

Application Design

Brief Outline

‘Dionysus’ is a hypothetical, digital assistant application built upon a food inventory, to track users’ food stock. The user manages the inventory through a voice user interface (VUI), using a voice command device that recognises certain commands used to invoke CRUD operations (create, read, update, delete) on the database. By managing the food inventory, the user gains access to utilities, such as automated shopping-lists, that are derived from user habits, preferences, and food stock.

1.1

Amazon (2022), who invented the digital assistant ‘Alexa’, claim that advances in trends such as ‘the internet of things’, and ‘artificial intelligence’, enable new experiences with VUI’s today. Amazon has application programming interfaces available to developers, to augment Alexa.

Data Structures

Food stock information includes food entities, and fields either related to all food entities, or some of them. The fields may have hierarchical relationships between them too. Microsoft (2022) refers to the arrangement of information this way as a *document*, for which non-relational databases are suitable stores.

1.2

```

[
  {
    "name" : "apple",
    "amount": 5
  },
  {
    "name" : "milk",
    "kinds" :
    [
      {
        "name" : "full-fat milk",
        "container" :
        {
          "name" : "carton",
          "volume" :
          {
            "amount" : 1,
            "unit" : "litre"
          }
        },
        "amount": 2,
        "expiry date" : "12-DEC-2025"
      }
    ]
  }
]

```

Figure 1. An example of non-relational data.

2.1

A suitable data structure to implement a non-relational database would be a hashmap. According to Ku (2020), a hashmap can be dynamically resized, and allows for the insertion, deletion and retrieval of data in amortised $O(1)$ time. This would provide the speed and capacity to allow for real-time data input, and updates commanded by a user.

Cormen et al. (2009: 253-256) explains that a hashmap stores *keys*, and a hashing function converts the keys into addresses where data is located. Hashmaps allow data to be mutable, so other hashmaps can be nested within them. The keys should be immutable, but as strings are immutable; hashmaps allow for semantically meaningful labels to be used as the keys to the data; akin to the fields of entities.

2.2

```

class HashMap(object):
    def __init__(self, storage):
        """The constructor of the object."""
        self.storage = storage # storage space is required for allocating data
        self.keys = KeysContainer(storage[0]) # some storage is used to track keys

    def Hash(self, k) -> int:
        """Returns a location in storage labelled by the key k."""

    def Insert(self, k, data):
        """Stores the data at the address labelled by the key k."""

    def Delete(self, k):
        """Deletes the data stored at the address labelled by the key k"""

    def Search(self, k) -> bool:
        """Returns whether data is stored at the address labelled by the key k"""

```

3.1

Figure 2. An example of an object interface for a hashmap.

Note. The keys attribute has been given its own data structure called a KeysContainer, which is a type of HashMap with no keys attribute, to gain constant-time insertions, deletions and lookups.

Algorithms

Hashing

According to Cormen (2009: 262), a suitable hash function is one that, for a random input, is equally likely to give any of its outputs, in $O(1)$ time.

```

def Hash(self, k):
    """Returns a location in storage labelled by the key k."""

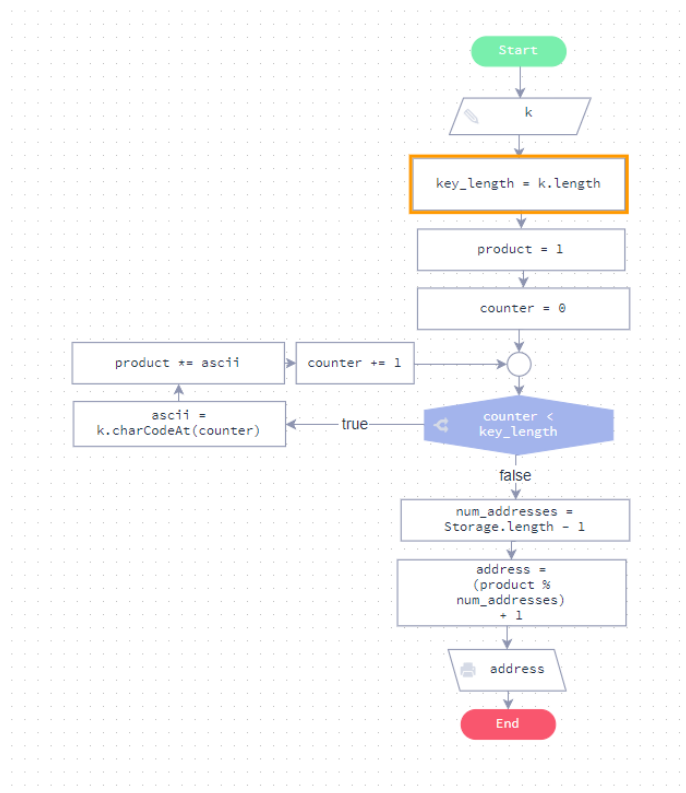
    # an integer is created by multiplying the ASCII values of its characters
    product = 1
    for char in k:
        ascii = ord(char)
        product *= ascii

    # transform the integer into an address, giving care to available storage space
    num_addresses = len(self.storage) - 1
    address = product % num_addresses
    return address + 1

