Data Structures

PoPI

Motivation

- All the algorithms that solve CS-related problems use data structures
- Example:
  - Given a sequence of numbers, print the average

    Input: 5 2 3 4

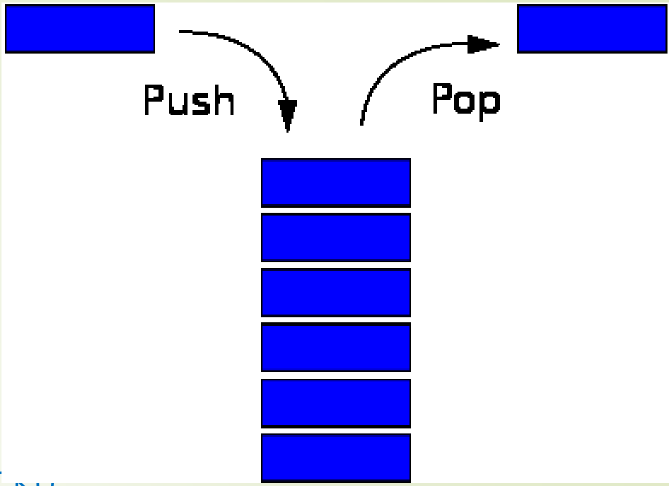
    ```
    seq_str = input().split()
    sum = 0
    count = 0
    for num_str in seq_str:
        sum += int(num_str)
        count += 1
    print(sum/count)
    ```

  - We use two integers (primitive data structures) and list
- Some data structures are of fundamental importance since used in many programs:
  - Stacks Queues Lists

Stacks and Queues

- These data structures are helpful to solve many CS-problems
- They are at the foundations: operating systems, compilers, communication protocols etc.
- Both store sequence of items (but give access to them in deferent ways)
- Can be implemented in multiples ways:
  - We will work with classes Stack and Queue providing required methods
  - We will not consider their implementation (in this lecture)

Stack: Basics



- Adding, removing and accessing elements can be done via the following operations only (for efficiency)

```
class Stack:
    def push(self, x):
        '''adds an element x at top'''
    def pop(self):
        '''removes the element at top'''
    def peek(self):
        '''returns the element of top'''
    def is_empty(self):
        '''returns True if stack is empty, otherwise False'''
```

- Example:

```
st1 = Stack()
print(st1.is_empty())      # True
st1.push('a')
st1.push('b')
st1.push('c')
print(st1.peek())          #c
st1.pop()
print(st1.peek())          #b
st1.pop()
print(st1.peek())          #a
print(st1.is_empty())      #False
```

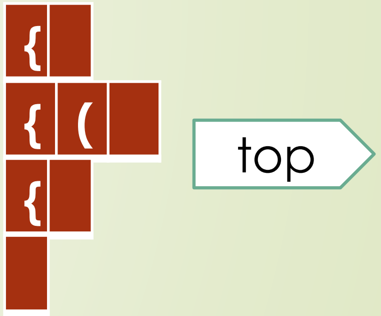
Stack: Motivation

- Consider the following problem:
  - Input: a string containing only symbols (,),{,}
  - Output: True of False depending on whether the input is correct
  - All of {}(), {()}, ({}()) are correct
  - All of)(, {), {(){} are not correct
  - Important in compilers!

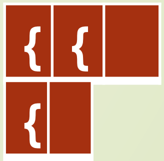
Stack: Motivation (cont.)

```
start with empty stack
for each symbol sym in the input:
        if sym is an opening symbol:
                stack.push(sym)
        else:        #sym is a closing symbol
                if stack.peek() matches sym: stack.pop()
                else: print "Incorrect" and exit
if stack.empty(): "Correct"
else: "Incorrect"
```

- Let str = "{()}"
  - After the 1st iteration of for loop:
  - After the 2nd iteration of for loop:
  - After the 3rd iteration of for loop:
  - After the 4th iteration of for loop:
  - "Correct"



- Let str = "{{})"
  - After 2nd iteratation:
  - After 3rd iteratation:
  - After 4th iteration: "Incorrect"



