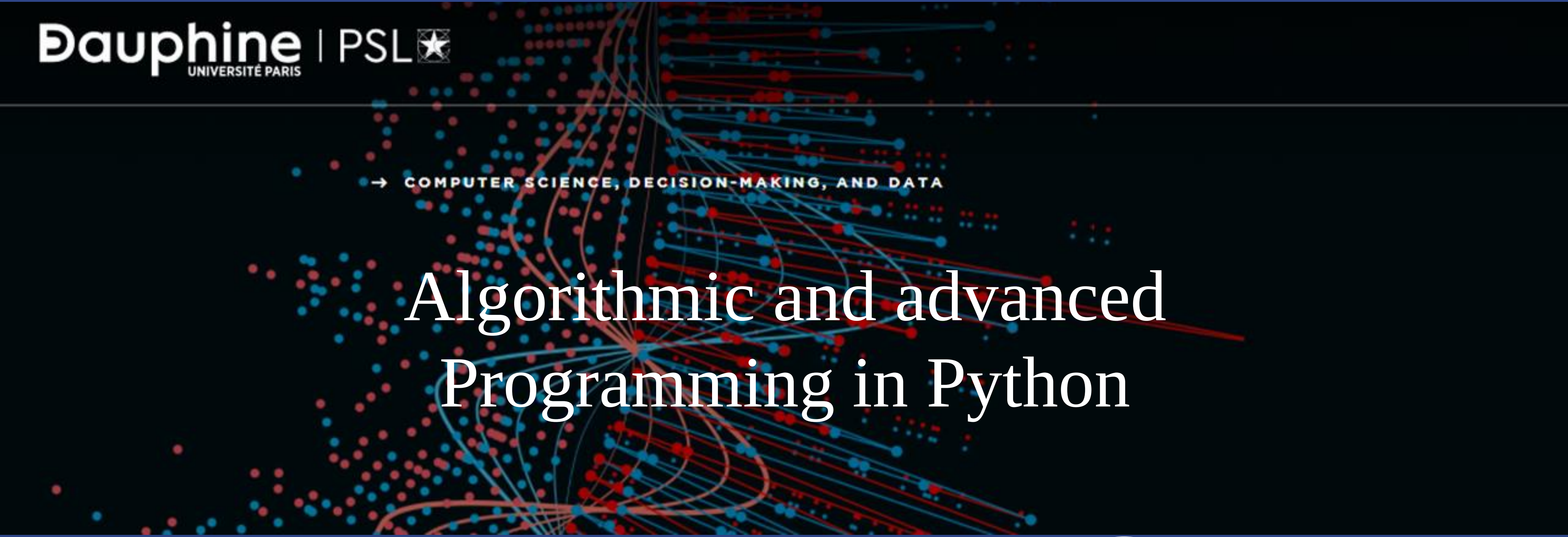

Dauphine UNIVERSITÉ PARIS | PSL
→ COMPUTER SCIENCE, DECISION-MAKING, AND DATA

Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Chien-Chung.Huang chien-chung.huang@ens.fr
Sofía Vázquez sofia.Vazquez@dauphine.eu

Master class 5

Outline

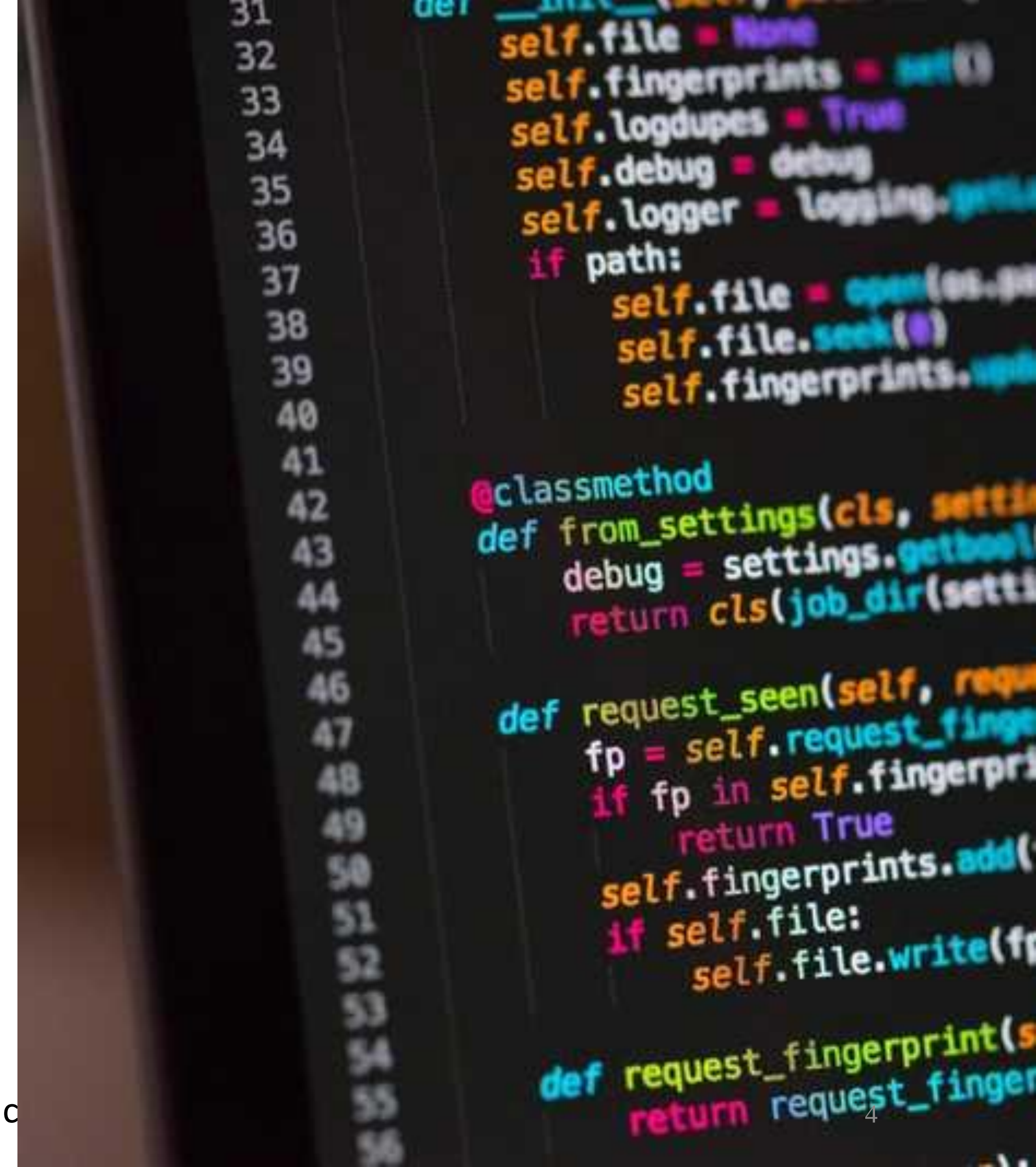
1. Introduction, motivations,
2. Linked list
3. Stacks
4. Queues, Trees

Evaluation principle

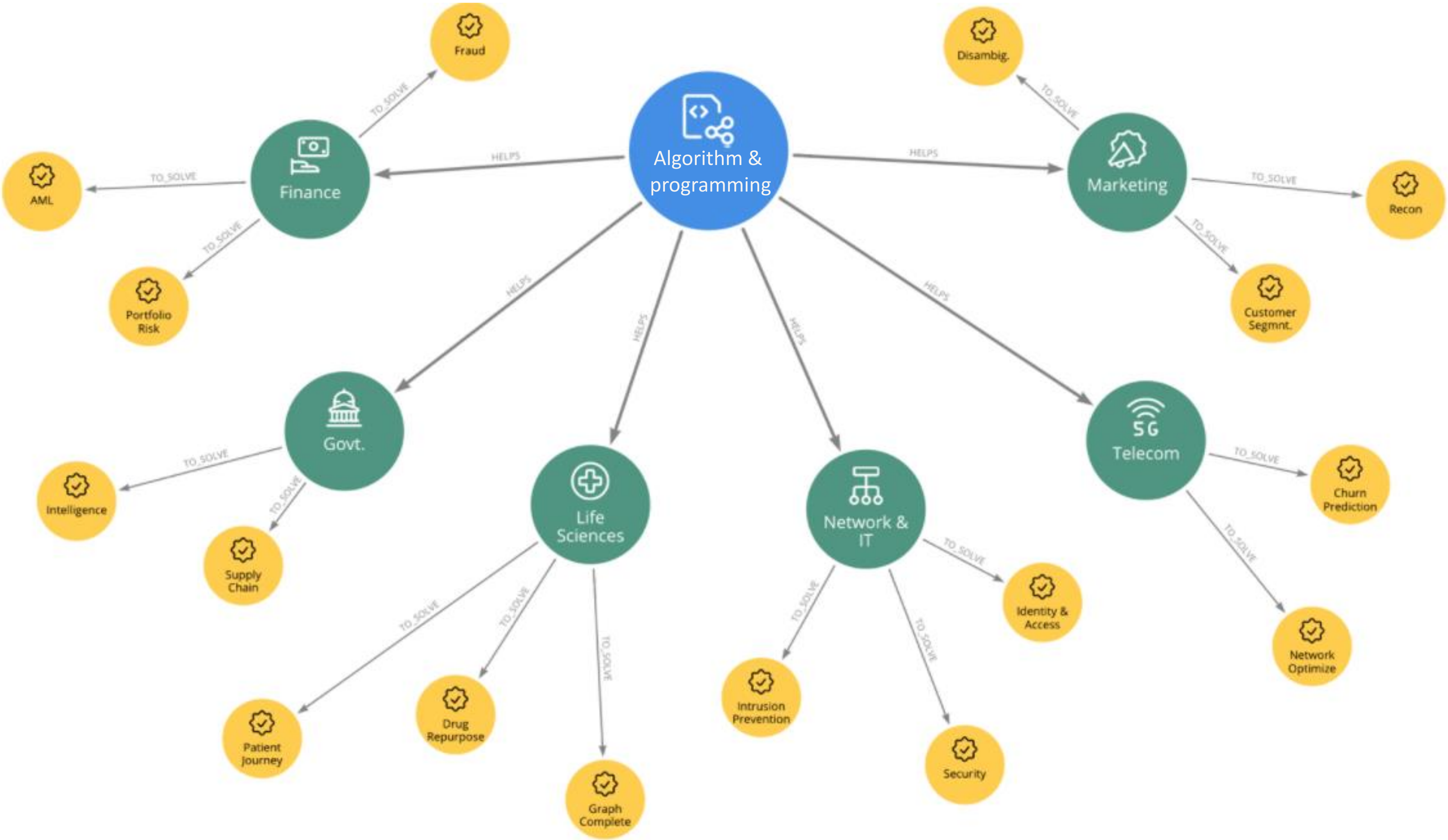
- 30% project to be finalized before 22 Dec 2021
- 70% written test to be done during week of January 10 2022

- For exceptional project, bonus of +20%

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



```
        self.file = None
        self.fingerprints = set()
        self.logdupes = True
        self.debug = debug
        self.logger = logging.
        if path:
            self.file = open(os.pa
            self.file.seek(0)
            self.fingerprints.upd

    @classmethod
    def from_settings(cls, setti
        debug = settings.getbool
        return cls(job_dir(setti

    def request_seen(self, requ
        fp = self.request_finge
        if fp in self.fingerpri
            return True
        self.fingerprints.add(
        if self.file:
            self.file.write(fp

    def request_fingerprint(s
        return request_finger
```


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

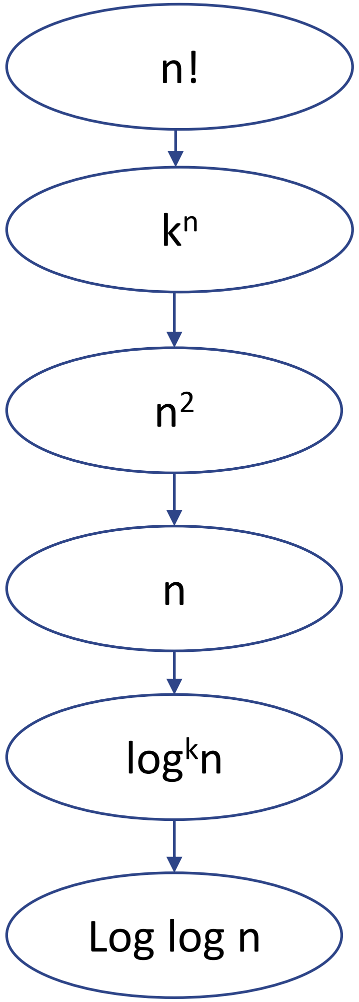
- Assume you want to prepare an omelette. What do you need?
    1. Get a frying pan
    2. Get the oil
        - a) Do we have the oil?
            - i. If yes, put in the pan
            - ii. If no, do we want to buy the oil?
                - i. If yes, then go and buy
                - ii. If no,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

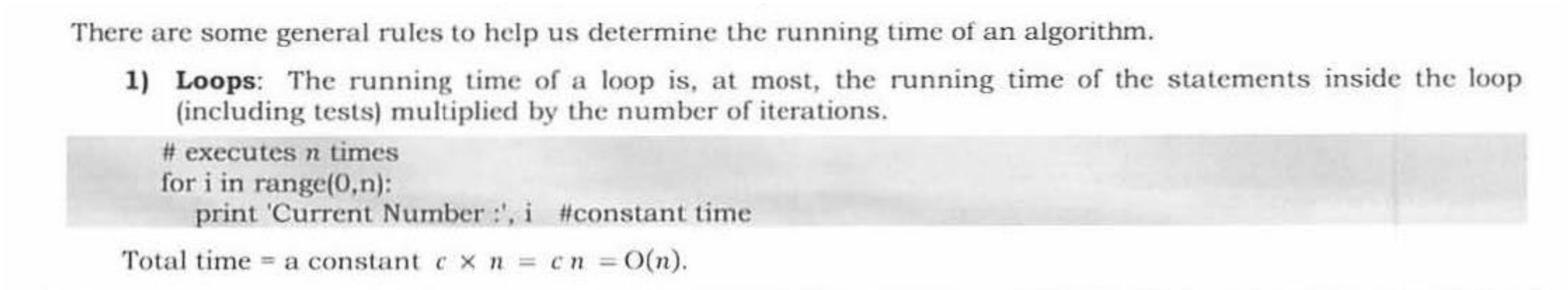


Type of analysis

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

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

Golden Rules for complexity

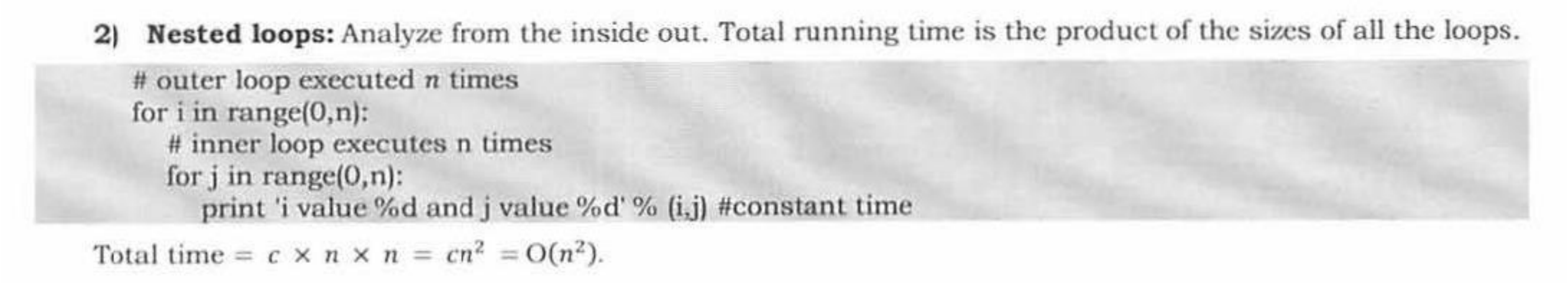


There are some general rules to help us determine the running time of an algorithm.

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 $c \times n = c\,n = O(n)$.

