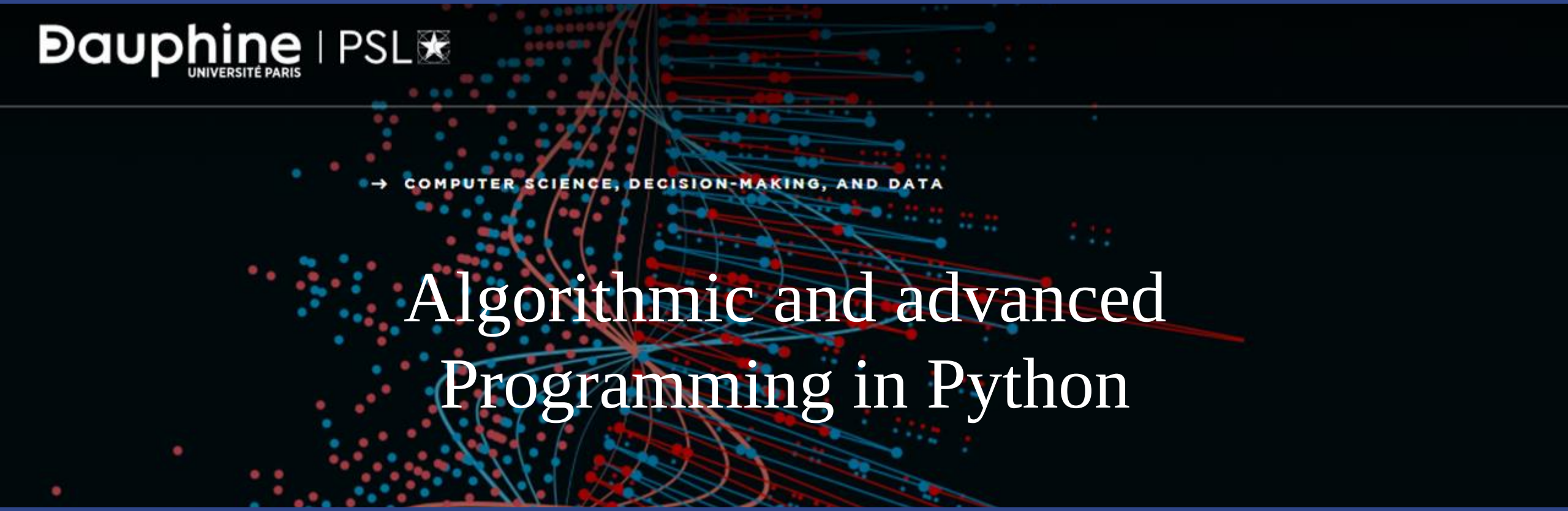


Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Remy Belmonte remy.belmonte@dauphine.eu

Masterclass 1

Outline

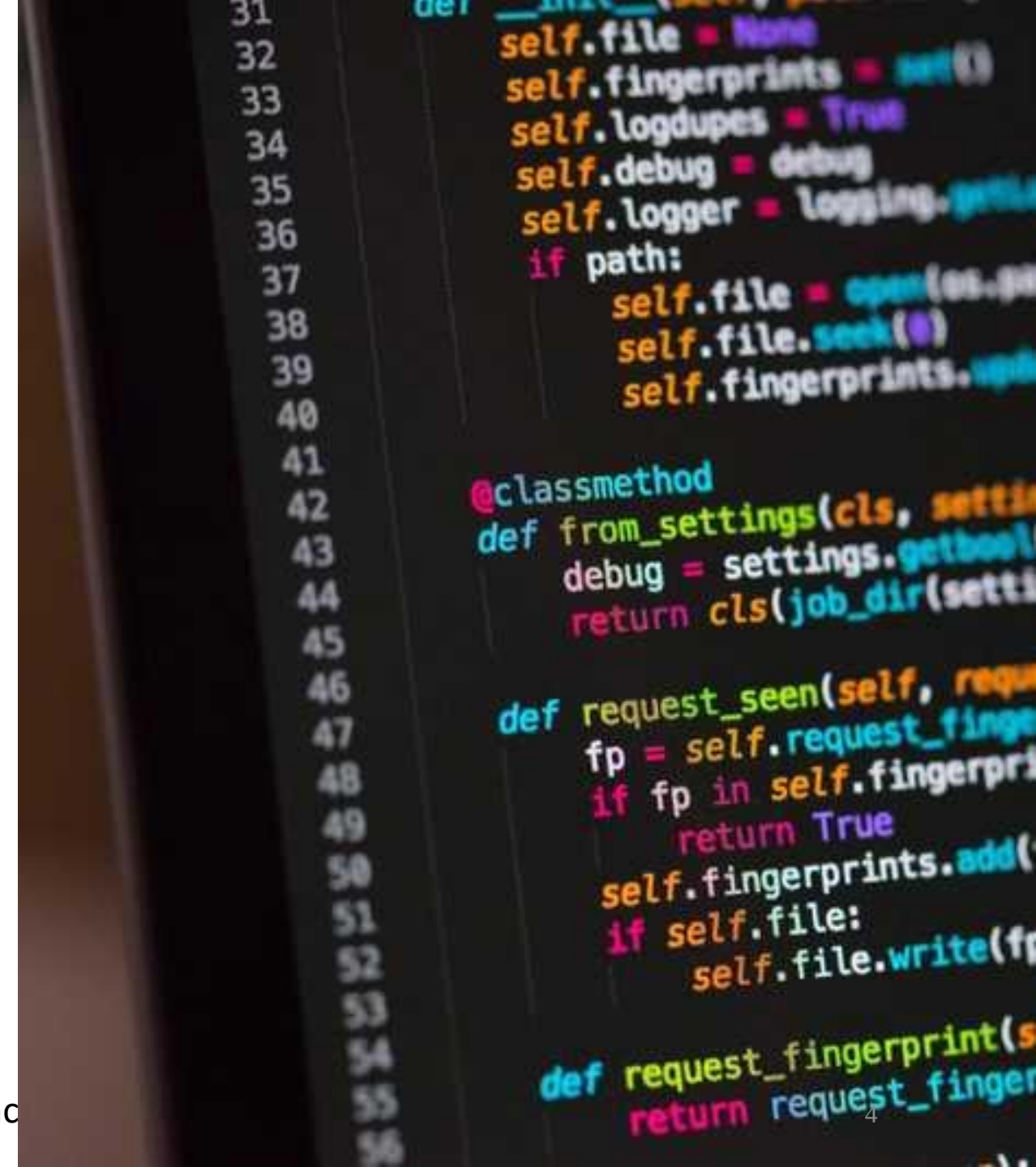
1. Introduction, motivations, Linked list
2. Stacks
3. Queues, Trees
4. Priority Queues and heaps, Disjoint Sets ADT
5. Graph algorithms,
6. Sorting
7. Searching, Selection algorithms
8. Symbol tables, Hashing
9. String algorithms, Algorithm designs
10. Greedy Algorithms, Divide and Conquer
11. Dynamic programming, Complexity

