Advanced Programming
INFO135

Lecture 2: Data Structures

Mehdi Elahi

University of Bergen (UiB)

Python *built-in* **Data Collections**

- **List**: ordered and mutable
- **Tuple**: ordered and immutable
- **Set**: unordered and mutable and unindexed
- **Dictionary**: unordered and mutable and indexed

Python *built-in* **Data Collections**

- **List**: ordered and mutable
- **Tuple**: ordered and immutable
- **Set**: unordered and mutable and unindexed
- **Dictionary**: unordered and mutable and indexed

Python **List**

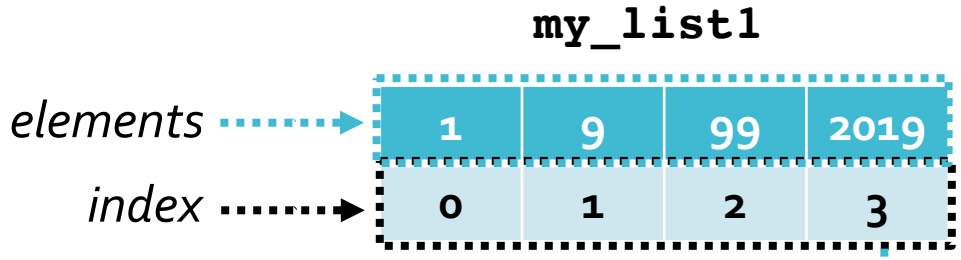
- Data collection that is **ordered** and **mutable**
- Can have **duplicates**
- Represented with **brackets** []

```
my_list = [1,9,99]
```

- **Appending** a new element to list:

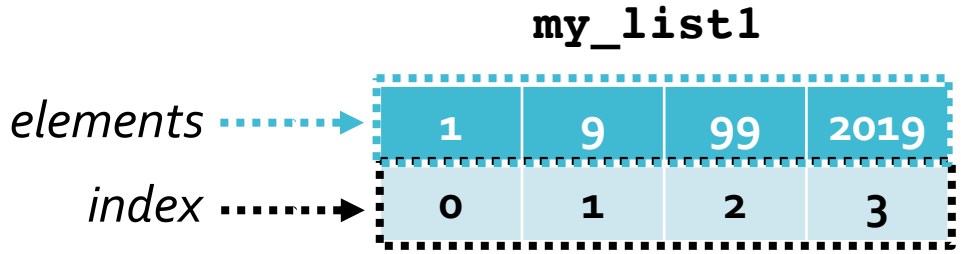
```
my_list.append(2019)
print(my_list)
[output]: [1, 9, 99, 2019]
```

Python **List**



- **Printing** a list:
  - `print(my_list1)`
  - **`[output]: [1,9,99,2019]`**
- **Accessing** elements:
  - `my_list1[3]: 2019`
  - `my_list1[-1]: 2019`

Python **List**

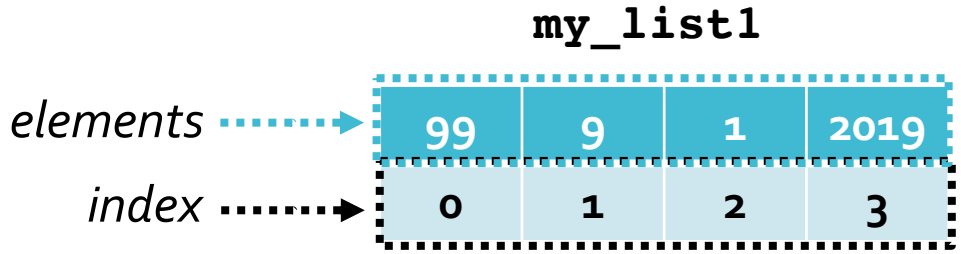


- **Slicing** a list:
  - `my_list1[0:2]:` **`[1,9]`**
  - `my_list1[1:]:` **`[9,99,2019]`**

- **Length** of list:
  - `len(my_list1):` **`4`**

- **Counting** the # of an element in the list:
  - `my_list1.`**`count`**`(99):` **`1`**

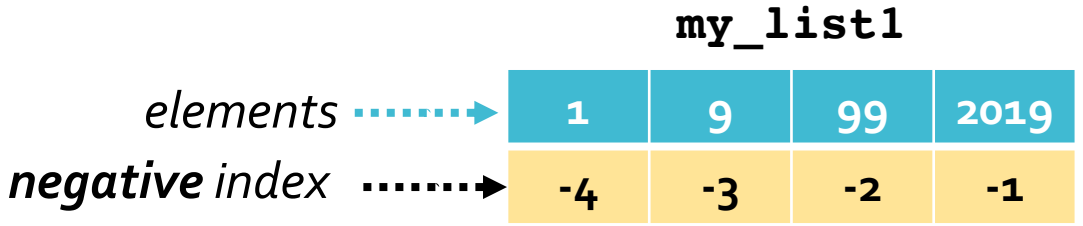
Python **List**



- **Sorting** a list:
  a) #method 1
     - `my_list1.sort(): [1,9,99,2019]`
  b) #method 2
     - `sorted(list1): [1,9,99,2019]`

- **Reversing** a list:
  - `my_list1.reverse(): [2019, 1, 9,99]`

Python **List**



- Further **slicing** a list:
  - `my_list1[-3:-1]: [9, 99]`
  - `my_list1[:-1]: [1, 9, 99]`
  - `my_list1[-3:]: [9, 99,2019]`

Quiz

- Which one is **correct**?
  a. `my_list1 = [1,99,[1,2020]]`
  b. `my_list2 = ["I","love","python"]`
  c. `my_list3 = [1,"love","python"]`
  d. `my_list4 = [1,"love","python", [1,2020]]`
  e. All

- How can we **clone** or **copy** a list in Python?

Answer

- Which one is **correct**?

```
a.my_list1 = [1,99,[1,2020]]
b.my_list2 = ["I","love","python"]
c.my_list3 = [1,"love","python"]
d.my_list4 = [1,"love","python", [1,2020]]
```

✅ e. **All**

- How can we **clone** or **copy** a list in Python?

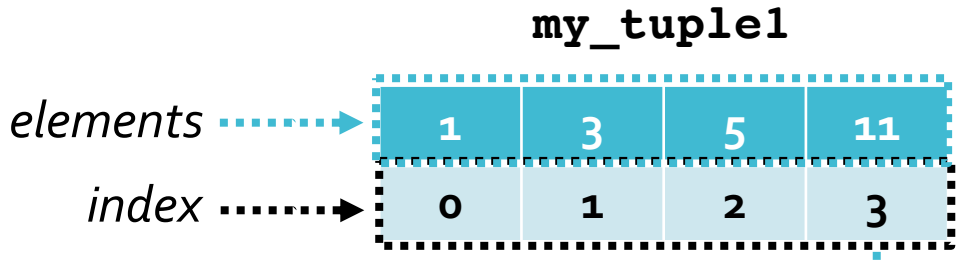
```
original_list = [5, 3, 2, 0, 7, 100, 99]
new_list = original_list.copy()

print(original_list)
print(new_list)
```

Python Tuple

- Data collection that is **ordered** and **immutable**
- Can have **duplicates**
- Represented with **parenthesis** ()

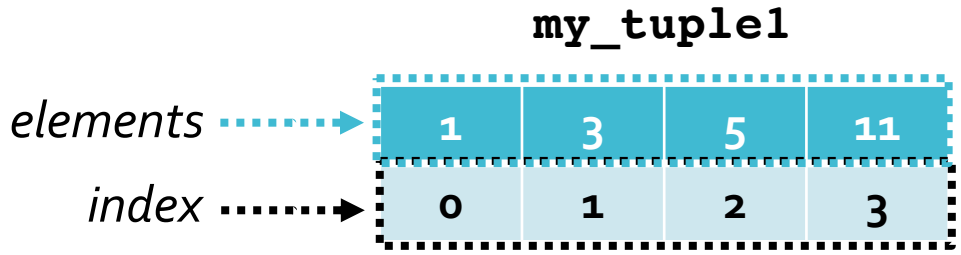
```
my_tuple1 = (1,3,5,11)
```

Python Tuple



- **Creating** a tuple:
  - `my_tuple1 = (1,3,5,11)`
- **Accessing** elements:
  - `my_tuple1[3]: 11`
- **Printing** a tuple:
  - print(`my_tuple1`): `(1,3,5,11)`

Python Tuple



- **Slicing** a tuple:
  - `my_tuple1[0:2]: (1,3)`
  - `my_tuple1[1:]: (3,5,11)`

- **Length** of tuple:
  - `len(my_tuple1): 4`

Python *built-in* **Data Collections**

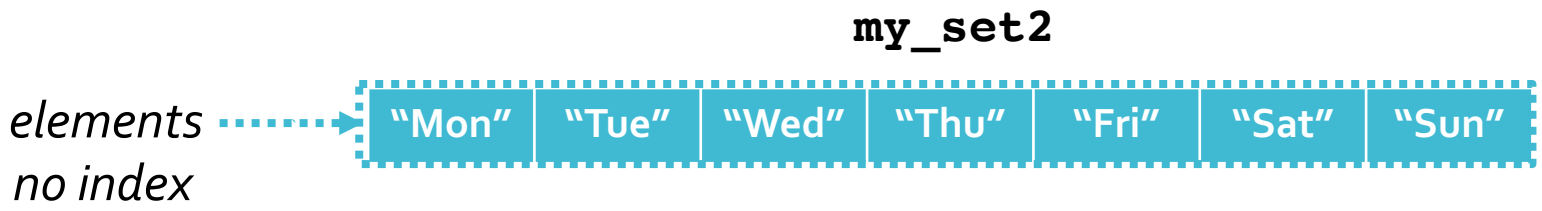
- **List**: ordered and mutable
- **Tuple**: ordered and immutable
- **Set**: unordered and mutable and unindexed
- **Dictionary**: unordered and mutable and indexed

Python Set

- Data collection that is **unordered**, **mutable**, and **unindexed**
- **No duplicates**
- Represented with with **curly bracket** {}
- Examples:

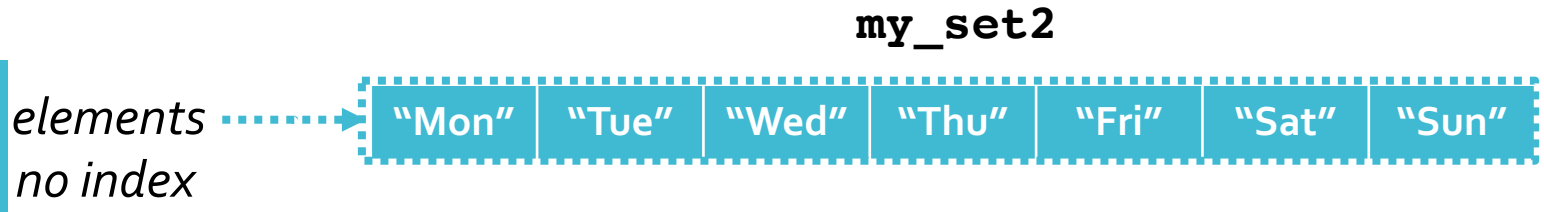
```
my_set1 = {3.14, 33, 5, 3}
my_set2 = {3.14,"Mon",3,"Sun"}
my_set3 = {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"}
```

