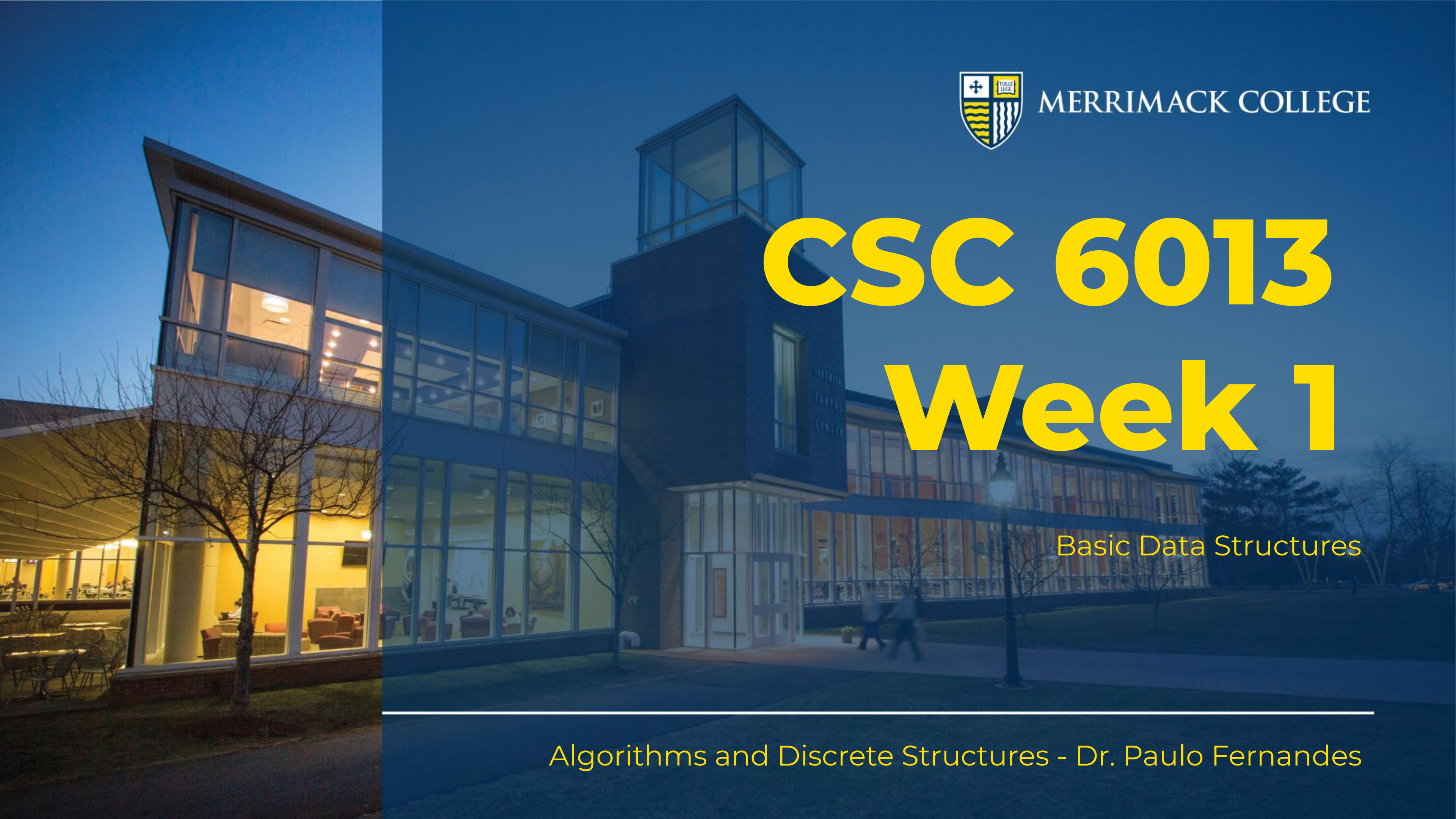


MERRIMACK COLLEGE

CSC 6013 Week 1

Basic Data Structures

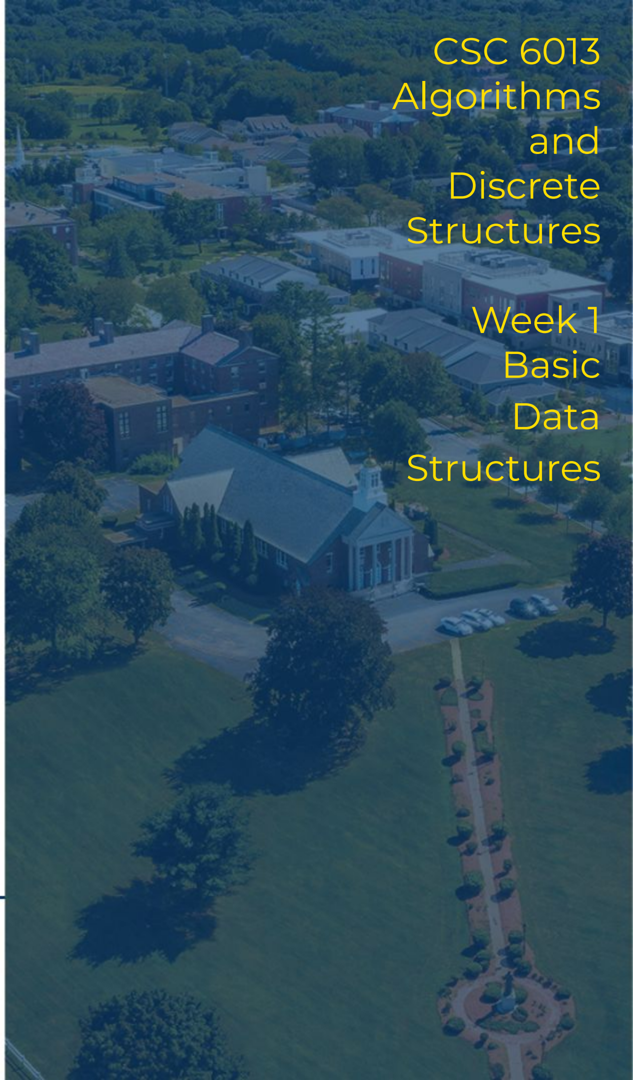
Algorithms and Discrete Structures - Dr. Paulo Fernandes

Presentation Agenda

Week 1

- Course Introduction
  a. Course format
  b. Evaluation
  c. Timeline
- Basic Data Structures
  a. Arrays
  b. Linked Lists
  c. Other Data Structures
- This Week's tasks

MERRIMACK COLLEGE



CSC 6013
Algorithms
and
Discrete
Structures

Week 1
Basic
Data
Structures

Course Introduction

We will introduce algorithm design and analysis principles.

Welcome!

This course is entitled CSC6013 Algorithms and Discrete Structures.

This course comes just after the Foundations of Programming course (CSC6003), and it followed by the Advanced Algorithms course (CSC6023), which is the continuation of the important topic of Algorithms. This is probably the more important course in the formation of a Computer Scientist.

- Course format
- Timeline
- Evaluation
- Python

Course Format

Success Coach and Graduate Advising

- ecs-grad-advising@merrimack.edu

This course is fully online, all weeks will have:

- Live sessions every Mondays 6:30pm to 8:30pm
- Office Hours every Wednesdays 8:30pm to 9:30pm

Every week, but the last, we will have:

- In-class exercises tasks until Friday
- Quizzes to be taken from Friday to Next Monday
- Coding assignments to be turned in until Next Monday

Last week has:

- Final exam to be done until Saturday

Student Tutors available!

Visit the Hub!