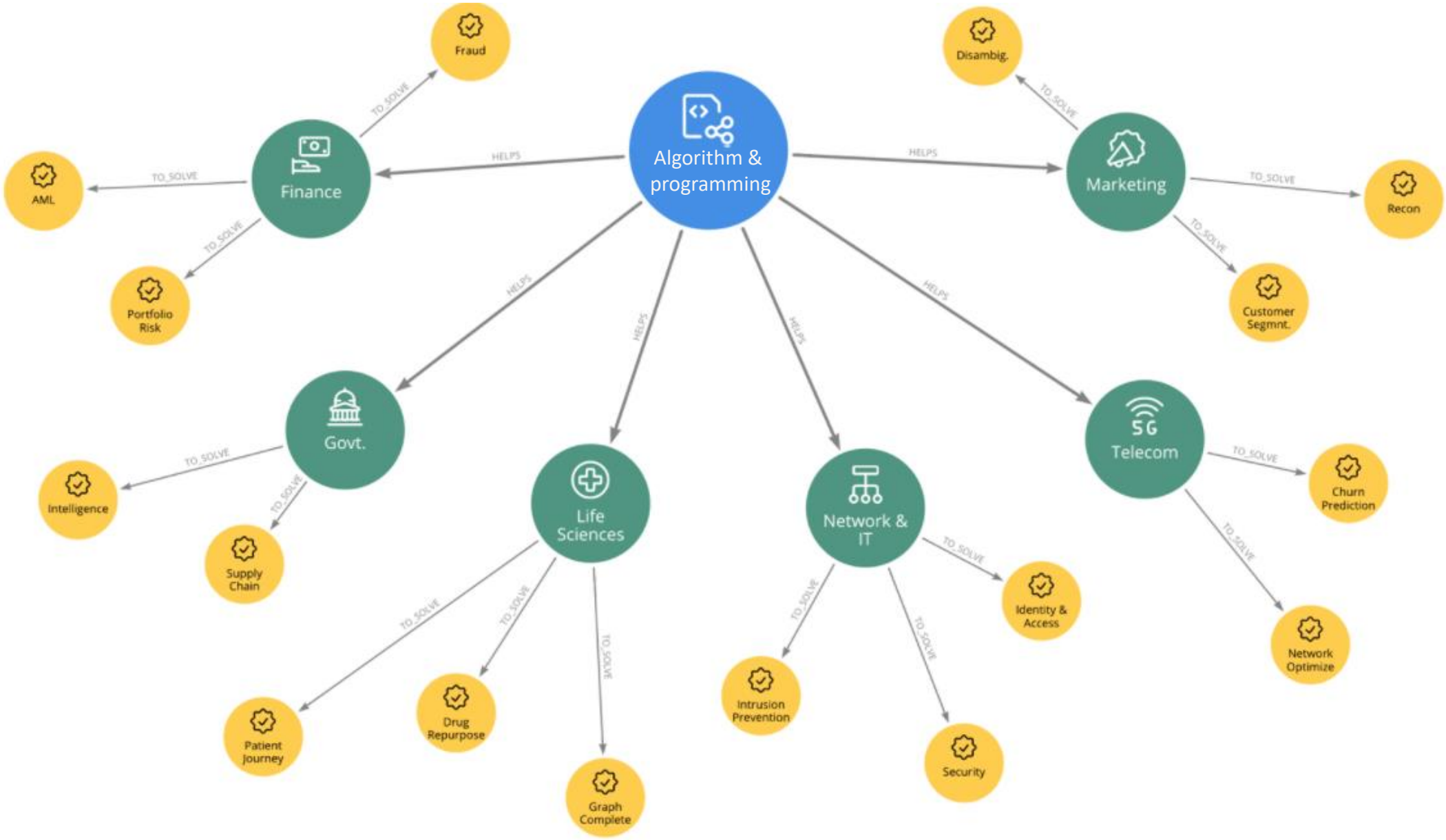
Evaluation principle

- 30% project to be finalized before 22 Dec 2021
- 70% written test to be done during week of January 10 2022

- For exceptional project, bonus of +20%

Algorithms and programming are at the center of all software applications, all fields combined




Algorithm & programming
HELPS
Finance
TO_SOLVE
Fraud
AML
Portfolio Risk
Marketing
Disambig.
Recon
Customer Segmnt.
Govt.
Intelligence
Supply Chain
Life Sciences
Patient Journey
Drug Repurpose
Graph Complete
Network & IT
Intrusion Prevention
Identity & Access
Security
5G Telecom
Churn Prediction
Network Optimize

Prerequisite

- Know how to write a simple algorithm in algorithmic terms:
  - handle Boolean, integer, real, character variables
  - handle arrays and character strings
  - know the control structures (tests, loops, ...)
  - know how to divide a program into functions and procedures
  - be familiar with the organization of memory

- Know how to implement all this in Python

What is a program?

