uc3m | Universidad Carlos III de Madrid

Grado en Ciencia e Ingeniería de Datos, 2018-2019

Unit 2 - Linear Abstract Data Type

Data Structures and Algorithms (DSA)

Author: Isabel Segura-Bedmar

Linear ADT

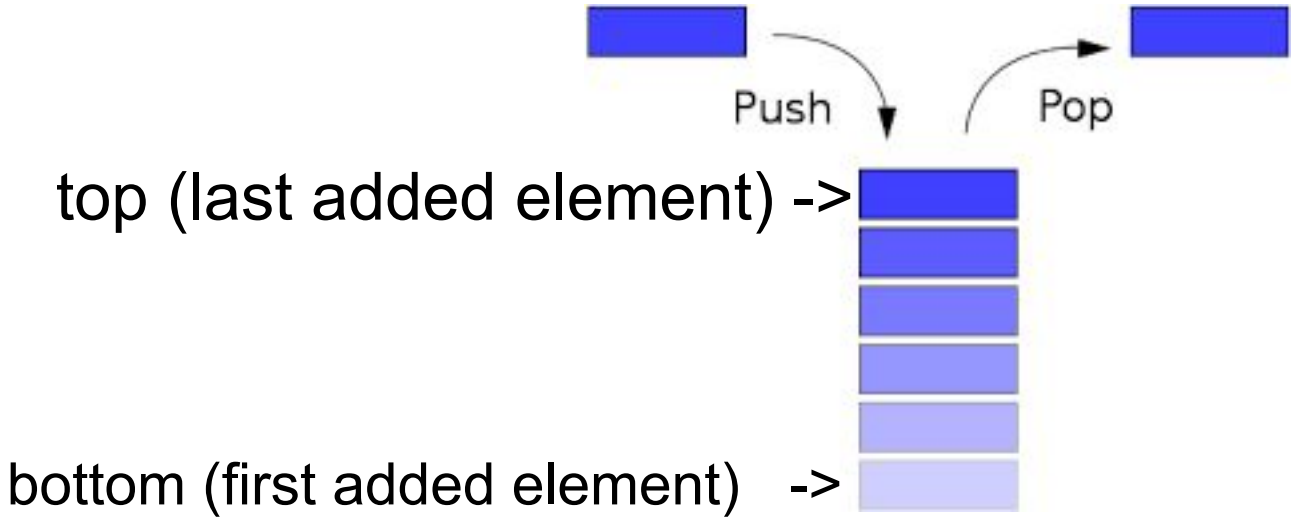
- Represents a sequence of elements of some type.
- Examples:
  - Person names (strings): Mary, John, Paul, …
  - integers: 5,6,1,-3,0,2,1,…
  - Objects (instances) of the Point class.
  - Objects (instances) of the Employee class.
- All the elements must belong to the same data type.

Index

2.1. Index of Stack ADT.

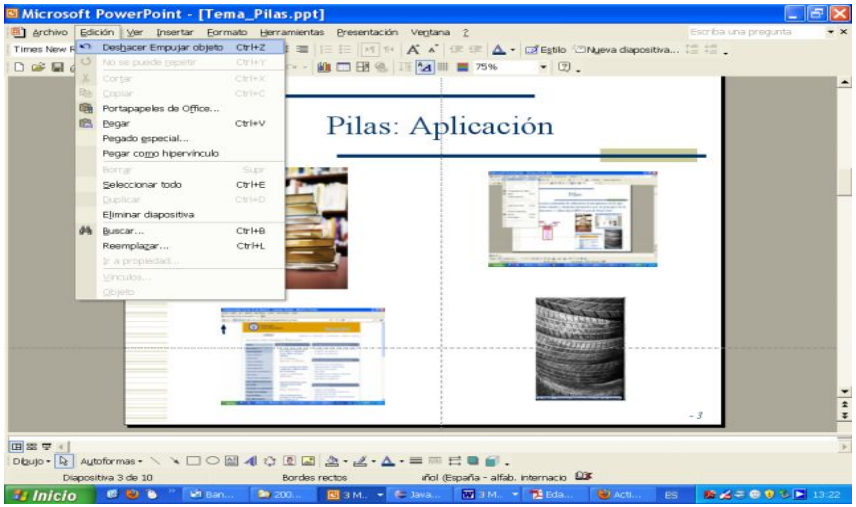
2.1.1. Stack Abstract Data Type

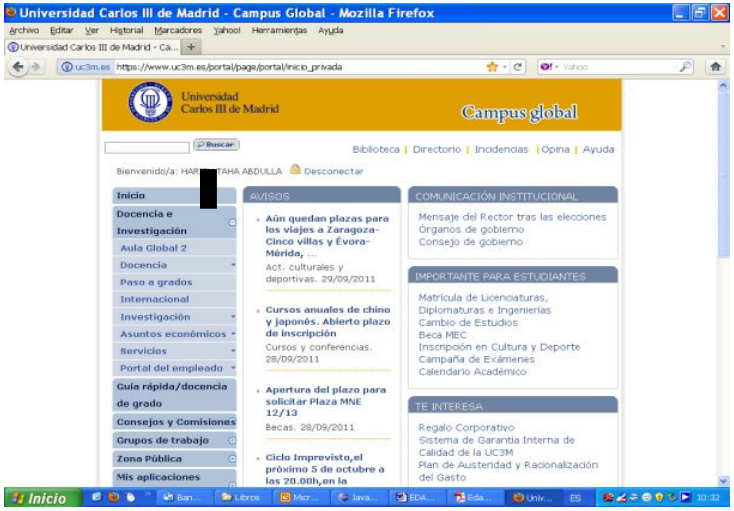
- Linear data structure based on **LIFO (last-in, first-out)** principle.
- Insertion (**push**) and removing (**pop**) operations are always performed at one single position, the **top** of the stack.



2.1.1. Stack Abstract Data Type

Undo operations





Back Navigation

2.1.1. Stack Abstract Data Type

- **top()**: returns the top element from the stack (without removing it). The stack is not modified. It throws an error if the stack is empty.
- **size**(): returns the number of elements stored on the stack.
- **isEmpty**(): returns true if the stack is empty, and false otherwise.

2.1.1. Stack Abstract Data Type

Operations	**Stack**	**Output**
s.isEmpty()	[]	True
s.push(‘W’)	[‘W’]	--
s.push(‘O’)	['W',’0’]	--
s.top()	['W',’0’]	‘O’
s.push(‘D’)	['W',’0’,’D’]	--
s.pop()	['W',’0’]	‘D’
s.size()	['W',’0’]	2
s.push(‘R’)	['W',’0’,’R’]	--
s.push(‘D)	['W',’0’,’R’,’D’]	

2.1.1. Stack Abstract Data Type. Quizzes

1. Given the following code, what is the top item on the stack?.

```
m = Stack()
m.push(5)
m.push(1)
m.pop()
m.push(3)
m.top()
```

Answers:

a) 5,

b) 1,

c) 3,

d) The stack is empty.

2.1.1. Stack Abstract Data Type. Quizzes

1. Given the following code, what is the top item on the stack?.

```
m = Stack()
m.push(5)
m.push(1)
m.pop()
m.push(3)
m.top()
```

Answers:

a) 5,

b) 1,

c) 3,

d) The stack is empty.

2.1.1. Stack Abstract Data Type. Quizzes

2. Given the following code, what is the output?.

```
m = Stack()
m.push(3)
m.push(5)
m.push(1)
while not m.isEmpty():
    print(m.pop())
    print(m.pop())
```

Answers:

a) 3,5

b) an error will occur

c) 1,5

2.1.1. Stack Abstract Data Type. Quizzes

2. Given the following code, what is the output?.

```
m = Stack()
m.push(3)
m.push(5)
m.push(1)
while not m.isEmpty():
    print(m.pop())
    print(m.pop())
```

Answers:

a) 3,5

b) an error will occur

c) 1,5

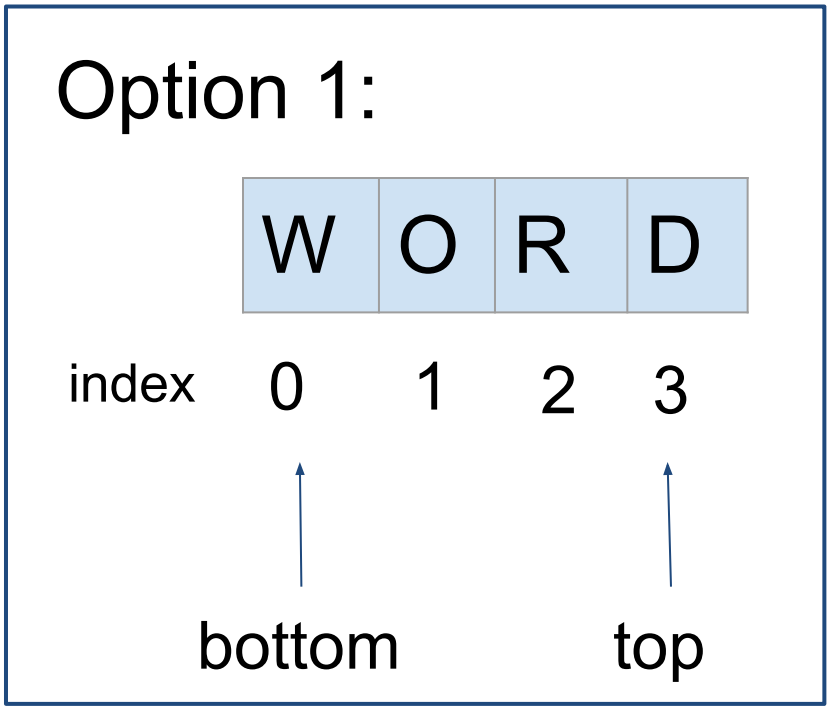
2.1. Index of Stack ADT.

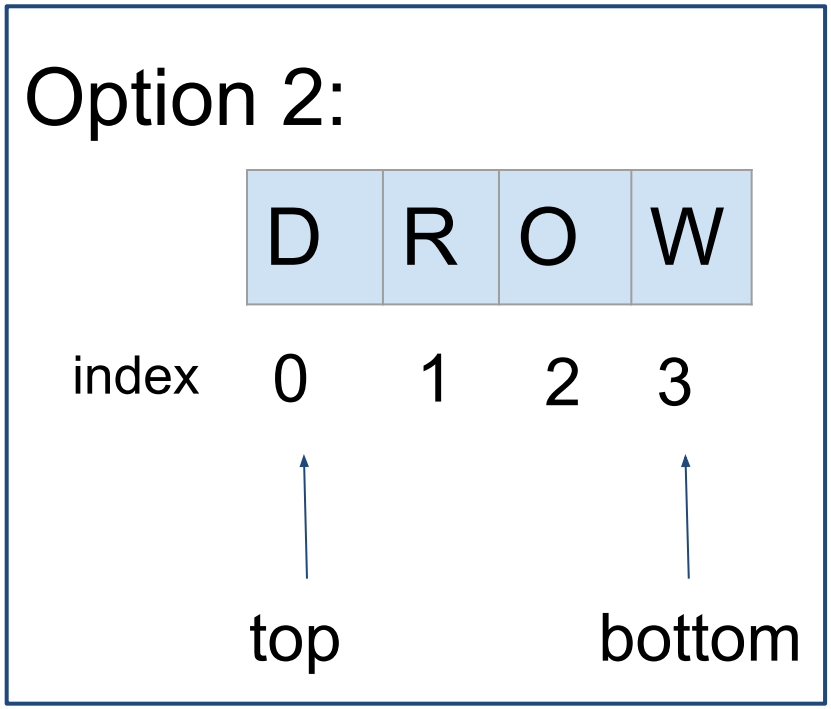
2.1.2. Implementation a stack using a Python list

- Python list already contains methods (append, pop, len, etc) to manipulate the data structure.
- Array-based implementation.

2.1.2. Implementing a stack using a Python list

- **Which end** of the list will be considered the **top** of the stack and which will be the **bottom**? Two options:





2.1.2. Implementing a stack using a Python list

- Option 1: **Top** is stored at the end of the list

```
class Stack1:
  """LIFO Stack implementation using a Python list as storage.
  The top of the stack stored at the end of the list."""

  def __init__(self):
    """Create an empty stack"""
    self.items=[]

  def push(self,e):
    """Add the element e to the top of the stack"""
    self.items.append(e)

  def pop(self):
    """Remove and return the element from the top of the stack"""
    if self.isEmpty():
      print('Error: Stack is empty')
      return None

    return self.items.pop() #remove last item from the list

  def top(self):
    """Return the element from the top of the stack"""
    if self.isEmpty():
      print('Error: Stack is empty')
      return None

    #returns last element in the list
    return self.items[-1]
```

2.1.2. Implementing a stack using a Python list

- Option 1: **Top** is stored at the end of the list

```
def len(self):
  """Return the number of elements in the stack"""
  return len(self.items)

def isEmpty(self):
  """Return True if the stack is empty"""
  return len(self.items)==0

def toString(self):
  #print the elements of the list
  return self.items
```

2.1.2. Implementing a stack using a Python list

Option 2: **Top** is stored at the beginning of the list

```
class ArrayStack:
  """LIFO Stack implementation using a Python list as storage.
  The top of the stack is stored at the beginning of the list."""
  def __init__(self):
        self.items = []

  #tests if the stack is empty
  def isEmpty(self):
    return self.items == []

  #adds at the beginning of the list
  def push(self, item):
    self.items.insert(0,item)

  #removes and returns the top element
  def pop(self):
    if self.isEmpty():
      print('Stack is empty')
      return None

    #return the first element
    return self.items.pop(0)

  #returns the top element
  def top(self):
    if self.isEmpty():
      print('Stack is empty')
      return None

    return self.items[0]
```

2.1.2. Implementing a stack using a Python list

- What implementation is better?
  - Option 1 (top at the end of the list)
  - Option 2 (top at the beginning of the list)

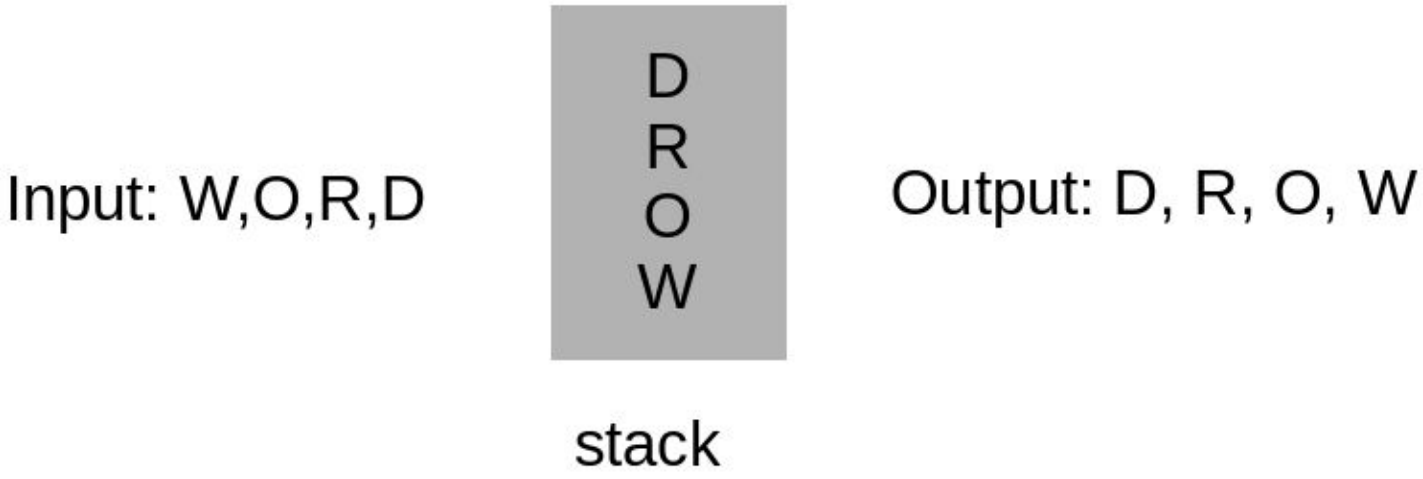
	push	pop
Option 1	O(1)	O(1)
Option 2	O(n)	O(n)

The first implementation requires less time complexity!!!

2.1. Index of Stack ADT.

2.1.3. Using stacks to solve problems

- Stacks are very useful for reversing data



2.1.3. Using stacks to solve problems

```
def reverse(word):
  """Returns the reverse word of the input word.
  It uses a stack"""
  s=ArrayStack()
  #push each character onto the stack
  for c in word:
    s.push(c)

  #variable for reverse word
  reWord=''
  while not s.isEmpty():
    #pop from the stack
    c=s.pop()
    #append at the end of the reverse word
    reWord=reWord+c

  return reWord

w=reverse('horse')
print(w)
```

2.1.3. Using stacks to solve problems

Evaluation of arithmetic and logical expressions requires checking if their parentheses are balanced:

Expression	**Balanced parenthesis?**
x= (((y+2)*5)/2 - 5) * 10	✓
(()())	✓
((()()))	✗

2.1.3. Using stacks to solve problems

- Parentheses must appear in a balanced way:
  - each opening symbol has a corresponding closing symbol
  - the pairs of parentheses are properly nested.
- A stack is a good data structure to solve this problem because closing symbols match opening symbols in the reverse order of their appearance.

