DATA STRUCTURE

Dr. Rania Baashirah

r.baashirah@ubt.edu.sa

Mobile: +966-559009202

Review

- We introduced the linear data structure.
- Collection of items → order depending.
- Stack
  - Has top and base → follow LIFO
  - Insert(0), Pop(0) → O(n)
- Queue
  - A list can hold any data type → follow FIFO
  - Circular implementation → O(1)
- DEques
  - Queue with both end flexibility
  - Palindrome check example → O(n)

Lists

- Lists has many algorithms such as:
  - Assigning to index position.
  - Joining 2 lists
    - This operation can be done in many different ways in Python; each with different complexity.
  - Append an item
  - Remove an item
  - And so on

Lists

- Assigning to an index position (mylist[0] = 22) → O(1)
- To grow a list: two ways are there:
  - Using concatenation operator (`mylist = mylist + [4, 5]`) → O(k)
  - Using Append method (`mylist.append([4, 5])` → O(1)
- There are 4 methods to generate a list:
  - Concatenate
  - Append
  - Comprehension
  - List range

Lists

```
def test1():
    l = []

    for i in range(1000):
        l = l + [i]

def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))
```

```
from timeit import Timer

t1 = Timer("test1()", "from __main__ import test1")
print("concat", t1.timeit(number=1000), "ms")

t2 = Timer("test2()", "from __main__ import test2")
print("append", t2.timeit(number=1000), "ms")
t3 = Timer("test3()", "from __main__ import test3")

print("comprehension", t3.timeit(number=1000), "ms")

t4 = Timer("test4()", "from __main__ import test4")
print("list range", t4.timeit(number=1000), "ms")
```

```
concat 6.54352807999 milliseconds
append 0.306292057037 milliseconds
comprehension 0.147661924362 milliseconds
list range 0.0655000209808 milliseconds
```

Lists

- Notice that pop() and pop(i) are different.
- Also append and insert are different.

Operation	Big-O Efficiency
indexx[]	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

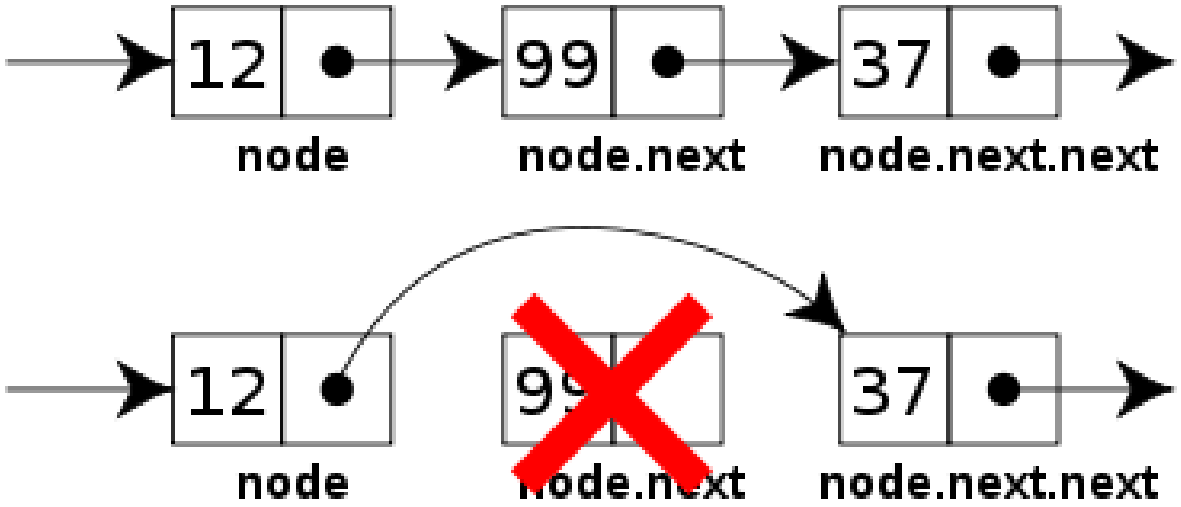
Table 2.2: Big-O Efficiency of Python List Operators