Python Set

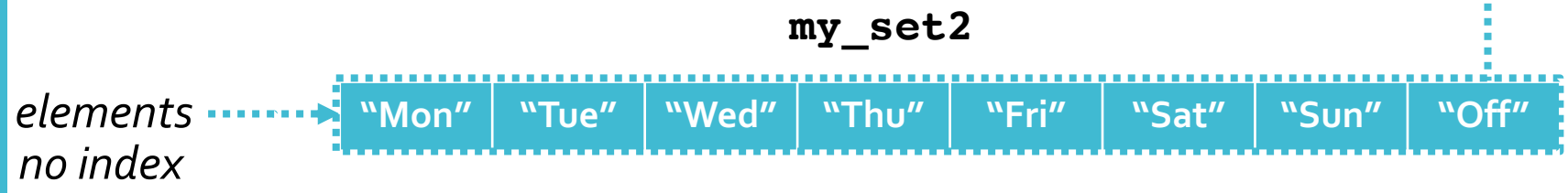


- **Creating** a set:
  - `my_set2 = {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"}`
- **Printing** a set:
  - `print(my_set2):`
    `{"Mon","Tue","Wed","Thu","Fri","Sat","Sun"}`
- **Length** of set:
  - len(my_set2): 7

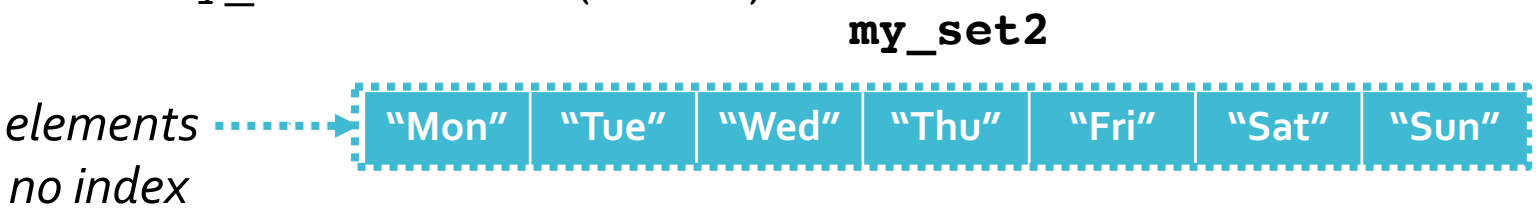
Python Set



- **Adding** elements
  - `my_set2.add("Off")`



- **Removing an** element:
  - `my_set2.remove("Off")`



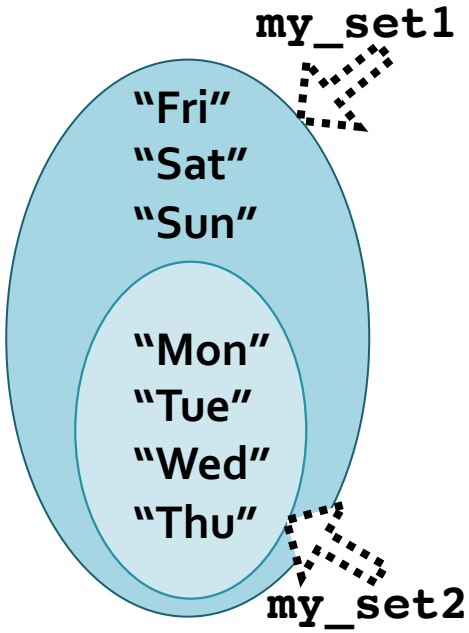
Python Set

my_set1

"Mon"	"Tue"	"Wed"	"Thu"	"Fri"	"Sat"	"Sun"

my_set2

"Mon"	"Tue"	"Wed"	"Thu"



- **Intersection** of sets:

```
my_set3 = my_set1 & my_set2
```

my_set3

"Mon"	"Tue"	"Wed"	"Thu"

Quiz

1) What is the **output** of the following code:

```
my_set2 = {"Mon","Tue","Wed","Thu","Fri","Fri"}
print(my_set2)
```

2) How to add **multiple new elements** to the following set?

```
my_set3 = {1,2,3,4,5,6}
```

3) How can we create a **union of two sets**?

Answers

1) What is the **output** of the following code:

```
my_set2 = {"Mon","Tue","Wed","Thu","Fri","Fri"}
print(my_set2)
```

[output]{"Mon","Tue","Wed","Thu","**Fri**"}

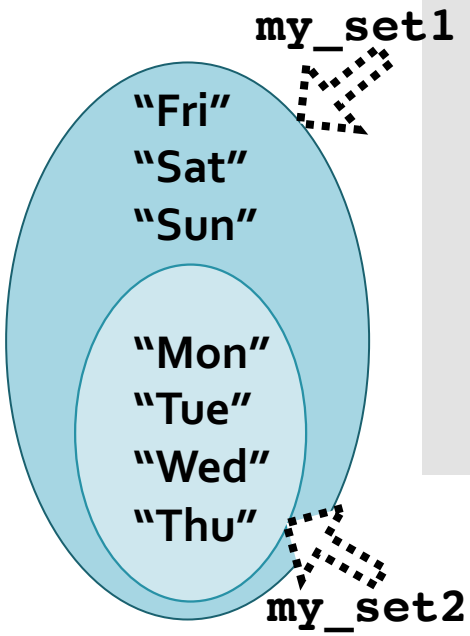
2) How to add **multiple new elements** to the following set**?**

```
my_set3 = {1,2,3,4,5,6}
my_set3.update([7,8])
print(my_set3)
```

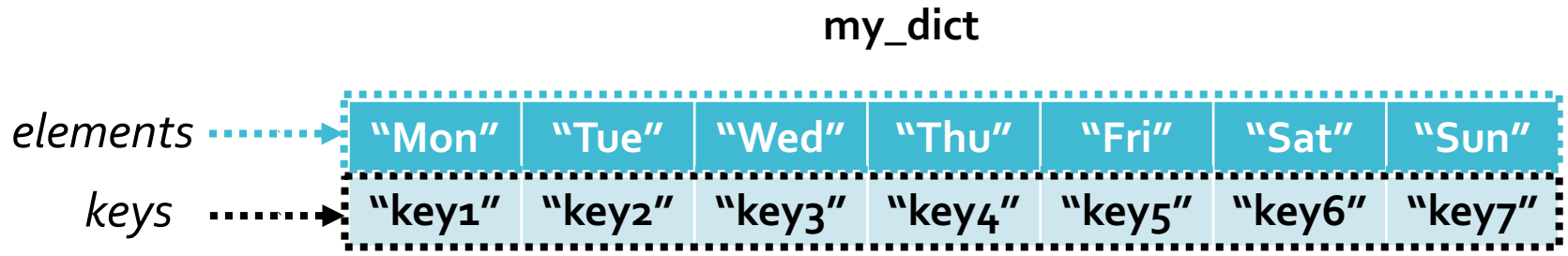
[output]{1,2,3,4,5,6,**7,8**}

3) How can we create a **union of two sets**?

```
union_set = my_set1 | my_set2
```



Python Dictionary

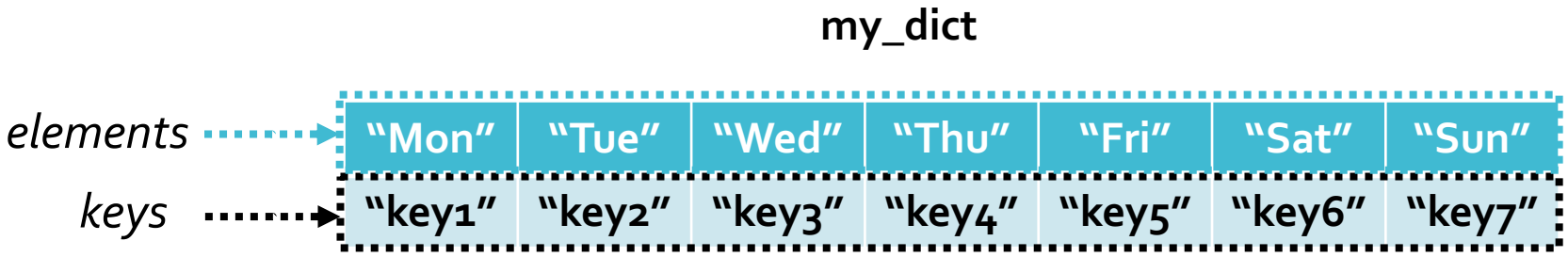


- Data collection that is **unordered**, **mutable**, and **indexed**
- Can have **duplicates**
- Represented with **curly brackets** and specifying the **key & value** pairs {key:value}

- Example:

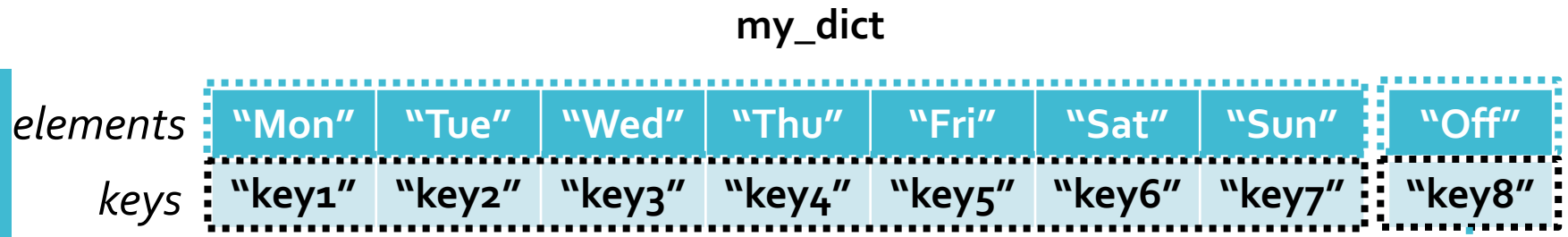
```
my_dict =
{"key1":"Mon","key2":"Tue","key3":"Wed",
 "key4":"Thu","key5":"Fri","key6":"Sat",
 "key7":"Sun"}
```

Python Dictionary



- **access** an element a dictionary
  - my_dict["key3"]
- get **keys** of a dictionary
  - my_dict.**keys**()
- check **if a key exists** in a dictionary:
  - "key1" **in** my_dict

Python Dictionary



- **add** to a dictionary
  - my_dict.**update**({"key8":"Off"})
- **delete** from a dictionary
  - **del**(my_dict["key8"])

Quiz

1) How can we **merge** the following two dictionaries?

```
student_dict1 = {1:“John”,2:”Sara”,3:”Ivan”}
student_dict2 = {4:”Lara”,5:”Mary”}
```

2) If we have the **list of keys** and **list of values,** separately, how can we create the **dictionary**?

```
list_of_keys = ["key1","key2","key3”]
list_of_values = ["value1","value2","value3”]
```

Answers

1)How can we **merge** the following two dictionaries?

```
student_dict1 = {1:“John”,2:”Sara”,3:”Ivan”}
student_dict2 = {4:”Lara”,5:”Mary”}

merged_dict = student_dict1.copy()
merged_dict.update(student_dict2)
```

Answers

2) If we have the **list of keys** and **list of values,** separately, how can we create the **dictionary**?

```
list_of_keys = ["key1","key2","key3”]
list_of_values = ["value1","value2","value3”]

zip_pairs = zip(list_of_keys,list_of_values)
my_dict = dict(zip_pairs)

print(my_dict)
```

[output]:

```
{'key1':'value1', 'key2':'value2', 'key3':'value3'}
```

Data Structures

- **Data Structures** are basically methods of storing data.
  - **Static** Data Structures have fixed size and when declared, the size do not change (e.g., **Array**).
  - **Dynamic** Data Structures can grow and shrink in size while the program is running (e.g., **List**).
- **Static Data Structures:**
  - programmer must **estimate the largest memory** space needed (the space can be wasted).
- **Dynamic Data Structures**:
  - uses only the **memory space needed** (efficient memory usage)

What computer memory looks like!

- Imagine you need to **store things** in some drawers.
- **One drawer** can hold **one thing**.
- You remember **which drawer** stored **which thing**.
- This is basically **an address** for that thing.