2.1.3. Using stacks to solve problems

Steps to solve the problem:

1. **Create an empty stack** to store the opening symbols.

2.1.3. Using stacks to solve problems

Steps to solve the problem:

2. Read the expression from left to right. For each symbol:

i) If the symbol is an opening symbol, push it on the stack.

2.1.3. Using stacks to solve problems

Steps to solve the problem:

2. Read the expression from left to right. For each symbol:

   i) If the symbol is an opening symbol, push it on the stack.

   ii) If the symbol is a closing symbol:

   1. If the stack is empty, there is no opening symbol for it!!!. Return false.
   2. Otherwise, remove the top of the stack (with pop) and continue.

2.1.3. Using stacks to solve problems

Steps to solve the problem:

3. When all characters have been readed, there is two options:
    a. If the stack is empty, this means that the expression is balanced.
    b. Otherwise, the expression is not balanced (there are still opening symbols in the stack)

2.1.3. Using stacks to solve problems

```
def balanced(exp):
    """Checks if the parenthesis in exp are balanced"""

    s=ArrayStack()
    for c in exp:
      if c=='(':
        s.push(c)
      elif c==')':
        if s.isEmpty():
          return False
        else:
          s.pop()
      else:
        #ignore any other character
        pass

    return s.isEmpty()
```

2.1.3. Using stacks to solve problems

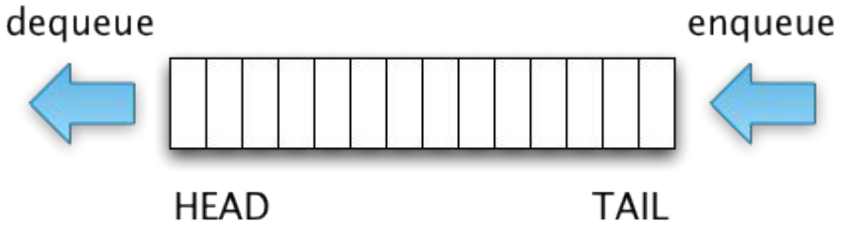
- The previous function only works for parenthesis.
- In the next lab class, we will extend it in order to deal also with:
  - Brace: ‘{‘ and ‘}
  - Brackets: ‘[‘ and ‘]’

Index

2.2. Index of Queue ADT.

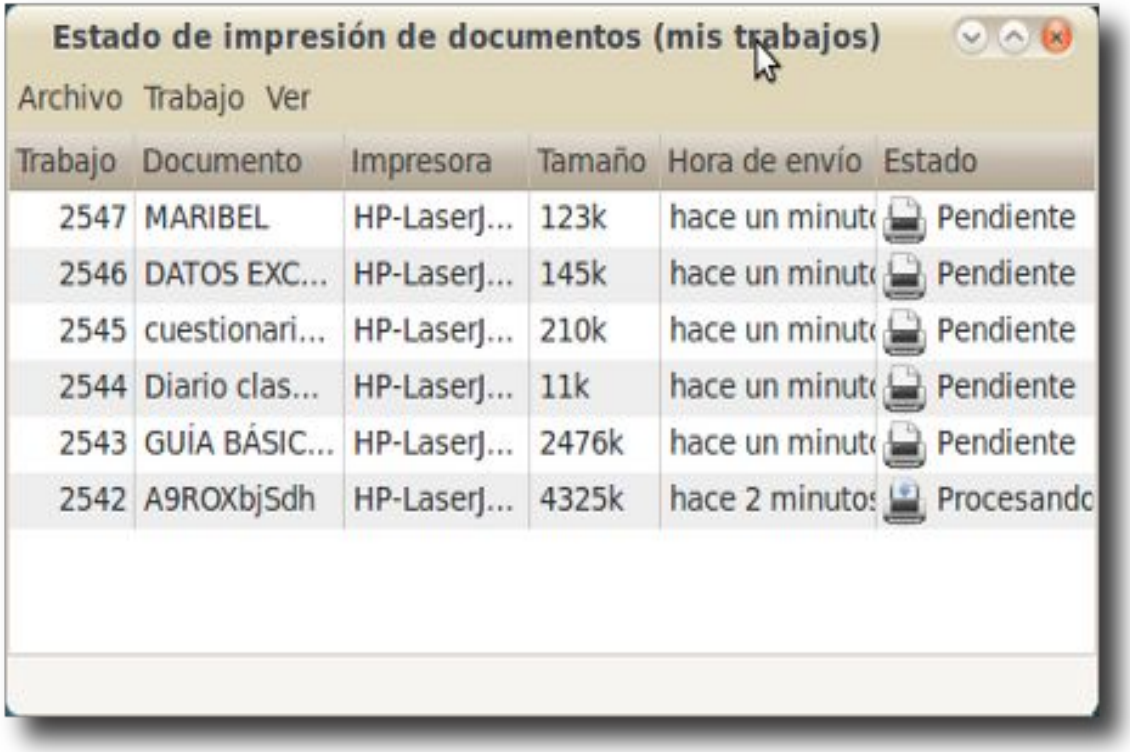
2.2.1. Queue Abstract Data Type

- Linear data structure based on **FIFO (first-in, first-out)** principle.
  - We insert (**enqueue**) a new element at the end (**tail**) of the queue.
  - We remove (**dequeue**) the first element (**head**) of the queue.



2.2.1. Queue Abstract Data Type

Applications of queues



2.2.1. Queue Abstract Data Type

- Collection of items which are added to the tail of the queue and removed from the beginning (front) of the queue.
- The queue operations are:
  - **Queue():** creates an empty queue.
  - **enqueue** (Object e): adds the element e at the tail of the queue.
  - **dequeue**(): removes and returns the first element of the queue. The queue is modified. If the queue is empty, an error is thrown.

2.2.1. Queue Abstract Data Type

- **front**(): returns the first element of the queue. The queue is not modified. If the queue is empty, it throws an error.
- **isEmpty**(): returns true if the queue is empty; otherwise false.
- **size**(): returns the numbers of elements in the queue.

2.2.1. Queue Abstract Data Type

Operations	Queue	Output
q.isEmpty()	[]	True
q.enqueue(1)	[1]	--
q.enqueue(2)	[1,2]	--
q.enqueue(3)	[1,2,3]	--
q.front()	[1,2,3]	1
q.dequeue()	[2,3]	2
d.size()	[2,3]	2

2.2.1. Queue Abstract Data Type - Quizz

Given the following code, what items are left in the queue?

```
m = Queue()
m.enqueue(3)
m.enqueue(5)
m.enqueue(1)
m.dequeue()
```

Answers:

a) 3,5,1

b) 5,1

c) 3,5

d) 3,1

2.2.1. Queue Abstract Data Type - Quizz

Given the following code, what items are left in the queue?

```
m = Queue()
m.enqueue(3)
m.enqueue(5)
m.enqueue(1)
m.dequeue()
```

Answers:

a) 3,5,1

b) 5,1

c) 3,5

d) 3,1

2.2. Index of Queue ADT.

2.2.2. Implementing a queue using a Python List

```
class ArrayQueue:
  """FIFO Queue implementation using a Python list as storage.
  We add new elements at the tail of the list (enqueue)
  and remove elements from the head of the list (dequeue)."""

  def __init__(self):
    """Create an empty queue"""
    self.items=[]

  def enqueue(self,e):
    """Add the element e to the tail of the queue"""
    self.items.append(e)

  def dequeue(self):
    """Remove and return the first element in the queue"""
    if self.isEmpty():
      print('Error: Queue is empty')
      return None
    #remove first item from the list
    return self.items.pop(0)

  def front(self):
    """Return the first element in the queue"""
    if self.isEmpty():
      print('Error: Queue is empty')
      return None
```

2.2.2. Implementing a queue using a Python List

```
def size(self):
  """Return the number of elements in the queue"""
  return len(self.items)

def isEmpty(self):
  """Return True if the queue is empty"""
  return len(self.items)==0

def toString(self):
  strQ=''
  for x in self.items:
    strQ=strQ+','+str(x)
  #print the elements of the list
  return strQ[1:]
```

2.2.2. Implementing a queue using a Python List

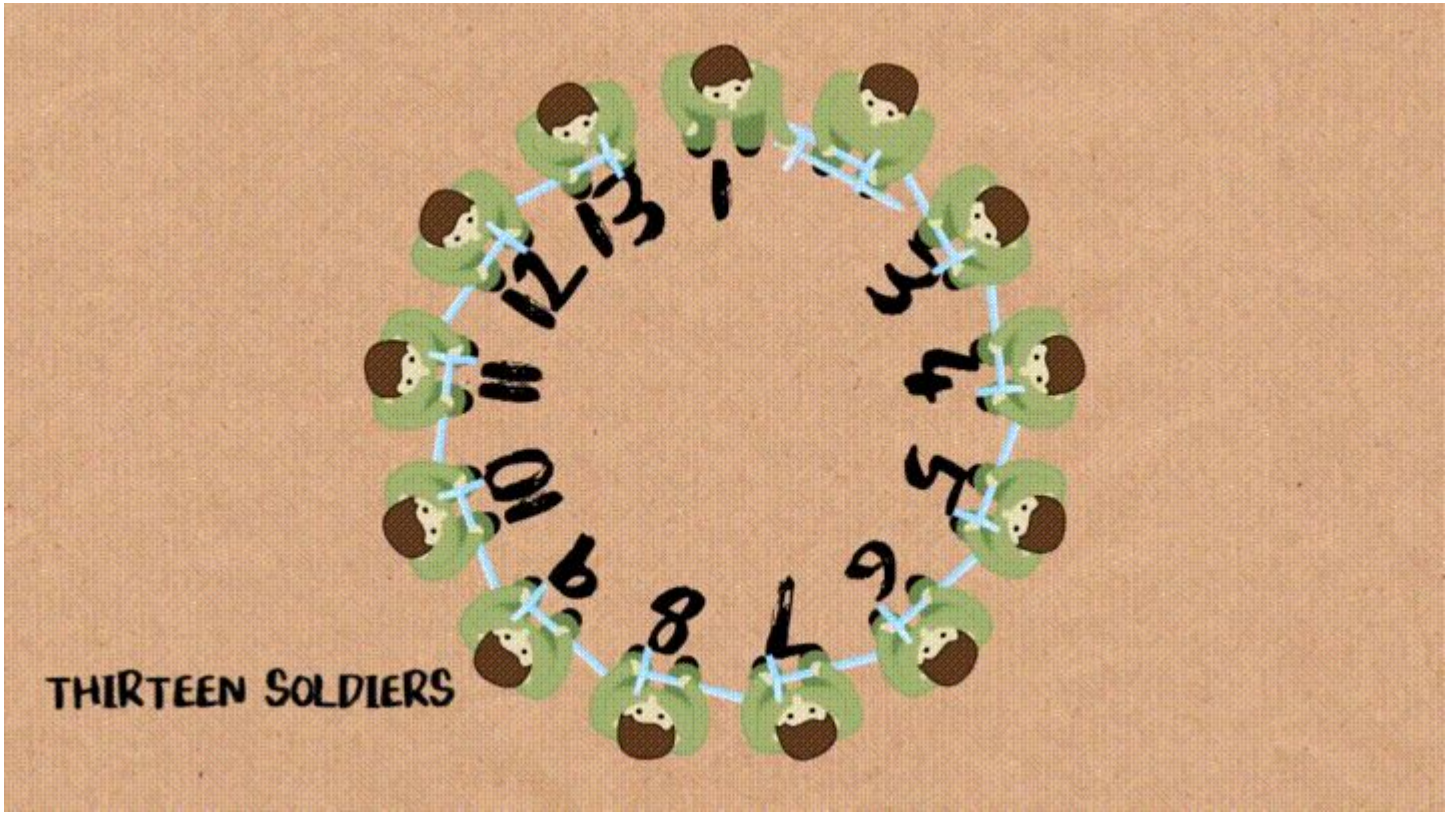
	Stack	Queue
push/enqueue	O(1)	O(1)
pop / dequeue	O(1)	O(n)

The Queue array-based implementation is worse than the array-based implementation for stacks!!!

2.2. Index of Queue ADT.

2.2.3. Using queues to solve problemas

Josephus Problem



2.2.3. Using queues to solve problemas

Josephus Problem

```
def josephus(num, k):
  q=ArrayQueue()
  #saved soldiers into the queue.
  for i in range(1,num+1):
    q.enqueue(i)

  while q.size()>1:
    count=1
    #k-1 dequeue/enqueue operations
    while count<k:
      q.enqueue(q.dequeue())
      count=count+1
    #kill the kth soldier
    print(str(q.dequeue()) + ' was killed')


  print('Surviving position: ' + str(q.front()))


josephus(40,5)
```

Index

2.3. Index of List ADT.

- **Definition of List ADT**
- How Python lists are implemented internally

2.3.1. Implementing a list ADT using a singly linked list

2.3.2. Implementing a list ADT using a doubly linked list

2.3. List Abstract Data Type

- Informal Specification:
  - Sequence of elements $\{a_1,a_2,\ldots,a_n\}$ where each element has one only predecessor (except the first one that does not have predecessor) and one only successor (except the last one that does not have successor).
  - The basic operations of a list ADT are:
    - **insert** an element to the list.
    - **remove** an element from the list
    - **read** an element in the list

2.3. List Abstract Data Type

- A collection of items where each item holds a relative position with respect to the others. Some possible operation are:
  - List() creates a new list.
  - **addFirst**(L,e) add the element e at the beginning of the list L.
  - **addLast**(L,e) add the element e at the tail of the list L.
  - **removeFirst**(L) removes the first element of the list L. It returns the element.
  - **removeLast**(L) removes the last element of the list L. It returns the element.

2.3. List Abstract Data Type

- More operations:
  - **isEmpty**(L): returns True if the list L is empty, False otherwise.
  - **size**(L): returns the number of items of the list.
  - **contains**(L,e): returns the first position of the element e in the list. If the element doesn't exist return -1.
  - **insertAt**(L,index,e): inserts the element e at the position index of the list L.
  - **removeAt**(L,index): removes the element at the position index of the list L. It returns the element.

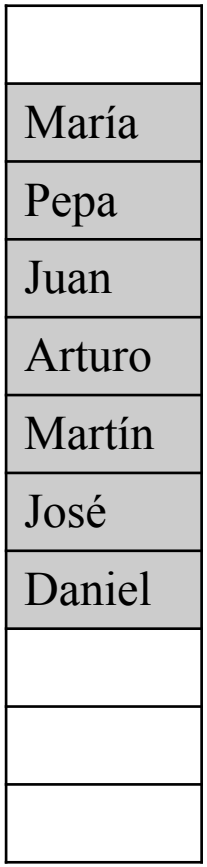
2.3. Index of List ADT.

- Definition of List ADT
- **How Python lists are implemented internally**

2.3.1. Implementing a list ADT using a singly linked list

2.3.2. Implementing a list ADT using a doubly linked list

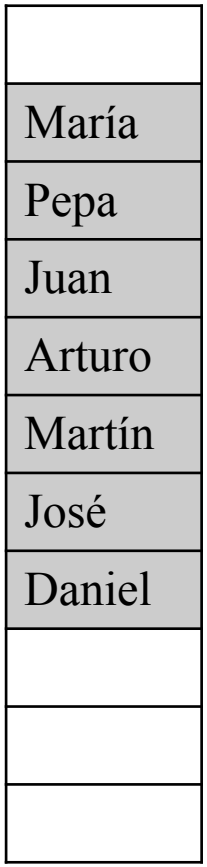
2.3. List Abstract Data Type



How are Python lists implemented internally?

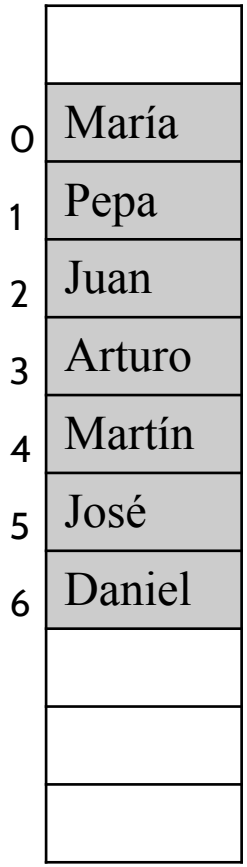
To save a list of n elements, we would need n consecutive cells in memory

2.3. List Abstract Data Type

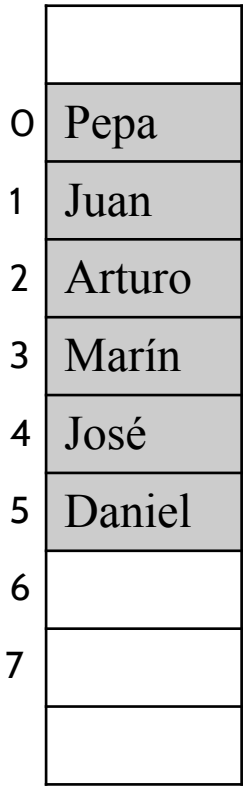


Easy and fast access to all its elements: A[i]

2.3. List Abstract Data Type



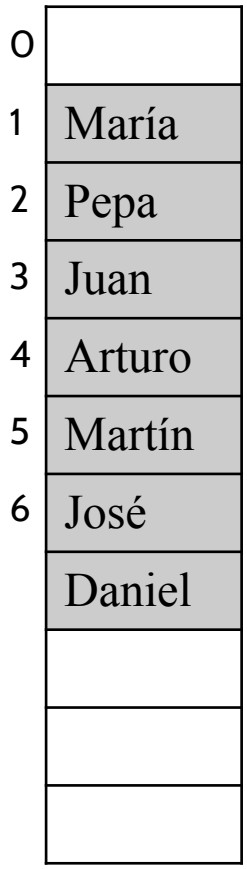
A.pop(0)



Move all the elements one spot to the left in the list

This is not efficient when your program has to perform a lot of remove operations

2.3. List Abstract Data Type

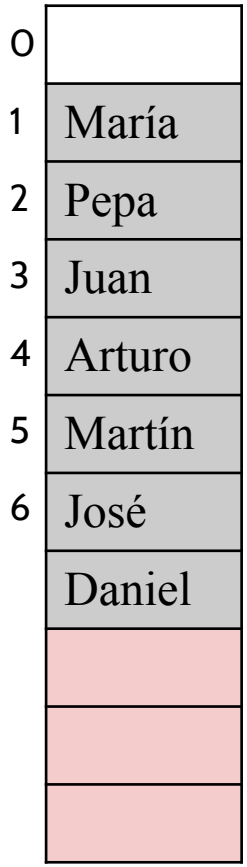


A.insert(3,"Isabel")

Move all the elements (index>=3) one position to the right

This is not efficient when your program has to perform a lot of insertion operations

2.3. List Abstract Data Type



A.append(”Isabel”)

If the next location after the last element is not free => search a new place for the whole list.

