Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

b-it Bonn-Aachen International Center for Information Technology

Algorithm Analysis & Data Structures

Mihir Mehta

Foundations Course Winter Semester 2021

17.09.2021

What is an Algorithm?

- Any Guess(es)?

What is an Algorithm? *(contd…)*

Techincally Speaking…

An Algorithm is a well-defined procedure that takes a set of values as input and produces set of values as output.

Simply Speaking…

An Algorithm is just a recipe/guide/manual of simple steps to follow in order to achieve some goal.

Essentially,

An Algorithm is a tool to solve a well defined *(computational)* problem.

What is an Algorithm? *(contd...)*

In terms of Computer Science...
An Algorithm is a set of steps a program takes to finish a task.

Time for an example – Algorithm in real life…

Goal: To bake a simple cake!

Algorithm/Steps:

- Preheat the oven.
- Gather the ingredients.
- Measure out the ingredients.
- Mix together the ingredients to form a batter.
- Grease/Oil a pan.
- Pour the prepared batter into the pan.
- Put the pan into an Oven.
- Set the timer.
- When timer rings, take the pan out of the oven.
- Enjoy! *(Not really a part of the Algorithm :P)*

A few other examples – Algorithm in real life…

Daily Activities such as:

- Driving to a particular destination.
- Doing Laundry.
- Preparing Coffee.
- Reaching to Office/University.
- Working and completing the assignments.

⋮

Have I ever seen/written an Algorithm?

Coming back to the definition of Algorithm – Computer Science…

An Algorithm is a set of steps a program takes to finish a task.

What if my goal was just to display “Hello World”?

```
print("Hello World")
```

```
Hello World
```

Congrats!

Why should we study about Algorithms?

A simple toy example:

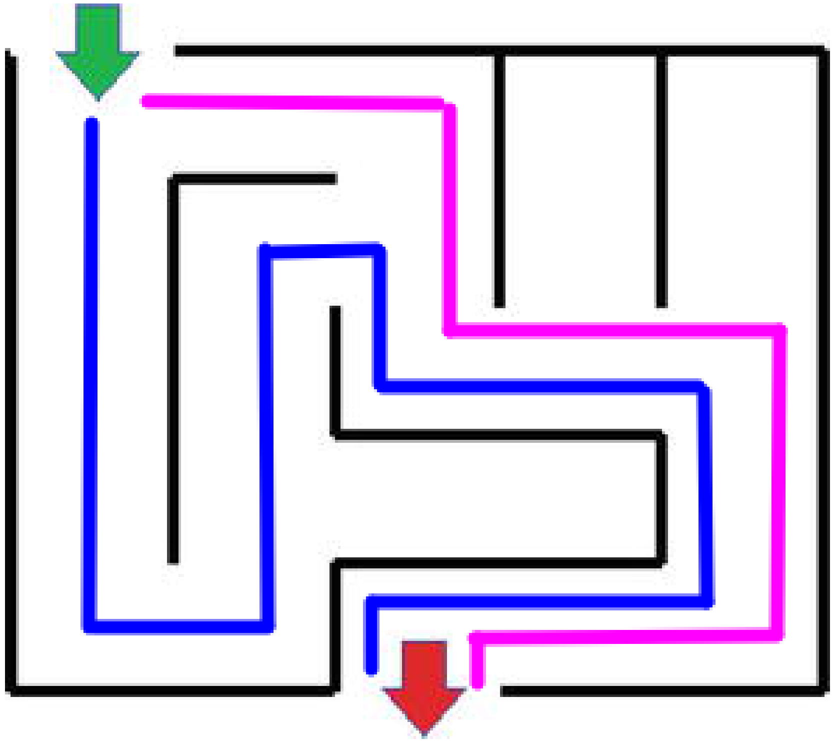


Figure 1: Solving a simple maze [1]

Comparing Algorithms – with Example

Suppose …

Figure 2: You have a book to read [2]

Comparing Algorithms – with Example

And …

Figure 3: You have to open a target page [3]

Comparing Algorithms (*contd…*)

Keeping it really simple –

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Let's say our Book only has 10 pages!

As depicted above.

And, there are **only** two ways to achieve our goal:

1. Simply turn one page at a time sequentially.
2. Dividing the book into half recursively and until we reach our target page.

Comparing Algorithms *(contd…)*

Keeping it really simple –

1st Approach.
Target Page: 3

1	2	3	4	5	6	7	8	9	10

Required Steps – 3

Comparing Algorithms *(contd...)*

Keeping it really simple –

2nd Approach.
Target Page: 3

1	2	3	4	5	6	7	8	9	10

Required Steps – 3

Comparing Algorithms *(contd…)*

Keeping it really simple –

1st Approach.
Target Page: 10

1	2	3	4	5	6	7	8	9	10

Required Steps – 10

Comparing Algorithms *(contd…)*

Keeping it really simple –

2nd Approach.
Target Page: 10

1	2	3	4	5	6	7	8	9	10

Required Steps – 4

Comparing Algorithms *(contd…)*

Keeping it really simple –

1st Approach.
Target Page: 100

1	2	3	• • •	50	• • •	97	98	99	100

Required Steps – ?