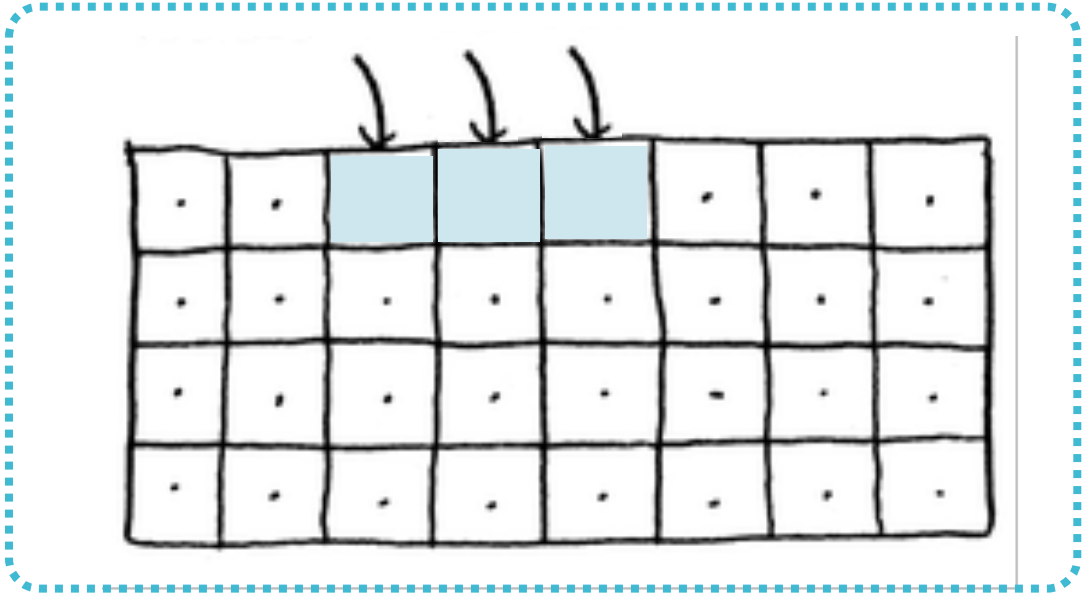
What computer memory looks like!

- **Computer memory** looks like similar.
- Computer memory looks like a giant set of drawers where **each** of them **has an address.**

computer memory

Computer Memory

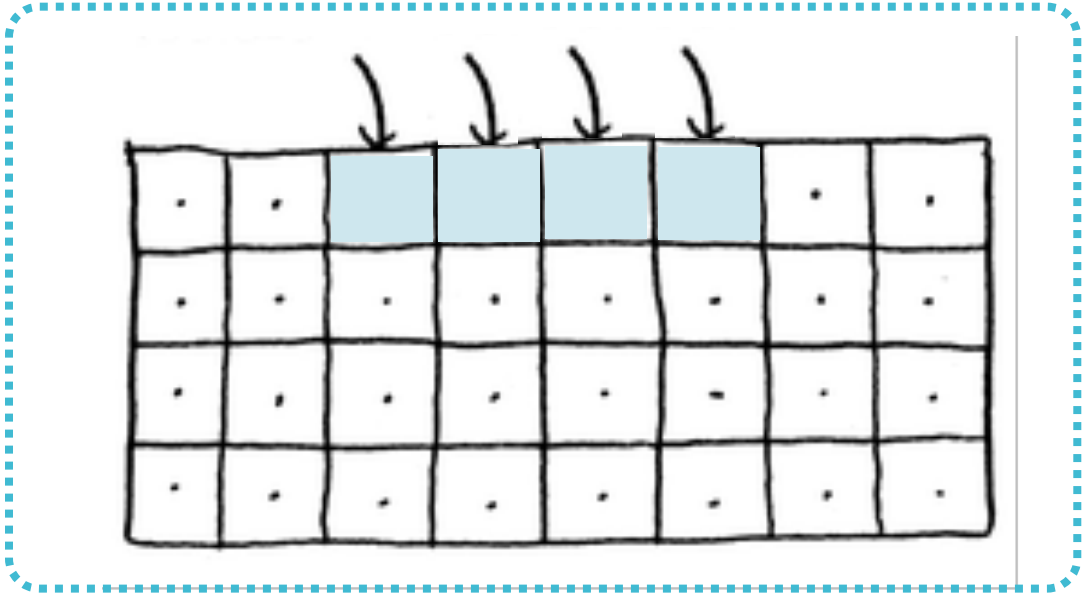
- When you need to store an item in memory, computer gives you **an address**.
- But what about storing **multiple items**, such as a **collection** of things?



computer memory

Computer Memory

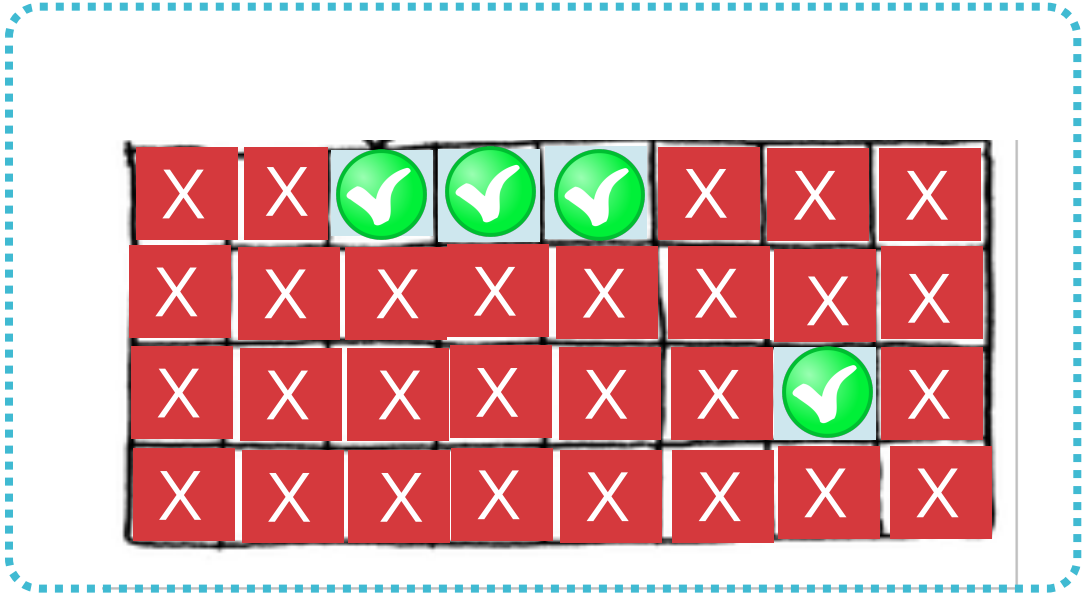
- What if you need even **more space**?
- Well … you can keep allocating new memory units.



computer memory

Computer Memory

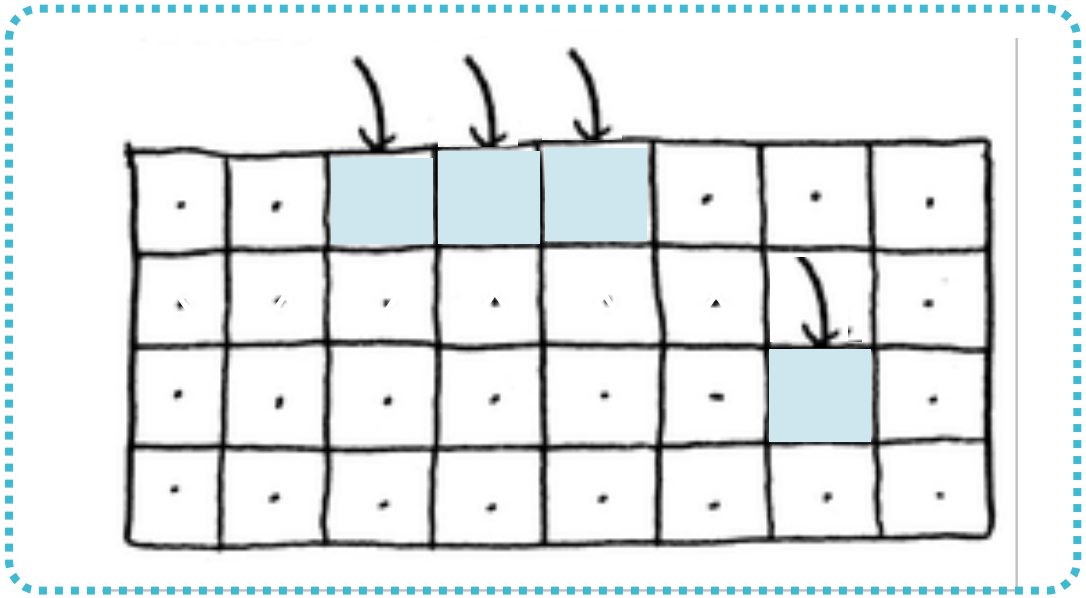
- What if the only **available memory units** are only the followings?



computer memory

Computer Memory

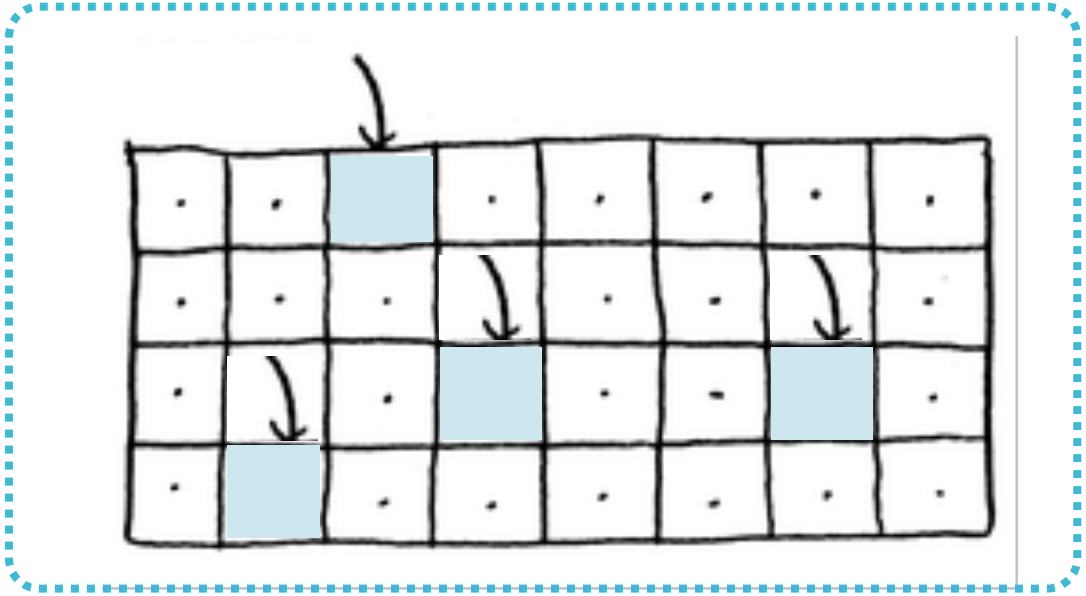
- What if the only **available memory units** are only the followings?