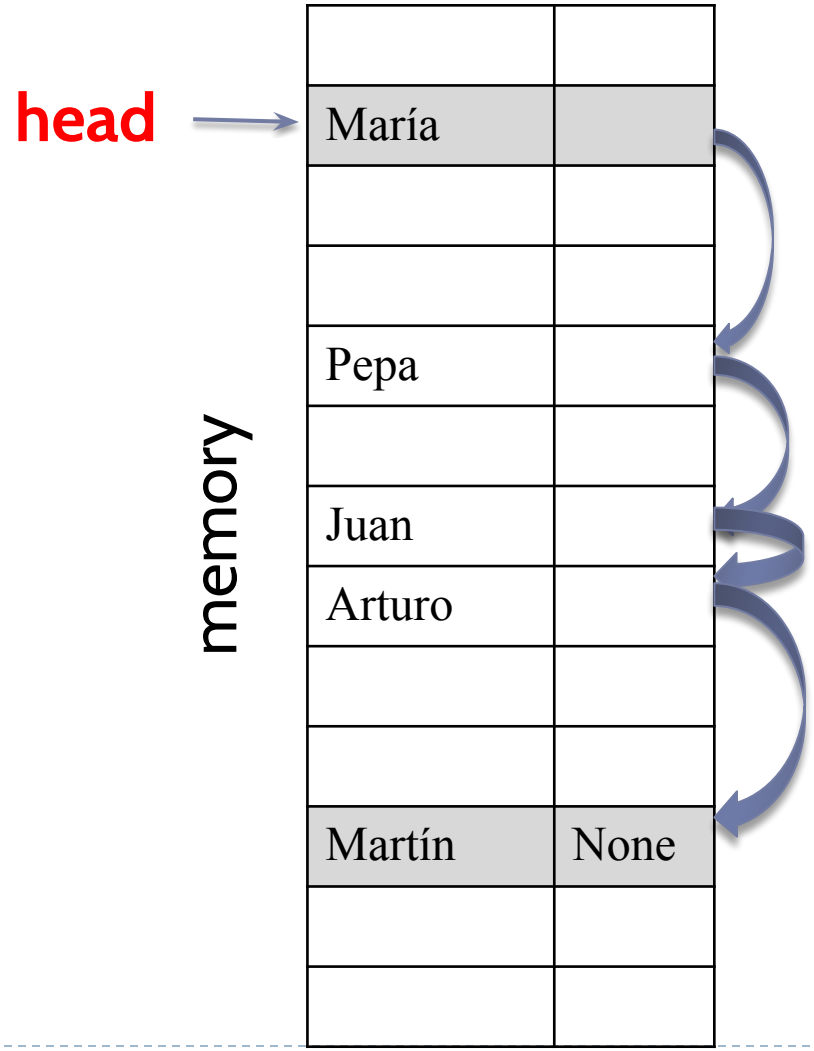
2.3. Index of List ADT.

- Definition of List ADT
- How Python lists are implemented internally

2.3.1. Implementing a list ADT using a singly linked list

2.3.2. Implementing a list ADT using a doubly linked list

2.3.1. Implementing a List using a singly linked list

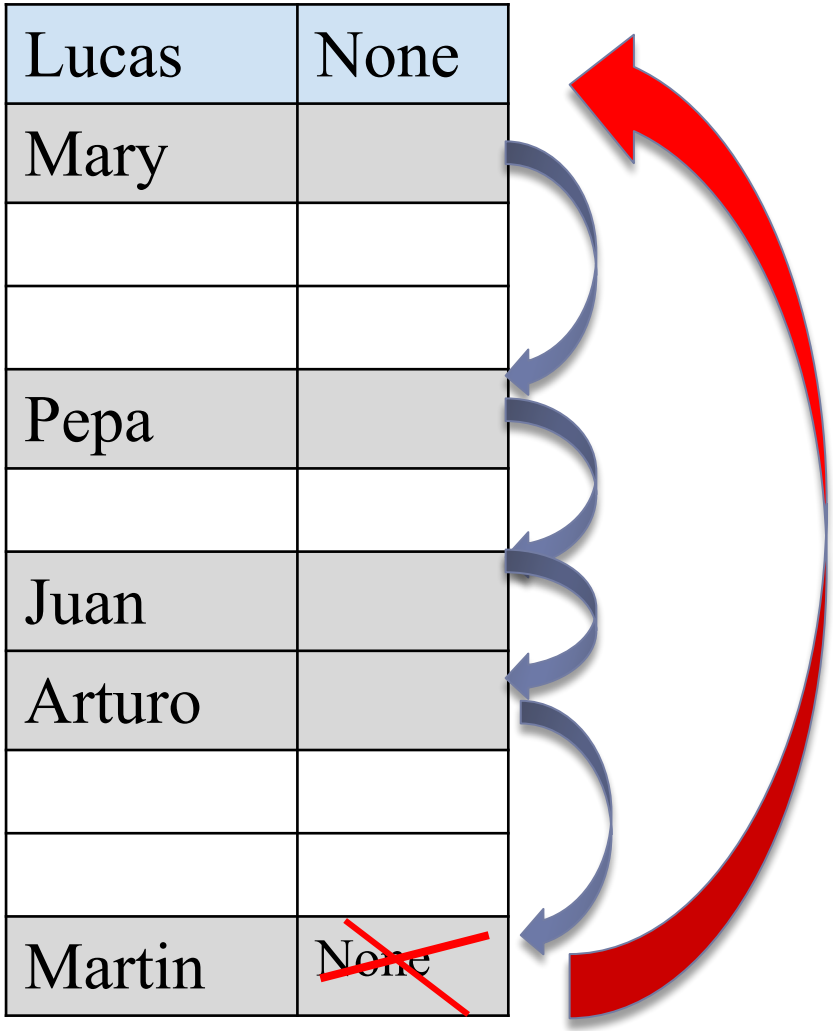


2.3.1. Implementing a List using a singly linked list

head →

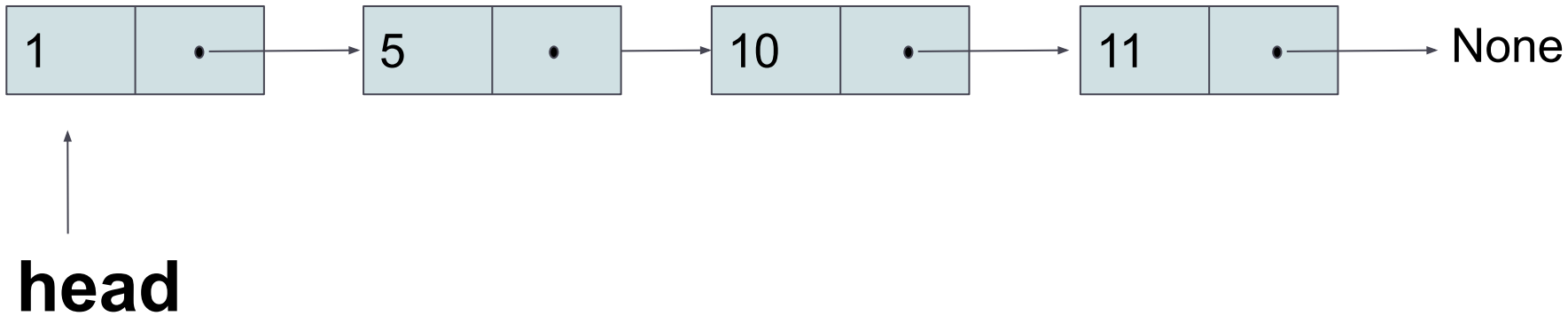
A.addLast("Lucas")

The gaps in memory allow that the physical and logical orders can be different.



2.3.1. Implementing a List using a singly linked list

- Each **node** stores an element of the sequence and a reference to the next node of the list.



2.3.1. Implementing a List using a singly linked list (pseudo-code for singly node constructor)

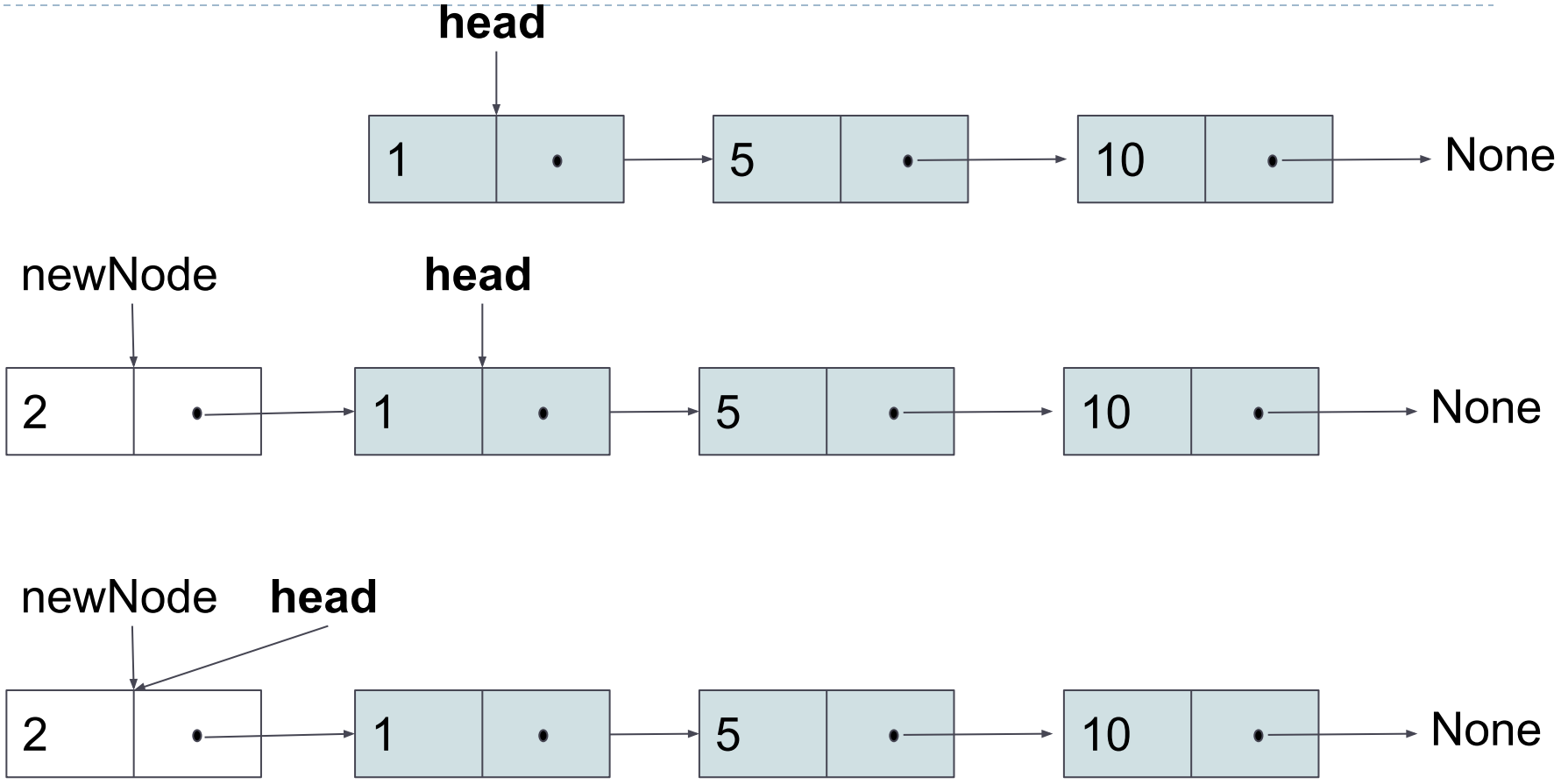
```
Algorithm Node(node,e):
        node.element=e
        node.next=None
```

2.3.1. Implementing a List using a singly linked list (pseudo-code for Singly Linked List constructor)

- The constructor creates an empty list.
- head must be None.

```
Algorithm SList(list):
      list.head=None
      list.size=0
```

2.3.1. Implementing a List using a singly linked list (addFirst method)



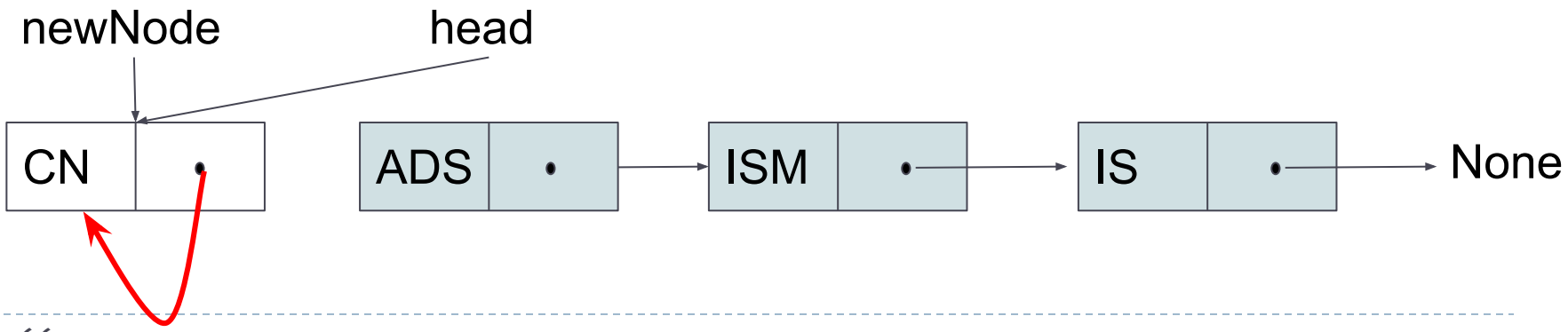
2.3.1. Implementing a List using a singly linked list (addFirst method, pseudo-code)

```
Algorithm addFirst(L,e):
    newNode=Node(e)      #1
    newNode.next=L.head  #2
    L.head=newNode       #3
    L.size=L.size+1      #4
```

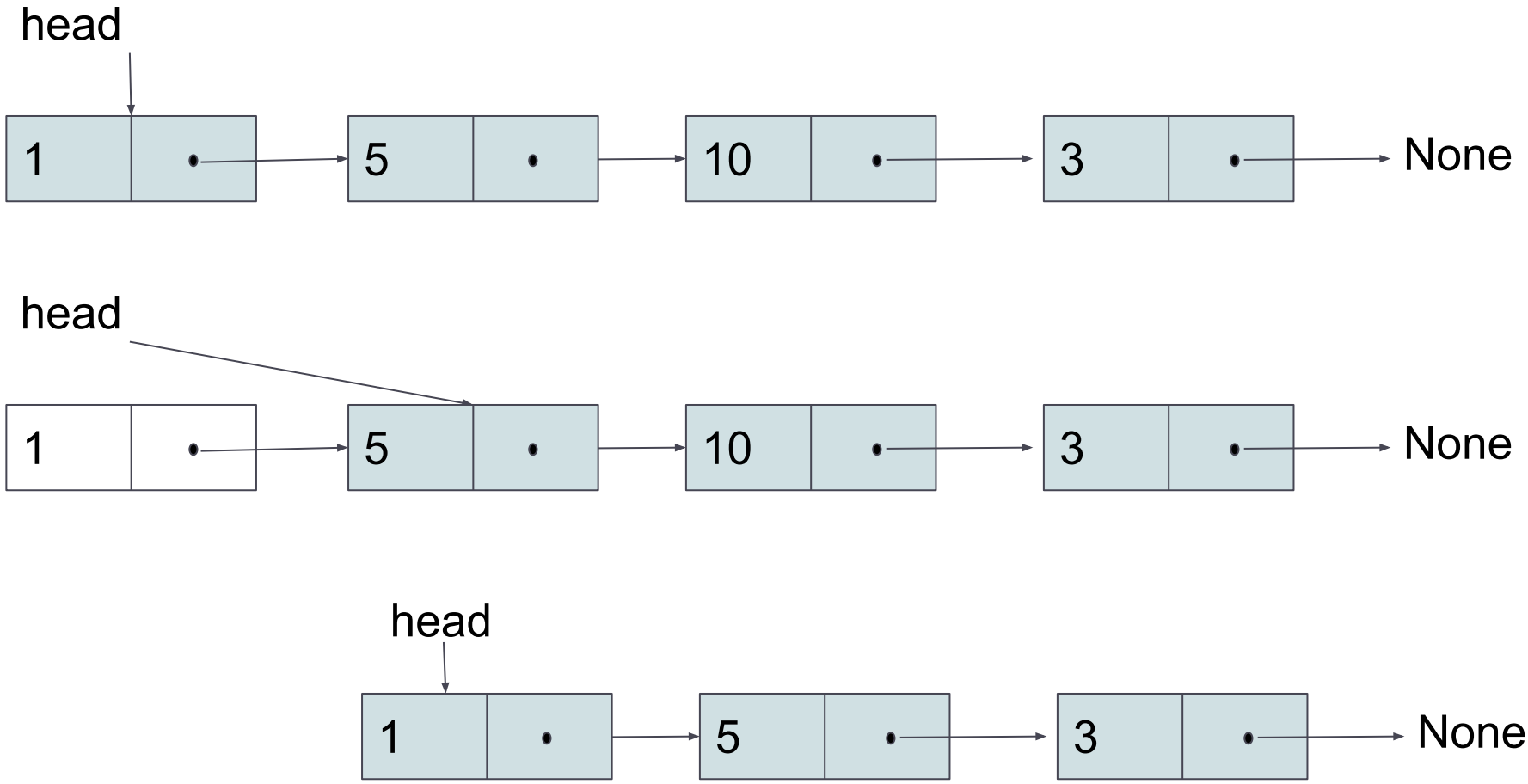
2.3.1. Implementing a List using a singly linked list (addFirst method, pseudo-code)