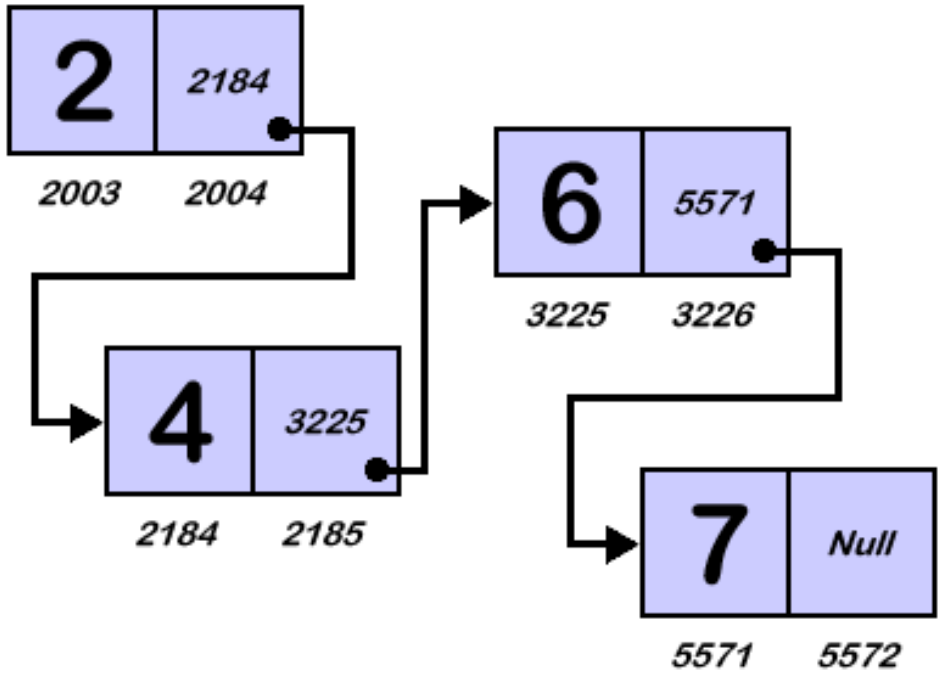
List ADT

- **add(x)** adds x to the list.
- **remove()** removes the item from the list.
- **search(x)** return True if x is found.
- **is_empty()** return True if empty.
- **size()** returns the number of items.
- **append(x)** adds x to end of the list.
- **index(x)** returns the position of item.
- **insert(p,x)** adds x to the list at position p
- **pop()** removes and returns the last item.
- **pop(p)** removes and returns the item at position p.

Linked List

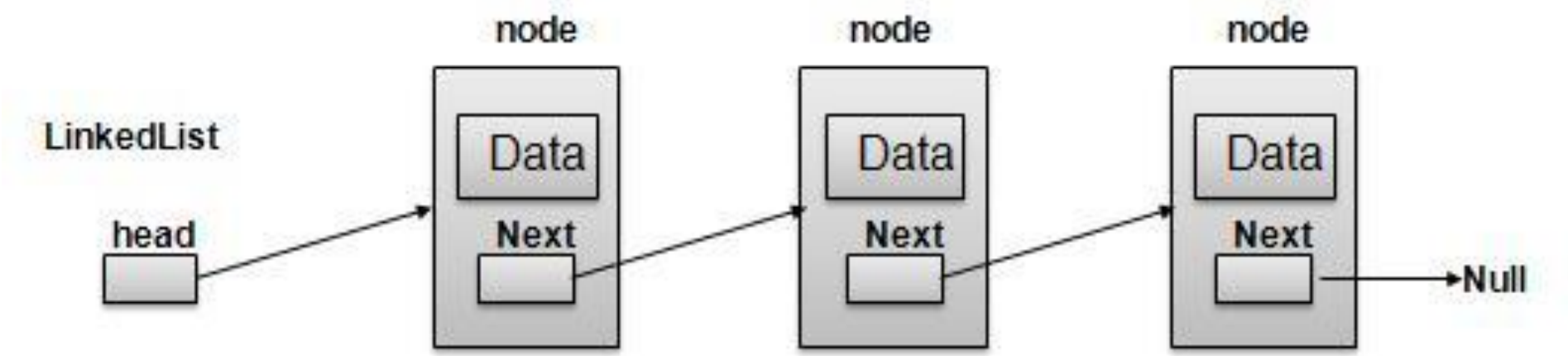
- We maintain the relative position of each item → by expressing the location where the item is stored
- We follow the link from location to another → until we reach the item





Linked List

- Linked List is basically nodes connected together → basic building block is “Node”
- Each node is composed of: List item “Data field” , “Reference” to the next node
- First element in the list is → Head
- Last element in the list is pointing to → null



Advantage of Linked List

- Flexible space, since no allocating space in advance. We only run out of space when the whole memory is actually full.
- Insertion and deletion.
  - No shifting is required.
- More efficient for moving large records (leave data in same place in memory, just change some pointers).