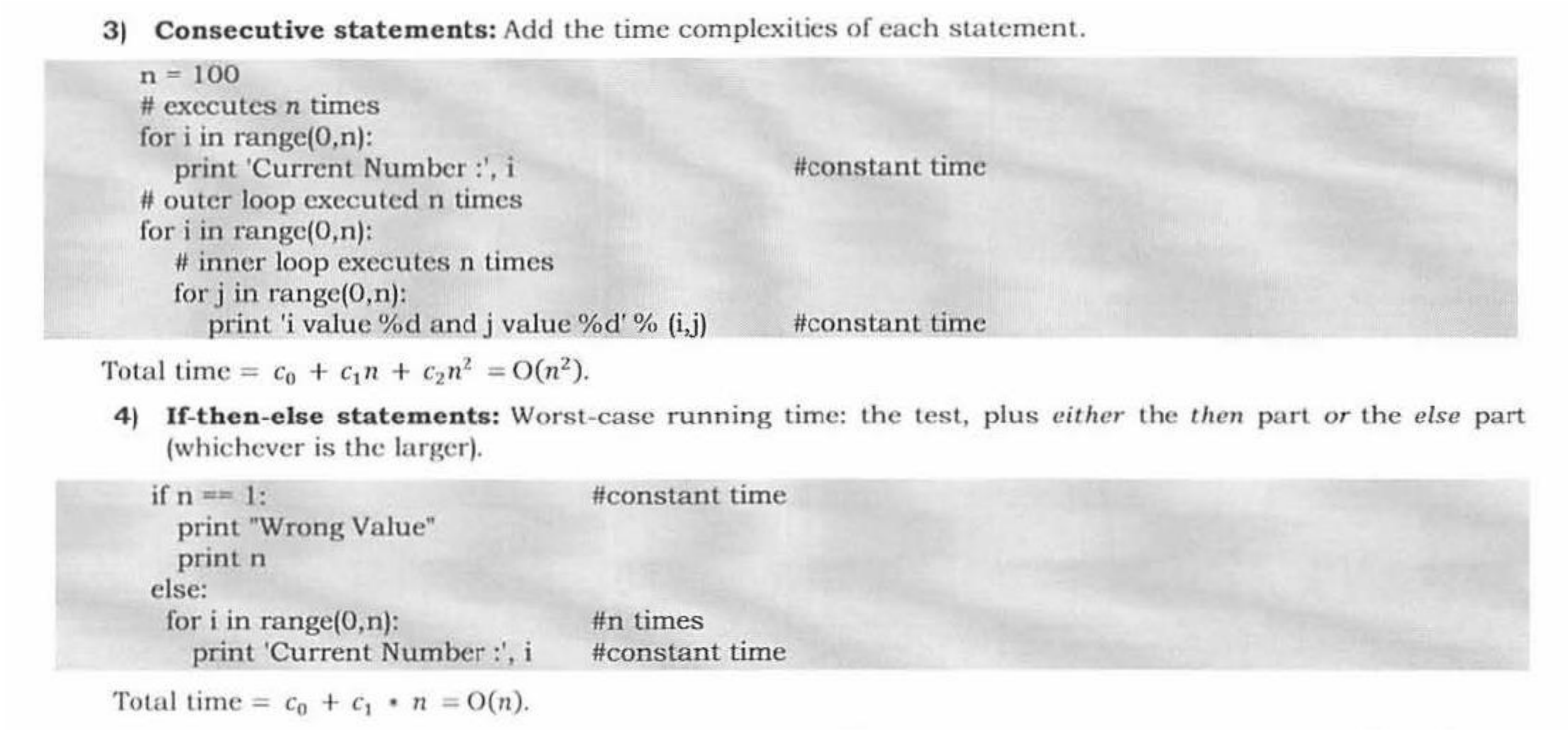


2) **Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
# 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
```

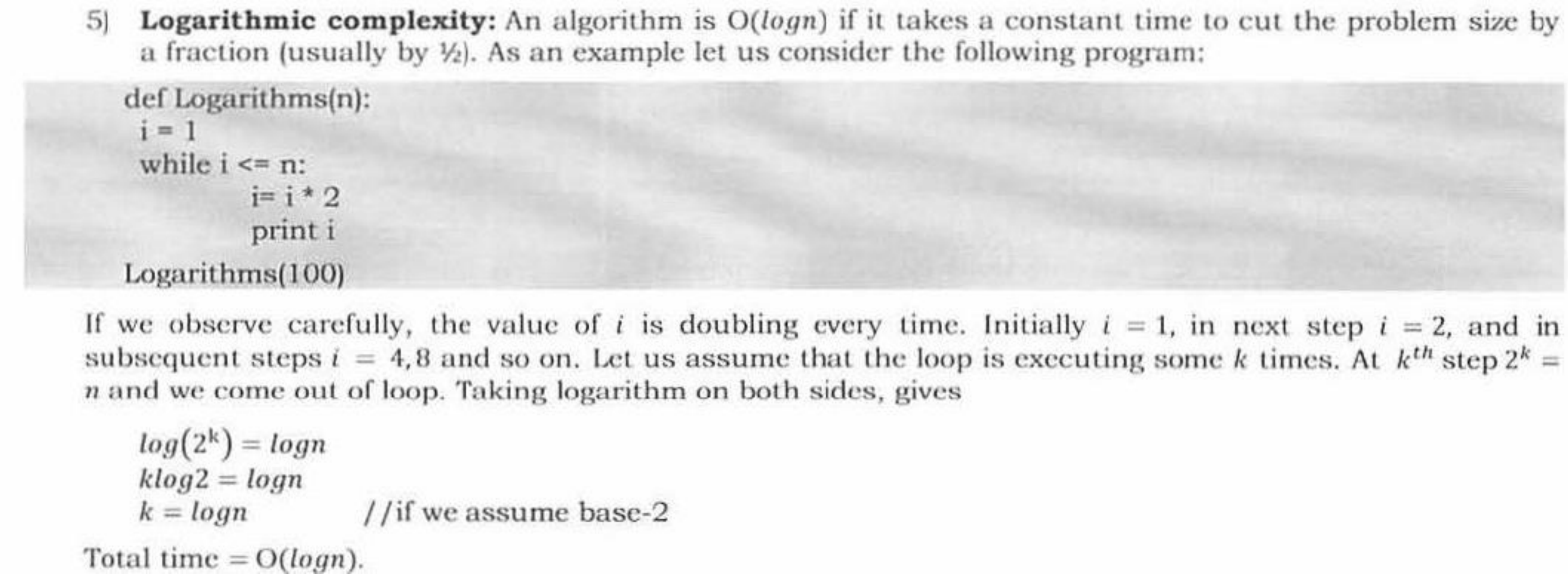
Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

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:

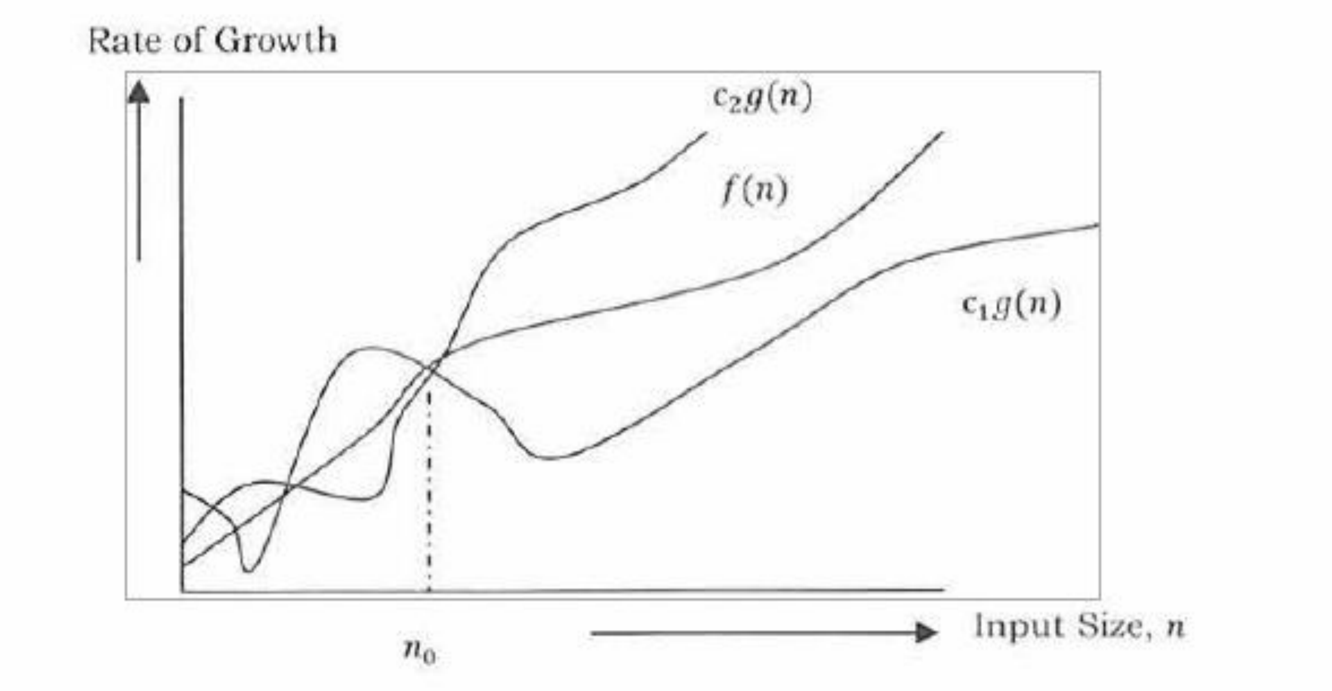
```
def Logarithms(n):
  i = 1
  while i <= n:
            i= i * 2
            print i
Logarithms(100)
```

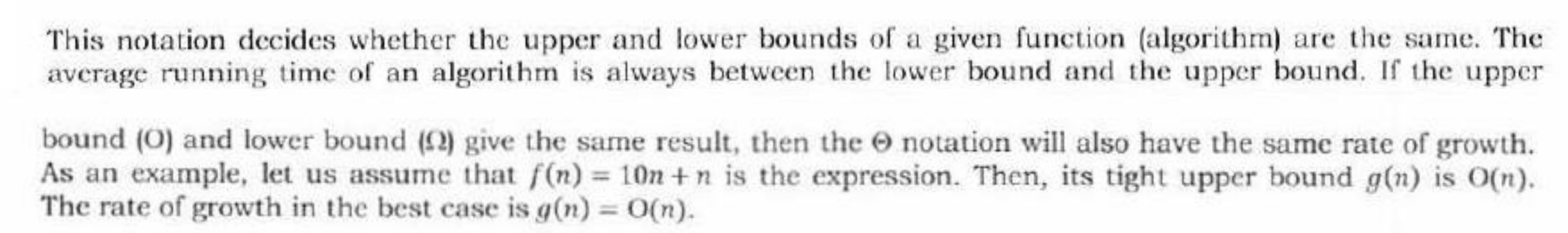
If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4, 8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^k = n$ and we come out of loop. Taking logarithm on both sides, gives

$$log(2^k) = logn$$
$$klog2 = logn$$
$$k = logn \qquad \text{//if we assume base-2}$$

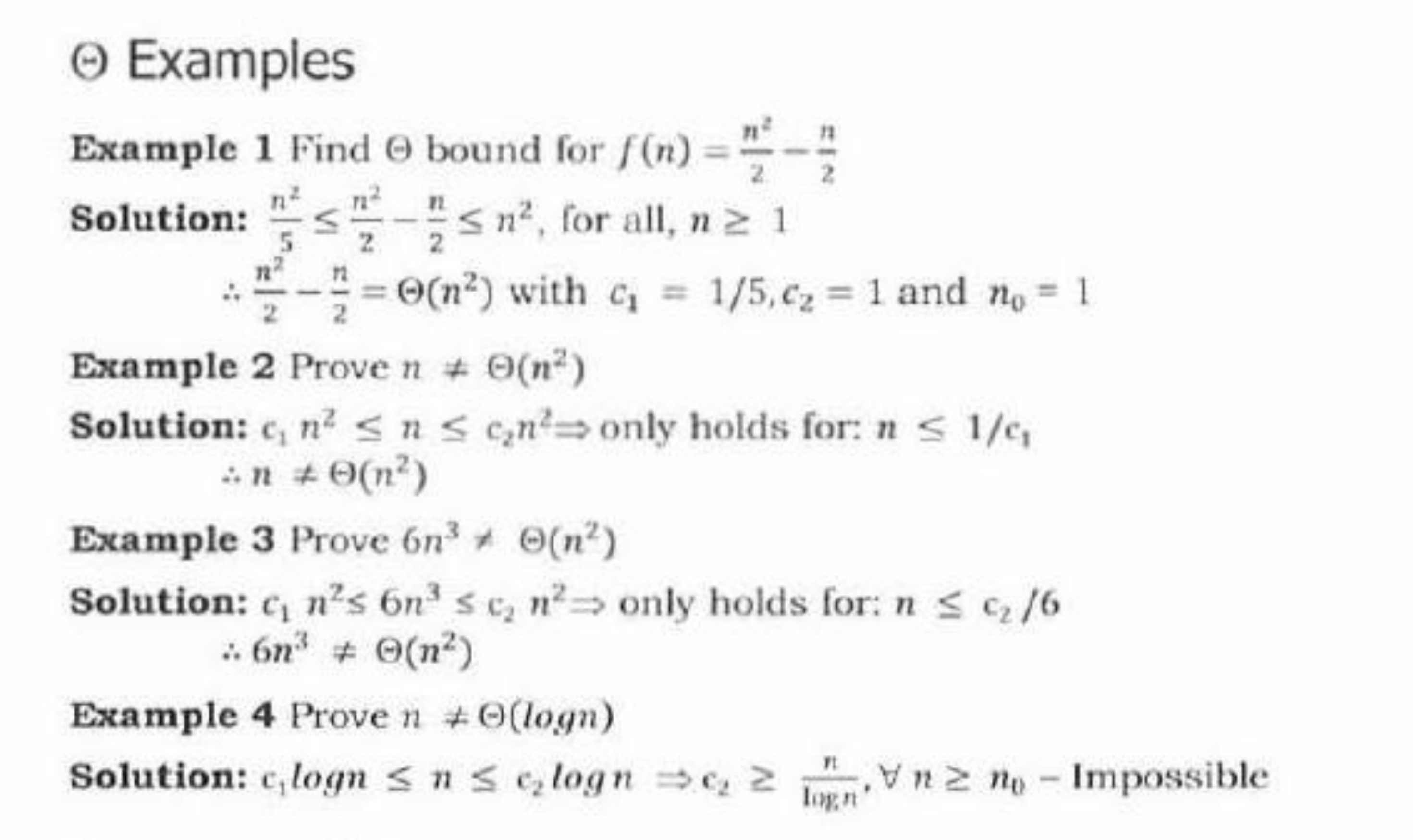
Total time = O(*logn*).

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

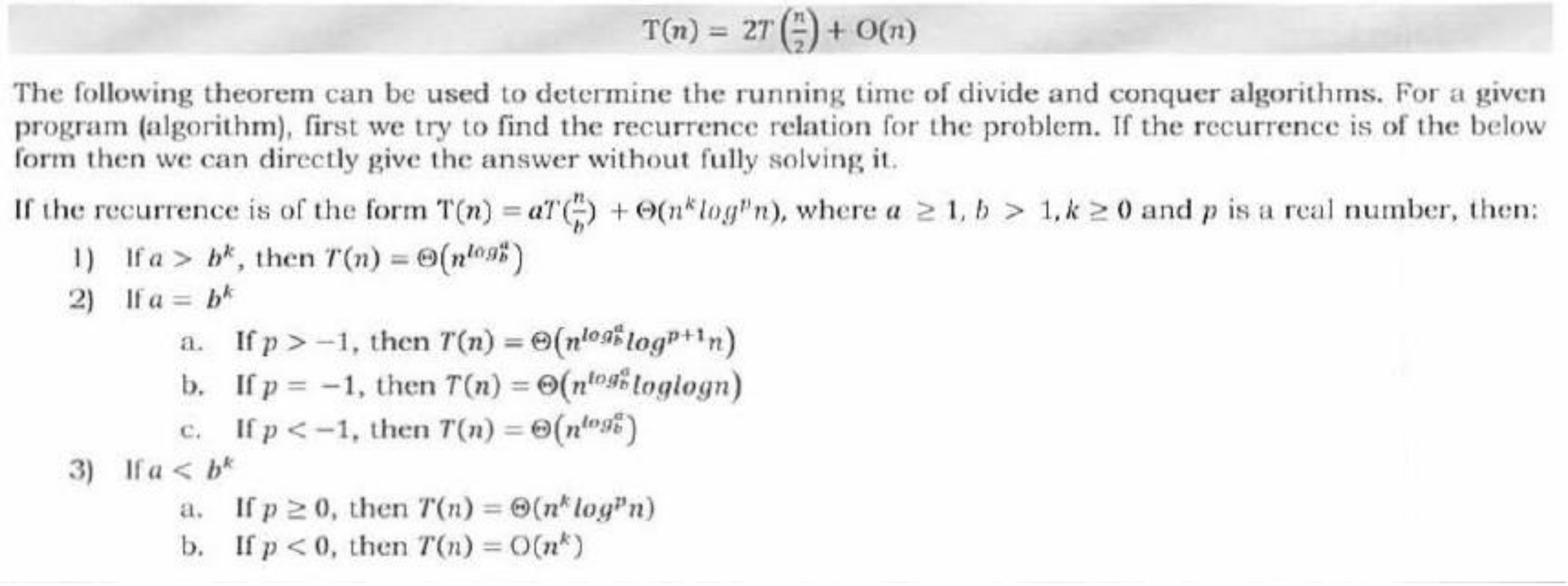
Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2/6$

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

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



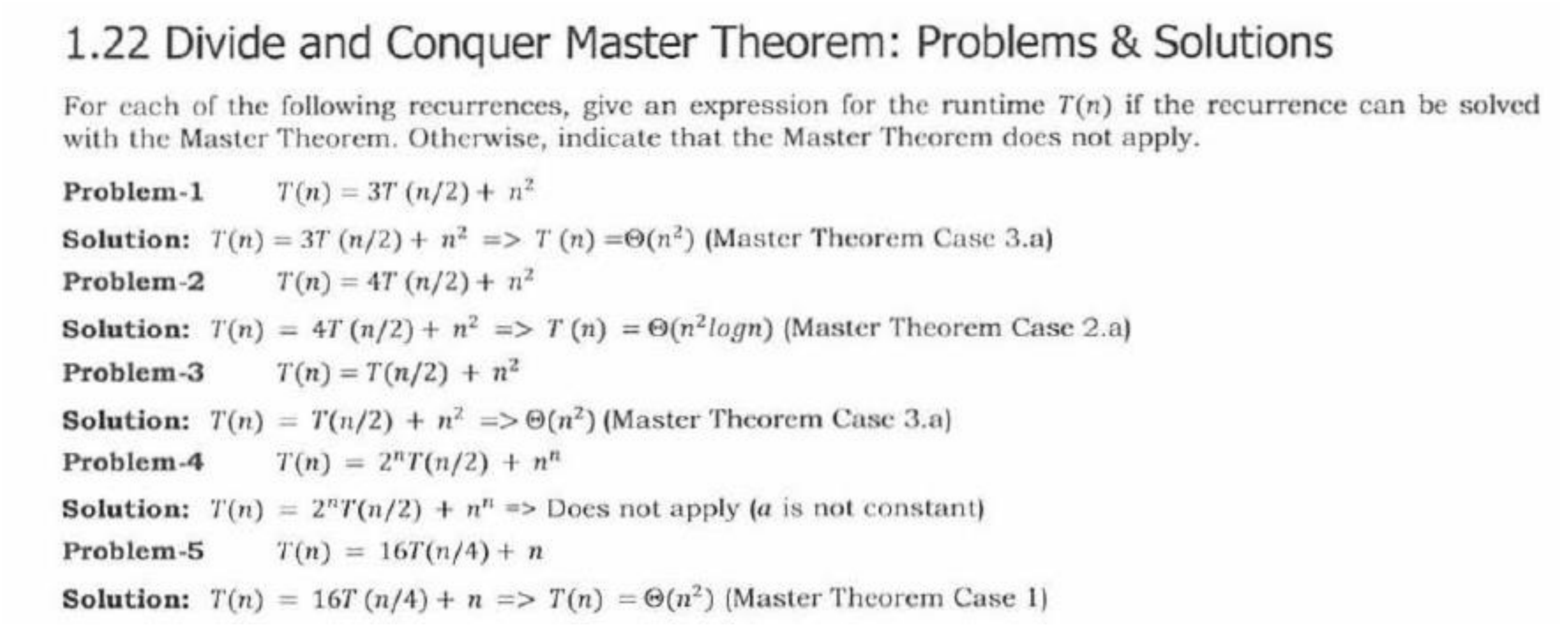
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

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: $T(n) = 4T\ (n/2) + n^2 \Rightarrow T\ (n) = \Theta(n^2 logn)$ (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)

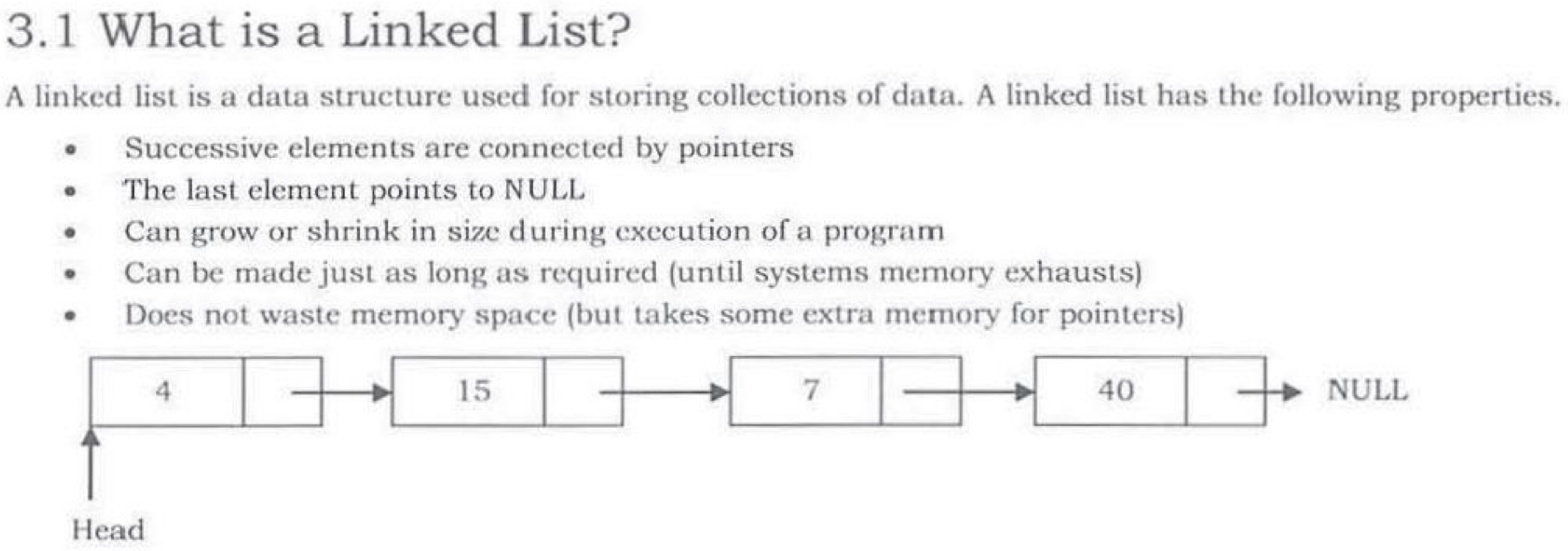
Problem-4 $T(n) = 2^n T(n/2) + n^n$

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



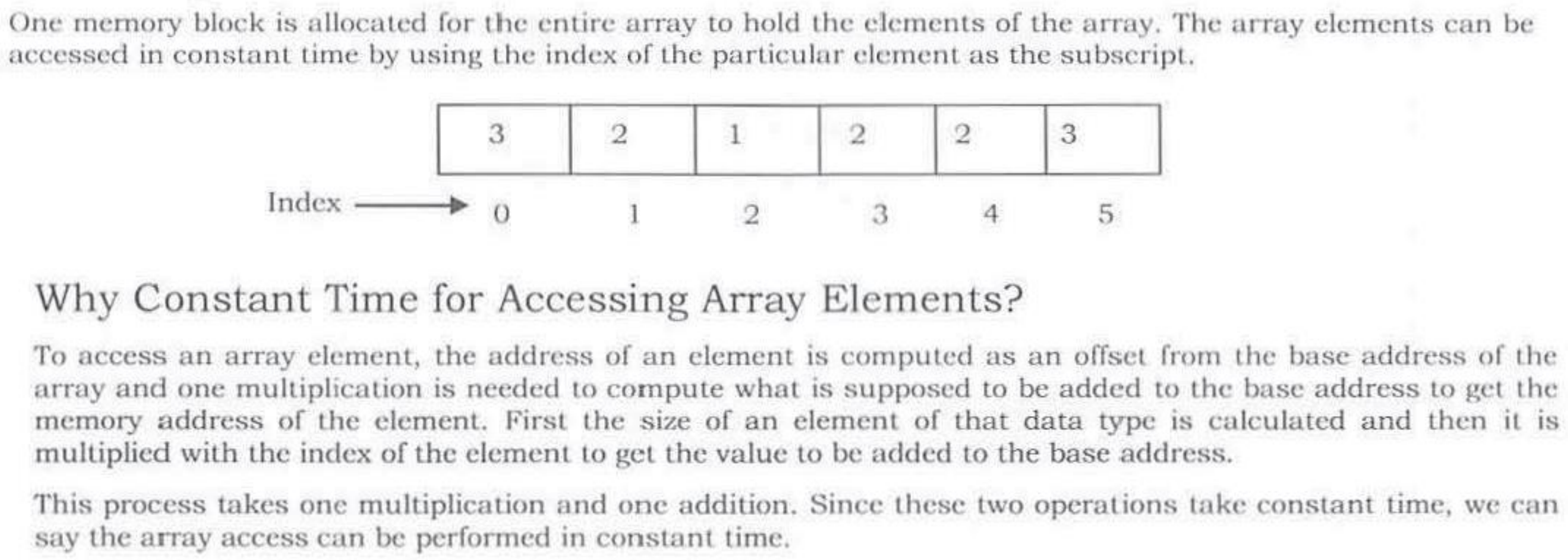
3.1 What is a Linked List?

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.

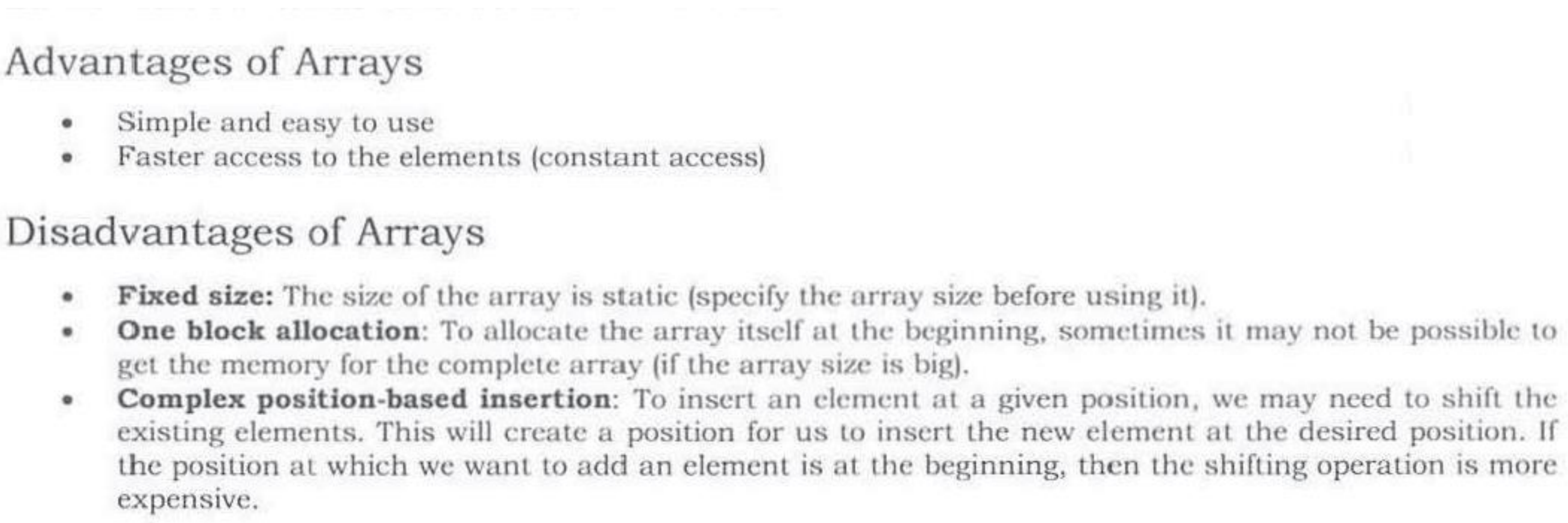
	3	2	1	2	2	3
Index →	0	1	2	3	4	5

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



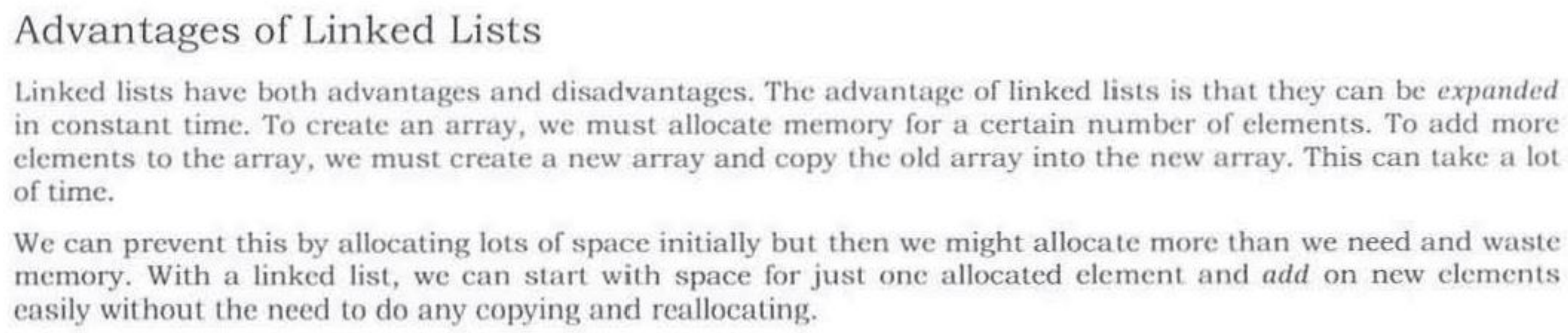
Advantages of Arrays

- 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

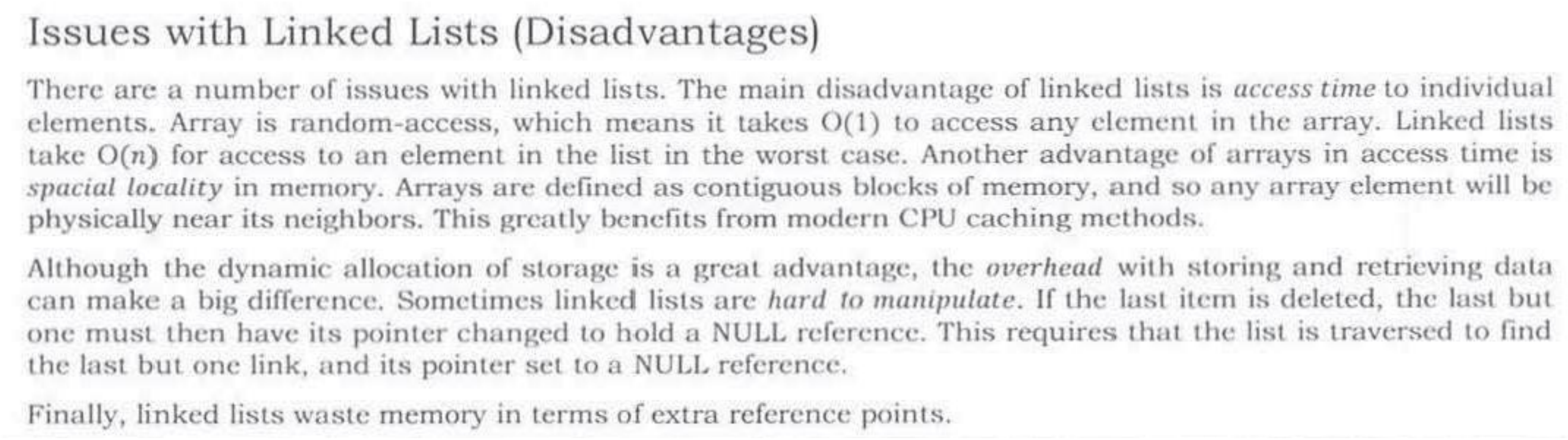


Advantages of Linked Lists

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



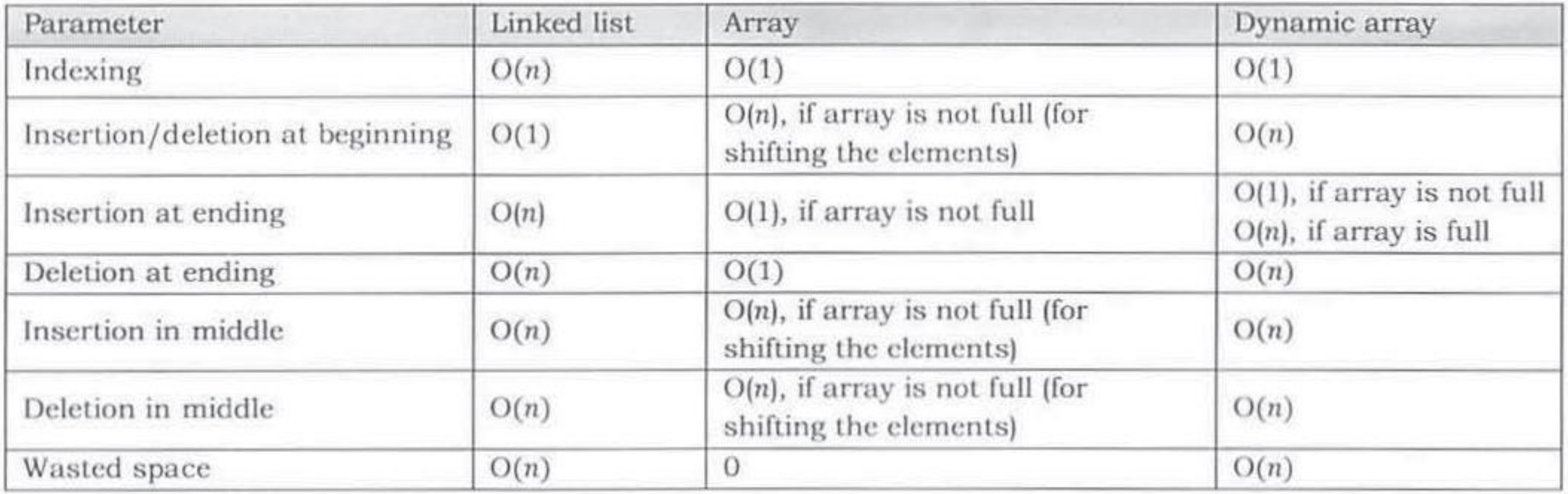
Issues with Linked Lists (Disadvantages)

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.

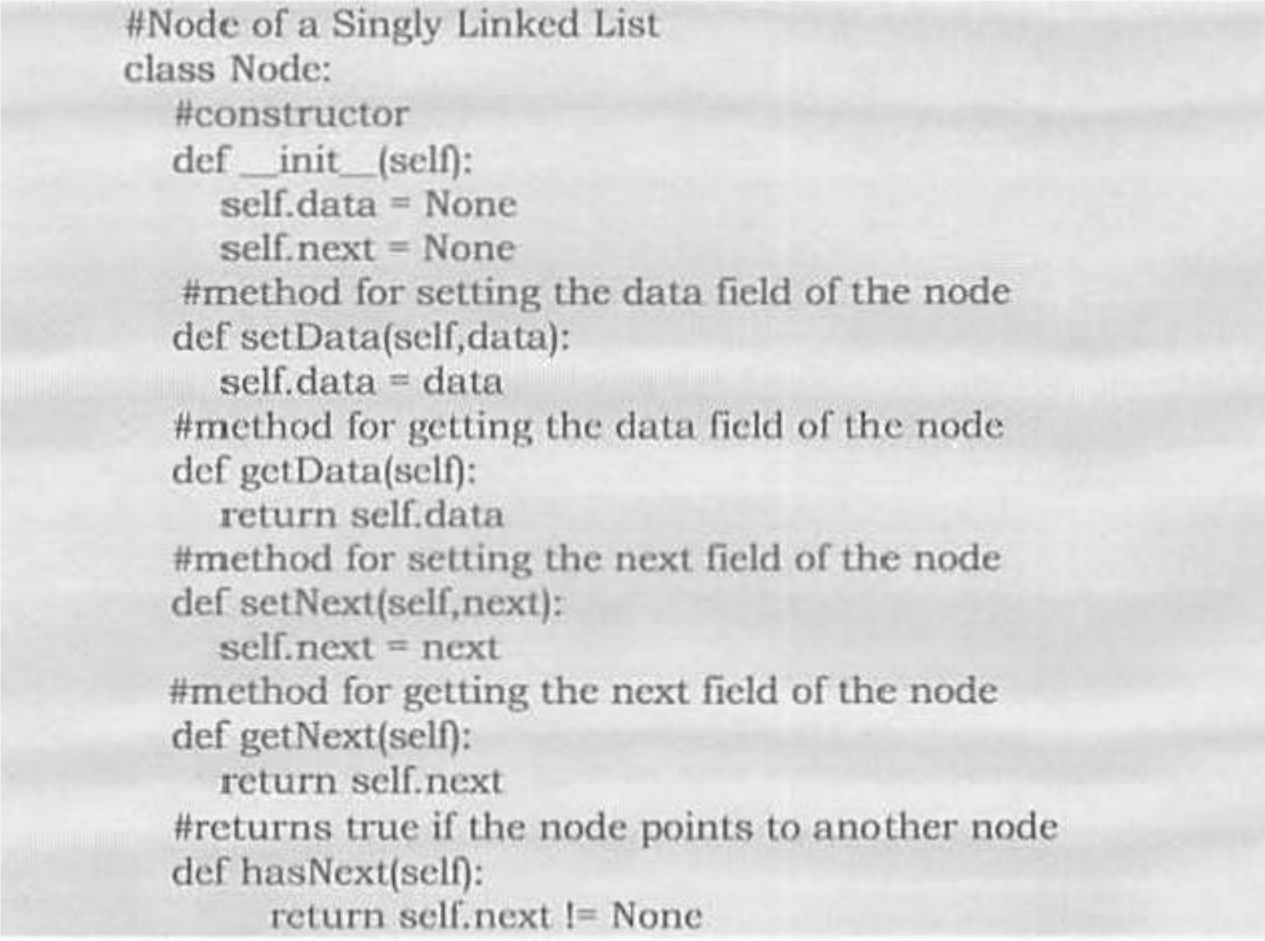
Finally, linked lists waste memory in terms of extra reference points.

Comparison linked list, array and dynamic array



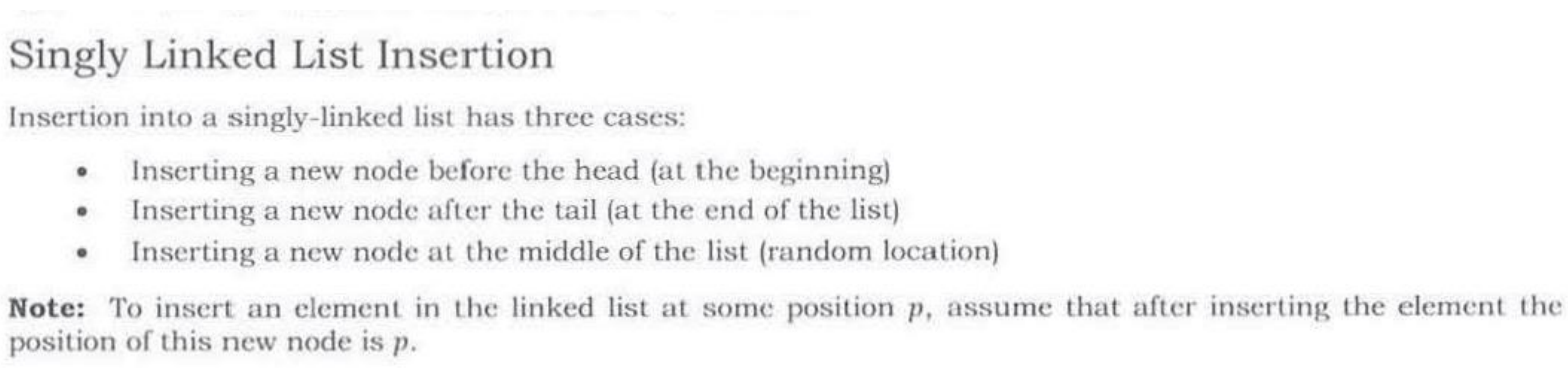
Parameter	Linked list	Array	Dynamic array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Insertion at ending	$O(n)$	$O(1)$, if array is not full	$O(1)$, if array is not full $O(n)$, if array is full
Deletion at ending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



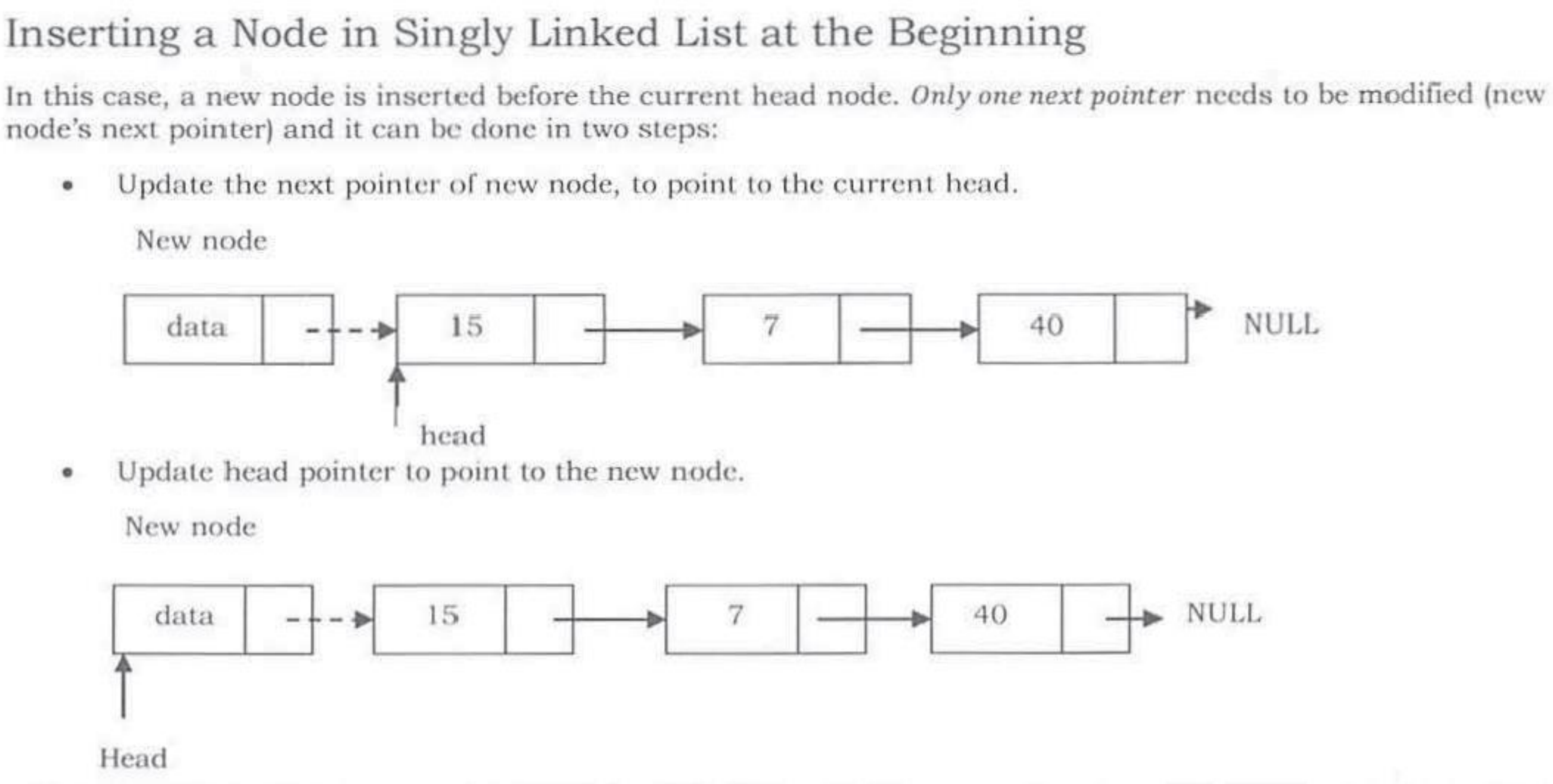
Singly Linked List Insertion

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:

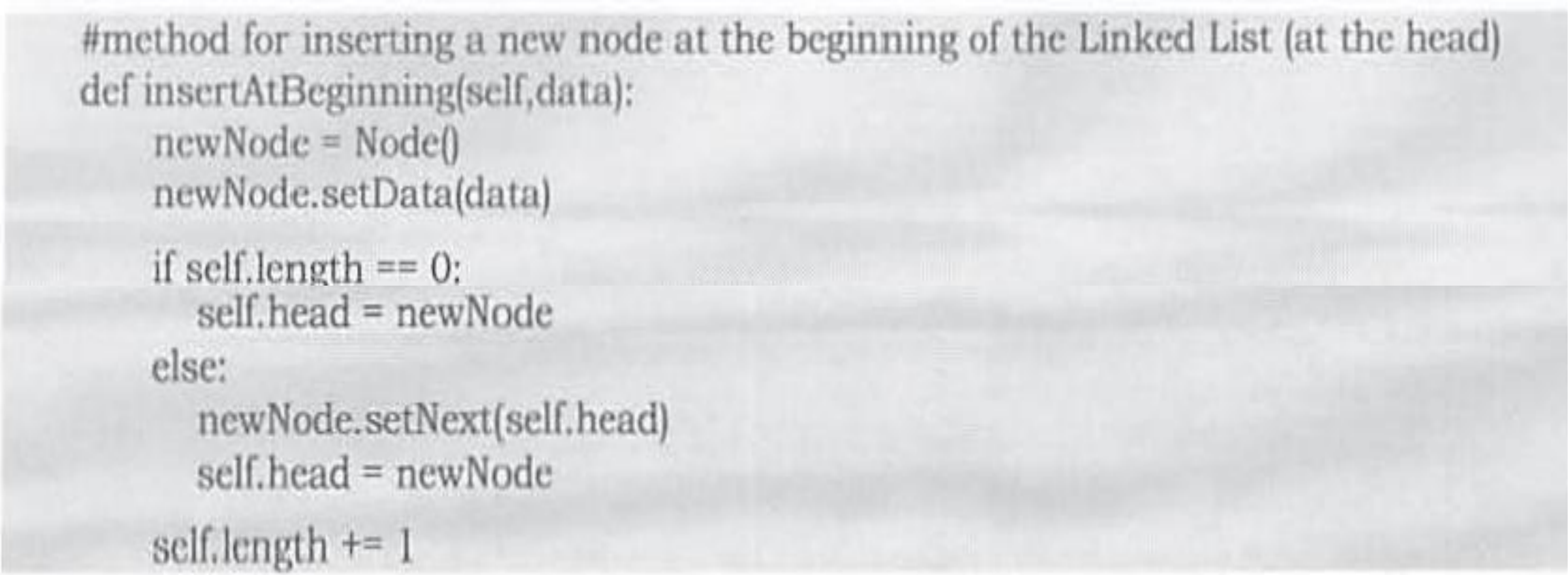
- Update the next pointer of new node, to point to the current head.

New node
data
15
7
40
NULL
head

- Update head pointer to point to the new node.

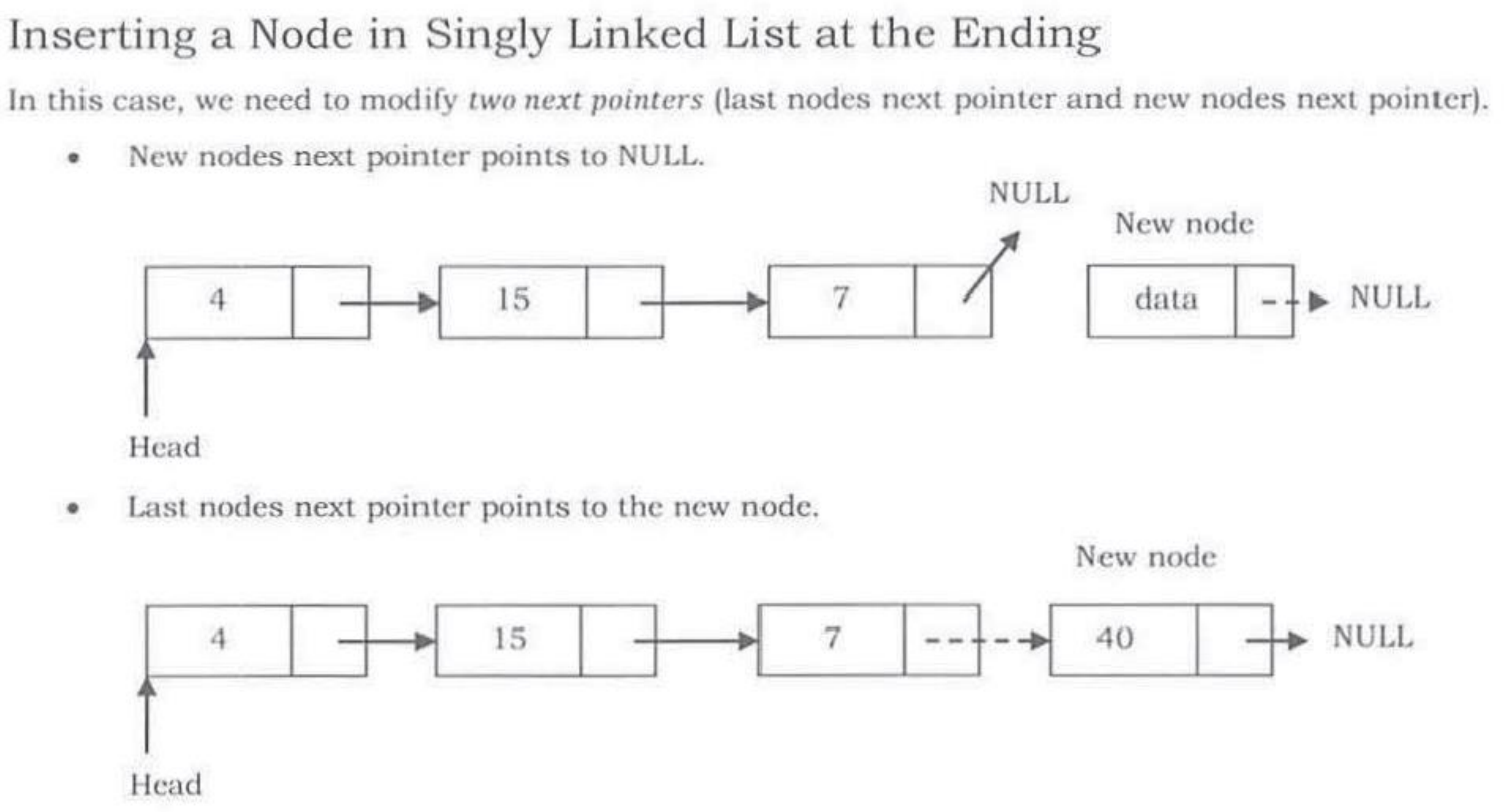
New node
data
15
7
40
NULL
Head

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).

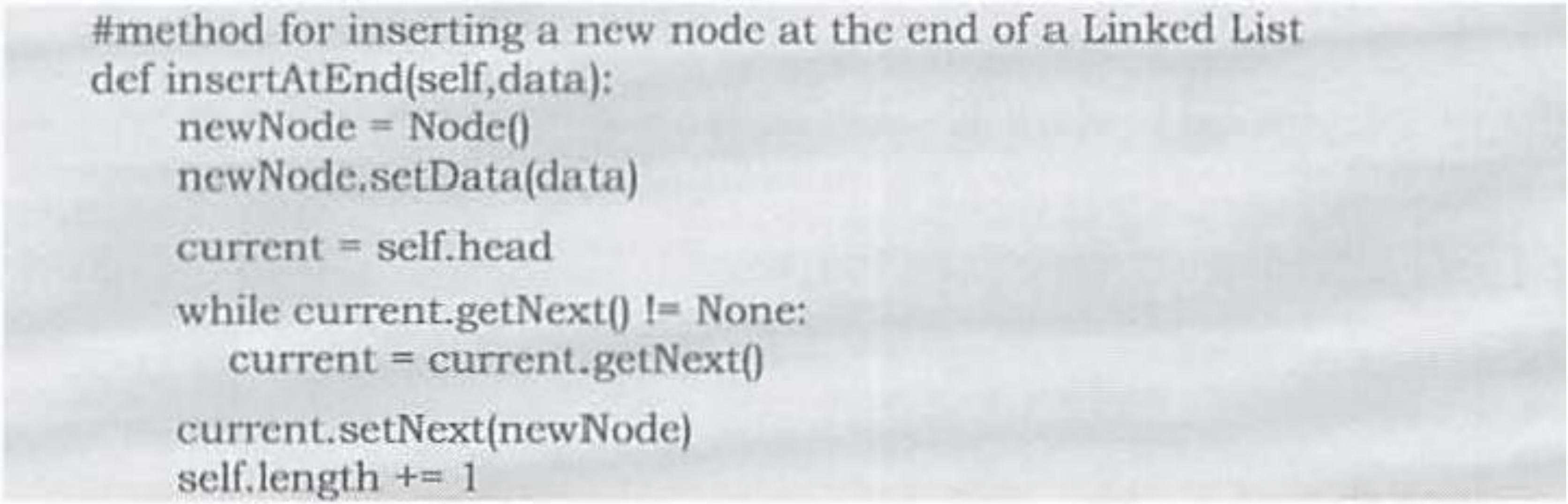
- New nodes next pointer points to NULL.

NULL
New node
4
15
7
data
NULL
Head

- Last nodes next pointer points to the new node.

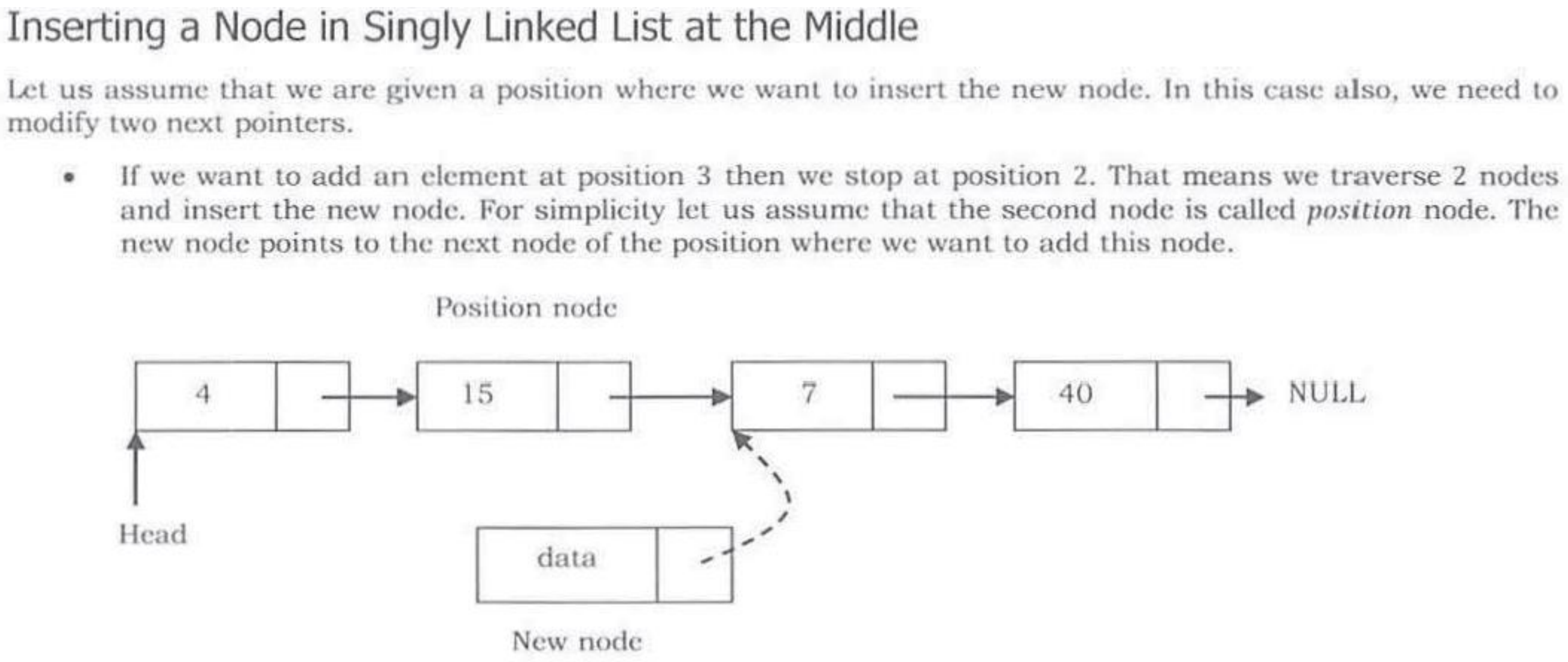
New node
4
15
7
40
NULL
Head

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
```

