Data Structures and Algorithms

Lesson 4. Lists and Pointer Structures

Alexandra Ciobotaru

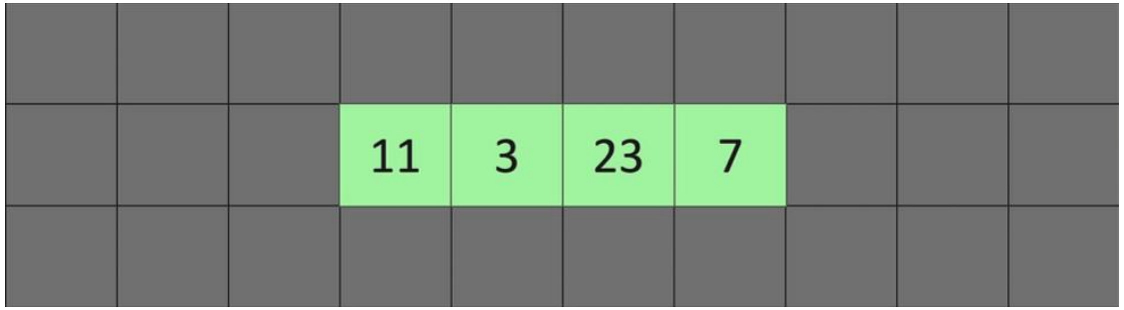
10 mar 2022

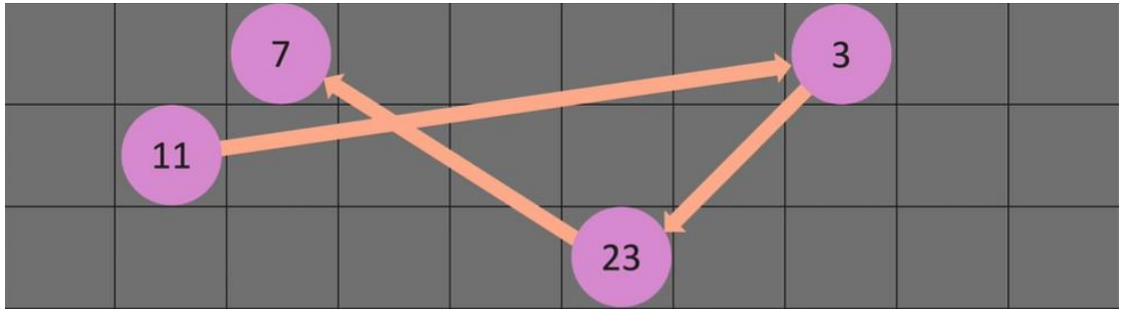
Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- Lesson 5. Stacks and Queues.
- Lesson 6. Trees.
- Lesson 7. Hashing and Symbol Tables.
- Lesson 8. Graphs and Other Algorithms.
- Lesson 9. Searching.
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