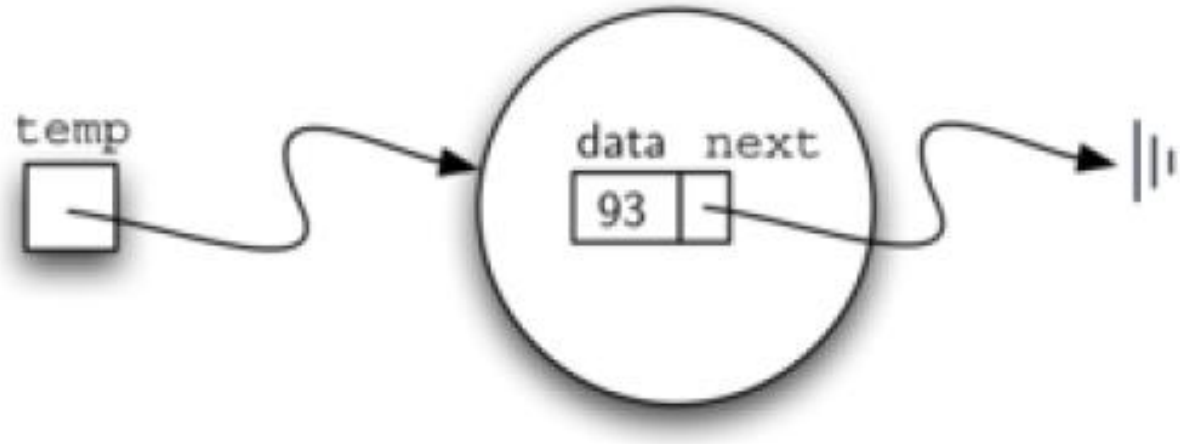
Disadvantage of Linked List

- Wasted space: We store both pointers and data.
- Slow access to i^{th} item, since we must start from the beginning and follow pointers until we get there.
- In the worst case, if there are n items in a list and we want the last one → we have to do n lookups.
- So retrieving an element from its position in the list is O(n).

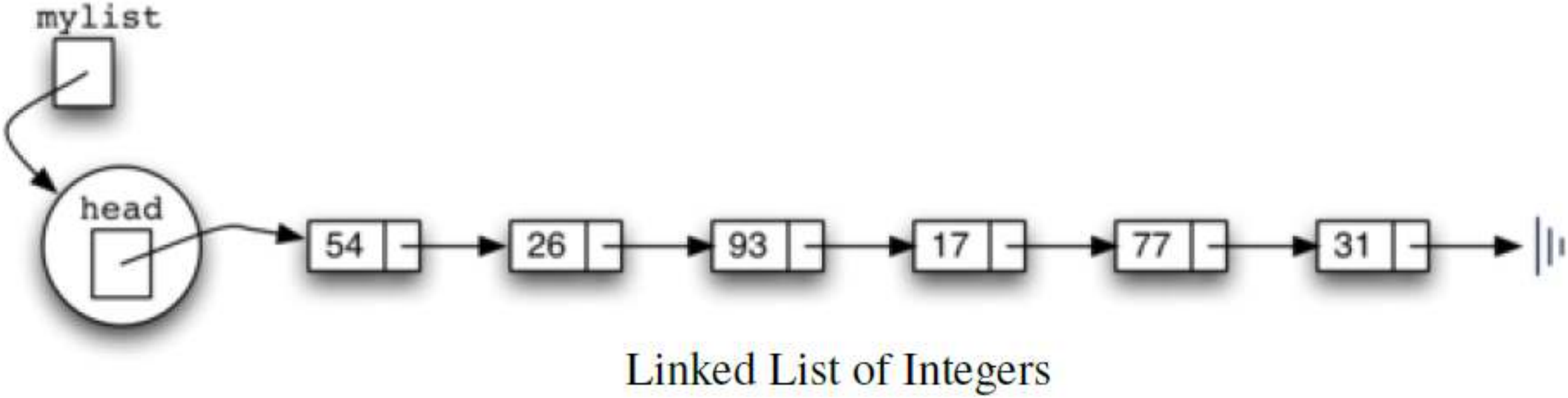
LinkedList Implementation in python

- First we should have class of nodes

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


# ------- Test --------
if __name__ == "__main__":
    temp = Node(93)
    print(temp.data)
```



Unordered List class

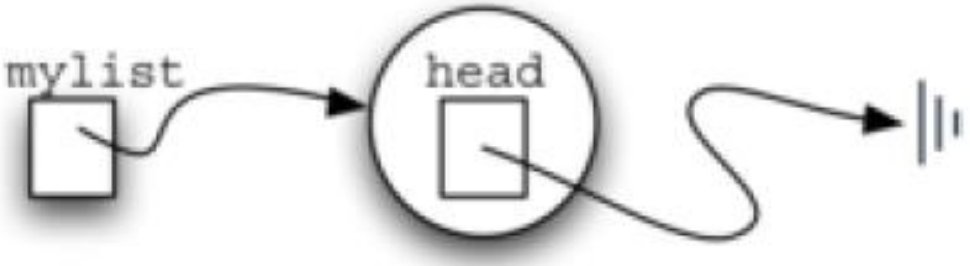
- Unordered list will be built from a collection of nodes
- The class of LinkedList contains only one node → head
- Head is linked to the next node and each node is linked to the next one.
- 

Linked List of Integers

Implementation of Linked List

- Initially when we construct a list, there are no items.
- The assignment statement.

