Programação e Algoritmia
--x--
Programming and Algorithms

1 – Object-Oriented Programming

Data Structures

Problems

- How to store/handle:
  - The information in a dictionary
  - A waiting list in a Hospital
  - Phone numbers to make a quick search
  - Information to rapidly find the best path between 2 cities

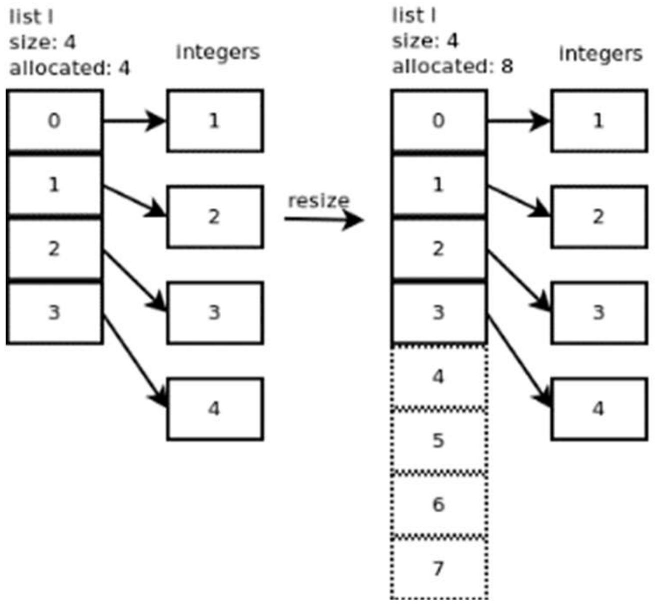
Python Data Structures and Collections

- Tuples
  - They are sequential data structures, that are similar to lists, but they're immutable.
- Dictionary
  - Dictionaries are python-specific data structures that are similar to hashtables, and are used for quick access of values.
- Sets
  - A collection of unordered distinct elements, that have quick access time.
- Collections
  - Collections are a group of optimized implementations for data structures like dictionaries, maps, tuples, queues.

Python List

- Operations:
  - **append(x)**
  - **extend(x)**
  - **sort(…)**
  - **reverse()**
  - **index()**
  - **insert()**
  - **count()**
  - **remove()**
  - **pop**

- Implementation
  - Using dynamic array
  - That changes in size (by blocks) when needed



Python List – more information on methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

code

Abstract Data Types
&
Data Structures

New data types

- Most programs/algorithms need to organize the data they work with in certain ways.
- So, all modern programming languages allow the programmer to define new data types
  - not predefined in the language.
- Often each kind of data used by an algorithm can be stored in more than one way but is used in essentially the same way.
- A good programmer pays considerable attention to the operations that the algorithm needs to perform on its data, and defines data types not by the specific way they store their information but by what operations are allowed to be performed on the information .
- He/she will design **abstract data types**.

Abstract data type

- An abstract data type is a purely mathematical type
  - defined independently of its concrete realization as code.

- Abstract data types enable the programmer to reason about algorithms and their cost separately from the task of implementing them.

Abstract data type

- An abstract data type is a data type that exposes to the rest of the program the ways the data it stores can be queried or modified (the interface),
  - without exposing to the rest of the program its internal workings (the implementation)

    be it how the data is actually stored or how the queries/modifications are actually performed.

- This will make the program more modular
  - because as long as the interface of an abstract data type is not changed, changes in its implementation will not affect how the rest of the program is coded.

Data Structures

- Many kinds of data consist of multiple parts, organized (structured) in some way

- A data structure is simply some way of organizing a value that consists of multiple parts
  - Hence, an array is a data structure, but an integer is not

- A data structure describes the **concrete implementation of a data type**
  - which data is stored, in what order, and how it is interpreted

Data Structures and Abstract Data Types

- If we talk about the possible values of, say, complex numbers, and the operations we can perform with them, we are talking about them as an **ADT**

- If we talk about the way the parts ("real" and "imaginary") of a complex number are stored in memory, we am talking about a **data structure**

- An ADT may be implemented in several different ways
  - A complex number might be stored as two separate doubles, or as an array of two doubles, or a tuple, …

Implementing the Data Structures from Scratch

- Implementing the data structures from scratch helps a lot.
- It helps you to understand the data structure in detail and helps in debugging, in case you run into any errors.

- Python offers in-build implementation for many data structures
  - but it is better to learn how to implement it at least once.

Different types of data structures

Linear Data Structures

- Linear data structures store elements in a **sequential manner**. These are the most fundamental and widely used data structures.

- Array/List
  - An array or list is a linear data structure where elements are stored in a sequential manner, numbered from 0 to n-1, where n is the size of the array. The array elements can be accessed using their index.
- **Stacks**
  - A stack is a linear data structure that stores data in a Last In, First Out (LIFO) manner.
- **Queues**
  - A data structure that stores data in a First In, First Out (FIFO) manner.
- **Linked Lists**
  - A linear data structure that stores elements sequentially but cannot be accessed directly using an index. It consists of links to the next item, along with the data.

In PA we will address structures highlighted

Non-Linear Data Structures

- Non-Linear data structures are more complex data structures, that are **not sequential in manner**.

- **Trees**
  - Trees store data in a tree-like manner. They consist of a root, which contains the data and links to children nodes. The children nodes may in turn contain more children. They are the building blocks of many other Data Structures.
- Heaps
  - A heap is a tree, which is used as a priority queue. There are max-heaps, and min-heaps that contain the maximum value and the minimum value at the root node, respectively.
- Hash Tables
  - Hash Tables are data structures that contain key-value pairs. They use the concept of hashing to determine the keys in the tables. Usually used for quick access of values.
- **Graphs**
  - Graphs are complex data structures that implement a collection of nodes, connected together by vertices.

