Lecture 14

Linked Lists

Motivation

- Suppose I have an array with 5 elements: 1, 3, 5, 7, 9

```
>> arr = np.array([1, 3, 5, 7, 9])
>> len(arr)
5
```

- I want to insert -1 before 1. What should I do?
  - Talk to each other

Motivation

- Suppose I have an array with 5 elements: 1, 3, 5, 7, 9

```
>> arr = np.array([1, 3, 5, 7, 9])
>> len(arr)
5
```

- I want to insert -1 before 1. What should I do?
  - Create a new bigger array
  - Shift 1, 3, 5, 7 , 9 to the right
  - Insert -1 before 1.
  - Change reference from old array to a new one.

Motivation

- I want to insert -1 before 1.
  - Create a new bigger array
  - Shift 1, 3, 5, 7 , 9 to the right
  - Insert 0 before 1.
  - Change reference from old array to a new one.

```
>>> arr = np.array([1, 3, 5, 7, 9])

>>> bigger = np.zeros(6, dtype = int)
```

Motivation

- I want to insert -1 before 1.
  - Create a new bigger array
  - Shift 1, 3, 5, 7 , 9 to the right
  - Insert 0 before 1.
  - Change reference from old array to a new one.

```
>>> arr = np.array([1, 3, 5, 7, 9])

>>> bigger = np.zeros(6, dtype = int)

>>> for i in range (1, 6):
        bigger[i] = arr[i-1]


>>> bigger
array([0, 1, 3, 4, 5, 7])
```

Motivation

- I want to insert -1 before 1.
  - Create a new bigger array
  - Shift 1, 3, 5, 7 , 9 to the right
  - Insert 0 before 1.
  - Change reference from old array to a new one.

```
>>> arr = np.array([1, 3, 5, 7, 9])

>>> bigger = np.zeros(6, dtype = int)

>>> for i in range (1, 6):
    bigger[i] = arr[i-1]

>>> bigger
array([0, 1, 3, 4, 5, 7])

>>> bigger[0] = -1
>>> arr = bigger
>>> arr
array([-1, 1, 3, 4, 5, 7])
```

Motivation

- I want to insert -1 before 1.
  - Create a new bigger array
  - Shift 1, 3, 5, 7 , 9 to the right
  - Insert 0 before 1.
  - Change reference from old array to a new one.

What is the complexity of this algorithm?

A: Theta (1)
B: Theta (log n)
C: Theta (n)
D: Theta (n^2)
E: None of the above

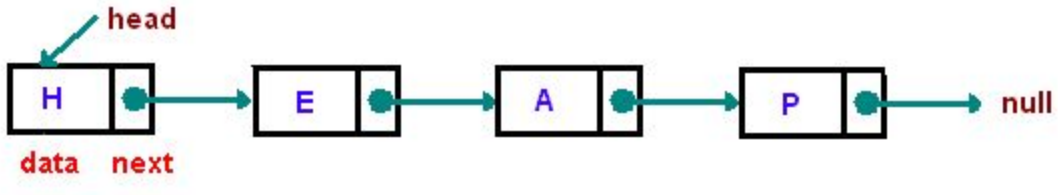
```
>>> arr = np.array([1, 3, 5, 7, 9])

>>> bigger = np.zeros(6, dtype = int)

>>> for i in range (1, 6):
        bigger[i] = arr[i-1]

>>> bigger
array([0, 1, 3, 4, 5, 7])

>>> bigger[0] = -1
>>> arr = bigger
>>> arr
array([-1, 1, 3, 4, 5, 7])
```

Disadvantage of using arrays

- arrays are static structures
  - cannot be easily extended or reduced to fit the data set
- Once you created an array, it can’t be changed anymore.
- You have to create a new one each time
  - Python lists are based on arrays.
- Arrays are also expensive to maintain new insertions and deletions

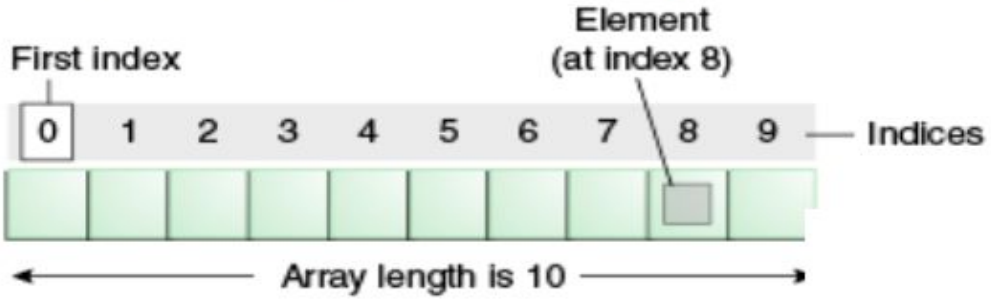
Linked Lists

- A linked list is a linear data structure where each element is a separate object.

