LINKED LISTS

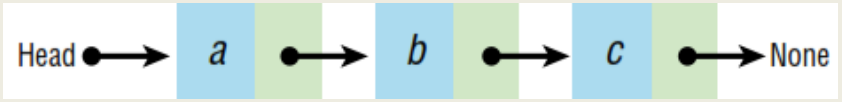
Mahsa Ziraksima
ACIBADEM UNIVERSITY
Fall 2025

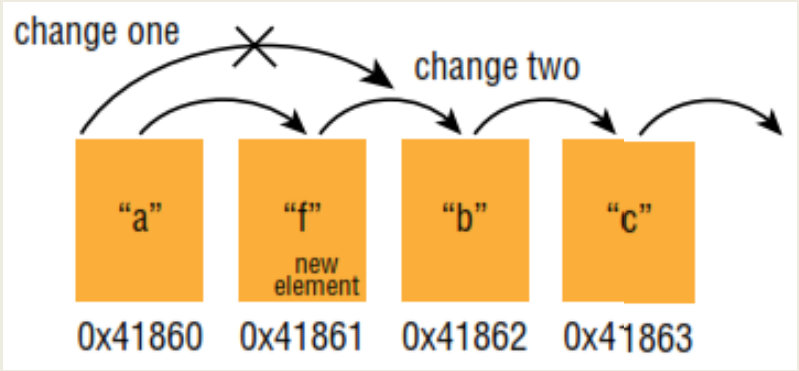
SUMMARY

- Linked List Performance
- Create a Linked List
- Search a Linked List
- Removing a Node from a Linked List
- Finding a Linked List Cycle

LINKED LIST

- It is another implementation of the list abstract data type, containing a chain of **nodes.**
- Elements in a linked list do not have indexes because your computer does not store the items in a linked list in sequential memory locations.
- The data in each node that stores the next node's location in the linked list is called a **pointer**.
- The first node in a linked list is called the **head**.
- The last node's pointer points to **None**.

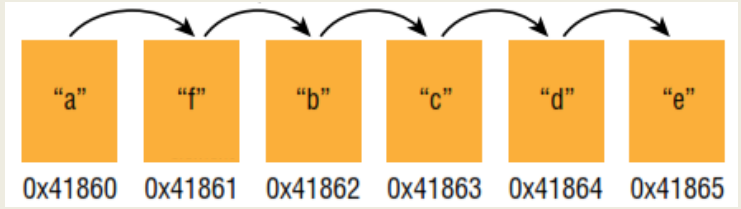




LINKED LIST

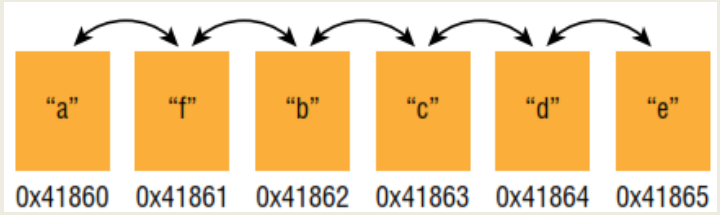
- Different types of linked lists:
  - ***Singly linked*** *is a list with pointers that point only to the next element.*

    Iterates by starting at the head
    and moving to the end.



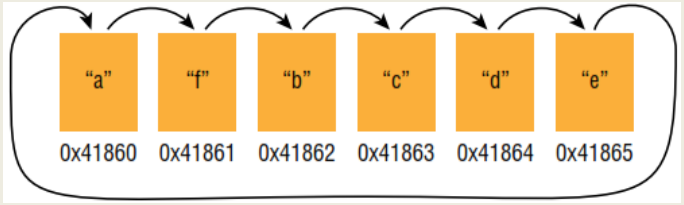
  - ***Doubly linked*** *list's nodes contain two pointers: one pointing to the next node and one pointing to the previous node.*

    Iterates from the head to the end
    also backward between nodes



  - ***Circular linked*** *list, has a last node which points back to the first node. which allows you to go from the last element of the list back to the front of the list.*

    Iterates from the head to the end
    also from the end to the head



LINKED LIST

- Performance:
  - *Accessing an item by linear search, O(n)*
  - *Adding and removing a node, O(1)*
  - *searching a linked list, like an array, O(n)*

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)

If you're writing an algorithm that adds and removes data often

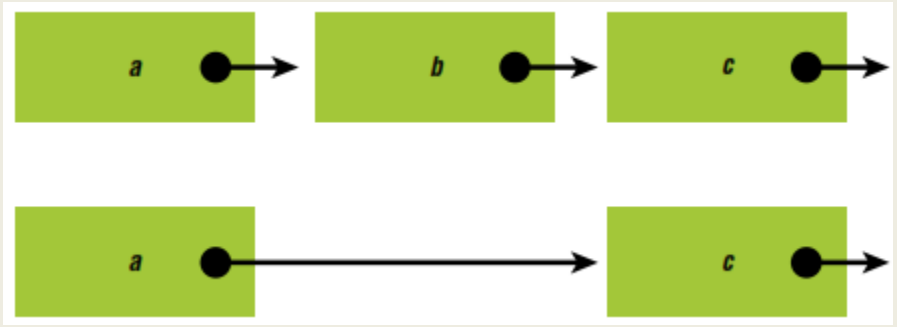
Each node must carry a pointer to the next node, using twice system resource.