In PA we will address structures highlighted

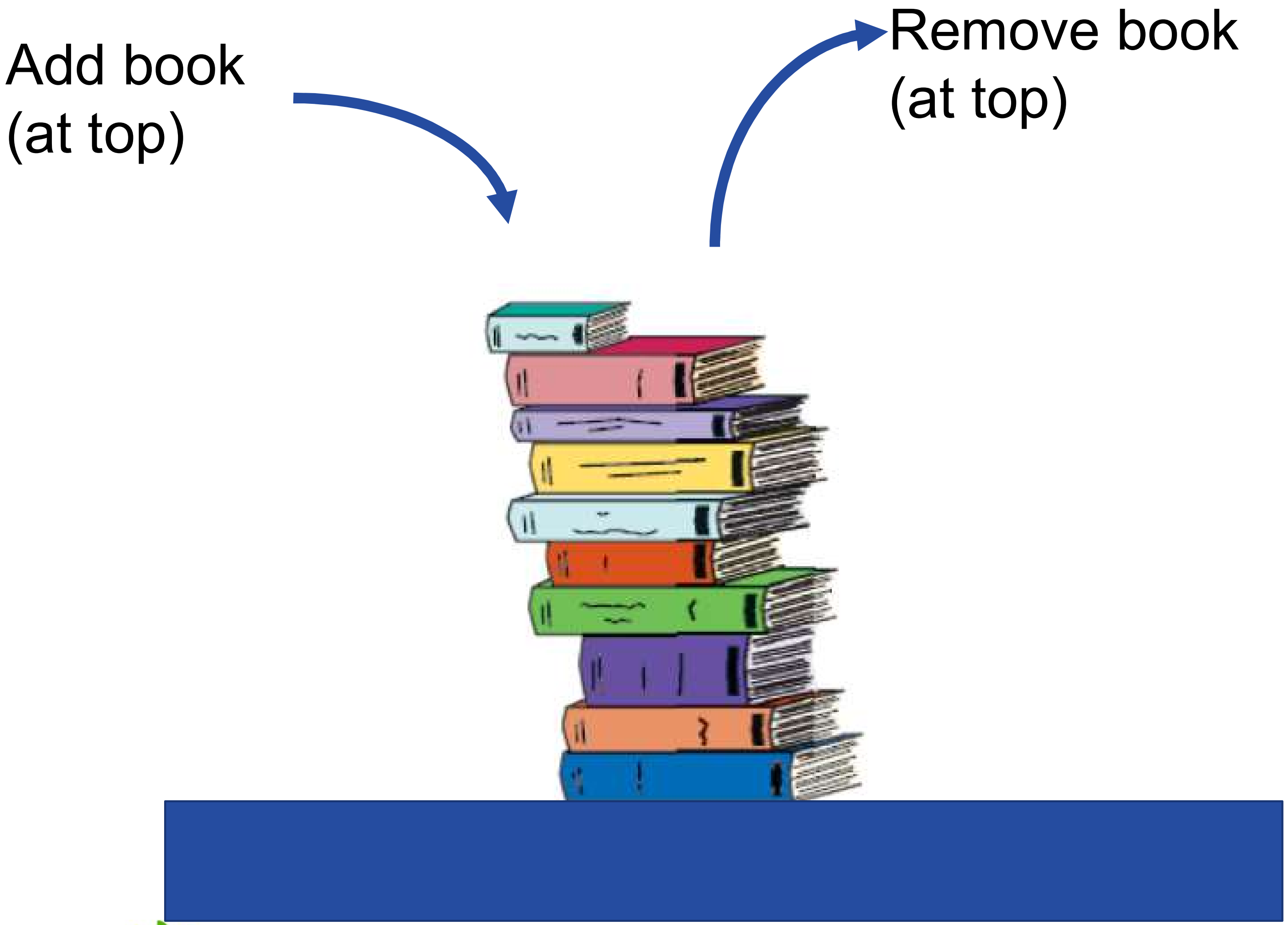
Stack

PT: Pilha

Stack

- Stack of Books

How it works ?



What is a Stack?

- A stack is a data structure of items such that items can be inserted and removed only at one end

- A stack is a LIFO (Last-In/First-Out) data structure

- A stack is sometimes also called a pushdown store.

What can we do with a Stack?

- Place an item on the stack (**push**)
- Look at the item on top of the stack, but do not remove it (**peek** or **top**)
- Remove the item on top of the stack (**pop**)

Stack ADT

- Operations:
  - creation of the stack
  - destruction of the stack
  - add a new element to the top of the stack, called push
  - remove the element at the top of the stack, called pop
  - take a look at the top element of the stack, called top (or peek)
  - determine the current size of the stack

  - check if stack is empty (⇔ size == 0)

Typical Stack Interface (in Python)

```
class Stack:
    def __init__(self):   # to create a Stack

    def push(self, item):
    def pop(self):
    def top(self):

    def size():

    def isEmpty():
```

Problem

- What happens if we try to pop an item off the stack when the stack is empty?
  - This is called a stack underflow.
- The pop method needs some way of telling us that this has happened.
  - We can use assertions or exceptions for example