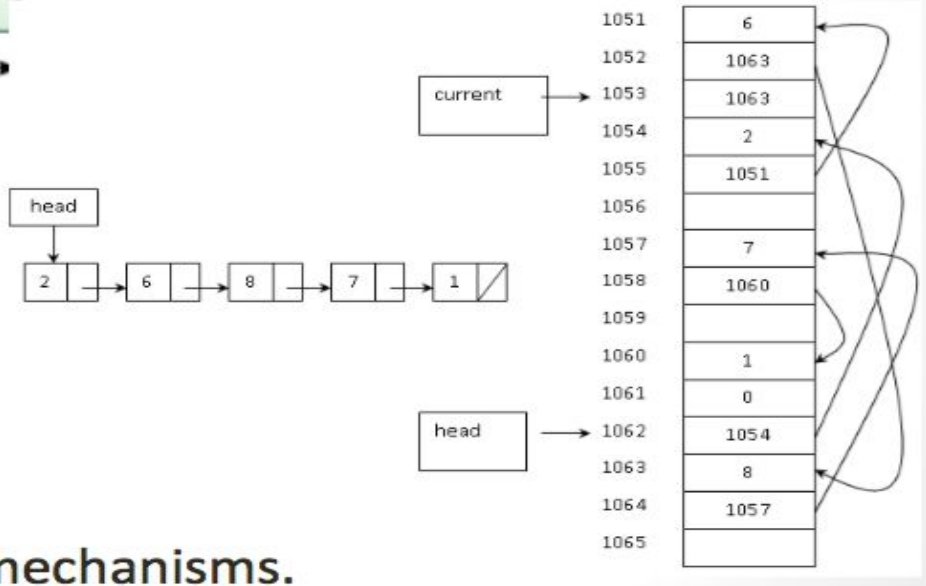


- A linked list is a **dynamic** data structure.
  - The number of nodes in a list is not fixed and can grow and shrink on demand.
- Any application which has to deal with an unknown number of objects will need to use a linked list.

- There are two fundamental kinds of data structures:
  - array of contiguous memory locations



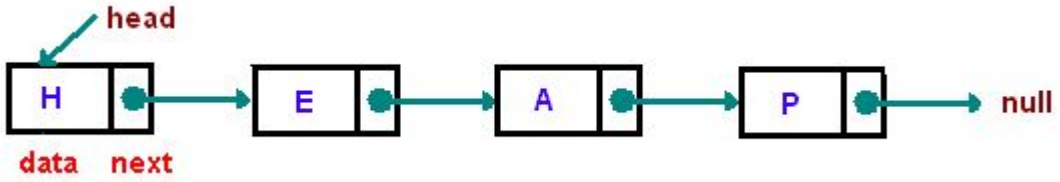
  - linked structures.



  - You can even combine the two mechanisms.