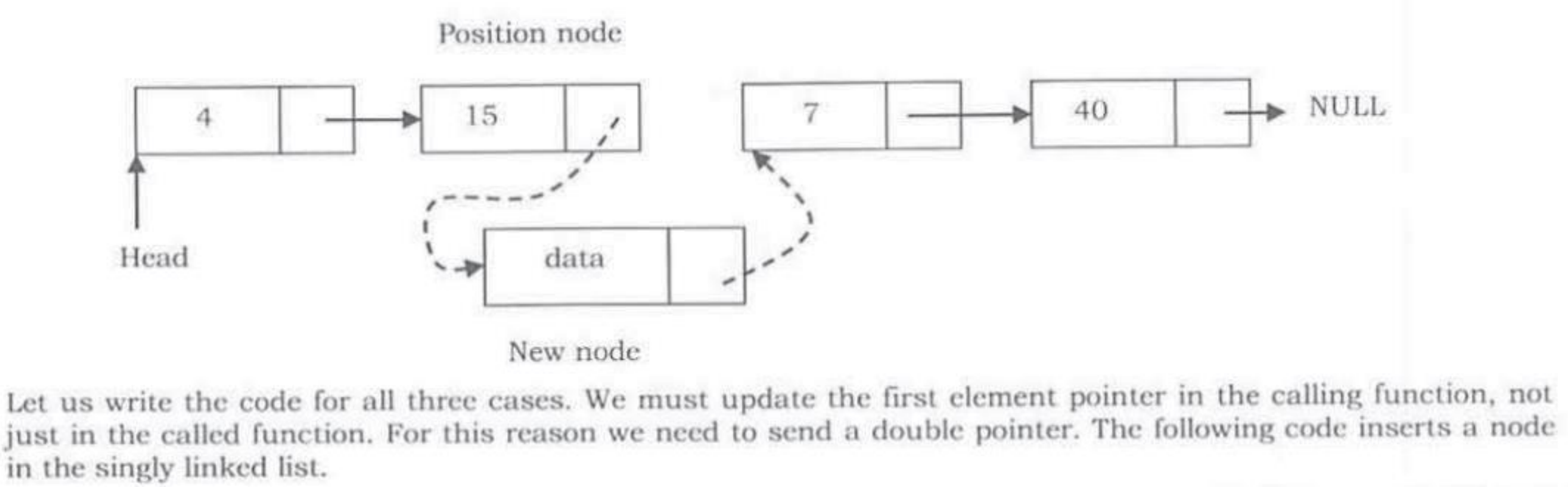
In the middle



Inserting a Node in Singly Linked List at the Middle

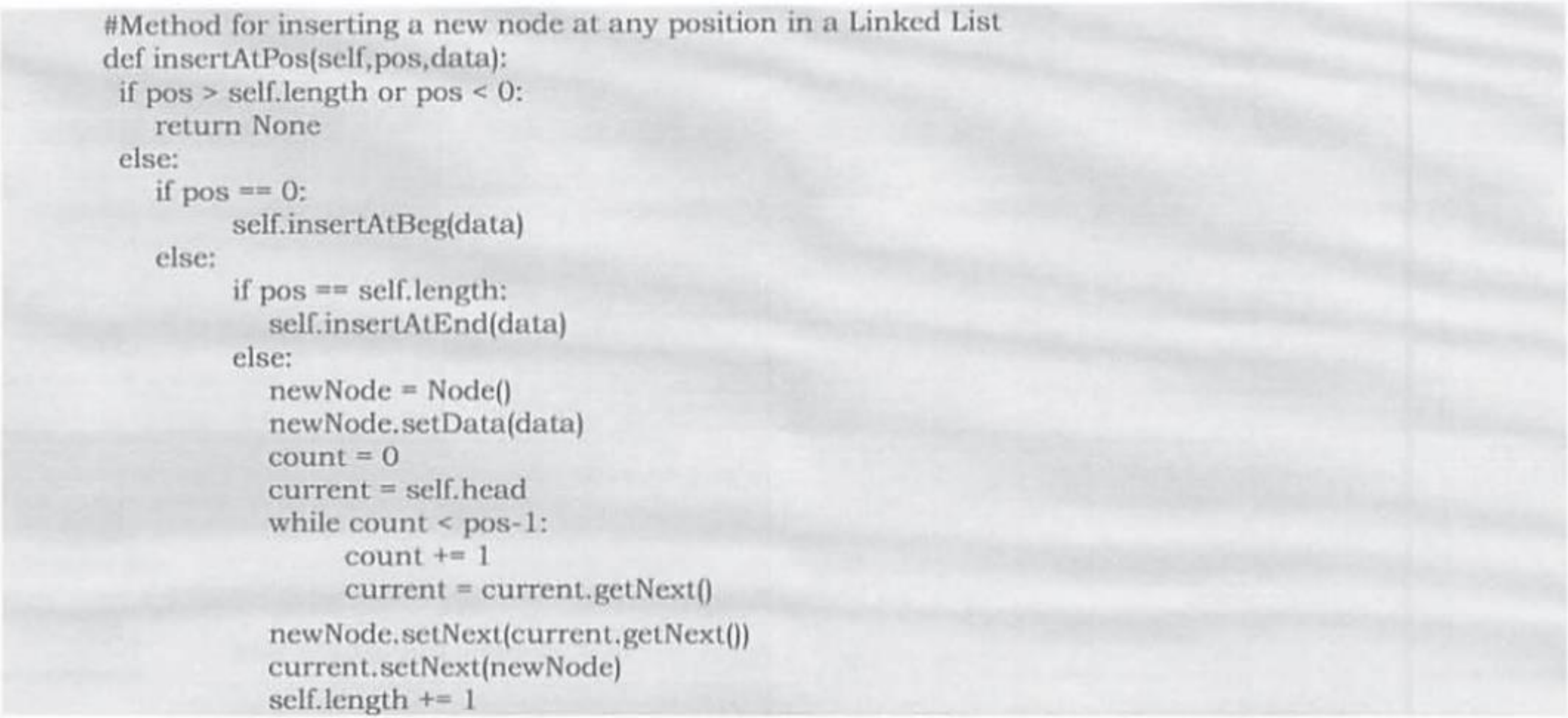
Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.

- 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.



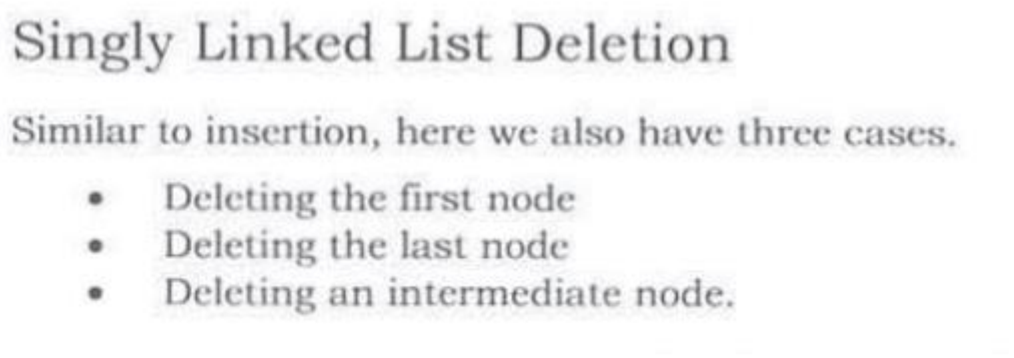
Let us write the code for all three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send a double pointer. The following code inserts a node in the singly linked list.

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

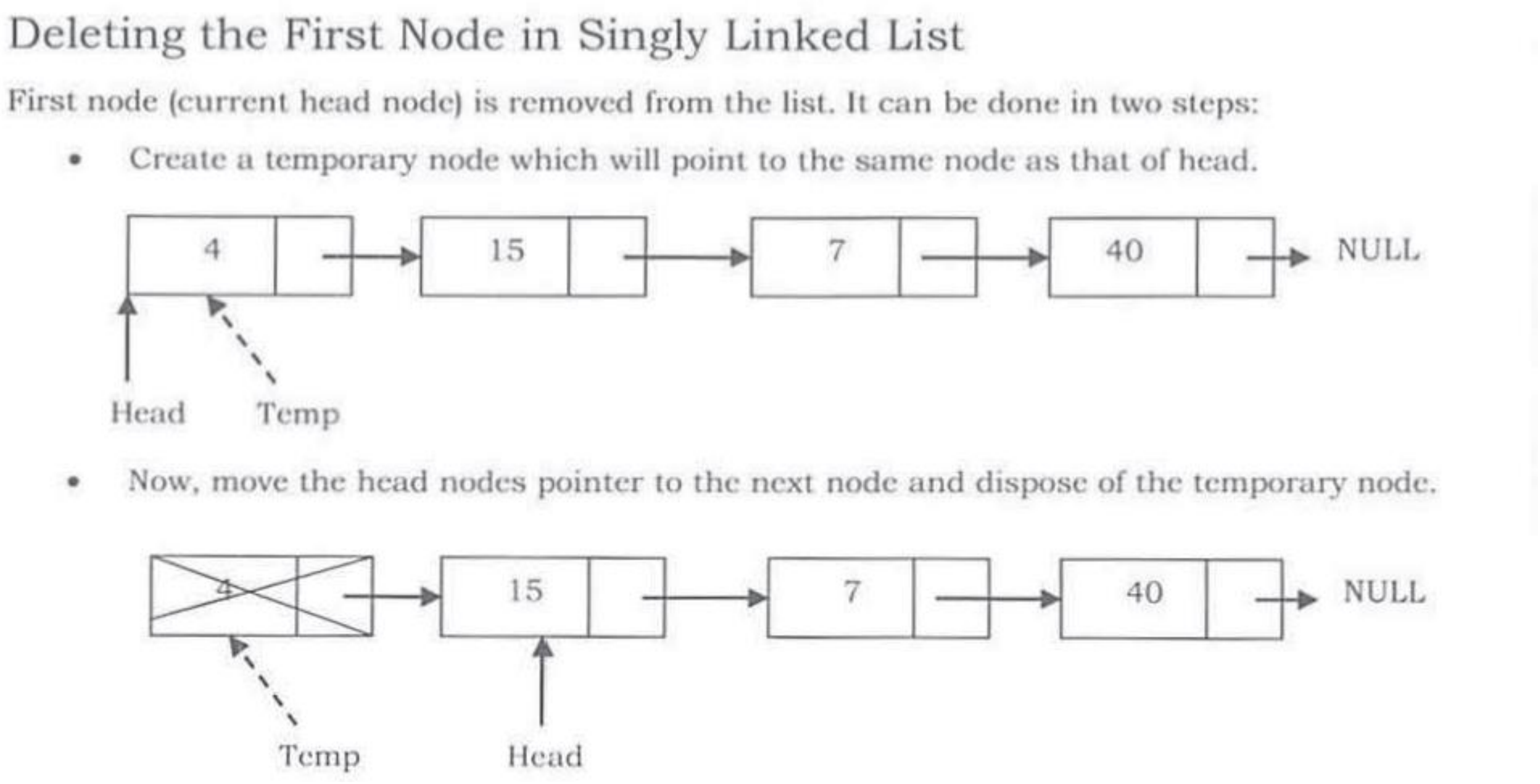


Singly Linked List Deletion

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:

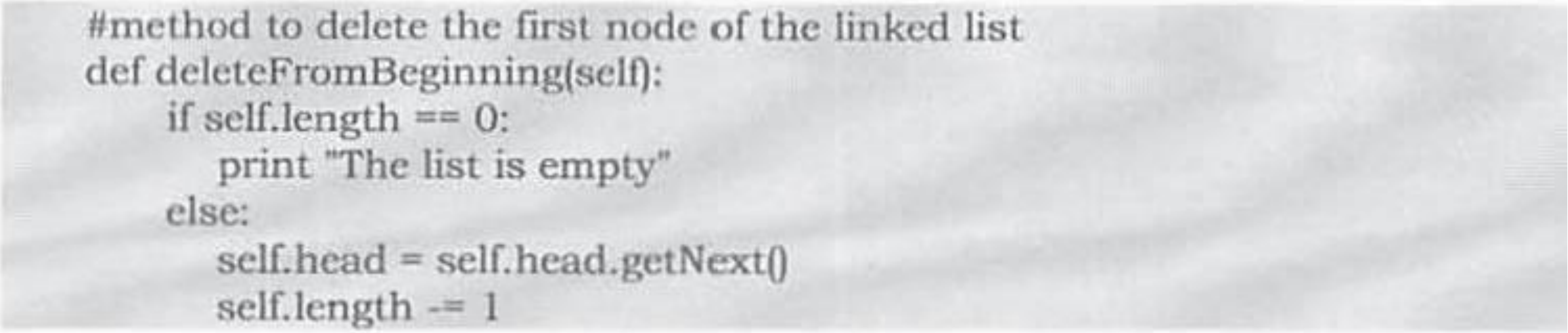
- Create a temporary node which will point to the same node as that of head.