Comparing Algorithms *(contd…)*

Keeping it really simple –

2nd Approach.
Target Page: 100

1	• • •	50	• • •	75	• • •	88	• • •	94	• • •	98	99	100

Required Steps – 7

Comparing Algorithms *(contd…)*

What did we just perform?

→ Algorithm Analysis

Comparisons of Algorithms are usually done on the basis of:

Two Important factors:

1. Time Complexity
2. Space Complexity

Ways to evaluate efficiency of an Algorithm

Three Important Cases:

1. Worst Case - O (called as Big Oh)
2. Best Case - Ω (called as Big Omega)
3. Average Case - Θ (called as Big Theta)

Worst Case – Approach 1

Keeping it really simple –

1^{st} Approach.
Input Size (n) : 100 Target Page : 100

Required Steps : 100

Time Complexity: O(?)

Worst Case – Approach 1

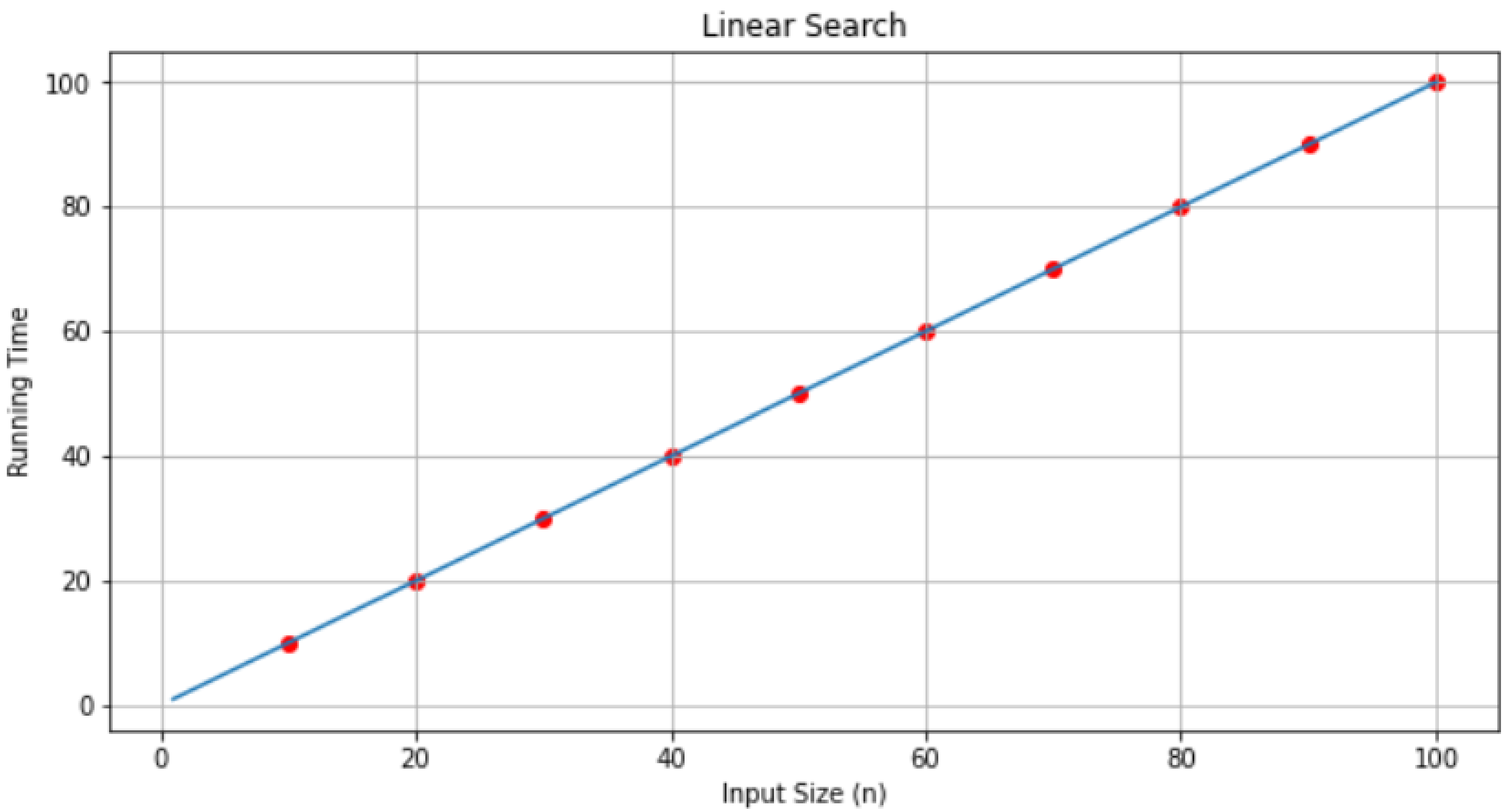


Figure 4: Run Time Complexity for Linear Search

Worst Case – Approach 1

Keeping it really simple –

1st Approach.
Input Size (n) : 100 Target Page : 100

Required Steps : 100

Time Complexity: O(n)

Worst Case – Approach 2

Keeping it really simple –

2^{nd} Approach.
Input Size (n) : 100 Target Page : 100

Required Steps : 7

Time Complexity: O(?)