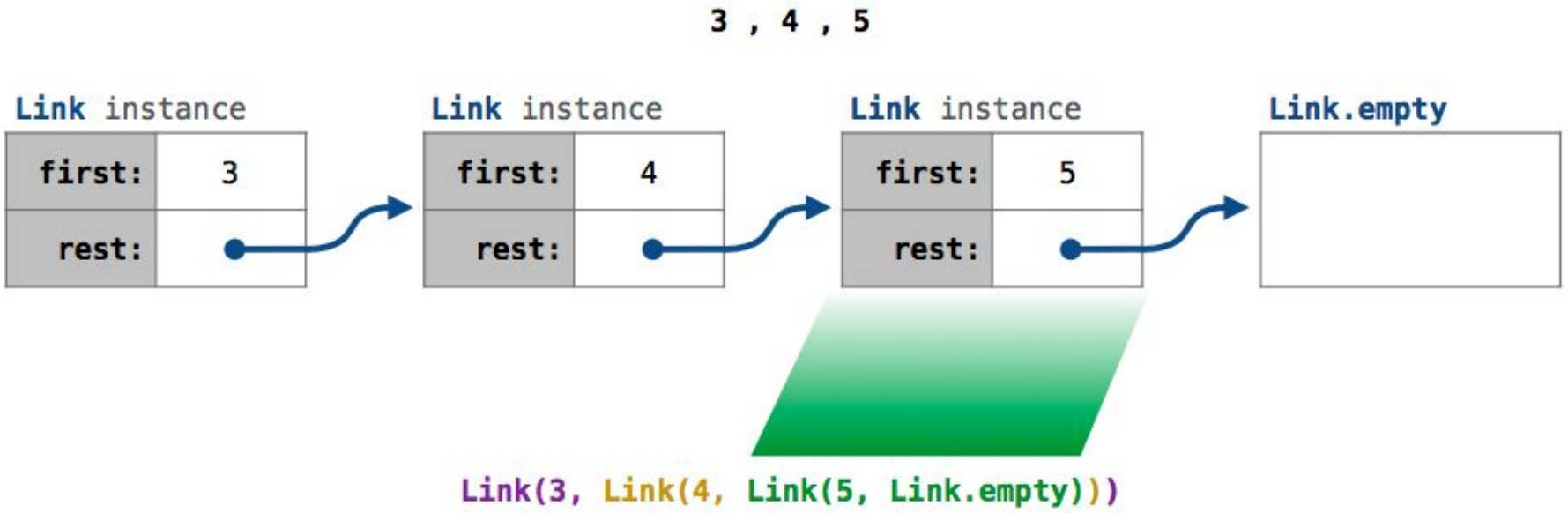
Disadvantage of Linked Lists

- One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements:
- If you want to access a particular item then you have to start at the head and follow the references until you get to that item.
- Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a *reference* to the next node.



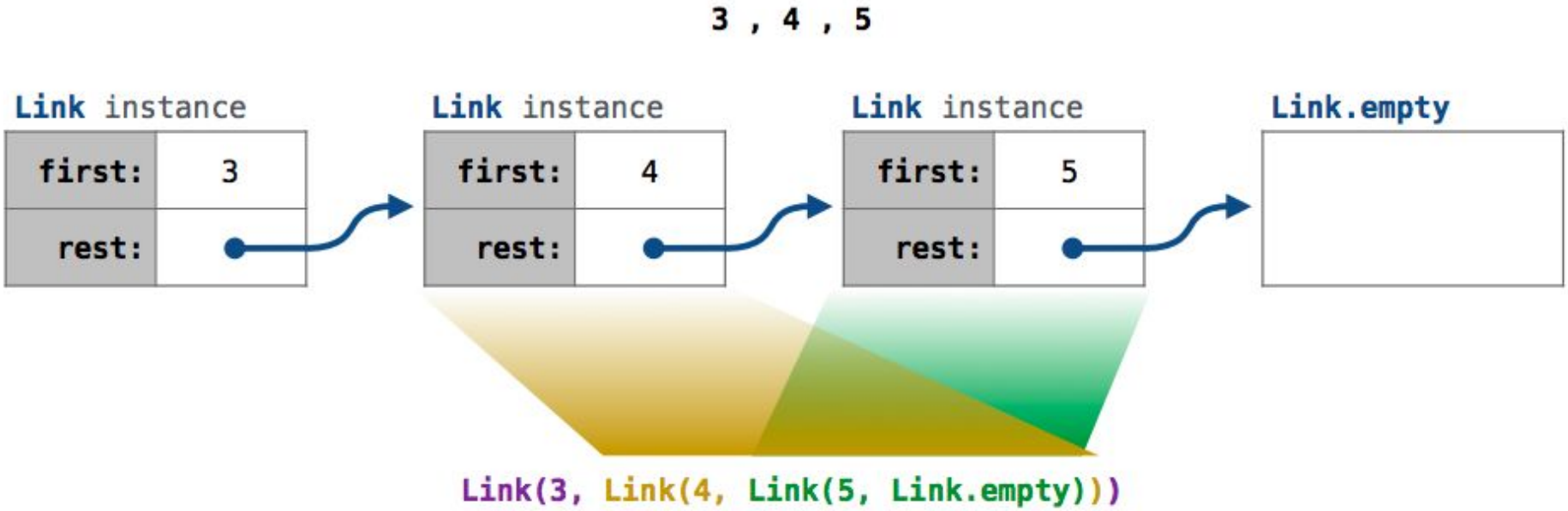
Linked List structure

A linked list is either empty **or** a first value and the rest of the linked list

