# 15CSE201 : Data Structures and Algorithms

## Linked Lists

By Ritwik M

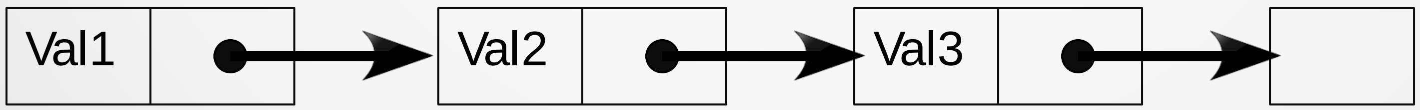
Based on the reference materials by Prof. Goodrich, OCW METU and Dr. Vidhya Balasubramanian

# Why Another Data Structure?

- Currently we have seen array based implementations
- Limitations:
  - Limited size
  - Can increase but it is quite tedious
- Solution:
  - Dynamically allocate and de-allocate memory as and when data is added or removed.

# Dynamically Allocating Elements

- Allocate elements one at a time
  - Each element keeps track of next element
- Results in a linked list of elements
  - Elements track next element with a pointer
- elements can easily be inserted or removed without reallocation or reorganization of the entire structure

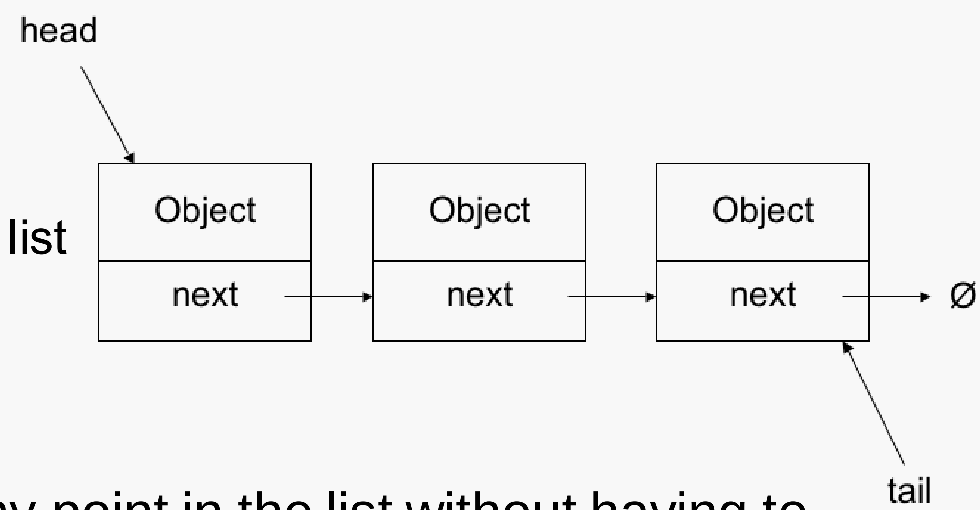


# Linked Lists

- Developed in 1955-56 by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation as the primary data structure for their Information Processing Language (IPL)
- Must have the following
  - Way to indicate end of list
  - NULL pointer
  - Indication for the front of the list
  - Head Node
  - Pointer to next element

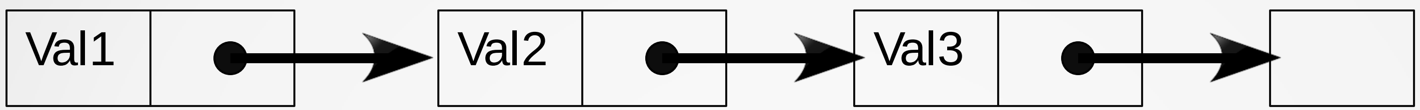
# Linked Lists: Basic Concepts

- Each record of linked list is an element or a node
- Each node contains
  - Data member which holds the value
  - Pointer “next” to the next node in the list
  - Head of a list is the first node
  - Tail is the last node
- Allows for insertion and deletion at any point in the list without having to change the structure
- Does not allow for easy access of elements (must traverse to find an element)

