example

Operation

put(5,A)

put(7,B)

put(2,C)

put(8,D)

put(2,E)

get(7)

get(4)

get(2)

getAll(2)

size()

remove(get(5))

get(5)

Output

(5,A)

(7,B)

(2,C)

(8,D)

(2,E)

(7,B)

null

(2,C)

(2,C),(2,E)

5

(5,A)

null

Dictionary

(5,A)

(5,A),(7,B)

(5,A),(7,B),(2,C)

(5,A),(7,B),(2,C),(8,D)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

(5,A),(7,B),(2,C),(8,D),(2,E)

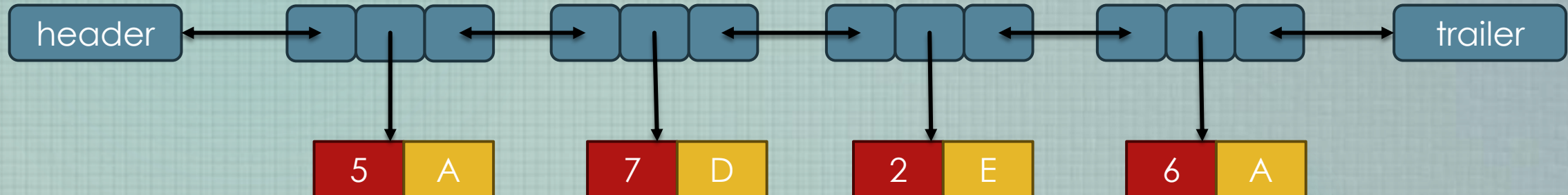
(5,A),(7,B),(2,C),(8,D),(2,E)

(7,B),(2,C),(8,D),(2,E)

(7,B),(2,C),(8,D),(2,E)

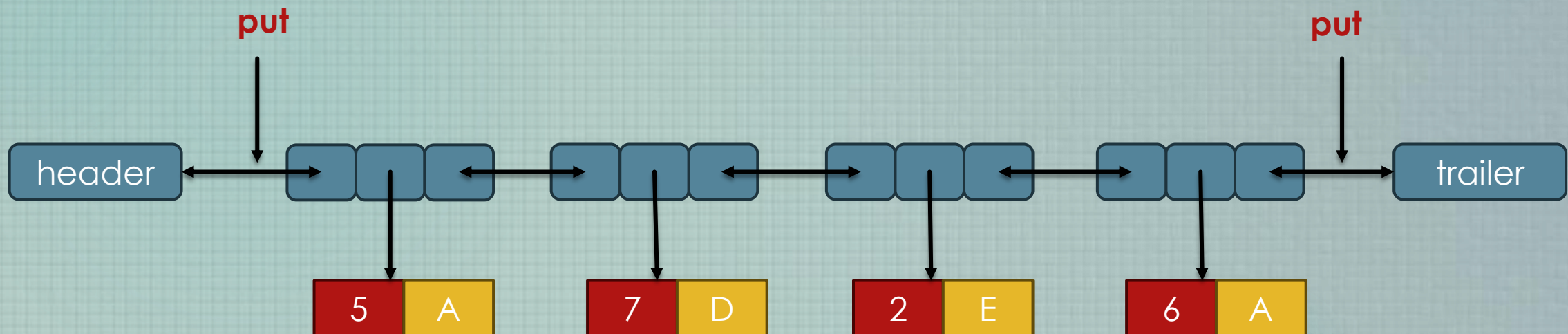
Implementation

- ▶ Dictionary can be implemented by an unsorted sequence
 - ▶ We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- ▶ Performance:
 - ▶ **put** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
 - ▶ **get** and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key