```

4.1

Figure 3. An example of a hashing algorithm for the HashMap.



4.2

Figure 4. A flow chart of the HashMap hash algorithm.

Dynamic Resizing

Cohen (2017) outlines different ways that programming languages, with implementations of hashmaps, handle dynamic resizing, not outlined here.

Insertion

```
def Insert(self, k, data):  
    """Stores the data at the address labelled by the key k."""  
  
    # HashMaps are dynamic data structures, that grow in size if required  
    if self.keys.isFull():  
        self.IncreaseStorageSpace()  
  
    # the key k is tracked  
    self.keys.Insert(k)  
  
    # the key is hashed into an address in storage  
    address = self.Hash(k)  
  
    # hash collisions may occur, that may need to be handled.  
    if self.storage[address]:  
        self.HandleHashCollision(address, data)  
    else:  
        self.storage[address] = data
```

5.1

Figure 5. Pseudocode for the HashMap insertion algorithm.

A hash collision is when two different keys are hashed to the same address. Cormen (2009: 257-258) suggests using a method such as *chaining* to handle hash collisions while inserting data. Chaining is the use of a linked list to create a path to any data that needs to be stored at the collision address.

```

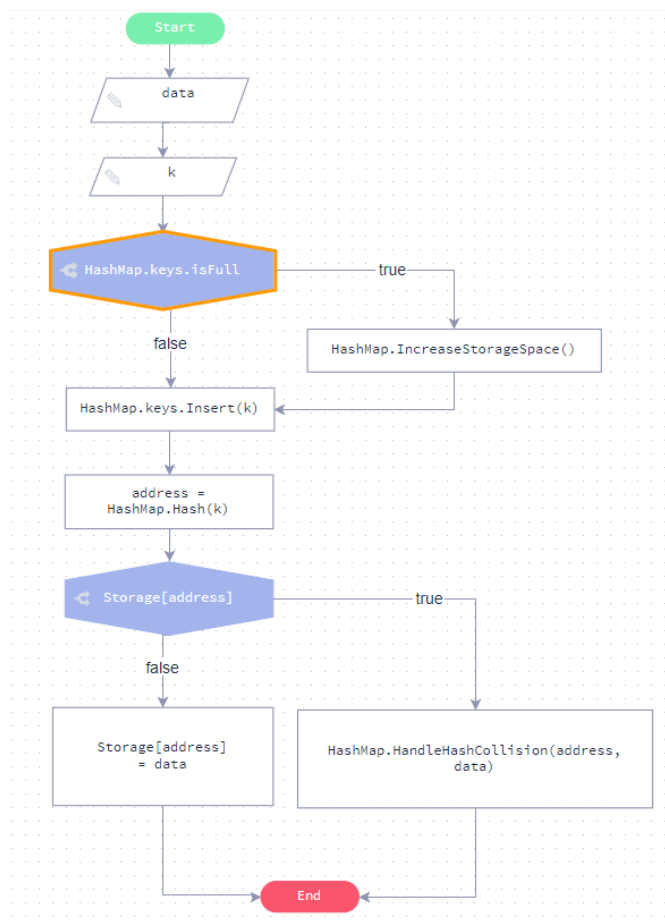
class LinkedList(object):
    def __init__(self, data):
        """The constructor for the LinkedList object."""
        head = Node(data)
        self.head = head
        self.tail = head

    def AddNode(self, data):
        """Adds a new node to the the tail of the linked list containing the data."""

class Node(object):
    def __init__(self, data):
        """The constructor for a Node object."""
        self.data = data
        self.next = None

```

Figure 6. An example of an object interface for a linked list, with a node helper class. This data structure is used to handle hash collisions.



