

HASH TABLES

**SKILLS
FOR LIFE**

SKILLS BOOTCAMPS



Department
for Education

Foundational Sessions Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(FBV: Mutual Respect.)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes.
You can submit these questions here:

[SE Open Class Questions](#) or [DS Open Class Questions](#)

Foundational Sessions Housekeeping cont.

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- Report a **safeguarding** incident: www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

Reminders!

Guided Learning Hours

By now, ideally you should have 7 GLHs per week accrued. Remember to attend any and all sessions for support, and to ensure you reach 112 GLHs by the close of your Skills Bootcamp.

Progression Criteria

✓ **Criterion 1: Initial Requirements**

- Complete 15 hours of Guided Learning Hours and the first four tasks within two weeks.

✓ **Criterion 2: Mid-Course Progress**

- Software Engineering: Finish 14 tasks by week 8.
- Data Science: Finish 13 tasks by week 8.

✓ **Criterion 3: Post-Course Progress**

- Complete all mandatory tasks by 24th March 2024.
- Record an Invitation to Interview within 4 weeks of course completion, or by 30th March 2024.
- Achieve 112 GLH by 24th March 2024.


✓ **Criterion 4: Employability**

- Record a Final Job Outcome within 12 weeks of graduation, or by 23rd September 2024.




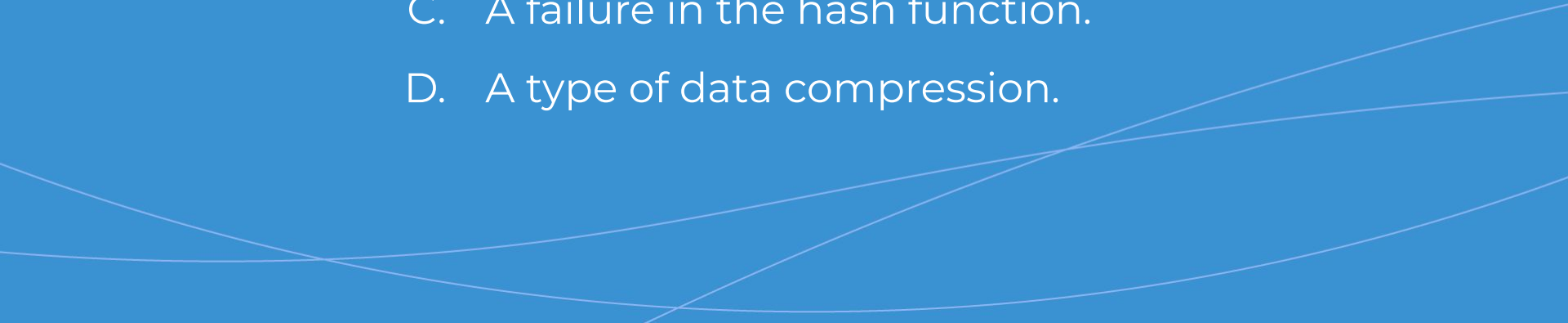
What is the primary purpose of a hash function in a hash table?



- A. To encrypt data.
 - B. To map data to unique hash codes.
 - C. To sort data in ascending order.
 - D. To reduce data size.
- 




What is a hash collision?

- 
- A. A security breach in a hash table.
 - B. When two different inputs produce the same hash code.
 - C. A failure in the hash function.
 - D. A type of data compression.
- 



How do Python dictionaries primarily store and access data?



- A. Using a list-based structure.
 - B. Through a tree structure.
 - C. Using a hash table mechanism.
 - D. Via direct indexing.
- 

Recap of Linear Data Structures



ASCII

ASCII is a character encoding standard used for electronic communication, representing text in computers and other devices with limited set of 128 code points

Dec	Hex	Chr
0	00	NUL
1	01	SOH
2	02	STX
3	03	ETX
4	04	EOT
5	05	ENQ
6	06	ACK
7	07	BEL
8	08	BS
9	09	HT
10	0A	LF
11	0B	VT
12	0C	FF
13	0D	CR
14	0E	SO
15	0F	SI

Dec	Hex	Chr
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

Dec	Hex	Chr
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

Dec	Hex	Chr
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

Dec	Hex	Chr
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

Dec	Hex	Chr
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Chr
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D]
94	5E	^
95	5F	_

Dec	Hex	Chr
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	
125	7D	}
126	7E	~
127	7F	DEL

Unicode

Unicode is a universal encoding system that allows the representation of a comprehensive character set, including those required for global multilingual text processing

- UTF-8, a popular Unicode format, encodes characters using 1 to 4 bytes, providing efficient data representation.

```
# UTF-8 Encoding
# Converting strings to their UTF-8 encoded byte representation
utf8_strings = ['A', 'λ', '😊']
utf8_encoded = [s.encode('utf-8') for s in utf8_strings]
print("UTF-8 Encoded Values:", utf8_encoded)
# UTF-8 Encoded Values: [b'A', b'\xce\xbb', b'\xf0\x9f\x99\x82']
```

