CoGrammar

Welcome to this session:

Coding Interview Workshop - Hash Tables

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.

Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:

Ian Wyles
Designated Safeguarding Lead

Simone Botes

Nurhaan Snyman

Rafiq Manan

Ronald Munodawafa

Tevin Pitts

Scan to report a safeguarding concern

or email the Designated Safeguarding Lead:
Ian Wyles
safeguarding@hyperiondev.com

CoGrammar HyperionDev

Skills Bootcamp Coding Interview Workshop

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- 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 Academic Sessions. You can submit these questions here: **Questions**

Skills Bootcamp Coding Interview Workshop

- 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**
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Learning Outcomes

- **Explain the concept of hashing** and how it facilitates efficient data lookup, storage, and retrieval operations.
- **Illustrate how hash tables are implemented** and the underlying mechanisms that allow for efficient data access, including the use of hash functions and handling collisions.
- **Demonstrate the process of inserting, deleting, and retrieving data** from a hash table in Python, emphasising the importance of choosing appropriate hash functions.
- **Evaluate real-world applications of hash tables**, identifying scenarios where hash tables offer significant advantages over other data structures.

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.

CoGrammar

Correct Answer: B

- ❖ In a hash table, key-value pairs are stored in an associative array.
- ❖ The position where the pair will be stored is determined using the hash function, which takes in the pair’s key.
- ❖ 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.

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.

CoGrammar

Correct Answer: B

- ❖ Since we have no way of knowing what values will be stored in a hash table, our hash function may map unique pairs to the same position in the associative array.
- ❖ This is known as a hash collision.
- ❖ We try to avoid this scenario as much as possible, but we have to account for collision resolution in our hash table implementations.

CoGrammar

How do Python dictionaries primarily store and access data?

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

CoGrammar

Correct Answer: C

- ❖ Dictionaries in Python are stored internally using hash tables.
- ❖ All the complexities involved in creating an efficient hash table are dealt with by Python.
- ❖ This is the easiest way to create hash tables in our code.

CoGrammar

Lecture Overview

→ Spell Checkers
→ Introduction to Hash Tables
→ Hash Collisions
→ Applications of Hash Tables
→ Built-in Hash Tables
→ Hash Table Implementation

CoGrammar

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.

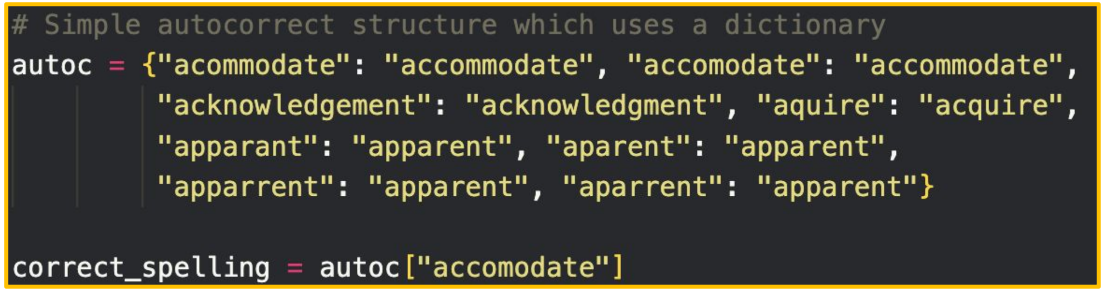
➢ What data structure can be used that will allow us to **create such a big data structure** efficiently and quickly?

➢ How can we store these pairs of words in a way that ensures that our data structure can be **searched, common misspellings looked up** and the correct spellings retrieved very quickly?