4
15
7
40
NULL
Head
Temp

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

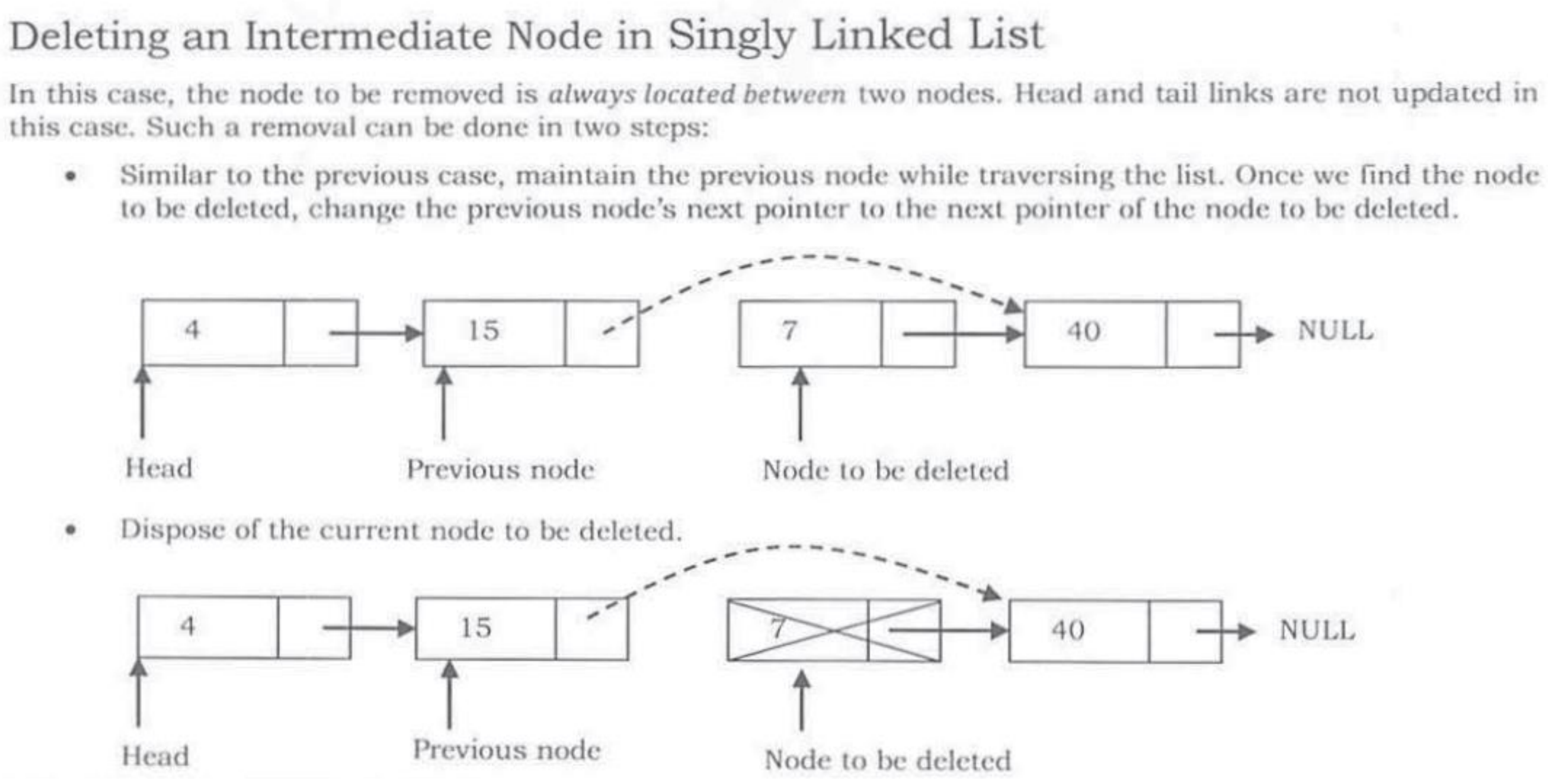
4
15
7
40
NULL
Temp
Head

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



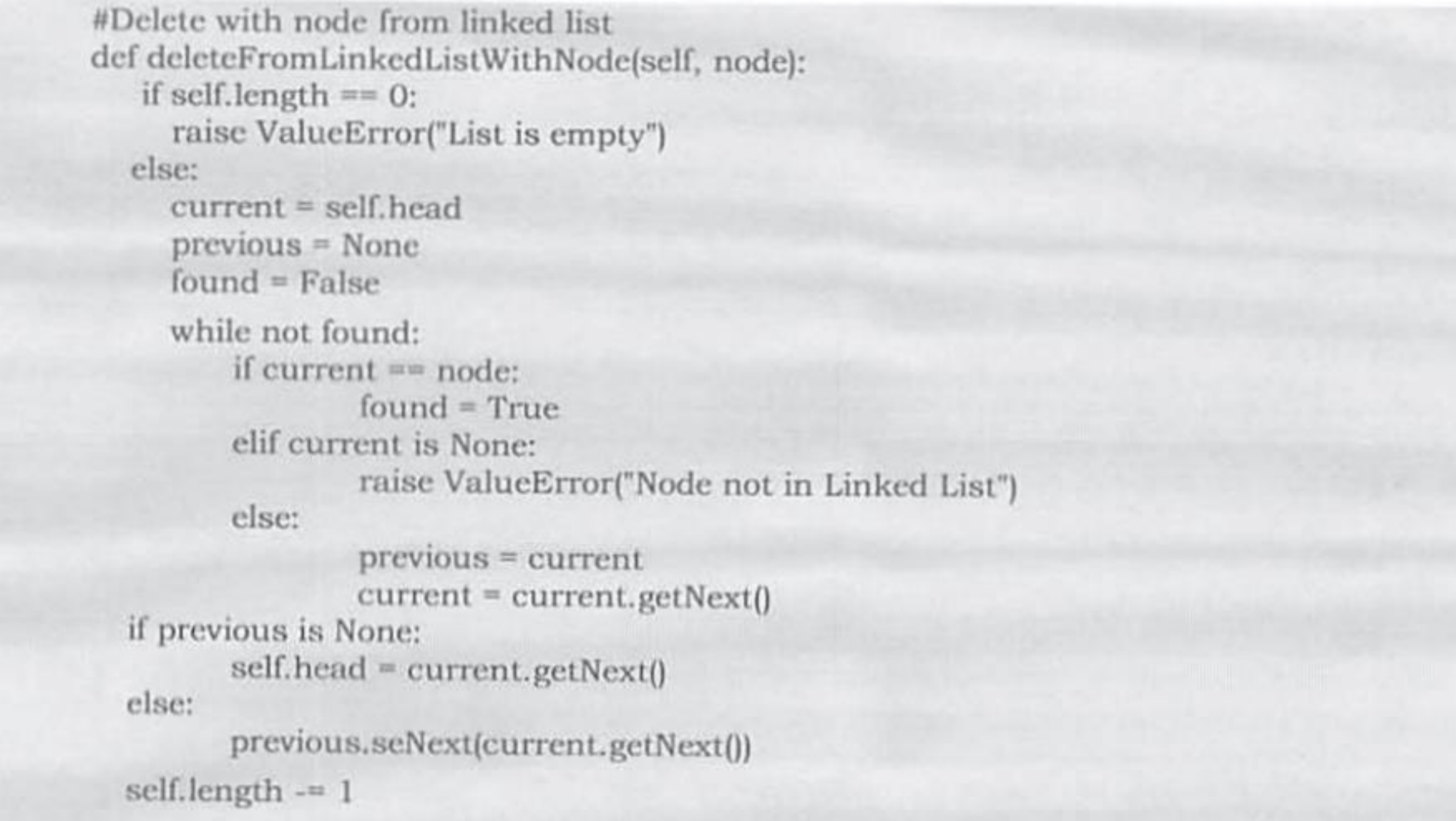
Deleting an Intermediate Node in Singly Linked List

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.

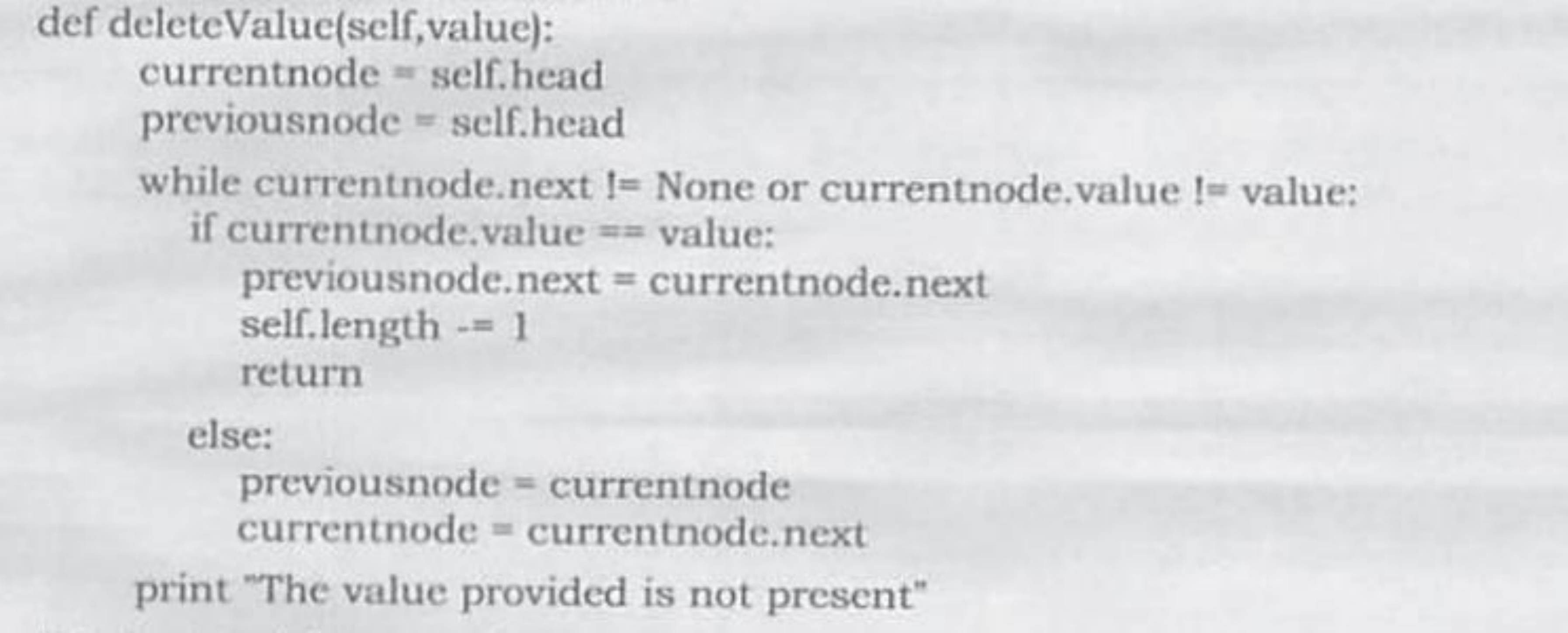
- Dispose of the current node to be deleted.

Code



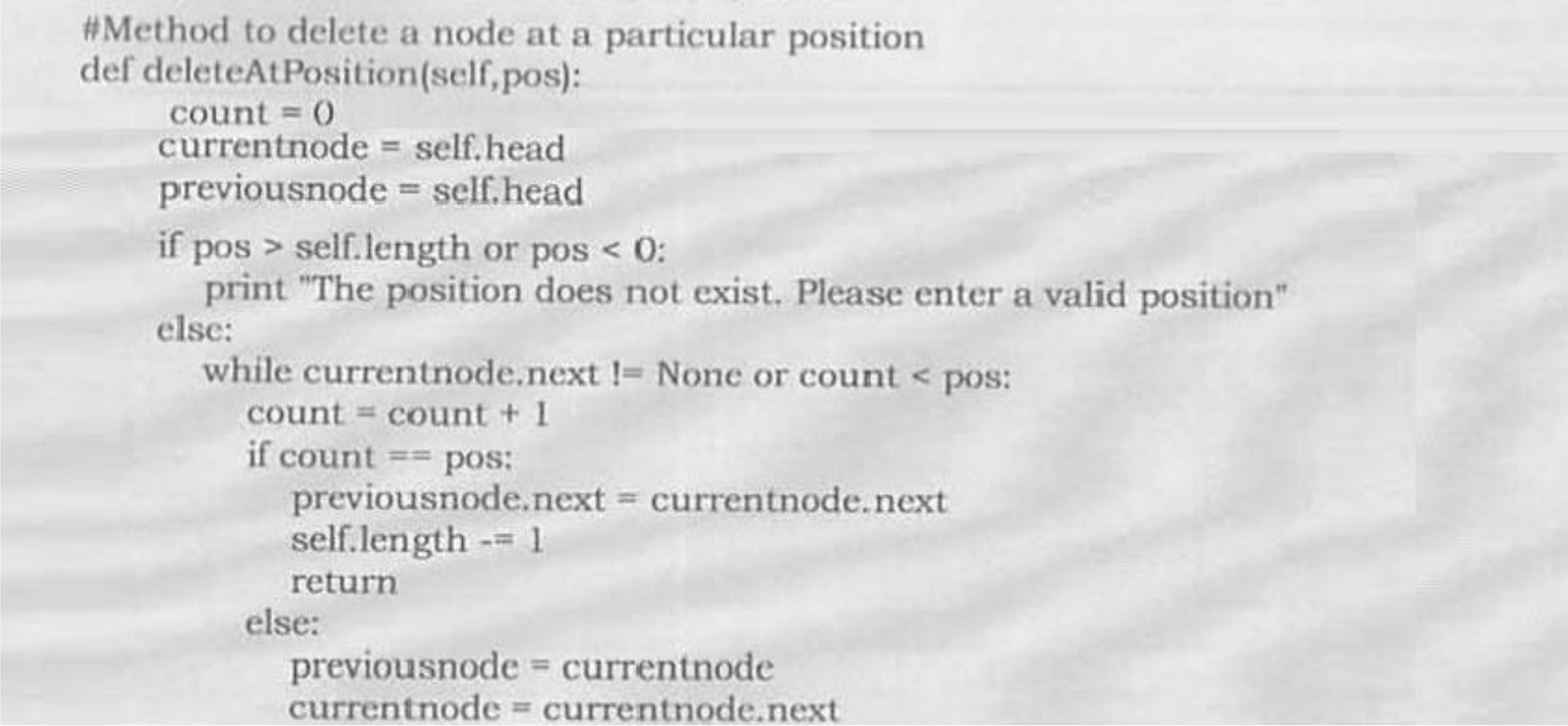
```
#Delete with node from linked list
def deleteFromLinkedListWithNode(self, node):
    if self.length == 0:
        raise ValueError("List is empty")
    else:
        current = self.head
        previous = None
        found = False
        while not found:
            if current == node:
                    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 Sofia Vasquez

Reminder of the objective of this course

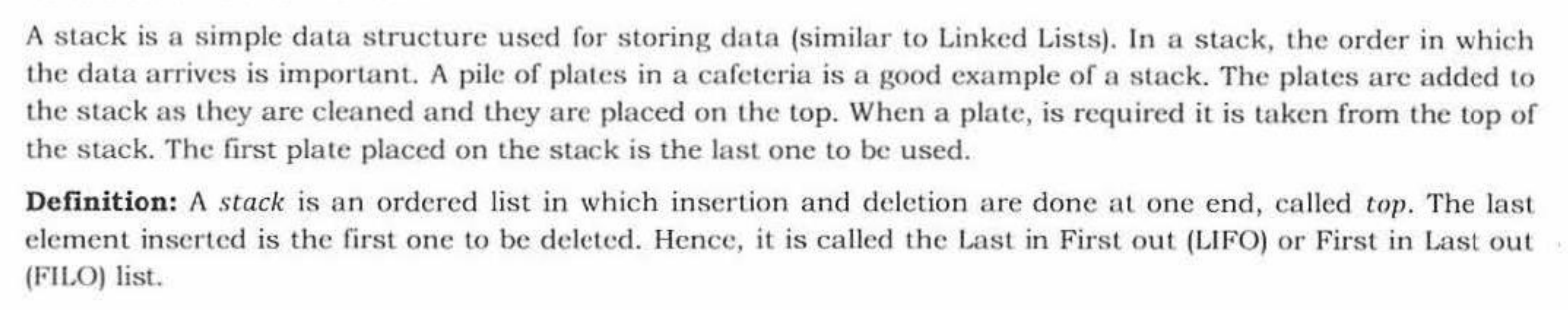
- People often learn about data structures out of context
- But in this course you will learn foundational concepts by building a real application with python and Flask (we will start in session 3)!

- To learn the ins and outs of the essential data structure, experiencing in practice has proved to be a much more powerful way to learn data structures

Outline

1. Stack
   a) Concepts
   b) Implementation choice
   c) Corresponding codes
2. Queue
   a) Concepts
   b) Implementation choice
   c) Corresponding codes

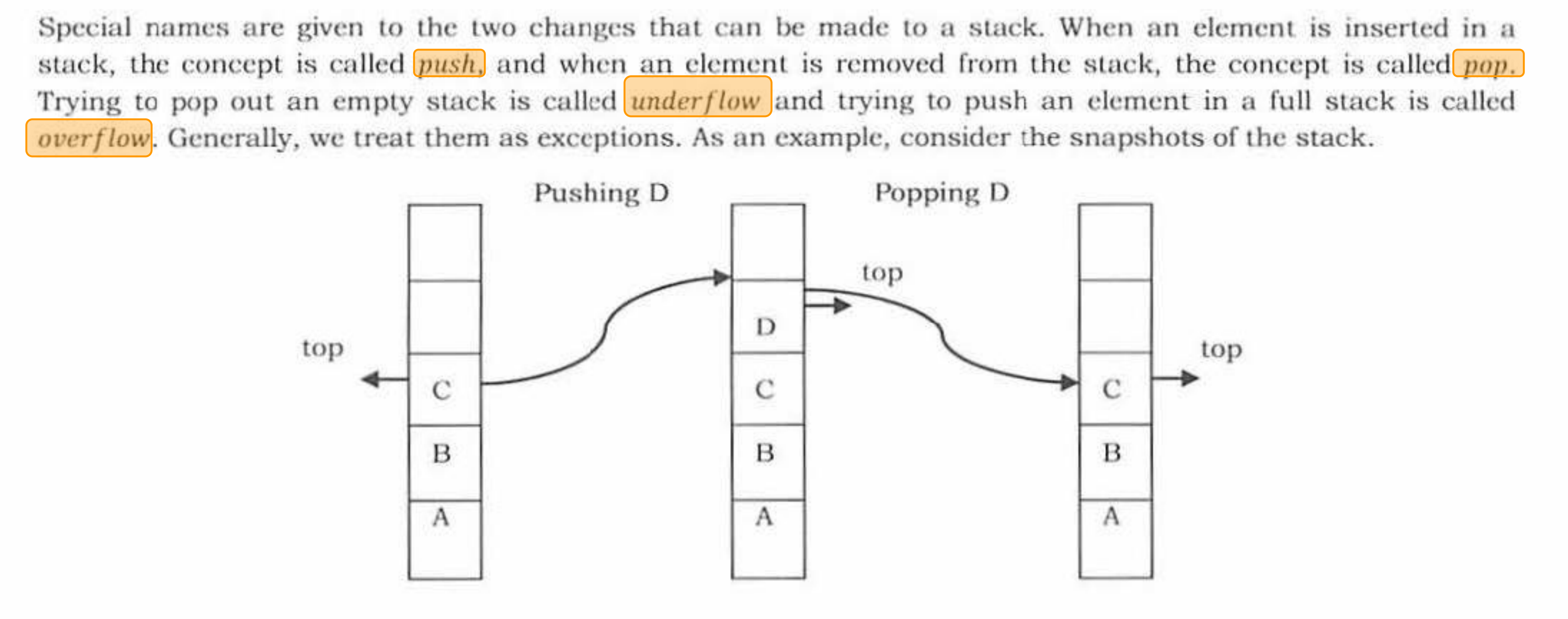
What is a stack



A stack is a simple data structure used for storing data (similar to Linked Lists). In a stack, the order in which the data arrives is important. A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and they are placed on the top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

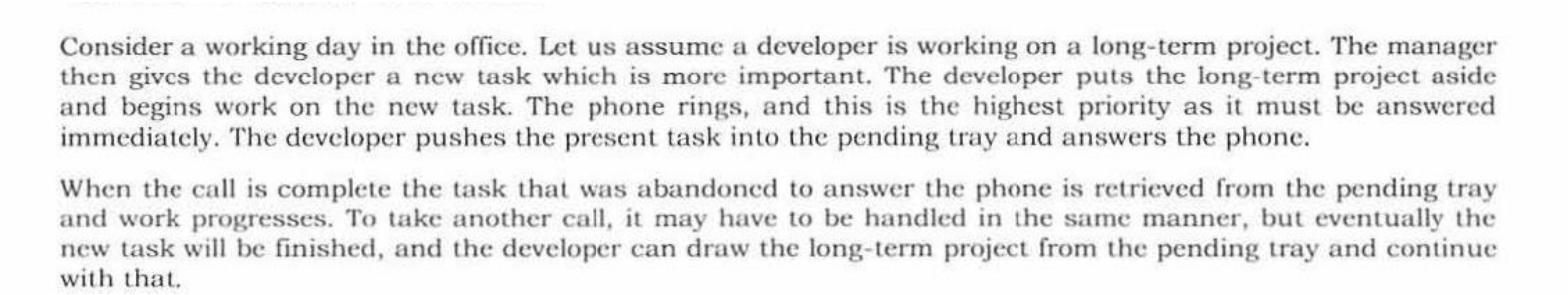
Definition: A *stack* is an ordered list in which insertion and deletion are done at one end, called *top*. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.

Special names



Special names are given to the two changes that can be made to a stack. When an element is inserted in a stack, the concept is called *push*, and when an element is removed from the stack, the concept is called *pop*. Trying to pop out an empty stack is called *underflow* and trying to push an element in a full stack is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshots of the stack.

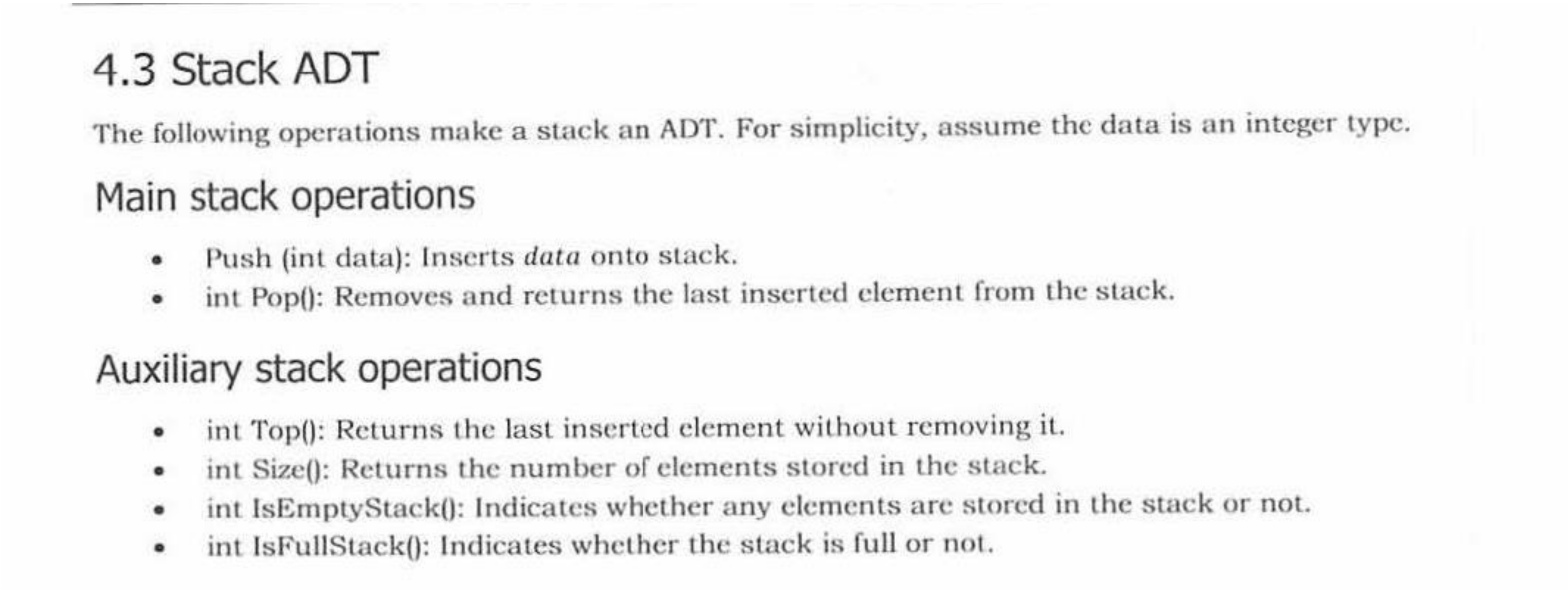
How stack are used?



Consider a working day in the office. Let us assume a developer is working on a long-term project. The manager then gives the developer a new task which is more important. The developer puts the long-term project aside and begins work on the new task. The phone rings, and this is the highest priority as it must be answered immediately. The developer pushes the present task into the pending tray and answers the phone.

When the call is complete the task that was abandoned to answer the phone is retrieved from the pending tray and work progresses. To take another call, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.

Stack advanced data structure



4.3 Stack ADT

The following operations make a stack an ADT. For simplicity, assume the data is an integer type.

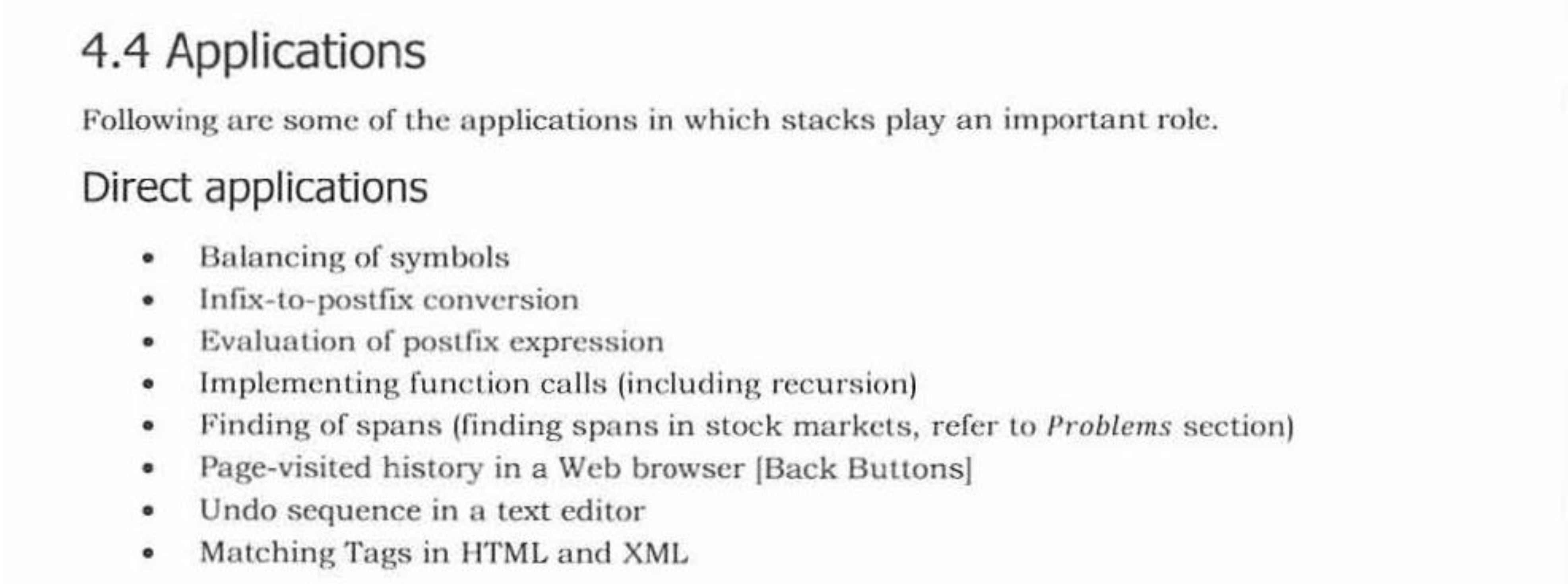
Main stack operations

- Push (int data): Inserts *data* onto stack.
- int Pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

- int Top(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in the stack.
- int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.

Direct applications



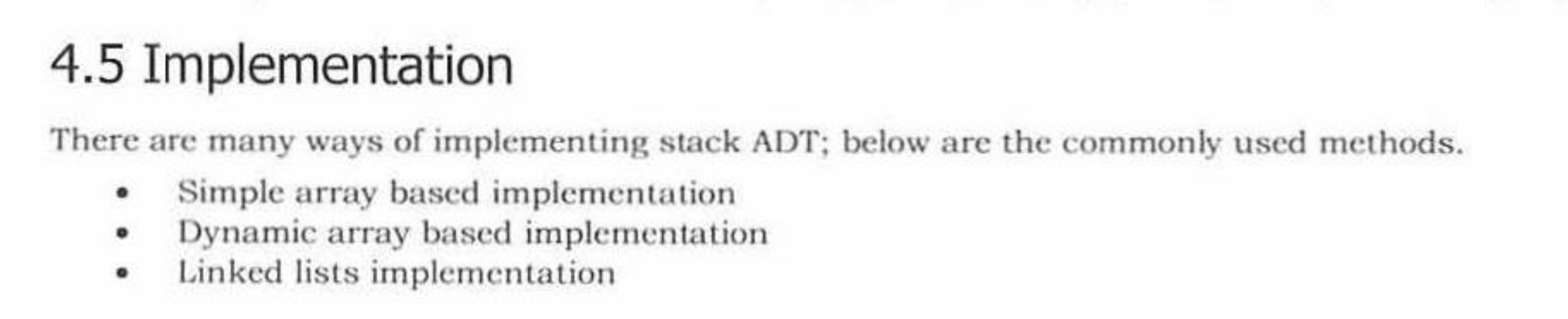
4.4 Applications

Following are some of the applications in which stacks play an important role.

Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer to *Problems* section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML

Implementation

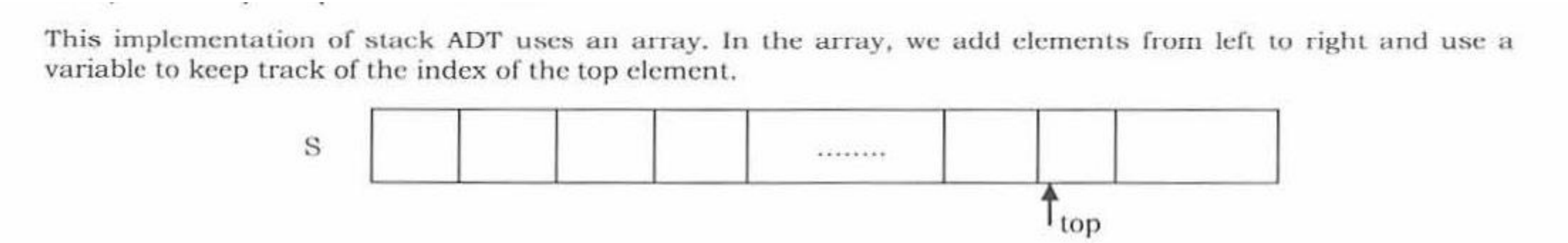


4.5 Implementation

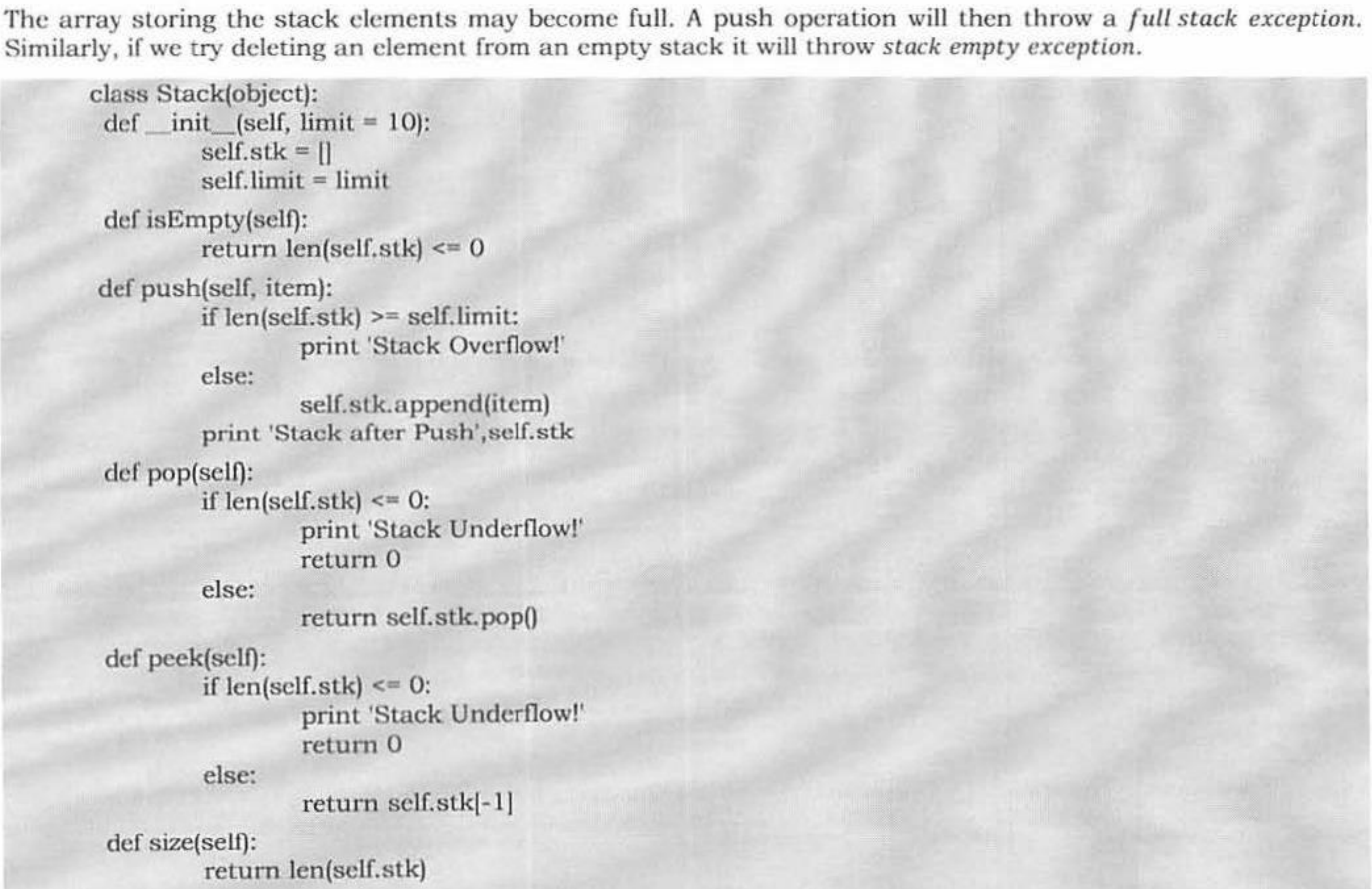
There are many ways of implementing stack ADT; below are the commonly used methods.

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation

Simple array implementation



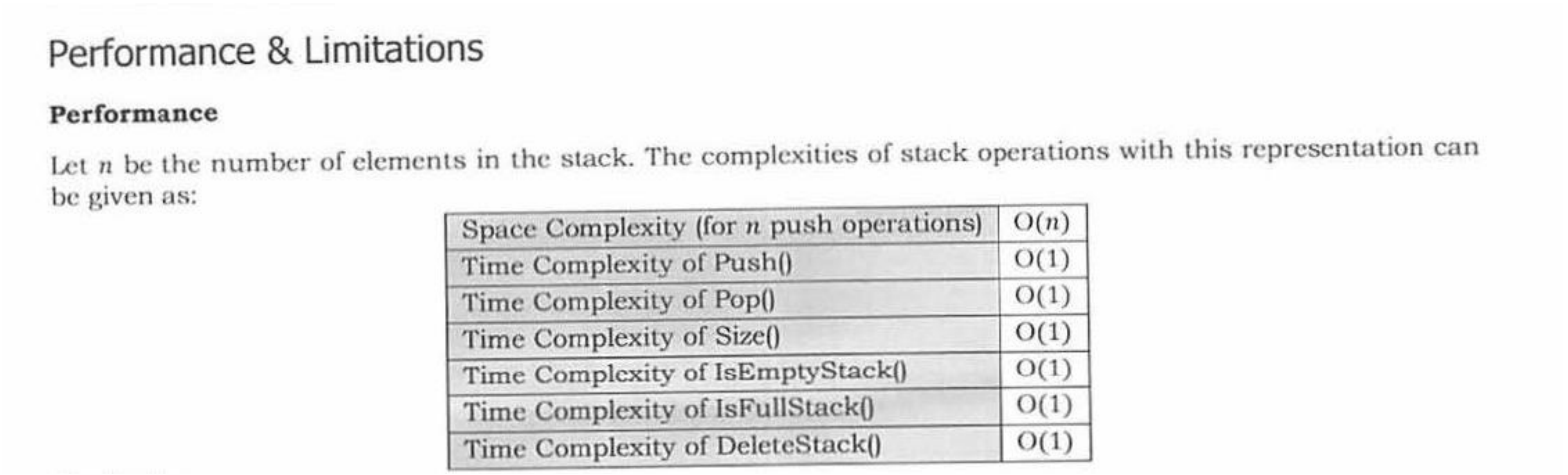
This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.



The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from an empty stack it will throw *stack empty exception*.