computer memory

Computer Memory

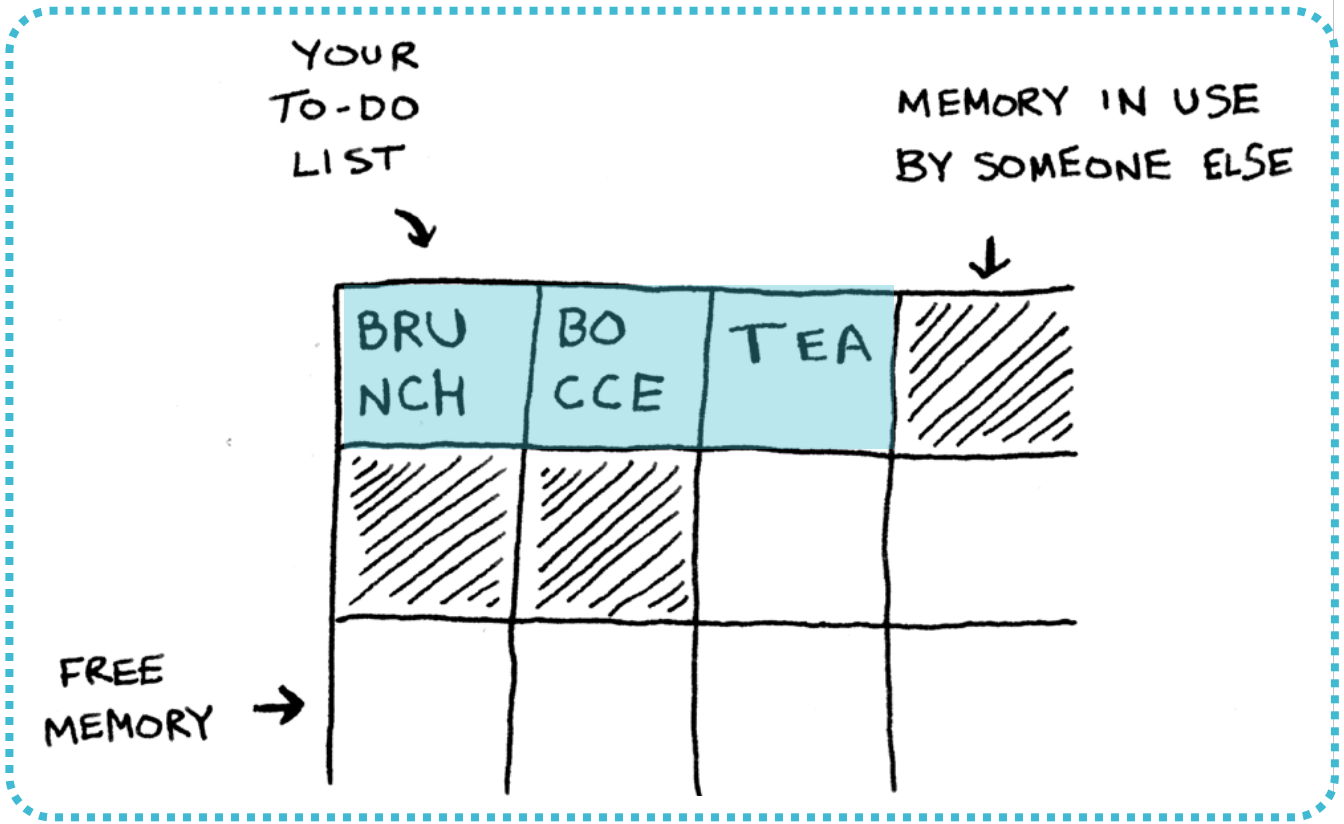
- Or even if the followings?
- What should we do for storage?



computer memory

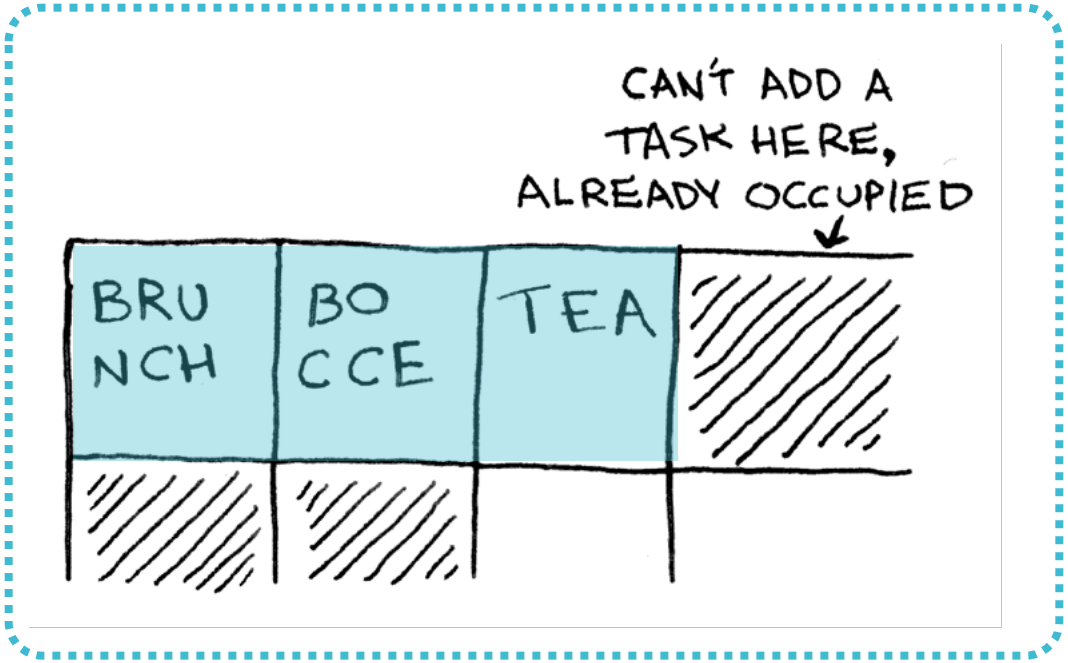
Computer Memory

- Lets look at an **example**:
  - You want to write a computer program that manages your **everyday *to-do* tasks**.
  - You need to store a collection of ***to-do* tasks** in computer memory.



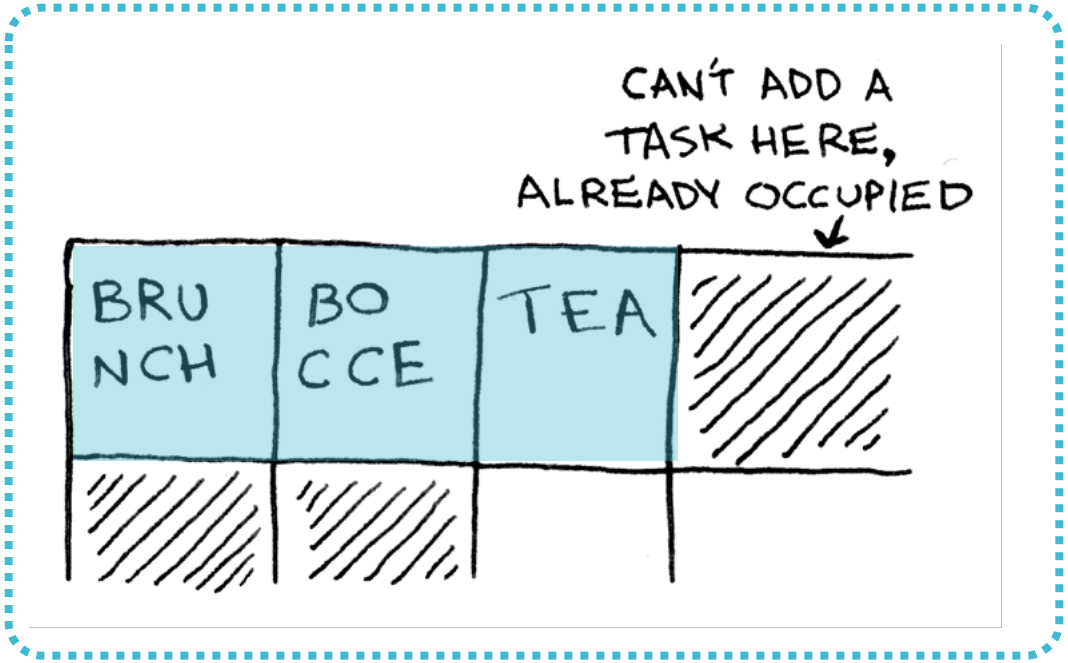
Array vs Linked-list

- Two possibilities:
  - **Array** (Python list resembles Array but it is *resizable*)
  - **Linked-list**
- **Array:** all your tasks are **stored contiguously** (right next to each other) in memory.



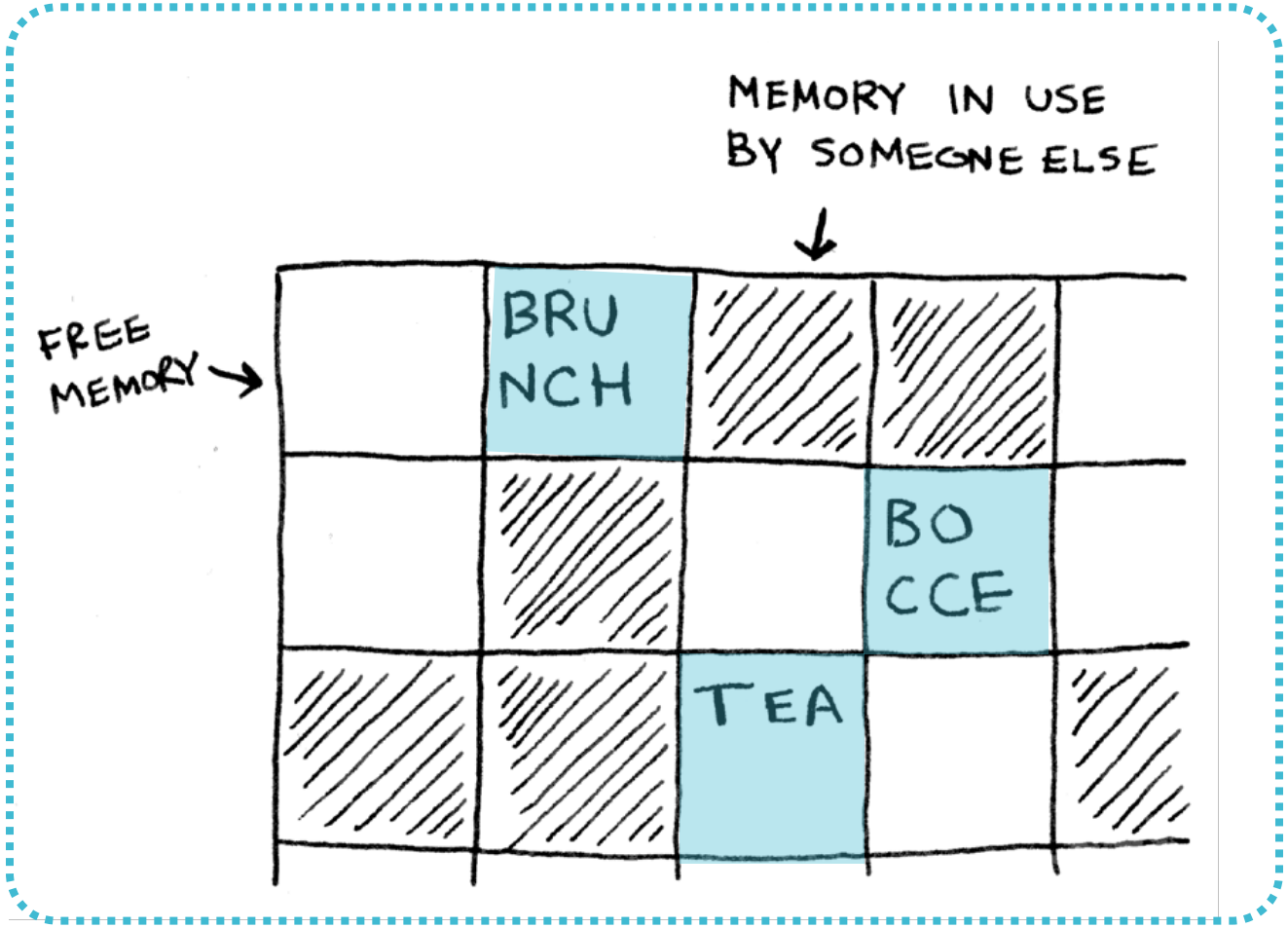
Array

- But if **memory is occupied**, then you need to move to a new spot in memory.
- You can try to ***hold seats***, i.e., to ask the computer for more units.
- And you may **waste** the memory.