Worst Case – Approach 2

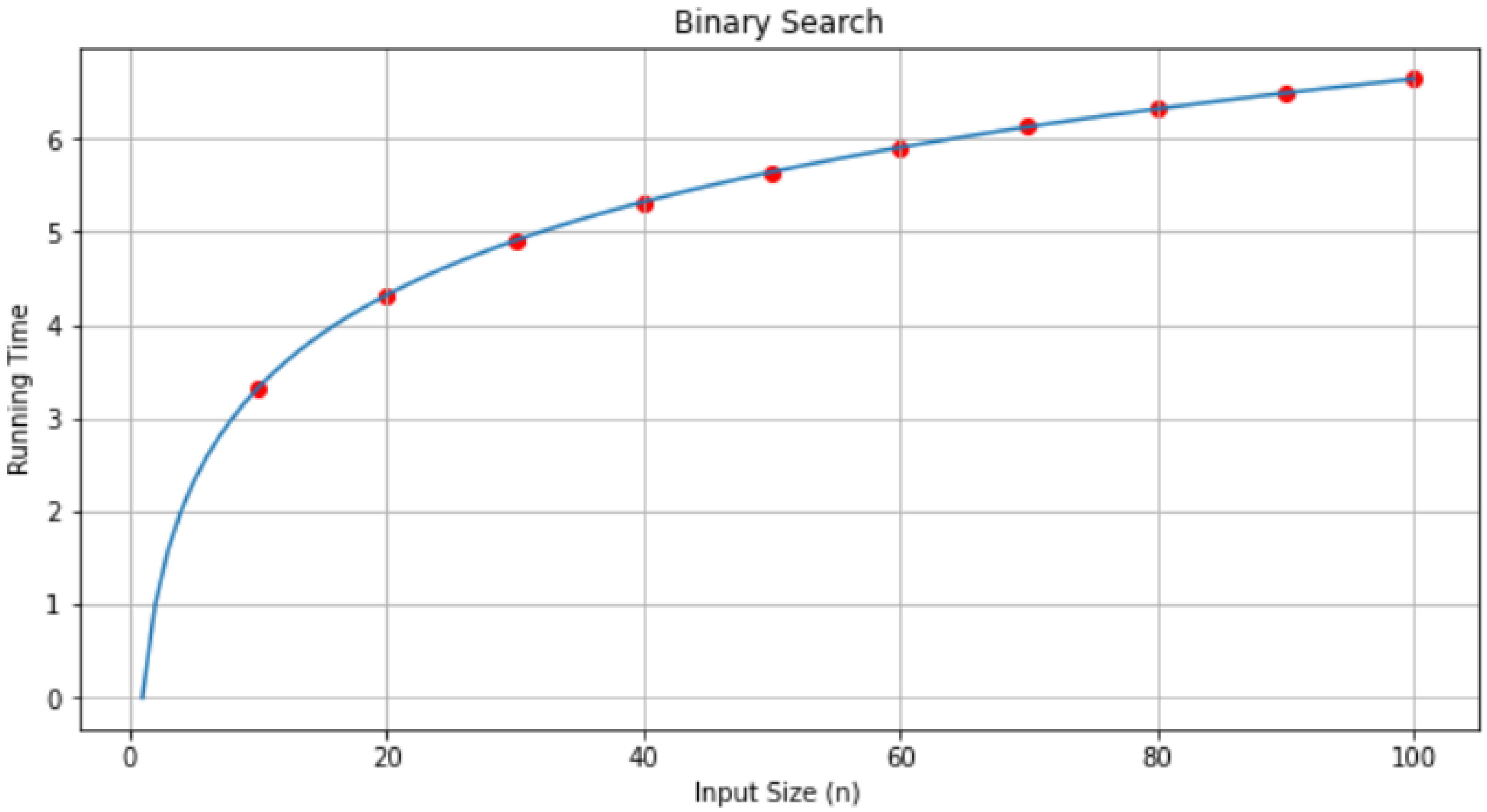


Figure 5: Run Time Complexity for Binary Search

Worst Case – Approach 2

Keeping it really simple –

2nd Approach.
Input Size (n) : 100 Target Page : 100

Required Steps : 7

Time Complexity: O($\log_2(n) + 1$)

Time Complexity: O($\boldsymbol{\log_2(n)}$)