- Better to first check if it is empty

Implementations

- Several implementations are possible
  - An array

    in this case the stack has a maximum size, specified when the stack is created

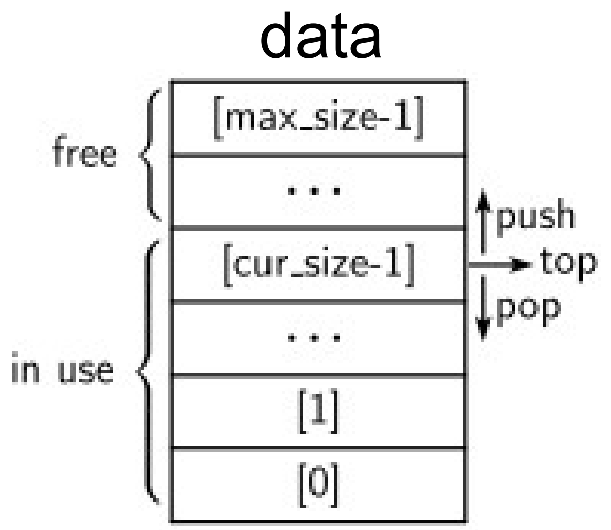
  - A list

  - A linked list (keeping the top of the stack at the head of the list)

    More information later …

Implementing a Stack (1)

- Implementing a stack using an array is easy
- The bottom of the stack is at data[0]
- The **top** of the stack is at data[cur_size-1]
- ***push*** onto the stack at data[cur_size]
- ***pop*** off the item at data[cur_size-1]
- The stack is empty when cur_size ==0



max_size is the maximum size of the stack

cur_size is the current size of the stack (= number of elements in the stack)

Implementing a Stack (2)

- Implementing a stack **using a list** isn't that bad either…

- Store the items in the stack in a list
- The top of the stack is the head, the bottom of the stack is the end of the list

- *push* by adding to the front of the list
- *pop* by removing from the front of the list

A Python class

```python
class Stack:
    '''Class implementing the ADT Stack with a list'''
    def __init__(self):
        '''creates the stack by initializing a list'''
        self.items = []

    def push (self, item):
        '''add a new element to the top of the stack'''
        self.items.append(item)

    def top (self):
        '''returns the top element of the stack without removing it'''
        return self.items[-1]

    def pop (self):
        '''removes the element at the top of the stack'''
        return self.items.pop()

    def size(self):
        '''returns the current size of the stack'''
        return len(self.items)


    def clear(self):
        '''destruction of all content of the stack'''
        self.items = []
```

Applications for Stacks

- Evaluating postfix expressions
  - Practical classes exercise

- Program execution
  - Use to store information when a function or method is called
- Reversing order
- Checking braces in expressions
- …

Reversing a Word

- We can use a stack to reverse the letters in a word
- How?

- Push each letter in the word onto a stack
- When you reach the end of the word, pop the letters off the stack and print them out
  - Until the stack is empty

- Implement it yourself…

Checking balance of braces

- (()) balanced braces
- ()((()())))) not balanced braces

- More complex expressions

  (a + b) * (c – d)   # OK

  (a + b) / c – d   # NOT OK

- How can you use a Stack to check a brace is balanced or not?

Checking balance of braces

- To simplify we consider expressions only composed by single character elements

- Algorithm:

```
1. Read the expression
2. Create a Stack
3. Initialize 2 variables: for open and close braces
4. FOR all chars in the expression
     • Extract char from expression
     • IF char is opening brace  push it
     • IF char is closing brace  get and pop element at top of stack
     •       Check if it is of same type of the last open brace
5. IF stack is empty expression is correct
```

Possible solution

```
from Stack import Stack

# Read the expression
expression = input("Insert you expression : ")

# Create a Stack
stack = Stack()

# Initialize 2 variables: for open and close braces
opening_braces = "(["    # NOTE opening and closing braces in same order
closing_braces = ")]"

# FOR all chars in the expression
valid = True
for char in expression:

    # IF char is opening brace  push it
    if opening_braces.find(char) != -1:
        stack.push(char)

    #IF char is closing brace  get and pop element at top of stack
    …
```

Code

Possible solution

```
    #IF char is closing brace  get and pop element at top of stack
    if closing_braces.find(char) != -1:
        if not stack.isEmpty():
            brace = stack.pop()

            # Check if it is of same type of the last open brace
            if closing_braces.find(char) != opening_braces.find(brace):
                print(f"Invalid expr.: '{char}' does not match '{brace}'")
                valid = False
                break
        else:
            print(f"Invalid expression: '{char}’ without an opening brace")
            valid = False
            break

# IF stack is empty (and no problems detected before) expression is correct
if valid and stack.isEmpty():
    print("Braces ok :)")
else:
    print("Braces NOT OK :(")
```

More info

- See:
  - Stack in Python - GeeksforGeeks

Queue

PT: Fila

How it works ?

Enter the queue
(at the end)

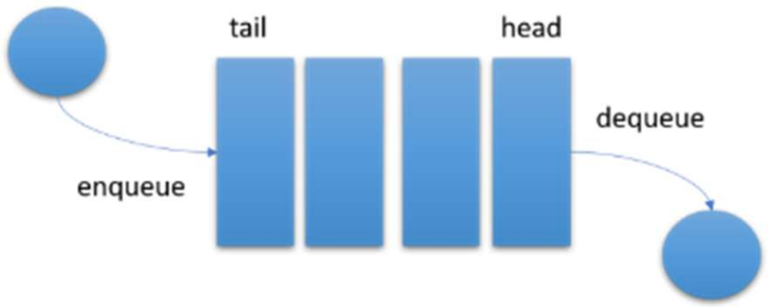
Exit the queue
(at the beginning)

What is a queue?

- A data structure of items such that items can be inserted only at one end and removed at the other end.
- Example
  - A line at the supermarket
- A queue is called a FIFO (First in-First out) data structure.

What can we do with a Queue?