*Not supporting **random access,** which means the ability to access any element in a collection directly, typically using an index.*

LINKED LIST

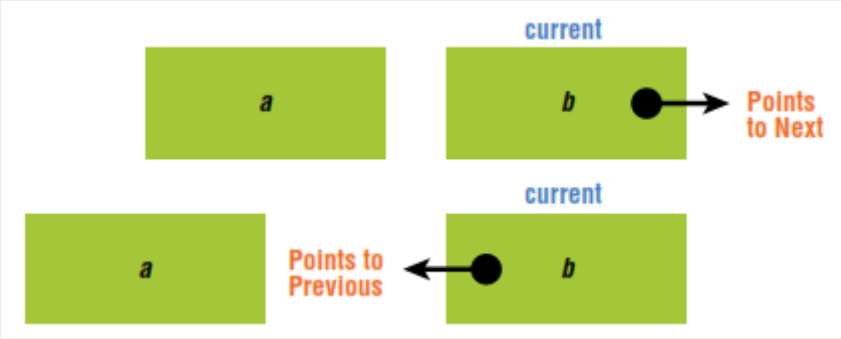
- For removing a Node from a Linked List, use a linear search to find a node in a linked list and delete it.

- You delete a node by changing the previous node's pointer so it no longer points to the node you want to delete.



LINKED LIST

- To reverse a linked list, you iterate through the list, keeping track of both the current node and the previous node. Then, you make the current node point to the previous node.



LINKED LIST

- In Python, you don't have to deal directly with memory addresses (like in the programming language C) because it handles that for you.
- When you create an object, like an instance of a class called Node, Python returns a pointer (or reference) to the object.
- This pointer is, essentially, an address of where the actual data resides in the computer's memory.
- When you assign objects to variables in Python, you are dealing with pointers (references), so you can easily link objects together because Python is doing all the underlying work for you.

LINKED LIST

```
class Node:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        if not self.head:
            self.head = Node(data)
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = Node(data)


a_list = LinkedList()
a_list.append("Tuesday")
a_list.append("Wednesday")
```

```
def printlinklist (self):
    node = self.head
    while node is not None:
        print(node.data, end= " ")
        node = node.next
    print("")

printlinklist(a_list)
```

```
Tuesday Wednesday
```

LINKED LIST

```
def search(self, target):
    current = self.head
    while current.next:
        if current.data == target:
            return True
        else:
            current = current.next
    return False

import random

a_list = LinkedList()

for i in range(0, 20):
    j = random.randint(1, 30)
    a_list.append(j)
    print(j, end= " ")
print("")
print(search(a_list, 10))
```

```
22 30 23 26 12 6 14 27 17 16 5 23 28 1 11 8 12 23 30 6
False
```

LINKED LIST

```
def remove(self, target):
    if self.head == target:
        self.head = self.head.next
        return
    current = self.head
    previous = None
    while current:
        if current.data == target:
            previous.next = current.next
        previous = current
        current = current.next
var =  a_list.head
printlinklist(a_list)
remove(a_list, var)
printlinklist(a_list)
```

```
22 30 23 26 12 6 14 27 17 16 5 23 28 1 11 8 12 23 30 6
30 23 26 12 6 14 27 17 16 5 23 28 1 11 8 12 23 30 6
```

LINKED LIST

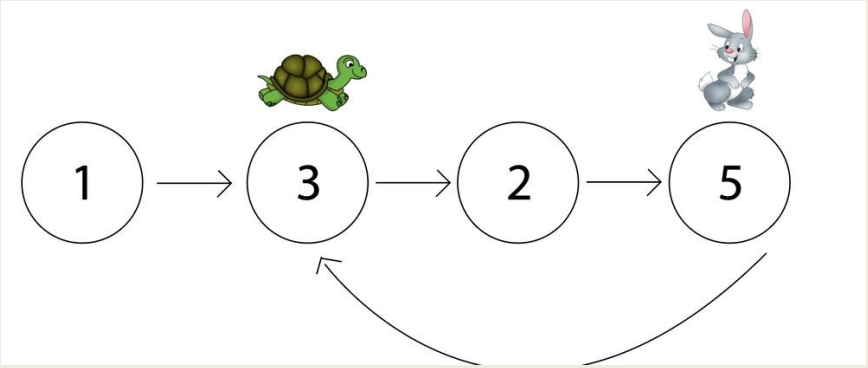
```
def reverse_list(self):
    current = self.head
    previous = None
    while current:
        next = current.next
        current.next = previous
        previous = current
        current = next
    self.head = previous

reverse_list(a_list)
printlinklist(a_list)
```

```
30 23 26 12 6 14 27 17 16 5 23 28 1 11 8 12 23 30 6
6 30 23 12 8 11 1 28 23 5 16 17 27 14 6 12 26 23 30
```

LINKED LIST

- To detect whether a linked list contains a cycle, the-*tortoise-and-the-hare algorithm is used.*
- In the algorithm, you iterate through your linked list at two different speeds, keeping track of nodes in a **slow** variable and a **fast** variable.
- If the linked list is a circle, eventually the fast variable will lap the slow variable, and both variables will be the same.
- If you reach the end of your linked list without it happening, you know it does not contain a cycle.



LINKED LIST

```
def detect_cycle(self):
    slow = self.head
    fast = self.head
    while True:
        try:
            slow = slow.next
            fast = fast.next.next
            if slow is fast:
                return True
        except:
            return False

print(detect_cycle(a_list))

var =  a_list.head.next.next.next
var.next = a_list.head
print(detect_cycle(a_list))
printlinklist(a_list)
```

```
3 19 5 8 11 20 4 13 21 17 25 5 7 21 24 13 25 24 24
False
True
3 19 5 8 3 19 5 8 3 19 5 8 3 19 5 8 3 19 5 8 3 19 5 8 3 19 5 8  ▪ ▪ ▪
```