Linked Lists

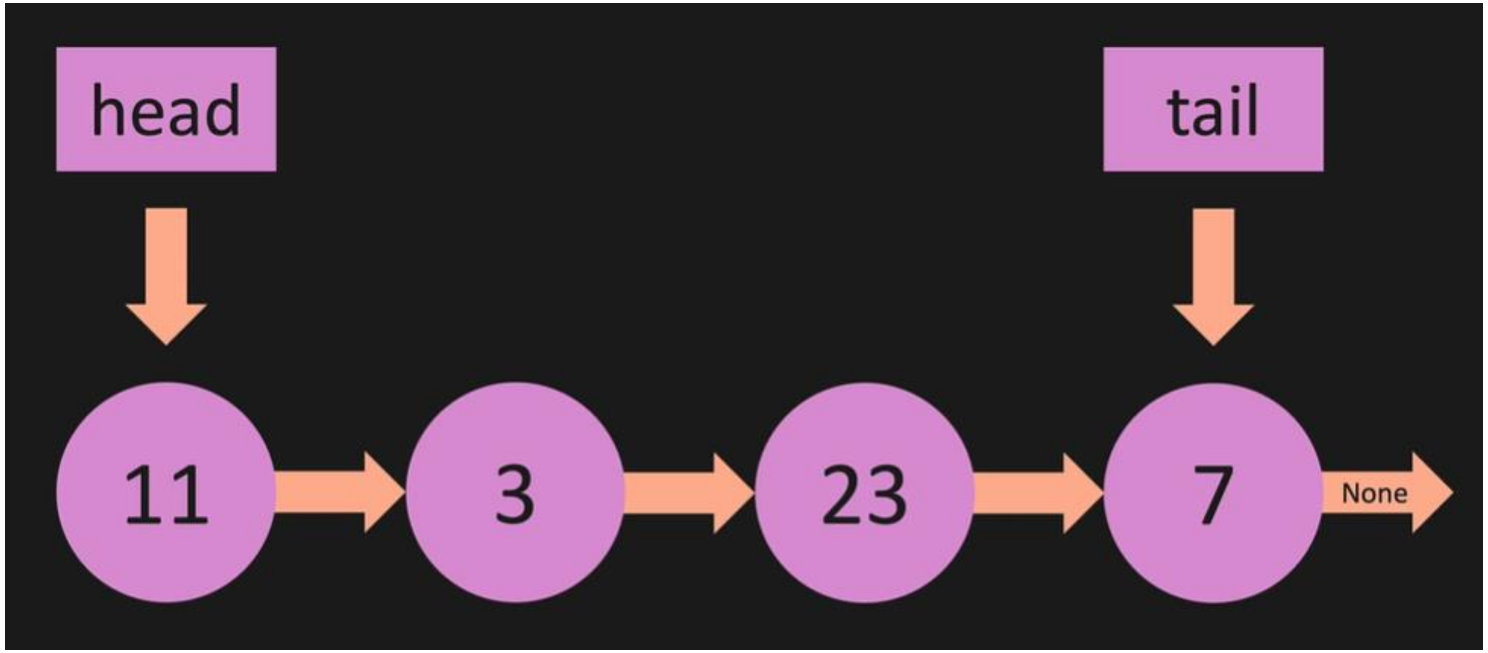
- First, let’s look at a list: x = [11,3,23,7]
- A linked list does not have indexes
- A list is a contiguous place in memory (elements in list are stored next to each other in memory)
- In a linked list, the elements (called nodes) in it are going to spread all over the place in memory





Linked Lists

- In a linked list, each node is pointing to the next one, and the last one is going to point to None
- We also have a variable called Head and a variable called Tail



Big O

- 1. Append a new node at the end of the list

- the number of operations to add one node at the end of the list is the same, regardless of the number of elements in the list: O(1)

- 2. Remove the item from the end of the list

- in order to have Tail point to the last node, we have to have that pointer there -> we need to iterate through the list until we reach the last pointer: O(n)

- 3. Adding an item to the front list

- we need to have the item to be inserted point to the first element in list, and the Head points there -> we set equal the pointer for the element to be inserted with the Head pointer: O(1)

- 4. Removing the item from the front list

- Head = Head(next) and we remove the element: O(1)