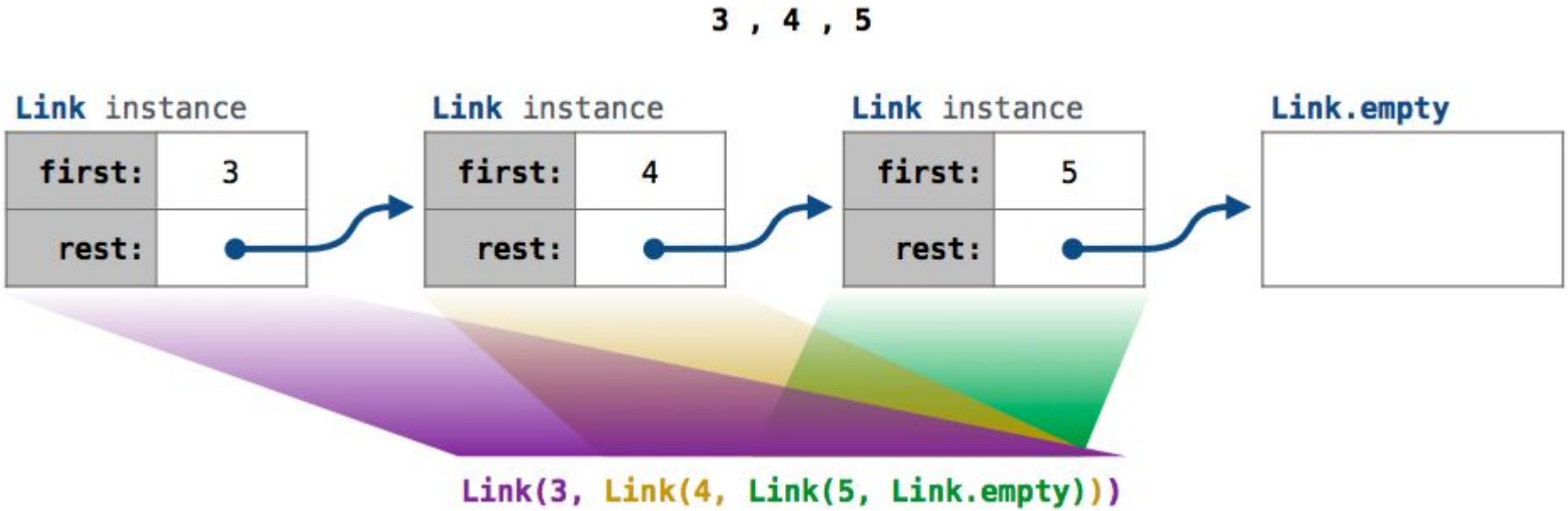


Linked List structure

A linked list is either empty **or** a first value and the rest of the linked list