- Enqueue
  - Add an item to the queue



- Dequeue
  - Remove an item from the queue
- Look at the item at the front
  - Without removing it
- Enqueue can also be called insert or add
- Dequeue can also have alternative designations:
  - remove
  - getFront

Queue ADT

- ❖ Operations:
  - – creation of the queue
  - – destruction of the queue

  - – add a new element to the end of the queue
  - – remove the element at the beginning of the queue
  - – take a look at the first element of the queue

  - – determine the current size of the queue

  - – check if queue is empty (⇔ size == 0)

Typical Queue Interface (in Python)

```
class Queue:
    def __init__(self):   # to create a Queue

    def add(self, item):
    def remove(self):
    def first(self):

    def size():

    def isEmpty():  # return True if queue is empty
```

Implementing a Queue (1)

- Just like a stack, we can implement a queue:
  - Using an array
  - Using a list

- Using an array to implement a queue is significantly harder than using an array to implement a stack.
- Why?
  - Unlike a stack, where we add and remove at the same end, in a queue we add to one end and remove from the other.

Implementing a Queue (2)

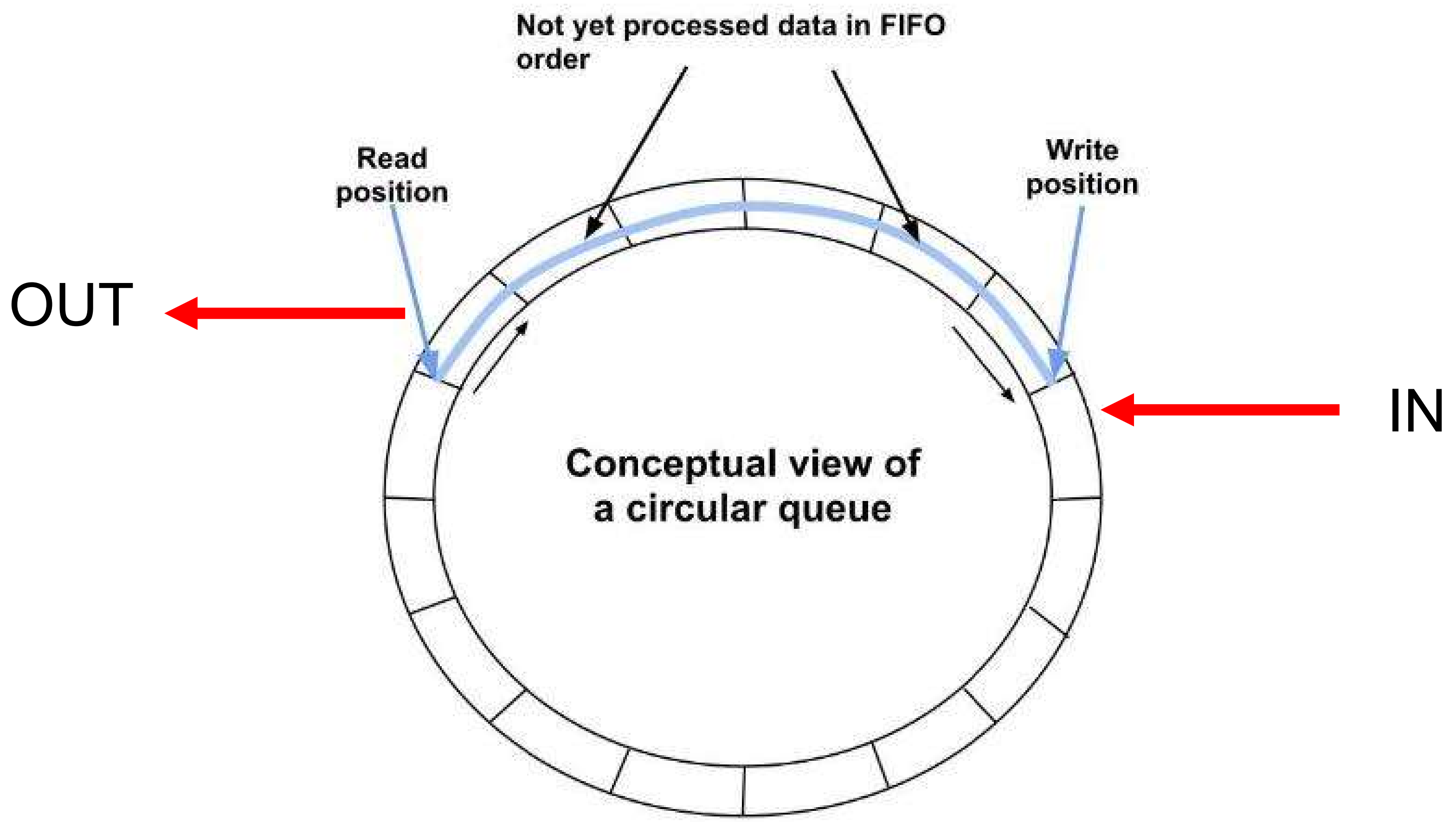
- There are two options for implementing a queue using an array:

- Option 1:
  - *Enqueue* at data[0] and **shift all the rest** of the items in the array down to make room.
  - *Dequeue* from data[numItems-1]
  - Not the best option ☹

    Shifting takes time

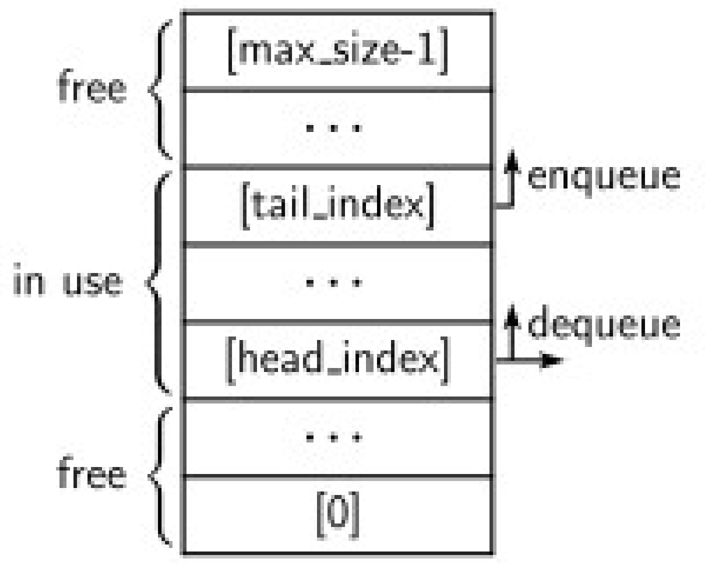
Implementing a Queue (3)

