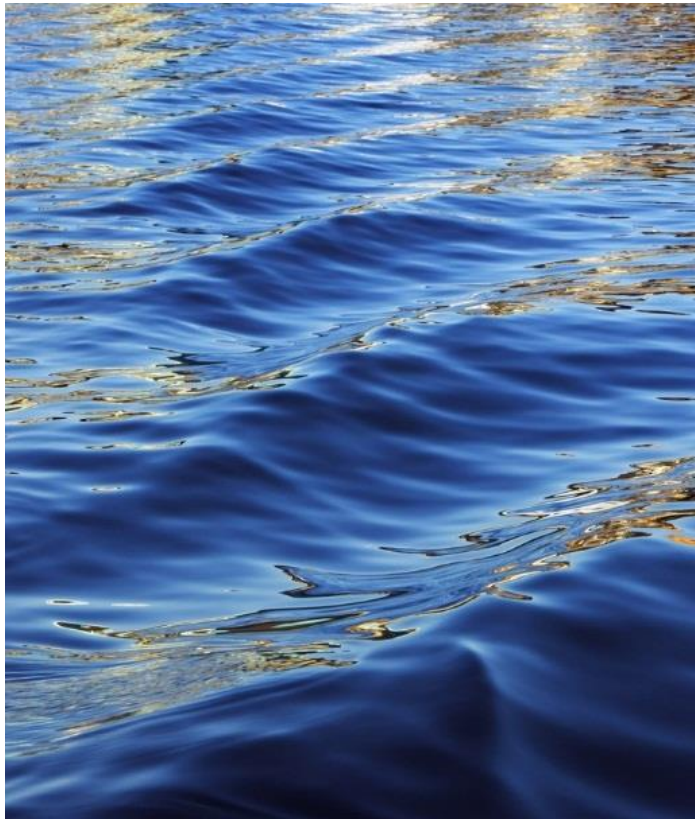
Searching and Sorting

Data structure and Algorithms (310-2101)



Searching

What is Searching?

We will now turn our attention to some of the most common problems that arise in computing, those of searching and sorting. In this section we will study searching.

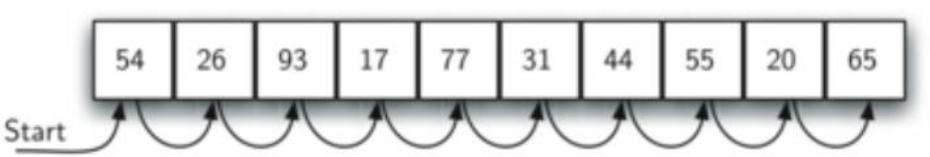
Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present. On occasion it may be modified to return where the item is found.

Python syntax for Searching

```
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
>>>
```

The Sequential Search

What is Sequential Search?



When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship.

In Python lists, these relative positions are the index values of the individual items.

Since these index values are ordered, it is possible for us to visit them in sequence.

```
1
2  def sequential_search(a_list, item):
3      pos = 0
4      found = False
5
6      while pos < len(a_list) and not found:
7          if a_list[pos] == item:
8              found = True
9          else:
10             pos = pos + 1
11     return found
12
13 if __name__ == '__main__':
14
15     test_list = [1, 2, 32, 8, 17, 19, 42, 13, 0]
16
17     print(sequential_search(test_list, 3))
18     print(sequential_search(test_list, 13))
```

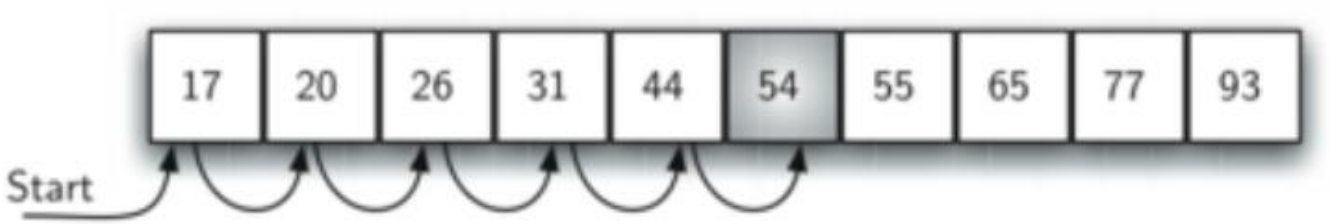
The Sequential Search (Analysis)

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

Best Case: we will find the item in the first place we look, at the beginning of the list.

Worst Case: we will not discover the item until the very last comparison, the nth comparison.