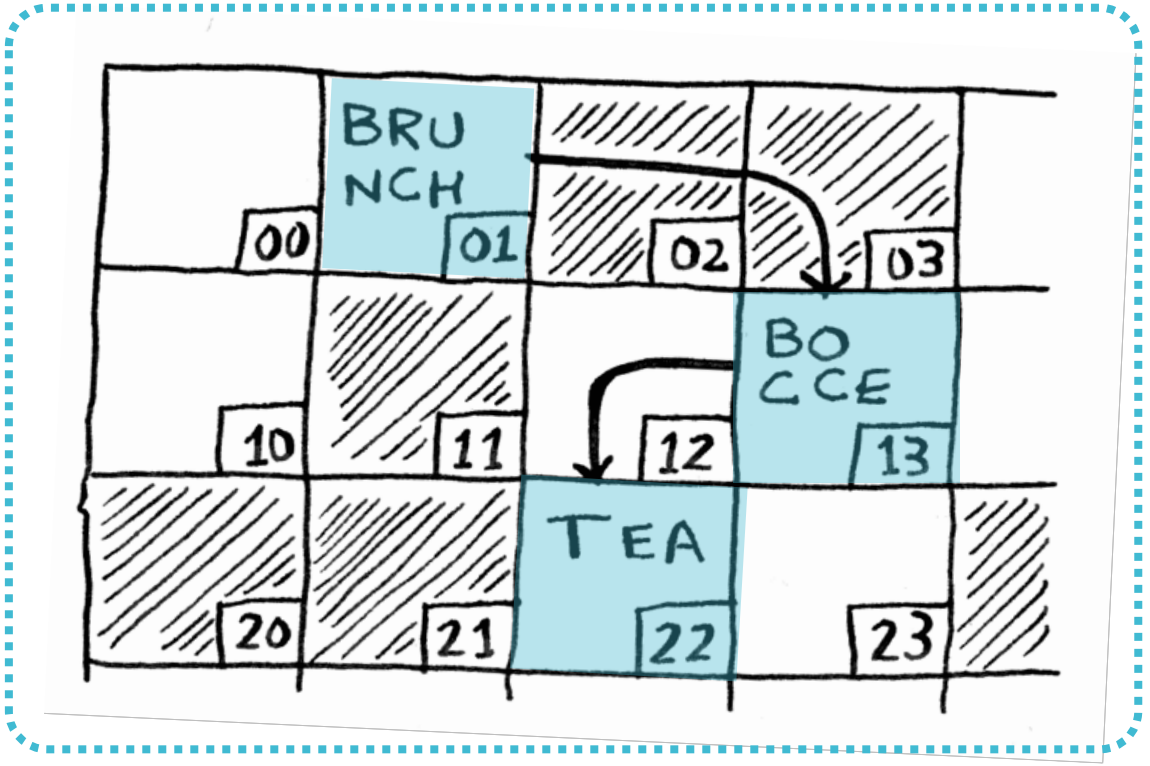
Linked-list

- Your items can be "**Any Where"** in memory.
- Each item stores **address** that links to the **next item** in list.



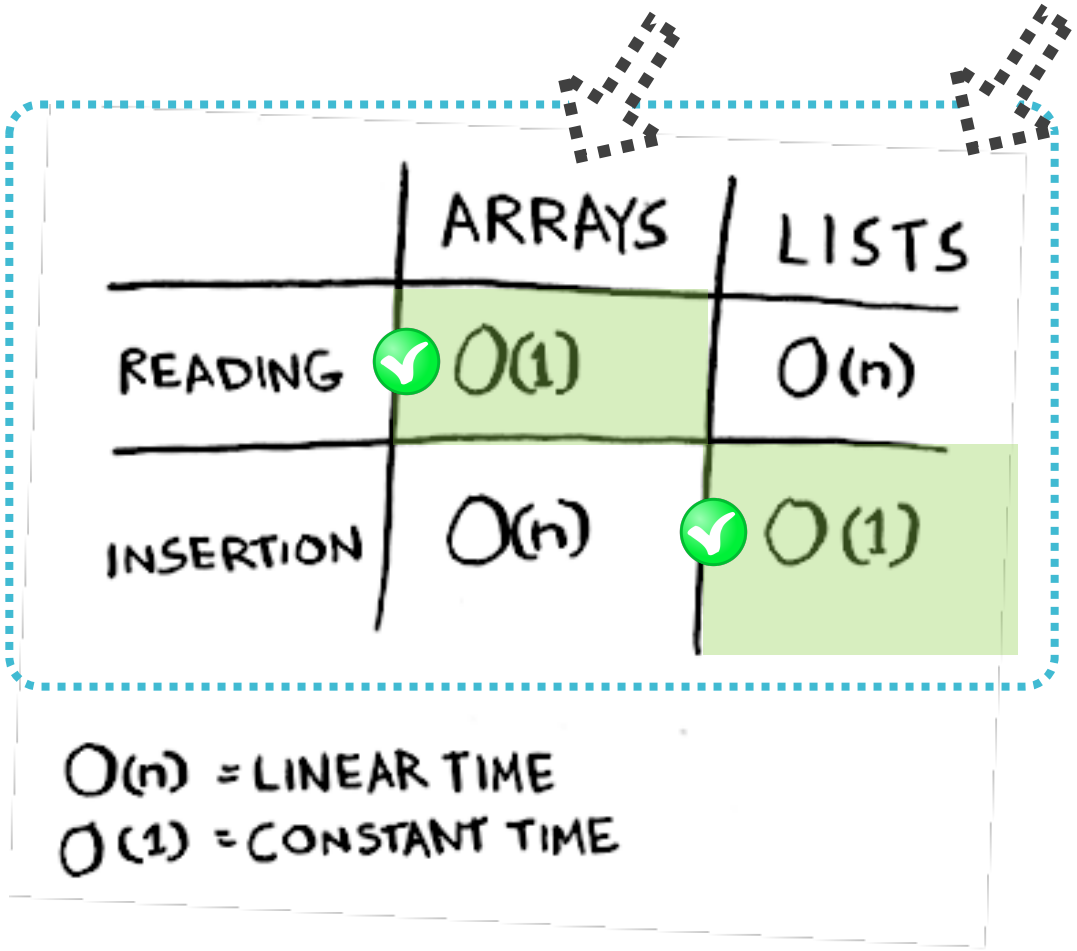
Linked-list

- Basically, (random) **addresses** are **linked** together.
- Then, you **never** need to **move** your items.
- This is great if you're reading your items **one at a time.**



Array vs Linked-list

- Run times for operations on **Array** versus **Linked-list**.



Quiz

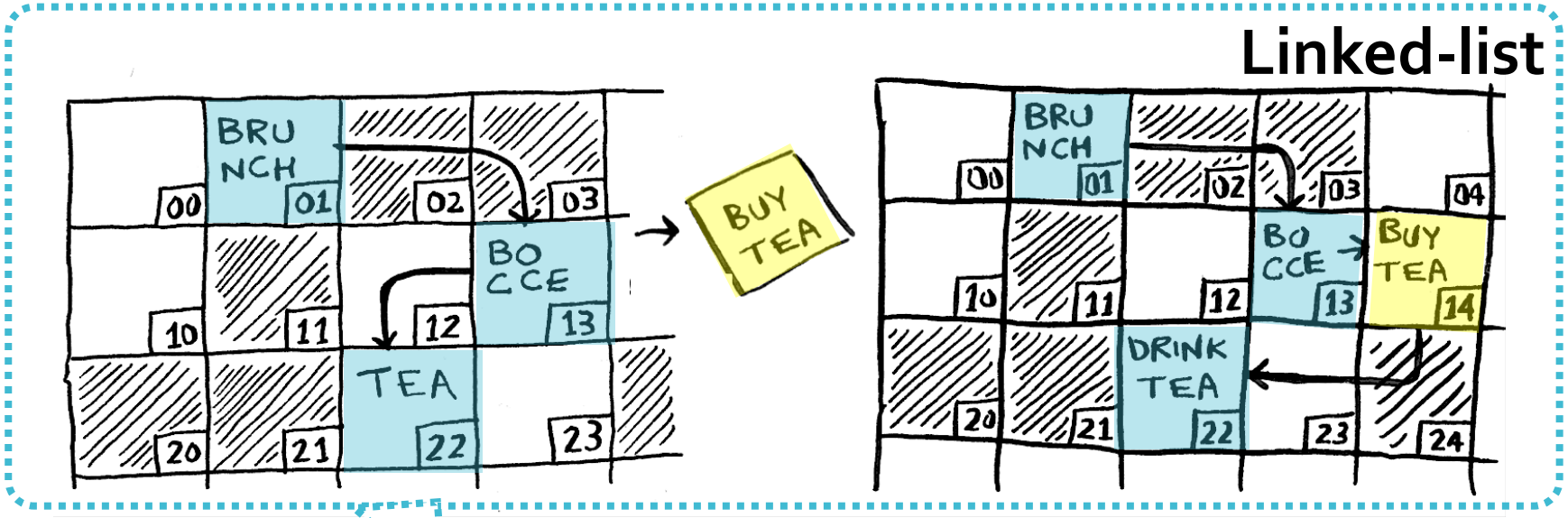
- Suppose your tasks are **ordered** according to their time (like a calendar).
- If you add a new task "Buy Tea" **in the middle**.
- Which one is better: **Array** or **Linked-list**? Why?

Answer

- **Answer:**
- **Linked-list** is a better and easier option.