CoGrammar

Example: Spell Checkers

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



```python
# 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?

Example: Spell Checkers

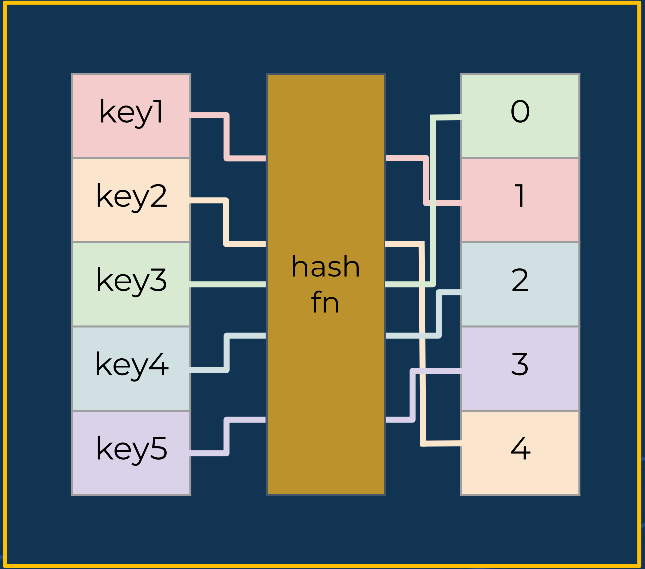
- ➢ This example can be extended to various Cyber Security applications:
  - ○ **Password storage** - storing username and password pairs so that credentials can be quickly verified.
  - ○ **Threat Intelligence** - storing indicators of compromise (IoC) to discover known threats on a system or within the network.

CoGrammar

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.



CoGrammar

Click here to visualize this data structure!

- **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.

CoGrammar

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)

CoGrammar

Collision-Resolution

- **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

CoGrammar

Applications of Hash Tables

Cyber Security

❖ **Intrusion Detection System**: Hash tables can store network events using unique identifiers (e.g., IP address + timestamp) as keys and event details as values. This enables fast lookup and aggregation of logs to detect patterns of malicious behavior, like repeated login attempts or unauthorized access.

❖ **Token Management**: Hash tables map session IDs to user data or session metadata. These tokens are validated quickly during user requests to ensure secure access.

Built-in Hash Tables

- **Dictionaries** are data structures which store key-value pairs.
- They are implemented internally using a **hash table**, using built-in hash functions.
- They are the easiest way to implement a hash table, since all the complexities involved in ensuring efficiency is abstracted away.
- 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.**

CoGrammar

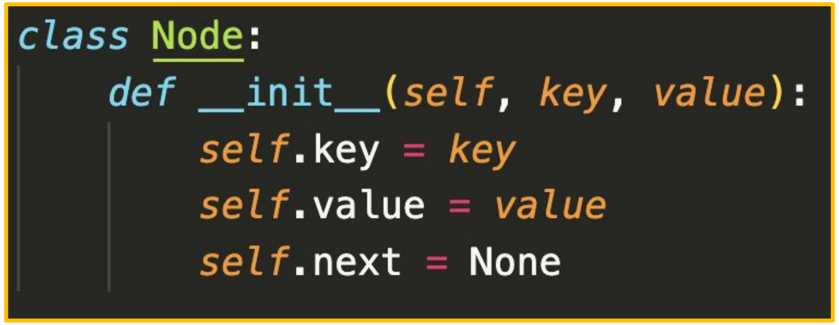
Let’s Breathe!

Let’s take a small break before moving on to the next topic.

CoGrammar

Hash Table Implementation

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

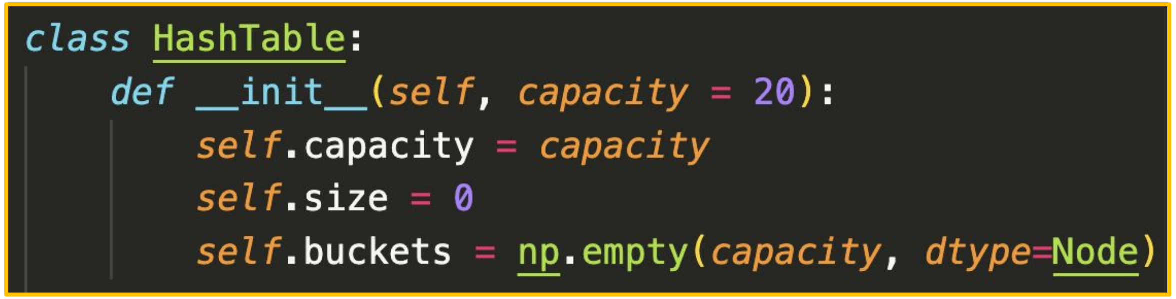


```python
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
```

CoGrammar

Hash Table Implementation

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

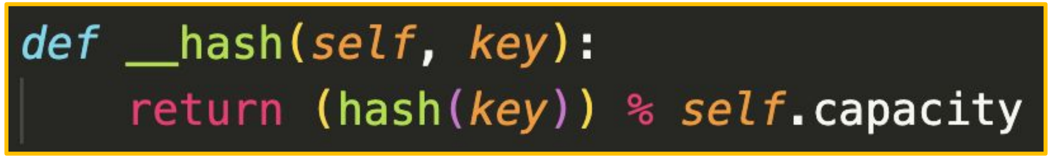