Some common time complexities

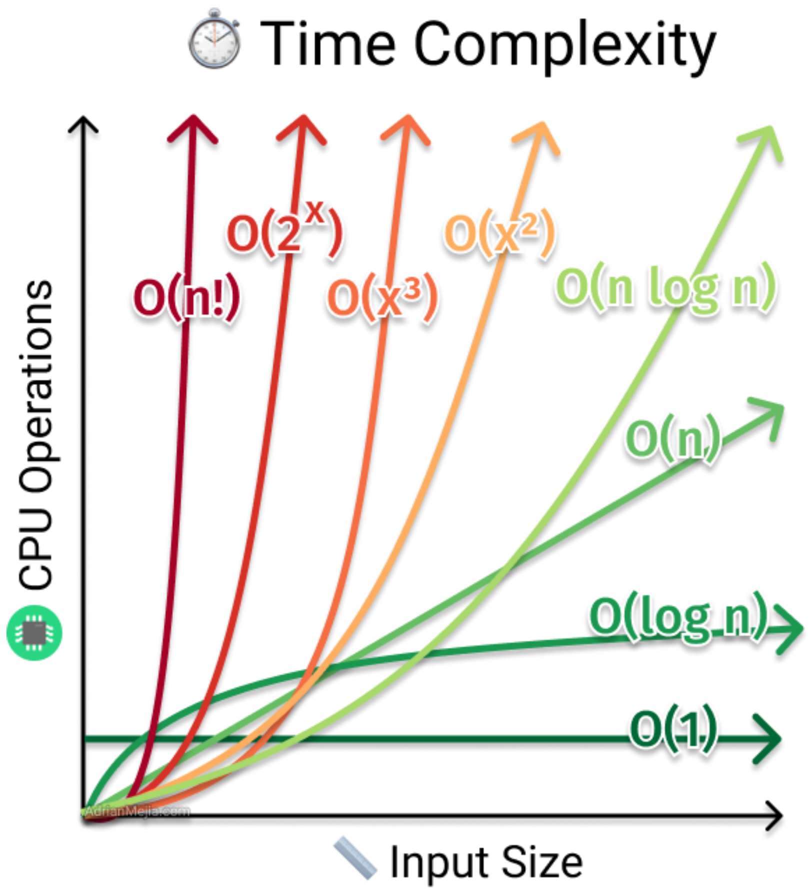


Figure 6: Comparing run-time complexity [4]

Taking a step back… to think

How are we storing the data now?

Both the approaches currently use a common way to store data.
A Python **List** (Few other languages may refer it as “Array”)

But, is the only way to store and access data in our programs?

Ways to store data...

- List
- Tuple
- Set
- Dictionary
- Linked Lists
- Stacks
- Queues
- ⋮

A short detour

- 3 properties of a Python Object
- Mutable vs Immutable Objects
- “is” operator vs “==” operator
- Container Objects
- Hashable Objects

Properties of a Python Object

- **Identity (ID) of an object**
  - Memory Address

- **Type**
  - Data Type *(int, list, set, etc. or user-defined class)*

- **Value**
  - The actual data/value held by the object

Mutable vs Immutable Objects

- **Mutable Objects**
  - Allow changing of values held by the object.
  - Example: set, user-defined classes, lists, etc.

- **Immutable Objects**
  - Do not allow changing of values held by the object.
  - Example: tuple, int, string, etc.

“is” operator vs “==” operator

- **<u>“is” operator</u>**
  - Compares identity of two objects.

- **<u>“==” operator</u>**
  - Compares the values of two objects.

Container Objects

- Objects containing another objects or reference to other objects.

- Example: list, dictionary, tuples, etc.

- **Note:** An immutable container's value which is referencing to a mutable object can be changed if that mutable object is changed.

