UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

1

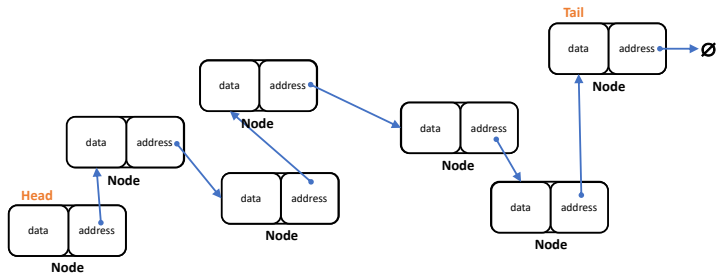
Announcements

- Assignment 1 extended deadline: 10/07
- Lab 05 will be due on time
- Career Fair Briefing today

2

Quick Recap

- Linear Data Structures
  - Stacks, Queues → using List [1, ,2, 3, 4, 5, …, n] → causes O(n)
- LinkedList



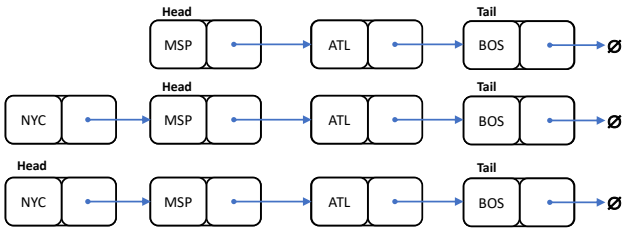
- Node

```
class _Node:
    """ Node class to create
    individual linked list  nodes """
    def __init__(self, element, next):
        self._element = element
        self._next = next
```


3

Quick Recap

- **Inserting element at the head**



self._head = _Node("NYC", self._head)

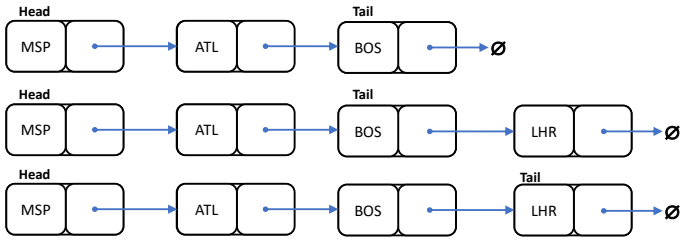
```
class _Node:
    """ Node class to create
    individual linked list  nodes """
    def __init__(self, element, next):
        self._element = element
        self._next = next
```

```
class LinkedList():
        def__init__(self):
                self._head = None
                self._tail = None
                self._size = None
```


4

Quick Recap

- **Inserting element at the tail**



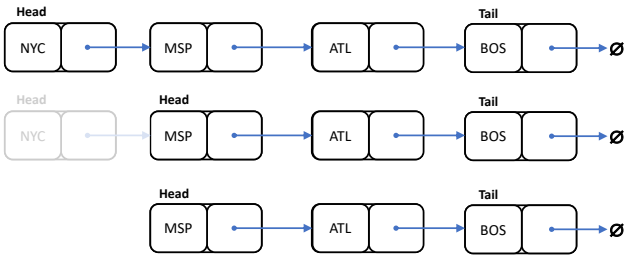
- newest = _Node("LHR", None)
- self._tail._next = newest
- self._tail = newest

```
class LinkedList():
        def__init__(self):
                self._head = None
                self._tail = None
                self._size = None
```


5

Quick Recap

- **Deleting element from the head**



- self._head = self._head._next

```
class LinkedList():
        def__init__(self):
                self._head = None
                self._tail = None
                self._size = None
```


6

Quick Recap

- **Deleting element from the tail**

- We cannot easily delete the last node of a singly linked list
  - Though, we can hold a reference to the Tail node
  - There is no way to determine the node preceding to the Tail node
- List traversal may help, but takes O(n) time

Linked Lists ADT | Doubly Linked List

- We were able to solve the issue of inserting/removing elements from the head of the list
- Also, we could add an element to the tail with O(1) cost
- Removing an element from the tail is still an issue because nodes:
  - only keep the reference of next node
  - do not have any way to find out its preceding (previous) node

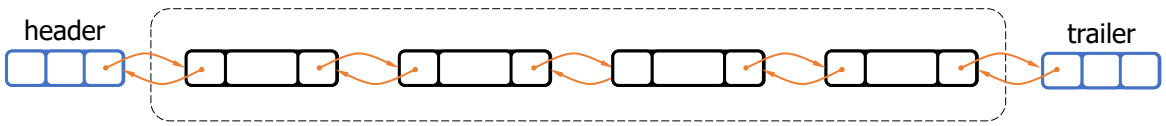
Doubly Linked List is developed in which nodes contain references of both `next` and `previous` nodes.