Queue: Basics

- Adding, removing and accessing elements done via the following operations only:

def Queue:

def enqueue(self, x):

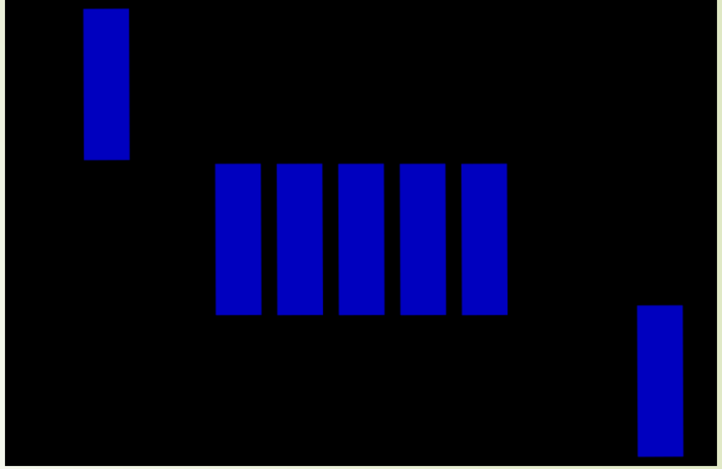
'''adds an element at back'''

def dequeue(self):

'''removes the element at front are returns it'''

def is_empty(self):

''' returns True if queue is empty, otherwise False'''



- Example:

```
q1 = Queue()
print(q.is_empty())          #True
q1.enqueue('a')
q1.enqueue('b')
q1.enqueue('c')
print(q1.dequeue())          #a
print(q1.dequeue())          #b
q1.enqueue('d')
print(q1.dequeue())          #c
```

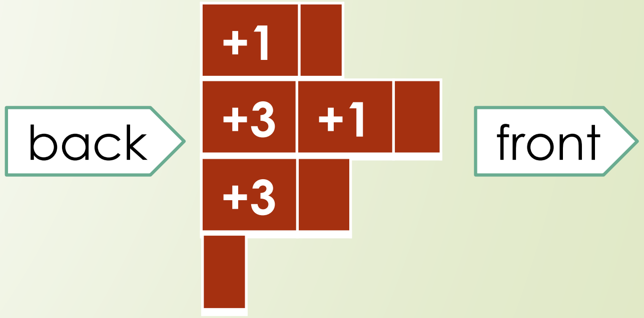
Queue: Motivation

- Consider the following problem:
  - Input: a sequence of negative and positive numbers representing a printer log:
    - a positive number +n at position i says that a job number n was sent to the printer at time i
    - a negative number -n at position i says that a job number n was completed by the printer at time i
  - Print: Correct of Incorrect depending on whether the log is complete and fair
  - Complete: each job that had been sent was completed
  - Fair: if job n had been sent earlier than job m, then n was completed earlier than m
  - +1 +3 +5 -1 -3 -5 is complete and fair
  - +1 +3 +5 -1 -3 is not complete but fair
  - +1 +3 +5 -3 -1 -5 is complete but not fair (because job 1 was sent earlier than 3 but 3 completed earlier)
  - +1 +3 +5 -3 -1 neither correct nor fair
  - Important in communication protocols (printers)

Queue: Example

```
start with empty queue
for each num in the input:
        if num is a positive number:
                queue.enqueue(num)
        else:       #num is negative
                if queue.is_empty(): print "Incorrect" and exit
                if queue.dequeue()+num != 0: print "Incorrect" and exit
if queue.is_empty(): print "Correct"
else: print "Incorrect"
```

- Let input be +1 +3 -1 -3
  - After the 1st iteration of for loop:
  - After the 2nd iteration of for loop:
  - After the 3rd iteration of for loop:
  - After the 4th iteration of for loop:



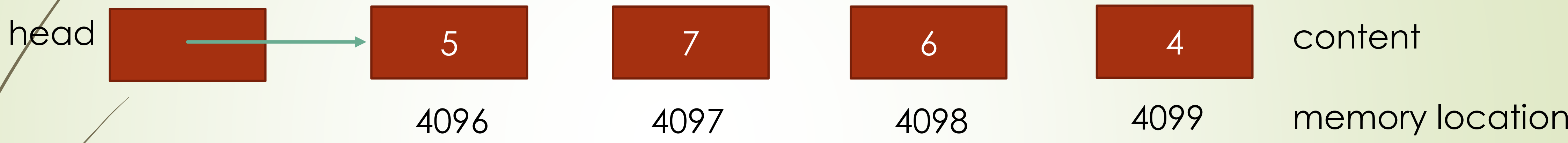
- Let input be +1 +3 -3 -1
  - After the 2nd iteration of for loop:

  - During the 3rd iteration of for loop: num = -3, dequeue =+1

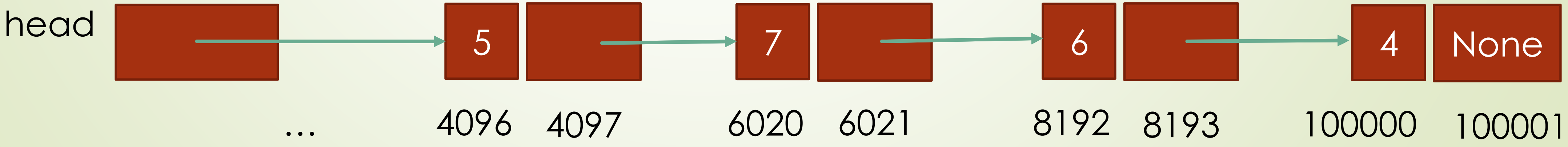
    "Incorrect"

Linked Lists: Motivation

- Take sequence [5, 7, 6, 4]. What is their placement in memory?
- Ideal situation: consecutive placement

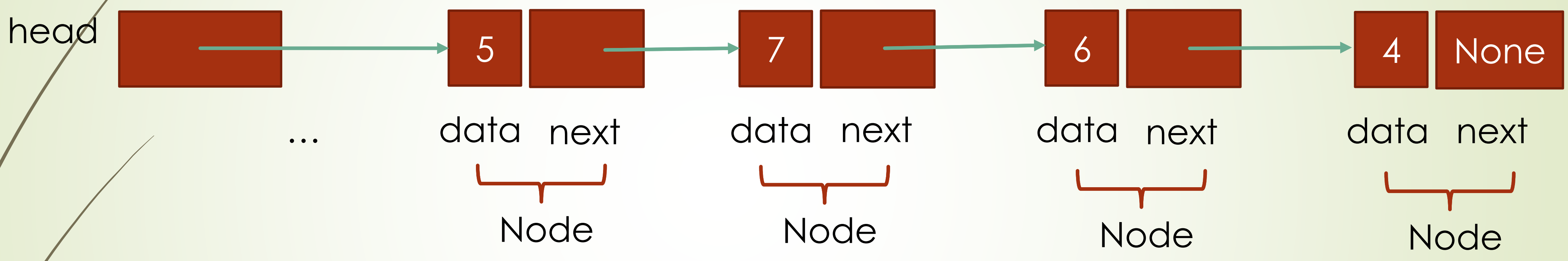


- Placing the elements of a list consecutively is not always possible
  - Such placement requires "reservation" and so a number of "rooms" has to be known in advance (before the program runs)
  - If we want to add elements to the sequence "as we go", we need linked lists
- More realistic approach:



Linked Lists

- Then, we need variables to refer to the data and next field of each node

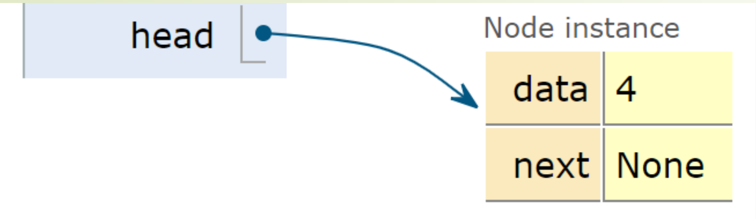


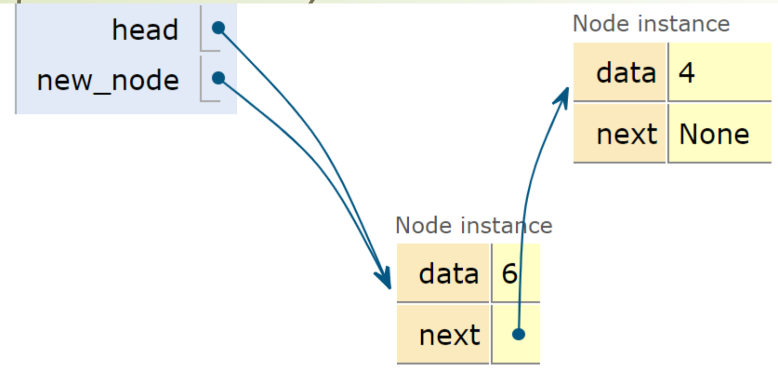
- We use a class Node with attributes data and next

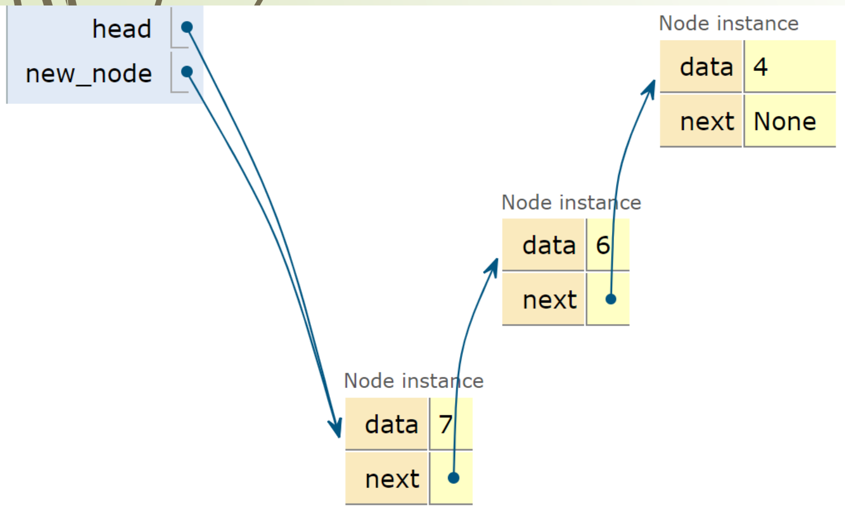
```
class Node:
    def __init__(self, init_data):
        self.data = init_data
        self.next = None
```