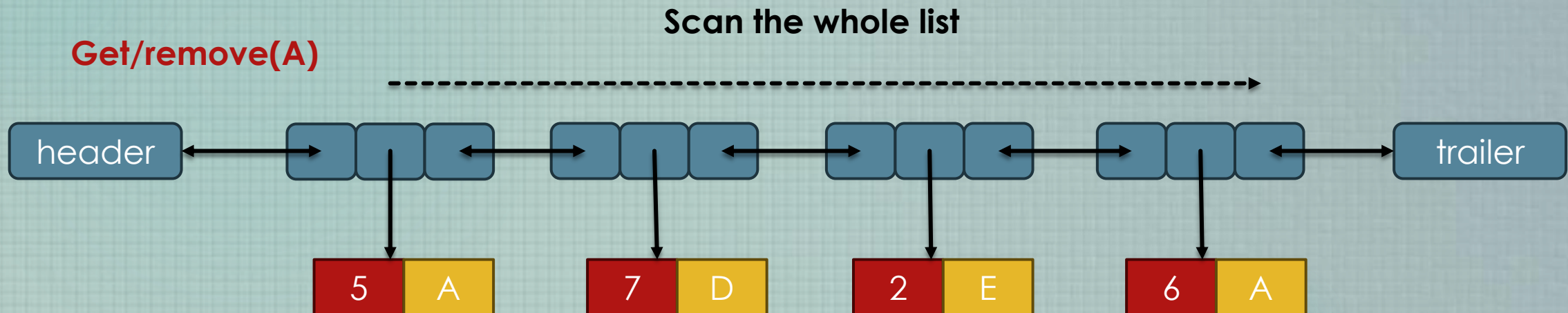
Implementation

- ▶ Dictionary can be implemented by an unsorted sequence
 - ▶ We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- ▶ Performance:
 - ▶ **put** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
 - ▶ **get** and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key



Implementation

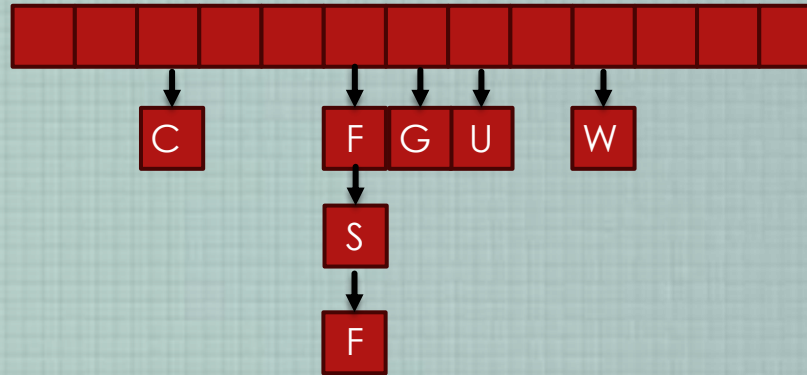
- ▶ Dictionary can be implemented by an unsorted sequence
 - ▶ We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- ▶ Performance:
 - ▶ **put** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
 - ▶ **get** and **remove** take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key



Implementation

- ▶ Using separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

$$h(k) = (k - 97) \% 13$$



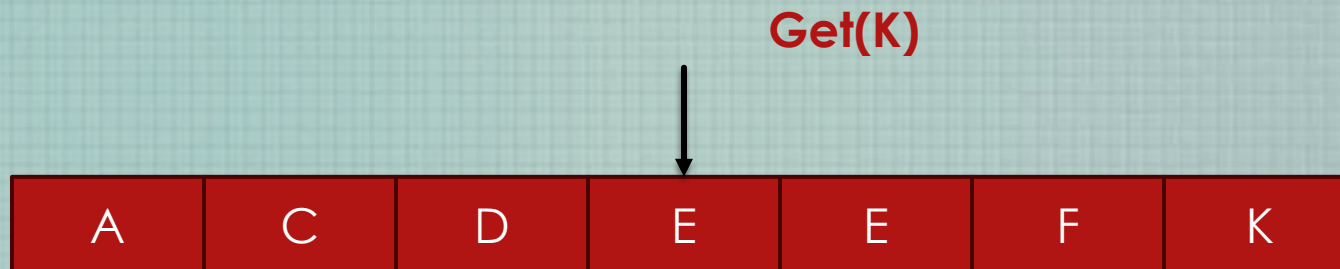
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal

A	C	D	E	E	F	K
---	---	---	---	---	---	---

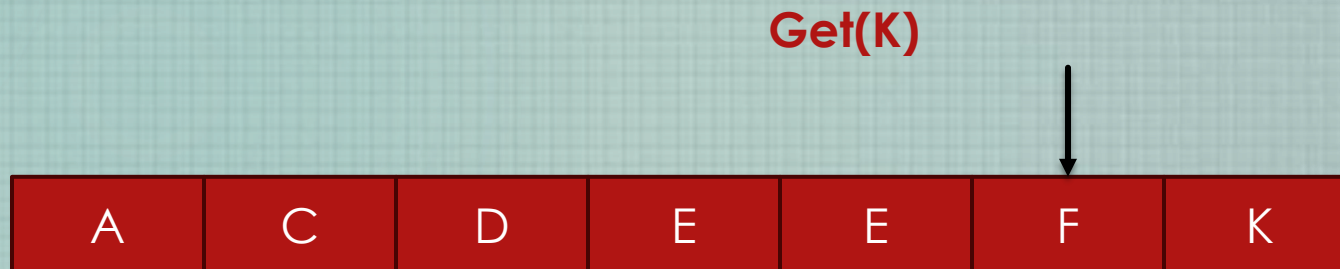
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



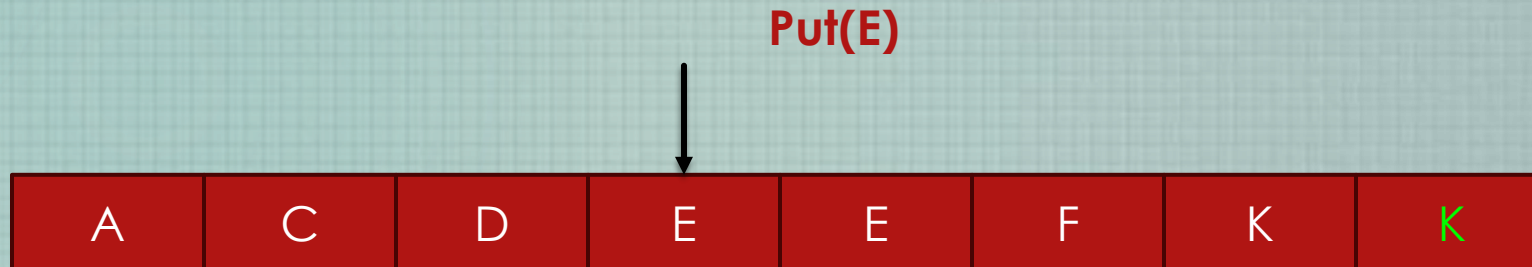
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



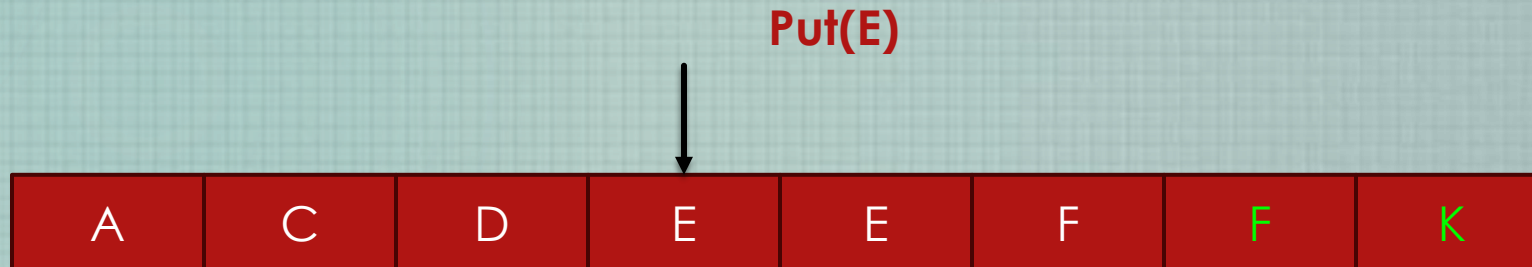
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