- Option 2 – using a circular buffer


Not yet processed data in FIFO order
Read position
Write position
OUT
IN
Conceptual view of a circular queue

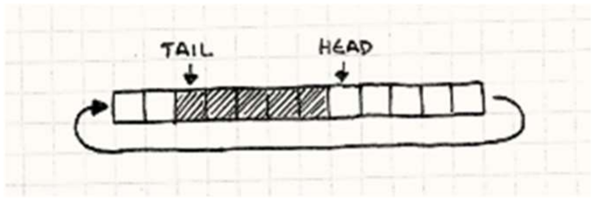
Implementing a Queue (4)

- Using a circular buffer
  - *Enqueue* at data[tail_index+1]
  - *Dequeue* at data[head_index]



- The **tail_index** variable always contains the index of the last item in the queue.
- The **head_index** variable always contains the index of the first item in the queue.

- When these variables reach the end of the array, wrap around to the front again.



Implementing a Queue (5)

- Implementing a queue **using a list** is easy as implementing the Stack
- Front of the queue is stored as the beginning of the list; rear of the queue is stored at the tail.
- *Enqueue* (**add**) by adding to the end of the list
- *Dequeue* (**remove**) by removing from the front of the list

Applications of queues

- Waiting lists
  - At hospitals, supermarkets, fast food restaurants
- Queueing of packets for delivery in networks
- Queueing of email messages to be sent
- Round-robin scheduling in processors
  - Use to decide the application they run
- Input/Output processing
  - Example: for the keys you press in your keyboard

More info

- Queue in Python - GeeksforGeeks

Linked List

PT: Lista encadeada / Lista ligada

Motivation

- Vectors (arrays) allow
  - keep elements preserving their order
  - random access,

    i.e., quick direct access to any element, in any order.
- However, vectors have limitations:
  - Their capacity must be fixed when they are created

    This forces you to oversize a vector when the number of elements is not known at the outset.

    Or, resize the vector when new elements arrive, with costs in processing time.
- Inserting or removing elements in an intermediate position can take a long time if you need to move too many elements.

Linked List
(PT: Lista encadeada / Lista ligada)

- A linked list is a sequence of data structures (items) which are connected together via links.
  - Forming a chain

Linked List
(PT: Lista encadeada / Lista ligada)

- Sequential data structure in which each element of the list contains a reference to the next element.
  - In the last element, the reference is null.

- Unlike the vector, it is **completely dynamic.**
- However, it requires sequential access.

- Uses a helper structure (a node) to store each element.

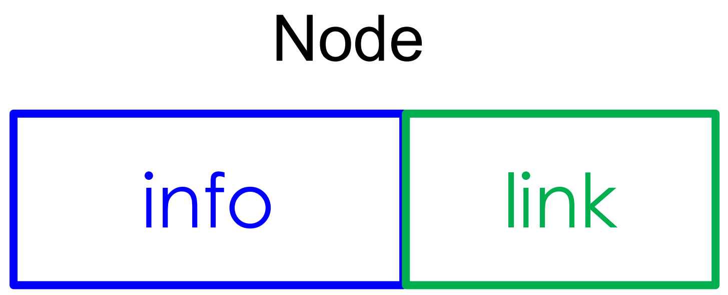
How it works?

- By creating node objects …
  - Holding the data or reference to objects containing data
- connecting them …
- and keeping information regarding (at least) where is the first node

Node

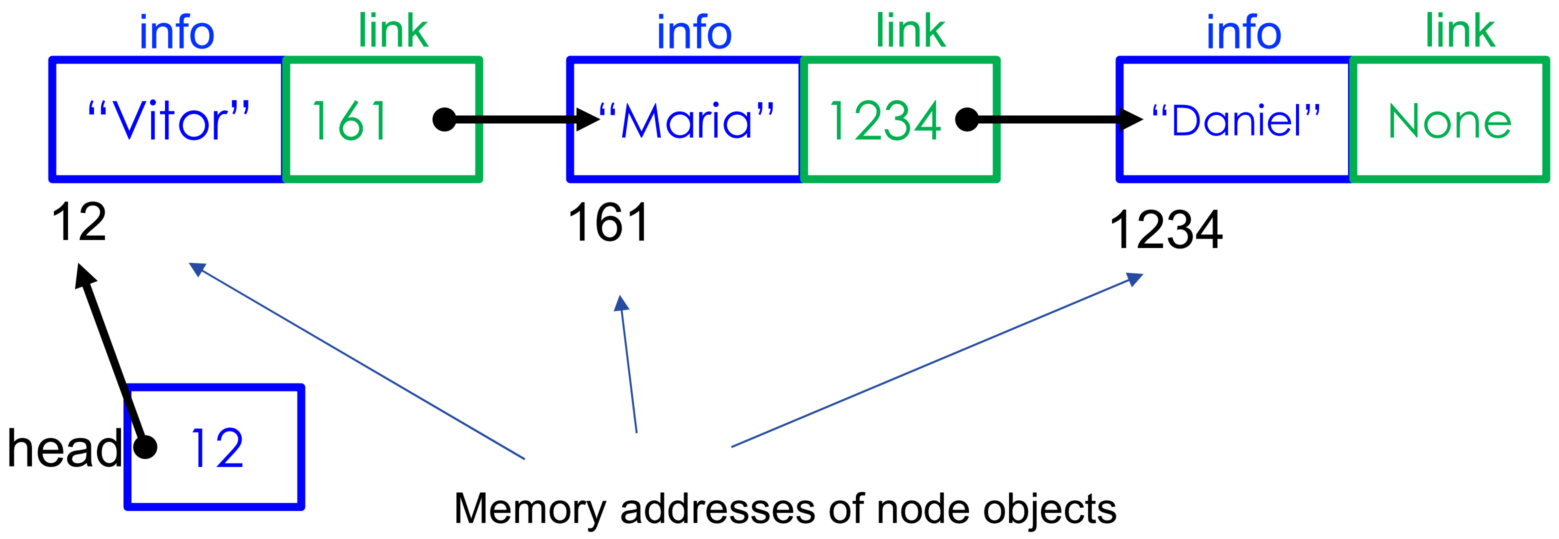
- The Node (represented graphically at right) is the key structure in a Linked List
- It represents a data structure composed by one or more information fields (**info**) and one or more connections (**link**)