Send me an email if you are planning to attend office hours

Email subjects have to start with CSC6013

Communication

- fernandesp@merrimack.edu

Course Format

- Live sessions every Mondays 6:30pm to 8:30pm
    - Meeting ID: 960 3746 8498 Passcode: CSC6013
- Office Hours every Wednesdays 8:30pm to 9:30pm
    - Meeting ID: 960 3746 8498 Passcode: CSC6013

- In-class exercises tasks until Friday
    - **Simple exercises**, few or no coding
- Quizzes to be taken from Friday to Next Monday
    - **Ten multiple choice** questions, open book, open notes, and **untimed**
- Coding Projects to be turned in until next Monday
    - **Programming tasks** (mostly)
- Final exam to be done until last Saturday of classes
    - **8 questions, 4 hours to take it**, open book, open notes

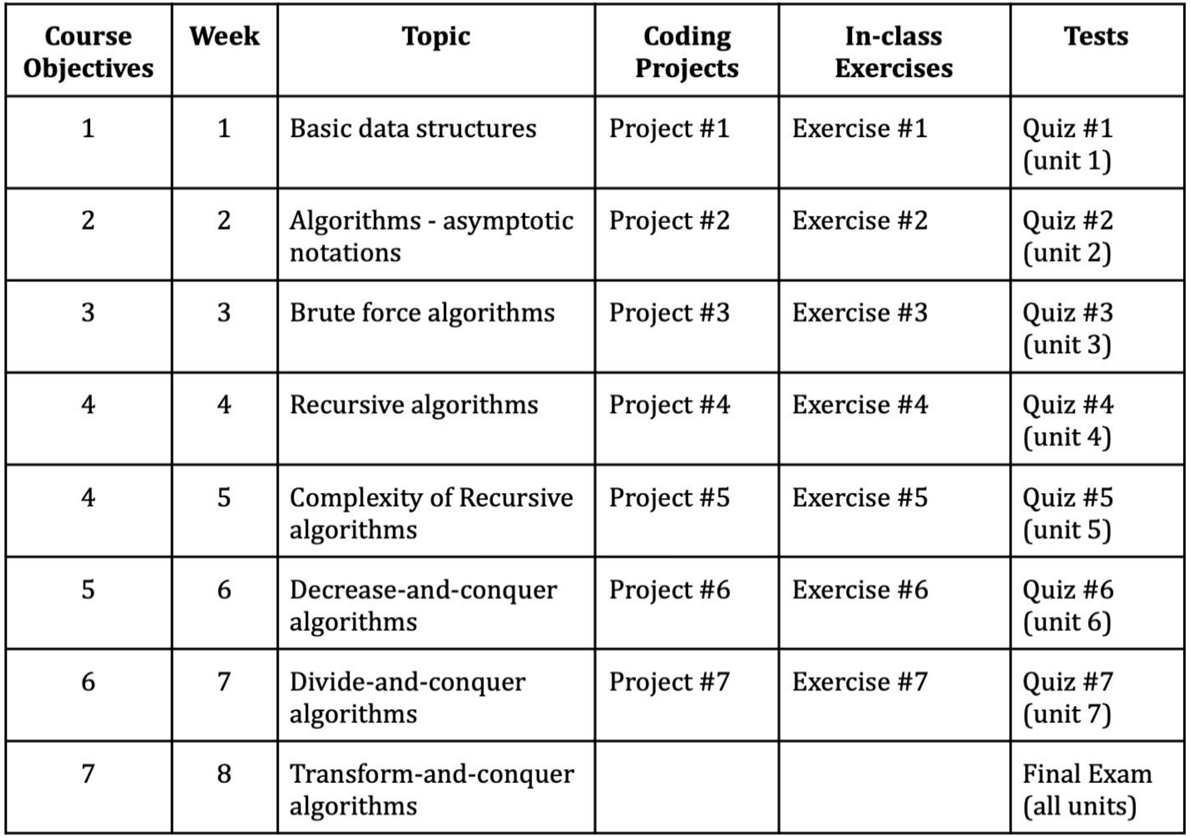
Attendance is optional, but strongly recommended!

Timeline

This is a very fast-paced course (better get used to).

Try to check it out the topics before the live session.

Any change will be informed in class and on Canvas.

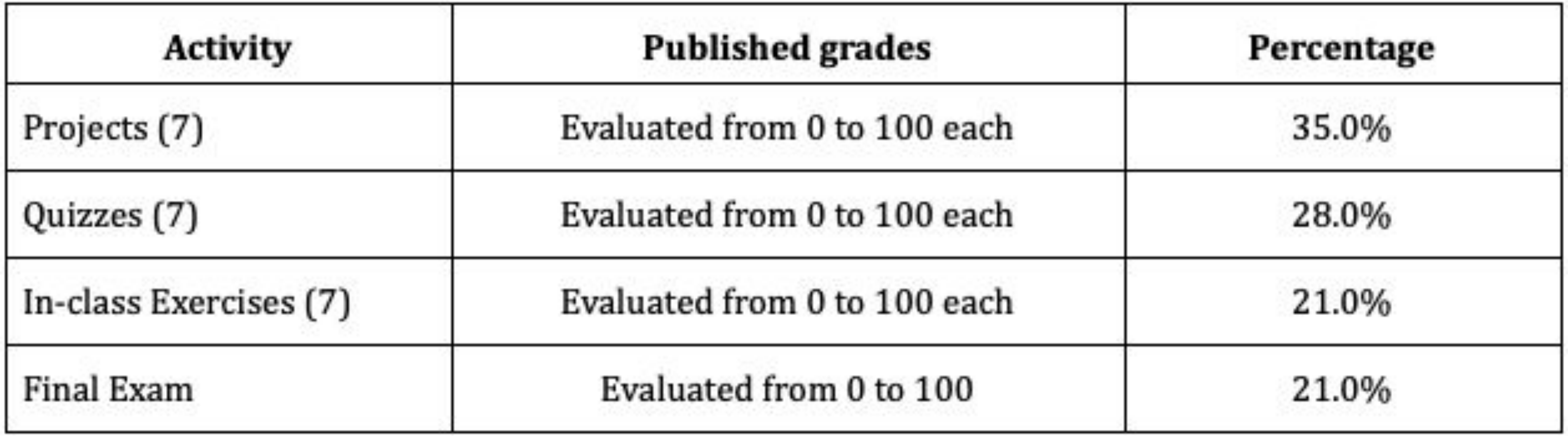


Course Objectives	Week	Topic	Coding Projects	In-class Exercises	Tests
1	1	Basic data structures	Project #1	Exercise #1	Quiz #1 (unit 1)
2	2	Algorithms - asymptotic notations	Project #2	Exercise #2	Quiz #2 (unit 2)
3	3	Brute force algorithms	Project #3	Exercise #3	Quiz #3 (unit 3)
4	4	Recursive algorithms	Project #4	Exercise #4	Quiz #4 (unit 4)
4	5	Complexity of Recursive algorithms	Project #5	Exercise #5	Quiz #5 (unit 5)
5	6	Decrease-and-conquer algorithms	Project #6	Exercise #6	Quiz #6 (unit 6)
6	7	Divide-and-conquer algorithms	Project #7	Exercise #7	Quiz #7 (unit 7)
7	8	Transform-and-conquer algorithms			Final Exam (all units)

Don't let tasks pile up!

Evaluation

The evaluated tasks are weighted as such (105%):



Activity	Published grades	Percentage
Projects (7)	Evaluated from 0 to 100 each	35.0%
Quizzes (7)	Evaluated from 0 to 100 each	28.0%
In-class Exercises (7)	Evaluated from 0 to 100 each	21.0%
Final Exam	Evaluated from 0 to 100	21.0%

All tasks are due in their stated deadline, the late penalties reduced 10% of the evaluation, plus 2% for each full day of delay.