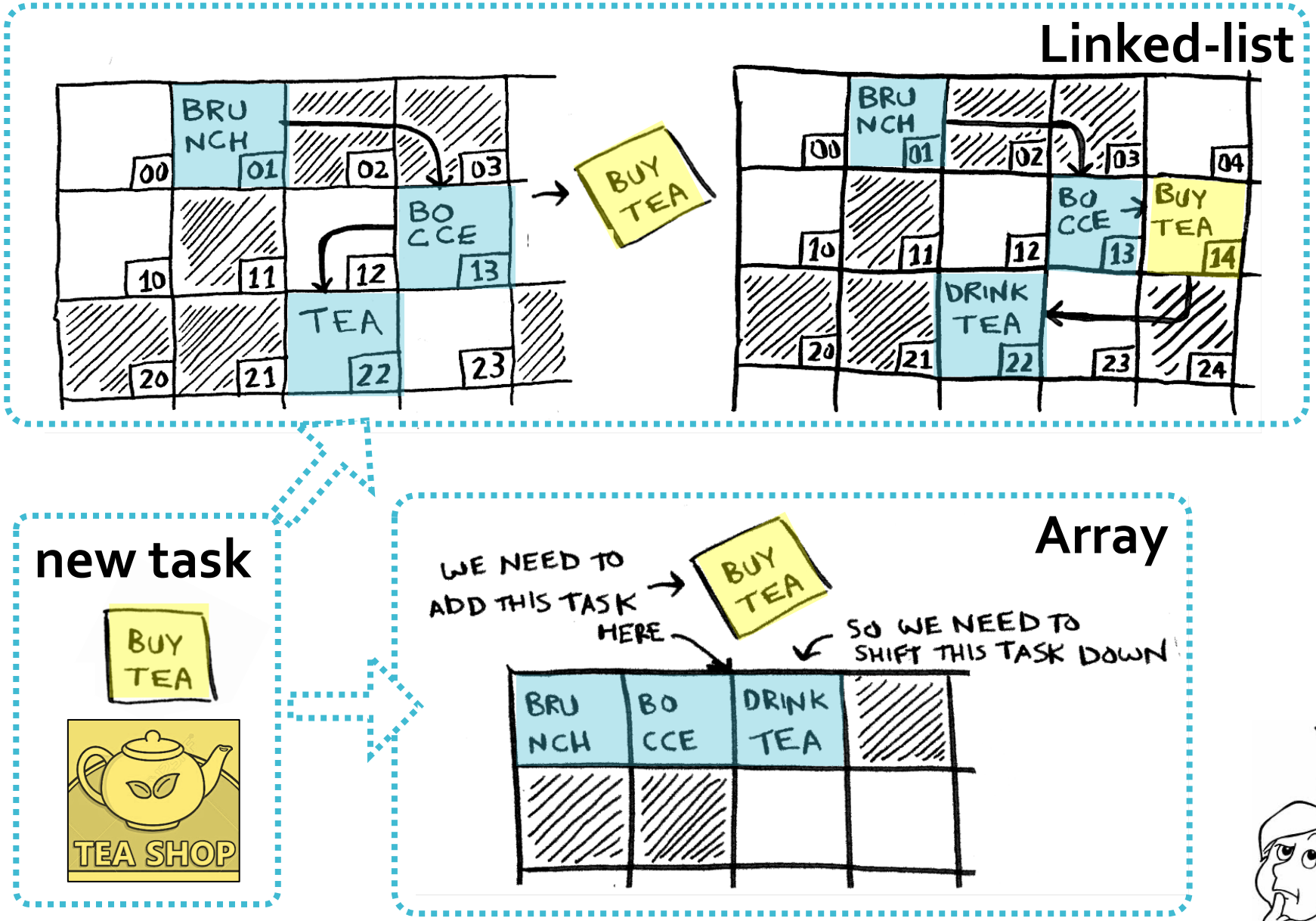
- Because:
- **Linked-list:**
  - you just change where the previous element **points** to!
- **Array:**
  - you must shift all the next elements.
  - If there's no space, you might have to copy everything to a new location!

Answer


Linked-list
BRU NCH
BO CCE
TEA
BUY TEA
DRINK TEA
00
01
02
03
04
10
11
12
13
14
20
21
22
23
24


new task
BUY TEA
TEA SHOP

Answer


Linked-list
BRU NCH
BO CCE
TEA
BUY TEA
DRINK TEA
00
01
02
03
04
10
11
12
13
14
20
21
22
23
24
new task
BUY TEA
TEA SHOP
Array
WE NEED TO ADD THIS TASK HERE
BUY TEA
SO WE NEED TO SHIFT THIS TASK DOWN
BRU NCH
BO CCE
DRINK TEA

Deletion

- What about **deleting a task?**

- **Linked-list:** the place the previous element points to changes.
- **Array:** everything needs to be shifted (moved).
- again **Linked-list is better!**
- **Note:** insertions may fail if there's no space left in memory. But delete always works.

Run time

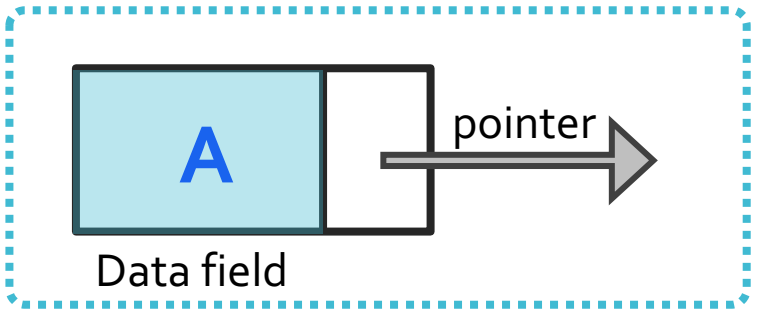
- **Run times** for operations on **Array** and **Linked-list**.

- **O(1): constant** time
- **O(n): linear** time

- **Note:** insertions and deletions are O(1) if you can instantly access the element to be deleted.

Linked-list

- The basic building block for the linked list is the **Node**.
- Each node contains:
  - **data field** of the node.
  - **reference** to the next node



Node

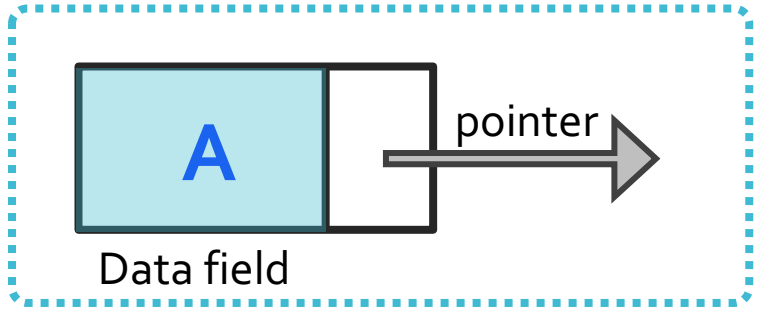
Linked-list

- Implementation of a simple **Node**

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

Data field

Pointer



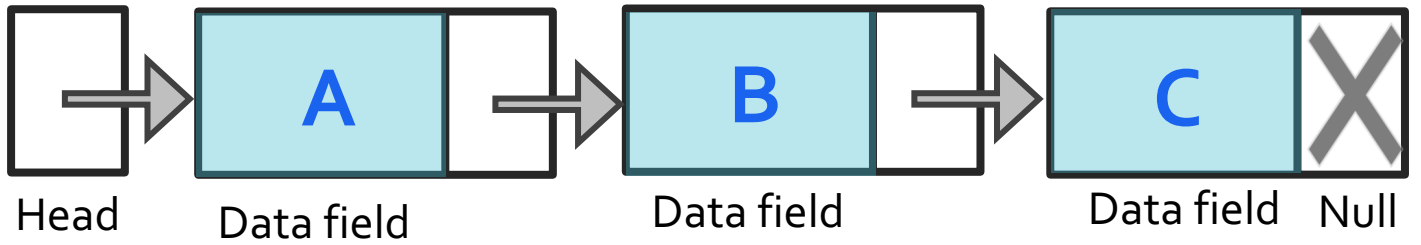
Node

Linked-list

- Nodes are singly **linked** in **one direction** only.

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


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

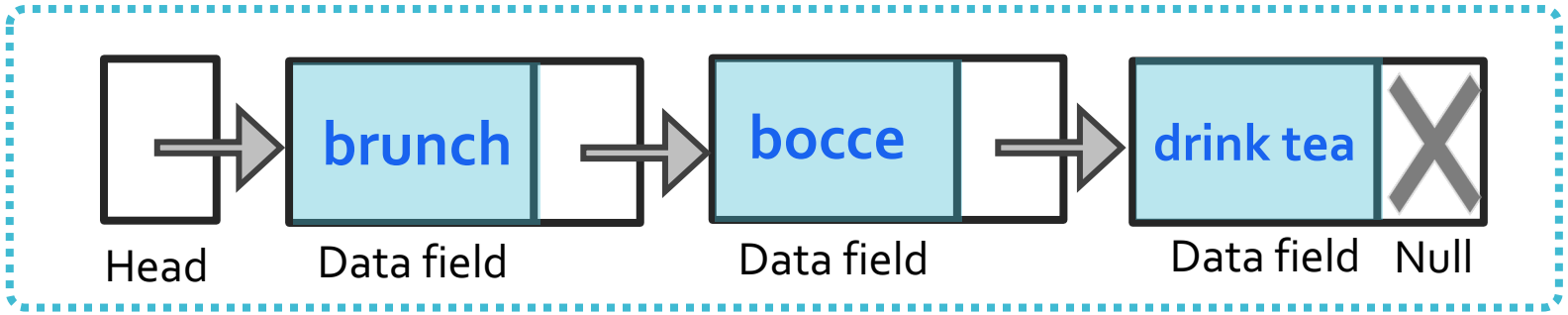
Singly Linked-list



Linked-list

- Example: Lets build our **todo linked-list!**

To do



Singly Linked-list

Linked-list

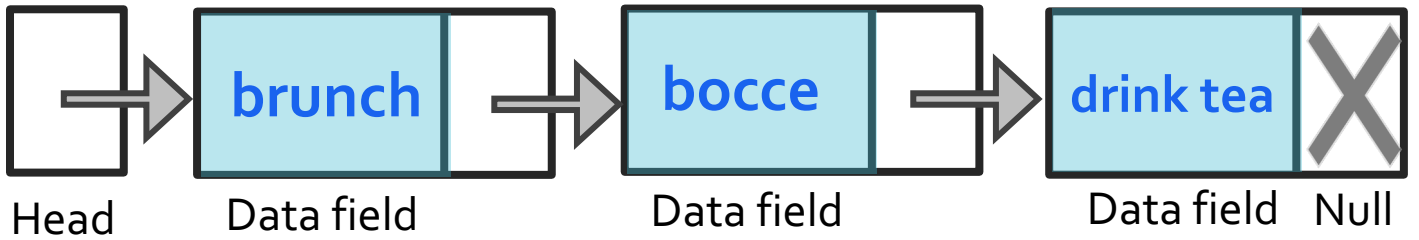
❖ Example: Lets build our **todo linked-list!**

```
node1 = Node("brunch")
node2 = Node("bocce")
node3 = Node("drink tea")

todo_list = LinkedList()
todo_list.head = node1
node1.next = node2
node2.next = node3
```

creating linked-list object

pointing to node 1

Singly Linked-list



Linked-list

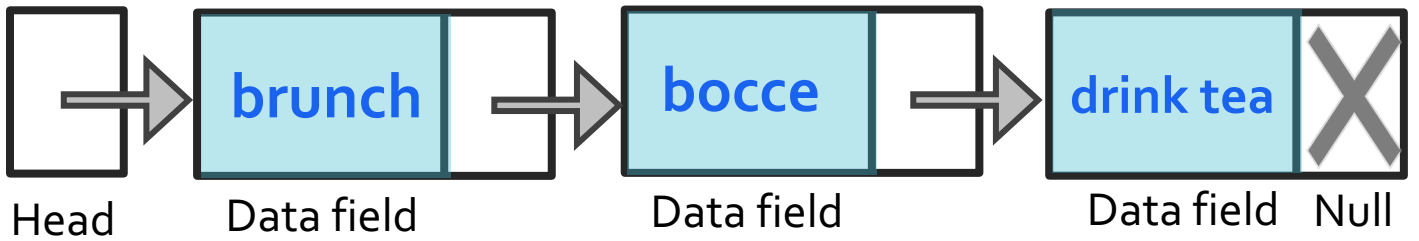
❖ Example: Lets print our **todo linked-list!**

```
head = todo_list.head
print('node 1:', head.data)
print('node 2:', head.next.data)
print('node 3:', head.next.next.data)
```