```python
class HashTable:
    def __init__(self, capacity = 20):
        self.capacity = capacity
        self.size = 0
        self.buckets = np.empty(capacity, dtype=Node)
```

CoGrammar

Hash Table Implementation

In Python, we will be using the built-in hash function which sufficiently spreads our elements and our keys are Hashable.

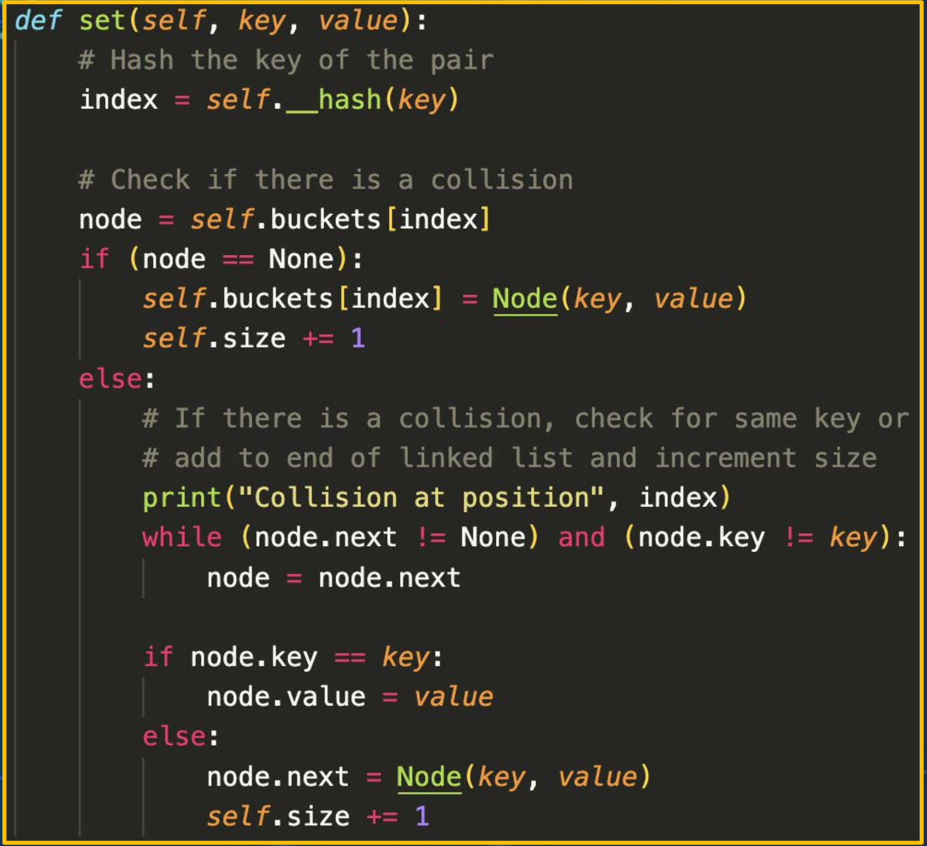


```python
def __hash(self, key):
    return (hash(key)) % self.capacity
```

CoGrammar

Hash Table Implementation

We implement three functions: **set**, **get** and **remove**. Other useful functions we might want to implement as well include a **resize** function, which can be used to manage the load factor of our hashtable, and a **print** function, which will allow us to visualise the hashtable

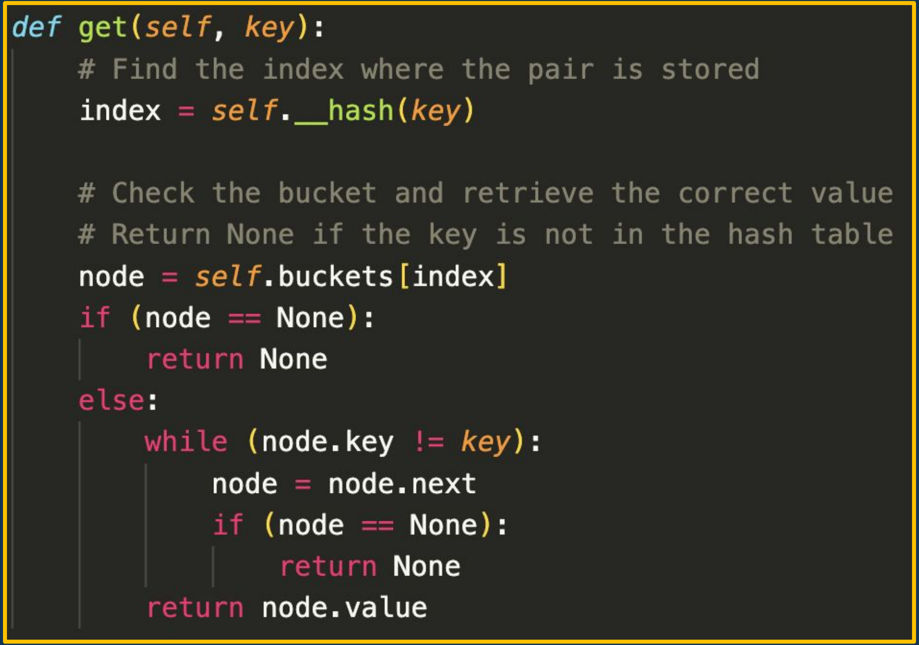