```
class Stack(object):
    def __init__(self, limit = 10):
        self.stk = []
        self.limit = limit
    def isEmpty(self):
        return len(self.stk) <= 0
    def push(self, item):
        if len(self.stk) >= self.limit:
            print 'Stack Overflow!'
        else:
            self.stk.append(item)
        print 'Stack after Push',self.stk
    def pop(self):
        if len(self.stk) <= 0:
            print 'Stack Underflow!'
            return 0
        else:
            return self.stk.pop()
    def peek(self):
        if len(self.stk) <= 0:
            print 'Stack Underflow!'
            return 0
        else:
            return self.stk[-1]
    def size(self):
        return len(self.stk)
```

Performance and limitations?



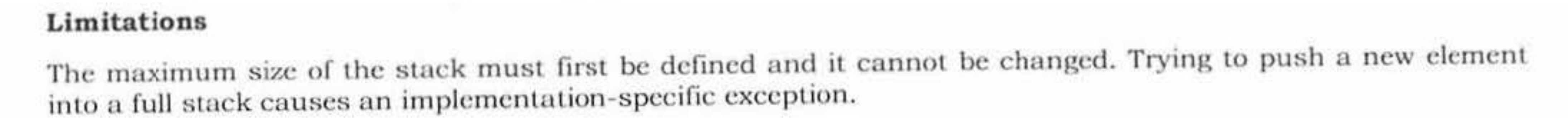
Performance & Limitations

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

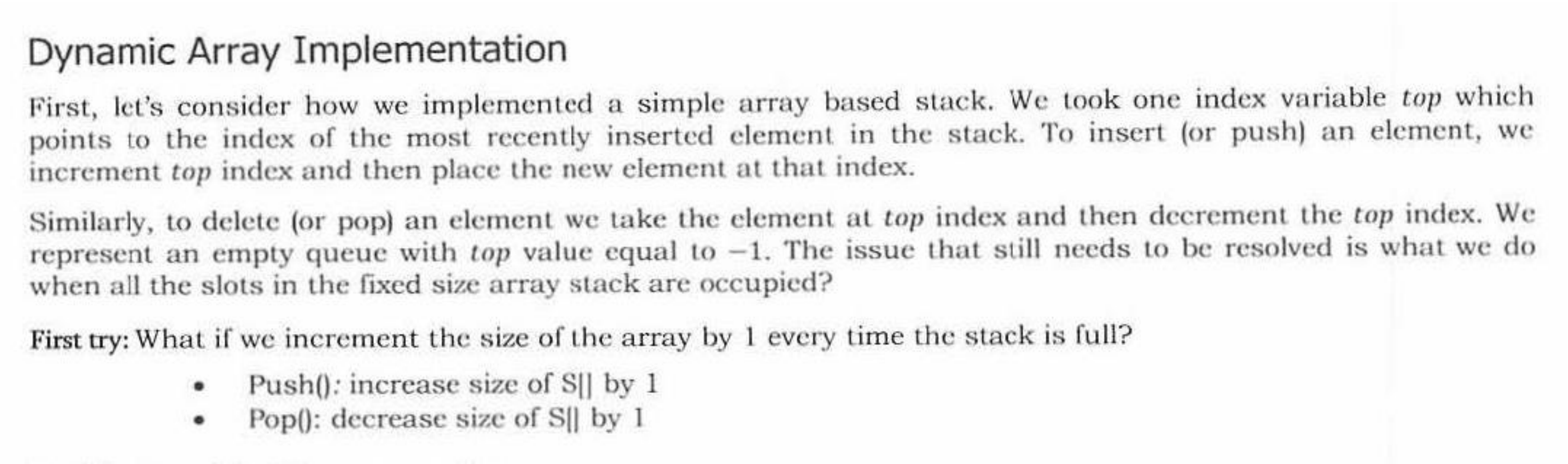
Space Complexity (for n push operations)	$O(n)$
Time Complexity of Push()	$O(1)$
Time Complexity of Pop()	$O(1)$
Time Complexity of Size()	$O(1)$
Time Complexity of IsEmptyStack()	$O(1)$
Time Complexity of IsFullStack()	$O(1)$
Time Complexity of DeleteStack()	$O(1)$

Limitations?



Limitations

The maximum size of the stack must first be defined and it cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.



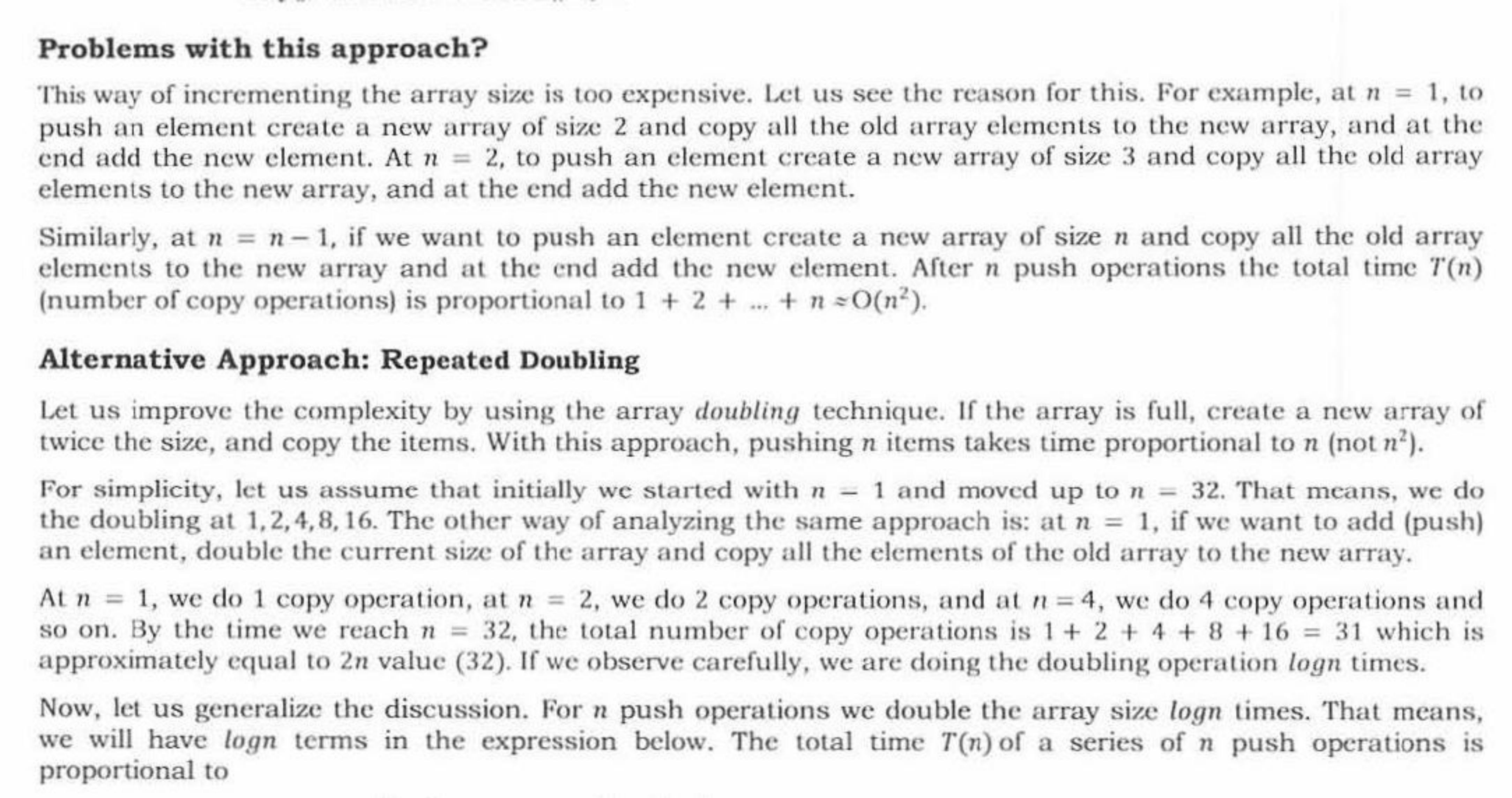
Dynamic Array Implementation

First, let's consider how we implemented a simple array based stack. We took one index variable *top* which points to the index of the most recently inserted element in the stack. To insert (or push) an element, we increment *top* index and then place the new element at that index.

Similarly, to delete (or pop) an element we take the element at *top* index and then decrement the *top* index. We represent an empty queue with *top* value equal to −1. The issue that still needs to be resolved is what we do when all the slots in the fixed size array stack are occupied?

First try: What if we increment the size of the array by 1 every time the stack is full?

- Push(): increase size of S|| by 1
- Pop(): decrease size of S|| by 1



Problems with this approach?

This way of incrementing the array size is too expensive. Let us see the reason for this. For example, at $n = 1$, to push an element create a new array of size 2 and copy all the old array elements to the new array, and at the end add the new element. At $n = 2$, to push an element create a new array of size 3 and copy all the old array elements to the new array, and at the end add the new element.

Similarly, at $n = n - 1$, if we want to push an element create a new array of size n and copy all the old array elements to the new array and at the end add the new element. After n push operations the total time $T(n)$ (number of copy operations) is proportional to $1 + 2 + ... + n \approx O(n^2)$.

Alternative Approach: Repeated Doubling

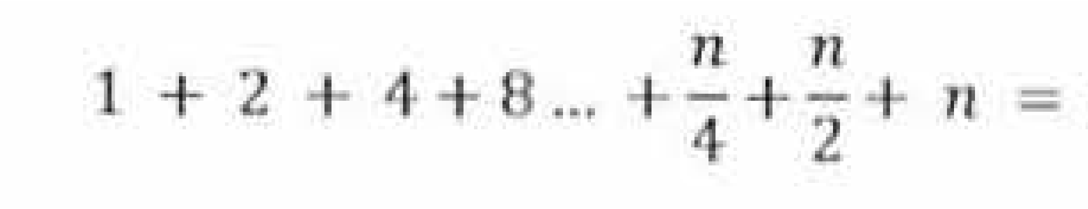
Let us improve the complexity by using the array *doubling* technique. If the array is full, create a new array of twice the size, and copy the items. With this approach, pushing n items takes time proportional to n (not n^2).

For simplicity, let us assume that initially we started with $n = 1$ and moved up to $n = 32$. That means, we do the doubling at $1, 2, 4, 8, 16$. The other way of analyzing the same approach is: at $n = 1$, if we want to add (push) an element, double the current size of the array and copy all the elements of the old array to the new array.

At $n = 1$, we do 1 copy operation, at $n = 2$, we do 2 copy operations, and at $n = 4$, we do 4 copy operations and so on. By the time we reach $n = 32$, the total number of copy operations is $1 + 2 + 4 + 8 + 16 = 31$ which is approximately equal to $2n$ value (32). If we observe carefully, we are doing the doubling operation *logn* times.

Now, let us generalize the discussion. For n push operations we double the array size *logn* times. That means, we will have *logn* terms in the expression below. The total time $T(n)$ of a series of n push operations is proportional to

Question compute the complexity?

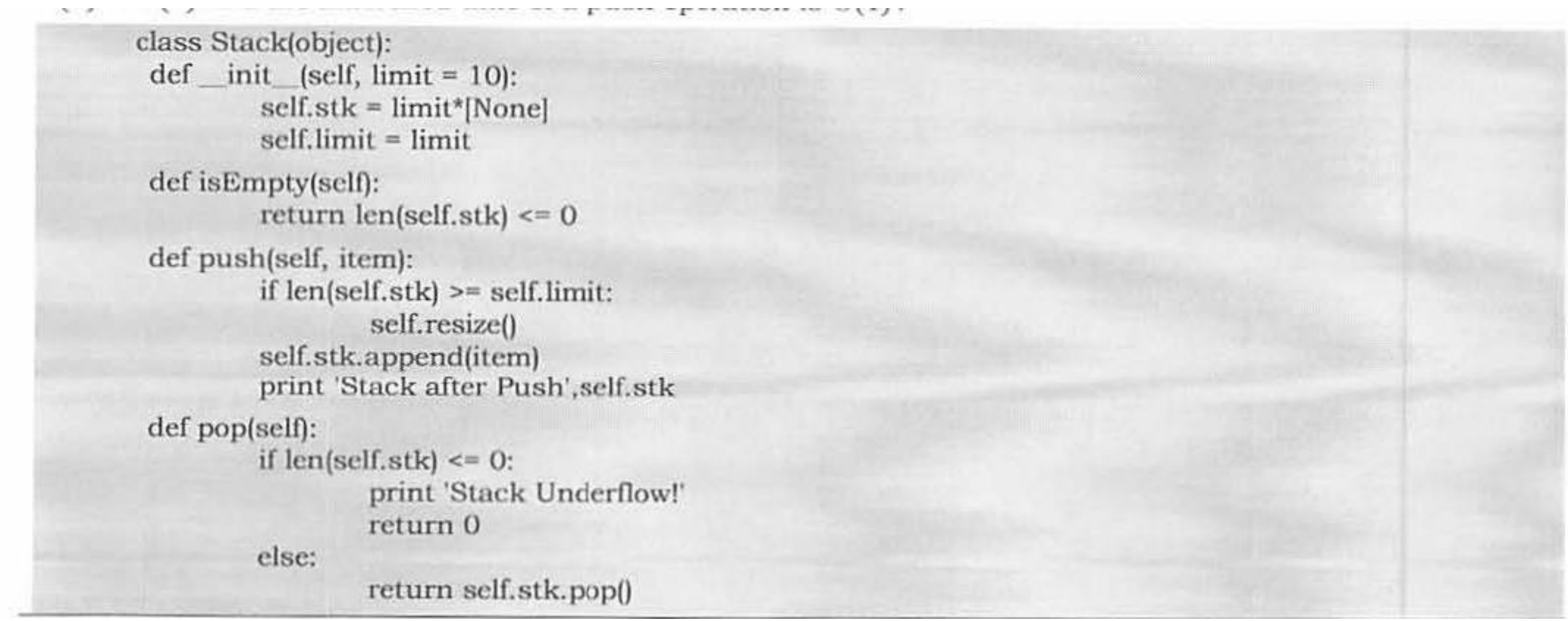


$$1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n = \text{ ?}$$

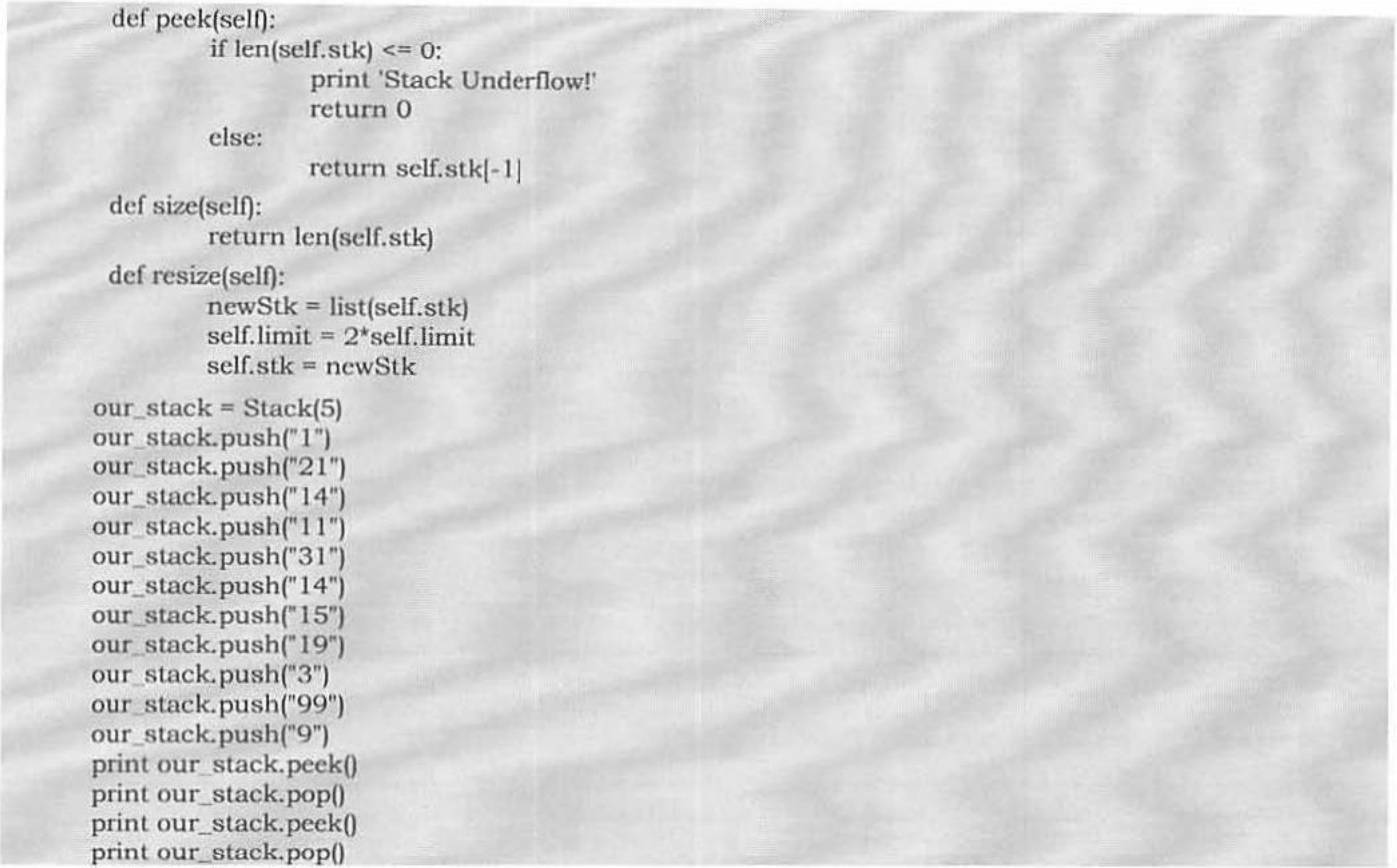
Solution

$$
\begin{aligned}
1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1 \\
&= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right) \\
&= n(2) \approx 2n = O(n)
\end{aligned}
$$

Implementation

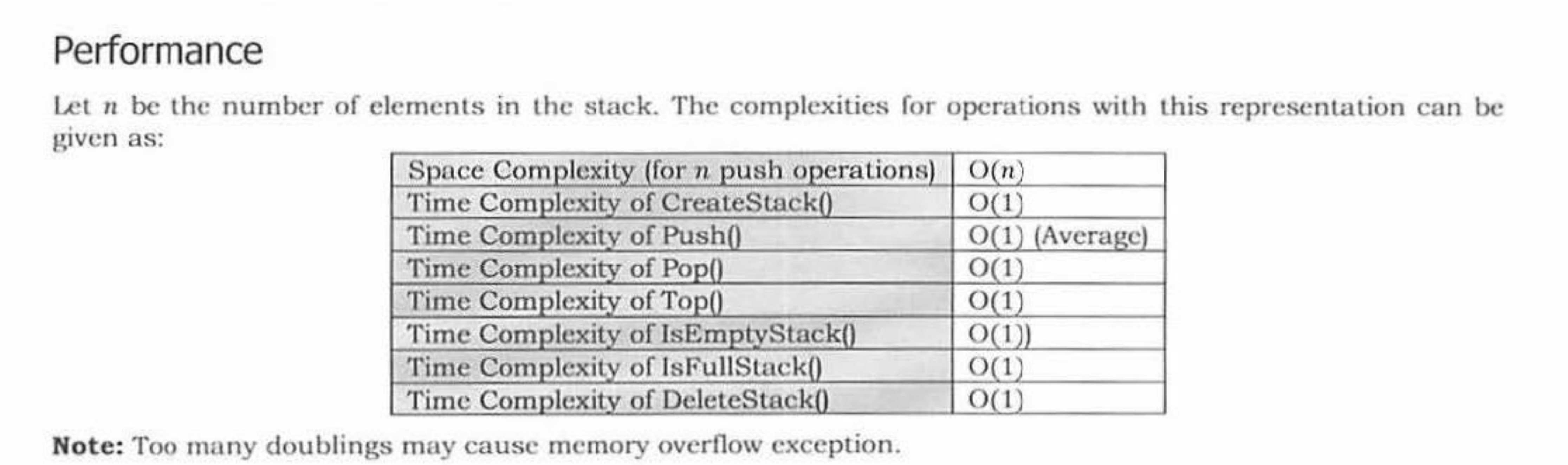