Final deadline on Friday of the course's last week

Evaluation

The final letter grade is computed according to this table:

A	A-	B+	B	B-	C+	C	C-	F
95 and up	90 to 94.9	87 to 89.9	83 to 86.9	80 to 82.9	77 to 79.9	73 to 76.9	70 to 72.9	69.9 and low

There are no grade D+, D, or D- in graduate courses. Only C or above are passing grades. An average grade B is required to the Masters of Science in Computer Science, lower than that puts you in probation.

Final grades published first Monday after the course's end.

The Python language

In the unlikely case you don't have Python in your machine, do install it now!

- Python Language - A new language after version 3;
  - Any version is fine, but to make things easier, I suggest you to use a version 3.9 or after, since before that there were more significant changes for basic Python commands;
- The default Integrated Development Environment (IDE) that comes with Python is IDLE, a simple and reliable environment;
  - You are free to use other IDE (PyCharm, VScode, etc.), but in this class we will assume everyone is using IDLE, since if a piece of code runs on IDLE it will run in any other environment, but the inverse is not true.

Resource: Python.org.

Basic Data Structures

To structure data we create an abstraction, a different, usually more restrictive, way to access it.

We structure data in order to better grasp it.

There are implementation choices, but according with the usage we define what can be done or not with a data structure. As such, the definition of what can be done may guide us to choose an implementation.

- **Arrays**;
- Linked Lists;
- Other Structures.

Basic Structures - Arrays

addr [0]	addr [1]	addr [2]	addr [3]	addr [4]
3	5	6	7	8
x	x+s	x+2s	x+3s	x+4s
x+0s	x+1s	x+2s	x+3s	x+4s

x - array address
s - size of elements

Arrays in Python

- In Python, depending the version details, arrays are implemented using Python Lists, which may or may not be actually implemented as regular arrays, with an amount of memory equal to:
  - Number of elements times size of each element;
- This amount of memory is indexed by a common arithmetic operation, so data has to be contiguously disposed in the memory;
- We will assume this is the way arrays are implemented in Python, as it serves our theoretical analysis needs.

Wikipedia: Array (data structure).

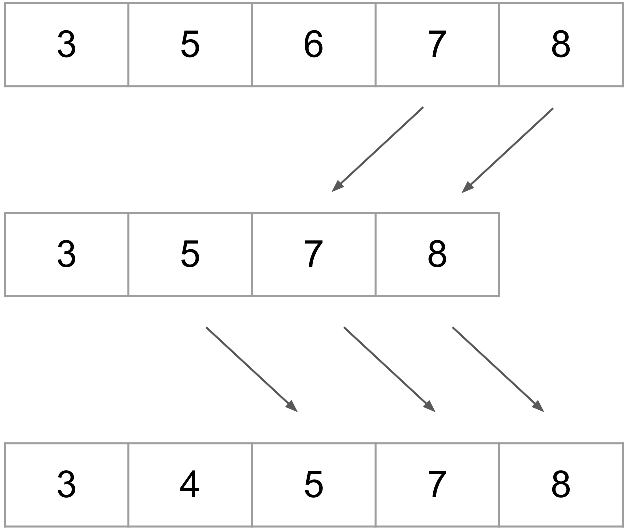
Basic Structures - Arrays

Arrays are problematic when you need to insert or remove elements:

- To insert or remove an element in an array you need to “scooch over” (copy along) elements.

Removing element 6 (third position)

Inserting element 4 at the second position



3	5	6	7	8

3	5	7	8

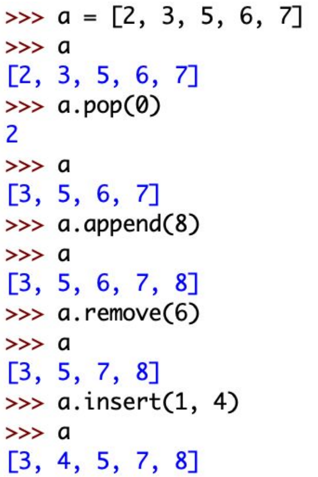
3	4	5	7	8

Arrays are kind of stiff...

Basic Structures - Arrays

In Python, insertion and removal of elements is encapsulated in List methods:

- *<list>*.***pop***(*<position>*)
  - Remove the element in *<position>*
- *<list>*.***append***(*<data>*)
  - Insert element *<data>* at the end;
- *<list>*.***remove***(*<data>*)
  - Remove first instance of element *<data>*;
- *<list>*.***insert***(*<position>*, *<data>*)
  - Insert element *<data>* in *<position>*;



```
>>> a = [2, 3, 5, 6, 7]
>>> a
[2, 3, 5, 6, 7]
>>> a.pop(0)
2
>>> a
[3, 5, 6, 7]
>>> a.append(8)
>>> a
[3, 5, 6, 7, 8]
>>> a.remove(6)
>>> a
[3, 5, 7, 8]
>>> a.insert(1, 4)
>>> a
[3, 4, 5, 7, 8]
```

Despite being single commands, each one implies operational costs.

Arrays are not the only option.

Basic Data Structures

To structure data we create an abstraction, a different, usually more restrictive, way to access it.

We structure data in order to better grasp it.

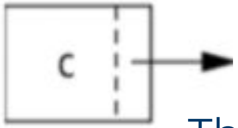
There are implementation choices, but according with the usage we define what can be done or not with a data structure. As such, the definition of what can be done may guide us to choose an implementation.

- Arrays;
- **Linked Lists**;
- Other Structures.

Basic Structures - Linked Lists

A structure based on nodes and pointers to other nodes

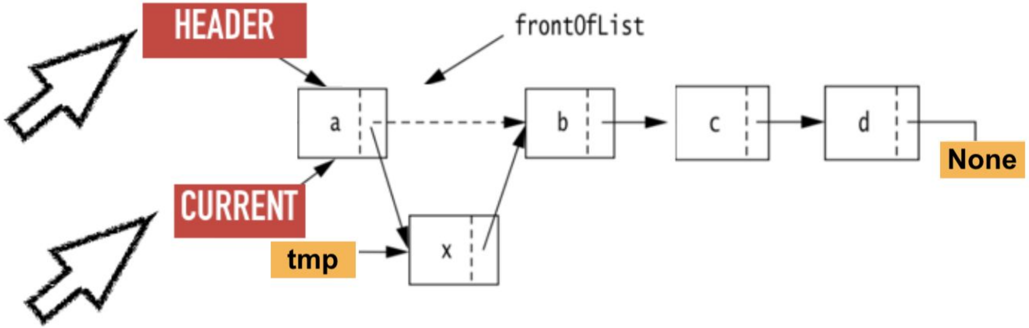
The Nodes



Data

The pointer
Next

The Linked List



Two pointers:

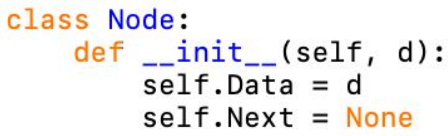
- Header
- Current

LinkedList and Node classes.

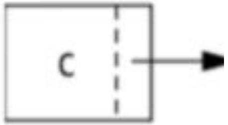
Basic Structures - Linked Lists

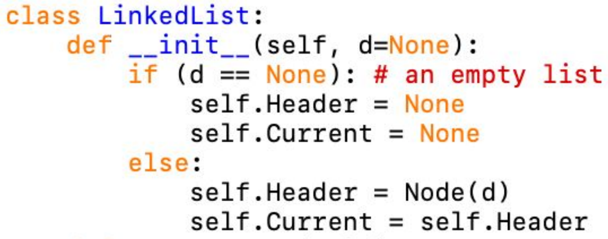
Two classes:

- The node
  - the data, and
  - a pointer to the next node;
- The linked list
  - a pointer to the first node (Header), and
  - a pointer to the current node (Current).