Big O

- 5. Insert an element somewhere in the list

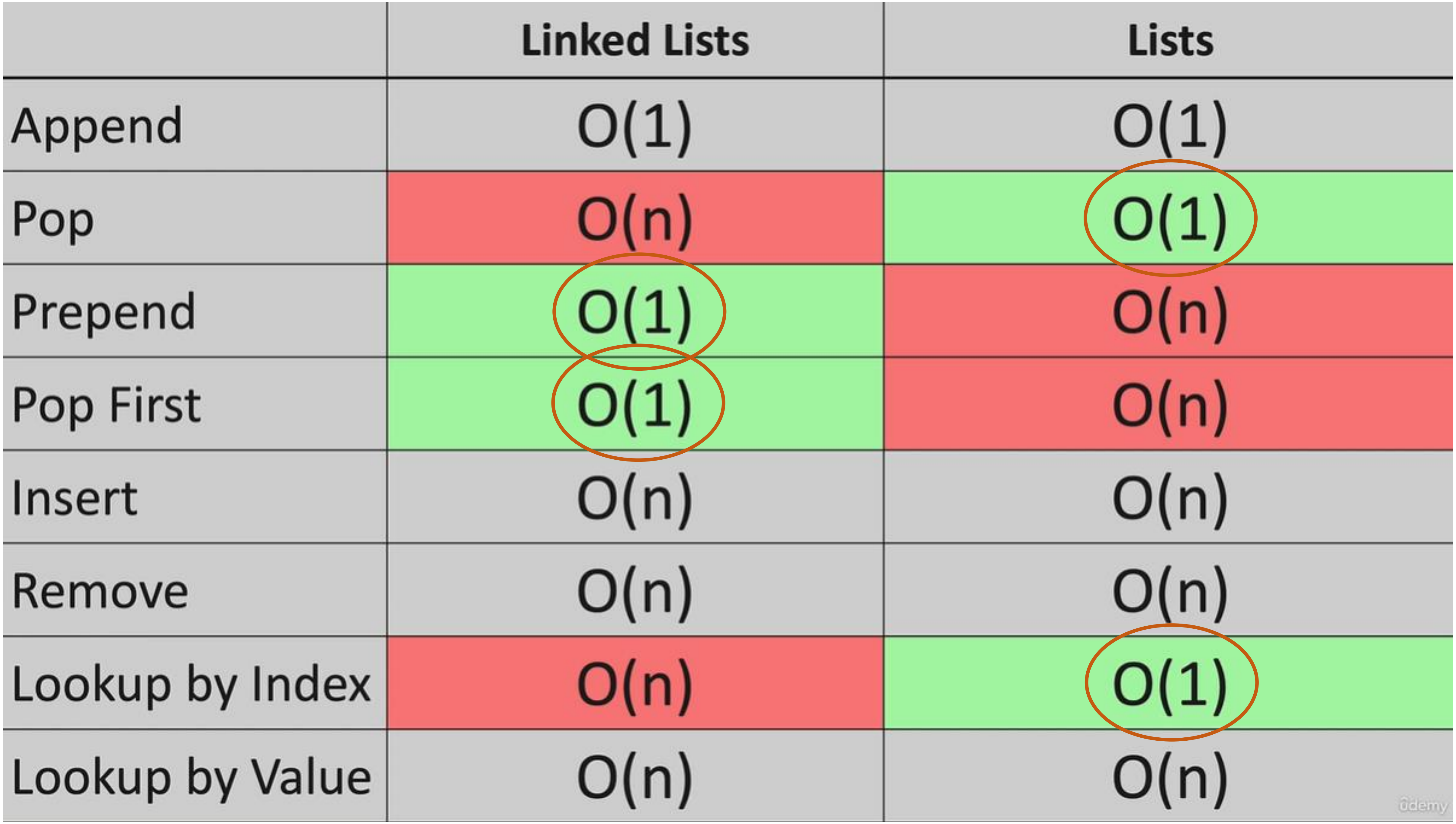
- we iterate through the list until we reach the desired pointer: O(n)

- 6. Removing the element inserted at that specific location

- we iterate through the list until we reach the desired pointer, and remove the element: O(n)

- 7. Look-up an element in list

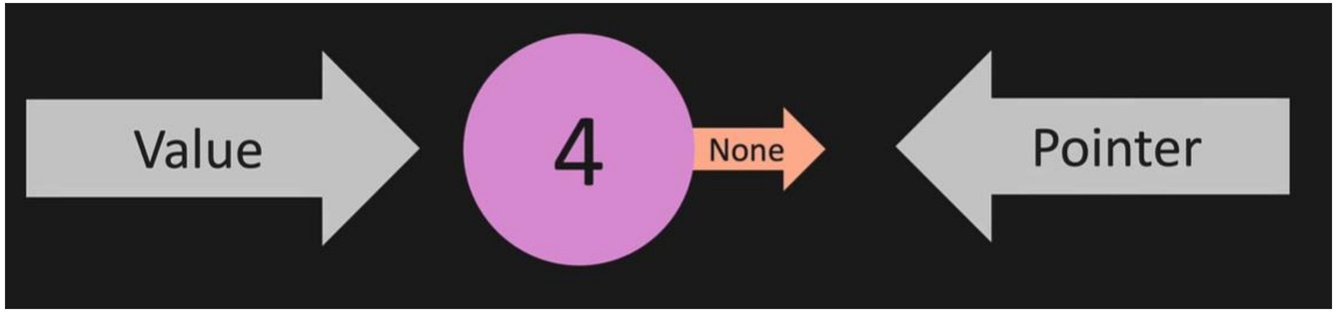
- we start from the head and ask if each element is the one we are looking for, until we find it: O(n)