Average Case: we will find the item about halfway into the list



The Sequential Search (Modified)

We can slightly modify the sequential search by ordering the item in list in ascending order.

We will still have the same number of comparisons to find the item.
However, if the item is not present there is a slight advantage.

```
1
2  def ordered_sequential_search(a_list, item):
3      pos = 0
4      found = False
5      stop = False
6      while pos < len(a_list) and not found and not stop:
7          if a_list[pos] == item:
8              found = True
9              if a_list[pos] > item:
10                 stop = True
11         else:
12             pos = pos + 1
13     return found
14
15 if __name__ == '__main__':
16
17     test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42]
18
19     print(ordered_sequential_search(test_list, 3))
20     print(ordered_sequential_search(test_list, 13))
```

The Sequential Search (Modified)

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	1	n	$\frac{n}{2}$

The Binary Search

How can we use the advantage of the ordered list?

In the sequential search, when we compare against the first item, there are at most $n - 1$ more items to look through if the first item is not what we are looking for. Instead of searching the list in sequence, a **binary search** will start by examining the middle item.

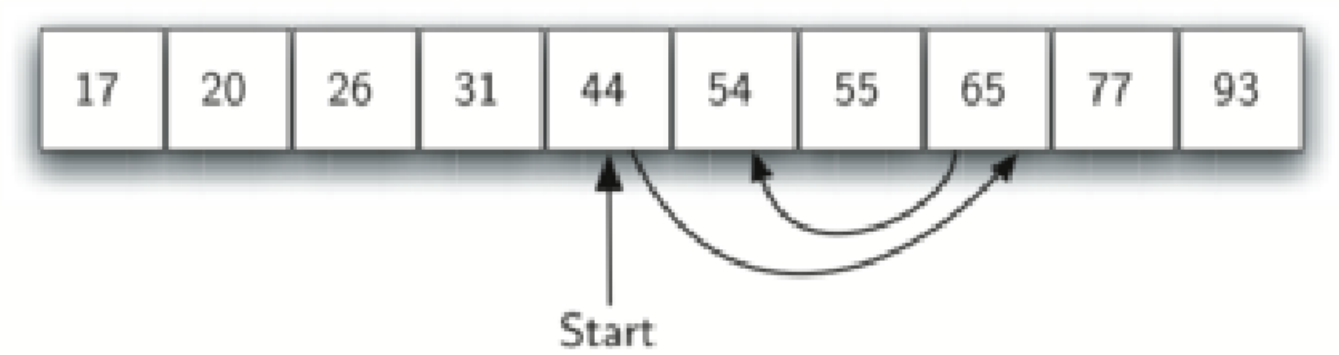
If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items.

If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

Divide and Conquer strategy

The Binary Search

How can we use the advantage of the ordered list?



The Binary Search

```
1
2   def binary_search(a_list, item):
3       first = 0
4       last = len(a_list) - 1
5       found = False
6
7       while first <= last and not found:
8           midpoint = (first + last) // 2
9           print(midpoint)
10          if a_list[midpoint] == item:
11              found = True
12          else:
13              if item < a_list[midpoint]:
14                  last = midpoint - 1
15              else:
16                  first = midpoint + 1
17      return found
18
19  if __name__ == '__main__':
20
21      test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42]
22
23      print(binary_search(test_list, 3))
24      print(binary_search(test_list, 13))
```

The Binary Search (Analysis)

Comparisons	Approximate Number Of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
$\ldots$	$\ldots$
i	$\frac{n}{2^i}$

The Binary Search (Recursive version)

```
1
2   def binary_search(a_list, item):
3       if len(a_list) == 0:
4           return False
5       else:
6           midpoint = len(a_list) // 2
7
8       if a_list[midpoint] == item:
9           return True
10      else:
11          if item < a_list[midpoint]:
12              return binary_search(a_list[midpoint], item)
13          else:
14              return binary_search(a_list[midpoint + 1:], item)
15
16  if __name__ == '__main__':
17
18      test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42]
19
20      print(binary_search(test_list, 3))
21      print(binary_search(test_list, 13))
```

Hashing

In previous sections we were able to make improvements in our search algorithm by taking advantage of information about where item are stored in the collection with respect to one another.

By knowing that a list was ordered, we could search in logarithmic time using a binary search.