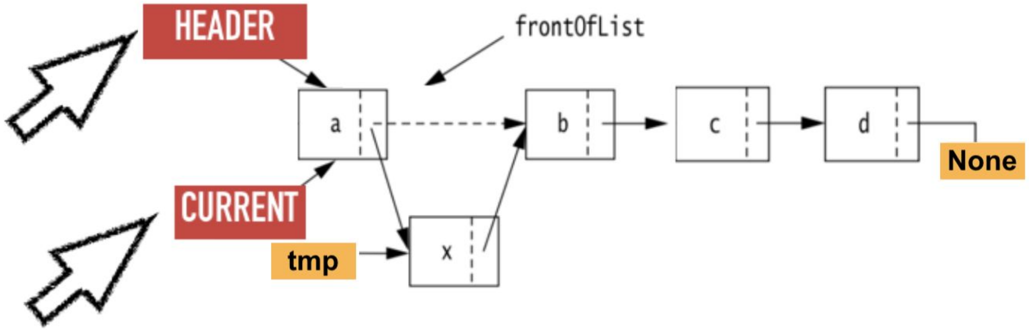


```
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None
```





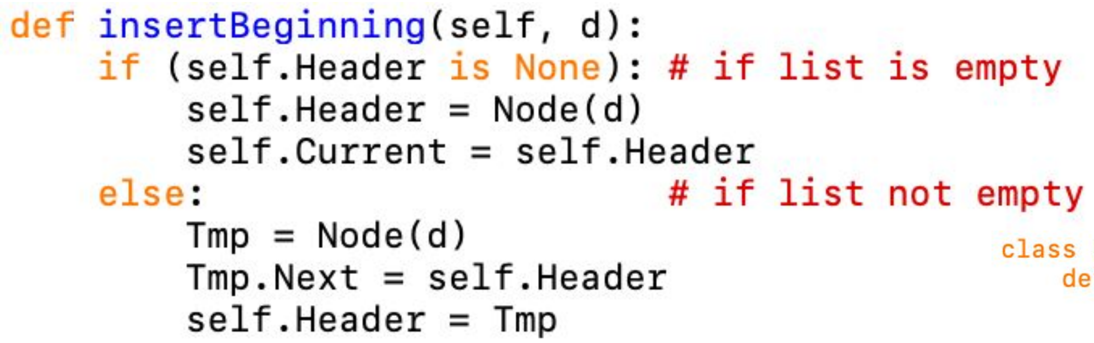
```
class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```



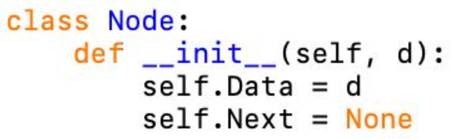
LinkedList and Node classes.

Basic Structures - Linked Lists

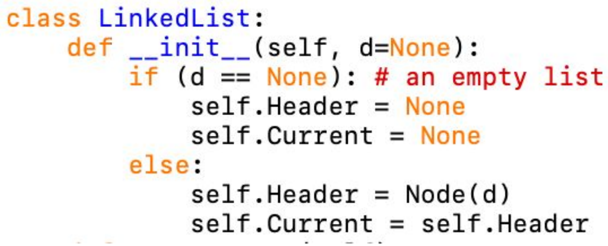
- Inserting a data ***d*** at the beginning:



```
def insertBeginning(self, d):
    if (self.Header is None): # if list is empty
        self.Header = Node(d)
        self.Current = self.Header
    else:                     # if list not empty
        Tmp = Node(d)
        Tmp.Next = self.Header
        self.Header = Tmp
```



```
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None
```



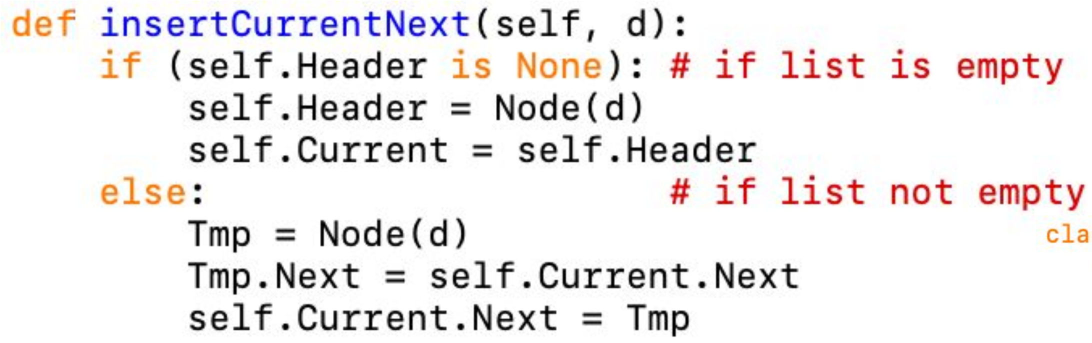
```
class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```

- If the list is empty, create a node to it;
- Else, create a new node (Tmp), this new node points to the currently first node (Header), then point Header to the new node.

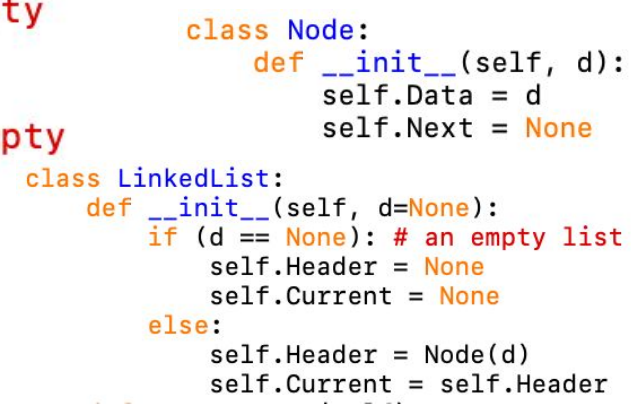
A mutator method.

Basic Structures - Linked Lists

- Inserting a data ***d*** at the next node of the Current:



```python
def insertCurrentNext(self, d):
    if (self.Header is None): # if list is empty
        self.Header = Node(d)
        self.Current = self.Header
    else:                     # if list not empty
        Tmp = Node(d)
        Tmp.Next = self.Current.Next
        self.Current.Next = Tmp
```