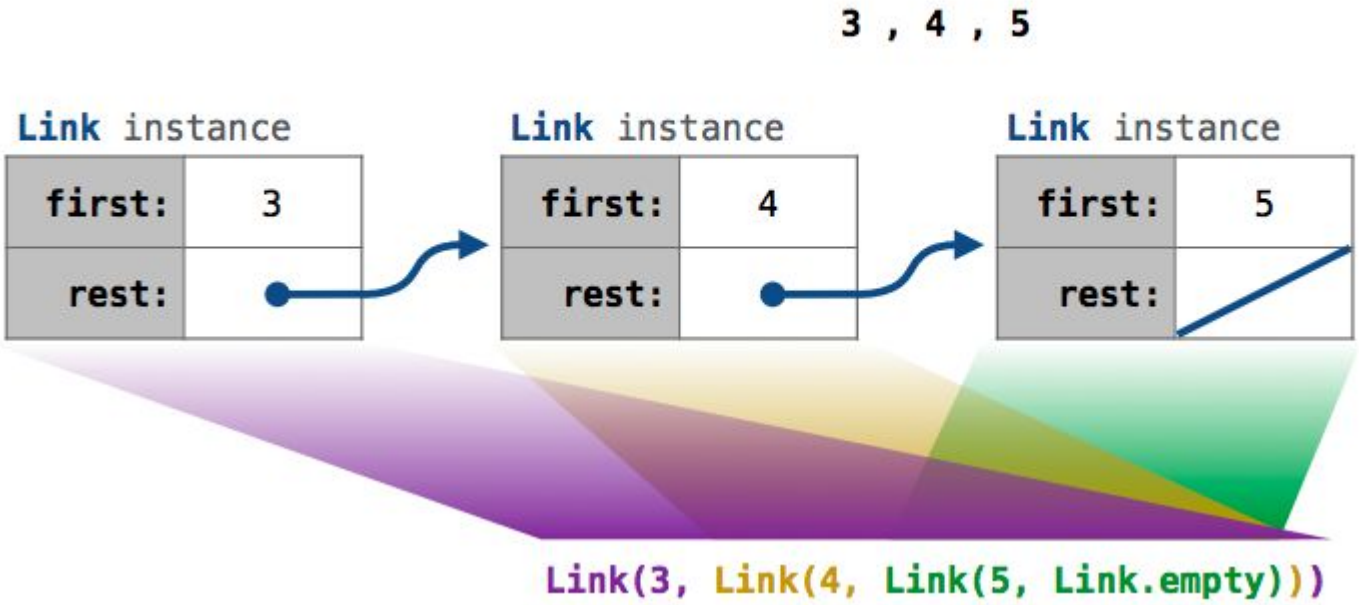


Linked List structure



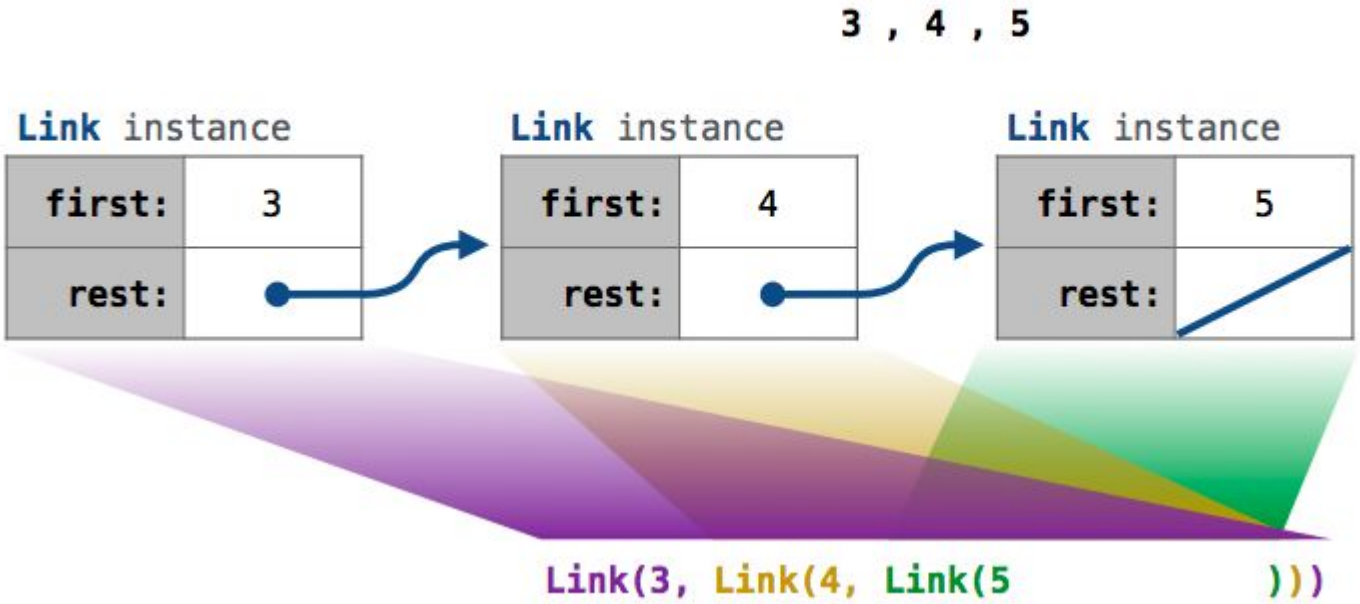
Linked List structure: convention for empty

A linked list is either empty **or** a first value and the rest of the linked list



Linked List structure: default for empty

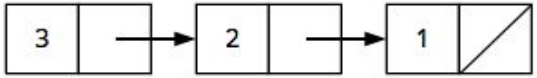
A linked list is either empty **or** a first value and the rest of the linked list



Linked List ADT (Abstract Data Type)

- Abstract Data type (ADT) is a type for objects whose behavior is defined by a set of values and a set of operations.

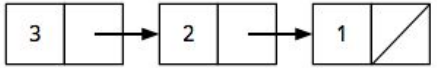
Linked list ADT (Abstract Data Type)



- Abstract Data type (ADT) is a type for objects whose behavior is defined by a set of values and a set of operations.

- `insert()` – Insert an element at any position of the list.

Linked list ADT (Abstract Data Type)



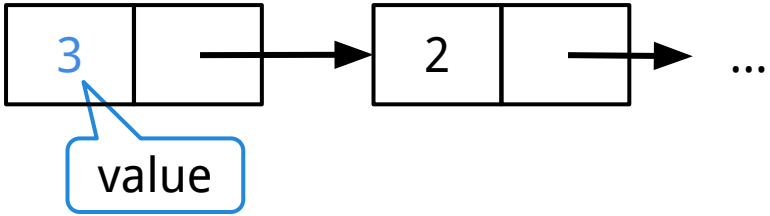
- Abstract Data type (ADT) is a type for objects whose behavior is defined by a set of value and a set of operations.

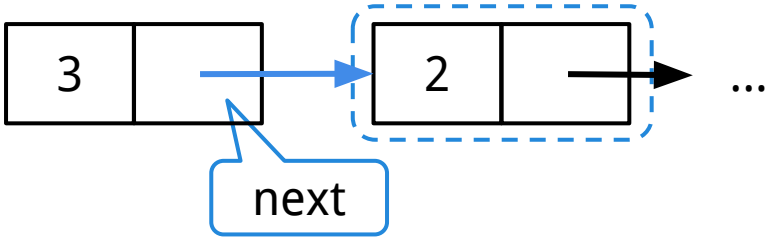
- `insert()` – Insert an element at any position of the list.