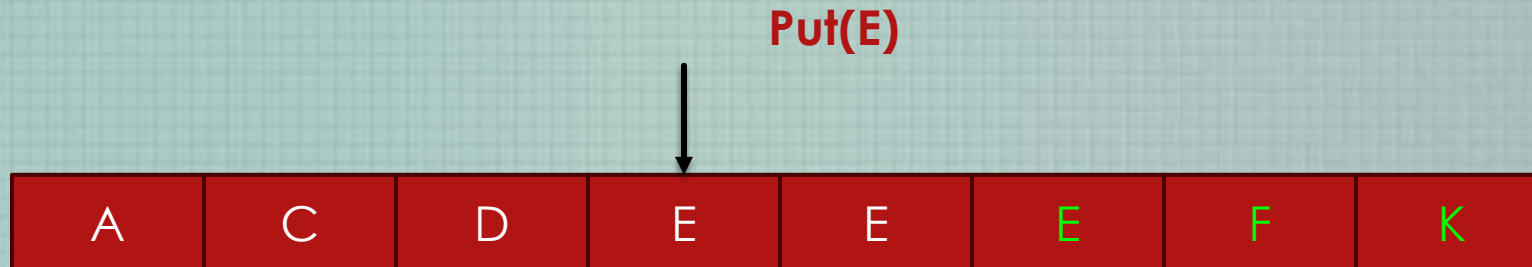
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



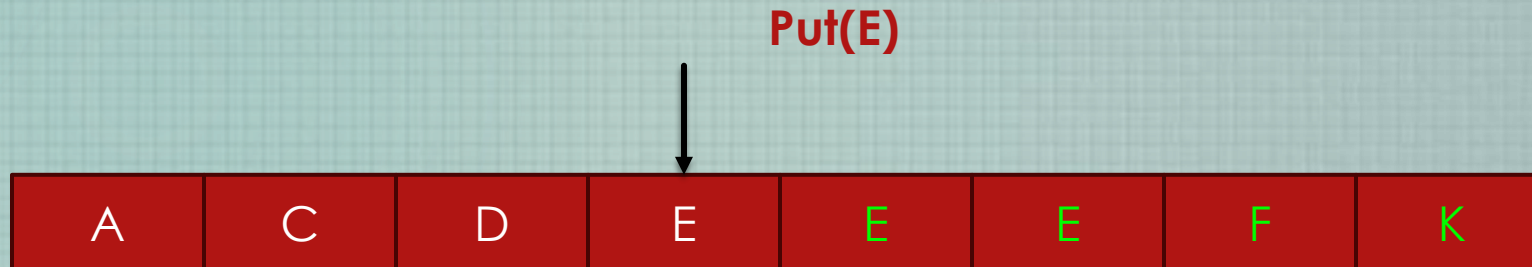
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



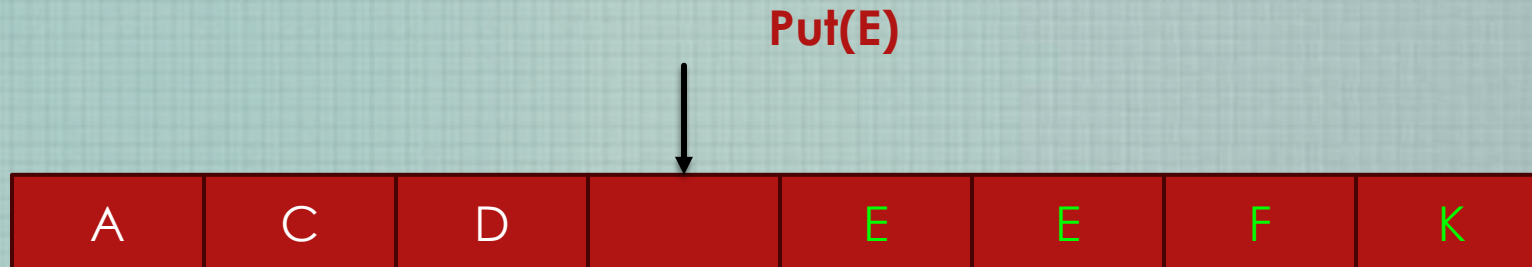
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