String Concatenations in Python

- **Strings in Python are immutable:**
 - '+' creates a new string by copying each character
 - Time complexity: $O(n^2)$
 - **join** creates a new string by copying characters of one string
 - Time complexity: $O(n)$

```
# String Concatenation using '+'
start_time = time.time()
concat_str = ""
for _ in range(10000):
    concat_str += "hello "
end_time = time.time()
print("Time with '+':", end_time - start_time)
# Time with '+': 0.019999027252197266
```

```
# String Concatenation using 'join()'
start_time = time.time()
join_str = ''.join(["hello " for _ in range(10000)])
end_time = time.time()
print("Time with 'join()':", end_time - start_time)
# Time with 'join()': 0.00049591064453125
```


Hash Tables Topics

1. Introduction to Hash Tables
2. Hash Collisions
3. Implementing a Hash Table
4. Comparing Hash Tables and Dictionaries

Efficient Spell-Checkers

Consider spell-checking tools which are used on your devices. Suggestions and corrections need to be made as you are typing so correct spellings of words need to be looked up very quickly. A data structure containing pairs of common incorrect spelled words and the correct spelling, but this structure would be very big.

- What data structure can be used that will allow us to **create such a big data structure** efficiently and quickly?
- How can we store these pairs of words in a way that ensures that our data structure can be **searched and common misspellings looked up** and the correct spellings retrieved very quickly ?

Example: Dictionaries in Python

The data structure known as a dictionary is a data structure that can easily solve our problem. In a dictionary, key-value pairs are stored and values are accessed using the key value.

```
# Simple autocorrect structure which uses a dictionary
autoc = {"acommodate": "accommodate", "accomodate": "accommodate",
        "acknowledgement": "acknowledgment", "aquire": "acquire",
        "apparant": "apparent", "aparent": "apparent",
        "apparrent": "apparent", "aparrent": "apparent"}

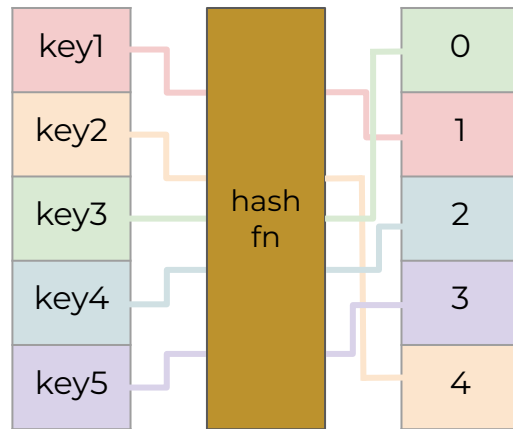
correct_spelling = autoc["accomodate"]
```

- It turns out that this data structure is very efficient, but how does it work? How is it so efficient?

Hash Tables

A data structure that is used to store key-value pairs which uses a hash function to transform the key into the index of an associated array element where the data will be stored.

- Hash tables allow for **efficient and fast** insertions, deletions and look-ups due to the manner in which values are stored.
- Since the **index where the values is stored is a function of the value**, the position where the value must be stored is always known.



- **Hashing** refers to using the hash function to calculate **the hash** which is the index of the array element where the key-value pair will be stored.
- Each array element is called a **bucket or slot** and can store one or more key-value pairs.
- The **hash function** can be any **deterministic** function which maps a key to the range of indices but the aim is that the function is **efficient**, it distributes keys **uniformly** and **multiple keys being mapped to the same index** is avoided.
- The **load factor** is the ratio of the number of stored elements to the total number of buckets and is used to determine whether the table can be downsized to improve performance.

Hash Collisions

When two or more distinct keys hash to the same array index.

- Although we try to avoid hash collisions, they are inevitable so we have to implement a **collision-resolution** process.
- When searching or storing data, a hash collision **increases the time complexity** of the task.

Operation	Average Case	Worst Case
Insert	$O(1)$	$O(n)$
Lookup/Search	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Collision-Resolution Methods

- **Chaining:** Linked Lists are used for each bucket, so multiple key-value pairs can be stored in the same bucket.
 - Pros: Simple to implement, dynamic resizing
 - Cons: Memory overhead implications, space inefficiencies
- **Linear Probing:** If there is a collision, place the pair in the next available slot in the hash table (linearly).
 - Pros: Simple to implement, memory-efficient
 - Cons: Primary clustering (large blocks of occupied elements), clustering results in more time inefficiencies

Hash Tables vs Dictionaries

- **Dictionaries** are data structures which store key-value pairs.
- They are implemented internally using a **hash table**, which makes use of Python's built-in `hash` function.
- Dictionaries are the easiest way to implement a hash table in Python, since all the complexities involved in ensuring efficiency is abstracted away and is done for us.
- A custom hash table would be primarily used when a **custom hash function** needs to be implemented.
- This is useful in the case where keys are of **custom or non-hashable data types**.