Is the order of instructions important?

```
Algorithm addFirst(L,e):
    newNode=Node(e)              #1
    L.head=newNode               #3
    newNode.next=L.head          #2
    L.size=L.size+1              #4
```



2.3.1. Implementing a List using a singly linked list (removeFirst method)



2.3.1. Implementing a List using a singly linked list (removeFirst method, pseudo-code)

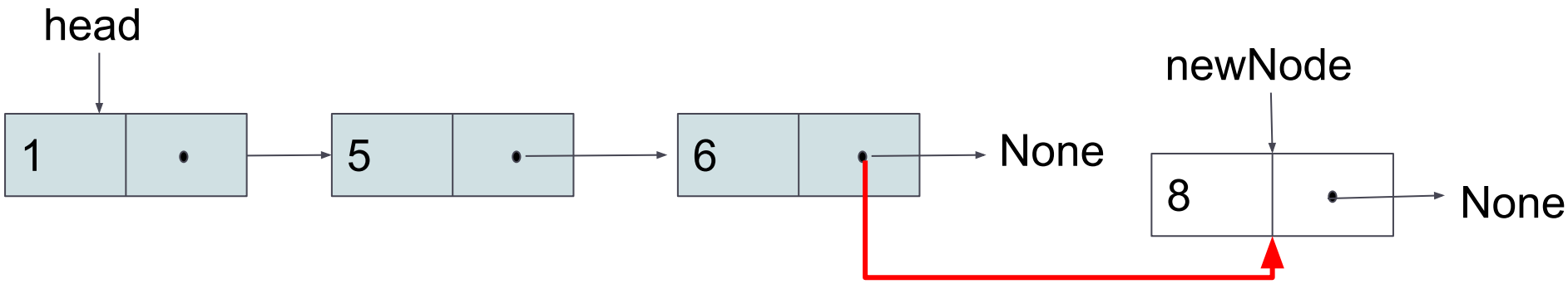
```
Algorithm removeFirst(L):
   e=L.head.element      #1
   L.head=L.head.next    #2
   L.size=L.size-1       #3
   return e              #4
```

It does not work when L is empty!!!

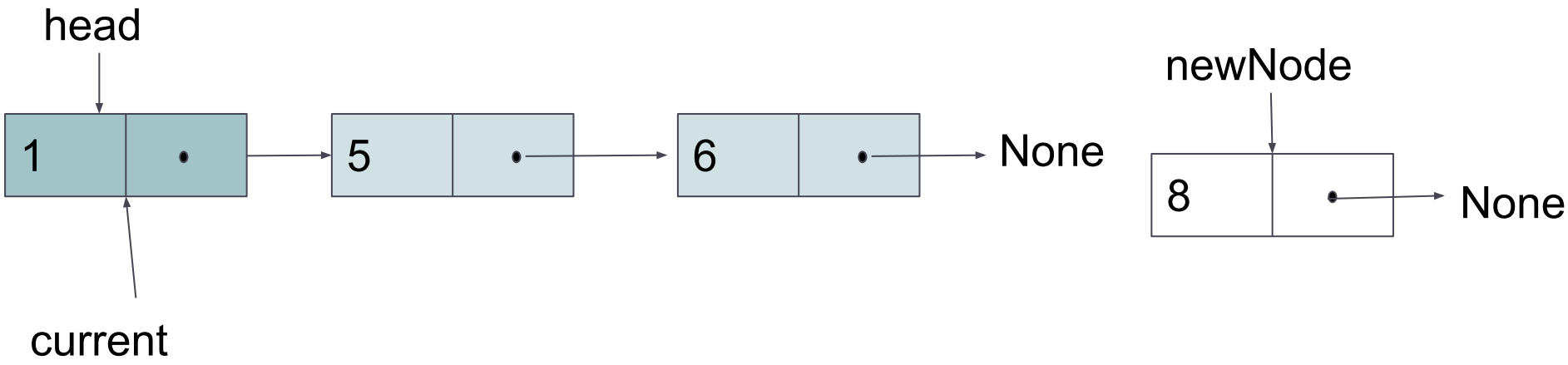
2.3.1. Implementing a List using a singly linked list (removeFirst method, pseudo-code)

```
Algorithm removeFirst(L):
If L.head is None then
    Show an error: list is empty

  e=L.head.element      #1
  L.head=L.head.next    #2
  L.size=L.size-1       #3
  return e              #4
```

2.3.1. Implementing a List using a singly linked list (addLast method)

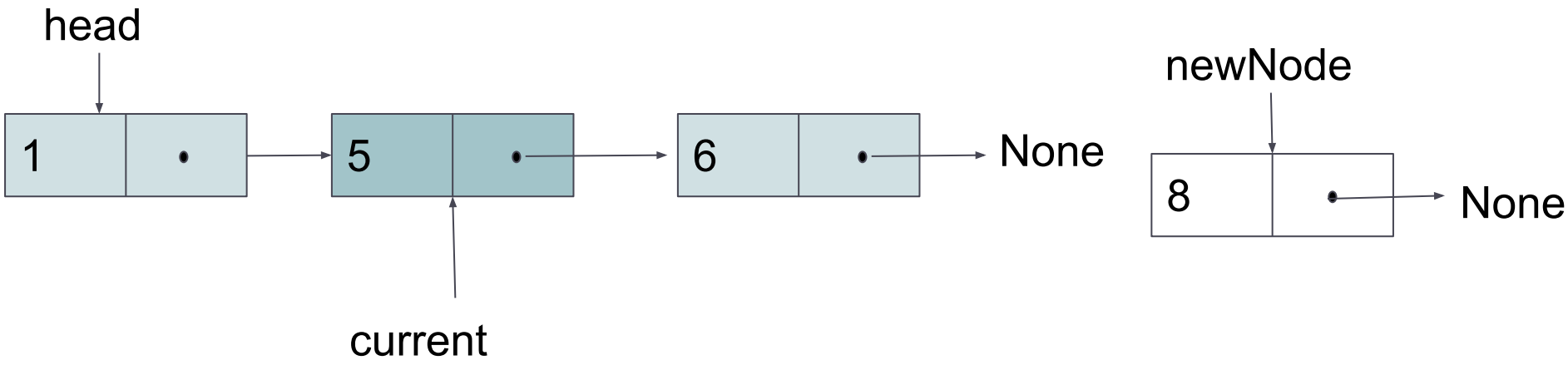


2.3.1. Implementing a List using a singly linked list (addLast method)



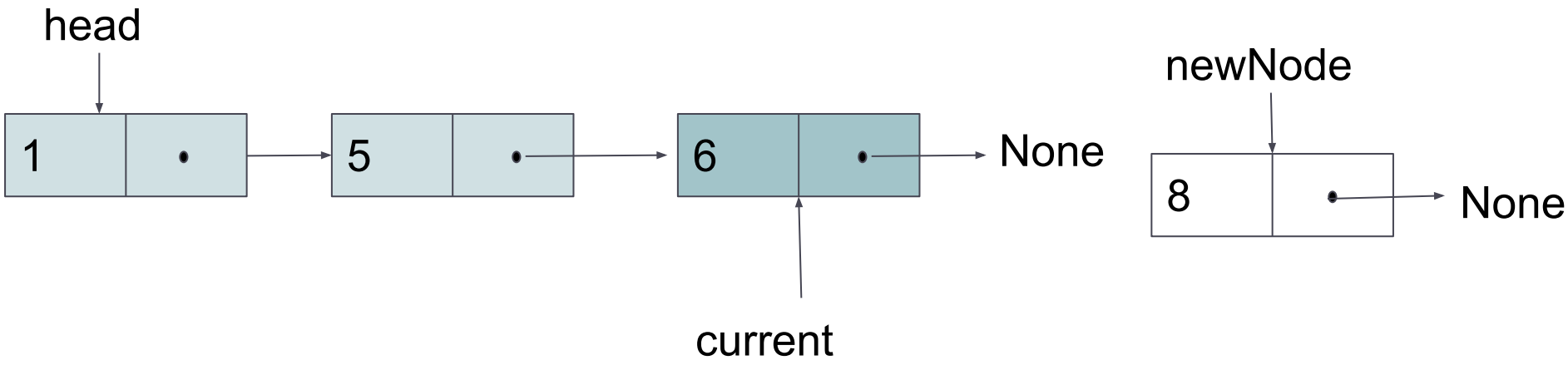
current = first

2.3.1. Implementing a List using a singly linked list (addLast method)



current = current.next

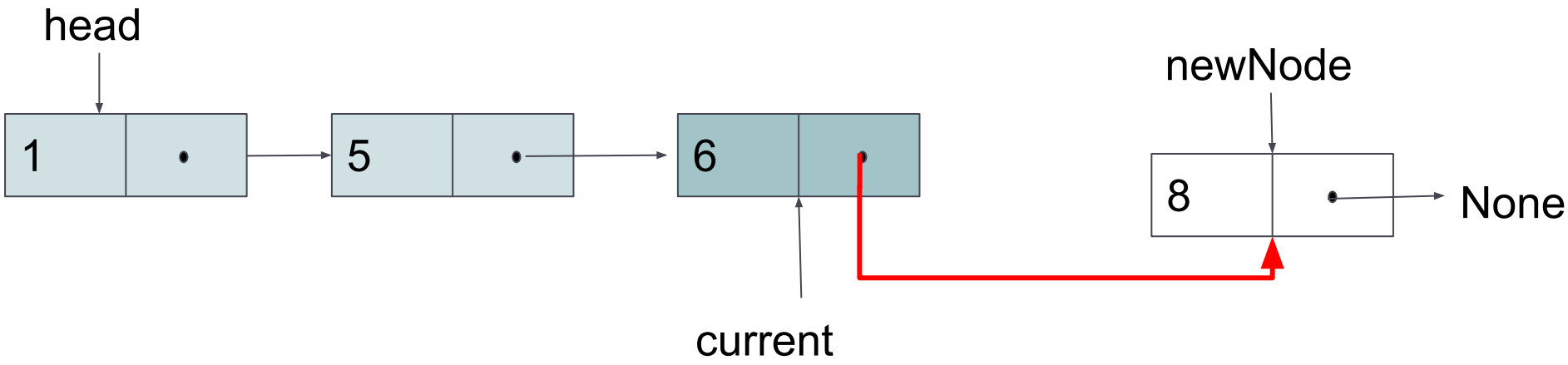
2.3.1. Implementing a List using a singly linked list (addLast method)



current = current.next

The last node is reached when current.next=None

2.3.1. Implementing a List using a singly linked list (addLast method)



current.next= newNode

The last node must point to the new node

2.3.1. Implementing a List using a singly linked list (addLast method, pseudo-code)

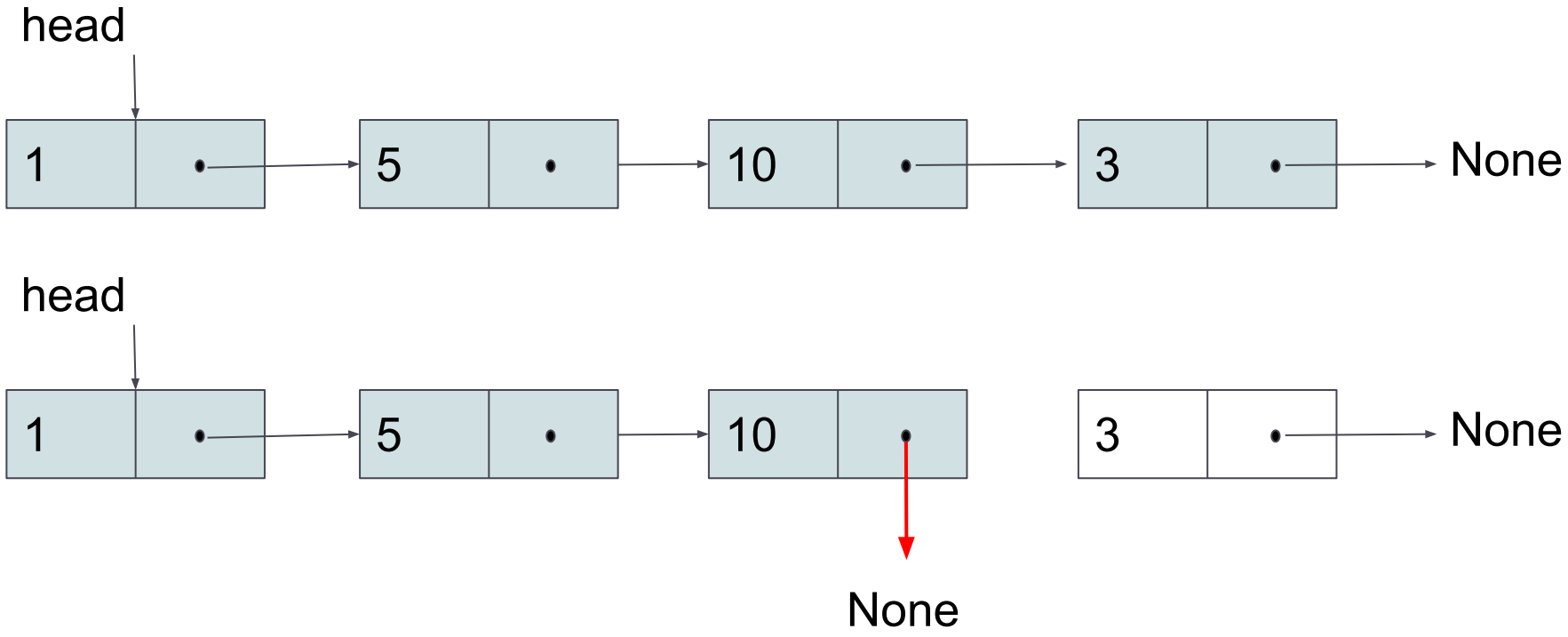
```
Algorithm addLast(L,e):
    newNode=Node(e)
    current=L.head
    while current.next is not None:
        current=current.next
    current.next=newNode
    L.size=L.size+1
```

Does this code work if L is empty?

2.3.1. Implementing a List using a singly linked list (addLast method, pseudo-code)

```
Algorithm addLast(L,e):
  if L.head==None:
    addFirst(L,e)

  newNode=Node(e)
  current=L.head
  while current.next is not None:
    current=current.next
  current.next=newNode
  L.size=L.size+1
```

2.3.1. Implementing a List using a singly linked list (removeLast method)



The penultimate node must point to None with its next attribute.