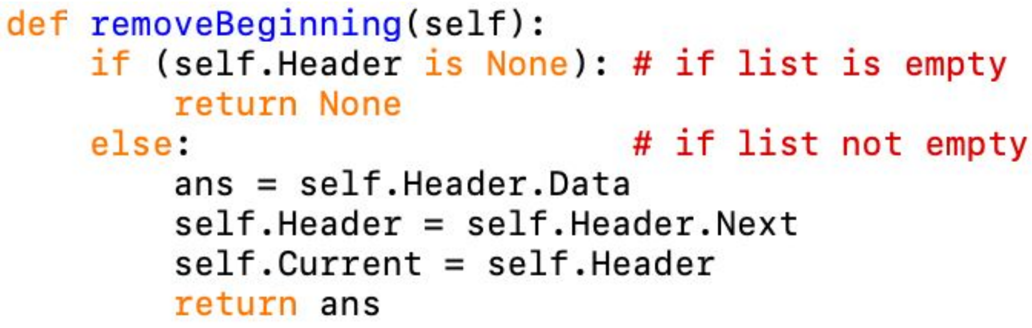
```python
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None

class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```

- If the list is empty, create a node to it;
- Else, create a new node (Tmp), this new node points to the currently next of Current node (Current.Next), then point Current.Next to the new node.

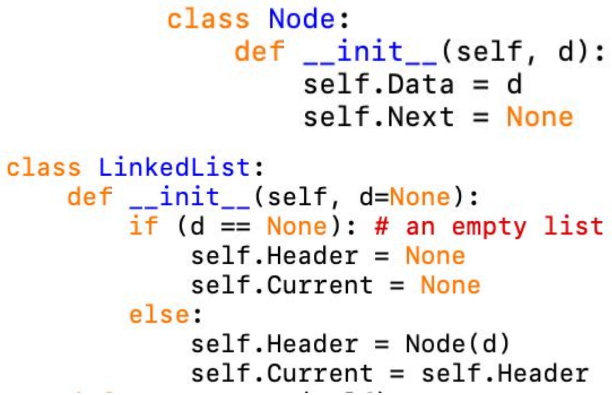
A mutator method.

Basic Structures - Linked Lists

- Removing the node at the beginning:



```
def removeBeginning(self):
    if (self.Header is None): # if list is empty
        return None
    else:                     # if list not empty
        ans = self.Header.Data
        self.Header = self.Header.Next
        self.Current = self.Header
        return ans
```



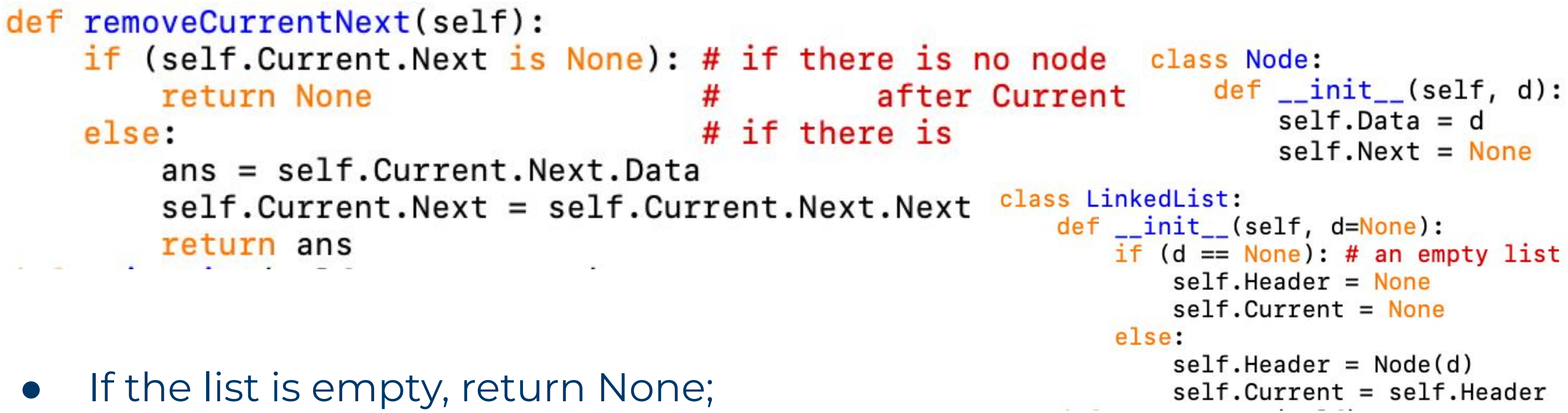
```
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None

class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```

- If the list is empty, return None;
- Else, get the the currently first node data (Header.Data), then by pass the first node by pointing Header to the currently second node.

A mutator method.

Basic Structures - Linked Lists

- Removing the node next of the Current:



```python
def removeCurrentNext(self):
    if (self.Current.Next is None): # if there is no node
        return None                 #        after Current
    else:                           # if there is
        ans = self.Current.Next.Data
        self.Current.Next = self.Current.Next.Next
        return ans
```

```python
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None
```

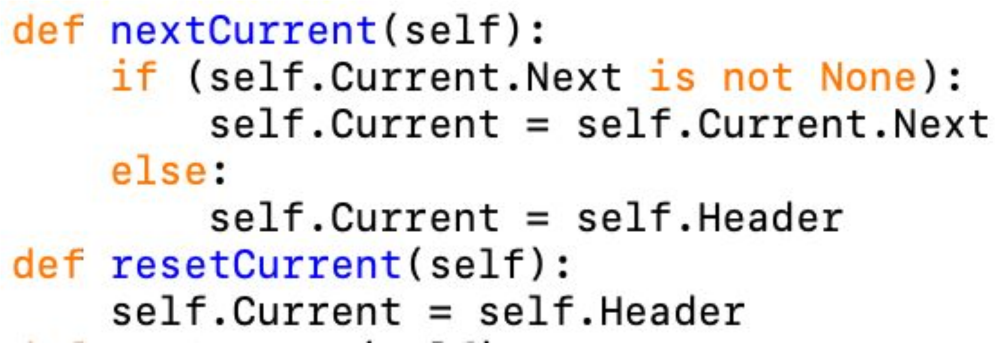
```python
class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```

- If the list is empty, return None;
- Else, get the the currently next of Current node data (Current.Next.Data), then by pass this node by pointing Current.Next to the node after it.

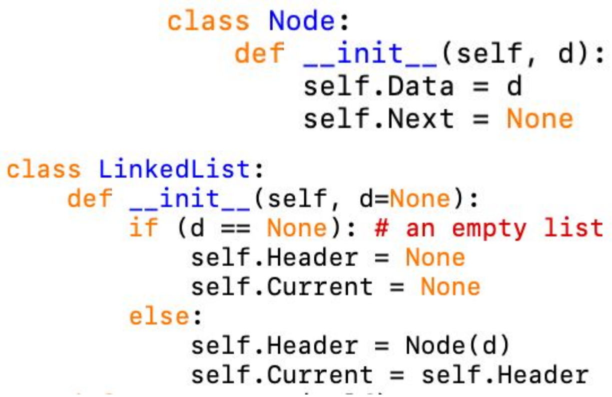
A mutator method.

Basic Structures - Linked Lists

- Moving the Current pointer:



```python
def nextCurrent(self):
    if (self.Current.Next is not None):
        self.Current = self.Current.Next
    else:
        self.Current = self.Header
def resetCurrent(self):
    self.Current = self.Header
```



```python
    class Node:
        def __init__(self, d):
            self.Data = d
            self.Next = None

class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```

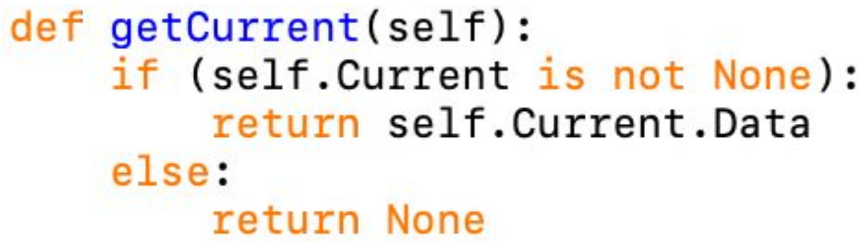
- If Current is not the last node, advance Current to the next;
- Else, reset it (Current points to the Header.