- **info** stores information or a reference for where information is stored



Linked list

- **link** allows pointing to another node object to create a chain
  - It contains the address of the next node in the list information or a reference to where information is stored



Types of Linked List

- Simple Linked List
  - Item Navigation is forward only.
  - Only 1 link

    For next

- Doubly Linked List
  - Items can be navigated forward and backward way
  - Nodes keep 2 links

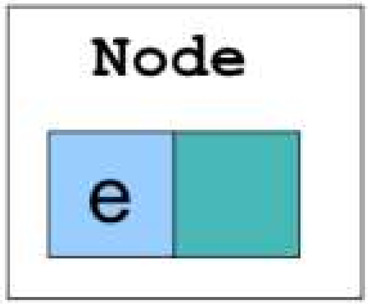
    One for next

    Other for previous

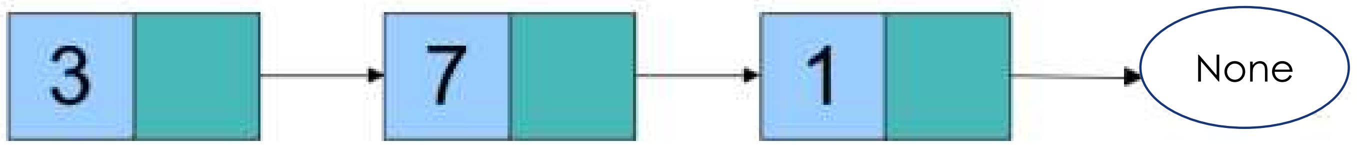
- Circular Linked List
  - Last item contains link of the first element as next and and first element has link to last element as previous.

Examples (1)

- Singly-Linked List

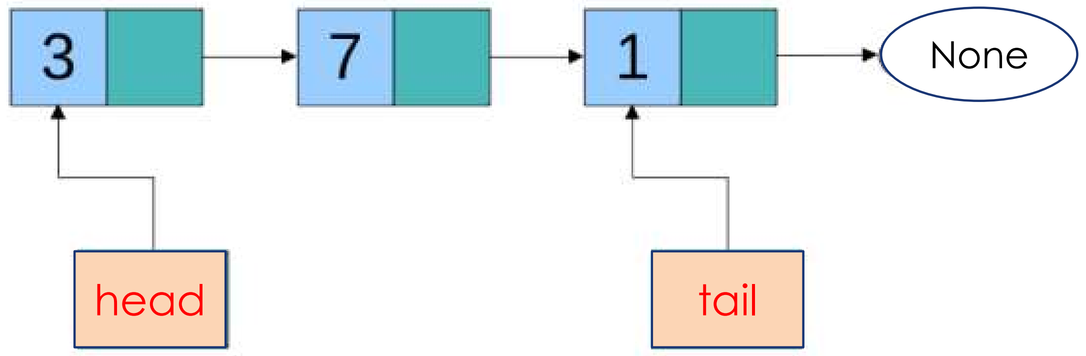


```
class Node:
    def __init__(self, dataval):
        self.info = dataval
        self.link = None
```



Examples (2)

- Example: list with elements 3, 7 and 1



- In this implementation, the list has direct access to first and last element
  - To make easier access to both
- It is easy to add elements at the beginning and end
- It is easy to remove elements at the beginning
- Why is not easy at the end ?

What can we do with it ?

- Insertion at the beginning
- Insertion at the end
- Insertion in a determined position

- Change the value of an element
- Sorted insertion
- Sorting
- Deletion of an element of the list
  - Based on content
  - Based on position
- inversion

ADT (for most relevant operations)

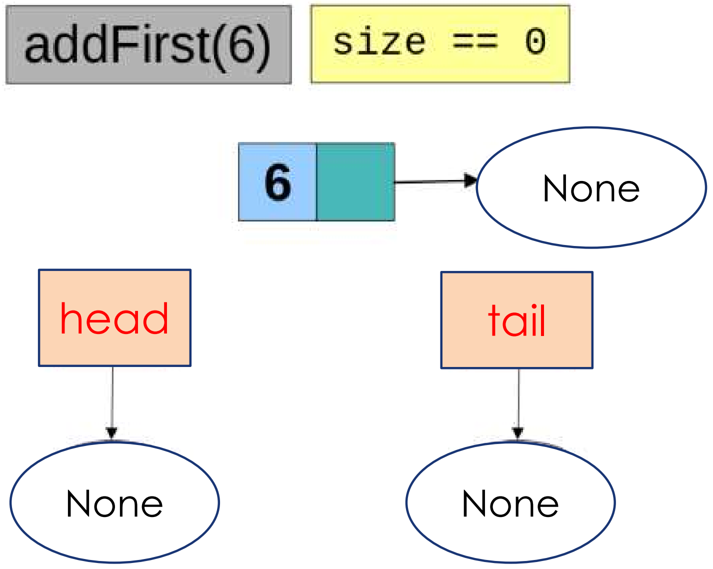
- **addFirst**
  - add an element at the beginning of the list.
- **addLast**
  - add an element at the end of the list.
- **first**
  - Return first element of the list.
- **last**
  - Return last element of the list
- **removeFirst**
  - delete element at the beginning of the list.
- **size**
- **isEmpty**
- **clear**