2.3.1. Implementing a List using a singly linked list (removeLast method, pseudo-code)

```
Algorithm removeLast(L):
    if L.head is None:
      show an error
      return None

    previous=None
    current=self.head
    while current.next is not None:
        previous=current
        current=current.next

    e=current.element
    previous.next=None

    self.size=self.size-1

    return e
```

2.3.1. Implementing a List using a singly linked list (getAt method, pseudo-code)

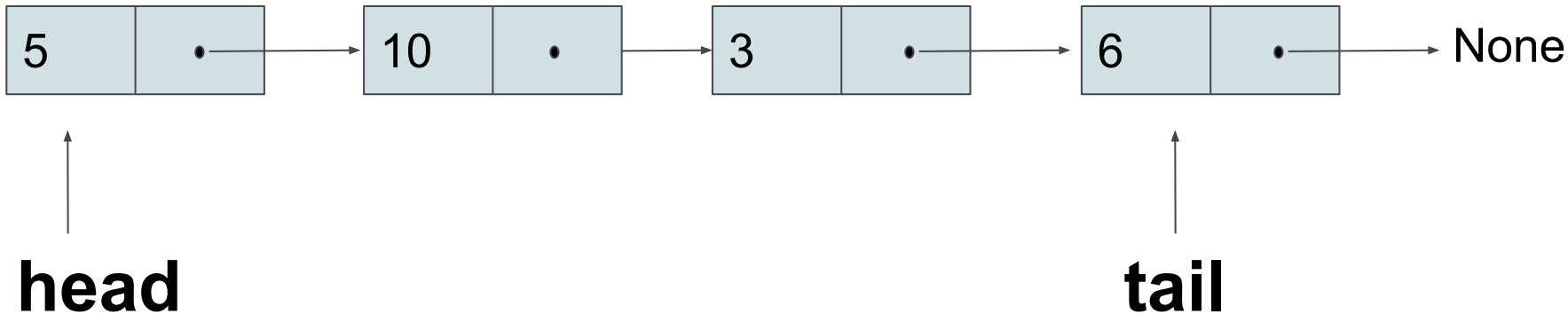
```
Algorithm getAt(L,index):
    if index<0 or index>=L.size:
      show "index out of index"
      return None

    i=0
    current=self.head
    while i<index:
        current=current.next
      i = i+1

    return current.element
```

2.3.1. Implementing a List using a singly linked list (Exercises)

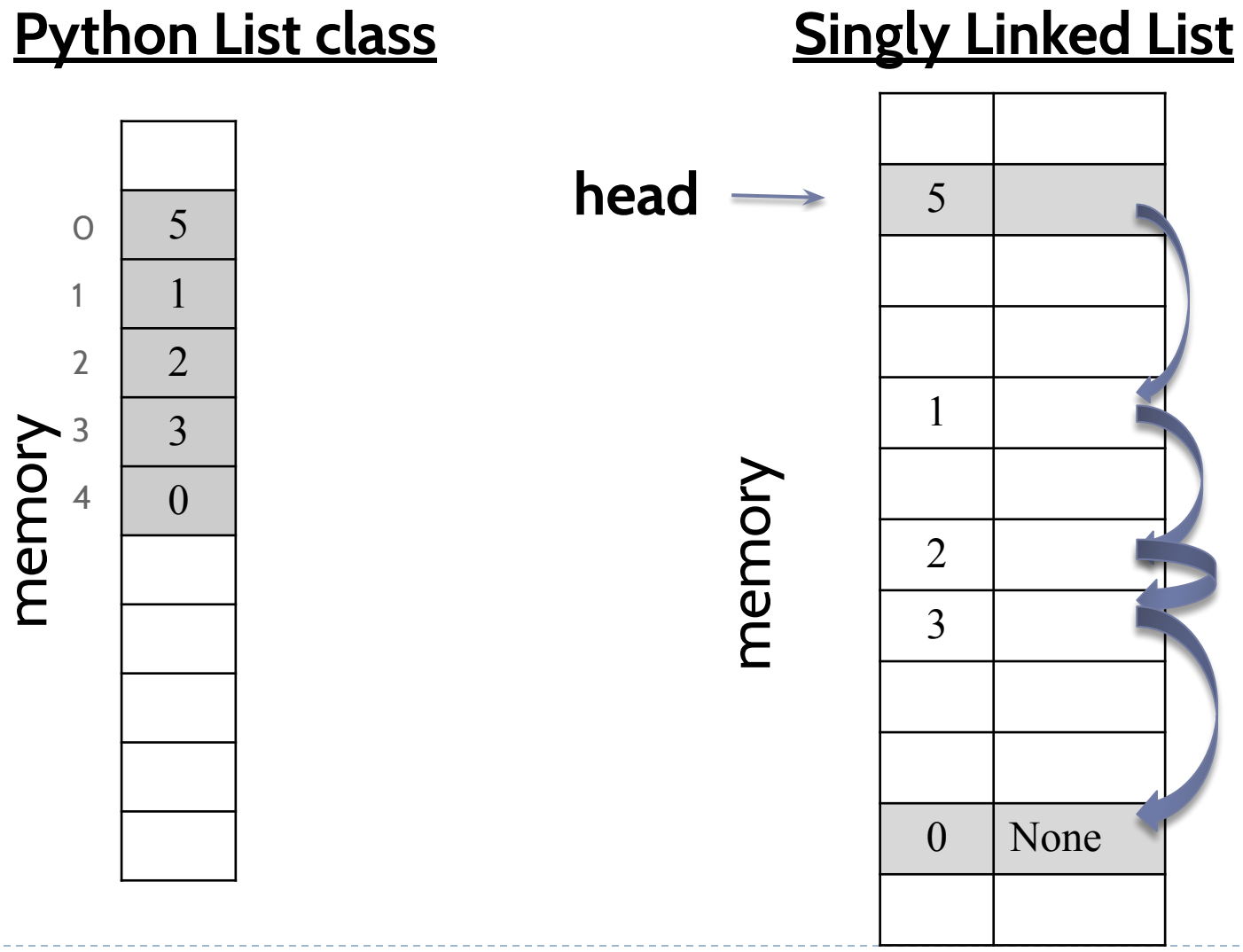
- Implement the following methods:
  - contains(L,e):returns the first position of the element e at the list. If e does not exist, then it returns -1.
  - insertAt(L,index,e): inserts the element e at the position index of the list L.
  - removeAt(L,index): removes the element at the position index from the list L.
- Implement a stack with a singly linked list.
- Implement a queue with a singly linked list.

2.3.1. Implementing a List using a singly linked list (Exercises)

- Implement a singly linked list that also includes a reference to the last node (tail).



2.3.1. Implementing a List using a singly linked list (Python List versus Singly Linked List)



2.3.1. Implementing a List using a singly linked list (Python List versus Singly Linked List)

Python List Implementation

(+) Fast and Easy access to the elements.

(-) Insertion and remove operations are slow (as shifting elements is required)

Singly Linked List

(-) Requires more space

(-) Sequential access (you must visit all the previous elements)

(+) Insertion and remove operations are more efficient.

2.3. Index of List ADT.

- Definition of List ADT
- How Python lists are implemented internally

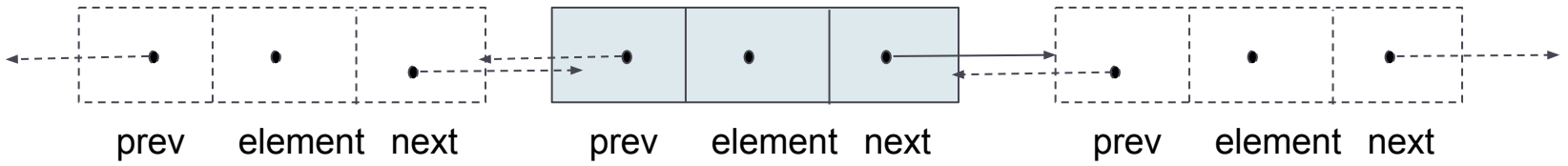
2.3.1. Implementing a list ADT using a singly linked list

2.3.2. Implementing a list ADT using a doubly linked list

2.3.2. Implementing a List using a doubly linked list

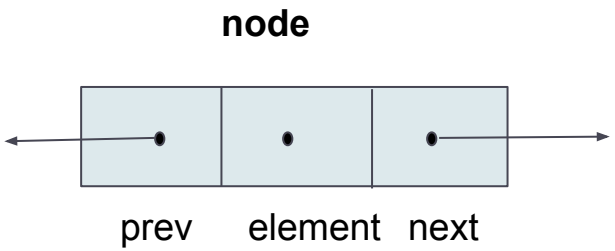
How improve the access to the nodes?

- In addition to the reference **next**, a node has an additional reference to the previous node (**prev**).
- It allows visiting the list from left to right, and also in reverse.



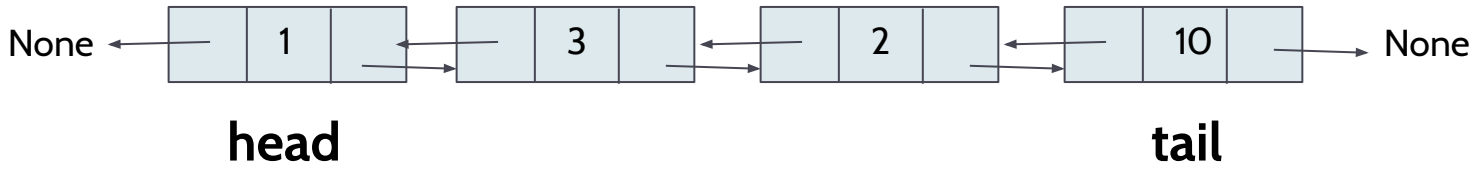
2.3.2. Implementing a List using a doubly linked list (pseudocode for doubly node constructor)

```
Algorithm Node(node,e):
    node.element=e
    node.next=None
    node.prev=None
```



2.3.2. Implementing a List using a doubly linked list

- Use two references:
  - head: points to the first node of the list.
  - tail: points to the last node of the list.



2.3.2. Implementing a List using a doubly linked list (pseudocode for Doubly Linked List constructor)

- The constructor creates an empty list.
- head and tail must be None.

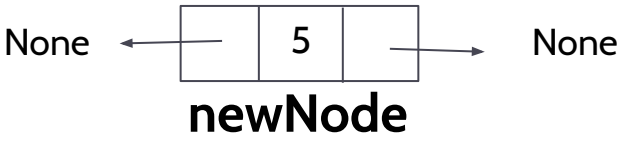
```
Algorithm DList(list):
    list.head=None
    list.tail=None
    list.size=0
```

2.3.2. Implementing a List using a doubly linked list

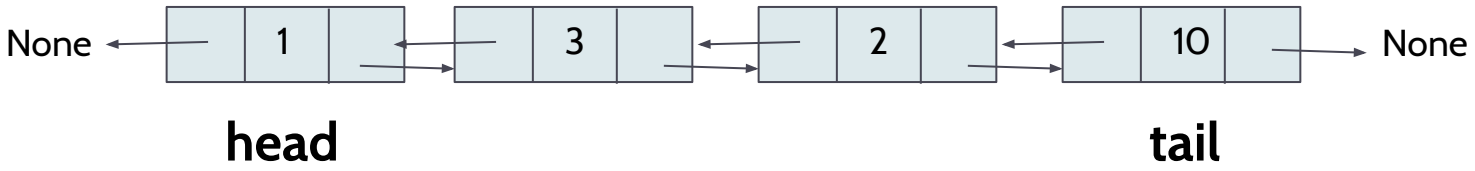
- **addFirst(L,e) adds the element e at the front of the list L.**
- addLast(L,e) adds the element e at the tail of the list L.
- removeFirst(L) removes the first element of the list L. It returns the element.
- removeLast(L) removes the last element of the list L. It returns the element.
- insertAt(L,index,e) inserts the element e at the index position of the list.

2.3.2. Implementing a List using a doubly linked list (addFirst method)

1) **Create a new node.**
2) Link the new node to the list
3) Update the head reference
4) Increase the size



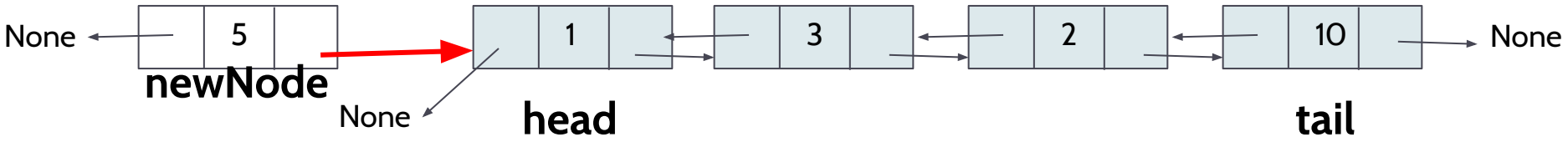
newNode=Node(e)



for example, l.addFirst(5)

2.3.2. Implementing a List using a doubly linked list (addFirst method)

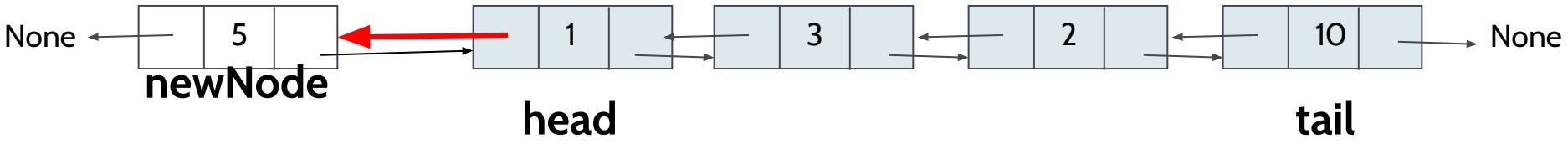
1) Create a new node.
2) **Link the new node to the list**
3) Update the head reference
4) Increase the size



newNode.next = L.head

2.3.2. Implementing a List using a doubly linked list (addFirst method)

1) Create a new node.
2) **Link the new node to the list**
3) Update the head reference
4) Increase the size

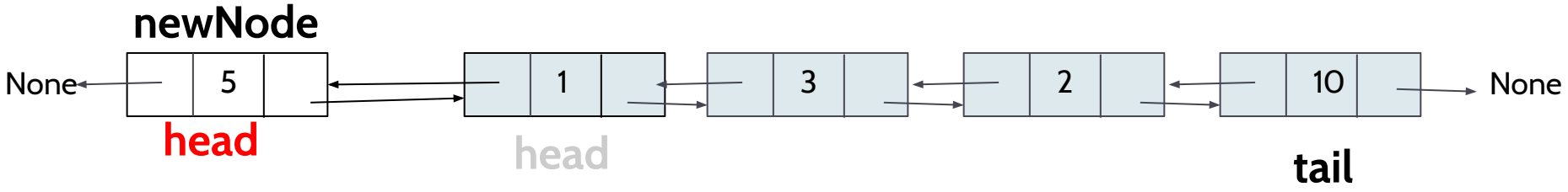


newNode.next = L.head

L.head.prev=newNode

2.3.2. Implementing a List using a doubly linked list (addFirst method)

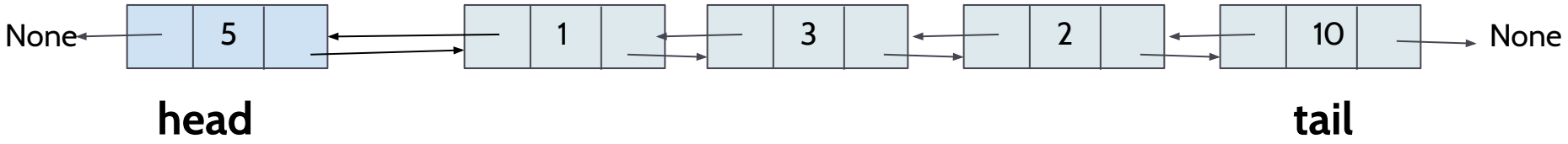
1) Create a new node.
2) Link the new node to the list
3) **Update the head reference**
4) Increase the size



L.head=newNode

2.3.2. Implementing a List using a doubly linked list (addFirst method)

1) Create a new node.
2) Link the new node to the list
3) Update the head reference
4) **Increase the size**



L.size=L.size+1

2.3.2. Implementing a List using a doubly linked list (addFirst method, pseudo-code)

```
Algorithm addFirst(L,e):
    newNode=Node(e)
    newNode.next=L.head
    L.head.prev=newNode
    L.head=newNode
    L.size=L.size+1
```

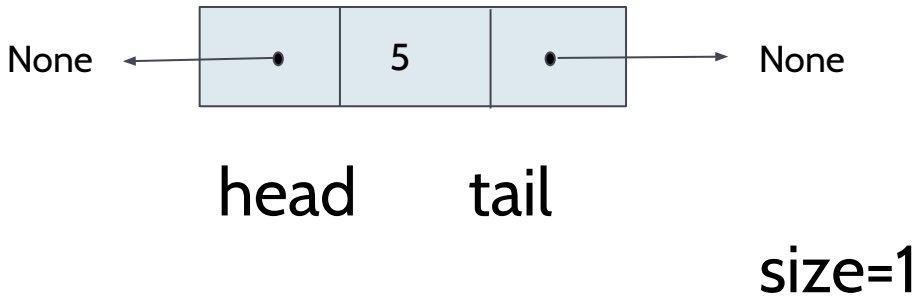
2.3.2. Implementing a List using a doubly linked list (addFirst method, pseudo-code)

```
Algorithm addFirst(L,e):
    newNode=Node(e)
    newNode.next=L.head
    L.head.prev=newNode
    L.head=newNode
    L.size=L.size+1
```

Does it work when the list is empty?

2.3.2. Implementing a List using a doubly linked list (addFirst method, pseudo-code)

- **If the list is empty:**
  1) Create a new node.
  2) Update the references head and tail for pointing to the new node.
  3) Increase the size



2.3.2. Implementing a List using a doubly linked list (addFirst method, pseudo-code)

```
Algorithm addFirst(L,e):
   if L.head=None:
      newNode=Node(e)
      L.tail=newNode
      L.head=newNode
      L.size=L.size+1
   else:
      …
```

2.3.2. Implementing a List using a doubly linked list (addFirst method, pseudo-code)

```
Algorithm addFirst(L,e):
    newNode=Node(e)

    if L.head=None:
      L.tail=newNode
    else:
      newNode.next=L.head
      L.head.prev=newNode

    L.head=newNode
    L.size=L.size+1
```

2.3.2. Implementing a List using a doubly linked list

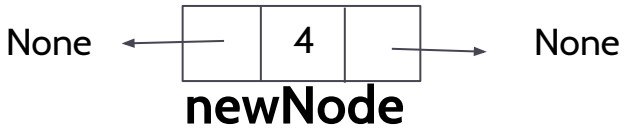
- addFirst(L,e) adds the element e at the front of the list L.
- **addLast(L,e) adds the element e at the tail of the list L.**
- removeFirst(L) removes the first element of the list L. It returns the element.
- removeLast(L) removes the last element of the list L. It returns the element.
- insertAt(L,index,e) inserts the element e at the index position of the list.