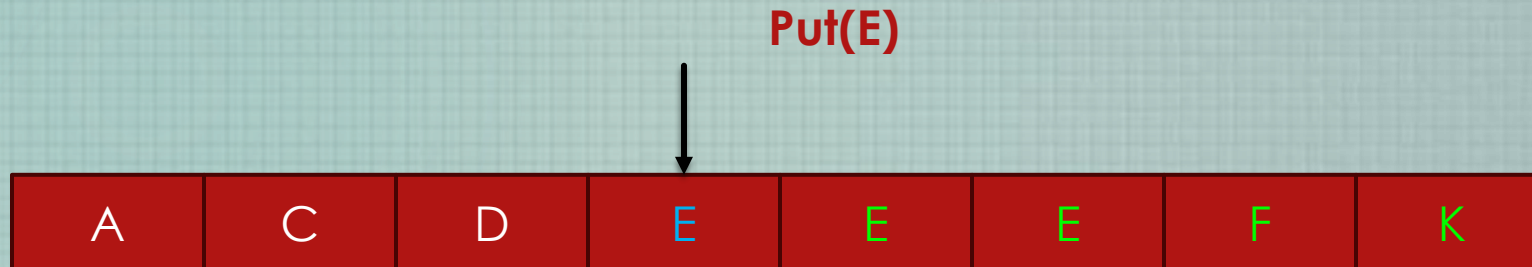
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



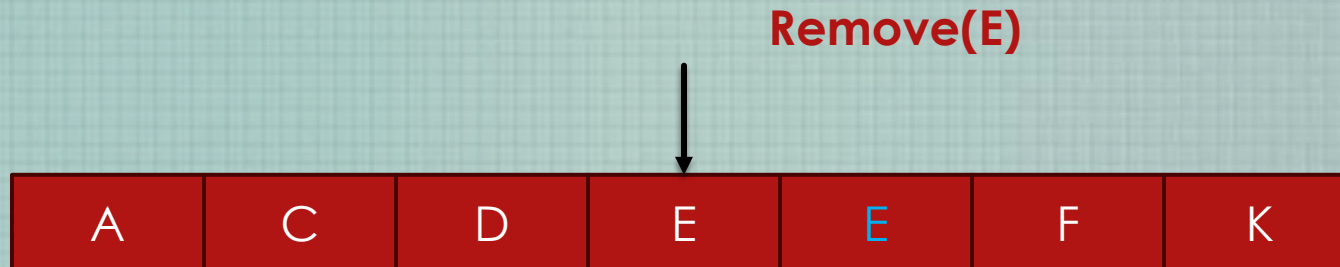
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



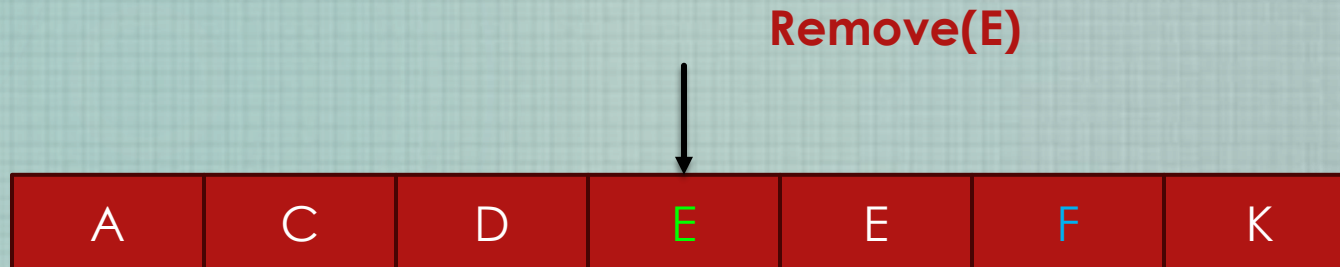
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