6.1

Figure 7. A flow chart of the HashMap insertion algorithm.

```
def AddNode(self, data):
    """Adds a new node to the the tail of the linked list containing the data."""
    new_node = Node(data)
    self.tail.next = new_node
    self.tail = new_node
```

Figure 8. Pseudocode for the LinkedList insertion algorithm.

```
def Insert(self, k):
    """Stores the key k at the address labelled by the key k."""

    # the key is hashed into an address in storage
    address = self.Hash(k)

    # hash collisions may occur, that may need to be handled.
    if self.storage[address]:
        self.HandleHashCollision(address, k)
    else:
        self.storage[address] = k
```

7.1

Figure 9. Pseudocode for the KeysContainer insertion algorithm, used during the HashMap insertion algorithm.

Deletion

```
def Delete(self, k):
    """Deletes the data stored at the address labelled by the key k."""

    # the key k is no longer tracked
    self.keys.Delete(k)

    # HashMaps are dynamic data structures, that shrink in size if required
    if self.keys.isEmpty():
        self.DecreaseStorageSpace()

    # the key is hashed into an address in storage
    address = self.Hash(k)

    # hash collisions may have occurred, that need to be handled.
    if self.storage[address].isinstance(LinkedList):
        self.Tombstone(address, k)
    else:
        self.storage[address] = None
```

7.2

Figure 10. Pseudocode for the HashMap deletion algorithm.

To delete data stored in a linked list, Cohen (2017) describes a technique called ‘tombstoning’. Essentially, it involves replacing the data of a node with a value indicating deletion.

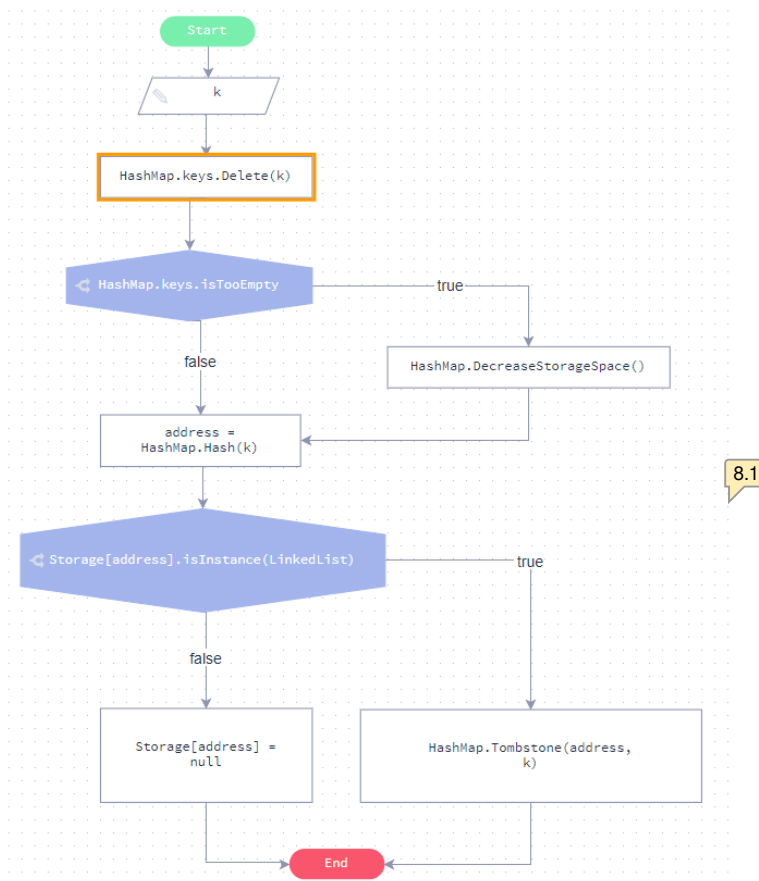


Figure 11. A flow chart depicting the HashMap

```

def Delete(self, k):
    """Deletes the key k."""

    # the key is hashed into an address in storage
    address = self.Hash(k)

    # hash collisions may have occurred, that need to be handled.
    if self.storage[address].isinstance(LinkedList):
        self.Tombstone(address, k)
    else:
        self.storage[address] = None
  
```

Figure 12. Pseudocode for the KeysContainer deletion algorithm, used during the HashMap deletion algorithm.