	Linked Lists	Lists
Append	O(1)	O(1)
Pop	O(n)	O(1)
Prepend	O(1)	O(n)
Pop First	O(1)	O(n)
Insert	O(n)	O(n)
Remove	O(n)	O(n)
Lookup by Index	O(n)	O(1)
Lookup by Value	O(n)	O(n)

What is a node in the list?

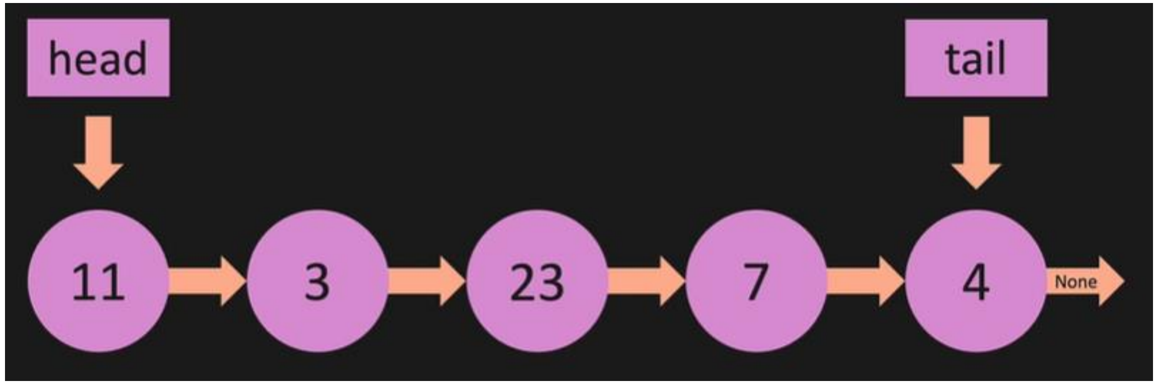
- The value and the pointer to the node, together make up the node