- A program contains data, variables and instructions

- Data structure should be adapted to the goal
  - System defined data types: in python, additional complexity as this is a language with no types -> so implicit typing
  - User defined data types (classes in python) or list
  - Abstract data types (ADTs)

- Instructions are the core of the algorithm
  - They are structured by the algorithm
  - They describe the actions to do on the data to process them and produce an output

What is an algorithm?

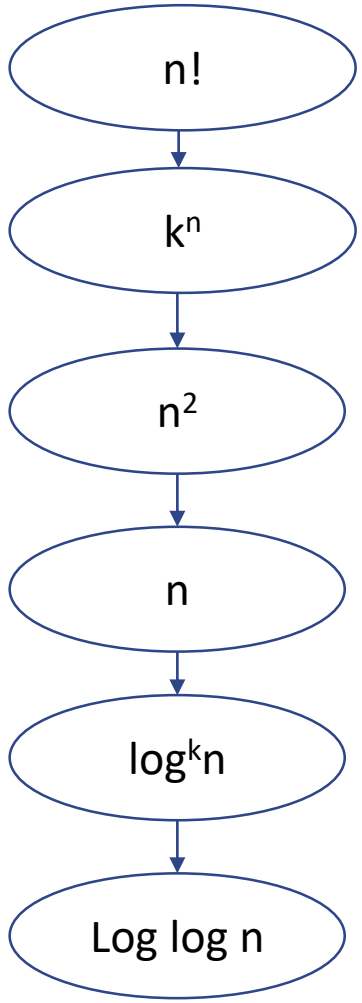
- Assume you want to prepare an omelette. What do you need?
    1. Get a frying pan
    2. Get the oil
        a) Do we have the oil?
            i. If yes, put in the pan
            ii. If no, do we want to buy the oil?
                i. If yes, then go and buy
                ii. If no, terminate
    3. Turn on the stove, etc…

-> So an algorithm is the step by step instructions to solve a given problem

Goal of the analysis of Algorithms

- Rate of growth of an algorithm:

Time complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
Log n	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
N log n	Linear log	Sorting n items by divide and conquer
n^2	Quadratic	Shortest path between 2 nodes in a graph
n^3	Cubic	Matrix multiplication
k^n	Exponential	Towers of Hanoi problem



Type of analysis

- Scenario analysis:
  - Worst case
  - Best case
  - Average case

- Asymptotic notation
  - O(n), … $O(n^2)$, etc…

Golden Rules for complexity

There are some general rules to help us determine the running time of an algorithm.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
# executes n times
for i in range(0,n):
    print 'Current Number :', i    #constant time
```

Total time = a constant $c \times n = cn = O(n)$.

2) **Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
# outer loop executed n times
for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j) #constant time
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

Rules

3) **Consecutive statements:** Add the time complexities of each statement.

```
n = 100
# executes n times
for i in range(0,n):
    print 'Current Number :', i                    #constant time
# outer loop executed n times
for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j)  #constant time
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
if n == 1:                          #constant time
    print "Wrong Value"
    print n
else:
    for i in range(0,n):            #n times
        print 'Current Number :', i #constant time
```

Total time = $c_0 + c_1 * n = O(n)$.

Logarithmic complexity

5) **Logarithmic complexity:** An algorithm is O($logn$) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

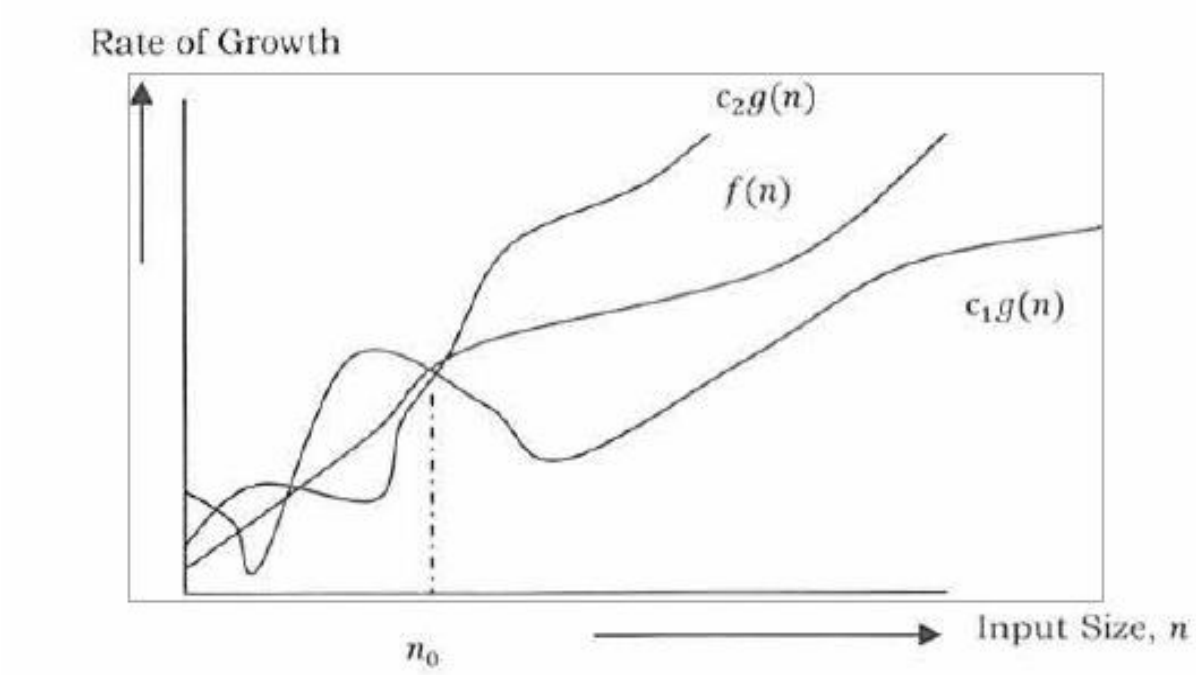
```
def Logarithms(n):
 i = 1
 while i <= n:
          i= i * 2
          print i
Logarithms(100)
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4,8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^k = n$ and we come out of loop. Taking logarithm on both sides, gives