```
class LinkedList:
    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head == None
```

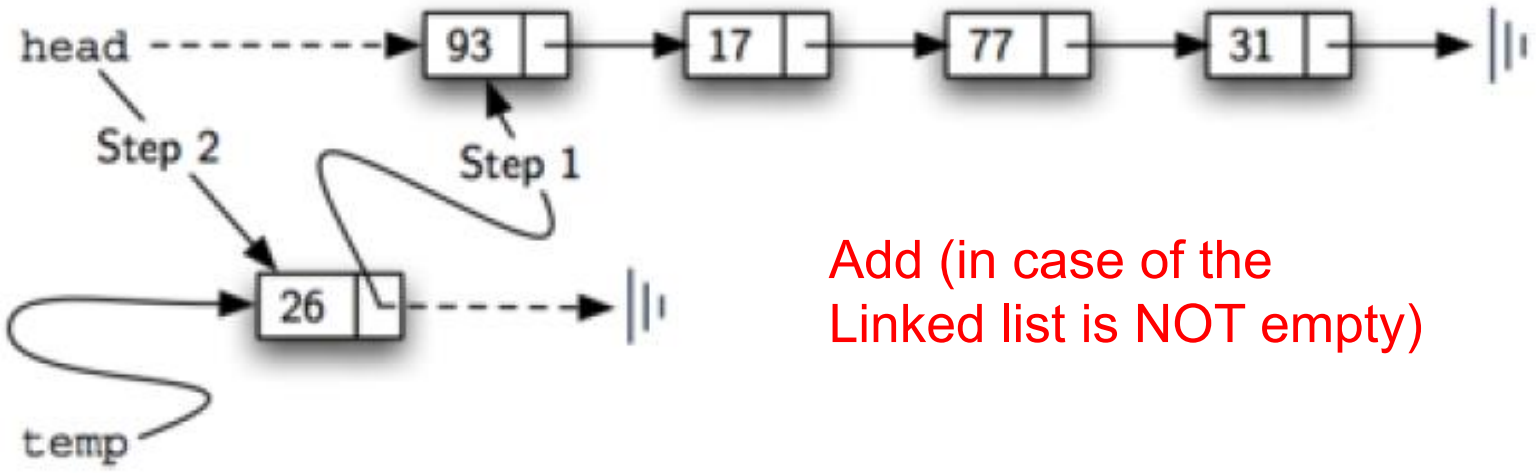


An Empty List

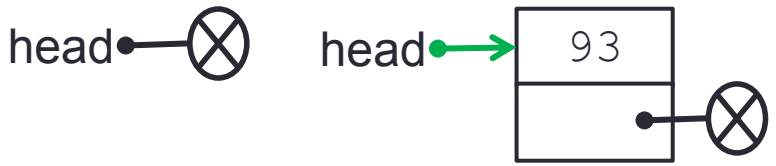
Linked List – add(x)

- This operation adds any new element at the ***beginning*** of the Linked list.

```
class LinkedList:
    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head == None