MERRIMACK COLLEGE

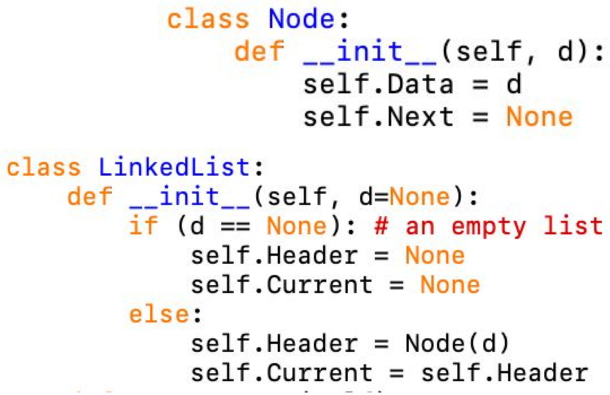
A mutator method.

Basic Structures - Linked Lists

- Checking the Current data:



```python
def getCurrent(self):
    if (self.Current is not None):
        return self.Current.Data
    else:
        return None
```



```python
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None

class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```

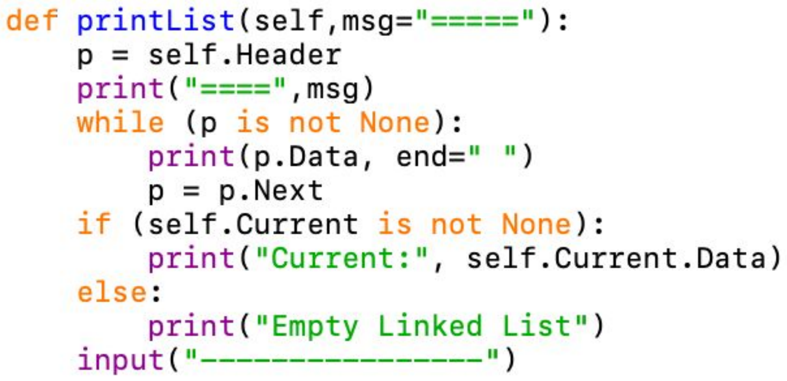
- If the list is not empty, delivers Current data;
- Else, return None.

MERRIMACK COLLEGE

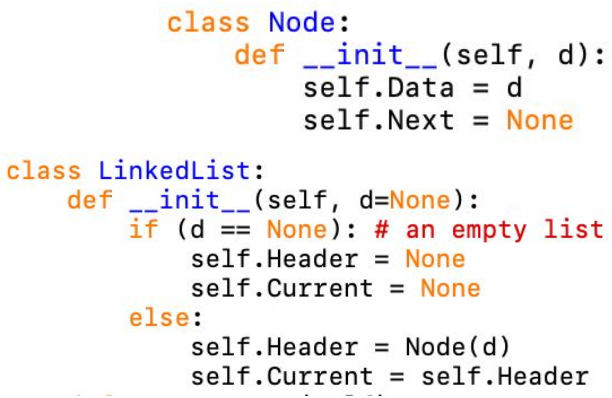
A accessor method.

Basic Structures - Linked Lists

- Printing out the list (for demo/debug purposes):



```python
def printList(self,msg="====="):
    p = self.Header
    print("====",msg)
    while (p is not None):
        print(p.Data, end=" ")
        p = p.Next
    if (self.Current is not None):
        print("Current:", self.Current.Data)
    else:
        print("Empty Linked List")
    input("-----------------")
```



```python
    class Node:
        def __init__(self, d):
            self.Data = d
            self.Next = None

class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
```

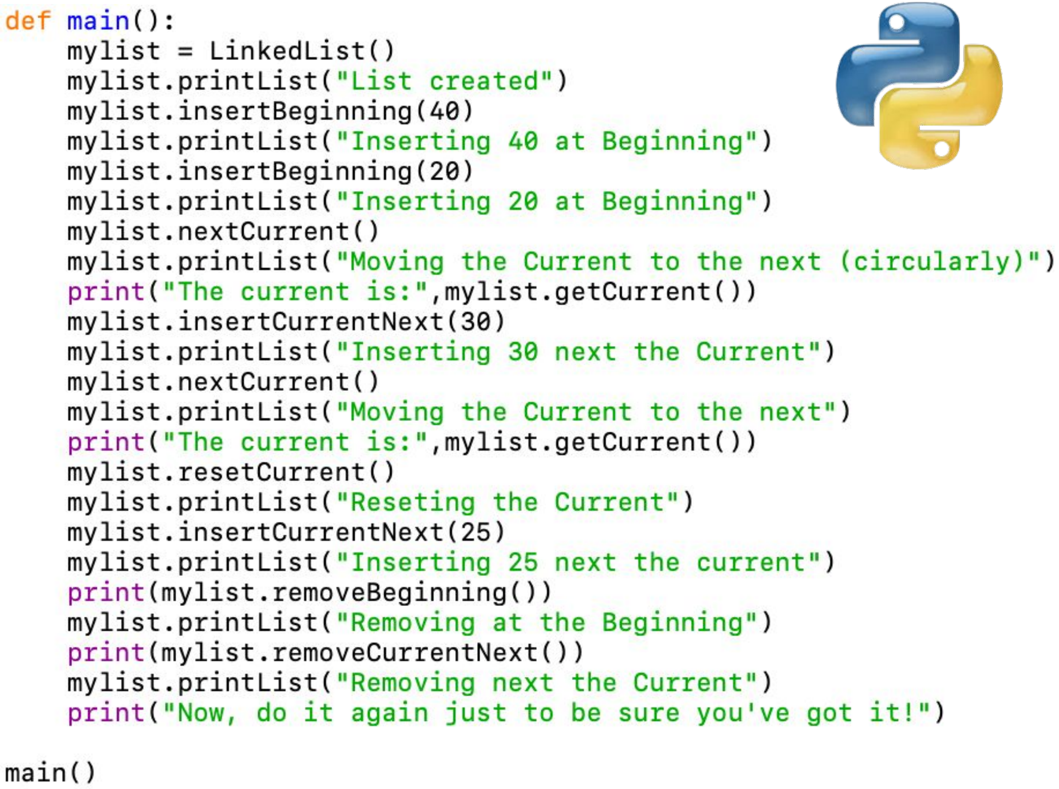
- Use a pointer p to traverse the linked list.

A accessor method.

Basic Structures - Linked Lists

- Testing it all:

Full code of Linked Lists demo here



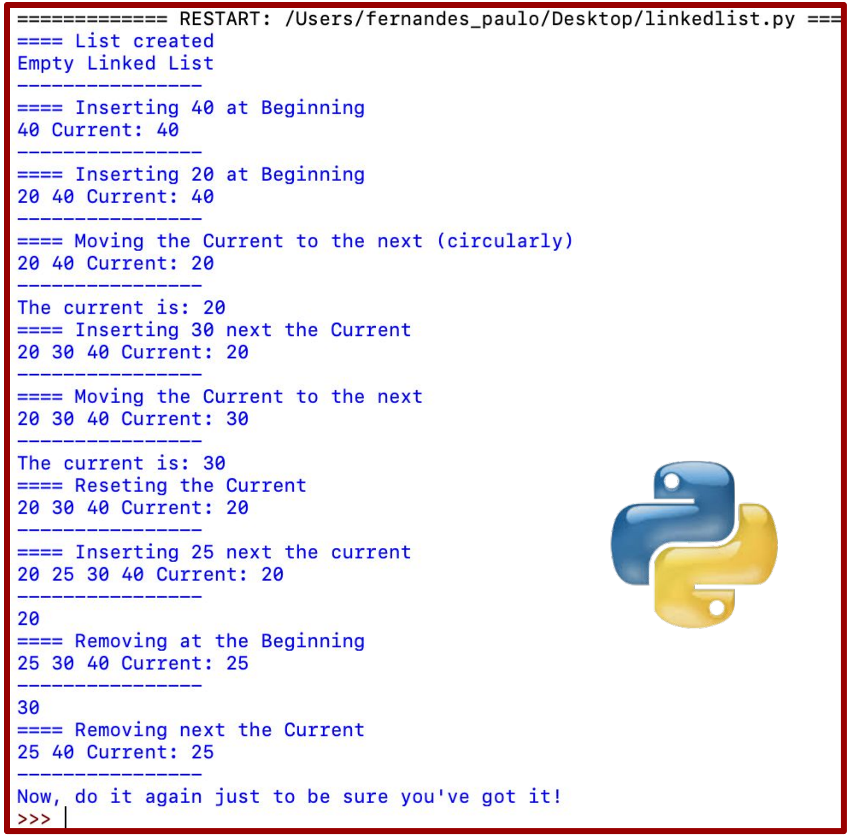
```python
def main():
    mylist = LinkedList()
    mylist.printList("List created")
    mylist.insertBeginning(40)
    mylist.printList("Inserting 40 at Beginning")
    mylist.insertBeginning(20)
    mylist.printList("Inserting 20 at Beginning")
    mylist.nextCurrent()
    mylist.printList("Moving the Current to the next (circularly)")
    print("The current is:",mylist.getCurrent())
    mylist.insertCurrentNext(30)
    mylist.printList("Inserting 30 next the Current")
    mylist.nextCurrent()
    mylist.printList("Moving the Current to the next")
    print("The current is:",mylist.getCurrent())
    mylist.resetCurrent()
    mylist.printList("Reseting the Current")
    mylist.insertCurrentNext(25)
    mylist.printList("Inserting 25 next the current")
    print(mylist.removeBeginning())
    mylist.printList("Removing at the Beginning")
    print(mylist.removeCurrentNext())
    mylist.printList("Removing next the Current")
    print("Now, do it again just to be sure you've got it!")