Skeleton

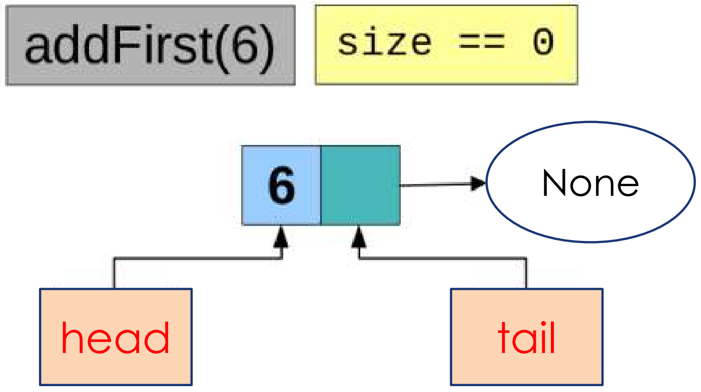
```
class LinkedList:
    def addFirst(self, elem):
        ''' add elem at the beginning '''
    def addLast (self, elem):
        ''' add elem at the end '‘’

    def first (self):
        ''' return value of first element '''
    def last (self):
        ''' return value of last element '‘’

    def removeFirst (self):
        ''' delete first element (returning it) '‘’

    def size (self):
        ''' return number of elements in the linked list '''
    def isEmpty (self):
        ''' return True if list is empty '''
    def clear (self):
        ''' reset list to an empty list '''
```

Implementation – addFirst (empty list)

1. Create a new node

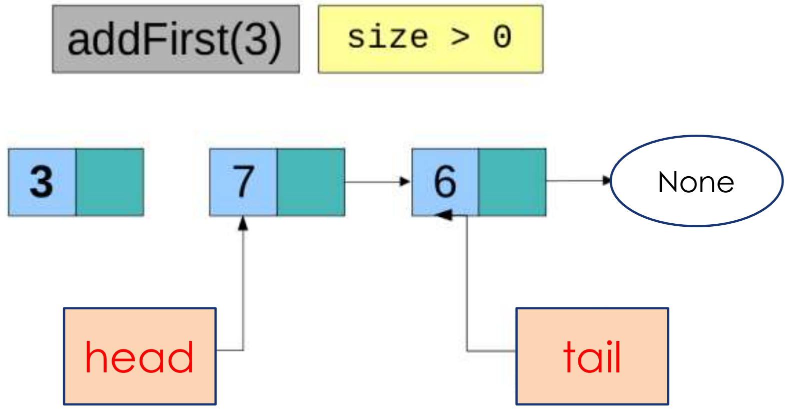


2. Update first and last
   - With node address

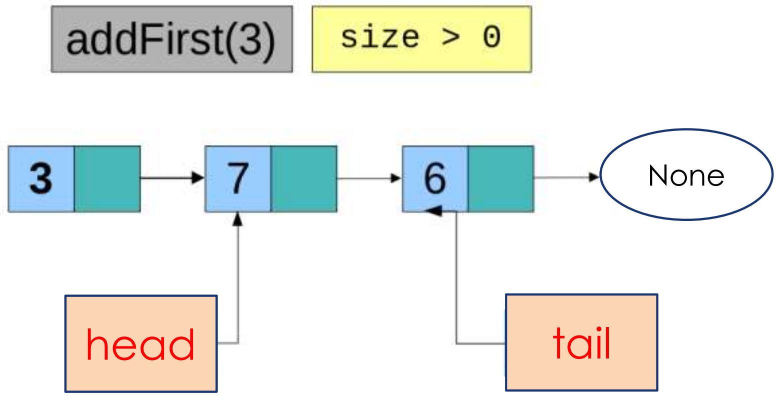


Implementation - addFirst (non-empty)

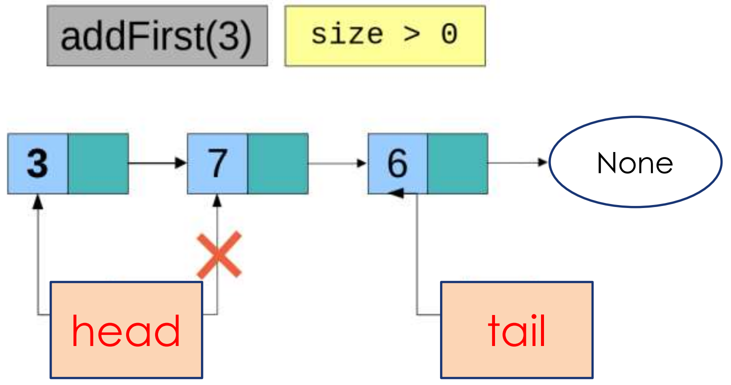
1. Create node



2. Use link to connect new node to current first node



3. Update first to the address of new node



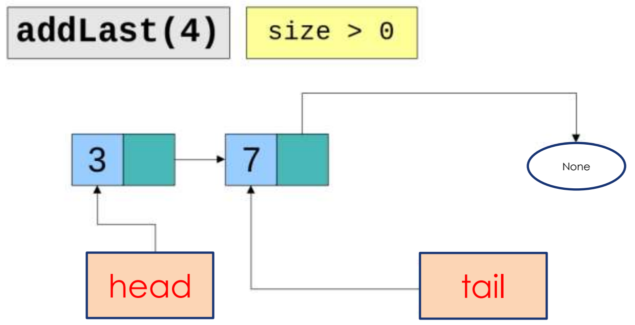
addFirst - Python code

```python
def addFirst(self,element):
        ''' add at the beginning'''

        # create new node
        new_node = Node(element)

        # connect new node to current head node
        new_node.link = self.head

        # update head (to the address of new node)
        self.head = new_node

        # if empty update tail (with new node address)
        if self.tail is None:
            self.tail = new_node

        # increase number of items
        self.nitems += 1
```