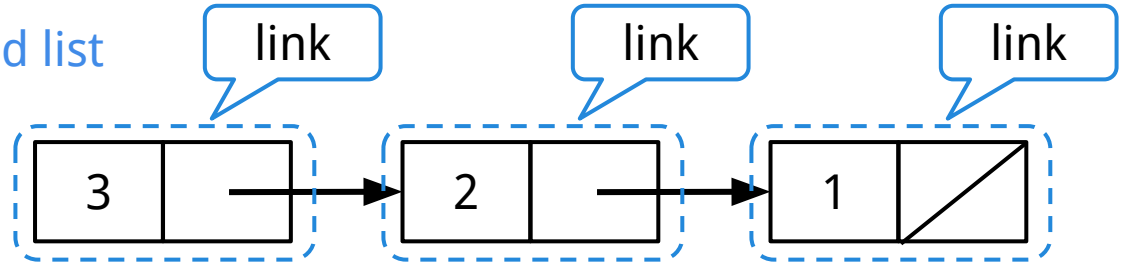
- `remove()` – Remove the first occurrence of any element from a non-empty list.
- removeAt( ) – Remove the element at a specified location from a non-empty list.
- replace( ) – Replace an element at any position by another element.
- size( ) – Return the number of elements in the list.

Linked List ADT: What is a linked list?

- Linked list: a type of list, built from "links" (nodes)
- Links have two parts:
  - value: the element in the node
  - next: the next link in the list
- Defined recursively
  - next of a linked list is another linked list







Linked List class

Linked list class: attributes are passed to `__init__`

```
class LinkNode:

    def __init__(self, value, nxt = None):
```

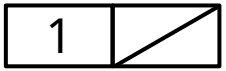
Linked List class

Linked list class: attributes are passed to `__init__`

```
class LinkNode:

    def __init__(self, value, nxt = None):
        self.value = value
        self.next = nxt
```

Linked List ADT: Are these valid linked lists?

```
class LinkNode:

def __init__(self, value, nxt = None):
        self.value = value
        self.next = nxt
```



`LinkNode(1)`

- Valid! (default is None)

`LinkNode()`

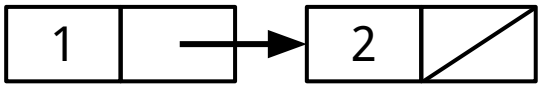
- Invalid: must have a first item, value

`LinkNode(1, 2)`

- Invalid: next must be another link

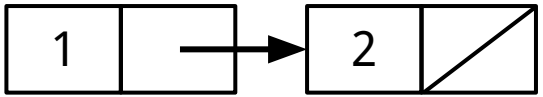
`LinkNode(1, LinkNode(2))`

- Valid!



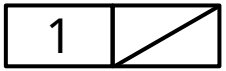
`LinkNode(1, LinkNode(2, None))`

- Valid!



Linked List ADT: Are these valid linked lists?

```
class LinkNode:

def __init__(self, value, nxt = None):
        self.value = value
        self.next = nxt
```



`LinkNode(1)`

- Valid! (default is None)

`LinkNode()`

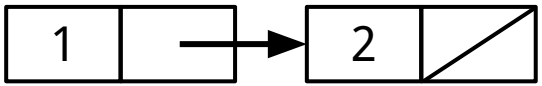
- Invalid: must have a first item, value

`LinkNode(1, 2)`

- Invalid: next must be another link

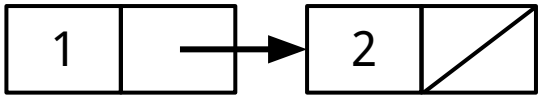
`LinkNode(1, LinkNode(2))`

- Valid!



`LinkNode(1, LinkNode(2, None))`

- Valid!



Demo

Linked List class

Linked list class: attributes are passed to `__init__`

```
class LinkNode:

    def __init__(self, value, nxt = None):
        self.value = value
        self.next = nxt
```

Sanity Check + Demo

```
class LinkNode:

    def __init__(self, value, nxt = None):

        assert isinstance(nxt, LinkNode)

        self.value = value

        self.next = nxt
```

Returns whether next is a LinkNode

Sanity Check + Demo with links, value, next

```
class LinkNode:

    def __init__(self, value, nxt = None):

        assert isinstance(nxt, LinkNode) or nxt is Null

        self.value = value
        self.next = nxt
```

Returns whether
next is a LinkNode

Let's practice with simple functions. Think

Assume there are elements in the list. In the lab you should cover the case where list might be empty.

```
class LinkNode:
    def __init__(self,value,nxt = None):
       assert ...
        self.value = value
        self.next = nxt
```

```
def print_list(lst):



nodes = LinkNode(1, LinkNode(2, None))
print_list(nodes)

1 2
```