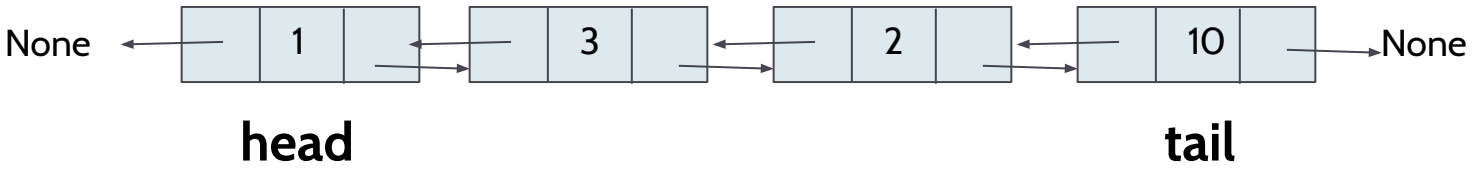
2.3.2. Implementing a List using a doubly linked list (addLast method)

1) **Create a new node.**
2) Link the new node to the list
3) Update the tail reference
4) Increase the size

for example, l.addLast(4)

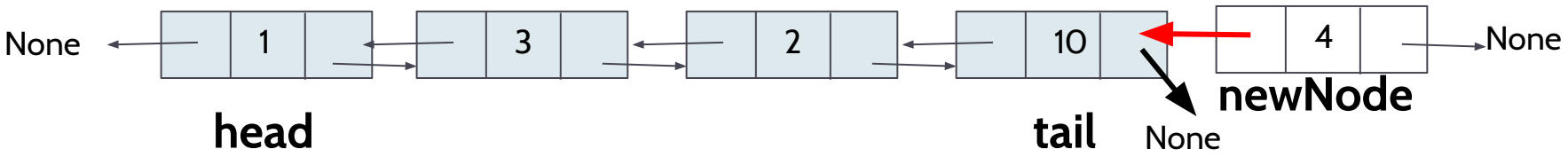
newNode=Node(e)





2.3.2. Implementing a List using a doubly linked list (addLast method)

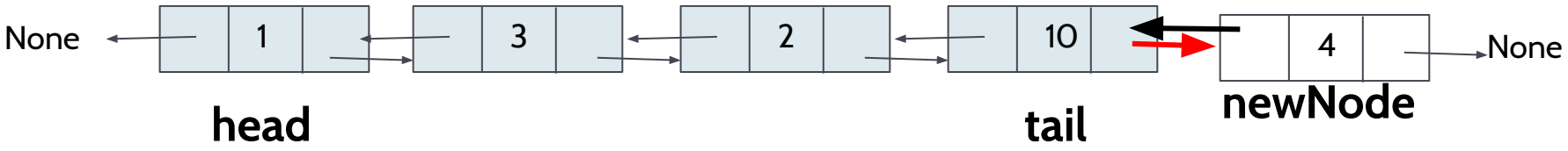
1) Create a new node.
2) **Link the new node to the list**
3) Update the tail reference
4) Increase the size



newNode.prev=L.tail

2.3.2. Implementing a List using a doubly linked list (addLast method)

1) Create a new node.
2) **Link the new node to the list**
3) Update the tail reference
4) Increase the size

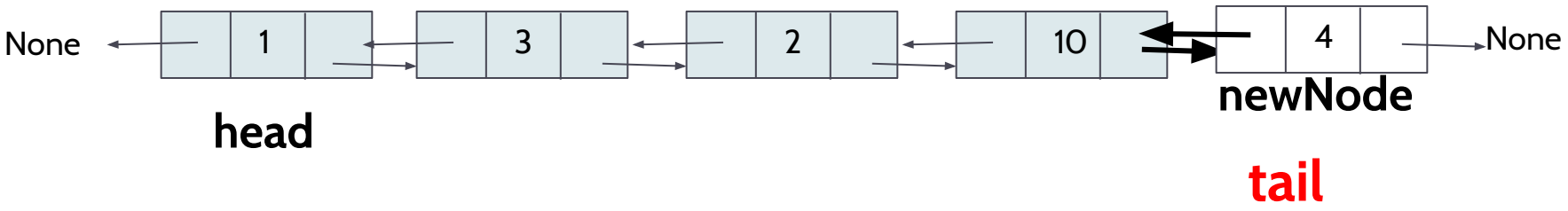


newNode.prev=L.tail

L.tail.next = newNode

2.3.2. Implementing a List using a doubly linked list (addLast method)

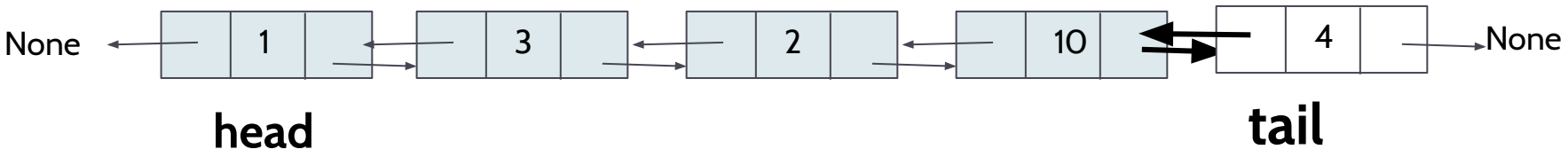
1) Create a new node.
2) Link the new node to the list
3) **Update the tail reference**
4) Increase the size



L.tail = newNode

2.3.2. Implementing a List using a doubly linked list (addLast method)

1) Create a new node.
2) Link the new node to the list
3) Update the tail reference
4) **Increase the size**



L.size = L.size + 1

2.3.2. Implementing a List using a doubly linked list (addLast method, pseudo-code)

```
Algorithm addLast(L,e):
    newNode=Node(e)
    newNode.prev=L.tail
    L.tail.next=newNode
    L.tail=newNode
    L.size=L.size+1
```

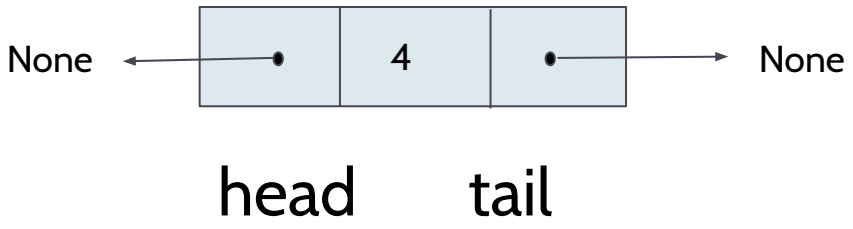
2.3.2. Implementing a List using a doubly linked list (addLast method, pseudo-code)

```
Algorithm addLast(L,e):
    newNode=Node(e)
    newNode.prev=L.tail
    L.tail.next=newNode
    L.tail=newNode
    L.size=L.size+1
```

Does it work when the list is empty?

2.3.2. Implementing a List using a doubly linked list (addLast method, pseudo-code)

- **If the list is empty:**
  1) Create a new node.
  2) Head and tail references must point to the new node.
  3) Increase the size



2.3.2. Implementing a List using a doubly linked list (addLast method, pseudo-code)

```
Algorithm addLast(L,e):
   if L.head=None:
      newNode=Node(e)
      L.tail=newNode
      L.head=newNode
      L.size=L.size+1
   else:
      …
```

2.3.2. Implementing a List using a doubly linked list (addLast method, pseudo-code)

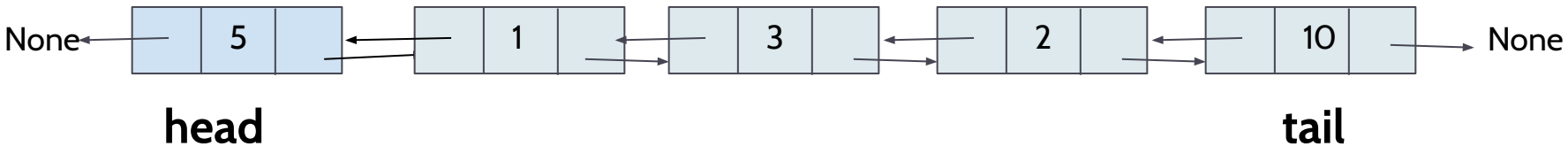
```
Algorithm addLast(L,e):
   newNode=Node(e)

   if L.head=None:
      L.head=newNode
   else:
      newNode.prev=L.tail
      L.tail.next=newNode

   L.tail=newNode
   L.size=L.size+1
```

2.3.2. Implementing a List using a doubly linked list

- addFirst(L,e) adds the element e at the front of the list L.
- addLast(L,e) adds the element e at the tail of the list L.
- **removeFirst(L) removes the first element of the list L. It returns the element.**
- removeLast(L) removes the last element of the list L. It returns the element.
- insertAt(L,index,e) inserts the element e at the index position of the list.

2.3.2. Implementing a List using a doubly linked list (removeFirst method)

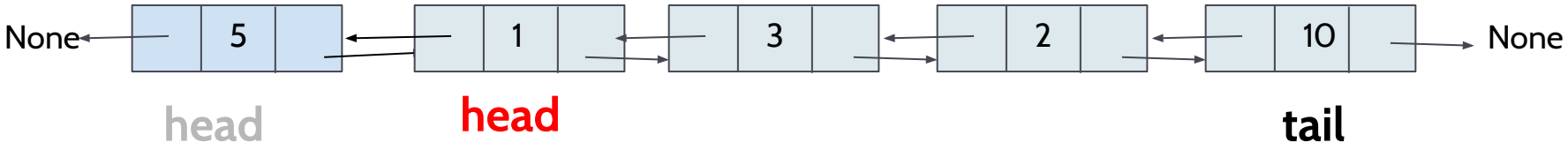
1) **Store the element into a variable**
2) Update the head reference
3) Update the head.prev reference to None
4) Decrease the size
5) Return the result.



result=L.head.element

2.3.2. Implementing a List using a doubly linked list (removeFirst method)

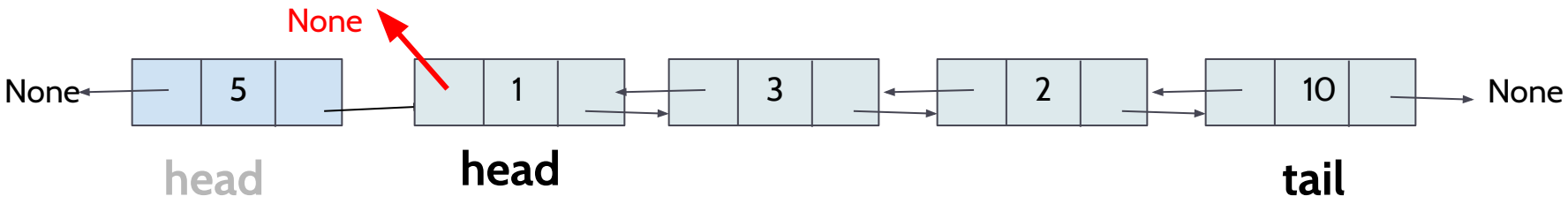
1) Store the element into a variable
2) **Update the head reference**
3) Update the head.prev reference to None
4) Decrease the size
5) Return the result



L.head=L.head.next

2.3.2. Implementing a List using a doubly linked list (removeFirst method)

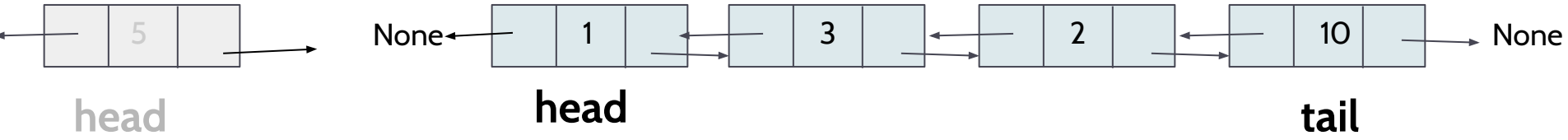
1) Store the element into a variable
2) Update the head reference
3) **Update the head.prev reference to None**
4) Decrease the size
5) Return the result



L.head.prev=None

2.3.2. Implementing a List using a doubly linked list (removeFirst method)

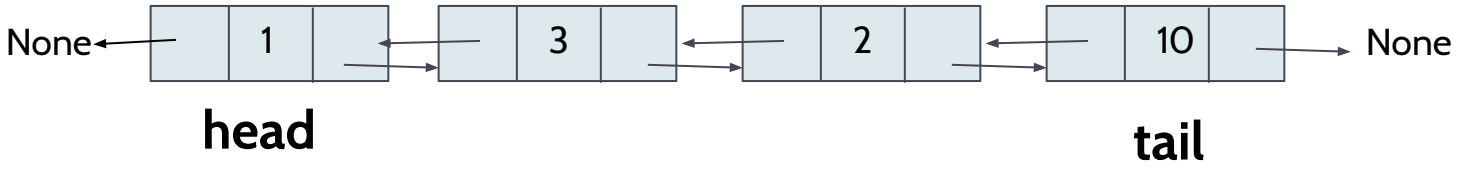
1) Store the element into a variable
2) Update the head reference
3) Update the head.prev reference to None
4) **Decrease the size**
5) Return the result



L.size = L.size - 1

2.3.2. Implementing a List using a doubly linked list (removeFirst method)

1) Store the element into a variable
2) Update the head reference
3) Update the head.prev reference to None
4) Decrease the size
5) **Return the result**



return result

2.3.2. Implementing a List using a doubly linked list (removeFirst method, pseudo-code)

```
Algorithm removeFirst(L):
    result=L.head.element
    L.head= L.head.next
    L.head.prev = None
    L.size=L.size-1
    return result
```

2.3.2. Implementing a List using a doubly linked list (removeFirst method, pseudo-code)

```
Algorithm removeFirst(L):
    If L.isEmpty():
        “show an error”
        return None
    result=L.head.element
    L.head= L.head.next
    L.head.prev = None
    L.size=L.size-1
    return result
```

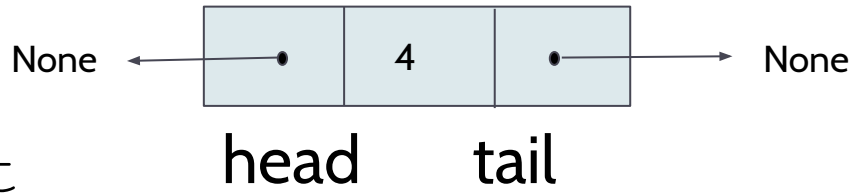
2.3.2. Implementing a List using a doubly linked list (removeFirst method, pseudo-code)

```
Algorithm removeFirst(L):
   If L.isEmpty():
      “show an error”
      return None
   result=L.head.element
   L.head= L.head.next
   L.head.prev = None
   L.size=L.size-1
   return result
```

For what case does this algorithm does not work?

2.3.2. Implementing a List using a doubly linked list (removeFirst method, pseudo-code)

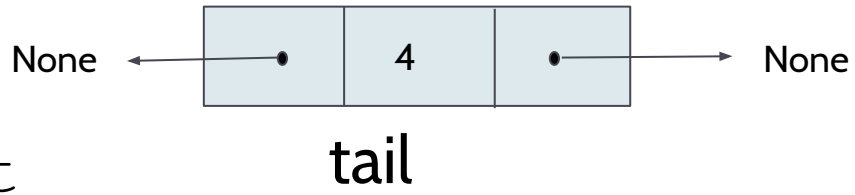
```
Algorithm removeFirst(L):
   If L.isEmpty():
      "show an error"
      return None
   result=L.head.element
   L.head= L.head.next
   L.head.prev = None
   L.size=L.size-1
   return result
```



Special case: the list only has one node

2.3.2. Implementing a List using a doubly linked list (removeFirst method, pseudo-code)

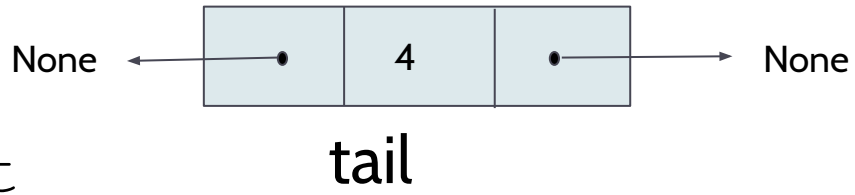
```
Algorithm removeFirst(L):
    If L.isEmpty():
        "show an error"
        return None
    result=L.head.element
    L.head= L.head.next
    L.head.prev = None
    L.size=L.size-1
    return result
```



Special case: the list only has one node. The tail reference is not updated!!!