It provides O(1) time operations including insertions and deletions at arbitrary position within the list

Linked Lists ADT — Doubly Linked List

- next → reference to next node
- prev → reference to previous node

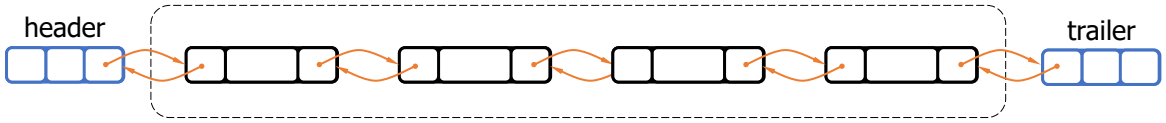
- To simplify operations further, we can add two dummy nodes (called sentinels):
  - "Header" → Head of the list
  - "Trailer" → Tail of the list



Linked Lists ADT — Doubly Linked List

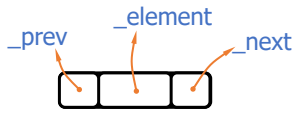
Advantages of using "sentinels"

- The head and tail nodes never change, only the elements/nodes between them
- All insert operations are uniform → new nodes are always added between two existing nodes
- All delete operations are uniform → node to be deleted is guaranteed to be between two nodes



Linked Lists ADT — Doubly Linked List

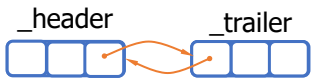
1. Create a Double Linked Node
   - private Node with additional feature of _prev node.




```
class _Node:
    """ nonpublic class for
    storing a doubly linked node. """

    def __init__(self, prev, element, next):
        self._prev = prev
        self._element = element
        self._next = next
```

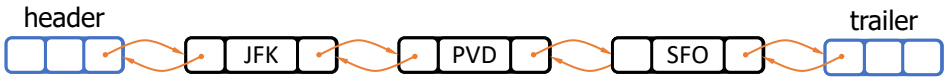
2. Create a base class for doubly linked list to encapsulate the required information



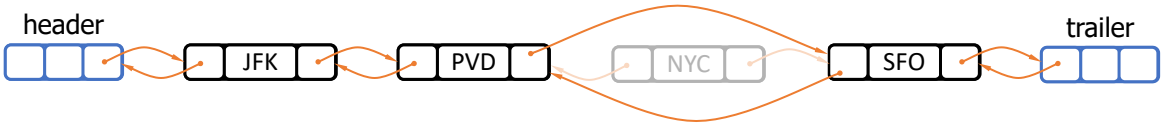

```
class _DoublyLinkedBase:
    """ Base class providing a doubly
    linked list representation. """
    def __init__(self):
        self._header = _Node(None, None, None)
        self._trailer = _Node(None, None, None)
        self._header._next = self._trailer
        self._trailer._prev = self._header
        self_size = 0
```


Linked Lists ADT — Doubly Linked List

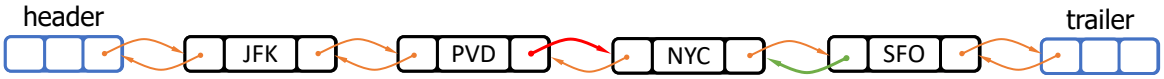
Insertion of a node



1. Create a new node with reference of `predecessor` and `successor` nodes



2. Then update _next and _prev pointers of predecessor and successor nodes to point to the new node



Linked Lists ADT

Doubly Linked List

Insertion of a node

1. Create a new node with reference of predecessor and `successor` nodes

2. Then update _next and _prev pointers of predecessor and successor nodes to point to the new node

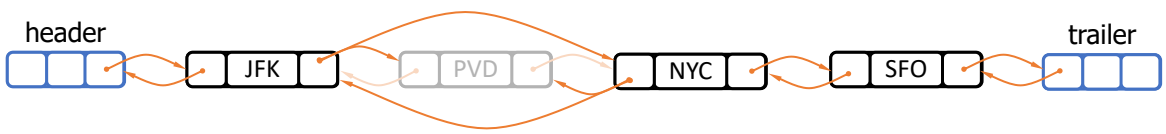
```
def _insert_between(self, predecessor, element, successor):
    newest = _Node(predecessor, element, successor)
    predecessor._next = newest
    successor._prev = newest
    self._size += 1
    return newest
```


Linked Lists ADT

Doubly Linked List

Deletion of a node

1. Link the neighbors of the node (to be deleted) with each other



2. We can optionally set the attributes of deleted node to be "None"

Linked Lists ADT

Doubly Linked List

Deletion of a node

1. Link the neighbors of the node (to be deleted) with each other.
2. We can optionally set the attributed of deleted node to be "None".

```
def _delete_node(self, node):
    """ Delete nonsentinal node from the list and return its element. """
    predecessor = node._prev
    successor = node._next

    predecessor._next = successor
    successor._prev = predecessor
    self._size -= 1

    element = node._element #recording deleted element
    node._prev = node._next = node._element = None
    return element          #return deleted element
```


Module 5

Recursion and Dynamic Programming

Recursion

Recursion is a method of solving a problem with a help of a function, when the function "recursively" call itself.

Example: Adding K integers.

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
print(sum(5))
```

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

k=0
k=1
k=2
k=3
k=4
k=5

17

Recursion

Recursion is a method of solving a problem with the help of a function, when the function "recursively" call itself.

Example: Adding K integers.

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
print(   15   )
```

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

k=0
0
k=1
1
k=2
3
k=3
6
k=4
10
k=5

18

Recursion | Function Call Stack

A function can be made to recursively call itself forever.

- In Python, infinite recursion can be quickly discovered using `RecursionError`
  - → Limit of recursion in Python depends on distribution
  - → typical default value is 1000

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

⋮

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

FunctionCall Stack