Hashable Objects

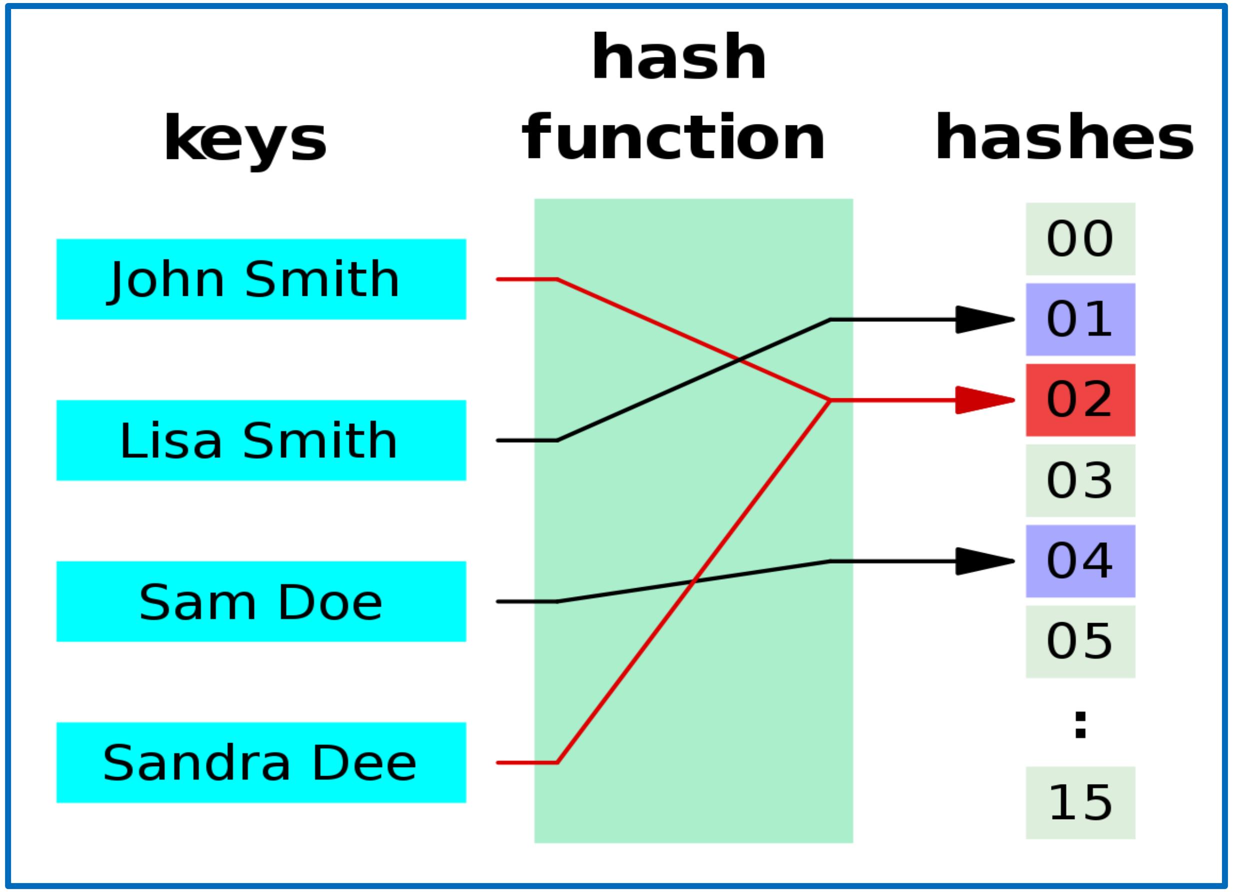


Figure 7: Dictionary as a Map [5]

Hashable Objects *(contd.)*

- The main idea is that, you can reduce a complex object to an index in an array.

- Example: Used in implementing dictionary, set, etc. data-structures.

- **Hashable data types:** int, float, str, tuple, NoneType, and user-defined classes.

- **Unhashable data types:** dict, list, and set.

Going back to – Data Structures in Python

- List
- Tuple
- Set
- Dictionary
- Linked Lists
- Stacks
- Queues
- ⋮

List (Array)

Value	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Index	0	1	2	3	4	5	6	7	8	9

- Stores duplicate data
- Items are ordered
- Mutable Object
- Can have items of different types

List (Array) *contd.*

Operation	Time Complexity (Worst Case)
Append	O(1)
Pop	O(1)
Insert	O(n)
Delete	O(n)
Setting value at particular index	O(1)
Getting value from particular index	O(1)
Get length	O(1)
Finding/Searching an element in List	O(n)

Tuple

```
simple_spells = ("Expelliarmus","Riddikulus","Alohamora")
spells = list(simple_spells)
spells.append("Wingardium Leviosaa")
simple_spells.append("Wingardium Leviosaa")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-69-0ed7f4ddeee7> in <module>
      2 spells = list(simple_spells)
      3 spells.append("Wingardium Leviosaa")
----> 4 simple_spells.append("Wingardium Leviosaa")

AttributeError: 'tuple' object has no attribute 'append'
```

- Stores duplicate data
- Items are ordered
- Immutable Object
- Can have items of different types

Set

```python
unforgivable_curses = ['Avada Kedavra','Crucio','Imperio',"Crucio","Imperio","Imperio"]
print(f"List: {unforgivable_curses}")
print("="*30)
print(f"Set :{set(unforgivable_curses)}")
```

```
List: ['Avada Kedavra', 'Crucio', 'Imperio', 'Crucio', 'Imperio', 'Imperio']
==============================
Set :{'Crucio', 'Imperio', 'Avada Kedavra'}
```

- Does not store duplicate data
- Items are unordered
- Mutable Object
- Can have items of different types

Set *(contd.)*

Operation	Time Complexity (Worst Case)
Finding element in set	O(1)
Intersection of two sets	O(min[len(set1), len(set2)])
Inserting an element	O(1)
Delete an element	O(1)
Iterating all elements in a set	O(n)
Get length	O(1)

Dictionary (hash-tables)