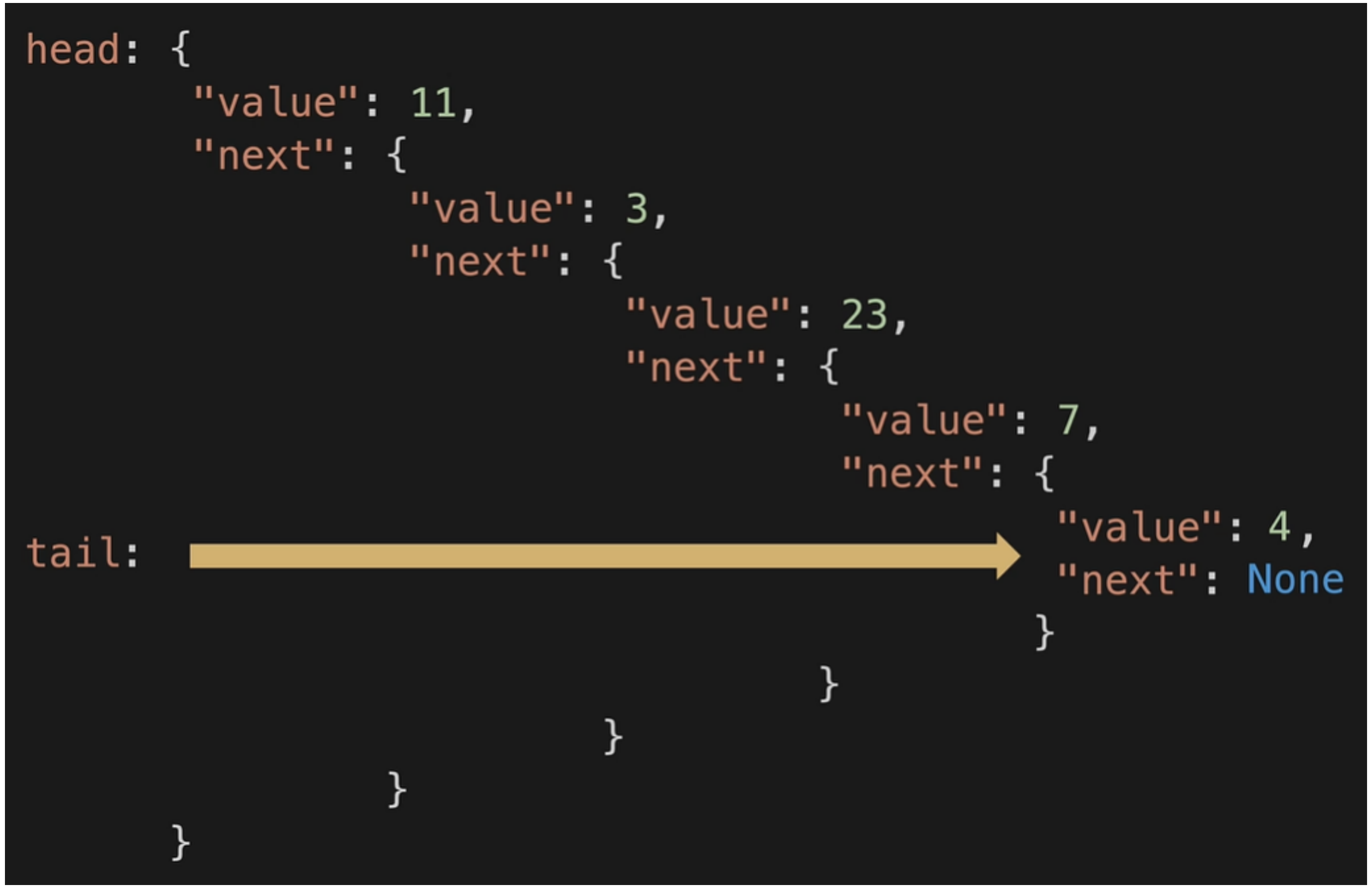


- A node is very similar to a dictionary (cannot be accessed like a dictionary, still):

```
{
    "value": 4,
    "next": None
}
```

Dictionary view



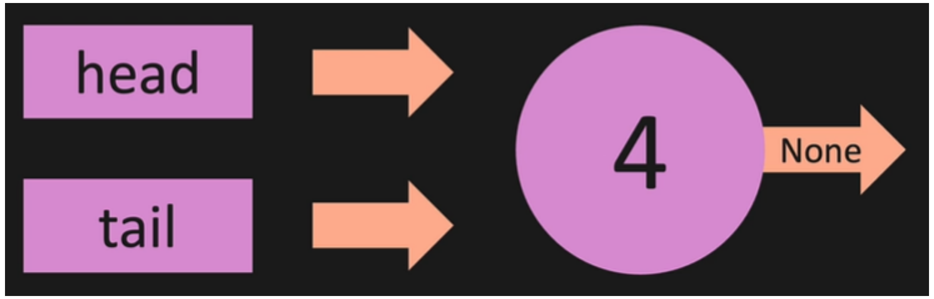


```
head: {
       "value": 11,
       "next": {
                "value": 3,
                "next": {
                         "value": 23,
                         "next": {
                                  "value": 7,
                                  "next": {
                                           "value": 4,
tail:
                                           "next": None
                                          }
                                 }
                        }
               }
      }
```

```
head = {
    "value":11,
    "next":{
        "value":3,
        "next":{
            "value":23,
            "next":{
                "value":7,
                "next": None
            }
        }
    }
}
print(head['next']['next']['value'])
# With a linked list:
#print(my_linked_list.head.next.next.value)
```