$log(2^k) = logn$
$klog2 = logn$
$k = logn$ //if we assume base-2

Total time = O($logn$).

Omega Ω Notation



This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper bound (O) and lower bound (Ω) give the same result, then the Θ notation will also have the same rate of growth. As an example, let us assume that $f(n) = 10n + n$ is the expression. Then, its tight upper bound $g(n)$ is $O(n)$. The rate of growth in the best case is $g(n) = O(n)$.

Example

Θ Examples

Example 1 Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$, for all, $n \geq 1$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 1$

Example 2 Prove $n \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$

$\therefore n \neq \Theta(n^2)$

Example 3 Prove $6n^3 \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Example 4 Prove $n \neq \Theta(logn)$

Solution: $c_1 logn \leq n \leq c_2 log n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$ – Impossible

Master theorem for divide and conquer

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form then we can directly give the answer without fully solving it.

If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, where $a \geq 1, b > 1, k \geq 0$ and p is a real number, then:

1) If $a > b^k$, then $T(n) = \Theta(n^{log_b^a})$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$
   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

Divide and conquer master theorem example

1.22 Divide and Conquer Master Theorem: Problems & Solutions

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T(n/2) + n^2$

Solution: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T(n/2) + n^2$

Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 log n)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + n^2$

Solution: $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$

Solution: $T(n) = 2^n T(n/2) + n^n \Rightarrow$ Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$

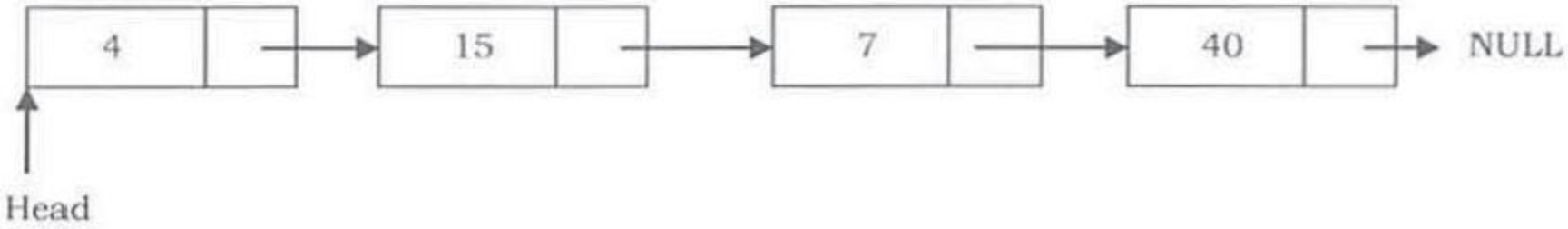
Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Linked list

3.1 What is a Linked List?

A linked list is a data structure used for storing collections of data. A linked list has the following properties.

- Successive elements are connected by pointers
- The last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)
- Does not waste memory space (but takes some extra memory for pointers)



Why linked lists?

- Difference with array

One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in constant time by using the index of the particular element as the subscript.

Why Constant Time for Accessing Array Elements?

To access an array element, the address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address.

This process takes one multiplication and one addition. Since these two operations take constant time, we can say the array access can be performed in constant time.

Advantage of array

Advantages of Arrays

- Simple and easy to use
- Faster access to the elements (constant access)

Disadvantages of Arrays

- **Fixed size:** The size of the array is static (specify the array size before using it).
- **One block allocation**: To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
- **Complex position-based insertion**: To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.

Pro and cons for linked list

Advantages of Linked Lists

Linked lists have both advantages and disadvantages. The advantage of linked lists is that they can be *expanded* in constant time. To create an array, we must allocate memory for a certain number of elements. To add more elements to the array, we must create a new array and copy the old array into the new array. This can take a lot of time.

We can prevent this by allocating lots of space initially but then we might allocate more than we need and waste memory. With a linked list, we can start with space for just one allocated element and *add* on new elements easily without the need to do any copying and reallocating.

Cons

Issues with Linked Lists (Disadvantages)

There are a number of issues with linked lists. The main disadvantage of linked lists is *access time* to individual elements. Array is random-access, which means it takes O(1) to access any element in the array. Linked lists take O(n) for access to an element in the list in the worst case. Another advantage of arrays in access time is *spacial locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

Although the dynamic allocation of storage is a great advantage, the *overhead* with storing and retrieving data can make a big difference. Sometimes linked lists are *hard to manipulate*. If the last item is deleted, the last but one must then have its pointer changed to hold a NULL reference. This requires that the list is traversed to find the last but one link, and its pointer set to a NULL reference.