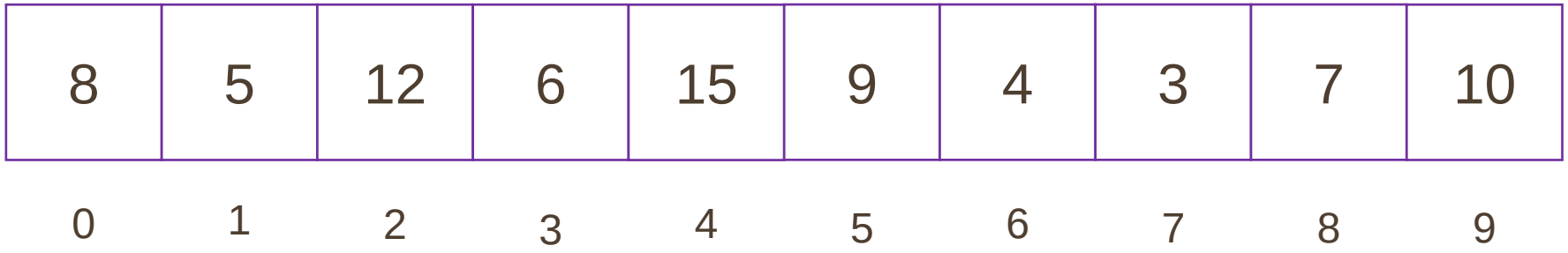
Now, we will attempt to go one step further by building a data structure that can be search in O(1). This concept is referred to as **hashing**.

Big-O (1)

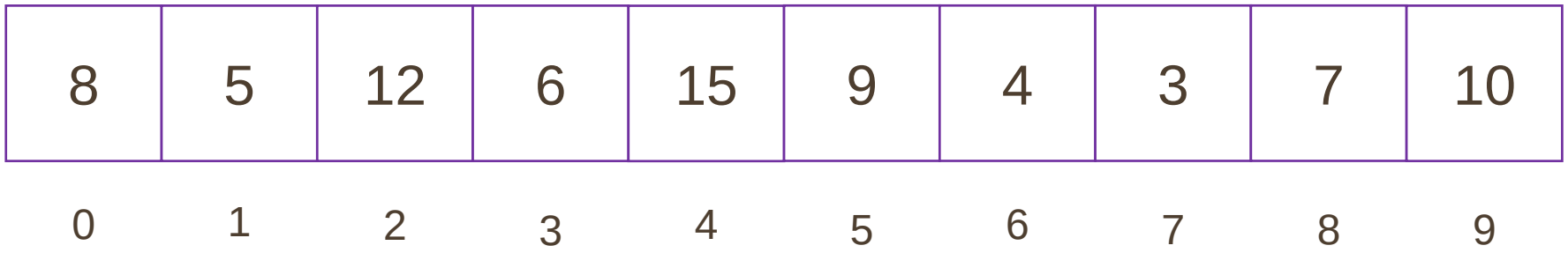
(ทำงาน 1 ครั้ง ไม่ว่า Input จะเป็นอะไรก็ตาม)

Hashing

Sequential Search **O (n)** ใช้เวลาในการทำงานเท่ากับจำนวนของ Input ที่ใส่ลงไป

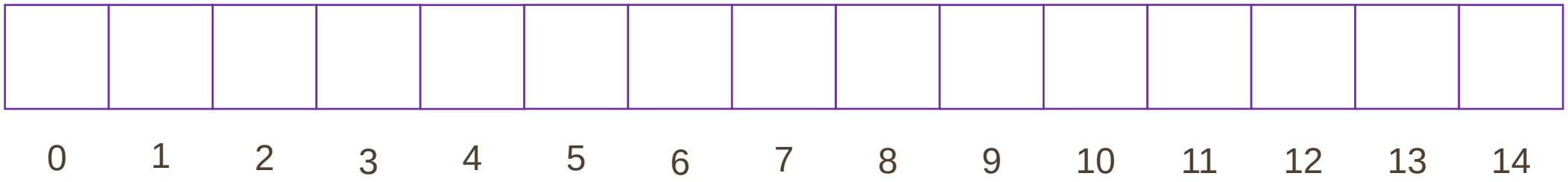


Binary Search **O (log n)** ลดจำนวน Loop ลง ครึ่งหนึ่งทุกครั้ง ที่ทำงานเสร็จไป 1 รอบ