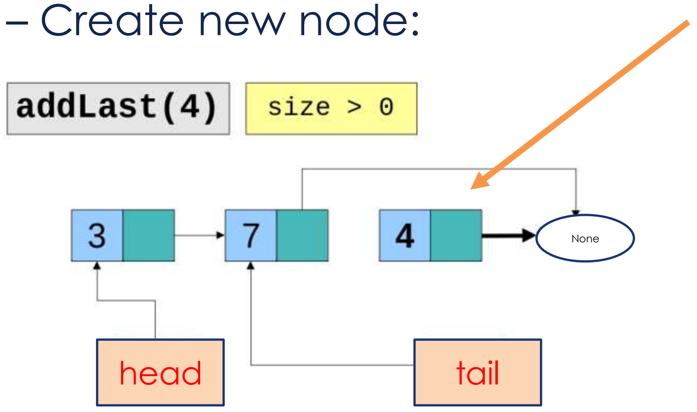
addLast

- For empty list is the same as addFirst

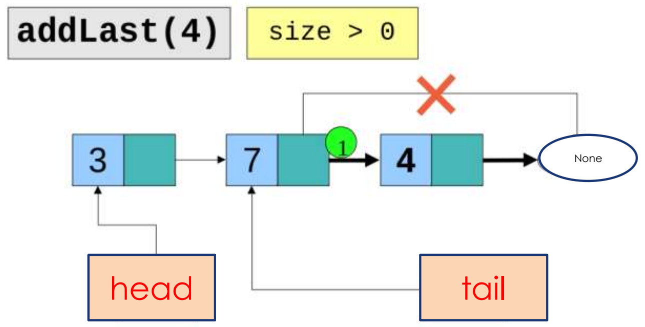
1 - Before insertion:



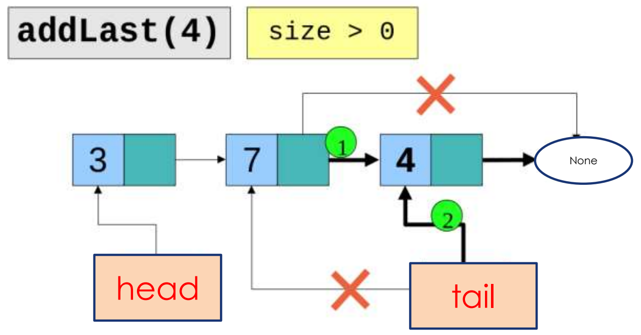
2 – Create new node:



3 – Connect new node (to node at end):



3 – Update last to de address of new node:



addLast - Python code

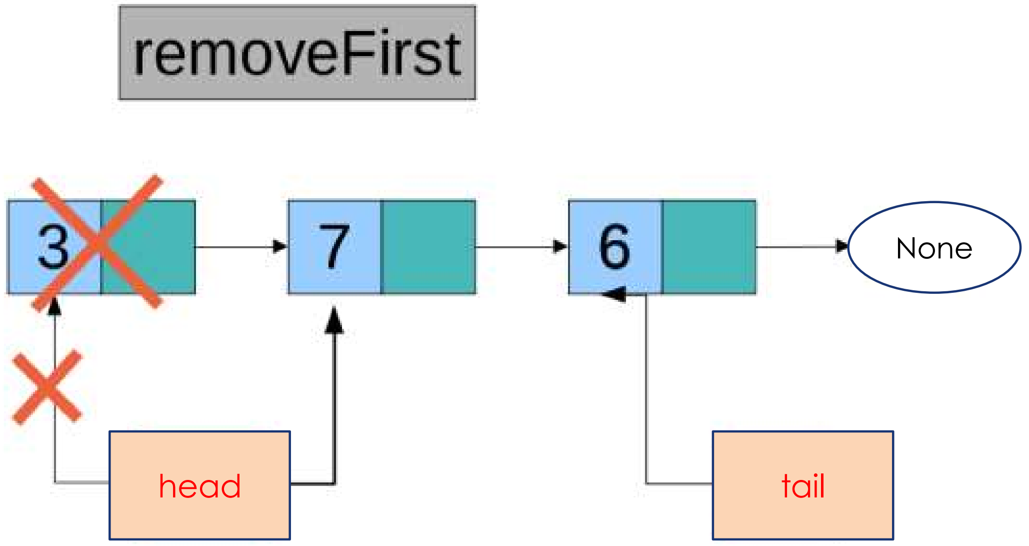
```
def addLast(self, element):
    ''' add element at the end '''

    # create new node
    new_node = Node(element)

    # update head if empty
    if self.head is None:
        self.head = new_node
    else:  # connect new node at the end
        self.tail.link = new_node

    # update tail (to the address of new node)
    self.tail = new_node

    # increase number of items
    self.nitems += 1
```

Remove First

- Size > 1

- Essentially changing the address kept in first to the second node

```
head = head.link
```



Python implementation

- A complete implementation is available as companion code for Practical Guide #3 (Data Structures)
  - Study it

Non-linear data structures