Finally, linked lists waste memory in terms of extra reference points.

Comparison linked list, array and dynamic array

Parameter	Linked list	Array	Dynamic array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Insertion at ending	$O(n)$	$O(1)$, if array is not full	$O(1)$, if array is not full $O(n)$, if array is full
Deletion at ending	$O(n)$	$O(1)$	$O(n)$
Insertion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Deletion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Wasted space	$O(n)$	0	$O(n)$

In python

```
#Node of a Singly Linked List
class Node:
    #constructor
    def __init__(self):
        self.data = None
        self.next = None
     #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
```

Linked list insertion

Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

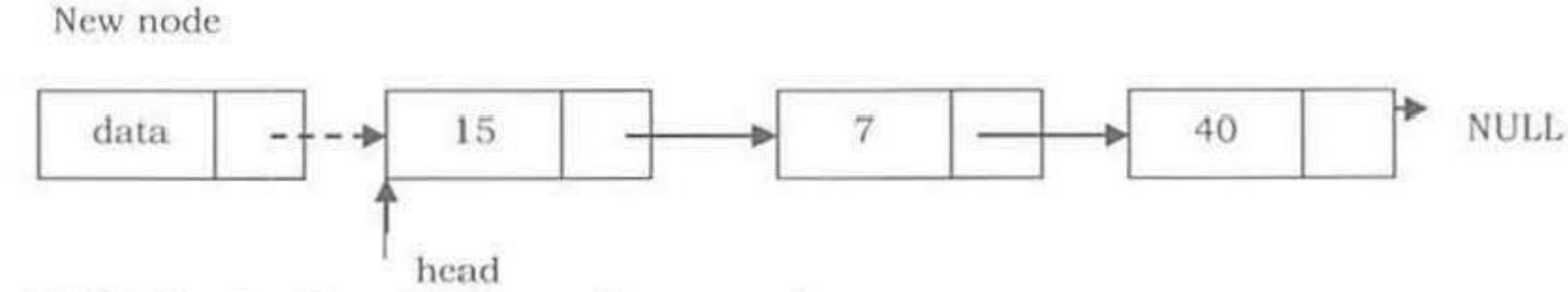
Note: To insert an element in the linked list at some position p, assume that after inserting the element the position of this new node is p.

Algorithm: insertion

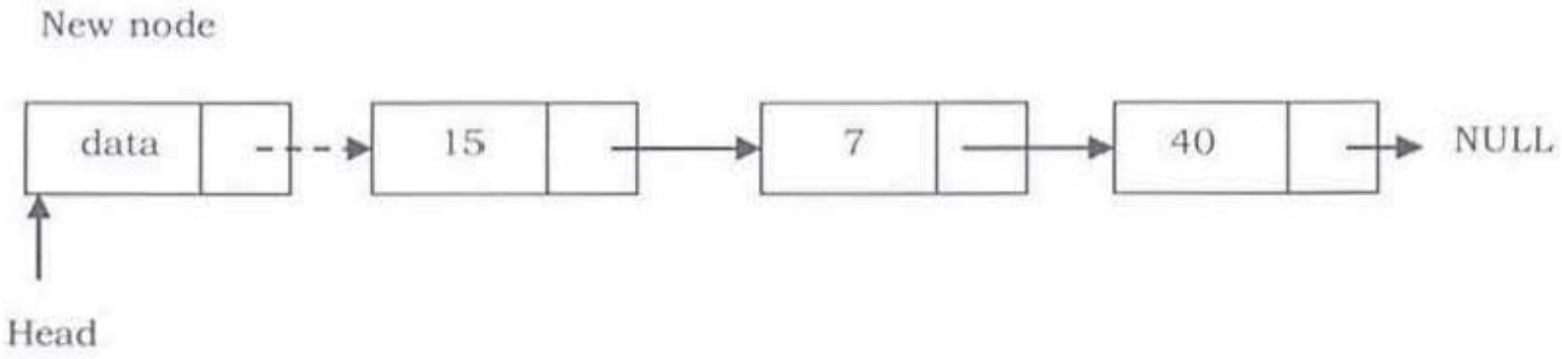
Inserting a Node in Singly Linked List at the Beginning

In this case, a new node is inserted before the current head node. *Only one next pointer* needs to be modified (new node's next pointer) and it can be done in two steps:

- Update the next pointer of new node, to point to the current head.



- Update head pointer to point to the new node.