Output

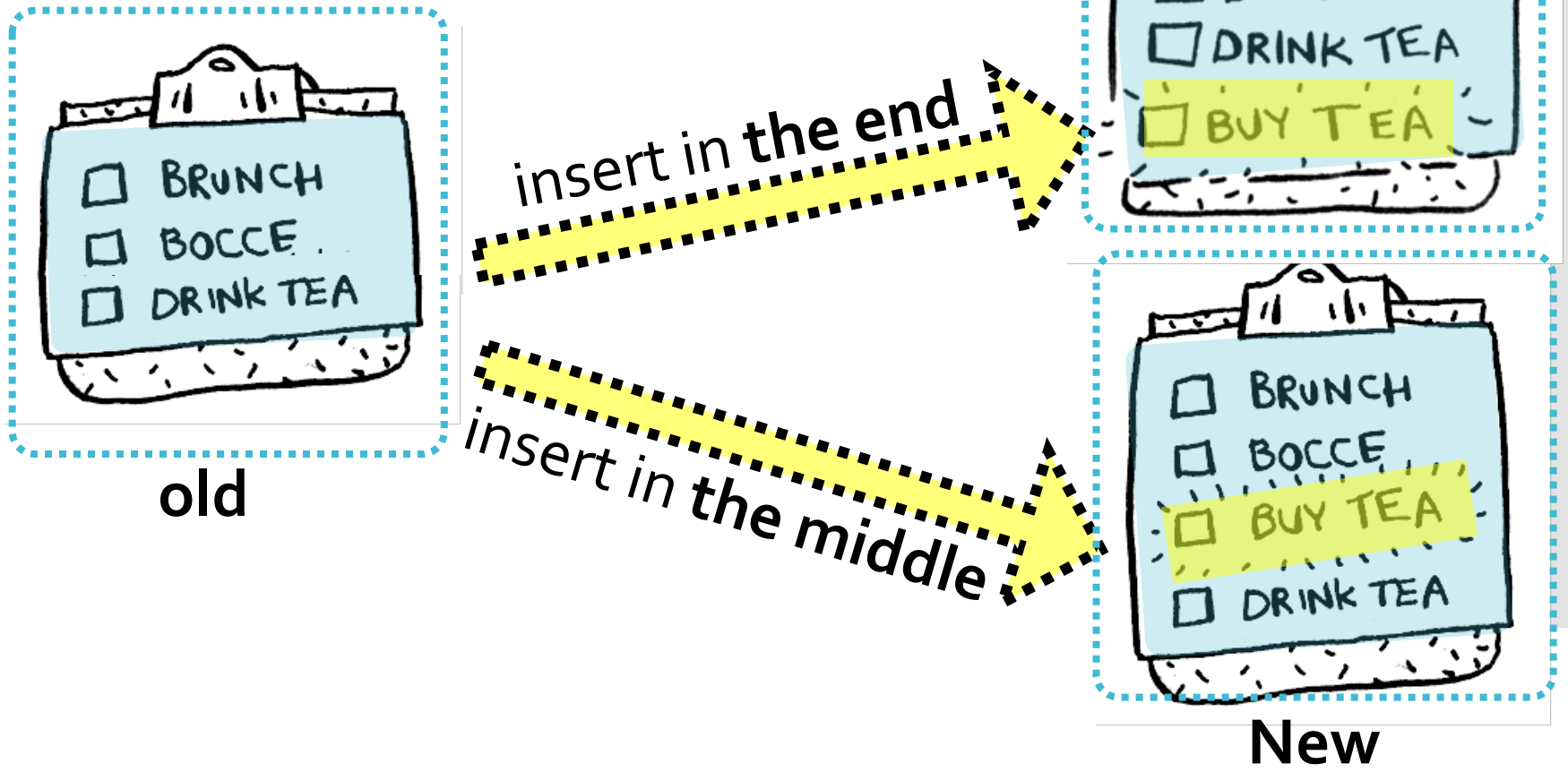
```
node 1: brunch
node 2: bocce
node 3: drink tea
```

Singly Linked-list



Quiz

- Write a piece of code that can **insert** a **new node**:
  a) in the **end** of linked-list
  b) in the **middle** of linked-list



old

New

Answer

❖ **Insert** a new node **in the end** of linked-list.

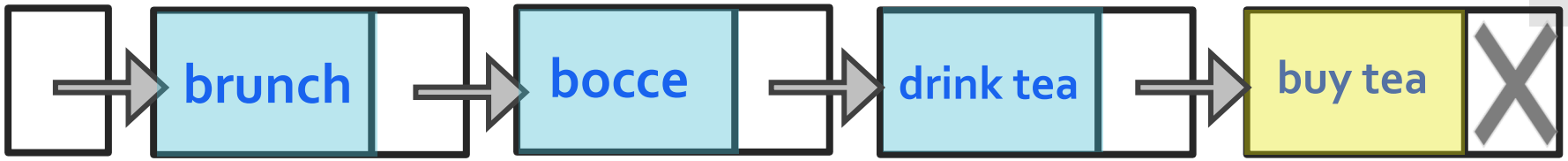
```
new_node = Node("buy tea")
head = todo_list.head
head.next.next.next = new_node

print('node 1:', head.data)
print('node 2:', head.next.data)
print('node 3:', head.next.next.data)
print('node 4:', head.next.next.next.data)
```

new node

Output:

```
node 1: brunch
node 2: bocce
node 3: drink tea
node 4: buy tea
```

new node



Answer

❖ **Insert** a new node **in the middle** of linked-list.

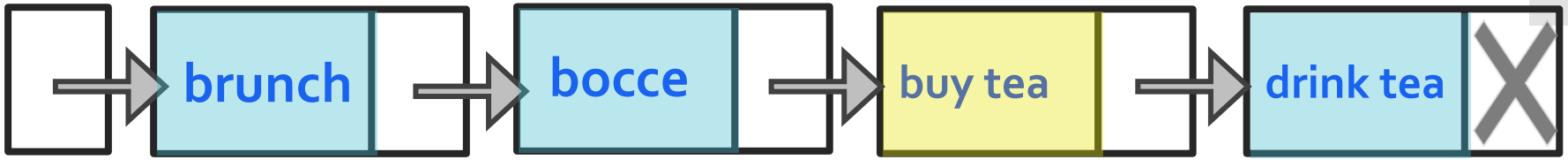
```
new_node = Node("buy tea")
head = todo_list.head
new_node.next = head.next.next
head.next.next = new_node

print('node 1:', head.data)
print('node 2:', head.next.data)
print('node 3:', head.next.next.data)
print('node 4:', head.next.next.next.data)
```

new node

Output:

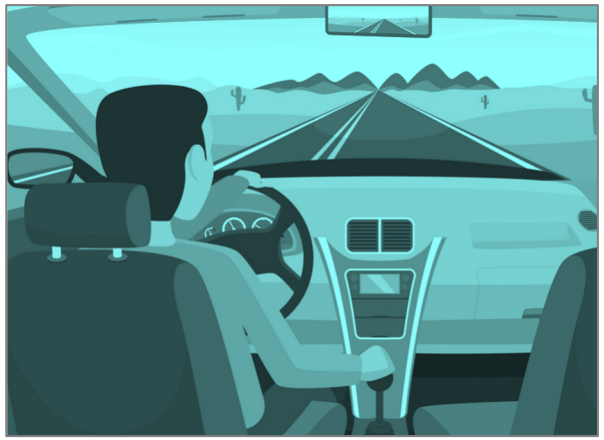
```
node 1: brunch
node 2: bocce
node 3: buy tea
node 4: drink tea
```

new node

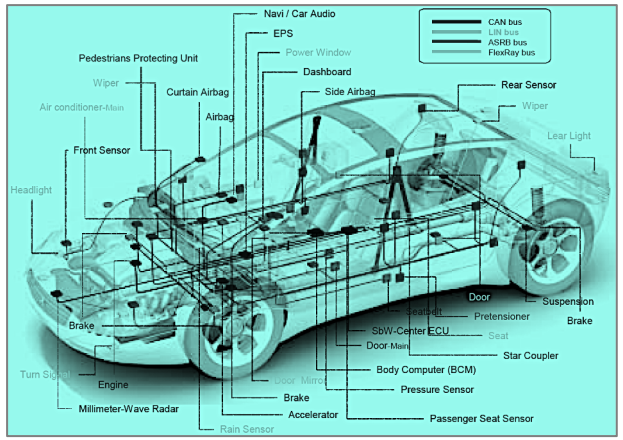


Information hiding

- **Implementation details** are hided from user.
- This is known as **information hiding.**
- **Example:**
  - we know how to drive a car **(interface)**
  - we don't know how it is made **(implementation).**



interface



implemendation

seperate

Abstract Data Type (ADT)

- In computer programming separate **interface** and **implementation** is separated.

- **Abstract Data Type (ADT):**
  - are well-defined data type that specifies a set of:
    a) **data values**
    b) **operations** (performed on data values).

  - are defined **independent of implementation**.
  - are concerned **only** with **data representation.**

Abstract Data Type (ADT)

- **Abstract Data Types** can be:
  - **Simple:** such as date.
  - **Complex:** such as Python list.

- **Data Structure**:
  - representation of how data is organized and manipulated in **Abstract Data Types.**
  - Examples: array, linked-list, stack, and queue.

Abstract Data Type (ADT)

- **Abstract Data Type** is accessed through:
  - the interface.
  - the defined operations.
- Allows us to **focus on functionality**, instead of implementation.
- Programmer can implement an Abstract Data Type in **different ways**.

Linked-list

- **`isEmpty()`**
  - tests to see whether the list is empty.
- **`add(item)`**
  - adds a new item to the list.
- **`remove(item)`**
  - removes the item from the list.
- **`search(item)`**
  - searches for the item in the list.
- **`size()`**
  - returns the number of items in the list.
- **`pop()`**
  - removes and returns the last item in the list.

Linked-list

- **`isEmpty()`**
  - tests to see whether the list is empty.
- **`add(item)`**
  - adds a new item to the list.

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

    def is_empty(self):
        return self.head == None

    def add(self, data):
        temp = Node(data)
        temp.next = self.head
        self.head = temp
```

Linked-list

- **`search(item)`**
  - searches for the item in the list.

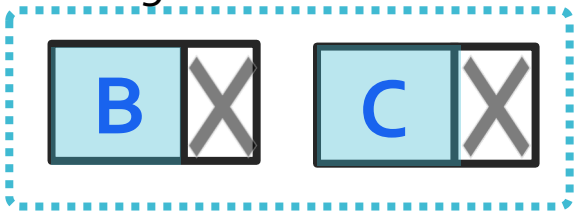
```
class LinkedList:
    def __init__(self):
        self.head = None
    def search(self, data):
        current = self.head
        found = False
        while current and not found:
            if current.data == data:
                found = True
            else:
                current = current.next
        return found
```

Linked-list

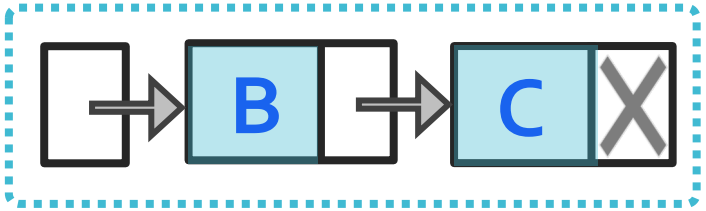
❖ Lets test it!