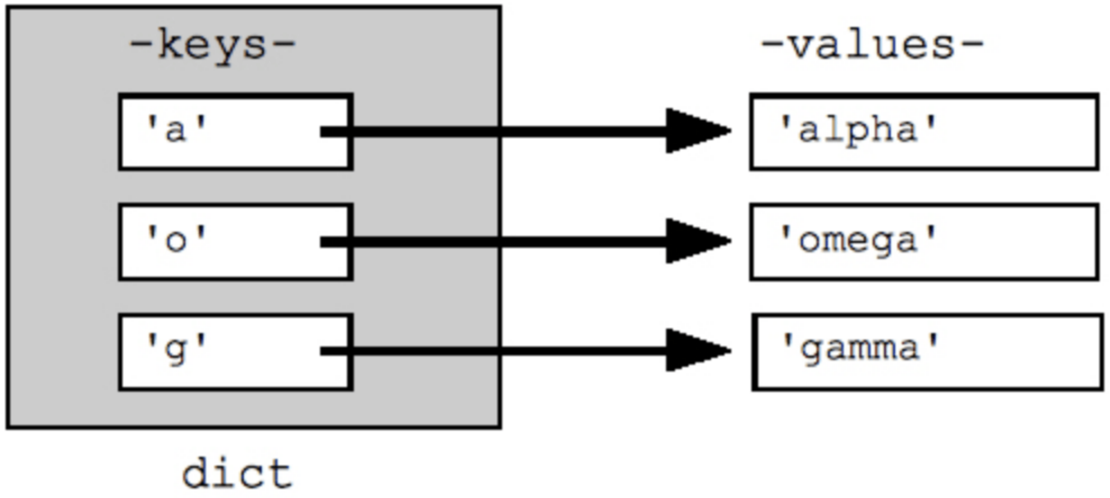


Figure 8: Dictionary as a Map [6]

- Does not store duplicate keys
- Mutable Object
- Keys should be hashable.

Dictionary *(contd.)*

Operation	Time Complexity (Worst Case)
Finding an element	O(1)
Copying the dictionary	O(n)
Inserting an element	O(1)
Get value from key	O(1)
Deleting an element	O(1)
Iterating all elements in a set	O(n)
Get length	O(1)

Linked Lists

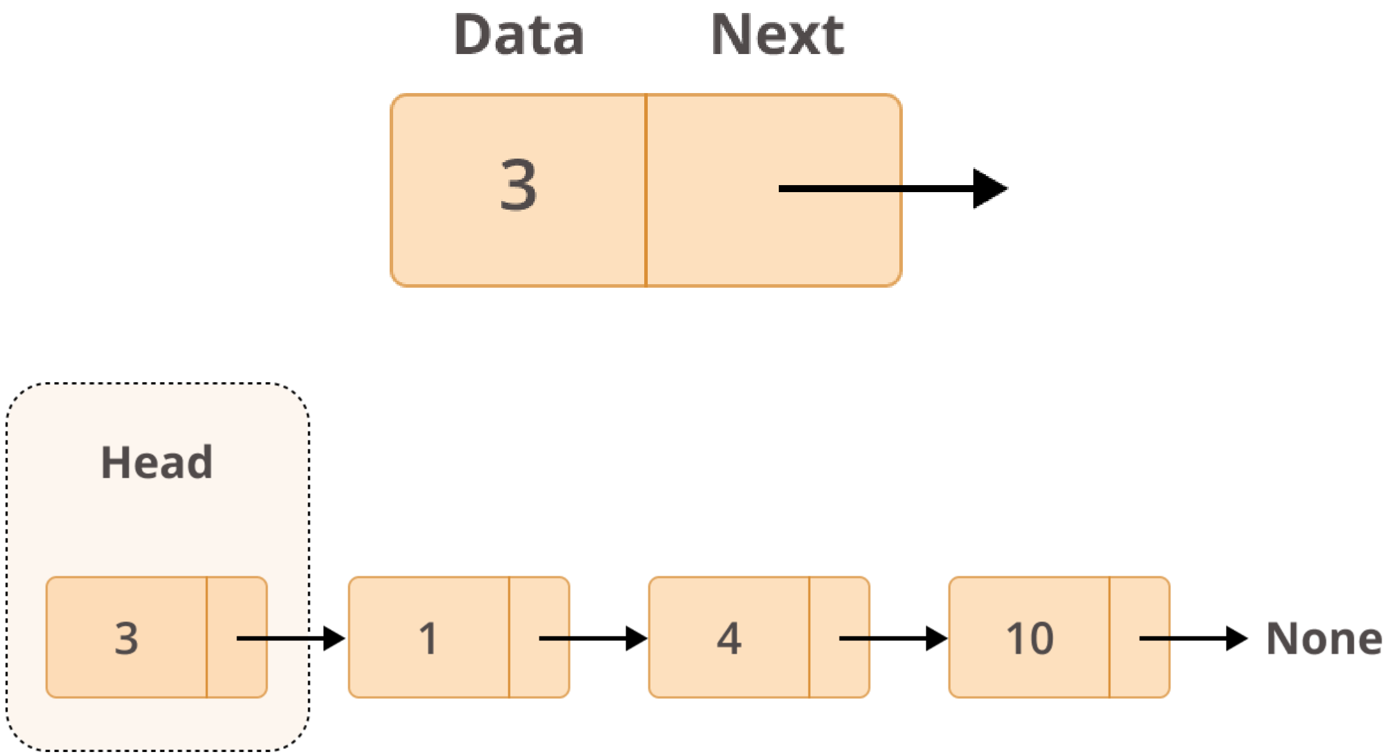
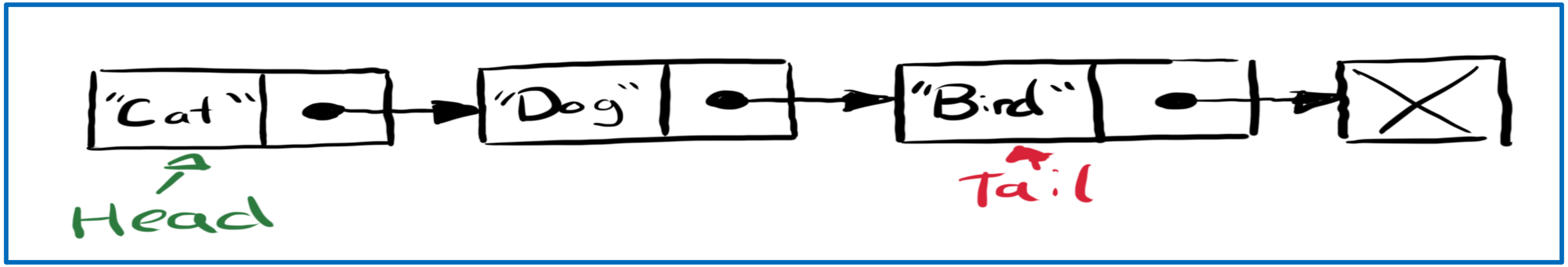