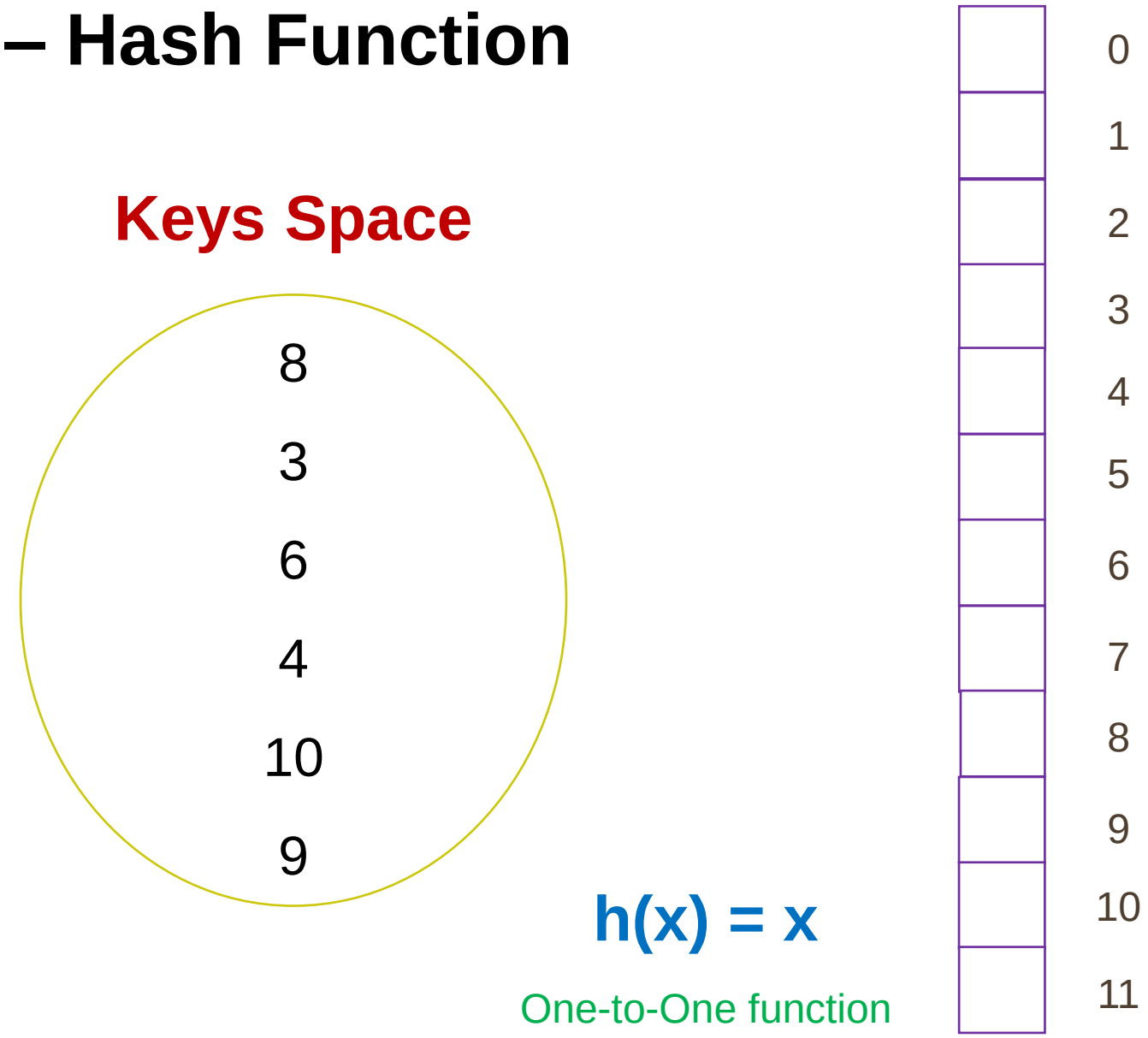
Hashing

Keys – 8, 3, 13, 6, 4, 10



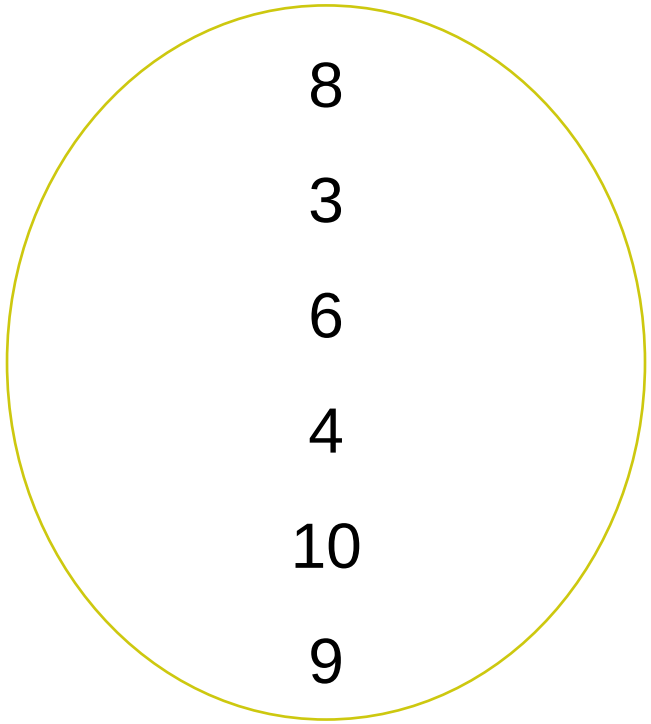
Memory VS **Performance**

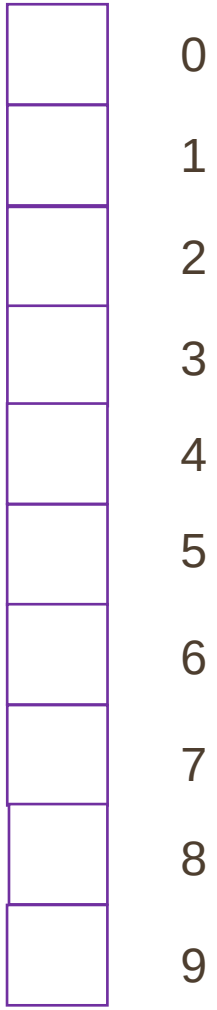
Hashing – Hash Function



Hashing – Hash Function

Keys Space





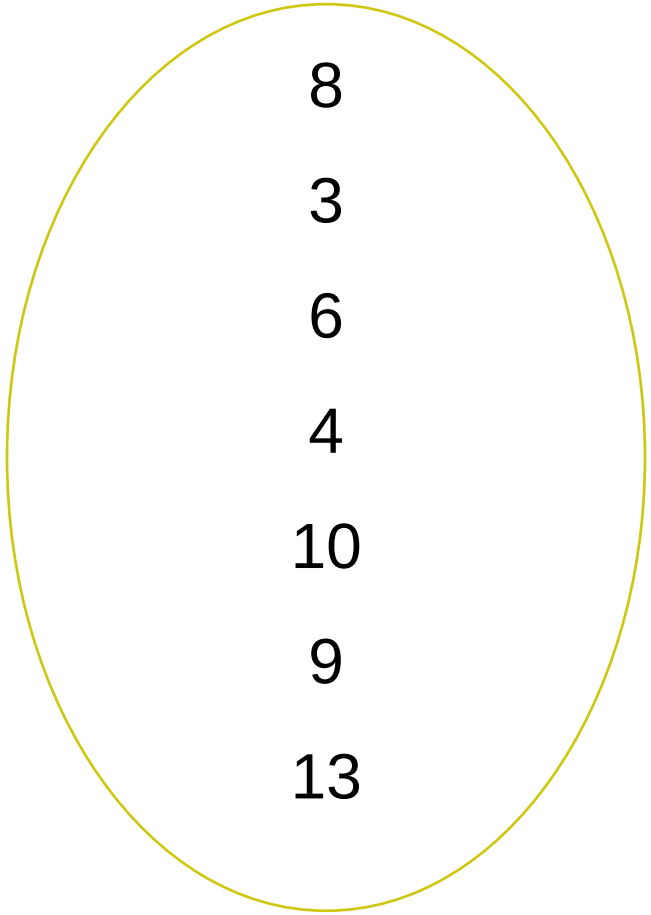
h(x) = x % 10

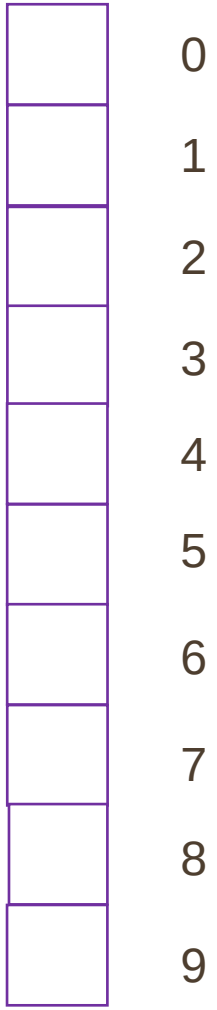
Many-to-One function

Size of Hash Table

Hashing – Hash Function (Collision)

Keys Space





h(x) = x % 10

Many-to-One function

Size of Hash Table

Hashing – How to create hash function

1.) Folding method

dividing the item into equal size of pieces (the last piece may not be of equal size).