Hash Table Implementation

For this implementation, we'll use chaining to handle hash collisions. We use a numpy array to store an array of Nodes containing the key-value pairs.

```
# Each element in our array needs to hold a key-value pair
# We create a node class to hold the pair
# We also implement a linked list for chaining
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
```

Hash Table Implementation

For maximum efficiency, we should constantly resize the hash table based on the load but we haven't included that in our implementation since we'll be using the hash table for small datasets.

```
class HashTable:
    # Initialise the Hash Table of arbitrary size
    # Use numpy empty array for the associative array
    def __init__(self, capacity = 20):
        self.capacity = capacity
        self.size = 0
        self.buckets = np.empty(capacity, dtype=Node)
```

Hash Table Implementation

We will be using the built-in hash function since our key value is not an Object type and the hash function sufficiently spreads the elements.

```
# We will be using the built in hash function
# Since this returns integers of arbitray sizes, we
# use the mod function to ensure they fit in the array
def hash(self, key):
    return hash(key) % self.capacity
```

We implement three functions: set, get and remove. Other useful functions we might want to implement as well include a resize and print function.

```
def set(self, key, value):  
    # Hash the key of the pair  
    index = self.hash(key)  
  
    # Check if there is a collision  
    node = self.buckets[index]  
    if (node == None):  
        self.buckets[index] = Node(key, value)  
        self.size += 1  
    else:  
        # If there is a collision, check for same key or  
        # add to end of linked list and increment size  
        while (node.next != None) and (node.key != key):  
            node = node.next  
  
        if node.key == key:  
            node.value = value  
        else:  
            node.next = Node(key, value)  
            self.size += 1
```



```
def get(self, key):  
    # Find the index where the pair is stored  
    index = self.hash(key)  
  
    # Check the bucket and retrieve the correct value  
    # Return None if the key is not in the hash table  
    node = self.buckets[index]  
    if (node == None):  
        return None  
    else:  
        while (node.key != key):  
            node = node.next  
            if (node == None):  
                return None  
        return node.value
```

To remove elements, first we find the correct element. The code is the same as a get function.

```
def remove(self, key):  
    # Find the index where the pair is stored  
    index = self.hash(key)  
  
    # Check the bucket and remove the correct node  
    # Return None if the key is not in the hash table  
    node = self.buckets[index]  
    prev = None  
    if (node == None):  
        return None  
    else:  
        while (node.key != key):  
            prev = node  
            node = node.next  
        if (node == None):  
            return None
```

Next, we remove the element and ensure the linked list is intact.

```
# Relink the list if a node is removed midlist
self.size -= 1
if prev == None:
    self.buckets[index] = node.next
else:
    prev.next = node.next

return node.value
```

Worked Example

Consider a simple memory cache to store recently accessed files from a disk.

How could we use a Hash Table to implement a memory cache that allows us to access a few recently accessed files faster and more efficiently?

1. How would you use the Hash Table data structure to store recently accessed data files in cache?
2. What would be the best way to remove items from the cache when the cache is full?
3. Describe the process of fetching a file from the disk, including the caching mechanism.
4. Implement a memory cache using a Hash Table.

Worked Example

Consider a simple memory cache to store recently accessed files from a disk.

How could we use a Hash Table to implement a memory cache that allows us to access a few recently accessed files faster and more efficiently?

1. How would you use the Hash Table data structure to store recently accessed data files in cache?

After a file is read from the disk, use the **set function** of the Hash Table to store the data in the cache. The key would be the **name of the data file**.

2. What would be the best way to remove items from the cache when the cache is full?

Most memory caches work on a **least recently used** basis. For this to work, the best collision resolution method would be **linear probing**.

3. Describe the process of fetching a file from the disk, including the caching mechanism.

Check if the file is in the cache (using the **get method**), if it's not fetch it from the disk then store it in the cache (using the **set method**).

Worked Example

Consider a simple memory cache to store recently accessed files from a disk.

How could we use a Hash Table to implement a memory cache that allows us to access a few recently accessed files faster and more efficiently?

4. Implement a memory cache using a Hash Table.

An example of how this can be implemented can be found in the source code for this lecture.