    def add(self, item):
        temp = Node(item)
        temp.next = self.head
        self.head = temp
```



Add (in case of the Linked list is NOT empty)

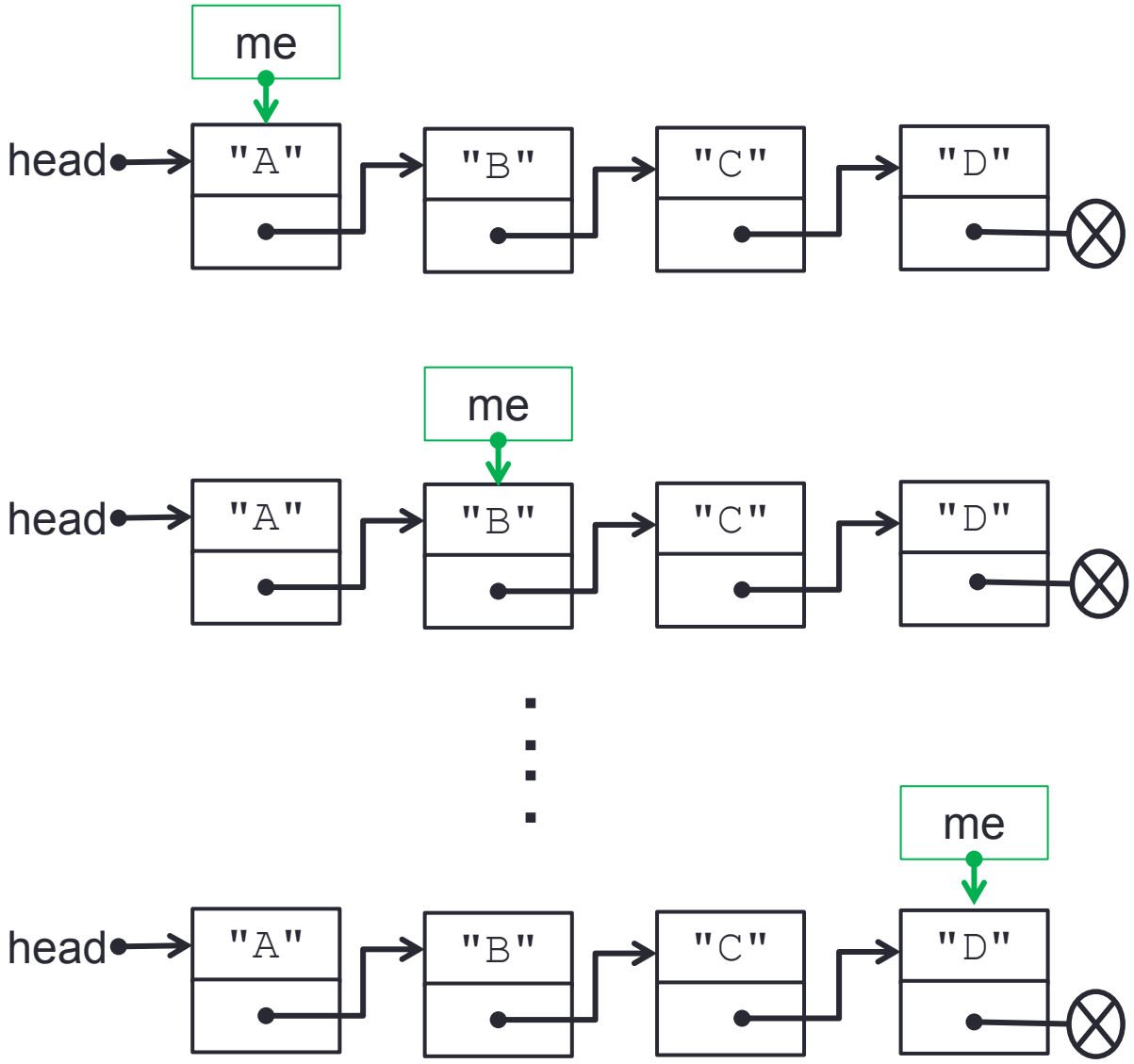


Add (in case of the Linked list is empty, i.e head is null)

Linked List – size( )

- This operation moves through all elements from the head to tail.
- We achieve this by passing a **temporary node** (me) starting from head till we find null

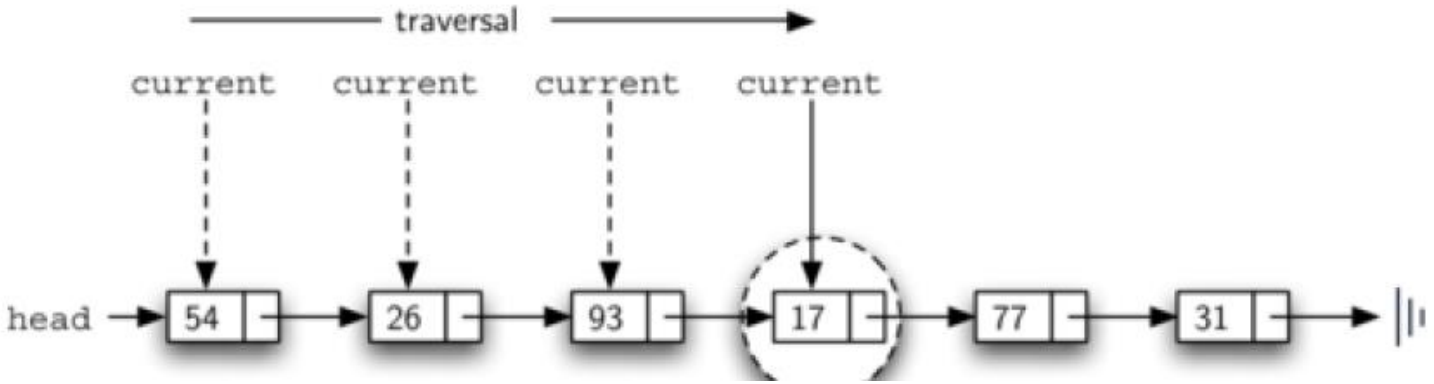
```
def size(self):
    me = self.head
    count = 0
    while me is not None:
        count += 1
        me = me.next