Linked List Constructor

```
class Node:
    # to create nodes
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self, value):
        #create new Node
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1

my_linked_list = LinkedList(4)
print(my_linked_list.head.value)
```



What do we want from LinkedList class?

```
def append(self, value):
        #create new Node
        #add Node to end
def prepend(self, value):
        #create new Node
        #add Node to beginning
def insert(self, index, value):
        #create new Node
        #insert Node where wanted
```

First, let’s create *append* and *print_list* functions

```
class LinkedList:
…
    def print_list(self):
          temp = self.head
          # we stop running the loop when temp is None
          while temp is not None:
              print(temp.value)
              temp = temp.next
      def append(self, value):
          new_node = Node(value)
          if self.length == 0:
              self.head = new_node
              self.tail = new_node
          else:
              self.tail.next = new_node
              self.tail = new_node
          self.length += 1
          return True
```

Value to be inserted

Create new node using the Node class

The situation when there are no items in the linked list

The situation when there are already items in the linked list

Optional

Now let’s test these methods

```
my_linked_list = LinkedList(4)
my_linked_list.append(3)
my_linked_list.append(23)
my_linked_list.append(15)
my_linked_list.print_list()
```

Now let’s create the *prepend* method

```
class LinkedList:
…
    def prepend(self, value):
        new_node = Node(value)
        if self.length == 0:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head = new_node
        self.length += 1
        return True
my_linked_list = LinkedList(4)
my_linked_list.append(3)
my_linked_list.prepend(1)
my_linked_list.print_list()
```

Value to be inserted

Create new node using the Node class

The situation when there are no items in the linked list

The situation when there are already some items in the linked list

We increment the length of the LL

Optional

Get

- The get methods gets passed in an index, and returns the node at that index:

```
 def get(self, index):
        if index < 0 or index >= self.length:
            return None
        temp = self.head
        for _ in range(index):
            temp = temp.next
        return temp.value
my_linked_list = LinkedList(0)
my_linked_list.append(1)
my_linked_list.append(2)
my_linked_list.append(3)
print(my_linked_list.get(2))
```

We test if the index is valid, because we can’t get a Node at any of those indexes

Remember head is an object of type Node

We don’t use the iterator in the for loop, that’s why we call it _

Let’s create a method for *inserting*

```
 def insert(self, index, value):
        if index < 0 or index > self.length:
            return False
        if index == 0:
            return self.prepend(value)
        if index == self.length:
            return self.append(value)
        new_node = Node(value)
        temp = self.get(index - 1)
        new_node.next = temp.next
        temp.next = new_node
        self.length += 1
        return True
my_linked_list = LinkedList(0)
my_linked_list.append(2)
my_linked_list.insert(1,1)
my_linked_list.print_list()
```

Insert a new Node that has a particular value, at a particular index

We can’t insert at an index that is **out of range**

If the index is zero, we already wrote code for this case

If the index is at the last element in list, we already wrote code for this case

We create the Node we want to insert

We need a variable that points to the node right before the index where we want the insertion, because we need to tell the pointer there to point to the node to be inserted

Search in LL

```
 def iter(self):
        """ Iterate through the list. """
        current = self.tail
        while current:
            val = current.value
            current = current.next
            yield val

    def search(self, value):
        """ Search through the list. Return True if data is found, otherwise return False. """
        for node in self.iter():
            if value == node:
                return True
        return False


my_linked_list = LinkedList(0)
my_linked_list.append(1)
my_linked_list.append(2)
print(my_linked_list.search(5))
```

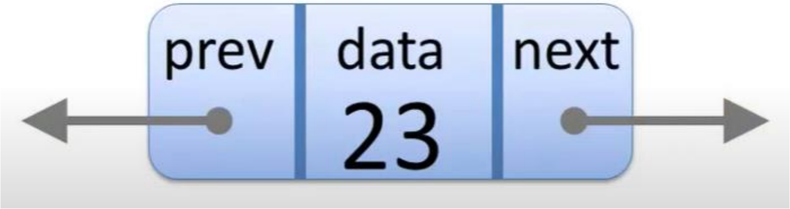
yield in Python can be used like the return statement in a function.
When done so, the function instead of returning the output,
it returns a generator that can be iterated upon.
We use yield when we want to iterate over a sequence, but don't
want to store the entire sequence in memory.