```
class Stack(object):
  def __init__(self, limit = 10):
          self.stk = limit*[None]
          self.limit = limit
  def isEmpty(self):
          return len(self.stk) <= 0
  def push(self, item):
          if len(self.stk) >= self.limit:
                  self.resize()
          self.stk.append(item)
          print 'Stack after Push',self.stk
  def pop(self):
          if len(self.stk) <= 0:
                  print 'Stack Underflow!'
                  return 0
          else:
                  return self.stk.pop()
```



```
    def peek(self):
        if len(self.stk) <= 0:
            print 'Stack Underflow!'
            return 0
        else:
            return self.stk[-1]
    def size(self):
        return len(self.stk)
    def resize(self):
        newStk = list(self.stk)
        self.limit = 2*self.limit
        self.stk = newStk
our_stack = Stack(5)
our_stack.push("1")
our_stack.push("21")
our_stack.push("14")
our_stack.push("11")
our_stack.push("31")
our_stack.push("14")
our_stack.push("15")
our_stack.push("19")
our_stack.push("3")
our_stack.push("99")
our_stack.push("9")
print our_stack.peek()
print our_stack.pop()
print our_stack.peek()
print our_stack.pop()
```

Performance

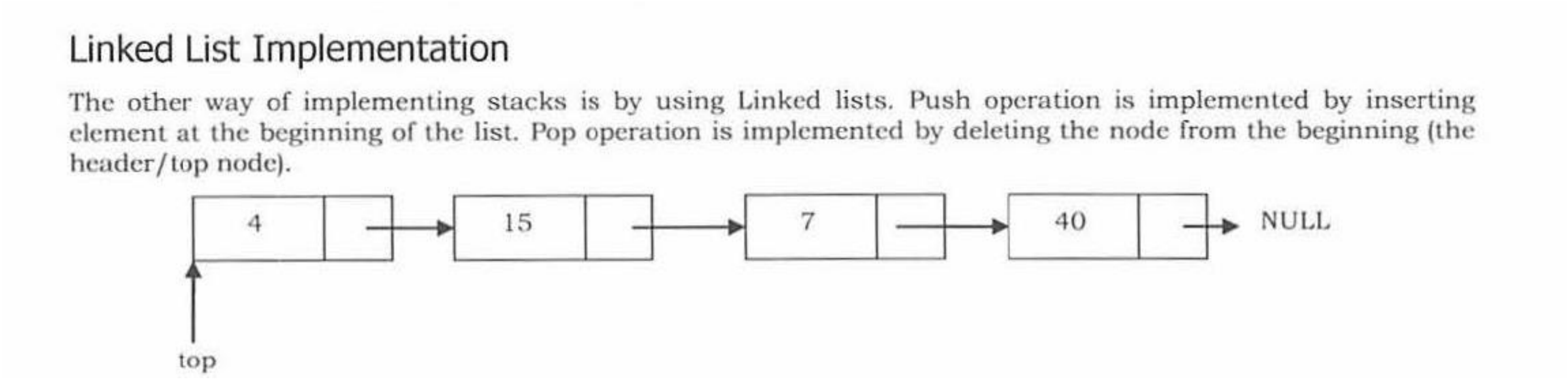


Performance

Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1))
Time Complexity of IsFullStack()	O(1)
Time Complexity of DeleteStack()	O(1)

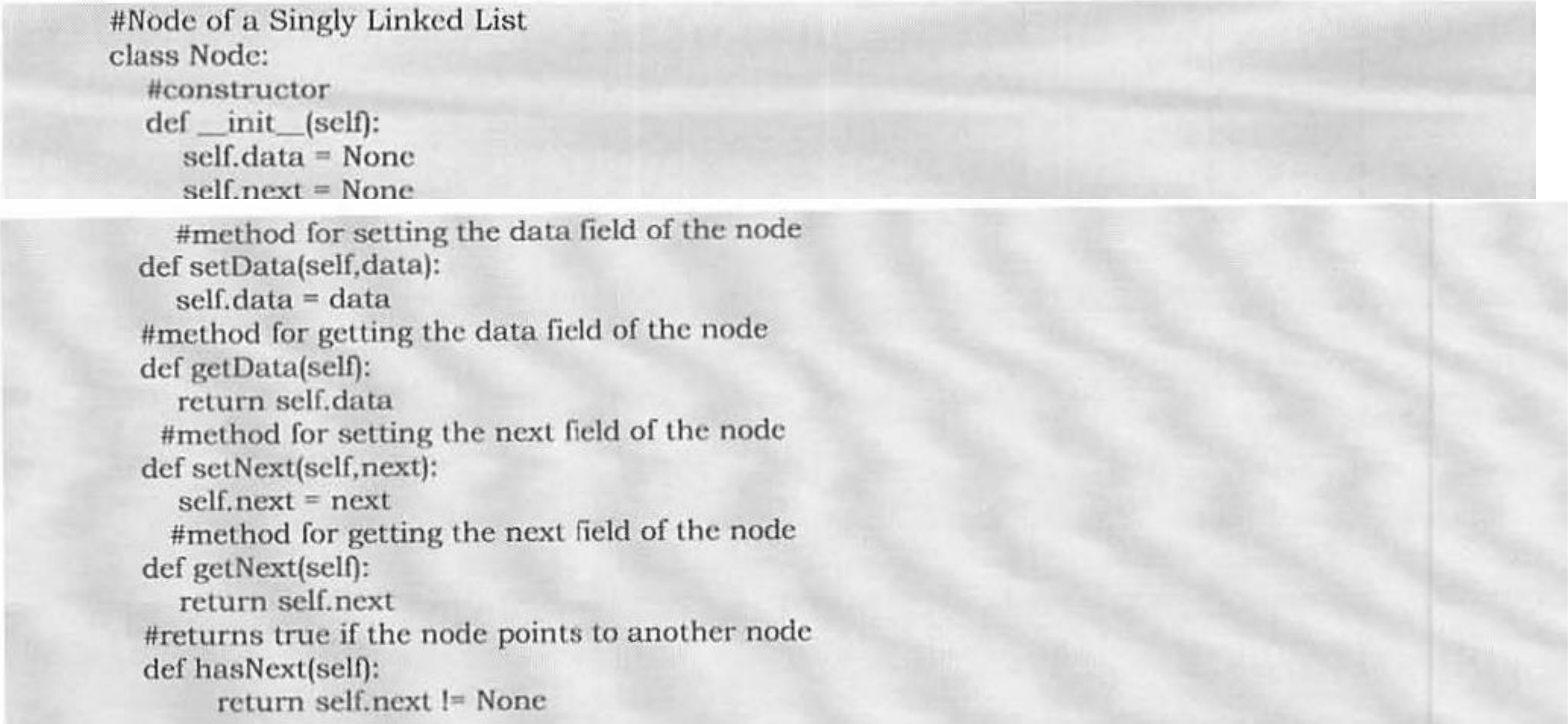
Note: Too many doublings may cause memory overflow exception.



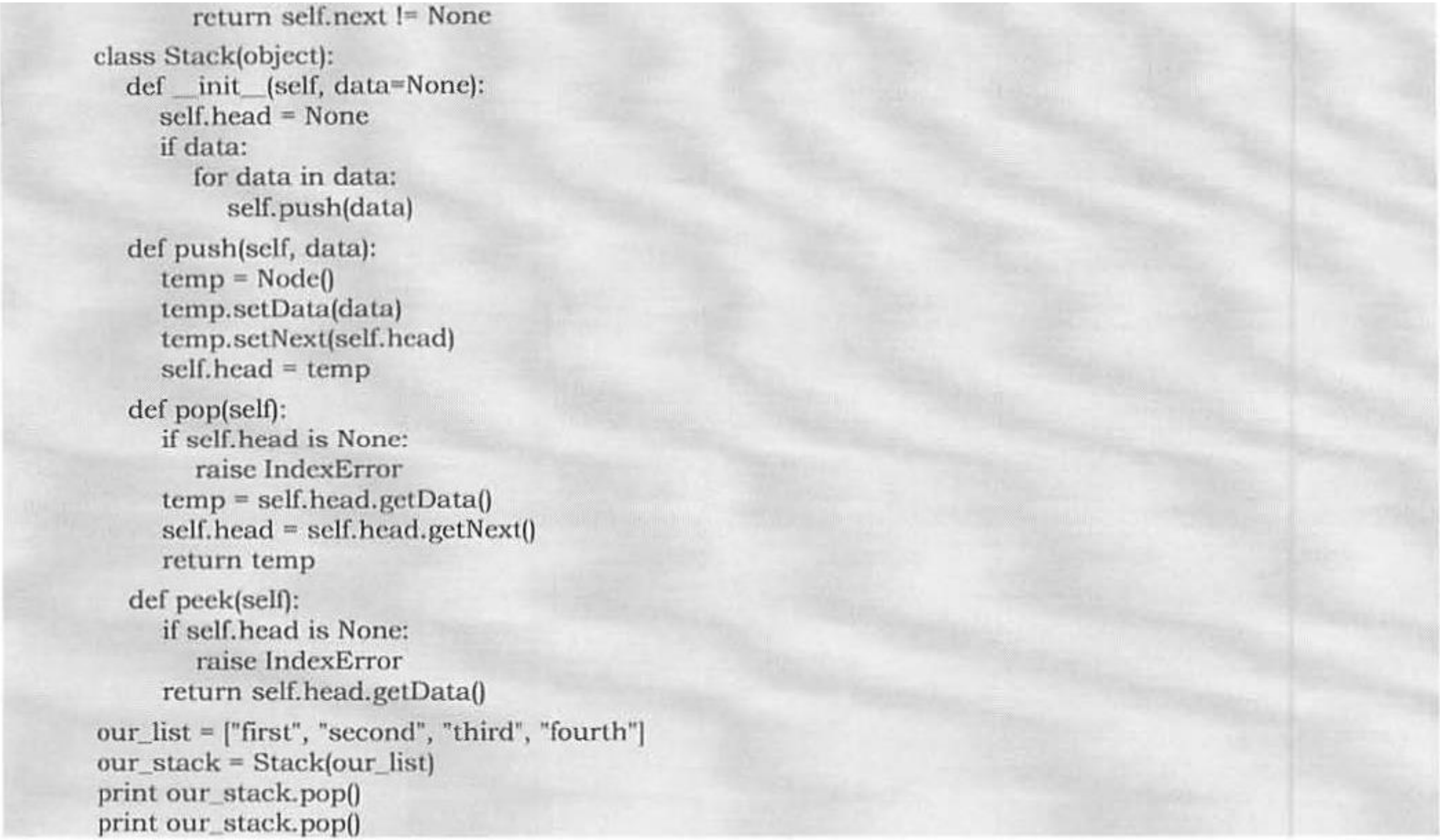
Linked List Implementation

The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).

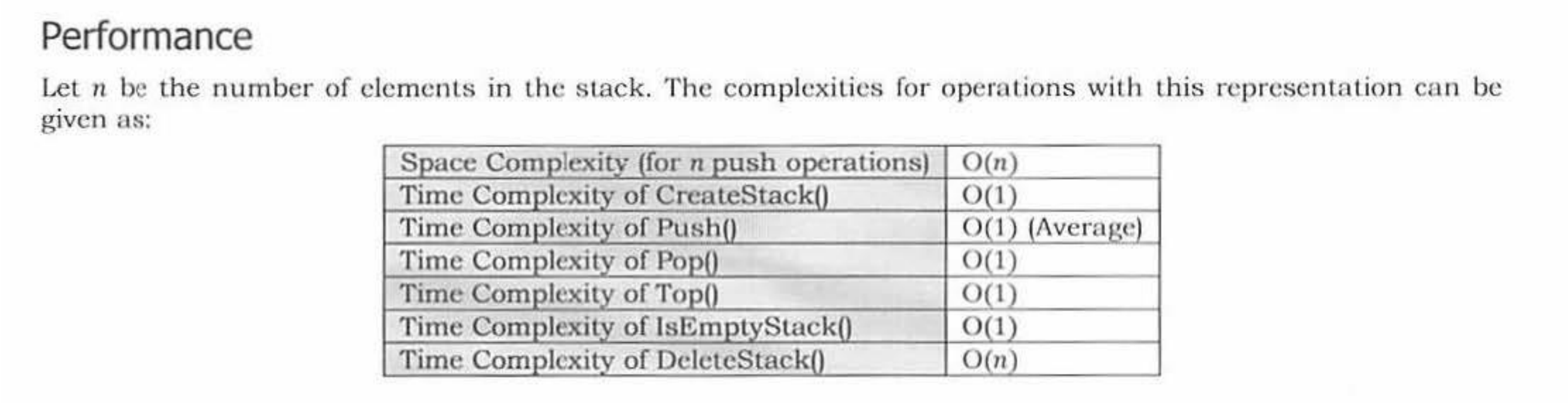
4
15
7
40
NULL
top



```
#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
```



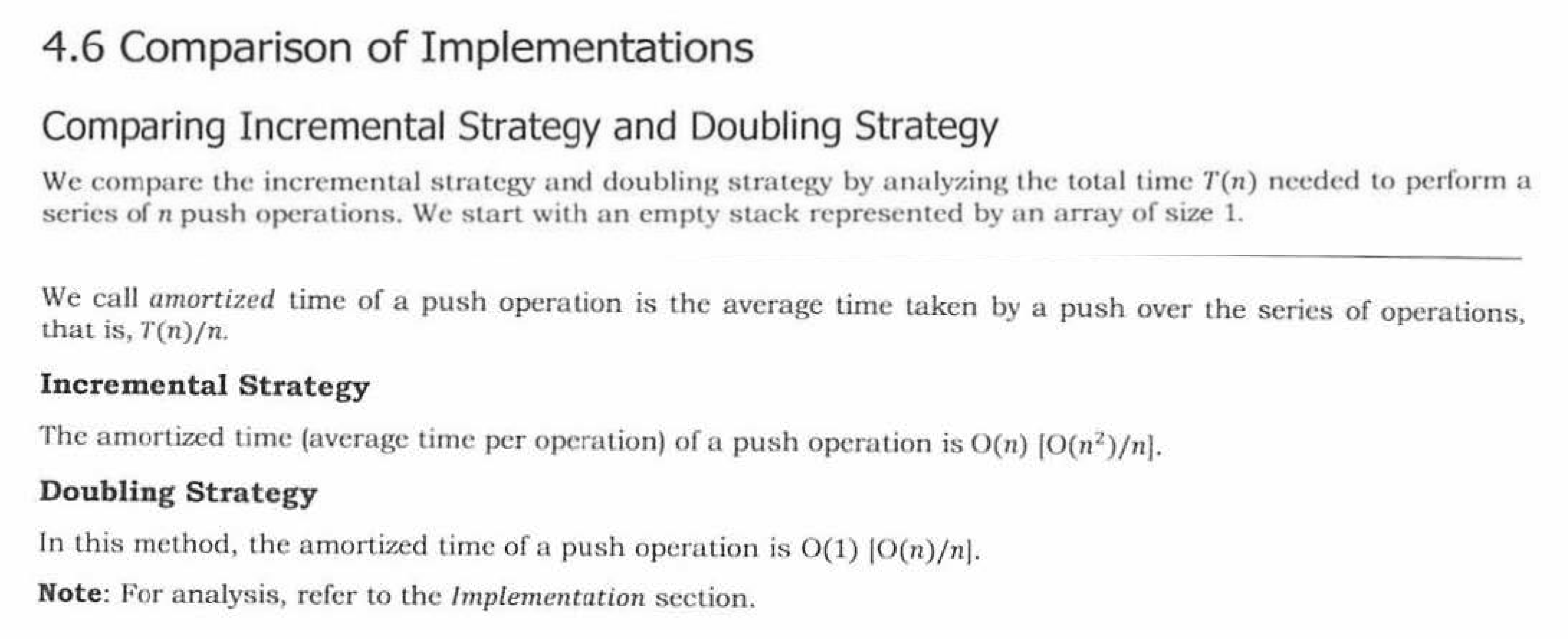
```
        return self.next != None

class Stack(object):
    def __init__(self, data=None):
        self.head = None
        if data:
            for data in data:
                self.push(data)

    def push(self, data):
        temp = Node()
        temp.setData(data)
        temp.setNext(self.head)
        self.head = temp

    def pop(self):
        if self.head is None:
            raise IndexError
        temp = self.head.getData()
        self.head = self.head.getNext()
        return temp

    def peek(self):
        if self.head is None:
            raise IndexError
        return self.head.getData()