    return count
```



Linked List – search(item)

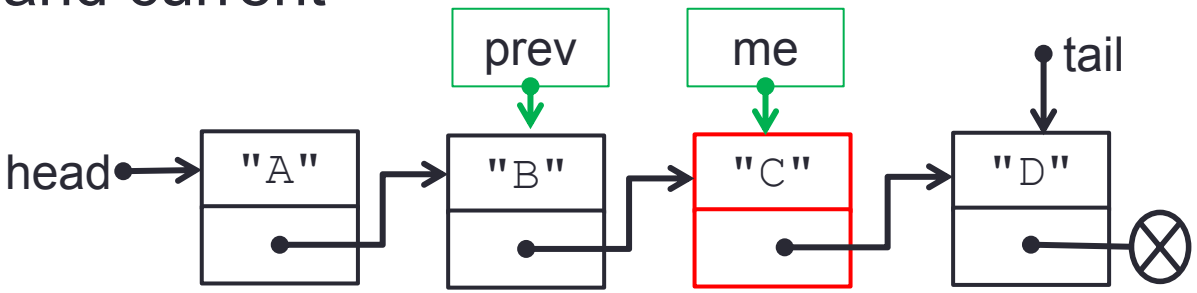
- Like traversal, but the temporary object moves up to k_{th} element or finds null.
- We check every node starting from the head → moving next