Let's practice with simple functions

Assume there are elements in the list. In the lab you should cover the case where list might be empty.

```
class LinkNode:
    def __init__(self,value,nxt = None):
       assert ...
        self.value = value
        self.next = nxt
```

```
def print_list(lst):

    while lst != None:
```

```
nodes = LinkNode(1, LinkNode(2, None))
print_list(nodes)
```

Let's practice with simple functions

Assume there are elements in the list. In the lab you should cover the case where list might be empty.

```
class LinkNode:
    def __init__(self,value,nxt = None):
       assert ...
        self.value = value
        self.next = nxt
```

```
def print_list(lst):

    while lst != None:
        print(lst.value, end = " ")
        lst = lst.next

    print()
```

```
nodes = LinkNode(1, LinkNode(2, None))
print_list(nodes)
```

Let's practice with simple functions

Assume there are elements in the list. In the lab you should cover the case where list might be empty.

```
class LinkNode:
    def __init__(self,value,nxt = None):
       assert ...
        self.value = value
        self.next = nxt
```

```
def print_list(lst):

    while lst != None:
        print(lst.value, end = " ")
        lst = lst.next

    print()
```

Print list?

A: Theta (1)
B: Theta (log n)
C: Theta (n)
D: Theta (n log n)
E: Theta (n ^ 2)

```
nodes = LinkNode(1, LinkNode(2, None))
print_list(nodes)
```

Side question, about “help hour”

I did not come on Sunday because:

A: Time/Day is bad time for me

B: I did not know about it

C: I understand everything (more or less), do not need it

D: Do not believe it is useful

E: Other

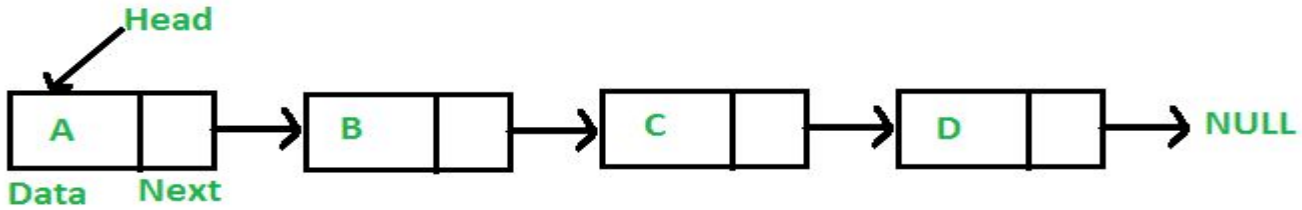
If the time is better, will you be interested

A: No, I’m doing fine

B: Yes, I will come

C: Maybe, depending on the topic

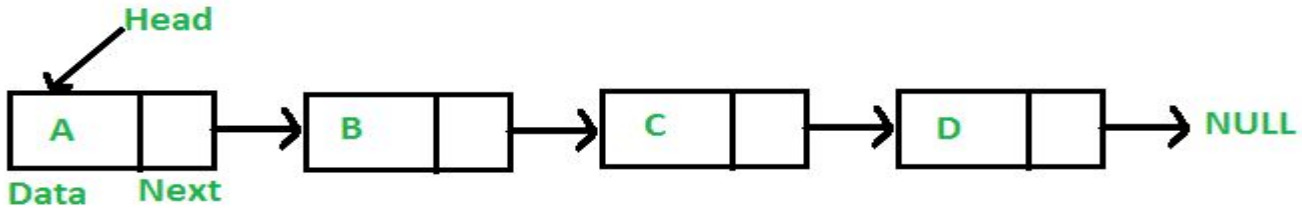
Running time for different operations



Insert new node as the first element in the list:

A: Theta (1)
B: Theta (log n)
C: Theta (n)
D: Theta (n log n)
E: Theta (n ^ 2)

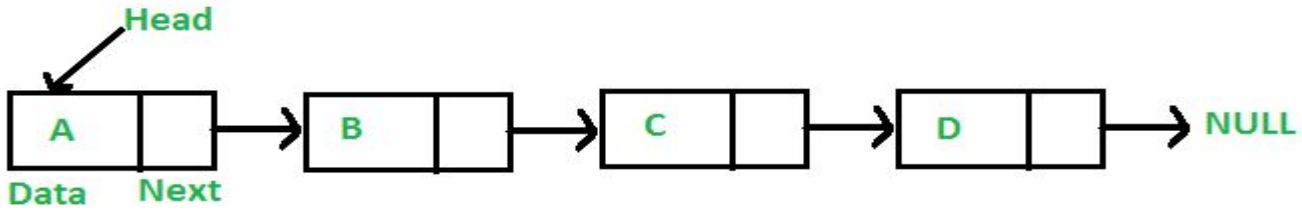
Running time for different operations



Insert new node as the last element in the list:

A: Theta (1)
B: Theta (log n)
C: Theta (n)
D: Theta (n log n)
E: Theta (n ^ 2)

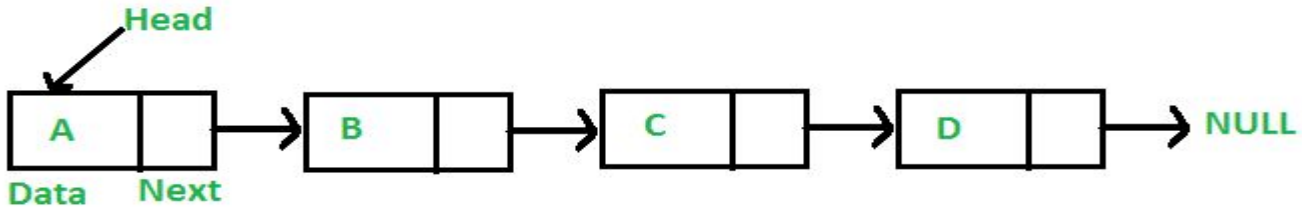
Running time for different operations



Find an element at a given index?

A: Theta (1)
B: Theta (log n)
C: Theta (n)
D: Theta (n log n)
E: Theta (n ^ 2)

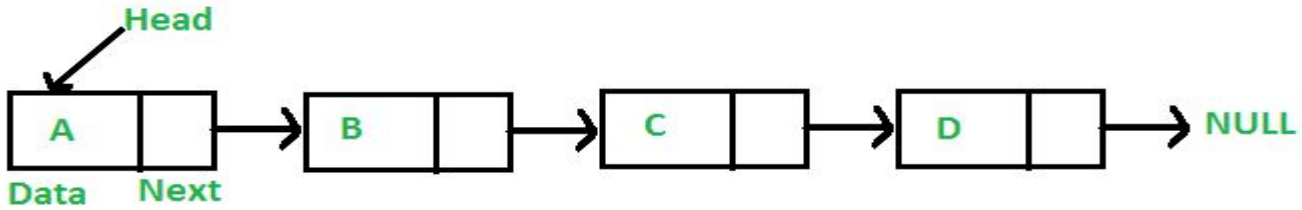
Running time for different operations



Remove the first element?

A: Theta (1)
B: Theta (log n)
C: Theta (n)
D: Theta (n log n)
E: Theta (n ^ 2)

Running time for different operations



Remove the last element?

A: Theta (1)
B: Theta (log n)
C: Theta (n)
D: Theta (n log n)
E: Theta (n ^ 2)

Replace Last element

Write a function that replaces the last element of the list with a given element:

```
def replace_last (lst, elem):
“””
>>> node2 = LinkNode(1, LinkNode(2))
>>> replace_last(node2, 3)
>>> print_list (node2)
1 3

>>> replace_last(LinkNode(None, None), 3)
“””
```

```
def replace_last (lst, elem):
    # check if empty:
```

Replace Last element

Write a function that replaces the last element of the list with a given element:

```
def replace_last (lst, elem):
“””
>>> node2 = LinkNode(1, LinkNode(2))
>>> replace_last(node2, 3)
>>> print_list (node2)
1 3

>>> replace_last(LinkNode(None, None), 3)
“””
```

```
def replace_last (lst, elem):
    # check if empty:
    if lst.next == None and lst.value == None:
        return lst
```

Replace Last element

Write a function that replaces the last element of the list with a given element:

```
def replace_last (lst, elem):
“””
>>> node2 = LinkNode(1, LinkNode(2))
>>> replace_last(node2, 3)
>>> print_list (node2)
1 3

>>> replace_last(LinkNode(None, None), 3)
“””
```

```
def replace_last (lst, elem):
    # check if empty:
    if lst.next == None and lst.value == None:
        return lst
    if lst.next == None and lst.value != None:
        lst.value = elem
```

Replace Last element

Write a function that replaces the last element of the list with a given element:

```
def replace_last (lst, elem):
“””
>>> node2 = LinkNode(1, LinkNode(2))
>>> replace_last(node2, 3)
>>> print_list (node2)
1 3

>>> replace_last(LinkNode(None, None), 3)
“””
```

```
def replace_last (lst, elem):
    # check if empty:
    if lst.next == None and lst.value == None:
        return lst
    if lst.next == None and lst.value != None:
        lst.value = elem

    while lst.next != None:
        lst = lst.next

    lst.value = elem
```