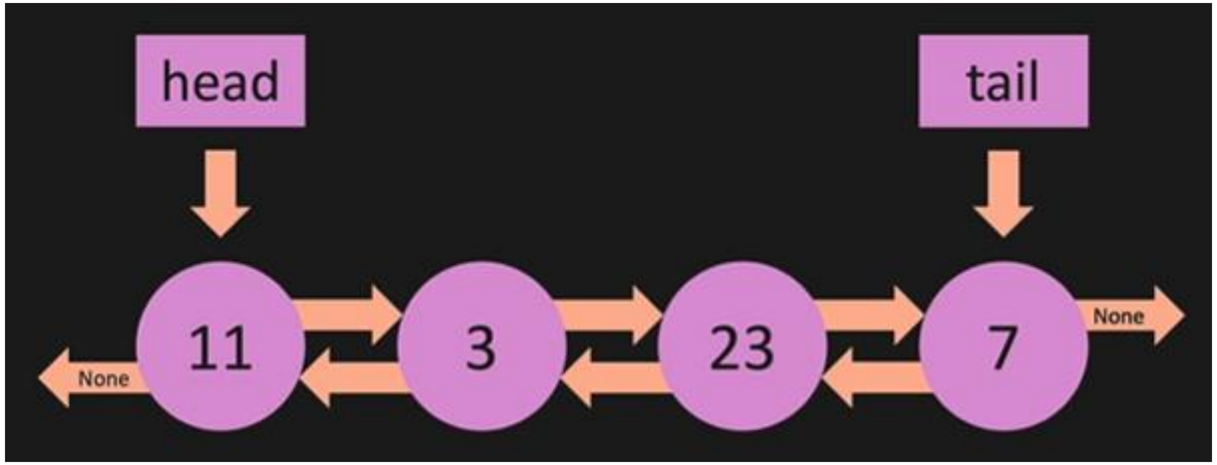
Some Preliminary Conclusions

- We had fun and created some of the methods you knew that lists have
- And we also created a new one! -> **Prepend**
  - lists that have the prepend method are called *deques* (https://www.geeksforgeeks.org/deque-in-python/)
- Regular lists are also called Singly Linked Lists

Doubly Linked Lists

- Or Bidirectional Linked Lists
- Now every Node has three parts:
  - Pointer to the previous Node
  - Node value
  - Pointer to the next Node





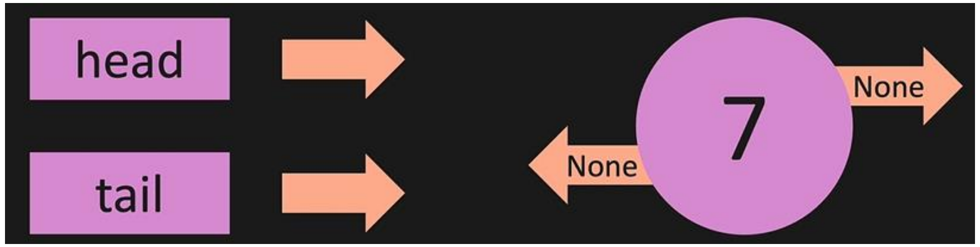
Advantages of Doubly Linked Lists

- You can iterate the list in either direction
- You can delete a Node without iterating through the list (if you have a pointer to that node)
  - you simply need to update the previous node and the next node
  - => so if Big O was O(n) in singly links, doubly links are O(1) for this operation

Doubly Linked List Constructor

```
class Node:
    # to create nodes
    def __init__(self, value):
        self.value = value
        self.next = None
        self.prev = None
class DoublyLinkedList:
    def __init__(self, value):
        #create new Node
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1
my_doubly_linked_list = DoublyLinkedList(7)
my_doubly_linked_list.print_list()
```