Code

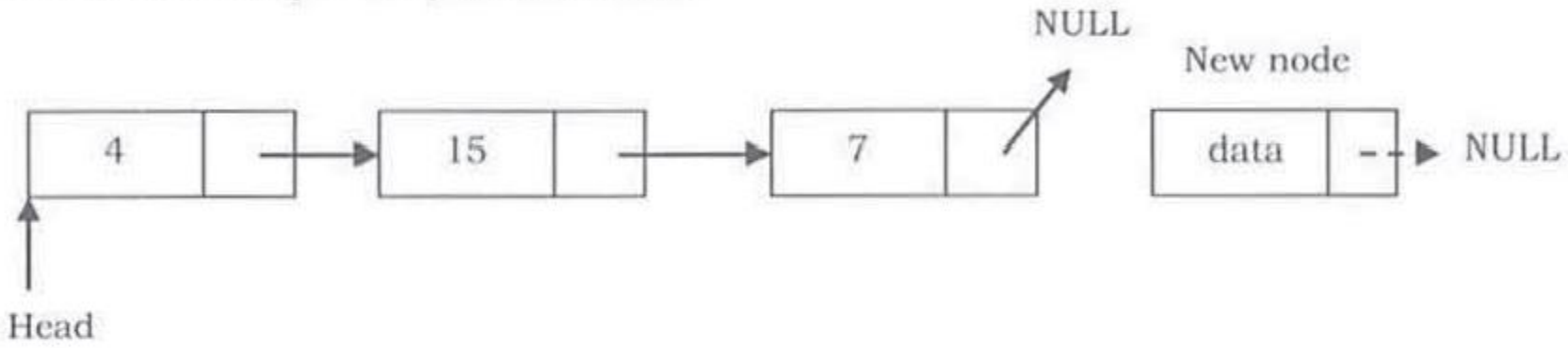
```
#method for inserting a new node at the beginning of the Linked List (at the head)
def insertAtBeginning(self,data):
    newNode = Node()
    newNode.setData(data)

    if self.length == 0:
        self.head = newNode
    else:
        newNode.setNext(self.head)
        self.head = newNode

    self.length += 1
```

At the end

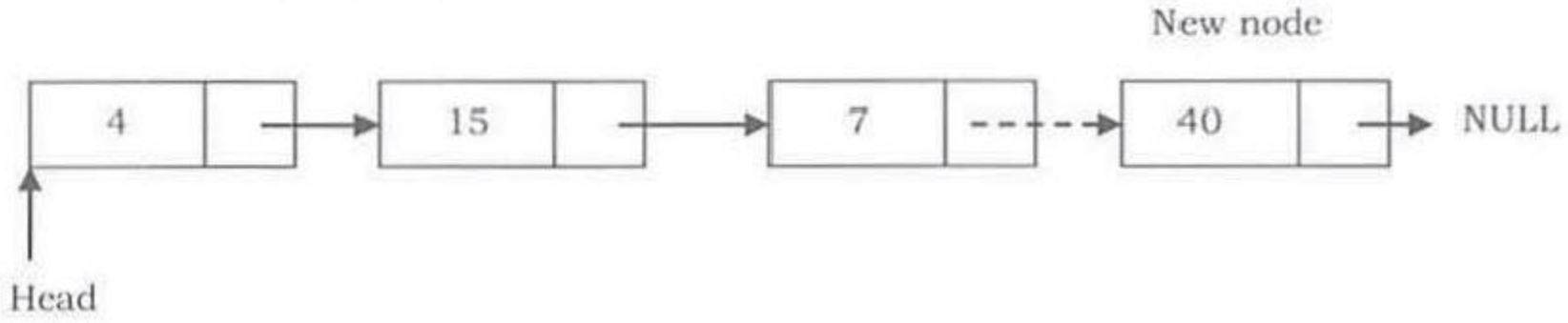
Inserting a Node in Singly Linked List at the Ending

In this case, we need to modify *two next pointers* (last nodes next pointer and new nodes next pointer).

- New nodes next pointer points to NULL.



- Last nodes next pointer points to the new node.



Code

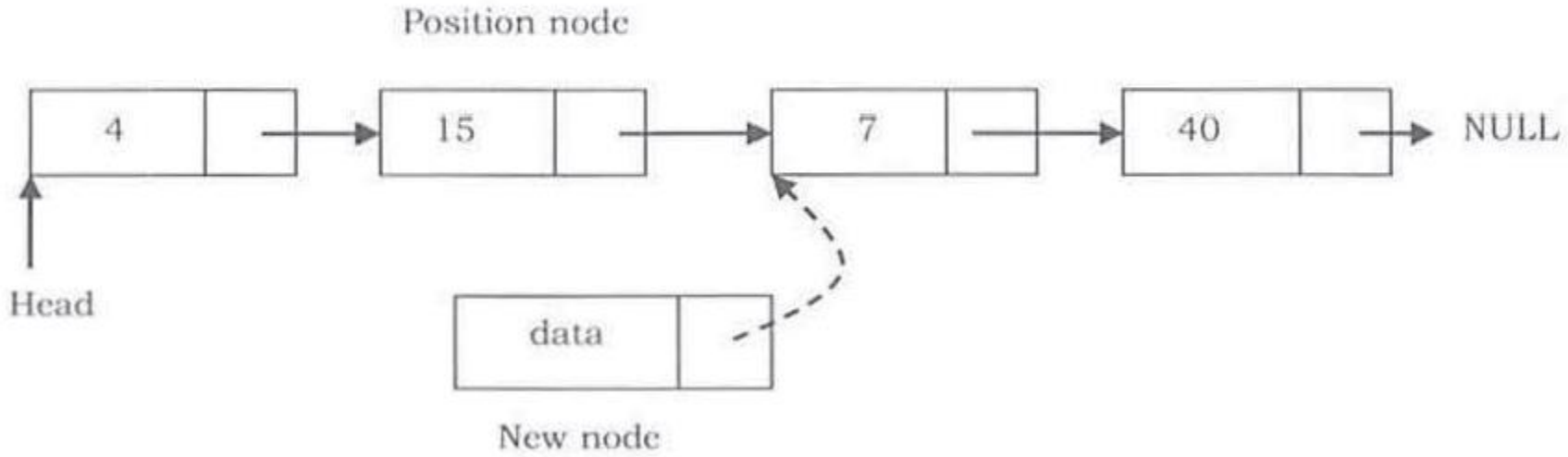
```
#method for inserting a new node at the end of a Linked List
def insertAtEnd(self,data):
    newNode = Node()
    newNode.setData(data)
    current = self.head
    while current.getNext() != None:
        current = current.getNext()
    current.setNext(newNode)
    self.length += 1
```