2.3.2. Implementing a List using a doubly linked list (removeFirst method, pseudo-code)

```
Algorithm removeFirst(L):
    If L.isEmpty():
        "show an error"
        return None
    result=L.head.element
    L.head= L.head.next
    if L.head is None:
        L.tail=None
    else:
        L.head.prev = None
    L.size=L.size-1
    return result
```



head → None

tail → None

2.3.2. Implementing a List using a doubly linked list (removeFirst method, pseudo-code)

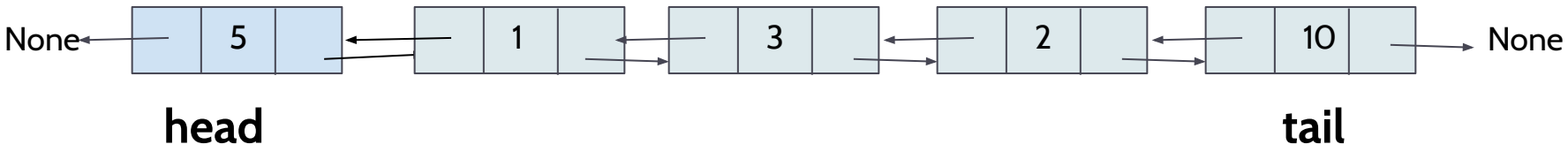
```
Algorithm removeFirst(L):
   If L.isEmpty():
      "show an error"
      return None
   result=L.head.element
   L.head= L.head.next
   if L.head is None:
      L.tail=None
   else:
      L.head.prev = None
   L.size=L.size-1
   return result
```

2.3.2. Implementing a List using a doubly linked list

- addFirst(L,e) adds the element e at the front of the list L.
- addLast(L,e) adds the element e at the tail of the list L.
- removeFirst(L) removes the first element of the list L. It returns the element.
- **removeLast(L) removes the last element of the list L. It returns the element.**
- insertAt(L,index,e) inserts the element e at the index position of the list.

2.3.2. Implementing a List using a doubly linked list (removeLast method)

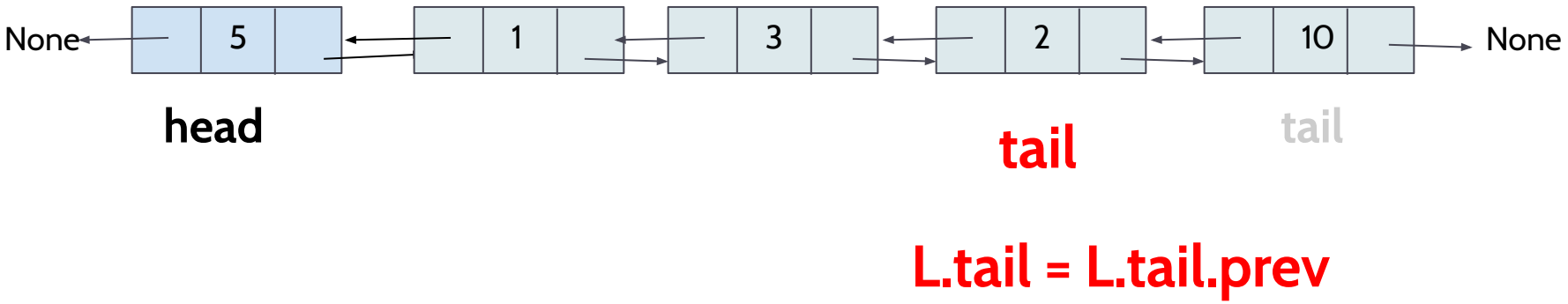
1) **Store the element into a variable**
2) Update the tail reference
3) Update the tail.next reference to None
4) Decrease the size
5) Return the result.



result=L.tail.element

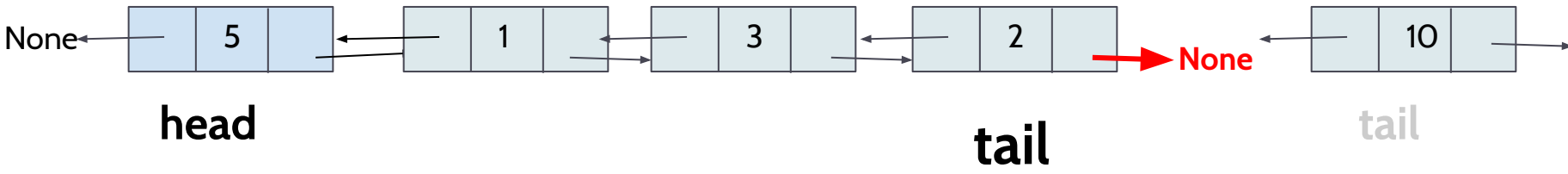
2.3.2. Implementing a List using a doubly linked list (removeLast method)

1) Store the element into a variable
2) **Update the tail reference**
3) Update the tail.next reference to None
4) Decrease the size
5) Return the result.



2.3.2. Implementing a List using a doubly linked list (removeLast method)

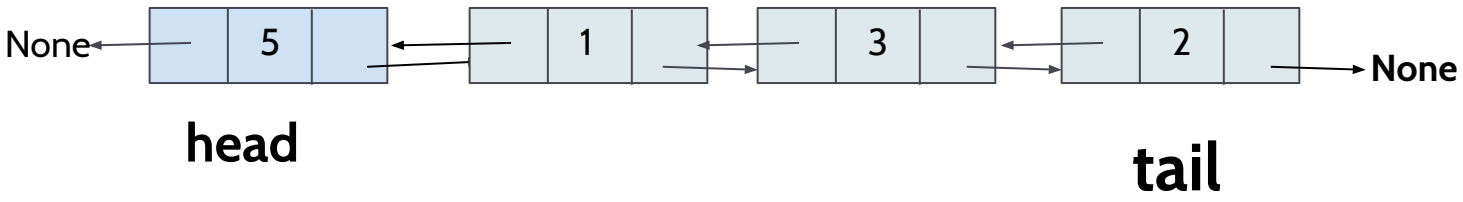
1) Store the element into a variable
2) Update the tail reference
3) **Update the tail.next reference to None**
4) Decrease the size
5) Return the result.



L.tail.next = None

2.3.2. Implementing a List using a doubly linked list (removeLast method)

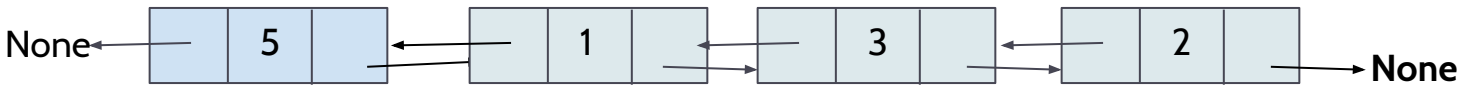
1) Store the element into a variable
2) Update the tail reference
3) Update the tail.next reference to None
4) **Decrease the size**
5) Return the result.



L.size = L.size -1

2.3.2. Implementing a List using a doubly linked list (removeLast method)

1) Store the element into a variable
2) Update the tail reference
3) Update the tail.next reference to None
4) Decrease the size
5) **Return the result.**



head

tail

return result

2.3.2. Implementing a List using a doubly linked list (removeLast method, pseudo-code)

```
Algorithm removeLast(L):
   If L.isEmpty():
      “show an error”
      return None
   result=L.tail.element
   L.tail= L.tail.prev
   L.tail.next = None
   L.size=L.size-1
   return result
```

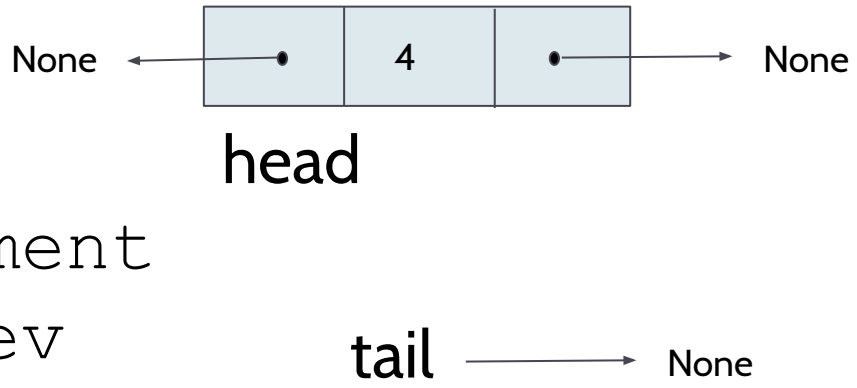
2.3.2. Implementing a List using a doubly linked list (removeLast method, pseudo-code)

```
Algorithm removeLast(L):
    If L.isEmpty():
        “show an error”
        return None
    result=L.tail.element
    L.tail= L.tail.prev
    L.tail.next = None
    L.size=L.size-1
    return result
```

This doen’s work if the list only has one node

2.3.2. Implementing a List using a doubly linked list (removeLast method, pseudo-code)

```
Algorithm removeLast(L):
   If L.isEmpty():
      "show an error"
      return None
   result=L.tail.element
   L.tail= L.tail.prev
   L.tail.next = None
   L.size=L.size-1
   return result
```



Special case: The list only has one node
The head reference is not updated!!!

2.3.2. Implementing a List using a doubly linked list (removeLast method, pseudo-code)

```
Algorithm removeLast(L):
   If L.isEmpty():
      “show an error”
      return None
   result=L.tail.element
   L.tail= L.tail.prev
   if L.tail is None:
      L.head = None
   else:
      L.tail.next = None
   L.size=L.size-1
   return result
```

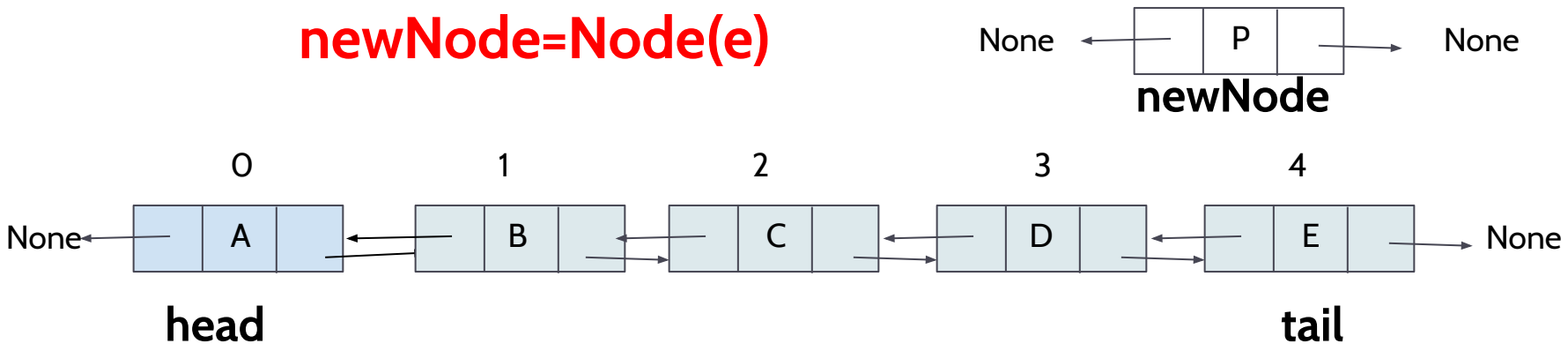
2.3.2. Implementing a List using a doubly linked list

- addFirst(L,e) adds the element e at the front of the list L.
- addLast(L,e) adds the element e at the tail of the list L.
- removeFirst(L) removes the first element of the list L. It returns the element.
- removeLast(L) removes the last element of the list L. It returns the element.
- **insertAt(L,index,e) inserts the element e at the index position of the list.**

2.3.2. Implementing a List using a doubly linked list (insertAt method)

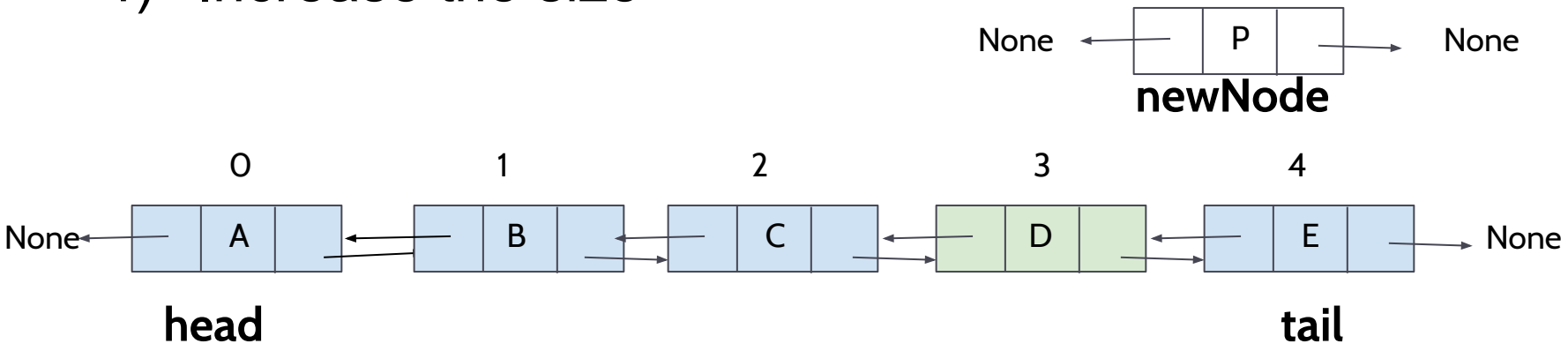
1) **Create a new node**
2) Traverse to the node at the given index
3) Link the new node to the list
4) Increase the size

for example, l.insertAt(3,'P')



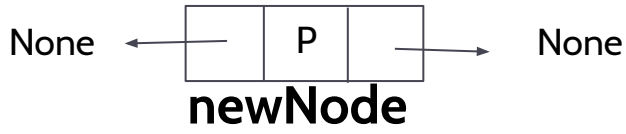
2.3.2. Implementing a List using a doubly linked list (insertAt method)

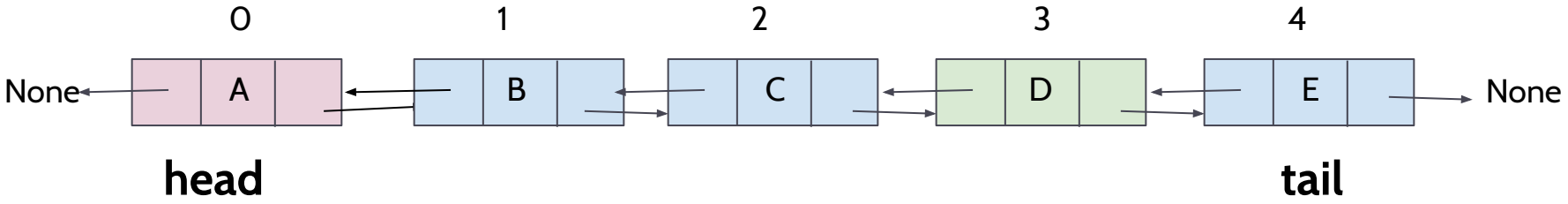
1) Create a new node
2) **Traverse to the node at the given index**
3) Link the new node to the list
4) Increase the size



2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) **Traverse to the node at the given index**
3) Link the new node to the list
4) Increase the size





aux,
i=0

aux= L.head

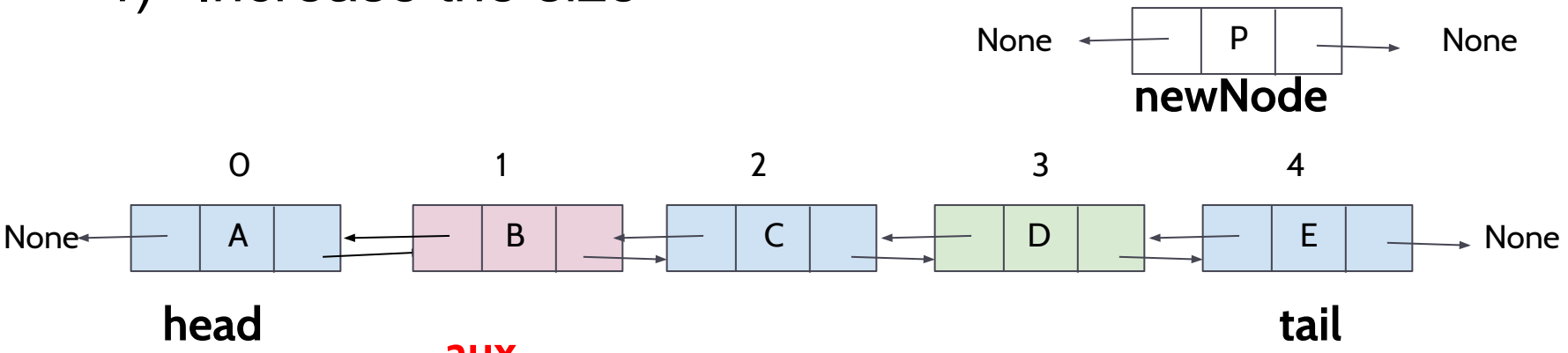
while i <index:

aux=aux.next

i=i+1

2.3.2. Implementing a List using a doubly linked list (insertAt method)

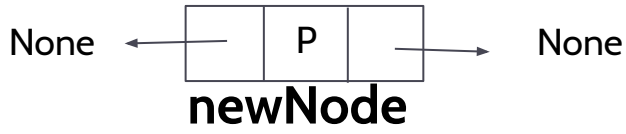
1) Create a new node
2) **Traverse to the node at the given index**
3) Link the new node to the list
4) Increase the size