Figure 9: Node of a Linked Lists, Example of Singly Linked List [7]

Linked Lists *(contd.)*



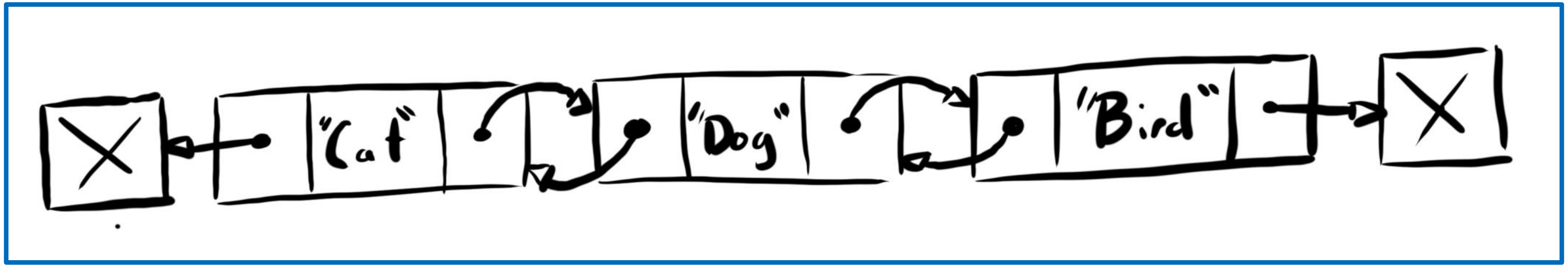


Figure 10: Singly and Doubly Linked Lists [8]

Linked Lists *(contd.)*

- Similar to Lists (Arrays) – The name has "List" in it, it was obvious!

- Memory Allocation:
    - Lists have sequential Memory Allocation
    - Linked Lists are stored in memory dynamically

- Insertion and deletion at beginning:
    - In Lists, it difficult and is usually O(n)
    - In LinkedList, it is very fast – usually O(1)

Stacks

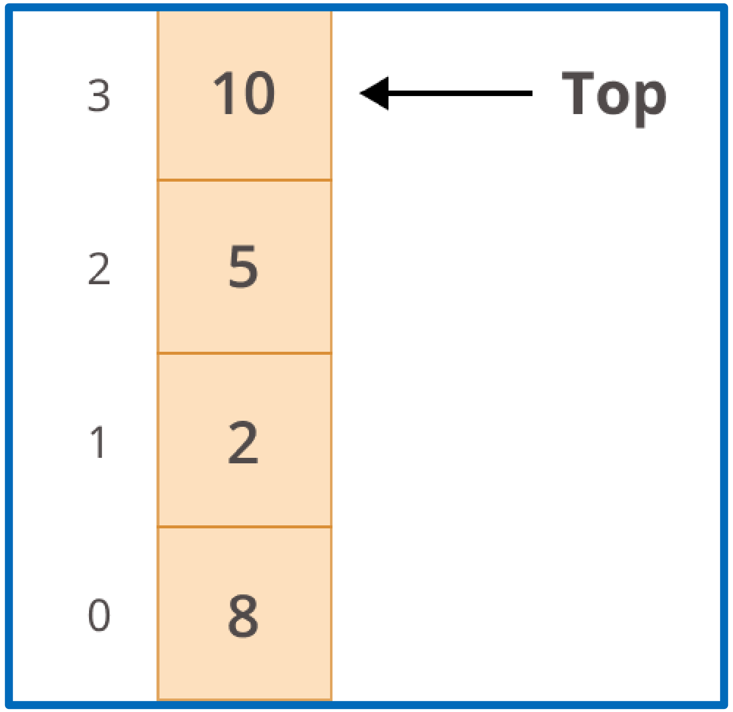


Figure 11: Linked Lists as Stack [7]

Stacks *(contd.)*

- Adding and deleting items only to and from one end!