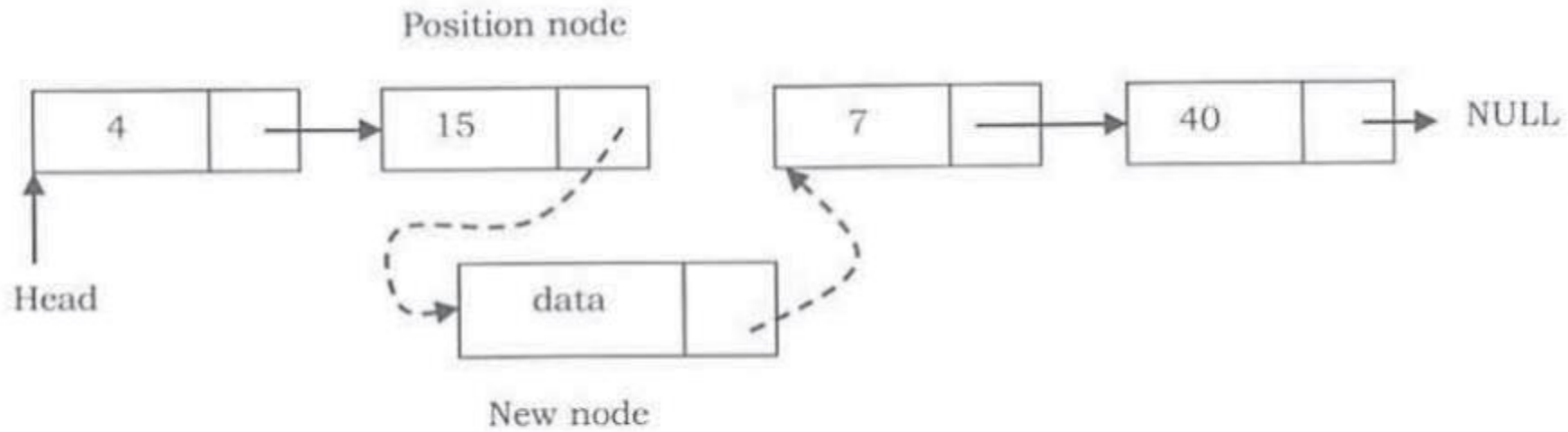
In the middle

Inserting a Node in Singly Linked List at the Middle

Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.

- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called *position* node. The new node points to the next node of the position where we want to add this node.





Let us write the code for all three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send a double pointer. The following code inserts a node in the singly linked list.

Code

```
#Method for inserting a new node at any position in a Linked List
def insertAtPos(self,pos,data):
 if pos > self.length or pos < 0:
    return None
 else:
    if pos == 0:
          self.insertAtBeg(data)
    else:
          if pos == self.length:
             self.insertAtEnd(data)
          else:
             newNode = Node()
             newNode.setData(data)
             count = 0
             current = self.head
             while count < pos-1:
                   count += 1
                   current = current.getNext()
             newNode.setNext(current.getNext())
             current.setNext(newNode)
             self.length += 1
```

Deleting in linked list

Singly Linked List Deletion

Similar to insertion, here we also have three cases.

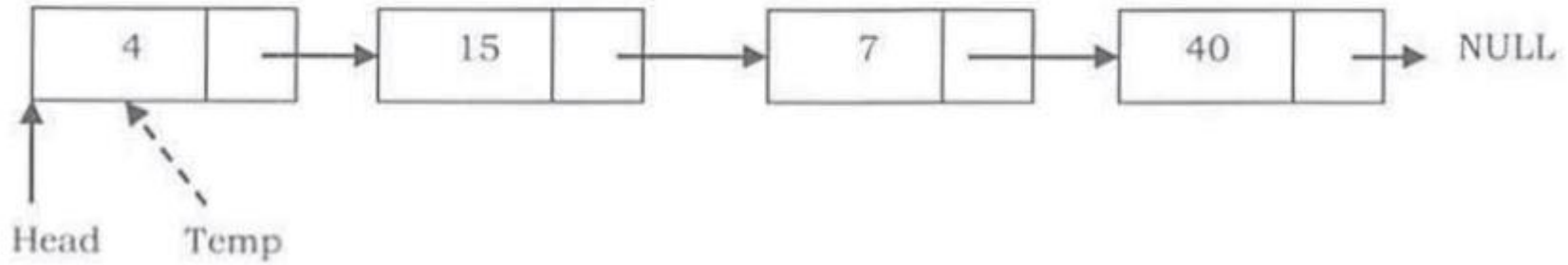
- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

Deleting the first node

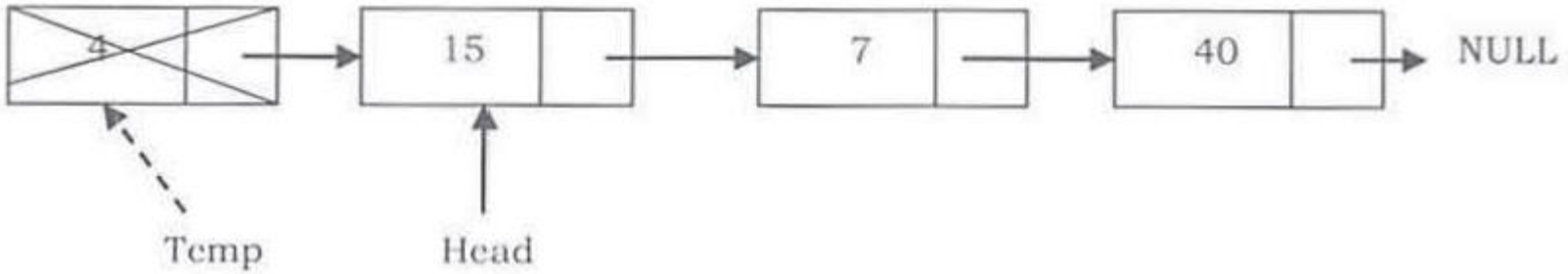
Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to the same node as that of head.