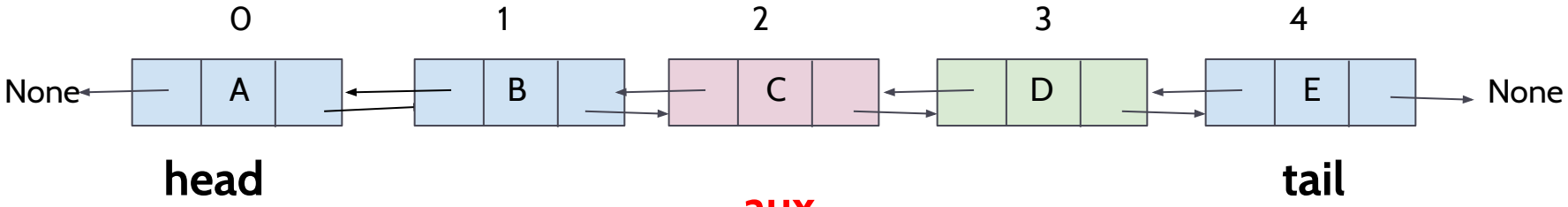



```
while i <index:
    aux=aux.next
    i=i+1
```

2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) **Traverse to the node at the given index**
3) Link the new node to the list
4) Increase the size



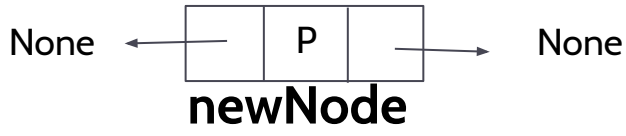


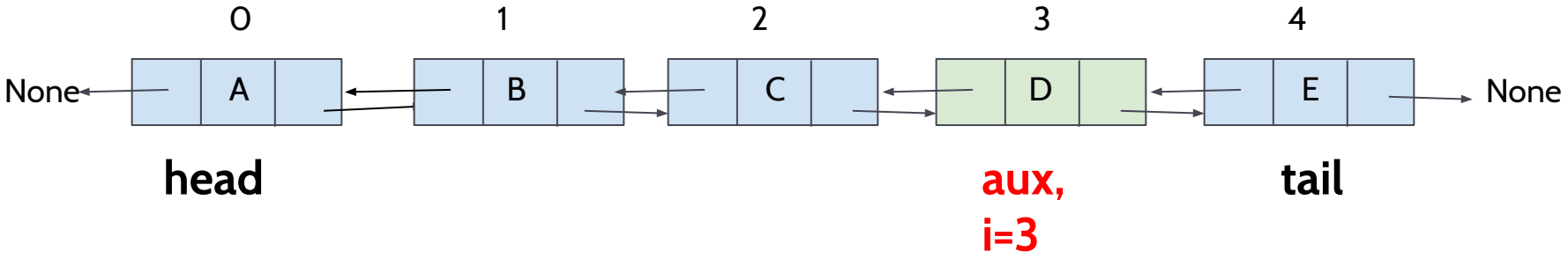
aux,
i=2

```
while i <index:
    aux=aux.next
    i=i+1
```

2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) **Traverse to the node at the given index**
3) Link the new node to the list
4) Increase the size





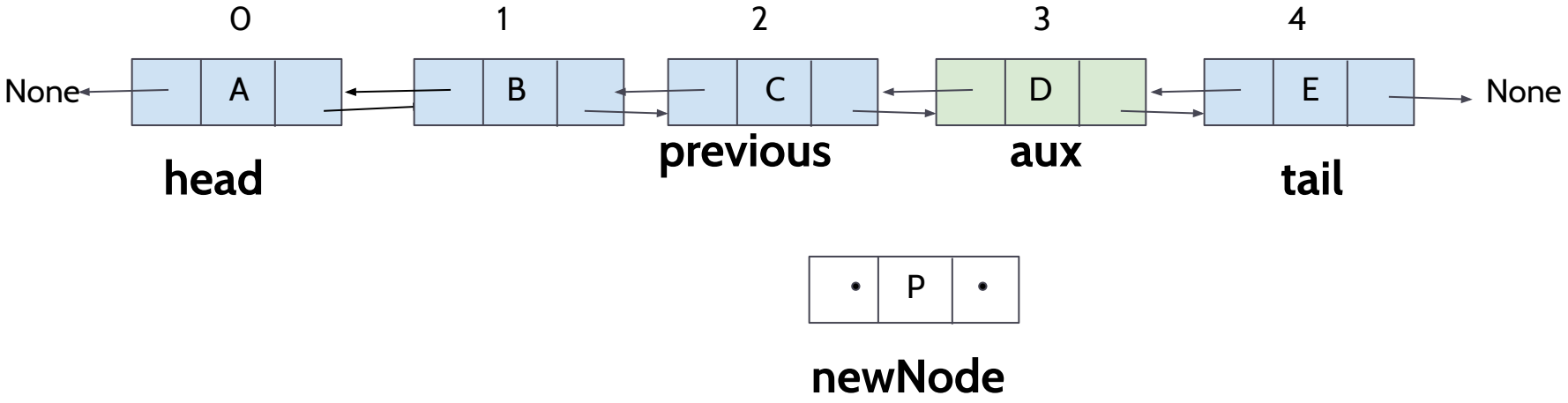
while i <index: **False**

aux=aux.next

i=i+1

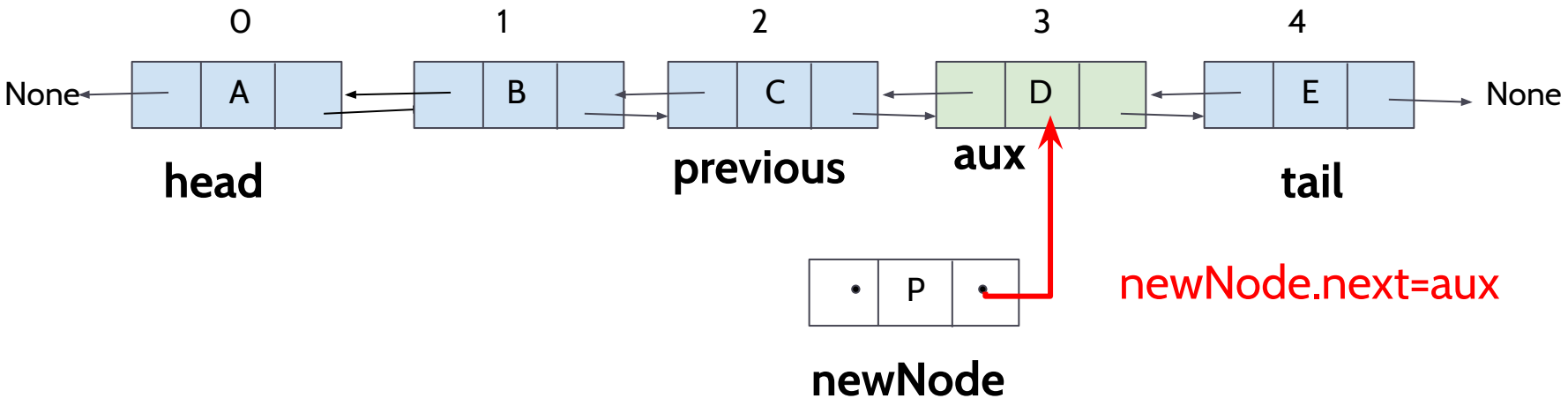
2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) Traverse to the node at the given index
3) **Link the new node to the list**
4) Increase the size



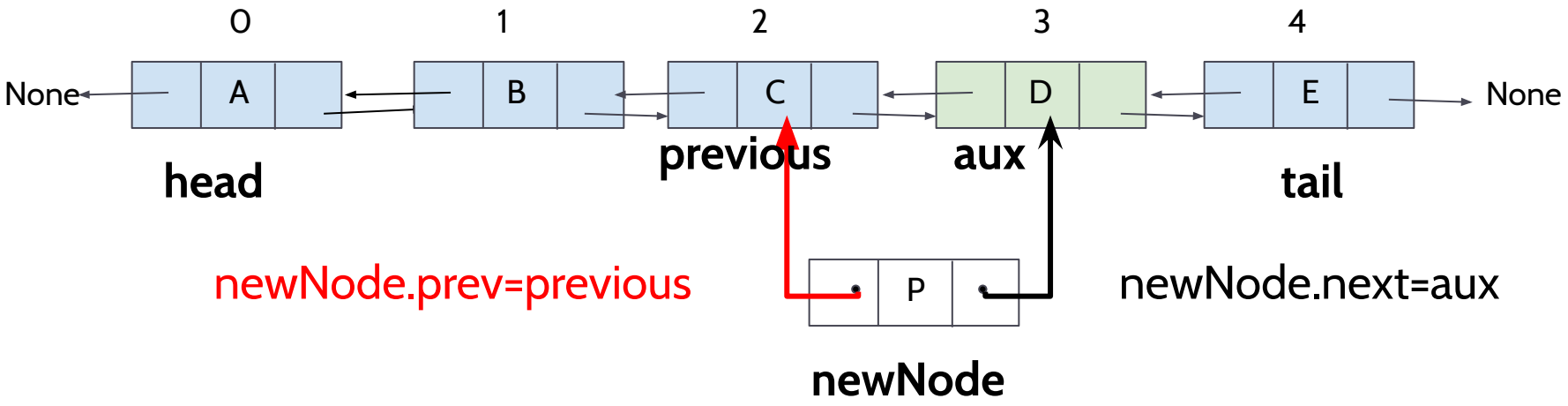
2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) Traverse to the node at the given index
3) **Link the new node to the list**
4) Increase the size



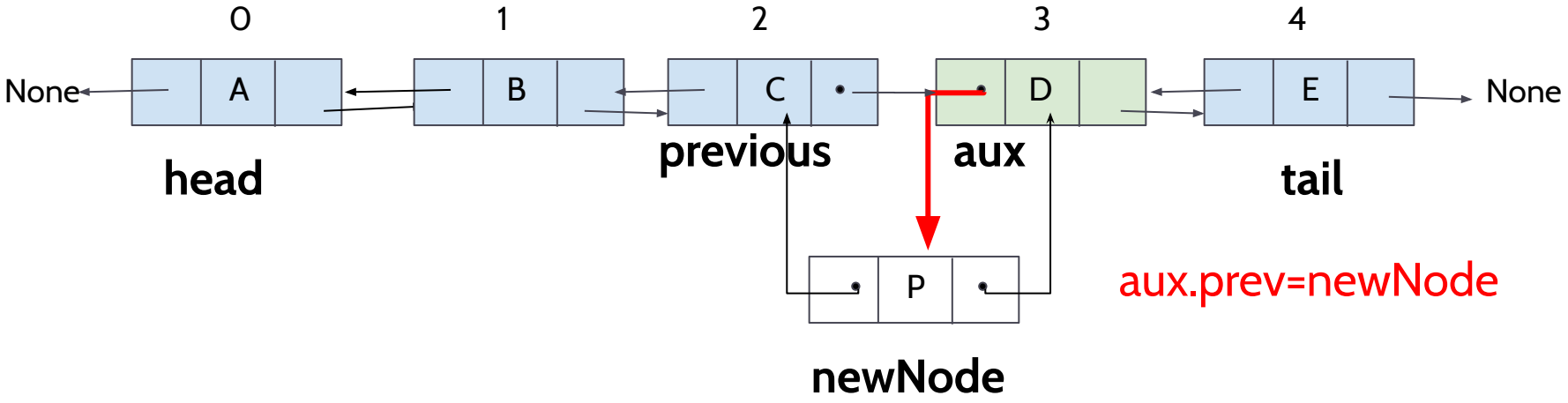
2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) Traverse to the node at the given index
3) **Link the new node to the list**
4) Increase the size



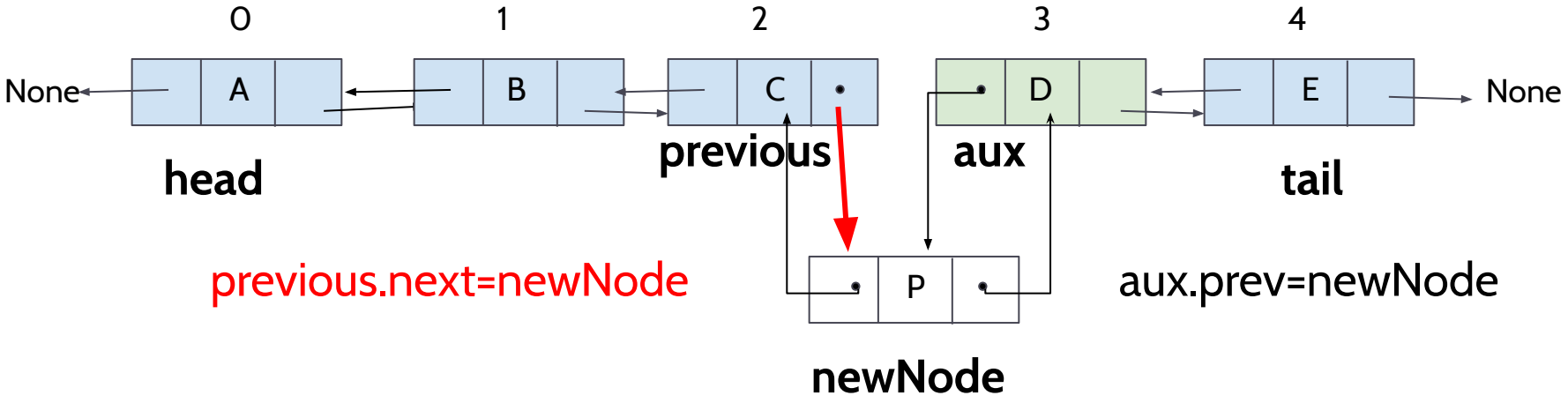
2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) Traverse to the node at the given index
3) **Link the new node to the list**
4) Increase the size



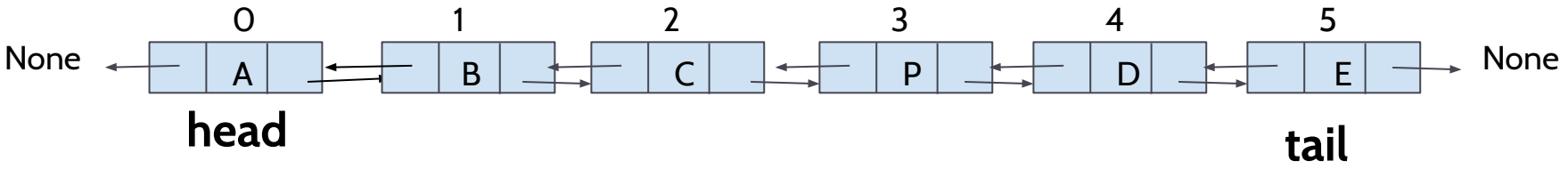
2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) Traverse to the node at the given index
3) **Link the new node to the list**
4) Increase the size



2.3.2. Implementing a List using a doubly linked list (insertAt method)

1) Create a new node
2) Traverse to the node at the given index
3) Link the new node to the list
4) **Increase the size**



L.size = L.size + 1

2.3.2. Implementing a List using a doubly linked list (insertAt method, pseudo-code)

```
Algorithm insertAt(L,index,e):
    If index<0 or index>L.size:
        “show an error”
        return

    i=0
    aux=L.head
    while i<index:
        aux=aux.next
        i=i+1
    #aux is the node at the index position
    previous=aux.prev
    newNode=Node(e)
    newNode.next=aux
    newNode.prev=previous
    aux.prev=newNode
    previous.next=newNode
    L.size= L.size+1
```

2.3.2. Implementing a List using a doubly linked list (insertAt method, pseudo-code)

```
Algorithm insertAt(L,index,e):
    If index<0 or index>L.size:
        "show an error"
        return

    i=0
    aux=L.head
    while i<index:
        aux=aux.next
        i=i+1
    #aux is the node at the index position
    previous=aux.prev
    newNode=Node(e)
    newNode.next=aux
    newNode.prev=previous
    aux.prev=newNode
    previous.next=newNode
    L.size= L.size+1
```

Warning!!!! If index=0, previous is None

Warning!!!! If index=size, aux is None

2.3.2. Implementing a List using a doubly linked list (insertAt method, pseudo-code)

```
Algorithm insertAt(L,index,e):
    If index<0 or index>L.size:
        "show an error"
        return
    if index==0
        L.addFirst(e)
    else if index==L.size
        L.addLast(e)
    else
        i=0
        aux=L.head
        while i<index:
            aux=aux.next
            i=i+1
        #aux is the node at the index position
        previous=aux.prev
        newNode=Node(e)
        newNode.next=aux
        newNode.prev=previous
        aux.prev=newNode
        previous.next=newNode
        L.size= L.size+1
```

2.3.2. Implementing a List using a doubly linked list (exercises)

- Implement the rest of the methods:
  - getAt(L,index) returns the element at the index position
  - contains(L,e) returns the first index of the element in the list. If e does not exist, then it returns -1.
  - removeAt(L,index) removes the element at the index position.
  - show(L,opc):
    - If opc=0, it prints the elements of the list.
    - if opc=1, it prints the elements of the list in reverse order (from right to left).

2.3.2. Implementing a List using a doubly linked list (exercises)

- A palindrome word is one that reads the same backward as forward.
- Examples:
  - *Anna, Level, Civic, Madam, Noon.*
- Implement a Python function that takes a word and returns true if it is palindrome, else false.
- In your solution, you **have to use a doubly linked list** where each node contains only one character of the input word.