Summary

Hash Tables

- ★ Data structure which stores key-value pairs efficiently using a hash function to determine the index of an array where the pair will be stored
- ★ Hash functions have to be deterministic and simple to compute

Hash Collisions

- ★ When the hash function maps a pair to an element where a pair has already been stored, this is known as a hash collision
- ★ Hash collisions can be handled using linear probing or chaining

Dictionaries

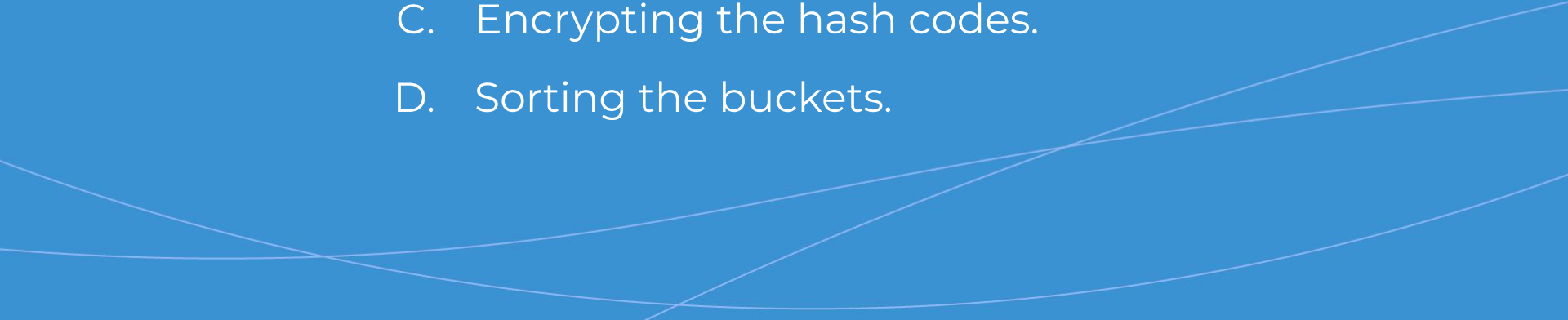
- ★ Dictionaries in Python are implemented using a hash table which makes use of the built-in hash function.


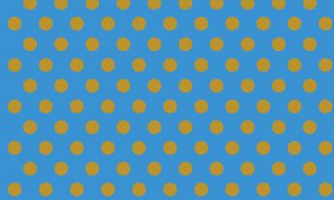
Further Learning

- [Hash Tables](#) - Section in the textbook “Algorithms” by Robert Sedgewick and Kevin Wayne
- [Introduction to Hash Tables](#) - Specifically for DS students
- [Comprehensive overview of Hash Tables](#) - Goes over all the topics covered as well as providing links to find out more
- [Hash Tables in Python](#) - Theory and Implementation of Hash Table in Python

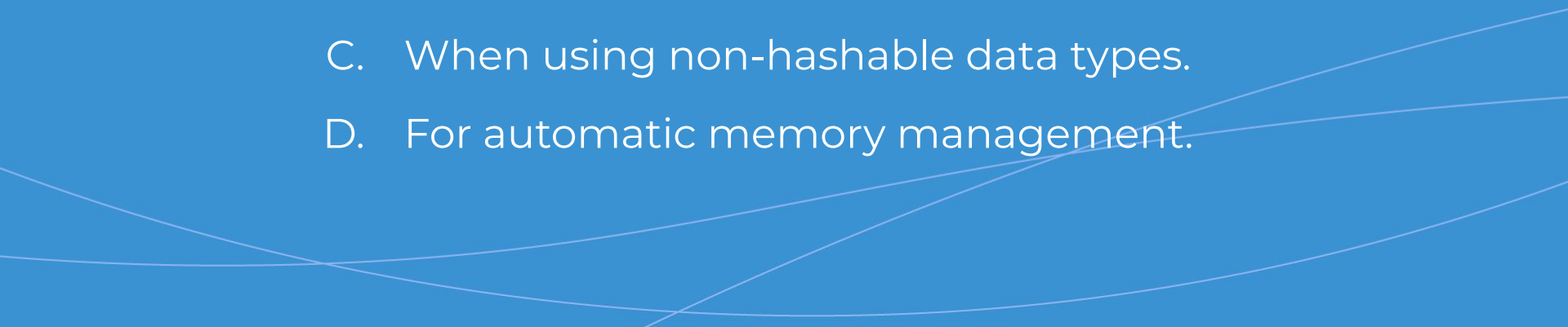


What is a common method to resolve hash collisions?

- A. Linear probing.
 - B. Increasing the hash table size.
 - C. Encrypting the hash codes.
 - D. Sorting the buckets.
- 



When would a custom hash table implementation be preferred over a Python dictionary?

- A. For smaller datasets.
 - B. For greater control over collision handling.
 - C. When using non-hashable data types.
 - D. For automatic memory management.
- 



Questions and Answers

Questions around Hash Tables