These pieces are then added together to give the resulting **hash value**.

Example: divide into groups of 2

436-555-4601	➔	43, 65, 55, 46, 01
	➔	43 + 65 + 55 + 46 + 01
	➔	210

if our hash size is 11 then, 210%11 = 1 ➔ assign to slot 1

Hashing – How to create hash function

2.) Mid-square method

We first square the item, and then extract some portion of the resulting digits.

Example:

44	➔	$44^2 = 1936$
Extract middle two digits	➔	93
Perform remainder step	➔	93 % 11
	➔	assign to slot 5

Hashing – How to create hash function

Take a Note

We can also create hash functions for character-based items such as strings.

The word “cat” can be thought of as a sequence of ordinal values.

c	a	t
99	97	116

99 + 97 + 116 = 312

312 % 11 ➔ Slot 4

Hashing – How to create hash function

```
1
2   def hash(a_string, table_size):
3       sum = 0
4       for pos in range(len(a_string)):
5           sum = sum + ord(a_string[pos])
6
7       return sum % table_size
8
9   if __name__ == '__main__':
10
11      text = "krisada"
12
13      hash_value = hash(text, 10)
14
15      print(hash_value)
```

Hashing – Collision Resolution Methods

1.) Open Hashing

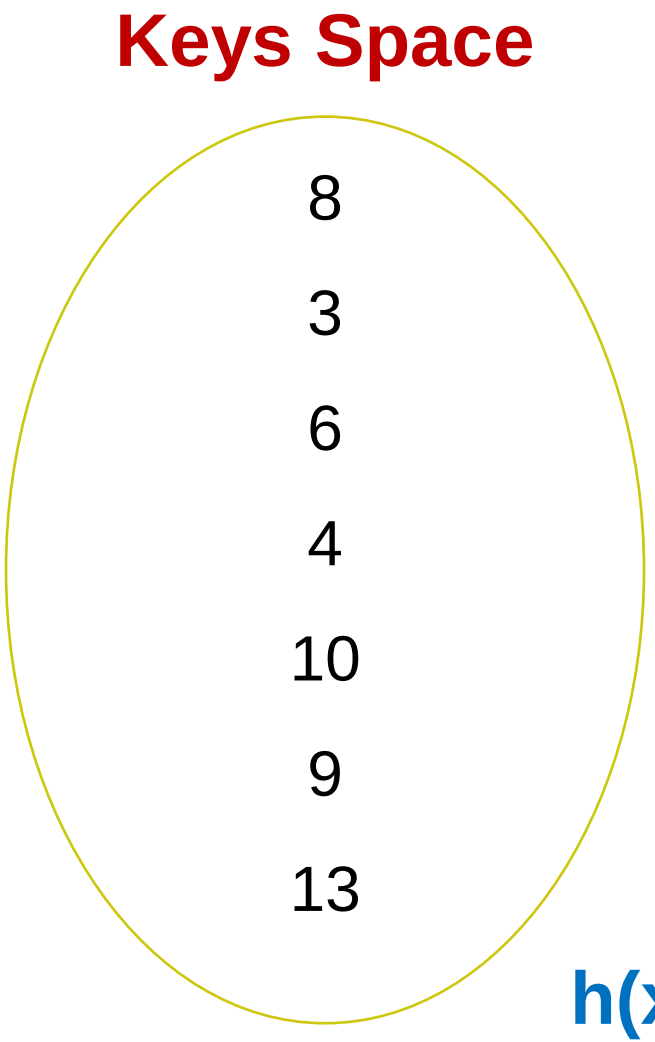
Chaining

2.) Closed Hashing

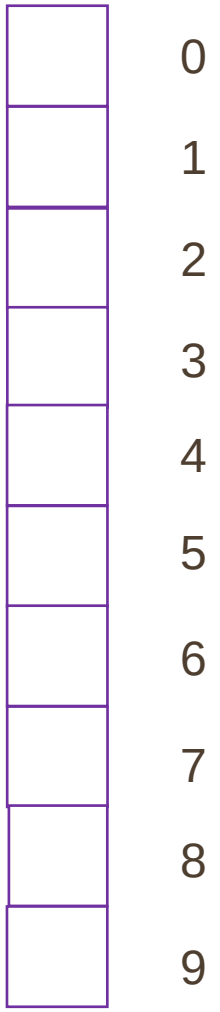
Open Addressing

- Linear Probing
- Quadratic Probing

Hashing – Collision Resolution – #1 Chaining

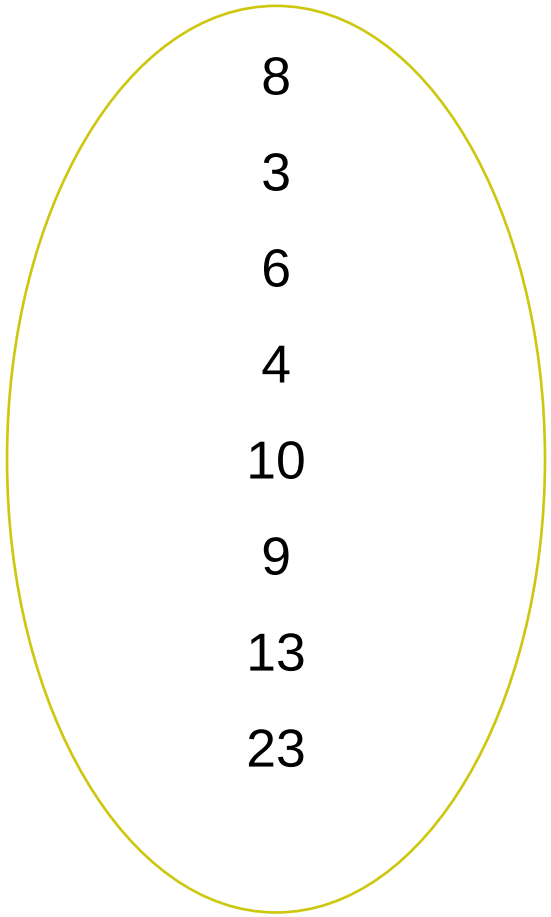


$h(x) = x \% 10$



Hashing – Collision Resolution – #2 Linear Probing

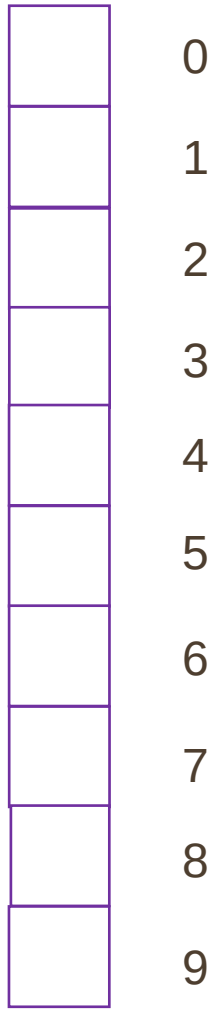
Keys Space



$$h(x) = x\%10$$

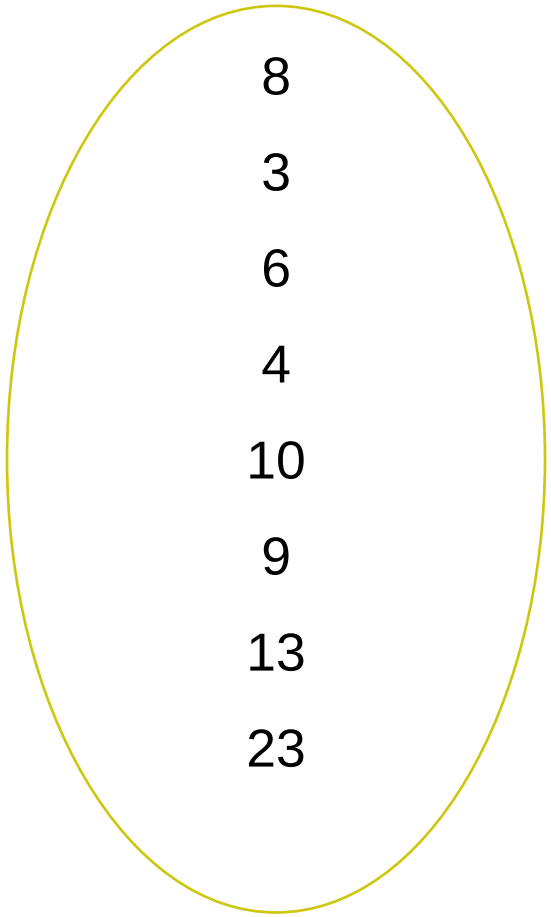
$$\hat{h}(x) = h(x) + i$$

i เพิ่มทุกครั้งที่ “ชน”