We define a property for the previous node as well



Append

```
class DoublyLinkedList:
…
 def append(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
        self.length += 1
        return True
```

Value to be inserted

Create new node using the Node class

The situation when there are no items in the DLL

The situation when there are already items in the DLL

Pop

- Popping the last element in the DLL

```
 def pop(self):
        if self.length == 0:
            return None
        temp = self.tail
        if self.length == 1:
            self.head = None
            self.tail = None
        else:
            self.tail = self.tail.prev
            self.tail.next = None
            temp.prev = None
        self.length -= 1
        return temp
my_doubly_linked_list = DoublyLinkedList(1)
my_doubly_linked_list.append(2)
print(my_doubly_linked_list.pop())
print(my_doubly_linked_list.pop())
print(my_doubly_linked_list.pop())
```

Situation when we don’t have any values in DLL

We create temp variable

Situation when we have only one value in DLL

Add return temp.value to see the Node object

Search in DLL

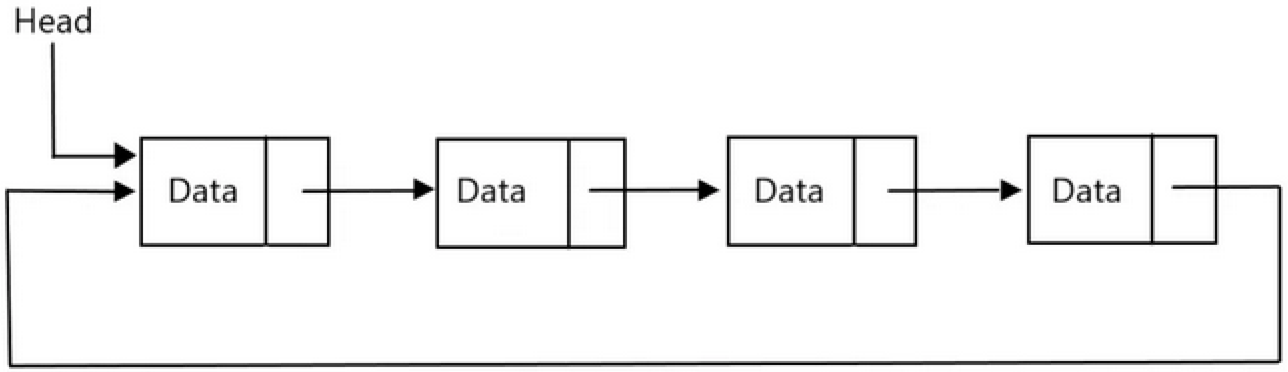
```
 def iter(self):
        """ Iterate through the list. """
        current = self.head #note subtle change
        while current:
            val = current.value
            current = current.next
            yield val

    def search(self, data):
        """Search through the list. Return True if data is found, otherwise False."""
        for node in self.iter():
            if data == node:
                return True
        return False
my_doubly_linked_list = DoublyLinkedList(1)
my_doubly_linked_list.append(2)
print(my_doubly_linked_list.search(1))
```

yield in Python can be used like the return statement in a function. When done so, the function instead of returning the output, it returns a generator that can be iterated upon.

We use yield when we want to iterate over a sequence, but don’t want to store the entire sequence in memory.

Circular Lists

- A circular linked list is a special case of a linked list.
- In a circular linked list, the endpoints are connected to each other. It means that the last node in the list points back to the first node.
- In other words, we can say that in circular linked lists all the nodes point to the next node (and the previous node in the case of a doubly linked list) and there is no end node, thus no node will point to Null.
- Circular lists can be based on both singly and doubly linked lists.



Wrapping it all up

- In this lesson we have learned about ***linked lists***
- We saw what are the core-concepts of LL – nodes and pointers to other nodes
- We implemented some of the operations that occur in these types of lists
- All other methods (append, prepend, insert etc.) for DLL are similar as for LL
- Homework: check the other methods in attachments (optional)

Thank you!