- Now, move the head nodes pointer to the next node and dispose of the temporary node.



Code

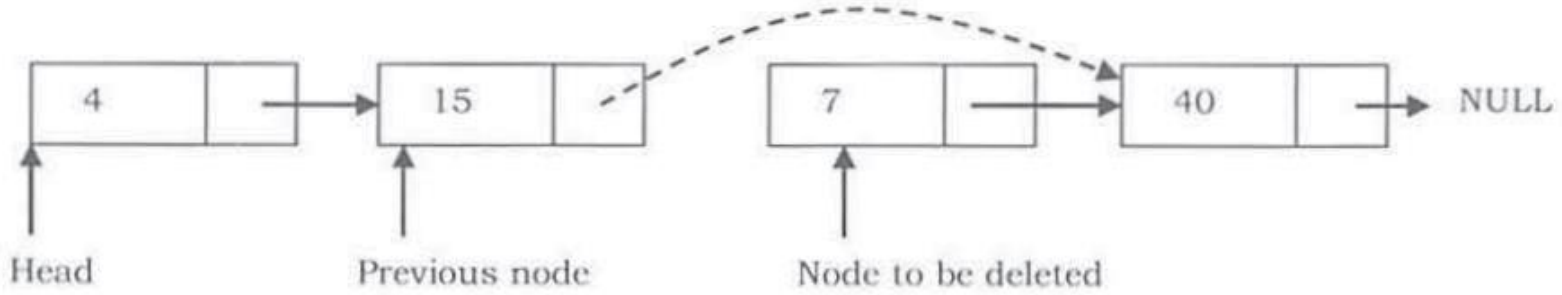
```
#method to delete the first node of the linked list
def deleteFromBeginning(self):
    if self.length == 0:
        print "The list is empty"
    else:
        self.head = self.head.getNext()
        self.length -= 1
```

Deleting an intermediate node in singly listed list

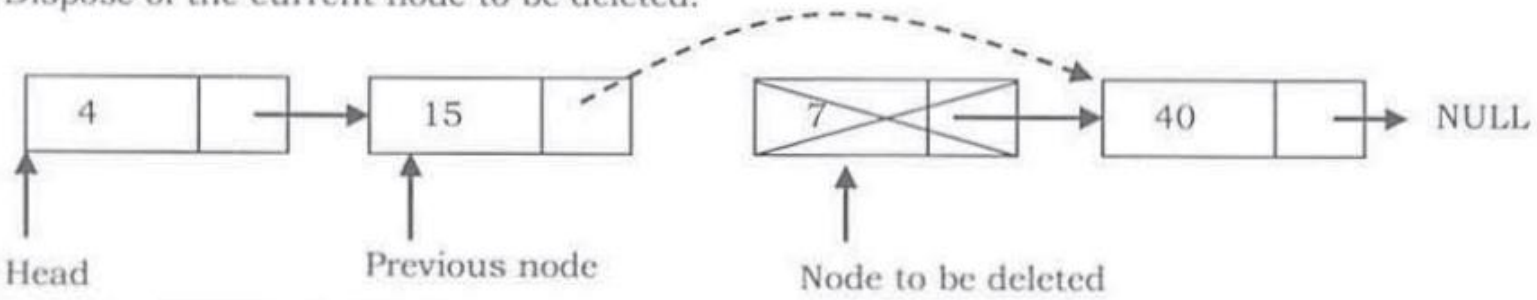
Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



- Dispose of the current node to be deleted.



Code

```
#Delete with node from linked list
def deleteFromLinkedListWithNode(self, node):
    if self.length == 0:
        raise ValueError("List is empty")
    else:
        current = self.head
        previous = None
        found = False
        while not found:
            if current == node:
                found = True
            elif current is None:
                raise ValueError("Node not in Linked List")
            else:
                previous = current
                current = current.getNext()
    if previous is None:
        self.head = current.getNext()
    else:
        previous.seNext(current.getNext())
    self.length -= 1
```

Code

```
def deleteValue(self,value):
        currentnode = self.head
        previousnode = self.head
        while currentnode.next != None or currentnode.value != value:
            if currentnode.value == value:
                previousnode.next = currentnode.next
                self.length -= 1
                return
            else:
                previousnode = currentnode
                currentnode = currentnode.next
        print "The value provided is not present"
```

Delete at position

```
#Method to delete a node at a particular position
def deleteAtPosition(self,pos):
        count = 0
        currentnode = self.head
        previousnode = self.head
        if pos > self.length or pos < 0:
            print "The position does not exist. Please enter a valid position"
        else:
            while currentnode.next != None or count < pos:
                count = count + 1
                if count == pos:
                    previousnode.next = currentnode.next
                    self.length -= 1
                    return
                else:
                    previousnode = currentnode
                    currentnode = currentnode.next
```

In Lab session

- You will play with the concepts and starts getting more and more familiar with how this works in real life
- This will be useful for your project

- Lab are done by Remy Belmonte