```
node1 = Node("B")
node2 = Node("C")

my_list = LinkedList()
my_list.head = node1
node1.next = node2

my_list.add('A')
my_list.print_list()
```

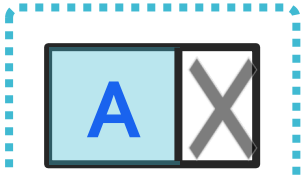
*creating the nodes **'B' and 'C'***



*creating the **linked-list***



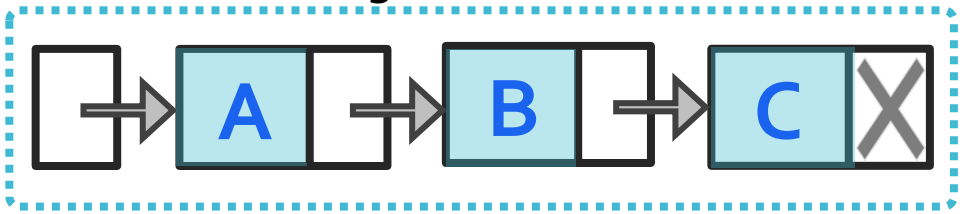
*creating the node **'A'***



[Output]

```
A
B
C
```

***adding** the node **'A'** to the linked-list*



Linked-list

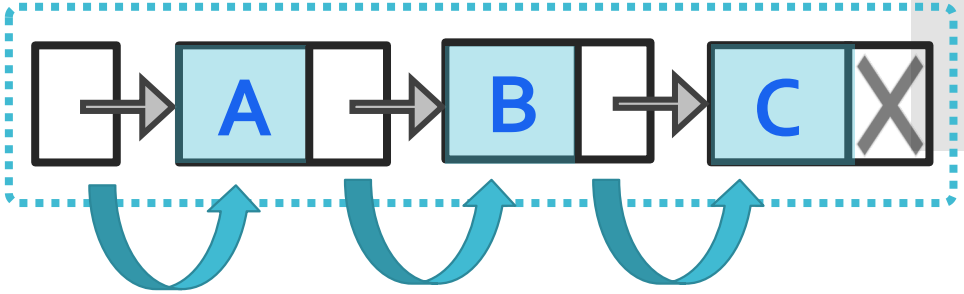
- Lets test it!

```
print('my_list is empty?')
print('Answer: ', my_list.is_empty())

print('my_list contains C?')
print('Answer: ', my_list.search("C"))
```

[Output]

```
my_list is empty?
Answer:  False
my_list contains C?
Answer:  True
```

Searching in the linked-list



Quiz

- Writa **a method** in Linkedin class that **prints the size of** Linked-list.

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

add your
method here

Answer

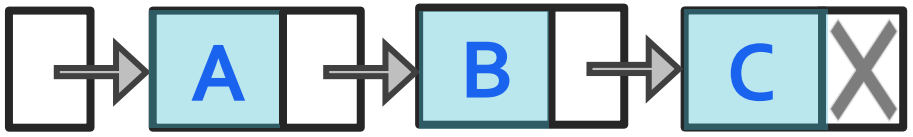
- Writa a method in Linkedin class that prints the size of Linked-list.

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

    def size(self):
        current = self.head
        count = 0
        while current:
            count = count + 1
            current = current.next
        return count
```

Answer

- Write a method in Linkedin class that prints the **size of Linked-list**.

```
node1 = Node("A")
node2 = Node("B")
node3 = Node("C")

my_list = LinkedList()
my_list.head = node1
node1.next = node2
node2.next = node3

print("linked-list size:", my_list.size())
```



method call

[Output]

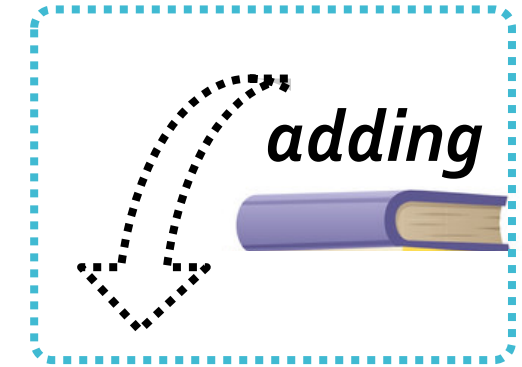
```
size: 3
```

Stack

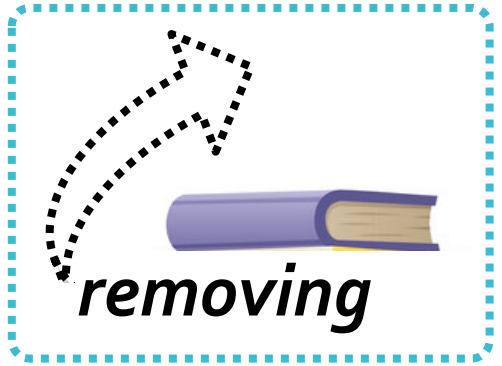
- What is Stack Structure?

Stack

- **Stacks** of books.
  - how do you **add** another book?
  - how do you **remove** a book?
  - from **top** or **button**?



?



Stack

- **Ordered collection** of items where **adding** a new item or **removing** an existing item takes place at the **top.**
- **Opposite** of the top is known as the **base**.

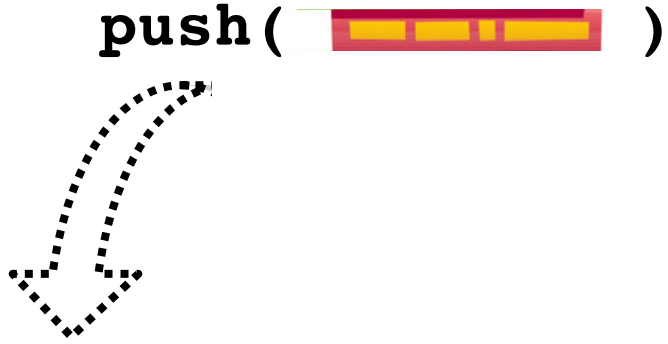
Stack

- **Last In First Out (LIFO)**:
  - the most **recently added** item is **removed first**.

- **Newer** items are near the **top.**
- **Older** items are near the **base**.

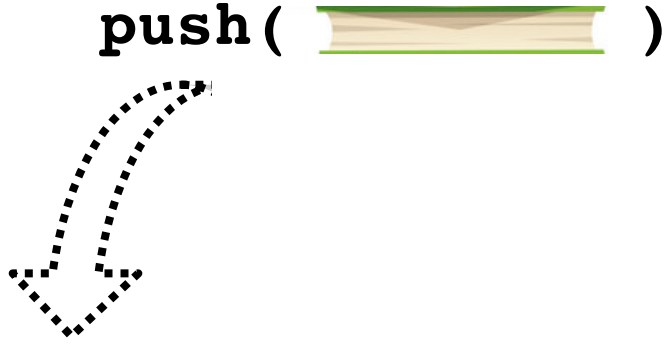
Stack

- adding a new item (**`push`**)



Stack

- adding a new item (**`push`**)



Stack

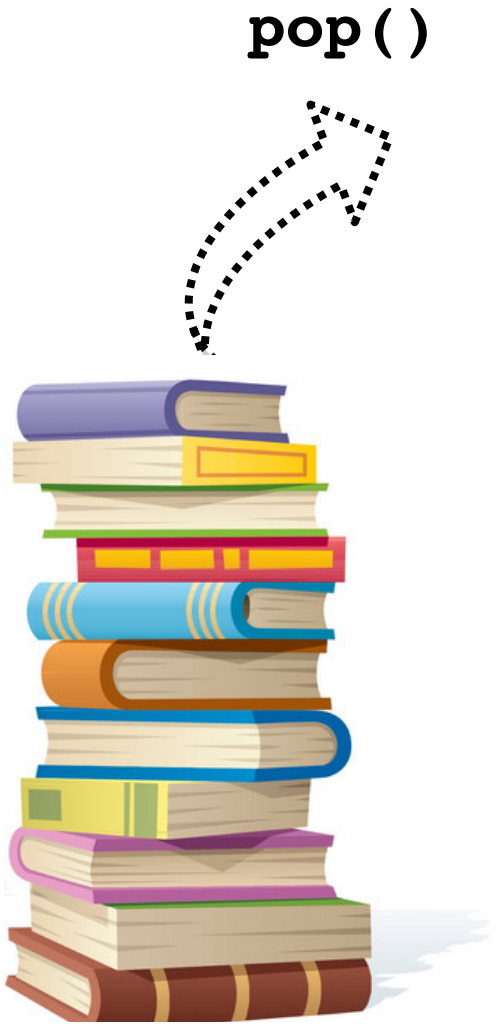
- adding a new item (**`push`**)

Stack

- adding a new item (**`push`**)

Stack

- removing an item (**pop**)



Stack

- removing an item (**pop**)

Stack

- removing an item (**pop**)

Stack

- removing an item (**pop**)

pop ( )

Stack

- removing an item (**pop**)

pop ( )

Stack

- **`push(item):`**
  - Adds the given item to the top of the stack.
- **`pop():`**
  - Removes and returns the top item of the stack.
- **`isEmpty():`**
  - Returns a boolean indicating if the stack is empty.
- **`size():`**
  - Returns the number of items in the stack.
- **`peek():`**
  - Returns a reference to item on top without removing it

Stack

Stack ADT implementation with Python list

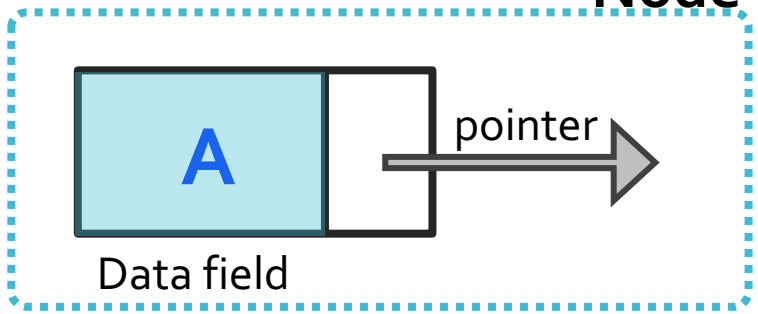
```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self,item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

Stack

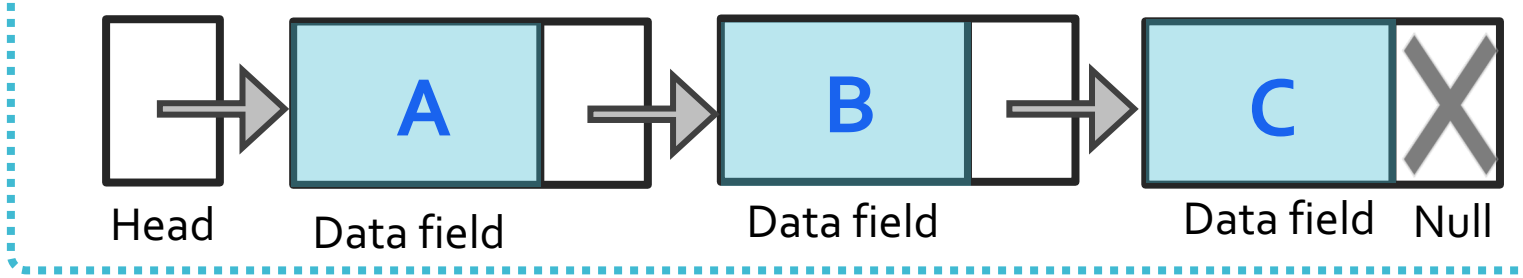
❖ **Stack** ADT implementation **with linked-list**

```
class Node:
    def __init__(self, data, link):
        self.data = data
        self.next = link
```

Node



Singly Linked-list



Stack

Stack ADT implementation **with linked-list**

```
class Stack:
    def __init__(self):
        self.top = None
        self.size = 0

    def pop(self):
        node = self.top
        self.top = self.top.next
        self.size -= 1
        return node.data

    def push(self, data):
        self.top = Node(data, self.top)
        self.size += 1

    def size(self):
        return self.size
```

Stack

❖ **Stack** ADT implementation **with linked-list**

```
my_stack = Stack()

my_stack.push('book in the bottom')
my_stack.push('Book in the middle')
my_stack.push('Book on the top')

print(my_stack.pop(), 'is removed!')
print(my_stack.pop(), 'is removed!')
print(my_stack.pop(), 'is removed!')
```

[Output]

```
Book on the top is removed!
Book in the middle is removed!
book in the bottom is removed!
```

Next Lesson

- **Search and sort algorithms** in Python (part1)