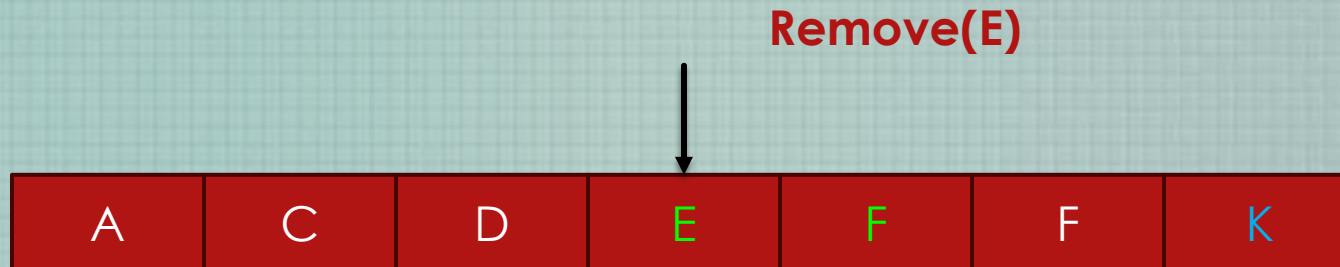
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



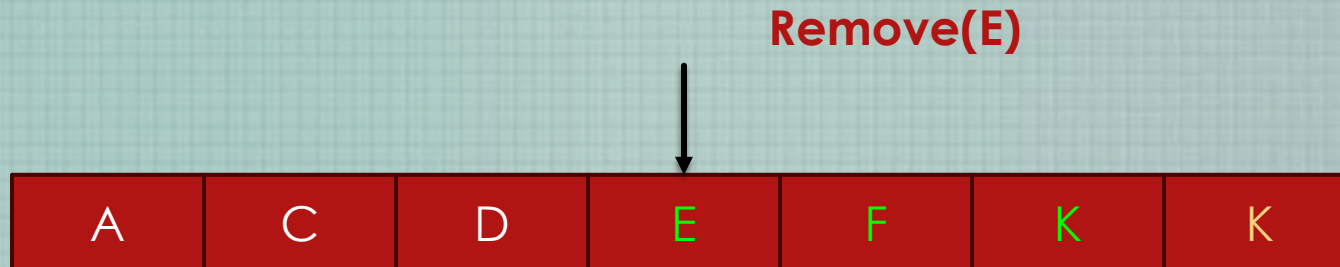
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



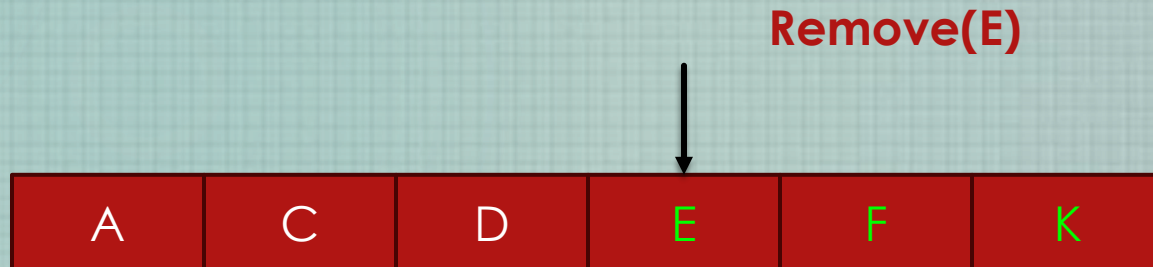
Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



Implementation

- ▶ A search table is a dictionary implemented by means of a sorted (ordered) array
 - ▶ We store the items of the dictionary in an array-based sequence, sorted by key
 - ▶ We use an external comparator for the keys
- ▶ Performance:
 - ▶ **get** takes $O(\log n)$ time, using binary search
 - ▶ **put** takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - ▶ **remove** takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal



Implementation

- ▶ A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed

Outline

- ▶ Maps
 - ▶ Map ADT
 - ▶ List-Based Map Implementation#
- ▶ Hash Tables
 - ▶ Bucket Array and Hash Functions
 - ▶ Collision Handling#
- ▶ Dictionaries
 - ▶ Dictionary ADT
 - ▶ Dictionary Implementations
- ▶ Applications

Application*

- ▶ Find the highest frequent word from document
- ▶ Read each word from the text.
- ▶ Check if the word is in the hashtable:
 - ▶ If 'yes', the corresponding value +1;
 - ▶ If 'no', create a new pair <word, 1> and insert into the hash table
- ▶ Sort the hash table based on the value and find the word with the highest frequency.