```
def search(self, item):
    me = self.head
    found = False
    while me != None and not found:
        if me.data == item:
            found = True
        else:
            me = me.next
    return found
```

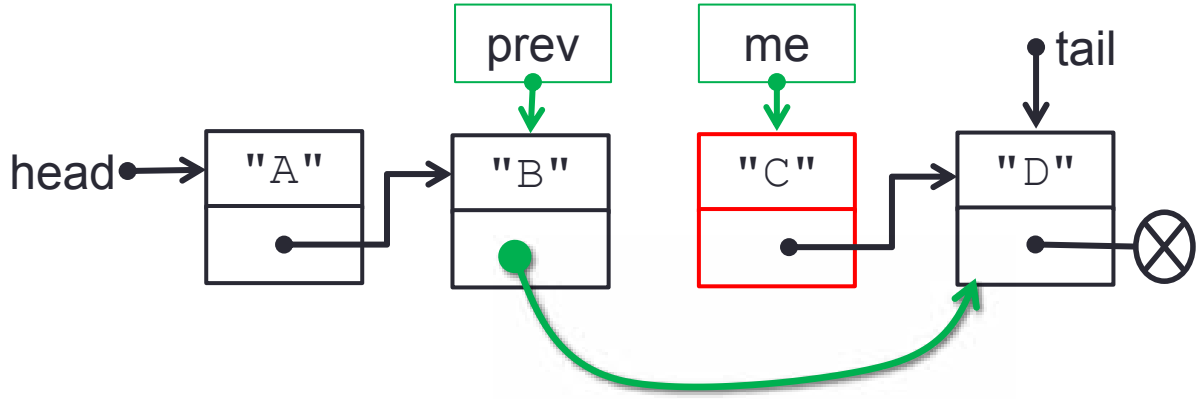


Linked List – Remove Node

- We have current node , and previous node linked to it
- We search for the node we want to remove.
- We break the link between the previous and current
- We connect the head with me.next



- Search = O(n)
- Deletion = O(1)



Linked List – Remove Node

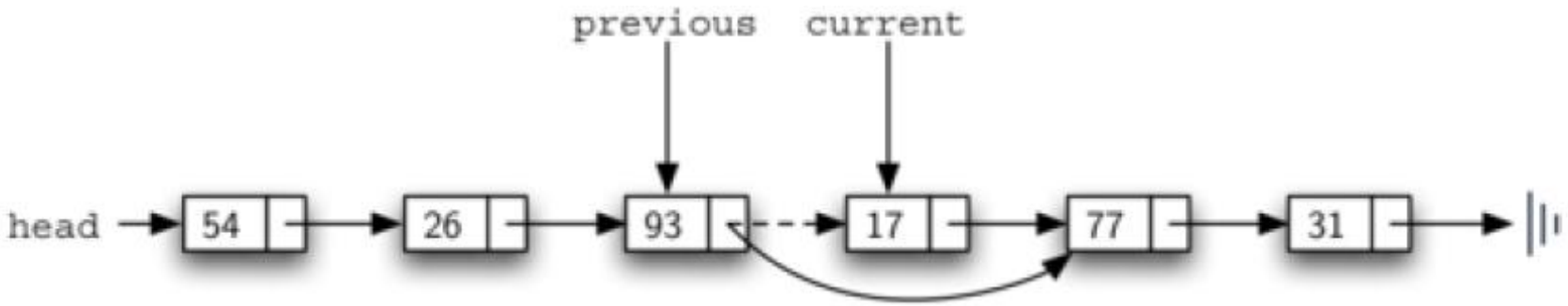


Figure 3.30: Removing an Item from the middle of the list

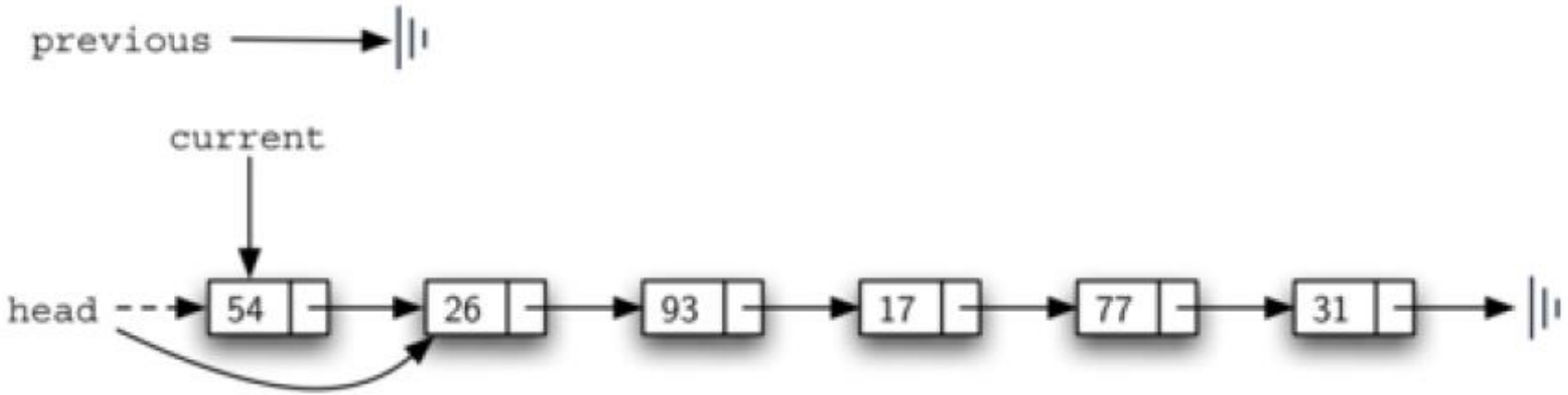


Figure 3.31: Removing the first node from the list

Ordered List

- When we have ordered list → sorted elements ascending or descending
- The operations of the ordered LinkedList are the same of the unordered
- We implement it the same way as head pointing to none
- The difference: search – add

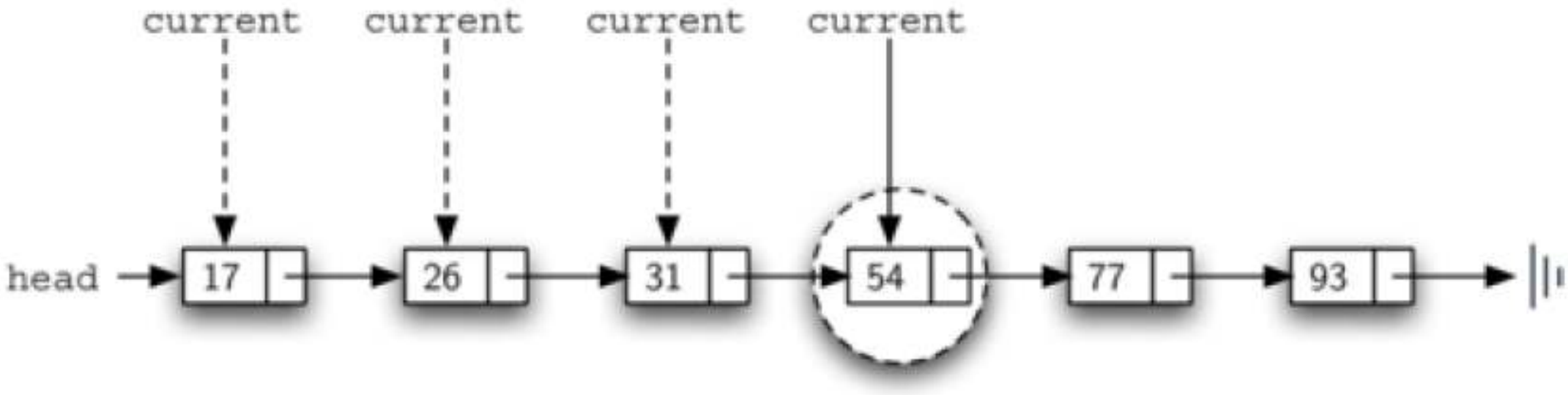


Figure 3.33: Searching an Ordered Linked List

Ordered List: Search(item)

- We still traverse the nodes until we reach the item, or run out of nodes