our_list = ["first", "second", "third", "fourth"]
our_stack = Stack(our_list)
print our_stack.pop()
print our_stack.pop()
```



Performance

Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	$O(1)$
Time Complexity of Push()	$O(1)$ (Average)
Time Complexity of Pop()	$O(1)$
Time Complexity of Top()	$O(1)$
Time Complexity of IsEmptyStack()	$O(1)$
Time Complexity of DeleteStack()	$O(n)$



4.6 Comparison of Implementations

Comparing Incremental Strategy and Doubling Strategy

We compare the incremental strategy and doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations. We start with an empty stack represented by an array of size 1.

We call *amortized* time of a push operation is the average time taken by a push over the series of operations, that is, $T(n)/n$.

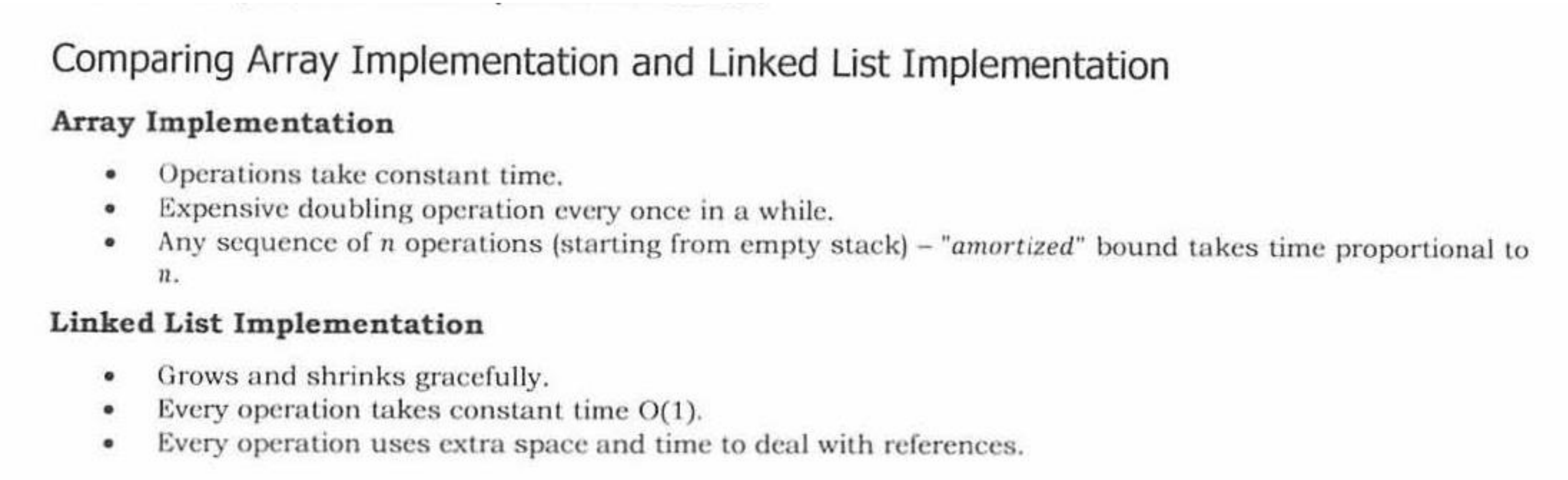
Incremental Strategy

The amortized time (average time per operation) of a push operation is $O(n)$ $[O(n^2)/n]$.

Doubling Strategy

In this method, the amortized time of a push operation is $O(1)$ $[O(n)/n]$.

Note: For analysis, refer to the *Implementation* section.



Comparing Array Implementation and Linked List Implementation

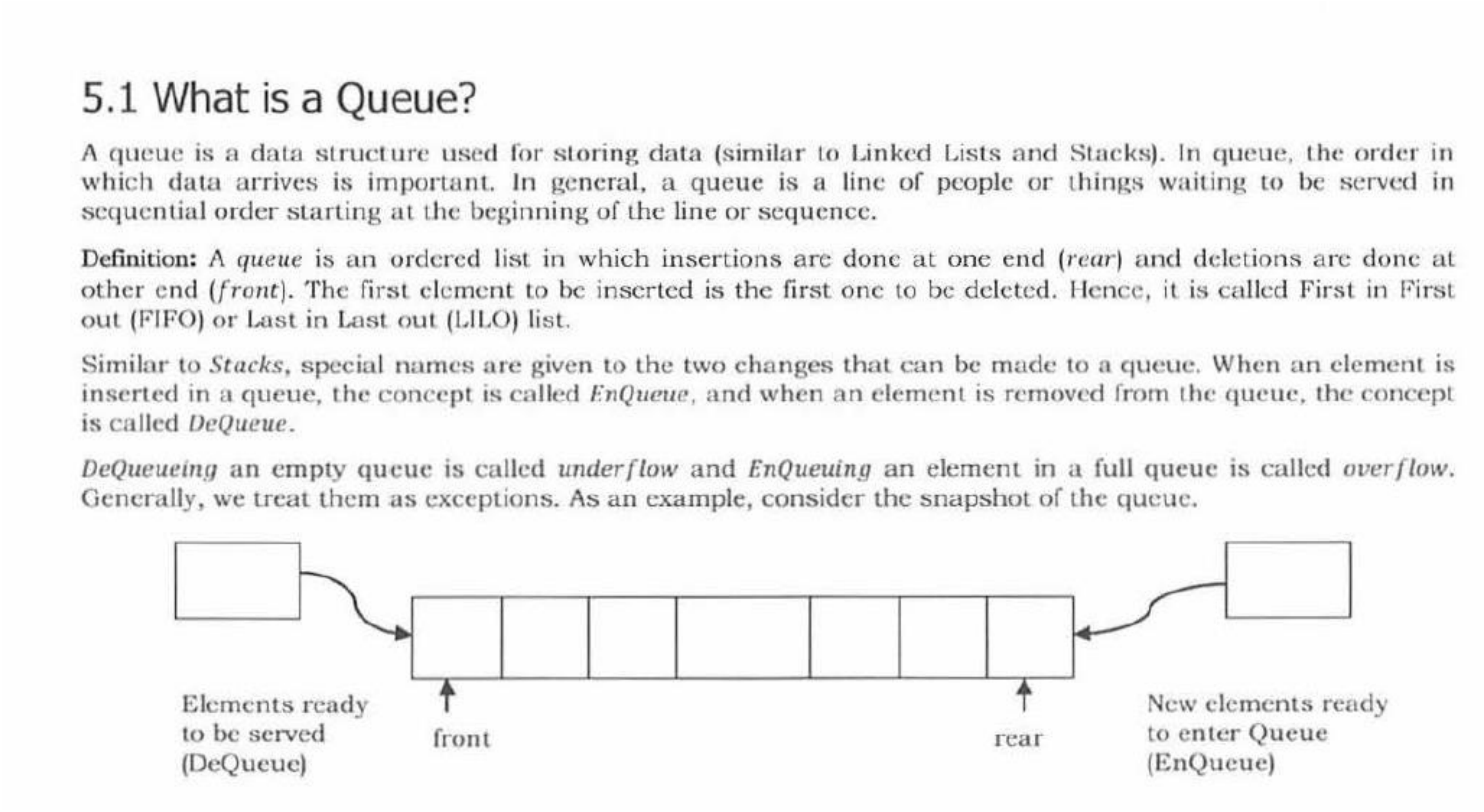
Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from empty stack) – "*amortized*" bound takes time proportional to n.

Linked List Implementation

- Grows and shrinks gracefully.
- Every operation takes constant time O(1).
- Every operation uses extra space and time to deal with references.

Queue



5.1 What is a Queue?

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Definition: A *queue* is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.

Similar to *Stacks*, special names are given to the two changes that can be made to a queue. When an element is inserted in a queue, the concept is called *EnQueue*, and when an element is removed from the queue, the concept is called *DeQueue*.

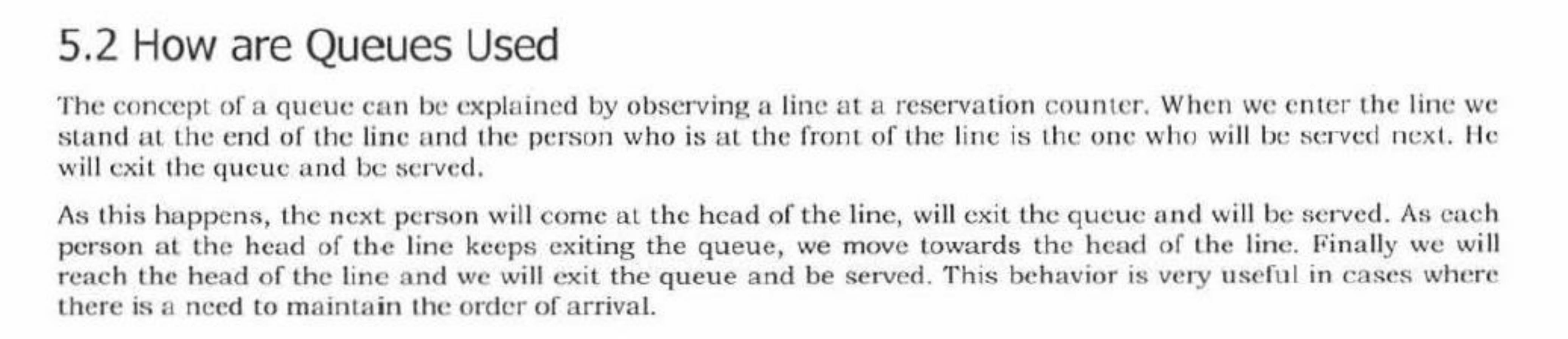
DeQueueing an empty queue is called *underflow* and *EnQueuing* an element in a full queue is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of the queue.

Elements ready to be served (DeQueue)

front

rear

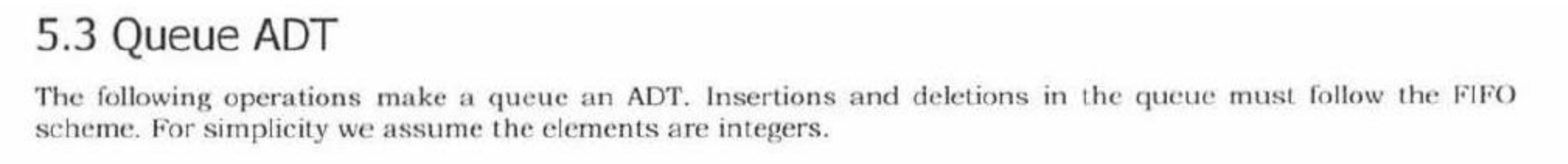
New elements ready to enter Queue (EnQueue)



5.2 How are Queues Used

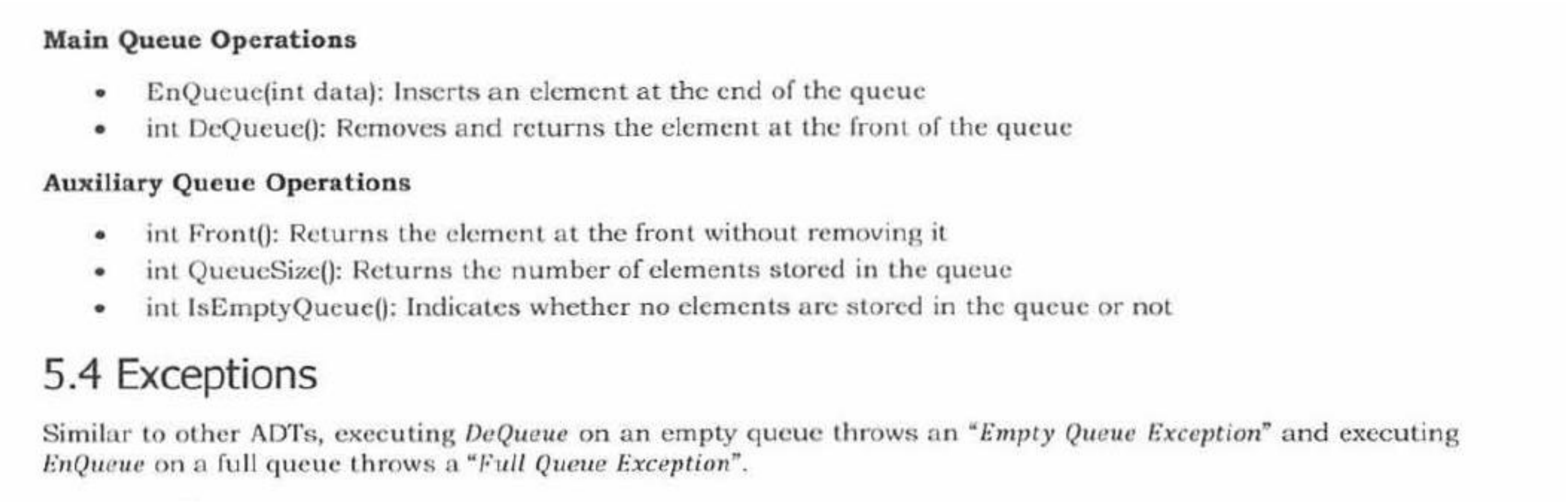
The concept of a queue can be explained by observing a line at a reservation counter. When we enter the line we stand at the end of the line and the person who is at the front of the line is the one who will be served next. He will exit the queue and be served.

As this happens, the next person will come at the head of the line, will exit the queue and will be served. As each person at the head of the line keeps exiting the queue, we move towards the head of the line. Finally we will reach the head of the line and we will exit the queue and be served. This behavior is very useful in cases where there is a need to maintain the order of arrival.



5.3 Queue ADT

The following operations make a queue an ADT. Insertions and deletions in the queue must follow the FIFO scheme. For simplicity we assume the elements are integers.



Main Queue Operations

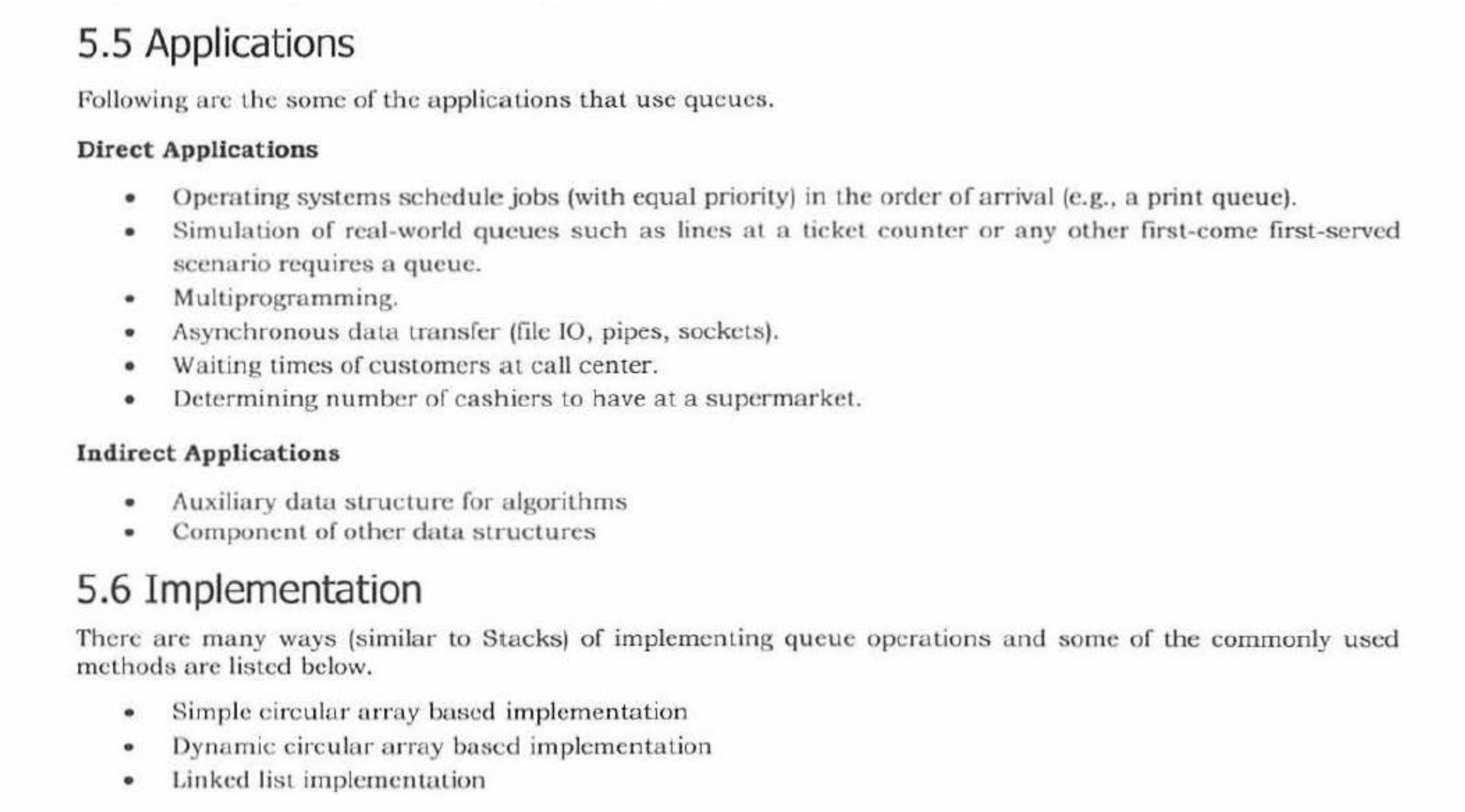
- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

Auxiliary Queue Operations

- int Front(): Returns the element at the front without removing it
- int QueueSize(): Returns the number of elements stored in the queue
- int IsEmptyQueue(): Indicates whether no elements are stored in the queue or not

5.4 Exceptions

Similar to other ADTs, executing *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and executing *EnQueue* on a full queue throws a "*Full Queue Exception*".



5.5 Applications

Following are the some of the applications that use queues.

Direct Applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

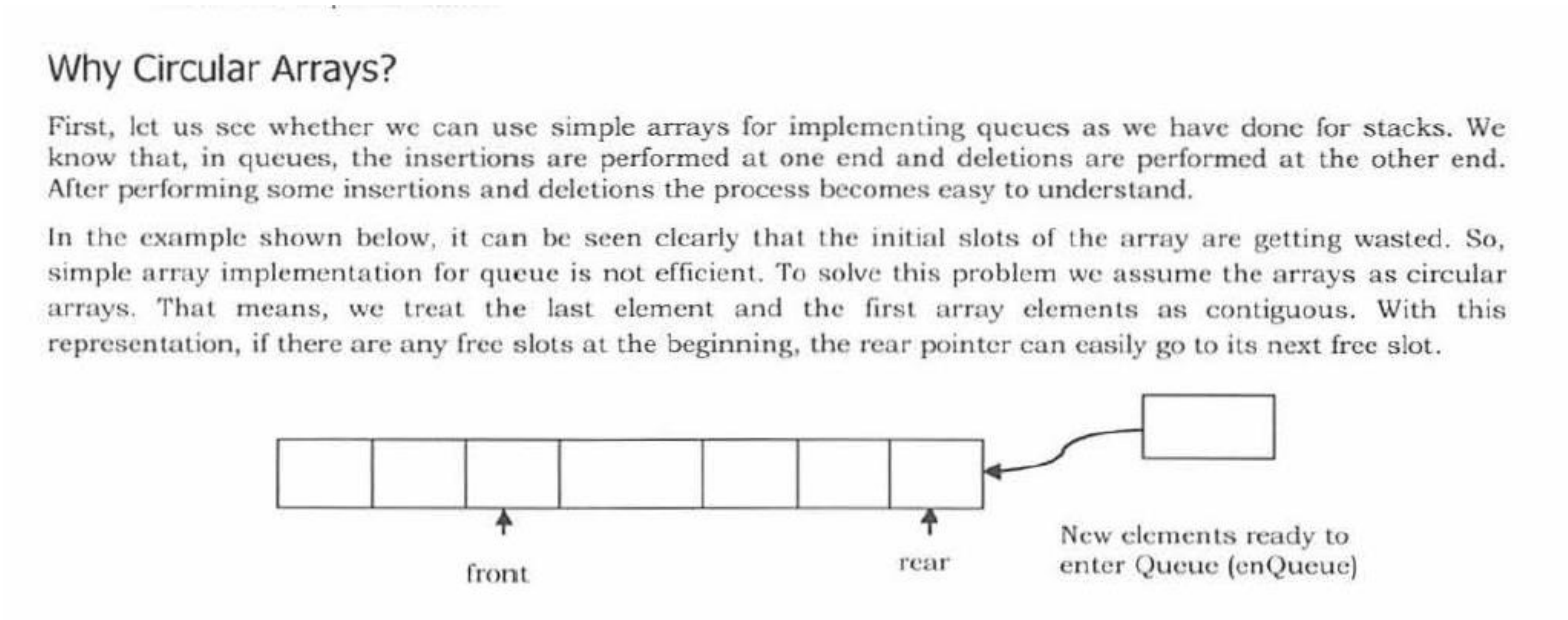
Indirect Applications

- Auxiliary data structure for algorithms
- Component of other data structures

5.6 Implementation

There are many ways (similar to Stacks) of implementing queue operations and some of the commonly used methods are listed below.

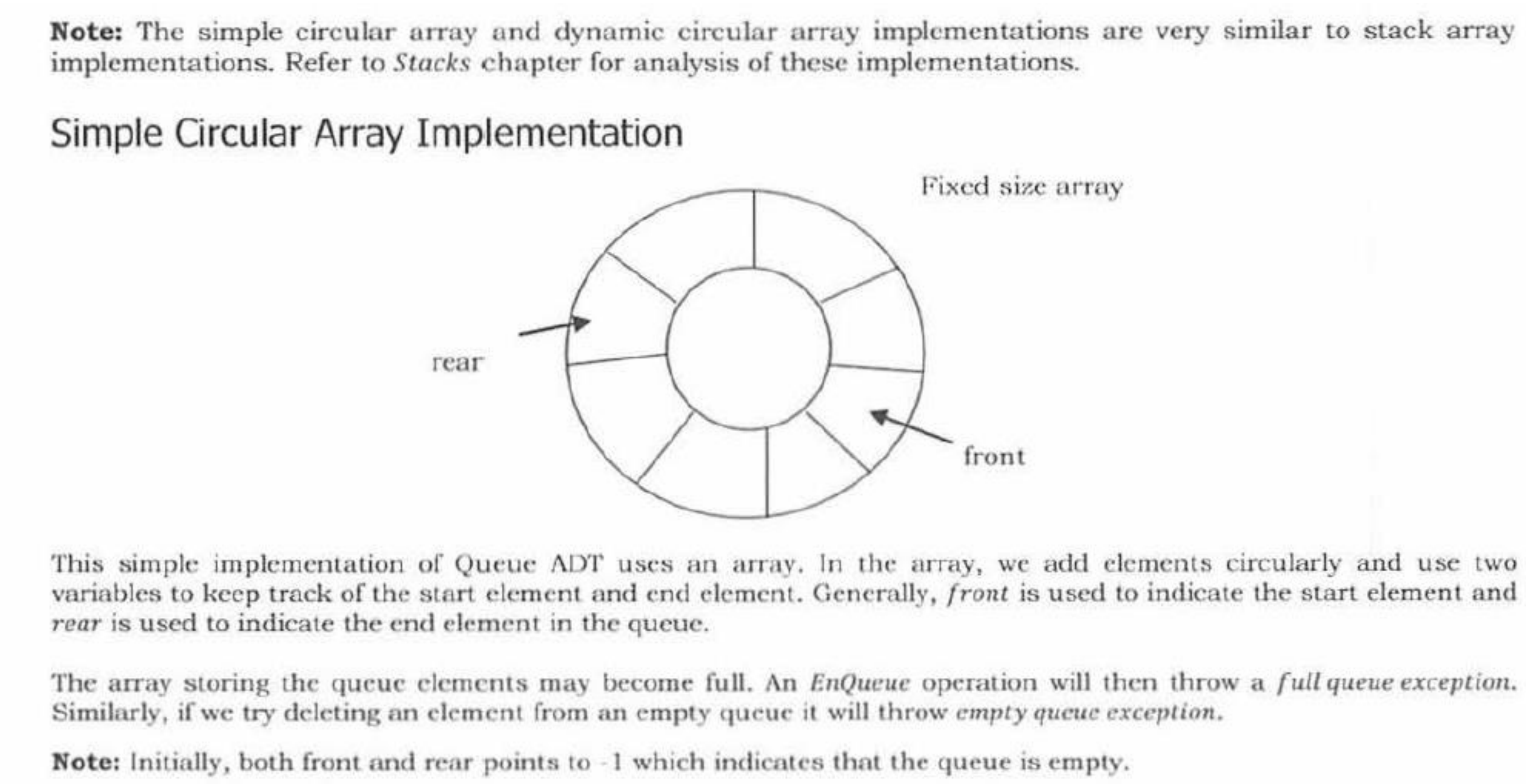
- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked list implementation



Why Circular Arrays?

First, let us see whether we can use simple arrays for implementing queues as we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at the other end. After performing some insertions and deletions the process becomes easy to understand.

In the example shown below, it can be seen clearly that the initial slots of the array are getting wasted. So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat the last element and the first array elements as contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slot.



Note: The simple circular array and dynamic circular array implementations are very similar to stack array implementations. Refer to *Stacks* chapter for analysis of these implementations.

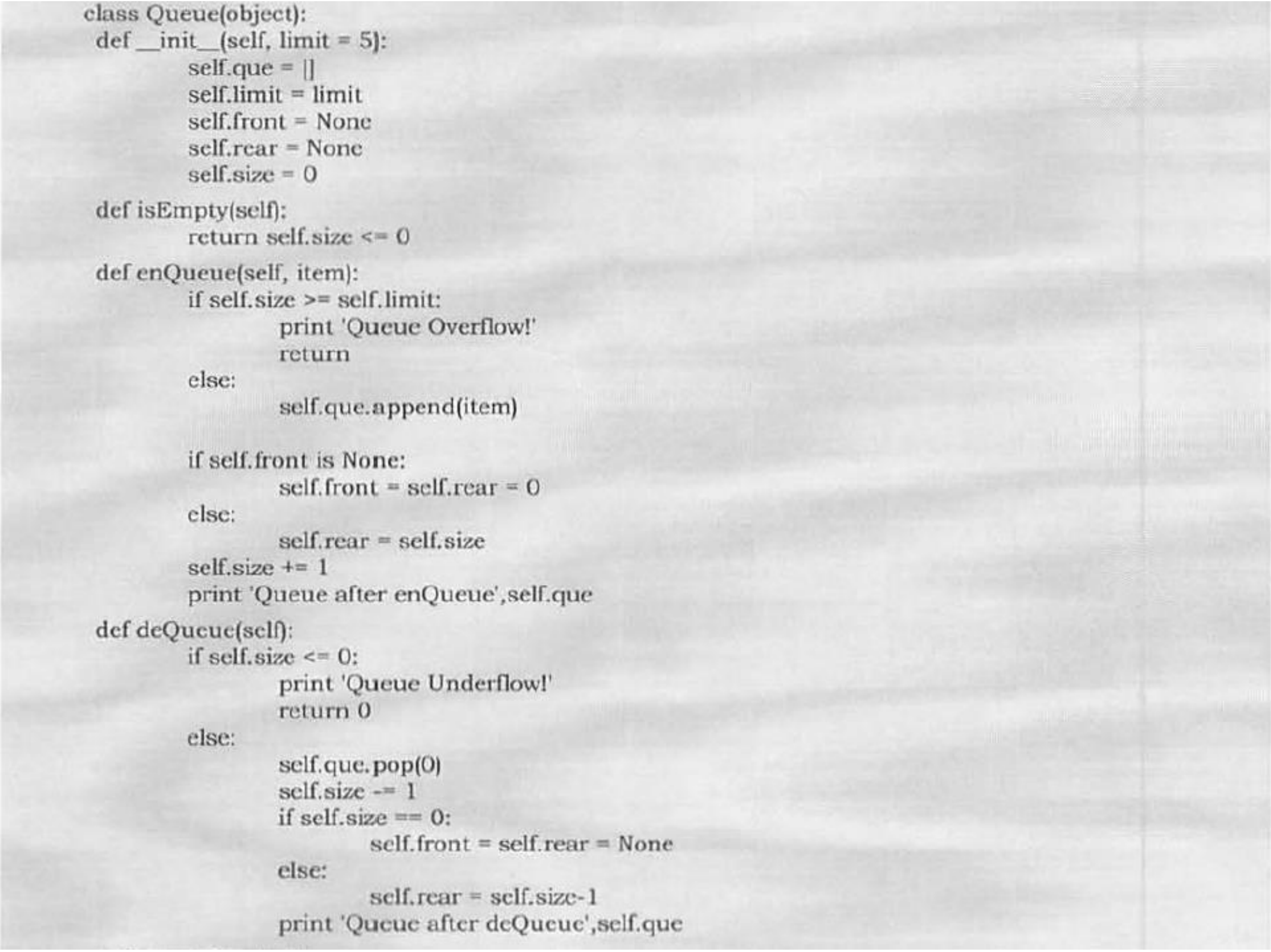
Simple Circular Array Implementation

Fixed size array
rear
front

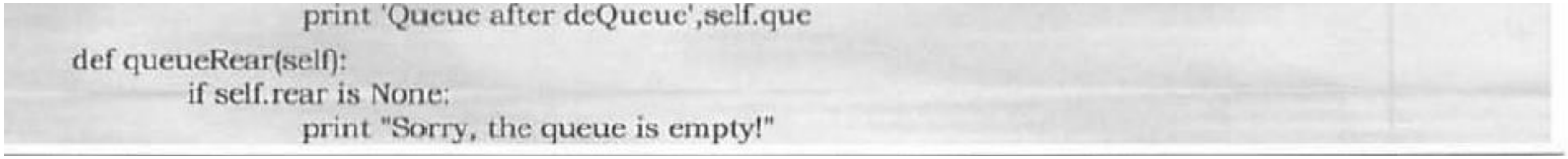
This simple implementation of Queue ADT uses an array. In the array, we add elements circularly and use two variables to keep track of the start element and end element. Generally, *front* is used to indicate the start element and *rear* is used to indicate the end element in the queue.

The array storing the queue elements may become full. An *EnQueue* operation will then throw a *full queue exception*. Similarly, if we try deleting an element from an empty queue it will throw *empty queue exception*.

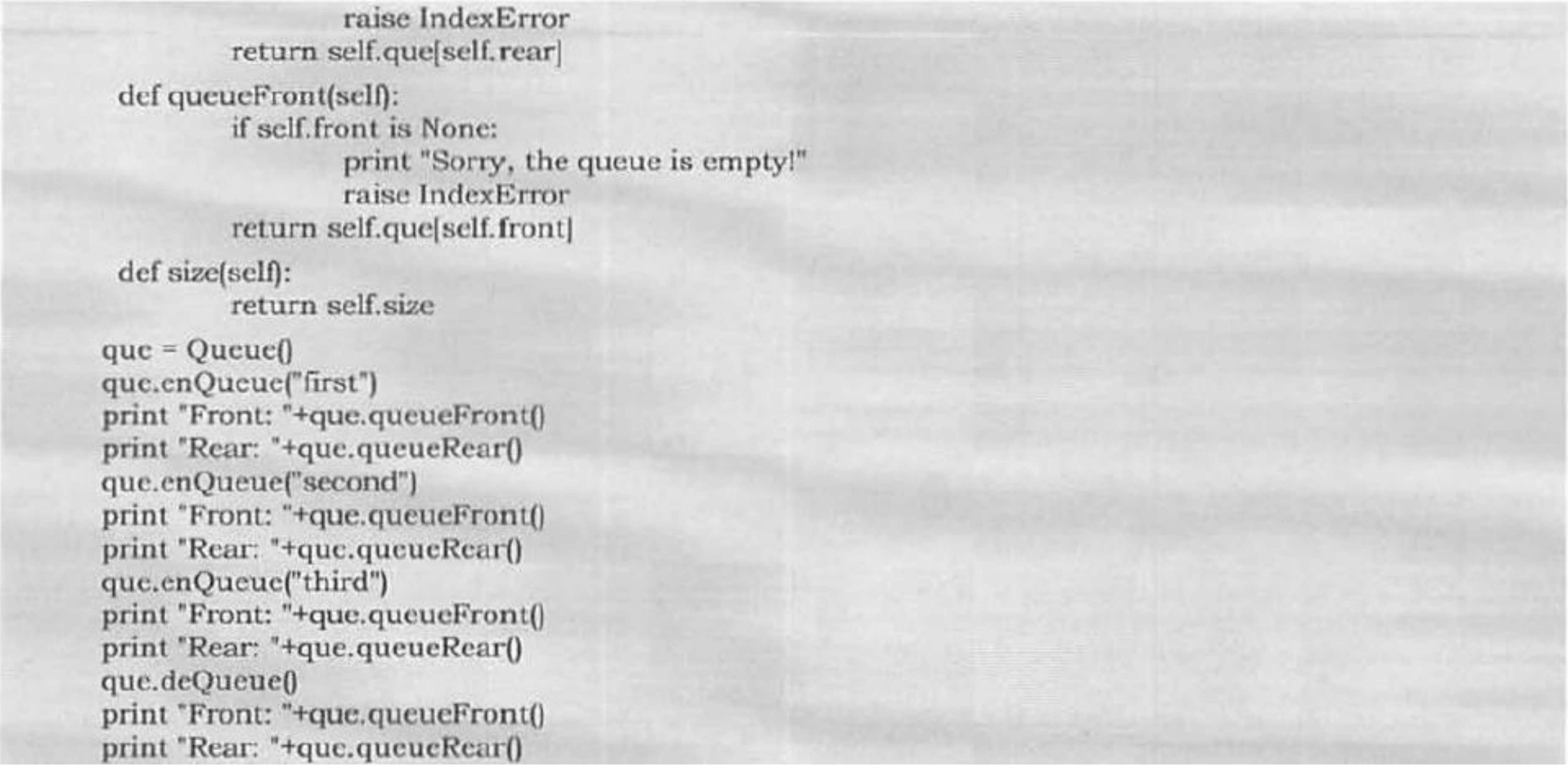
Note: Initially, both front and rear points to -1 which indicates that the queue is empty.



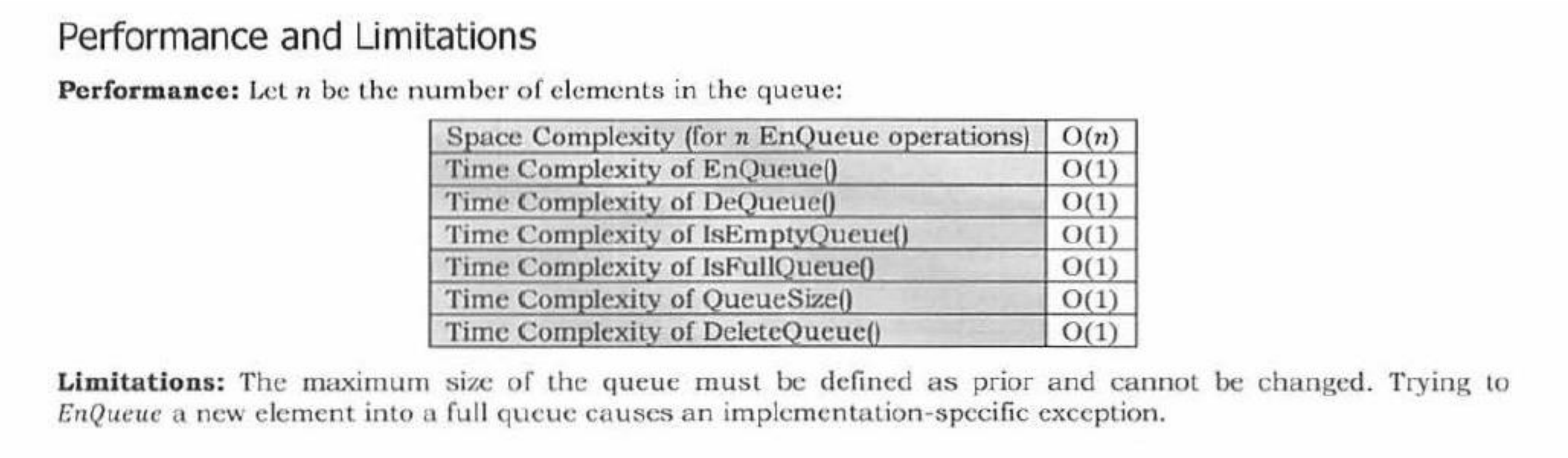
```
class Queue(object):
 def __init__(self, limit = 5):
         self.que = []
         self.limit = limit
         self.front = None
         self.rear = None
         self.size = 0
 def isEmpty(self):
         return self.size <= 0
 def enQueue(self, item):
         if self.size >= self.limit:
                 print 'Queue Overflow!'
                 return
         else:
                 self.que.append(item)

         if self.front is None:
                 self.front = self.rear = 0
         else:
                 self.rear = self.size
         self.size += 1
         print 'Queue after enQueue',self.que
 def deQueue(self):
         if self.size <= 0:
                 print 'Queue Underflow!'
                 return 0
         else:
                 self.que.pop(0)
                 self.size -= 1
                 if self.size == 0:
                         self.front = self.rear = None
                 else:
                         self.rear = self.size-1
                 print 'Queue after deQueue',self.que
```