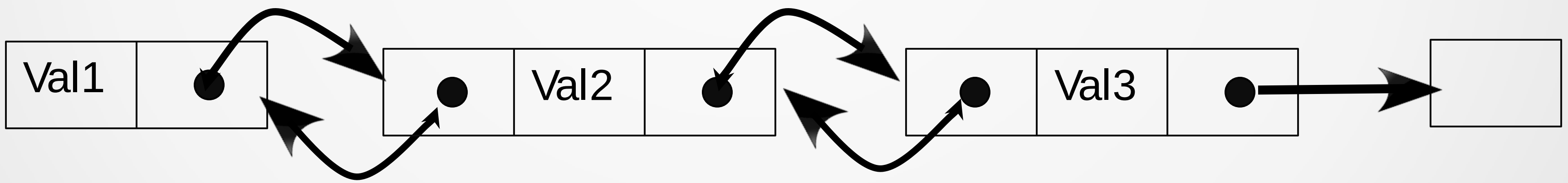


# Linked Lists: Types

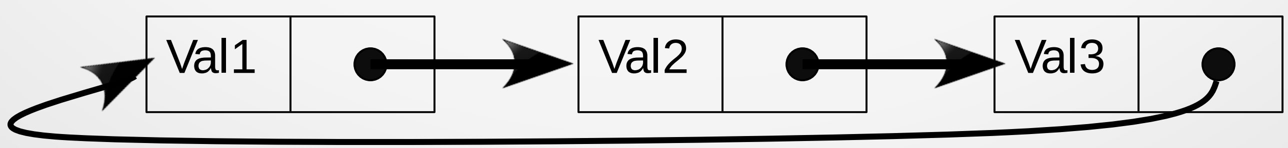
- Singly Linked List




- Doubly Linked List

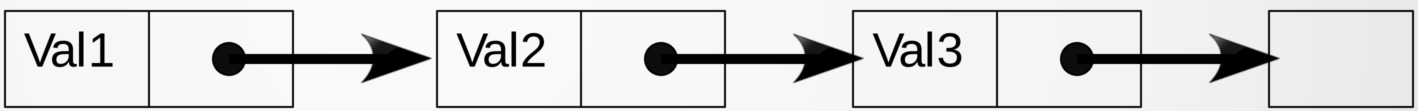



- Circular Linked List



# Singly Linked Lists

- Keeps elements in order
  - Uses a chain of next pointers
  - Does not have fixed size, proportional to number of elements
- Node
  - Element value
  - Pointer to next node




- Head Pointer
  - A pointer to the header is maintained by the class

# Basic Linked List Definition

- class Node():

  element // The data being stored in the node

  next // A reference to the next node, null for last node, of
  // the type Node

- class List():

  self.head = None

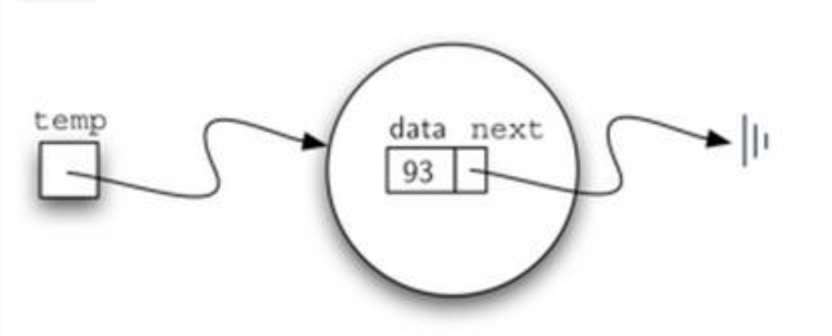
  // points to first node of list; null for empty list

  // this is also known as the head

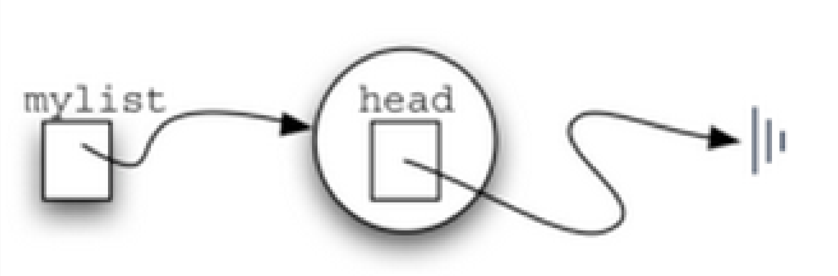
  >>> mylist = UnorderedList()