main()
```

Try this code by yourself.

Basic Structures - Linked Lists

- Testing it all:

Full code of Linked Lists demo here



```
============= RESTART: /Users/fernandes_paulo/Desktop/linkedlist.py ===
==== List created
Empty Linked List
----------------
==== Inserting 40 at Beginning
40 Current: 40
----------------
==== Inserting 20 at Beginning
20 40 Current: 40
----------------
==== Moving the Current to the next (circularly)
20 40 Current: 20
----------------
The current is: 20
==== Inserting 30 next the Current
20 30 40 Current: 20
----------------
==== Moving the Current to the next
20 30 40 Current: 30
----------------
The current is: 30
==== Reseting the Current
20 30 40 Current: 20
----------------
==== Inserting 25 next the current
20 25 30 40 Current: 20
----------------
20
==== Removing at the Beginning
25 30 40 Current: 25
----------------
30
==== Removing next the Current
25 40 Current: 25
----------------
Now, do it again just to be sure you've got it!
>>> |
```

After trying this code by yourself, make some changes.

Basic Data Structures

To structure data we create an abstraction, a different, usually more restrictive, way to access it.

We structure data in order to better grasp it.

There are implementation choices, but according with the usage we define what can be done or not with a data structure. As such, the definition of what can be done may guide us to choose an implementation.

- Arrays;
- Linked Lists;
- **Other Structures**.

MERRIMACK COLLEGE

Other Structures - Stacks and Queues

When using specific data structures, as queues and stacks, it may be more interesting to use an array, or a linked list implementation.

Queues:

- a list in which elements can only be inserted by one end, and the elements removed can only be removed by the other end of the list;

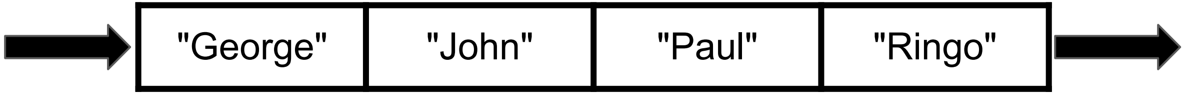
Stacks:

- a list in which elements can only be inserted and removed by one end of the list.

Lists, in CS, are any form of organized content that can be accessed in an orderly fashion, but not necessarily by a simple indexing procedure.

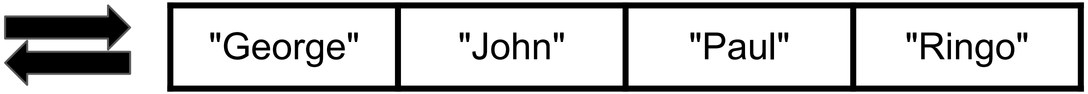
Other Structures - Stacks and Queues

- **Queues** - a special kind of list in which elements can only be inserted by one end, and the elements removed can only be removed by the other end of the list - FIFO



"George"	"John"	"Paul"	"Ringo"

- **Stacks** - a special kind of list in which elements can only be inserted and removed by one end of the list - LIFO

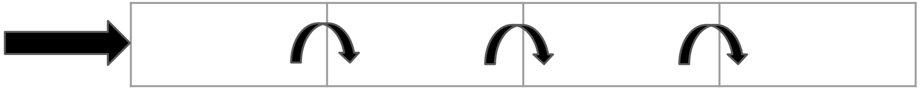


"George"	"John"	"Paul"	"Ringo"

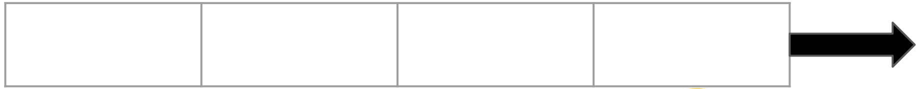
Text: Difference Between Stack and Queue Data Structures.

Stacks and Queues - with Arrays

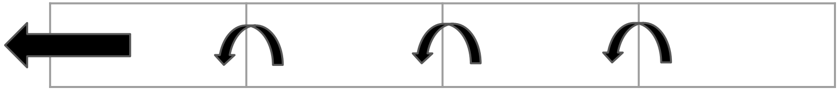
- For an **Array**
  - Inserting at the beginning ⚠



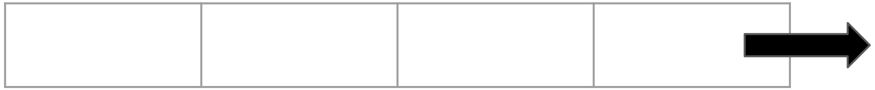
  - Inserting at the end ✓



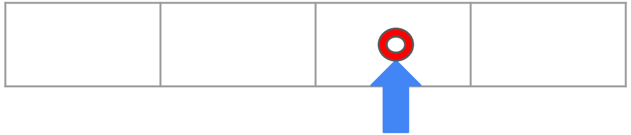
  - Removing from the beginning ⚠



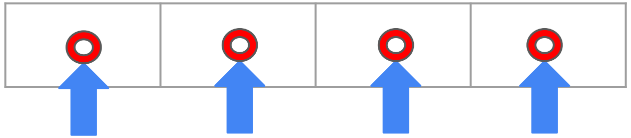
  - Removing from the end ✓



  - Accessing an element ✓



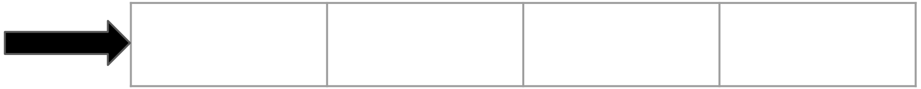
  - Accessing all elements ✓



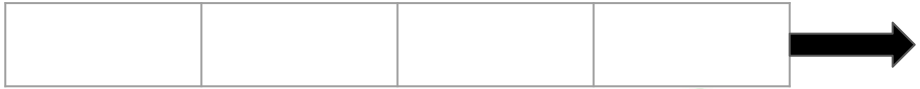
Arrays are bad handling the beginning of the list.

Stacks and Queues - with Arrays

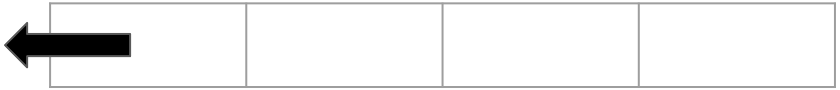
- For an **LinkedList**
  - Inserting at the beginning ✓



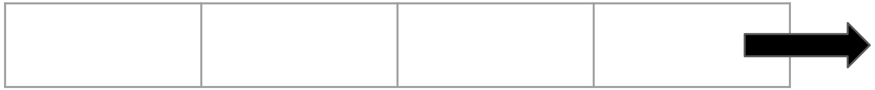
  - Inserting at the end ✓



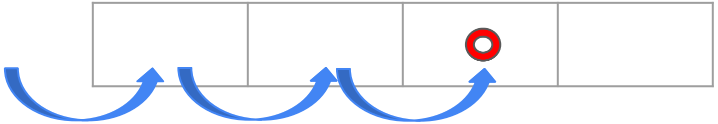
  - Removing from the beginning ✓



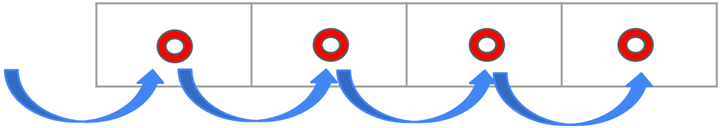
  - Removing from the end ✓



  - Accessing an element !



  - Accessing all elements !



MERRIMACK COLLEGE

Linked Lists are bad accessing elements freely.

Other Structures - Stacks and Queues

When needing a queue where access of the elements is frequently needed:

- Maybe arrays is an interesting implementation, because:
  - Arrays are better accessing elements freely;

When needing a queue where no access of the elements is needed:

- Surely linked lists is an interesting implementation, because:
  - Linked Lists are good accessing both ends of the list;

When needing a stack where access of the elements is frequently needed:

- Surely arrays is an interesting implementation, because:
  - Linked Lists are bad accessing elements freely;

When needing a stack where no access of the elements is needed:

- Maybe linked lists is an interesting implementation, because:
  - Arrays and Linked Lists are good accessing only one end of the list.

It depends...

Other Structures - Trees and Graphs

Trees are frequently implemented similarly to linked lists:

- Nodes with data and as many pointers as possible children of each node;
    - check out the lecture on trees from CSC6003.

Graphs are frequently implemented using:

- if using adjacency matrix, double dimension arrays (a matrix);
- if using adjacency list, either and array or linked lists;
    - check out the lecture on graphs from CSC6003.

According to the processing needs, the choice of implementation may play a very large role about the code efficiency.

It is not a bad idea to revise trees and graphs for some algorithms we will see in the next classes.

This Week's tasks

- In-class Exercise E#1
- Coding Project P#1
- Quiz Q#1

Tasks

- Use the LinkedList and Node class.
- Create a Python program to handle a LinkedList object.
- Quiz #1 about this week topics.

In-class Exercise - E#1

Use the LinkedList and Node class to manipulate a LinkedList doing the following operations:

- Include in this order the following numbers at the beginning of the list (they will be in reverse order because of it):
  - 76, 88, 11, 34, 56, 91;
- Print out the current status of the list;
- Push the Current to the third element of the list;
- Remove the next to the current element;
- Insert 23 next to the current element of the list;
- Print out the current status of the list.

You have to submit a ***.pdf*** file with your main code, plus the output for the executions above.

This task counts towards the In-class Exercises grade and the deadline is This Friday.

First Coding Project - P#1

- Create a program that reads a list of Integer numbers from a file named ***data.txt*** (create your own file with about 16 numbers - no repetitions and one number per line);
- Store those numbers into an array ***a*** and sort it: ***a.sort()***;
- Use the LinkedList and Node classes seen in class to store the ordered elements of ***a*** into a ***LinkedList*** structure ***L***;
- Ask the user an Integer value ***x***;
- Look for the position to insert ***x*** in ***L***;
    - If the value ***x*** is already in ***L***, remove it;
    - If it is not, insert ***x*** in the appropriated position so ***L*** remains sorted.

Your task:

- Go to Canvas, and submit your .py file within the deadline.

This assignment counts towards the Projects grade and the deadline is Next Monday.

First Coding Project - P#1

- This program must be your own, do not use someone else's code:
  - Make sure you fully understand the examples you use as learning source,
  - do not copy, neither retype seeing the source;
- To additional help:
  - Book a time slot with the student tutors to any help (see "Start Here" module),
  - Any specific questions about it, please bring to the Office hours meeting this Wednesday or contact me by email (put CSC6013 in the subject);
- This may be a challenging program, and it is intended to make sure you are mastering Python data structure manipulation;
- The documentation is not required, but include in your code your name.
- You can (and should) submit a .py file, no need to zip or rename the file extension.

This assignment counts towards the Projects grade and the deadline is Next Monday.

First Quiz - Q#1

- The first quiz in this course covers the topics of Week 1;
- The quiz will be available this Friday, and it is composed by 10 questions;
- The quiz should be taken on Canvas (Module 1), and it is not a timed quiz:
  - You can take as long as you want to answer it (a quiz taken in less than one hour is usually a too short time);
- The quiz is open book, open notes, and you can even use any language Interpreter to answer it;
- Yet, the quiz is evaluated and you are allowed to submit it only once.

Your task:

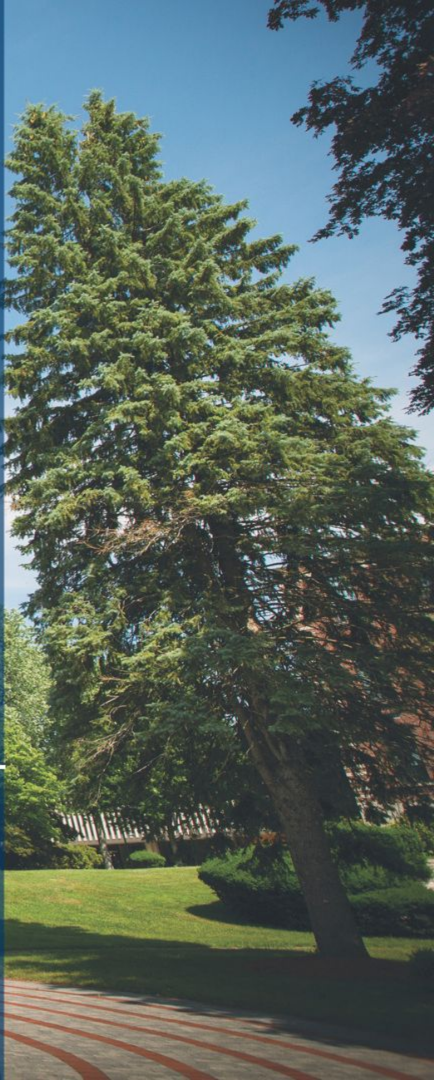
- Go to Canvas, answer the quiz and submit it within the deadline.

This quiz counts towards the Quizzes grade and the deadline is Next Tuesday.

Welcome to CSC 6013

- Read the syllabus
- Do In-class Exercise E#1 until Friday;
- Do Quiz Q#1 (available Friday) until next Monday;
- Do Coding Project P#1 until next Monday.

Next Week - Algorithms - asymptotic notations



MERRIMACK COLLEGE