Hashing – Collision Resolution – #3 Quadratic Probing

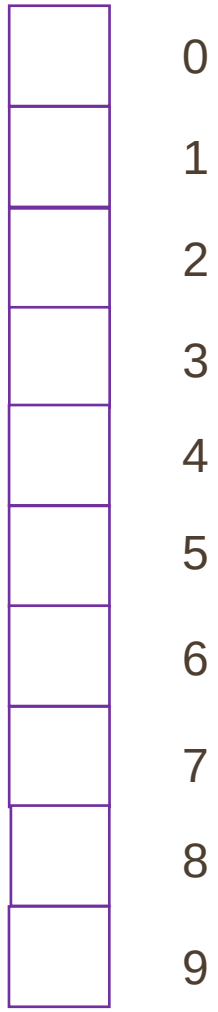
Keys Space



$$h(x) = x\%10$$

$$\hat{h}(x) = h(x) + i^2$$

i เพิ่มทุกครั้งที่ “ชน”



Hashing

```
1
2   class HashTable:
3       def __init__(self):
4           self.size = 11
5           self.slots = [None] * self.size
6           self.data = [None] * self.size
7
8       def put(self, key, data):
9           hash_value = self.hash_function(key, len(self.slots))
10
11          if self.slots[hash_value] == None:
12              self.slots[hash_value] = key
13              self.data[hash_value] = data
14          else:
15              if self.slots[hash_value] == key:
16                  self.data[hash_value] == data
17              else:
18                  next_slot = self.rehash(hash_value, len(self.slots))
19                  while self.slots[next_slot] != None and self.slots[next_slot] != key:
20                      next_slot = self.rehash(next_slot, len(self.slots))
21
22                  if self.slots[next_slot] == None:
23                      self.slots[next_slot] = key
24                      self.data[next_slot] = data
25                  else:
26                      self.data[next_slot] = data
```

Hashing

```
27
28      def hash_function(self, key, size):
29          return key % size
30
31      def rehash(self, old_hash, size):
32          return (old_hash + 1) % size
33
34      def get(self, key):
35          start_slot = self.hash_function(key, len(self.slots))
36
37          data = None
38          stop = False
39          found = False
40          position = start_slot
41          while self.slots[position] != None and not found and not stop:
42              if self.slots[position] == key:
43                  found = True
44                  data = self.data[position]
45              else:
46                  position = self.rehash(position, len(self.slots))
47                  if position == start_slot:
48                      stop = True
49          return data
50
```

Hashing

```
50
51      def __getitem__(self, key):
52          return self.get(key)
53
54      def __setitem__(self, key, data):
55          self.put(key, data)
56
```

Dunder or magic methods in Python

Dunder here means “Double Under (Underscores)”.

These are commonly used for operator overloading.

https://www.geeksforgeeks.org/dunder-magic-methods-python

Hashing

```
57  if __name__ == '__main__':
58
59      h = HashTable()
60      h[54] = "cat"
61      h[26] = "dog"
62      h[93] = "lion"
63      h[17] = "tiger"
64      h[77] = "bird"
65      h[31] = "cow"
66      h[44] = "goat"
67      h[55] = "pig"
68      h[20] = "chicken"
69      h[1] = "A"
70      h[2] = "B"
71
72      print(h.slots)
73      print(h.data)
74
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] python -u "d:\Project\Python\Data Structure\Hashing\Example1_Hash.py"
[77, 44, 55, 20, 26, 93, 17, 1, 2, 31, 54]
['bird', 'goat', 'pig', 'chicken', 'dog', 'lion', 'tiger', 'A', 'B', 'cow', 'cat']

[Done] exited with code=0 in 0.083 seconds
```

Workshop

แบ่งกลุ่ม 3 – 4 คน นำเสนอ Algorithms โดยเลือกให้แต่ละกลุ่มไม่ซ้ำกัน จากเว็บไซต์นี้

https://visualgo.net/en

1. Data structure ในการเก็บข้อมูลเป็นแบบใด
2. Algorithm มีเป้าหมายอะไร
3. นำเสนอตัวอย่างการทำงาน ขั้นตอนของ Algorithm

Workshop

Single-Source Shortest Paths

[1] BFS(s)

[2] Dijkstra(s)

[3] DFS(s)

[4] DP(s)

Convex Hull

[1] Andrew Chain

[2] CH Graham

[3] Shamos

Suffix Array

[1] Longest Common Prefix

[2] Longest Repeated Substring

[3] Longest Common Substring

Suffix Tree

[1] Longest Repeated Substring

[2] Longest Common Substring

Cycle Finding

[1] Floyd's Tortoise-Hare

Computational Geometry (Polygon)

- perimeter
- area
- isConvex
- inPolygon
- cutPolygon