- Follows the principle of LIFO (Last In First Out).

- "Top" always points to the topmost element of the stack.

- Basic Stack Operations:
    - Push (Insertion)
    - Pop (Deletion)
    - Top (Returns topmost element)
    - IsEmpty (Which returns TRUE if stack is empty)

Queues

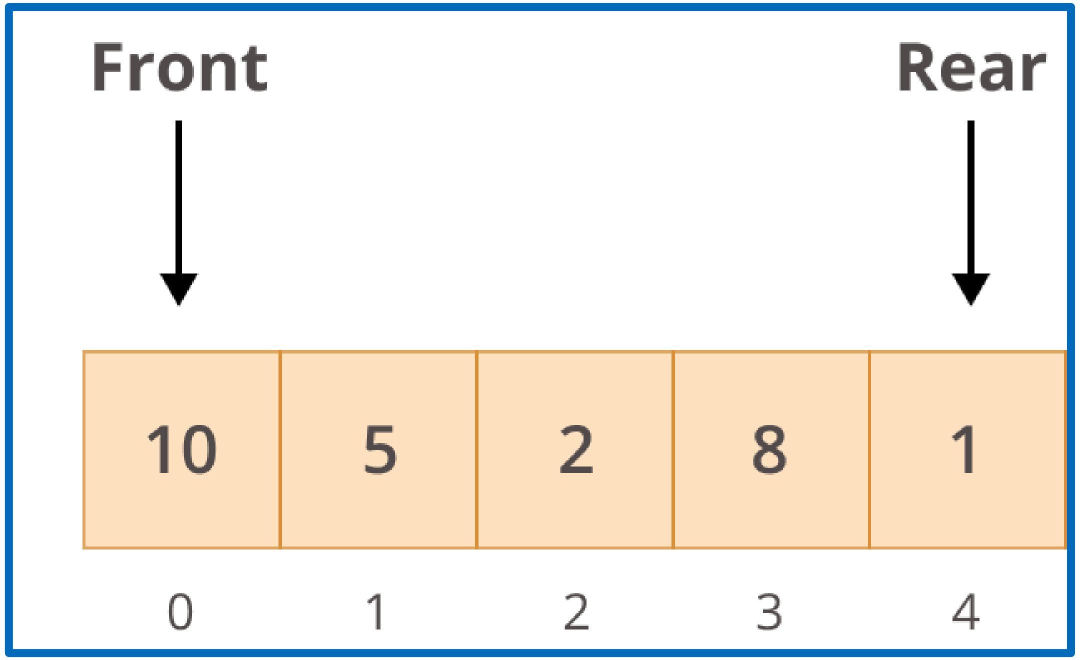


Figure 12: Linked Lists as Queue [7]

Queues *(contd.)*

- The 1st element is inserted from one end called the REAR *(also called tail)*, and the removal of existing element takes place from the other end called as FRONT*(also called head)*!

- Follows the principle of FIFO (First In First Out).

- "FRONT" always points to the first element of the Queue. "TAIL" always points to the last element in the Queue.

Queues *(contd.)*

- Basic Queue Operations:
  - Push (Insertion) (enqueue)
  - Popleft (Deletion) (dequeue)
  - Peek (Returns the first element of Queue without popping it)
  - IsEmpty (Which returns TRUE if queue is empty)

Linked Lists, Stacks, and Queues

Operation	Time Complexity (Worst Case)
Finding an element (Linked List)	O(n)
Copying the data-structure	O(n)
Appending	O(1)
Append left (Queue – enqueue)	O(1)
Pop	O(1)
Pop left (Queue – dequeue)	O(1)
Deleting an element (Linked List)	O(n)
Iterating all elements in a set	O(n)

References (1/2)

[1] https://www.intechopen.com/chapters/65828

[2] https://www.goodreads.com/user_status/show/286383699

[3] https://www.wattpad.com/959498332-origami-a-draco-malfoy-fan-fiction-chapter-1-page

[4] https://www.joyk.com/dig/detail/1608665582509127

References (2/2)

[5] https://en.wikipedia.org/wiki/Hash_function

[6] https://www.datacamp.com/community/tutorials/python-dictionaries

[7] https://realpython.com/linked-lists-python/

[8] https://www.mongodb.com/developer/how-to/introduction-to-linked-lists-and-mongodb/

Further Reading(s)