```python
def set(self, key, value):
    # Hash the key of the pair
    index = self.__hash(key)

    # 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
        print("Collision at position", index)
        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
```

CoGrammar



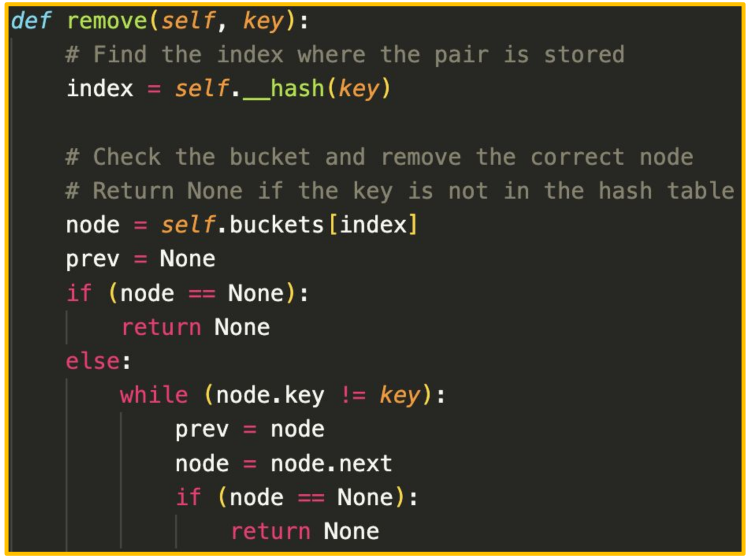
```python
def get(self, key):
    # Find the index where the pair is stored
    index = self.__hash(key)

    # 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
```

CoGrammar

Hash Table Implementation

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



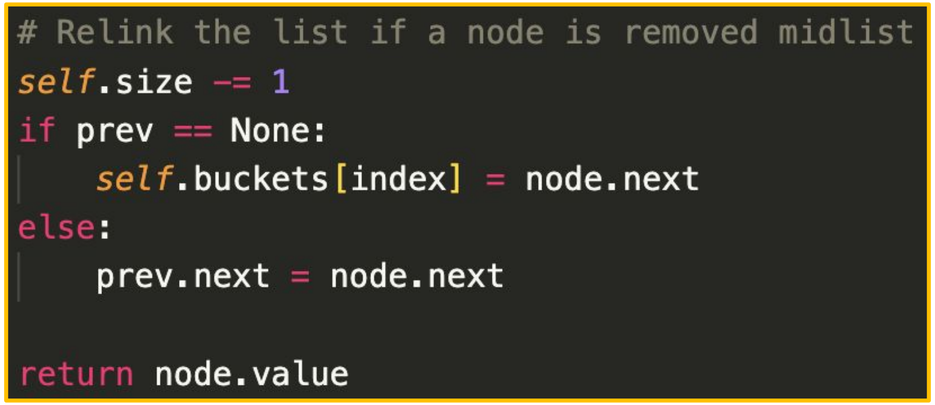
```python
def remove(self, key):
    # Find the index where the pair is stored
    index = self.__hash(key)

    # 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
```

CoGrammar

Hash Table Implementation

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



```
# 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
```

CoGrammar

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.

CoGrammar

Correct Answer: A

- Linear Probing is a Collision-Resolution method which involves looking for an open element in the hash table when a collision occurs.
- Increasing the hash table size is not an effective collision resolution method since this will increase the load factor of the hash table and affect performance.

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

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

CoGrammar

Correct Answer: C

- ❖ Some keys cannot easily be hashed using simple hash functions.
- ❖ Objects cannot be used as keys in Dictionaries and Maps.
- ❖ In these cases, a custom hash table implementation can be used to create a hash function which better suits the needs of the hash function.

CoGrammar

Homework

Practise the skills we've developed by completing the following problems:

- The next slide contains four questions to test your theoretical understanding of hash tables in a real world scenario.
- We'll be going through two LeetCode examples in the lecture over the weekend, attempt them yourself and come ready with questions:
  - Example 1
  - Example 2
- Practice speaking through your solutions and explaining how you approached each problem.

Homework 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.

Homework 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**).

Homework 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

CoGrammar

Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.

Thank you for attending

CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education