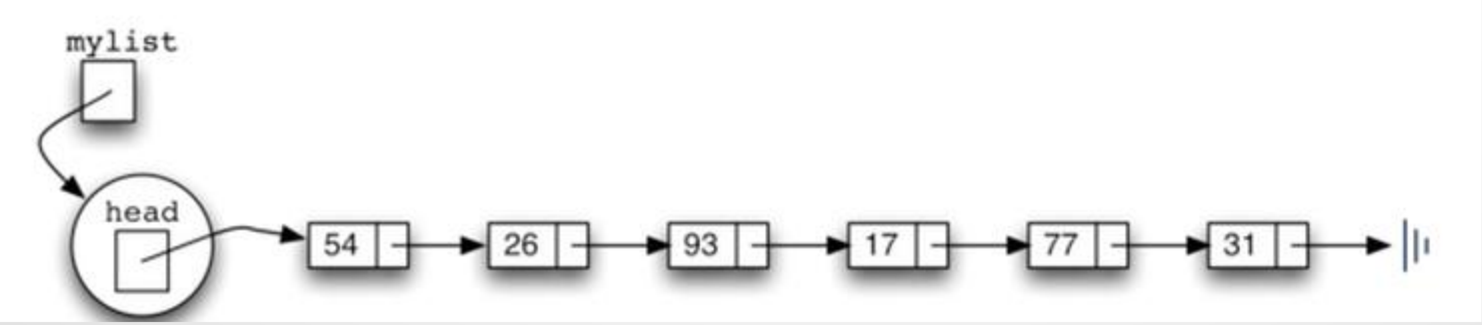
# Implementation Details

- The Node Class




- The List Class
  - O(n)



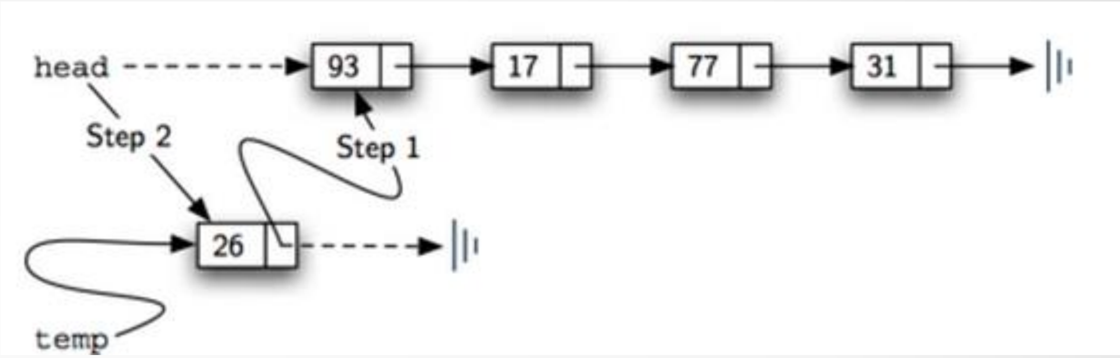



# Insertion and Deletion

- Insertion can be at head or tail
  - Create new node, and make new node point to head, and make it the new head
  - If using tail pointer, point next of tail to new node, and next of new node to null
- Deletion
  - Requires the reorganization of next pointers

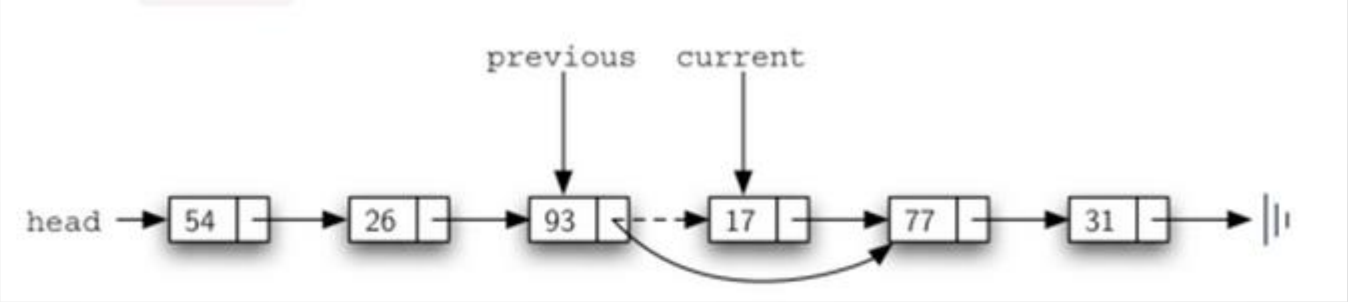
# The Code

- Insertion at head
  - def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp




- Deletion
  Search through the list to find the element (marked as current)

  previous.setNext(current.getNext())



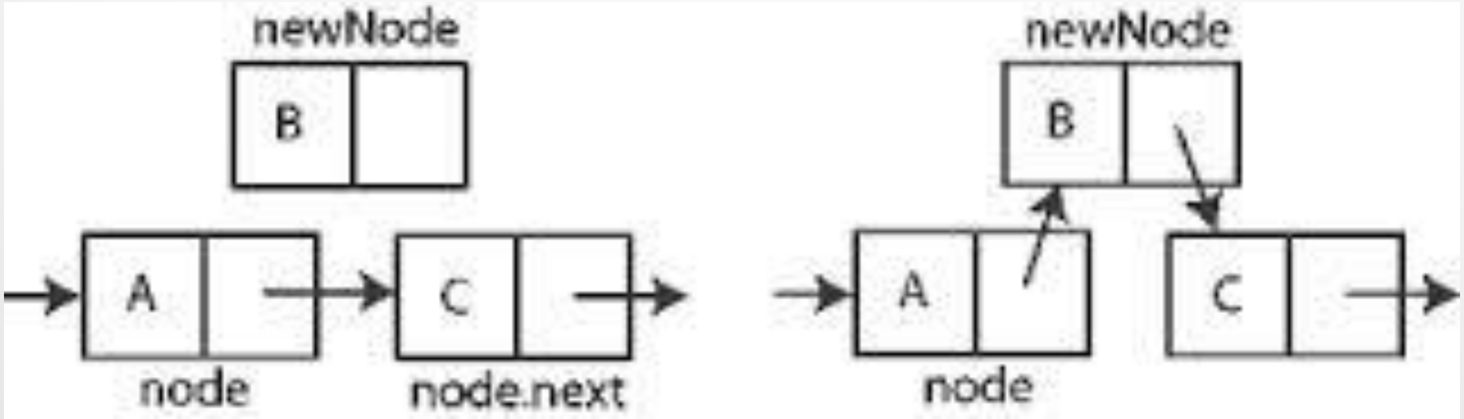
# List ADT: Functions

Algorithm insertAfter(Node node, Node newNode)

    // insert newNode after node

        newNode.next ← node.next

        node.next ← newNode



# List ADT Functions:

- Algorithm insertFirst(List list, Node newNode)

  // insert node before current first node

  newNode.next := list.firstNode

- Algorithm insertLast(List list, Node newNode)

  // insert node after the current tail node

  tail.next ← newNode

  newNode.next ← NULL

# List ADT: Delete Functions

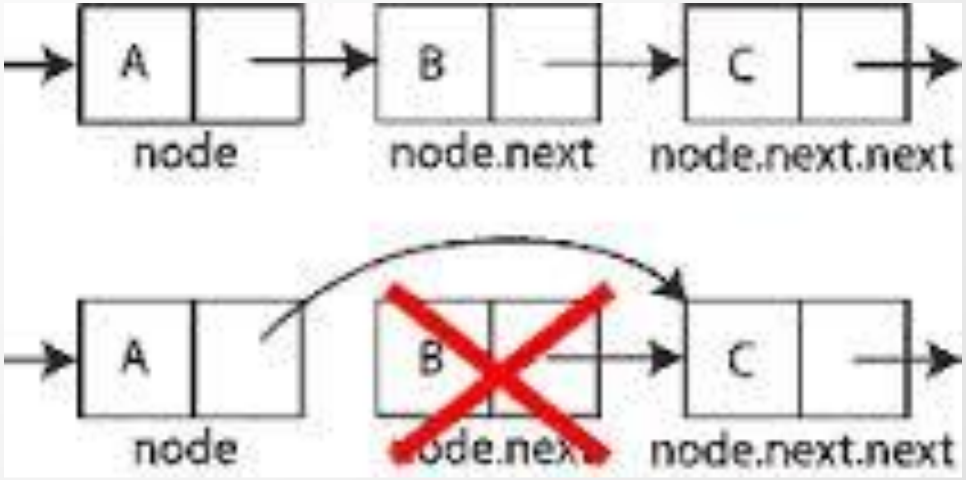
- Algorithm removeAfter(Node node)