```
            print 'Queue after deQueue',self.que
    def queueRear(self):
        if self.rear is None:
            print "Sorry, the queue is empty!"
```



```
            raise IndexError
        return self.que[self.rear]
    def queueFront(self):
        if self.front is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.que[self.front]
    def size(self):
        return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```



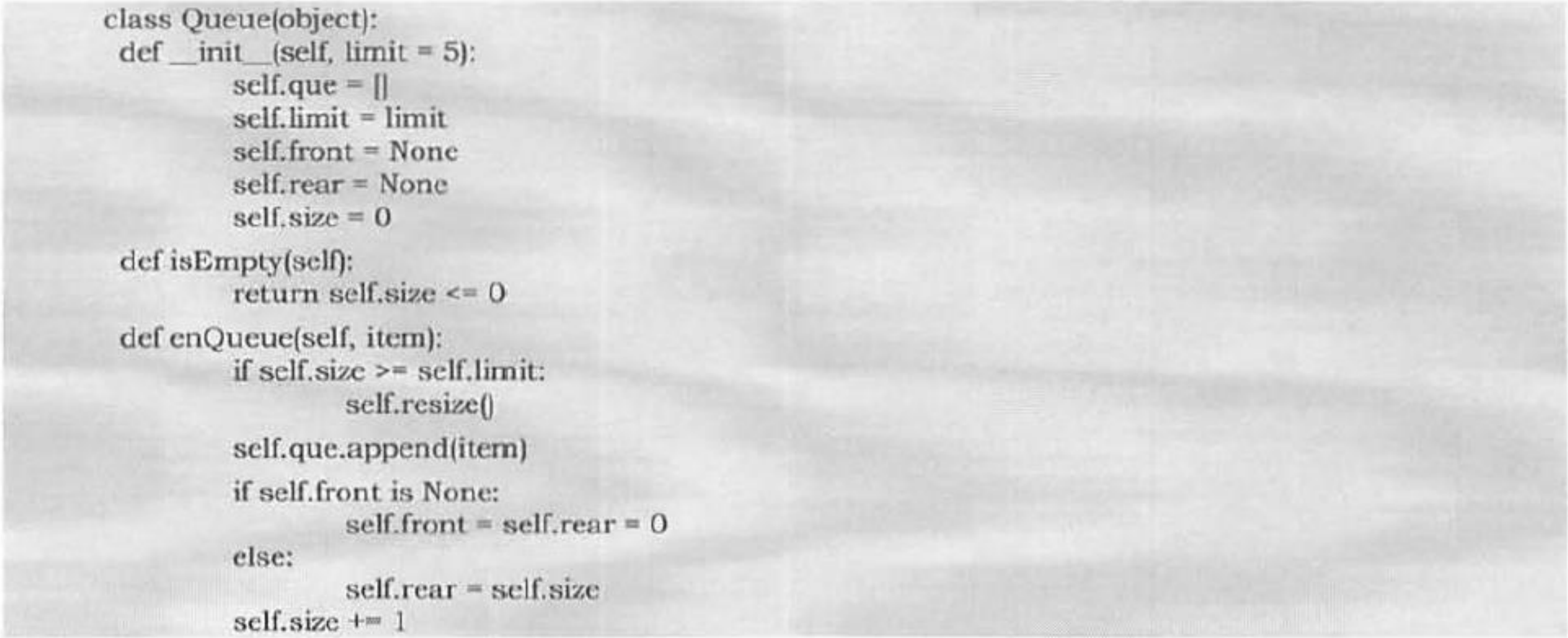
Performance and Limitations

Performance: Let n be the number of elements in the queue:

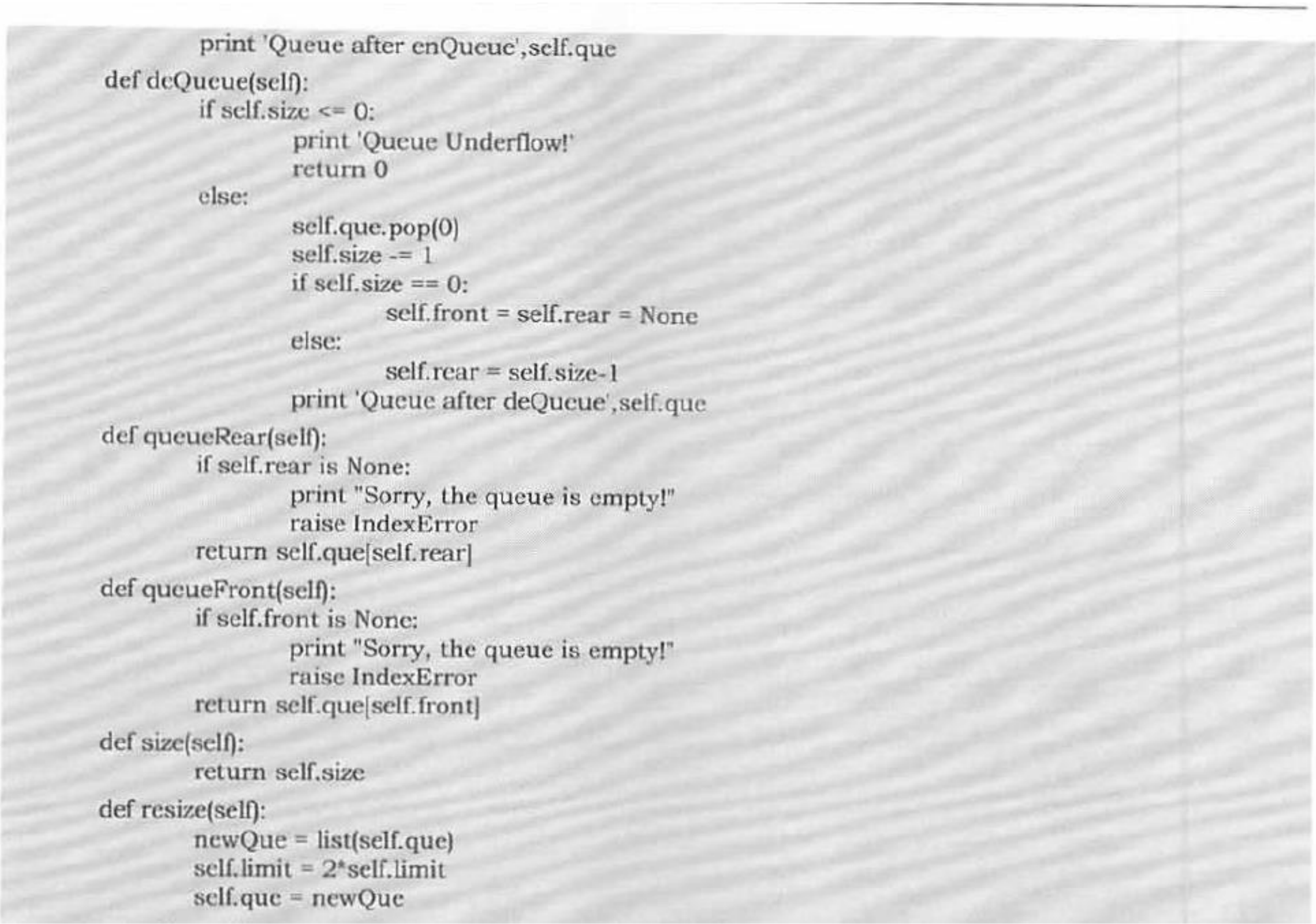
Space Complexity (for n EnQueue operations)	$O(n)$
Time Complexity of EnQueue()	$O(1)$
Time Complexity of DeQueue()	$O(1)$
Time Complexity of IsEmptyQueue()	$O(1)$
Time Complexity of IsFullQueue()	$O(1)$
Time Complexity of QueueSize()	$O(1)$
Time Complexity of DeleteQueue()	$O(1)$

Limitations: The maximum size of the queue must be defined as prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.

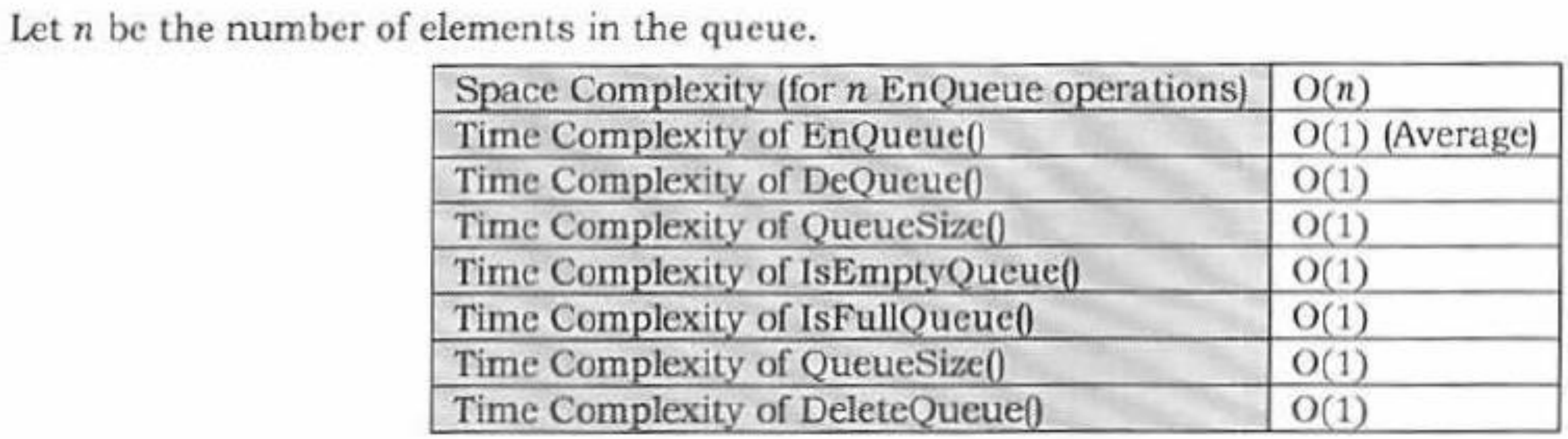
Dynamic Circular Array Implementation



```
class Queue(object):
  def __init__(self, limit = 5):
              self.que = []
              self.limit = limit
              self.front = None
              self.rear = None
              self.size = 0
  def isEmpty(self):
              return self.size <= 0
  def enQueue(self, item):
              if self.size >= self.limit:
                        self.resize()
              self.que.append(item)
              if self.front is None:
                        self.front = self.rear = 0
              else:
                        self.rear = self.size
              self.size += 1
```



```
        print 'Queue after enQueue',self.que
    def deQueue(self):
        if self.size <= 0:
            print 'Queue Underflow!'
            return 0
        else:
            self.que.pop(0)
            self.size -= 1
            if self.size == 0:
                self.front = self.rear = None
            else:
                self.rear = self.size-1
            print 'Queue after deQueue',self.que
    def queueRear(self):
        if self.rear is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.que[self.rear]
    def queueFront(self):
        if self.front is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.que[self.front]
    def size(self):
        return self.size
    def resize(self):
        newQue = list(self.que)
        self.limit = 2*self.limit
        self.que = newQue
```

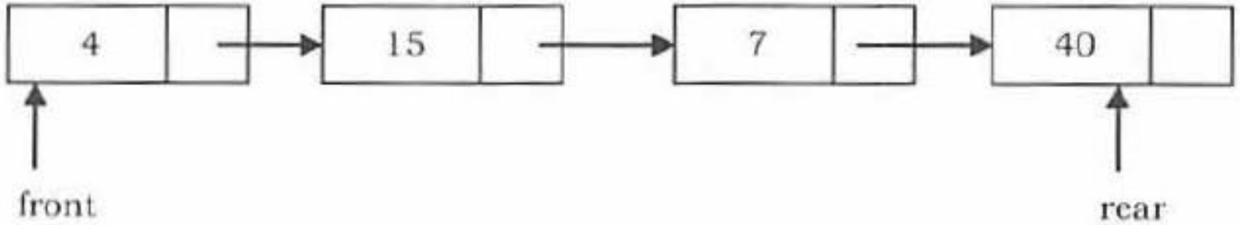


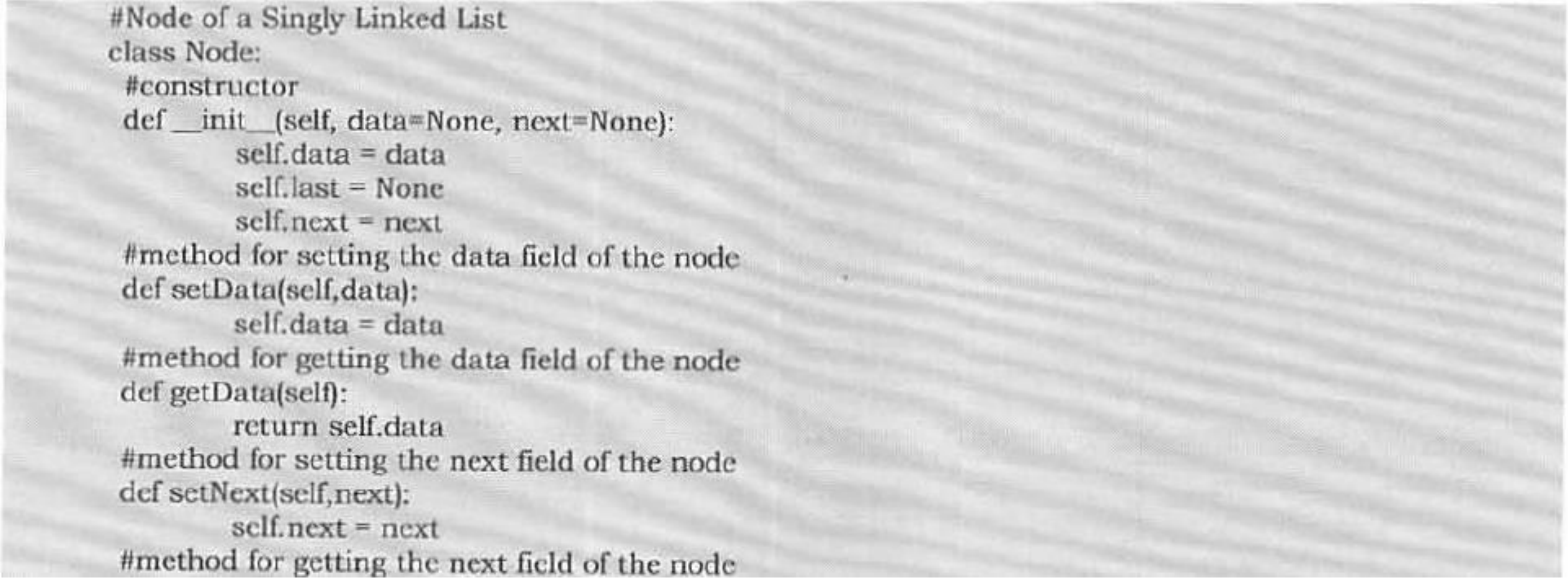
Let n be the number of elements in the queue.

Space Complexity (for n EnQueue operations)	$O(n)$
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

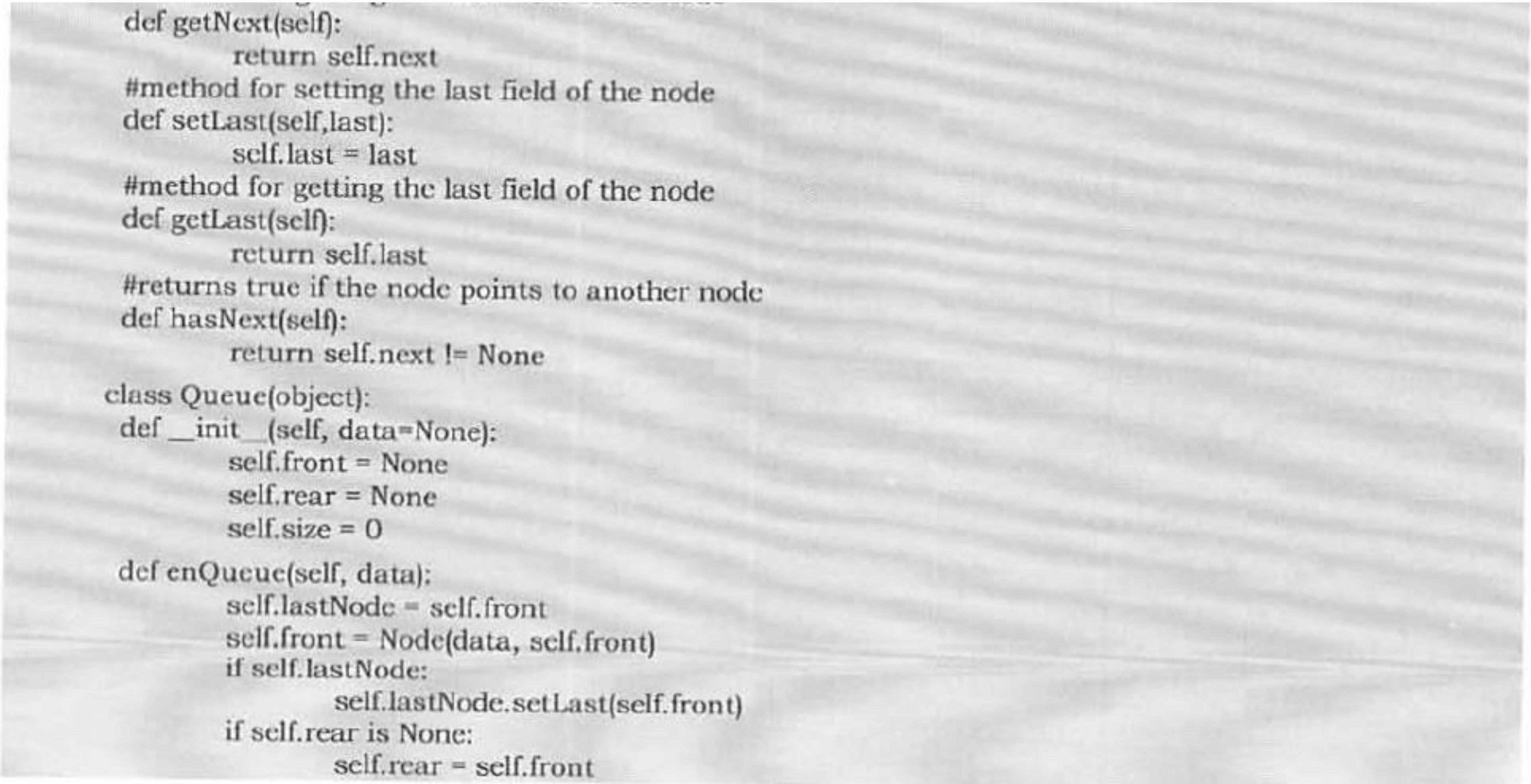
Linked List Implementation

Another way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting an element at the end of the list. *DeQueue* operation is implemented by deleting an element from the beginning of the list.

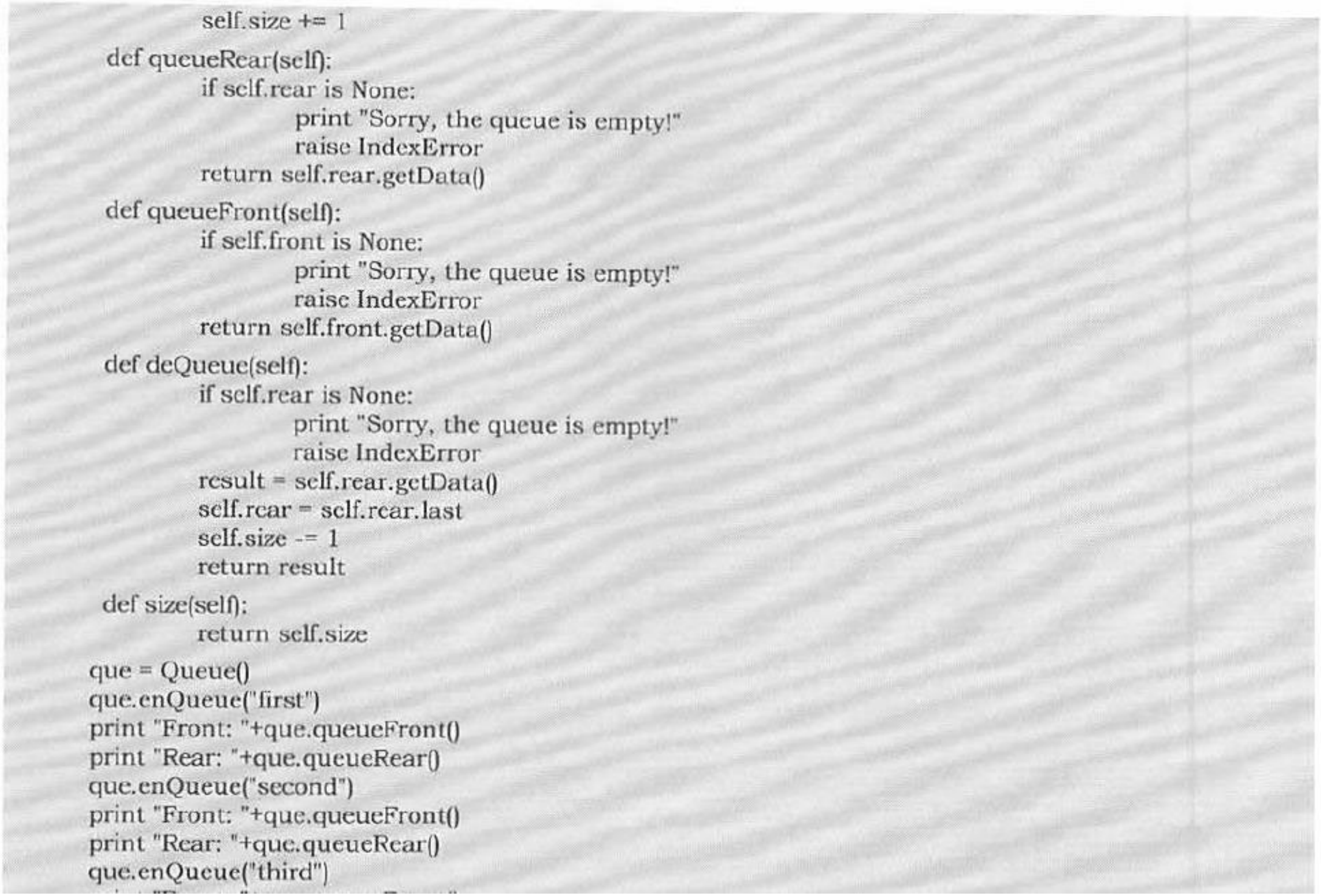




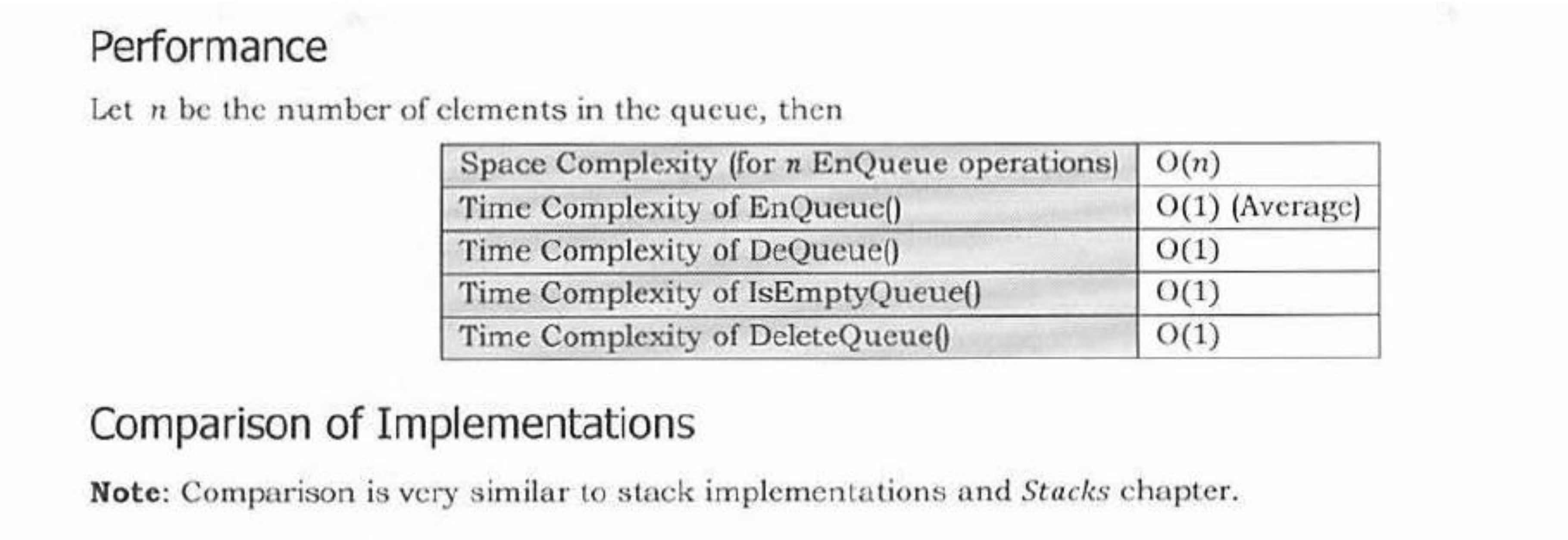
```
#Node of a Singly Linked List
class Node:
  #constructor
  def __init__(self, data=None, next=None):
          self.data = data
          self.last = None
          self.next = next
  #method for setting the data field of the node
  def setData(self,data):
          self.data = data
  #method for 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
    #method for setting the last field of the node
    def setLast(self,last):
            self.last = last
    #method for getting the last field of the node
    def getLast(self):
            return self.last
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
class Queue(object):
    def __init__(self, data=None):
            self.front = None
            self.rear = None
            self.size = 0
    def enQueue(self, data):
            self.lastNode = self.front
            self.front = Node(data, self.front)
            if self.lastNode:
                    self.lastNode.setLast(self.front)
            if self.rear is None:
                    self.rear = self.front
```



```
        self.size += 1
    def queueRear(self):
        if self.rear is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.rear.getData()
    def queueFront(self):
        if self.front is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.front.getData()
    def deQueue(self):
        if self.rear is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        result = self.rear.getData()
        self.rear = self.rear.last
        self.size -= 1
        return result
    def size(self):
        return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
```



Performance

Let n be the number of elements in the queue, then

Space Complexity (for n EnQueue operations)	$O(n)$
Time Complexity of EnQueue()	$O(1)$ (Average)
Time Complexity of DeQueue()	$O(1)$
Time Complexity of IsEmptyQueue()	$O(1)$
Time Complexity of DeleteQueue()	$O(1)$

Comparison of Implementations

Note: Comparison is very similar to stack implementations and *Stacks* chapter.

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 is done by Sofia Wasquez