- We can stop the search when we reach a larger/lower item

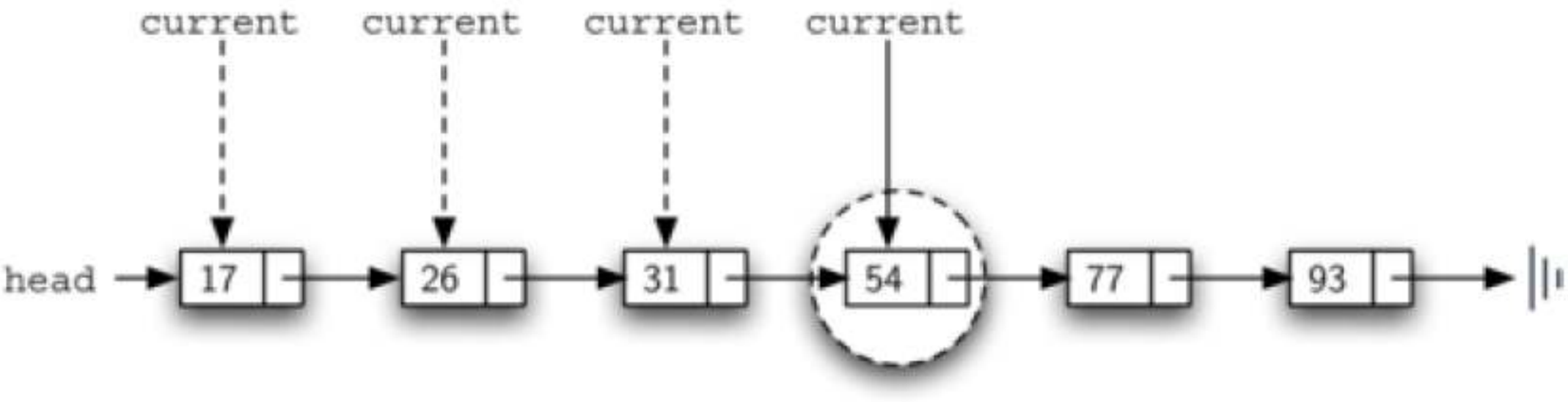


Figure 3.33: Searching an Ordered Linked List

Ordered List: Add(x)

- In unordered list → we add at the head
- In ordered list → we discover a specific place
- We traverse the node → we find a position
- We use external references → previous, current
- We assign the current of our item as “previous of list”
- We assign the previous of our item as “current of list”

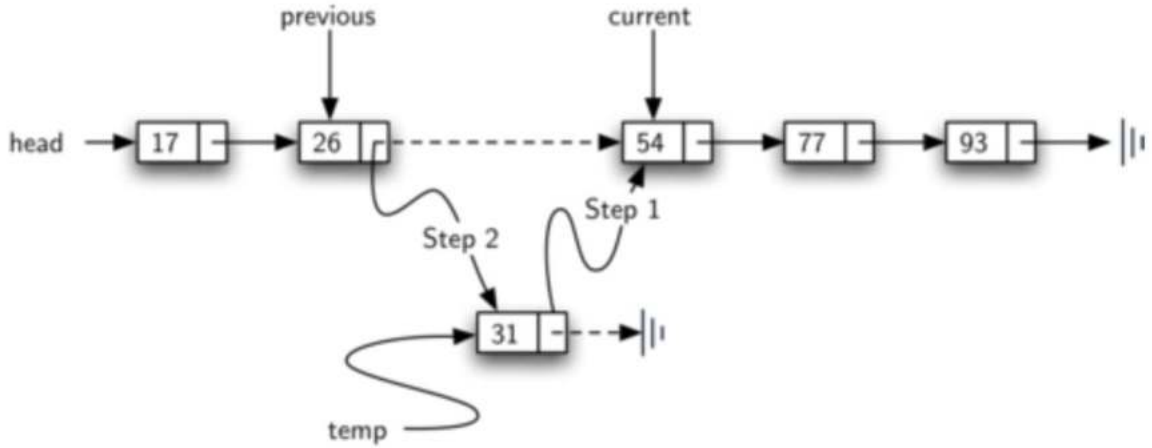


Figure 3.34: Adding an Item to an Ordered Linked List

Linear Data Structure - Summary

- We have learned 4 types of linear data structures
  - Arrays
  - Linked Lists
  - Stacks
  - Queues
- On each data structure we have learned how to apply the basic operation on them such as: Add() , Remove() , Find(), and Traverse().
- There are other advanced linear data structure such as:
  - Doubly Linked List
  - Double-Ended Queue
  - Priority Queue

Time and Space Complexity for most common linear data structures

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)

http://bigocheatsheet.com/