    // remove node past this one

    obsoleteNode ← node.next

    node.next ← node.next.next

    destroy obsoleteNode



# List Traversal

Algorithm Traverse()

Node ← list.firstNode

while node not null

do something with node.element

node ← node.next

# Linked List ADT: Python

```
Class Node():
    def __init__(self, value, next):
Class LinkedList():
    def __init__(self):
    self.length = 0
    self.head = None
    def insertFirst(self, e)
    def insertLast(self, e)
    def insertAfter(self, p, e) //insert node with value e after node p
    def removeAfter(self, p) // where p is the node after which it must be deleted
```

# Other list functions

- first() : return the first node of the list, error if S is empty
- last(): return last node of the list, error if S is empty
- isFirst(p): returns true if p is the first or head node
- isLast(p): returns true if p is the last node or tail
- before(p): returns the node preceding the node at position p
- getNode(i): return the node at position i
- after(p): returns the node following the node at position p
- size() and isEmpty() are the usual functions

# List: Update Functions

- replaceElement(p,e): Replace element at node at p with element e
- swapElements(p,q): Swap the elements stored at nodes in positions p and q
- insertBefore(p,e) Insert a new element e into the list S before node at p

# Complexity Analysis

- Time Complexity
  - size – O(n)
  - isEmpty – O(1)
  - first(), isFirst(), isLast()– O(1)
  - before(p), after(p) - O(1)
- Space Complexity
  - O(n)

# Exercises / Quiz

1. Give an algorithm for finding the penultimate node in a singly linked list where the last element is indicated by a null next pointer
2. Give an algorithm for concatenating two singly linked lists L and M, with header nodes, into a single list L' where
   - L' contains all nodes of L in their original order followed by all nodes of M (in original order)
   - What is the running time of your algorithm if n is the number of nodes in L, and m is the number of nodes in M?
   - What if instead of concatenating, you merge the lists L and M such that that nodes from M and L are arranged in <u>alternate order</u>? Mention the time complexity of your algorithm.

# Stack: Linked List Based Implementation

- Top element is stored as the head (first node) of the linked list
- Insertion and deletion always at the front
- The stack class has the following variables

  Node topnode //top is the head node
- Initialized to NULL

  sz //variable to keep track of the size of the list
- initialized to 0

# Stack ADT Functions

- Algorithm size()

  return sz

- Algorithm isEmpty()

  return (sz == 0)

- Algorithm top()

  if isEmpty() then

  throw a StackEmptyException

  return topnode.element

# Stack ADT Functions

- Algorithm push(o)

  if size() = N then

  throw a StackFullException

  newNode ← new Node(o, topnode)

  topnode ← newNode

  sz++

- Algorithm pop()

  if isEmpty() then

  throw a StackEmptyException

  Node oldNode ← topnode

  topnode ← topnode.next

  sz--

  o ← oldNode.element

  delete oldNode

  return o

# Queue: Linked List Based Implementation

- Can be done similarly
- Here insertion is done at the tail
- Deletion is at the head

# Exercises

- Design and implement an SLList method, secondLast(), that returns the second-last element of an SLList. Do this without using the member variable, n, that keeps track of the size of the list.
- Describe and implement the following List operations on an SLList
  - get(i) // get the node at position i
  - set(i,x) // set the value of node at ith position to x
  - add(i,x) // add a node with value x with position i
  - remove(i). //remove node at position i
  - Each of these operations should run in O(1 + i) time.

# Doubly Linked List vs Singly Linked List

- Doubly linked lists occupy more space per node
  - Basic operations are more expensive
- Allow access in both directions
  - Advantage: All you need is the address of any one node to traverse the entire list
  - Singly linked lists can move in only one direction
- Circular Lists can also traverse the entire list.

# Class Exercises

- Develop an algorithm to print the middle element of a given singly linked list L in linear time
- Given only a pointer to a node to be deleted in a singly linked list, develop an algorithm to you delete it.
- Given a singly linked list of characters, write an algorithm for a function that returns true if the given list is palindrome, else false
- Given a linked list which is sorted, write an algorithm to insert a new element that retains the sorted nature of the list.

# End of Lecture 7

Ritwik M