Searching

```
def Search(self, k):  
    """Returns whether data is stored at the address labelled by the key k"""  
  
    # the k is searched for in the KeyContainer  
    return self.keys.Search(k)
```

Figure 13. Pseudocode for the HashMap search algorithm.

```
def Search(self, k):  
    """Returns whether the key k is stored at the address labelled by the key k"""  
  
    # the key is hashed into an address in storage  
    address = self.Hash(k)  
  
    # hash collisions may have occurred, that need to be handled.  
    if self.storage[address].isinstance(LinkedList):  
        return self.SearchList(address, k)  
    else:  
        return self.storage[address] == k
```

Figure 14. Pseudocode for the KeyContainer search

```
def SearchList(self, address, k):  
    """Returns whether k is stored in the linked list at the address location."""  
  
    # loop through each node of the linked list, checking values  
    node = self.storage[address].head  
    while node.next:  
        data = node.data  
        if data == k:  
            return True  
        node = node.next  
    return False
```

9.1

Figure 15. Pseudocode for a search algorithm that searches a linked list, used in the KeysContainer search algorithm.

Sorting

The hashmap algorithms outlined above enable CRUD operations on the food database. To enable further intelligence of the food inventory, it would be useful to implement a sorting mechanism. Hashmaps have no inherent order, so an additional

data structure is needed to provide indexing, like an array. Winand (2012) points out that SQL uses balanced trees to provide indexing for relational databases though.

Peters (2002), the creator of the standard sorting algorithm for arrays in Python 3.11, states that Python uses a hybrid of merge sort, and insertion sort. Merge sort takes $O(n \log n)$ time, while insertion sort takes $O(n^2)$ time. Insertion sort is quicker for small arrays thanks to it having less overhead. This hybrid algorithm would enhance our database.

```
def MergeSort(array):
    if len(array) == 1:
        return array
    else:
        mid = len(array) // 2
        left = MergeSort(array[:mid])
        right = MergeSort(array[mid:])
        return CleverMerge(left, right)

def CleverMerge(left_array, right_array):
    merge = []
    while left_array and right_array:
        if left_array[0] <= right_array[0]:
            merge.append(left_array.pop(0))
        else:
            merge.append(right_array.pop(0))

    while left_array:
        merge.extend(left_array)
        left_array = []

    while right_array:
        merge.extend(right_array)
        right_array = []

    return merge
```

Figure 16. Pseudocode for the merge sort algorithm.

```
def InsertionSort(array):
    for i in range(1, len(array)):
        for j in range(i, 0, -1):
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
    return array
```

10.1

Figure 17. Pseudocode for the insertion sort algorithm.