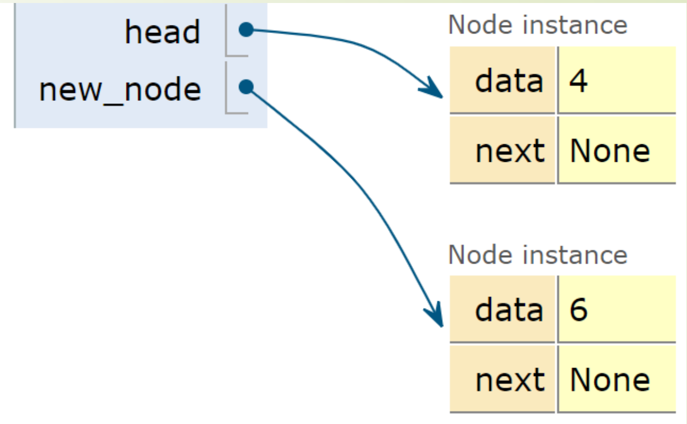
Constructing Linked Lists

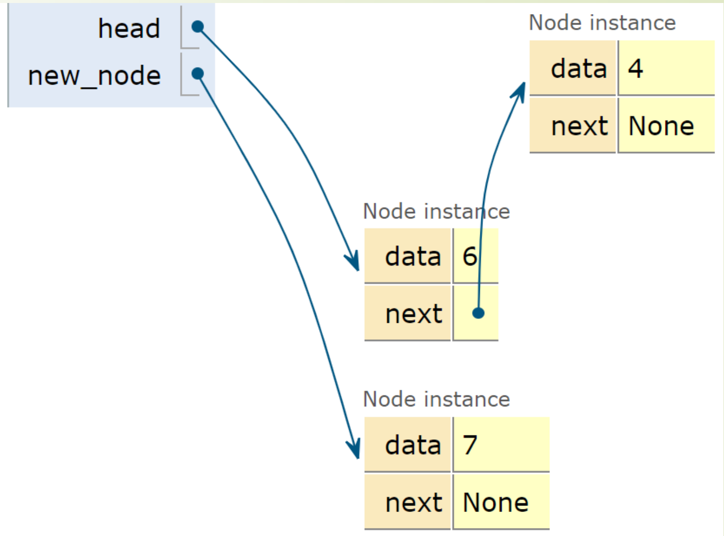
```
class Node:
    def __init__(self, init_data):
        self.data = init_data
        self.next = None

head = Node(4)
new_node = Node(6)
# to form desired sequence one of new_node and init_node needs to refer to other
new_node.next = head
# let's make the variable new_node
#       available for creating further nodes

head = new_node
new_node = Node(7)
new_node.next = head
head = new_node
```





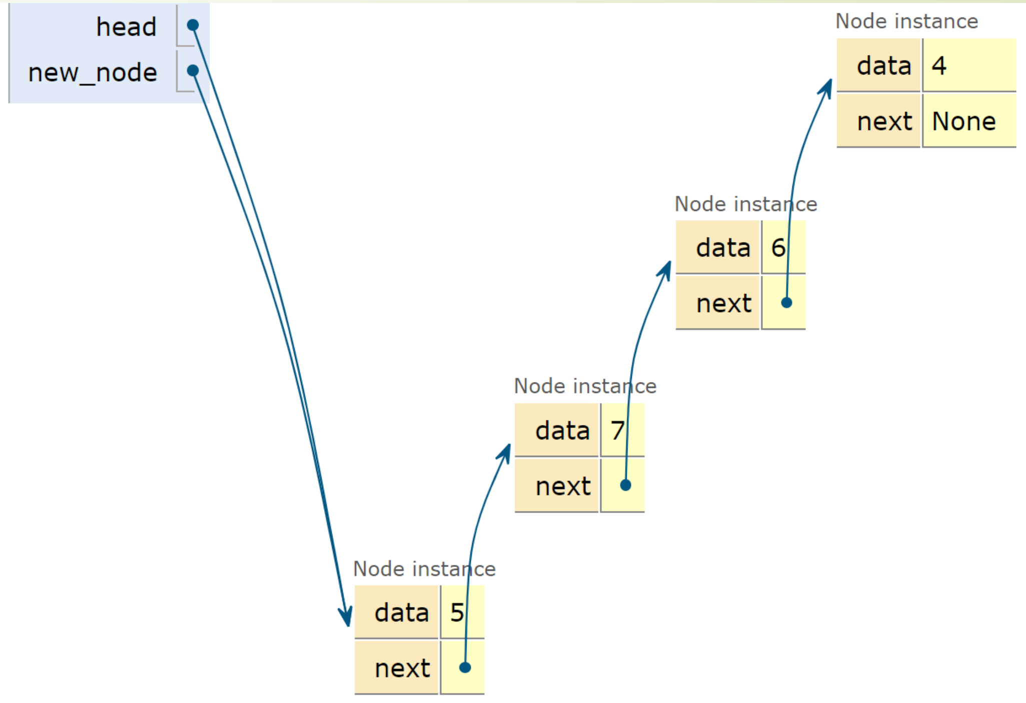






Accessing Elements in Linked Lists

- Given a sequence of items, how many operations will it take to get the n-th item?




```
…
# get element at position 3 in linked
#    list "starting at" head
current_node = init_node
for i in range(0,3):
        current_node = current_node.next
print(current_node.data)
# See https://goo.gl/vwXSzp
```