Application*

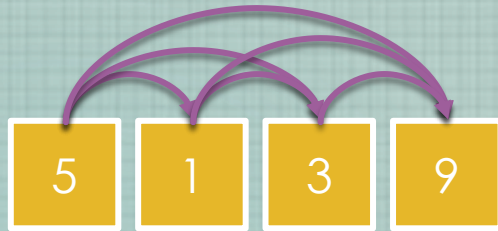
- ▶ Find the duplicated documents in large amount of documents
- ▶ Question: what if the key is not a number?*
- ▶ Solution: MD5 hashing
- ▶ And input can be mapped into a fixed-length hash (128 bits, or 16 bytes)
- ▶ For example:
- ▶ `md5("abc") = 900150983cd24fb0d6963f7d28e17f72`
- ▶ `md5("Hello, World!") = fc3ff98e8c6a0d3087d515c0473f8677`
- ▶ `md5("") = d41d8cd98f00b204e9800998ecf8427e`
- ▶ Finding the duplicated documents;
 - ▶ Using MD5 hashing, if the key never appeared before, set the value = 1.
 - ▶ Otherwise, it should be a duplicated document.

Application

For a given array and a given number t , to determine if there exists two int variables, i and j , where $i + j = t$.



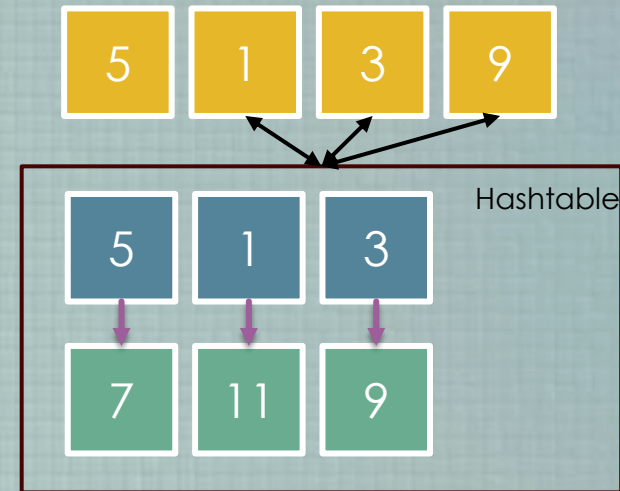
$t = 12$



```
for i ← 0 to n-1, do:  
  for j ← i+1 to n, do:  
    if ( $t == A[i] + A[j]$ ){  
      return  $A[i], A[j]$ ;
```

The **Auxiliary space** used is $O(1)$

The running time is $O(n^2)$



```
for i ← 0 to n, do:  
  if (hash.get( $t - A[i]$ )){  
    hash.put( $t - A[i]$ );  
  }  
  else{  
    return  $A[i], t - A[i]$ ;
```

The **Auxiliary space** used is $O(n) \uparrow$

The running time is $O(n) \downarrow$

Conclusion of LGT

- ▶ The definition of Maps, Hash table, and dictionary#
 - ▶ The difference between hash table and dictionary#
 - ▶ How to deal with hash collision, especially for the linear probing and double hashing. You should be able to solve the question in our mock test#
 - ▶ Using linked list to implement maps? #
 - ▶ Applications of hashing*
 - ▶ Worst case of hashing algorithm has been updated, not $O(n)$ anymore *
-
- ▶ Data Structure Visualisation - <https://visualgo.net/en/hashtable>

Thank you

Dr Lin Gui

Lin.1.Gui@kcl.ac.uk

www.kcl.ac.uk/people/lin-gui