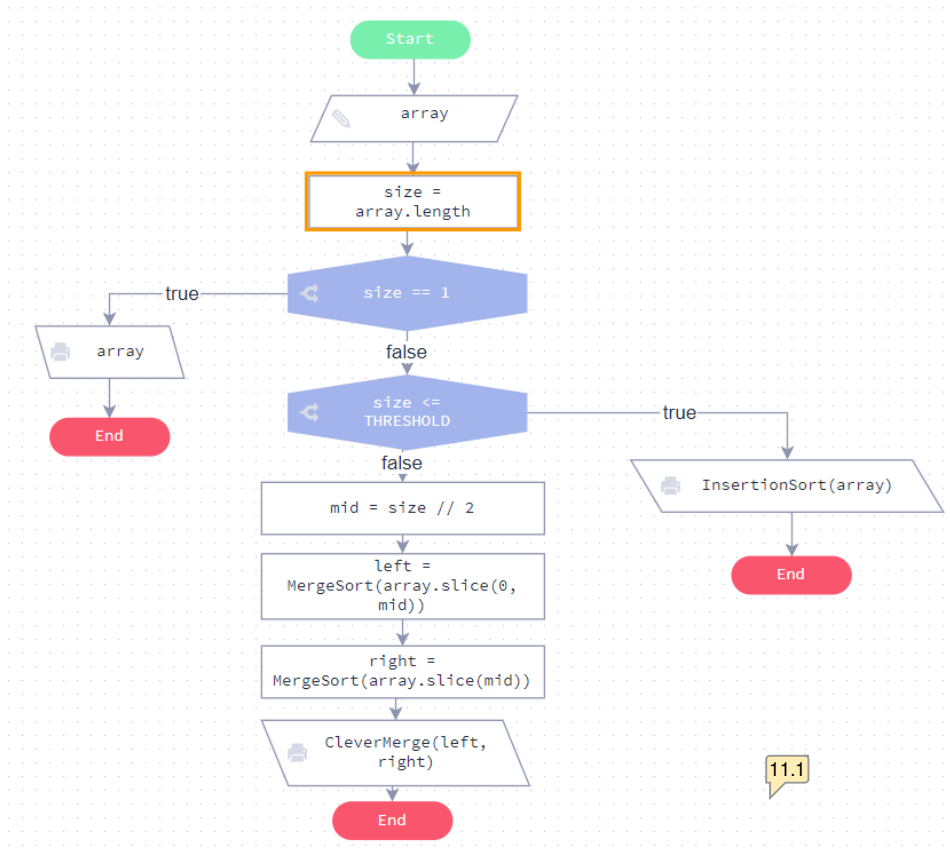


Figure 18. A flowchart depicting the hybrid merge sort algorithm.

Quality Assurance

In order to provide quality assurance for the application, various types of tests are recommended. Unit tests can be designed for each algorithm, that check that: i) the output of the algorithm is correct for different test cases, ii) object attributes are modified correctly, and iii) guardians such as type restrictions and input restrictions work correctly. Microsoft (2021) describes how [SQL](#) injections use input as an attack vector on relational databases; so cyberthreats need to be considered too.

CRUD operations are built from data structure interfaces, combining algorithms, so integration tests can be designed to check they do as expected. The

asynchronous behaviour of CRUD operations can be tested too. Yu (2009) describes a framework, centred on 'transactions' interacting on a database, with properties to prevent destructive interactions. These properties are known as ACID properties (atomicity, consistency, isolation, durability).

12.1

Performance tests can test how the application handles large amounts of data while beta testing provides black-box testing. Guttag (2021) defines black-box testing, as software testing without looking at the code. Feedback is important too.

References

Amazon. (2022) *What Is a Voice User Interface?*. Available from: <https://developer.amazon.com/en-GB/alexa/alexa-skills-kit/vui> [Accessed 09/11/2022].

Cohen. (2017) *An Analysis of Hash Map Implementations in Popular Languages*. Available from: <https://rcoh.me/posts/hash-map-analysis/> [Accessed 13/11/2022].

Cormen, T., Leiserson, C., Rivest, R. & Stein, C. (2009) *Introduction to Algorithms*. 3rd ed. Massachusetts: The MIT Press. 253-256, 262.

Guttag, J. (2021) *Introduction to Computing and Programming Using Python*. 3rd ed. Massachusetts: The MIT Press. 149.

Ku, J. (2020) Lecture 4: Hashing [Lecture Video] 6.006 SPRING 2020 Introduction to Algorithms. Massachusetts Institute of Technology. Available from:

https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/video_galleries/lecture-videos/ [Accessed 11/11/2022].

Microsoft. (2021) *SQL Injection*. Available from: <https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16> [Accessed 13/11/2022].

Microsoft. (2022) *Non-relational data and NoSQL*. Available from: <https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data> [Accessed 11/11/2022].

Peters, T. (2002) *Timsort*. Available from: <https://bugs.python.org/file4451/timsort.txt> [Accessed 12/11/2022].

Winand, M. (2012) *Anatomy of an SQL Index*. Available from: <https://use-the-index-luke.com/sql/anatomy> [Accessed 12/11/2022].

Yu, S. (2009) *ACID Properties in Distributed Databases*. Available from: https://www.cs.helsinki.fi/group/cinco/teaching/2009/advanced-businesstransactions-seminar/papers/ACID_in_Distributed_Database_Shiwei_Yu.pdf [Accessed 13/11/2022].

Index of comments

- 1.1 To me, somehow vague. You must have made it very clear as this is the key message that the reader will read first
- 1.2 Once again, not clear. You must have made all the points very clear
- 2.1 Ok
- 2.2 Information wise good.
- 3.1 OK
- 4.1 Code looks fine to me.
- 4.2 OK
- 5.1 Very impressive.
- 6.1 Excellent
- 7.1 Good work
- 7.2 Looks fine to me.
- 8.1 Flowchart looks alright to me.
- 9.1 Looks ok
- 10.1 Excellent use of codes
- 11.1 Nice one.
- 11.2 Ok but at which checkpoint are you going to do the test?
- 12.1 The whole test plan section is somehow not quite clear
- 13.1 Very good use of academic sources.