19CSE201 :Advanced Programming

Lecture 25
ADTs in Python

By
Ritwik M
Assistant Professor(SrGr)
Dept. Of Computer Science & Engg

Based on Multiple Sources on the Internet

A Quick Recap

- STLs in C++
  - Stacks
  - Queues
  - Linked List
  - Hash Table

Definition

- Abstract Data Type (ADT)
  - Is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics
  - Is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects
- ADT defines
  - the set of operations supported by a data structure and
  - the semantics, or meaning, of those operations
- The definition can vary based on the type of programming language
  - Imperative Definition
  - Object Oriented Definition

Imperative Definition

- An abstract data structure is considered an entity that may be in different states at different times
  - Operations may change the state of the ADT
  - Order of operations important
- Implementation
  - Use concept of abstract variables (simplest nontrivial ADT), that admits two operations
  - store(V,x), x is any value
  - fetch (V)
    - Returns value that is most recently stored.

Imperative ADT

- Instance Creation
  - One may need to create a new instance of a stack
  - create() operation is used for this purpose
- Example: Abstract Stack defines state of stack S
  - push(S,x)
  - pop(S)
  - create() - creates a new stack different from other stacks

Object Oriented Definition

- Defines the ADT as a class containing
  - Variables
  - Functions
- E.g Stack ADT in Object Oriented Paradigm

Advantages of Abstract Data Typing

- Encapsulation
  - Guarantees that the ADT has certain properties and abilities
  - User does not need to know the implementation details to use the ADT
- Localization of Change
  - Change to ADT implementation does not impact the code that uses the ADT
  - The implementation must still comply with the ADT definition, hence the interface will not change

Advantages of Abstract Data Typing Cont.

- Flexibility
  - Can use different implementations of an ADT in a code
    - All have same properties and abilities
  - The efficiency of different implementations maybe different
- Can use the most suitable implementation

Implementation Strategy

- Usually implemented as modules
- The module's interface declares procedures that correspond to the ADT operations
- Can be multiple implementations of a single ADT

Stack ADT: Main Operations

- push (o)
  - Push object o onto the stack
  - Input: object; Output: None
- pop ( )
  - Remove the last element that was pushed
  - Input: none; Output: object
- size ( )
  - Returns the number of objects in the stack
- isEmpty ( )
  - Returns a Boolean indicating if a stack is empty
- top ( )
  - Return top object of the stack without removing it
  - Input: None; Output: Object

Stack Exceptions

- Some operations may cause an error causing an exceptions
- Exceptions in the Stack ADT
  - StackEmptyException
    - pop () and top () cannot be performed if the stack is empty
  - StackFullException
    - Occurs when the stack has a maximum size limit
    - push (o) cannot occur when the stack is full

Stack ADT

- class MyStack():
  - def push(self, value): #pushes the value into the stack
  - def pop(self): # returns top element of stack if not empty
else throws exception
  - def top(self): # returns top element without removing it
if the stack is not empty, else throws
exception
  - def size(self):#returns the number of elements currently
#in stack
  - def isEmpty(self): #returns True if stack is empty

Stack ADT Functions

- Algorithm push(o)
  - if size() = N then
    - throw a StackFullException
  - t ← t+1
  - S[t]←o
- Algorithm pop()
  - if isEmpty() then
    - throw a StackEmptyException
  - o ← S[t]
  - t ← t-1
  - return o
- Algorithm size()
  - return t+1
- Algorithm isEmpty()
  - return (t<0)
- Algorithm top()
  - if isEmpty() then
    - throw a StackEmptyException
  - return S[t]

Queue ADT: Main Operations

- `enqueue(o)`
  - Inserts an object o at the end of the queue
  - Input: object; Output: None
- `dequeue()`
  - Removes and returns the first element in the queue
    - Input: none; Output: object
    - Error occurs if queue is empty
- `size()`
  - Returns the number of objects in the queue
- `isEmpty()`
  - Returns a Boolean indicating if a queue is empty
- `first()`
  - Return first element of the queue without removing it. Error if queue is empty
  - Input: None; Output: Object

Queue Exceptions

- Some operations may cause an error causing an exceptions
- Exceptions in the Queue ADT
  - QueueEmptyException
    - dequeue() and front() cannot be performed if the queue is empty
  - QueueFullException
    - Occurs when the queue has a maximum size limit
    - enqueue(o) cannot occur when the queue is full

Queue ADT

- class MyQueue():
  - def enqueue(self, value): #pushes the value into the front of the queue
  - def dequeue(self): # returns and removes element at the front of the queue if not empty else throws exception
  - def front(self): # returns front element without removing it if the queue is not empty, else throws exception
  - def size(self):#returns the number of elements currently #in queue
  - def isEmpty(self): #returns True if queue is empty

List based implementation of a Queue

- A Queue may be implemented by using a simple array (list)
  - An N-element array
    - Queue is limited by the size of the array
  - Two variable to keep track of front and rear
    - Integer f denotes the index of the front element
    - Integer r denotes the position immediately past the rear element
- Strategy
  - Elements are added left to right

Linked Lists: Basic Concepts

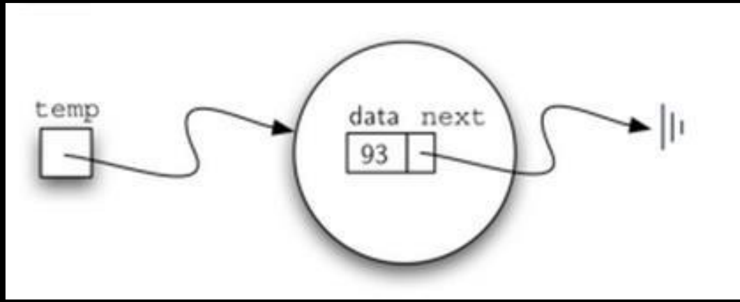
- Each record of linked list is an element or a node
- Each node contains
  - Data member which holds the value
  - Pointer "next" to the next node in the list
  - Head of a list is the first node
  - Tail is the last node
- Allows for insertion and deletion at any point in the list without having to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

Singly Linked Lists

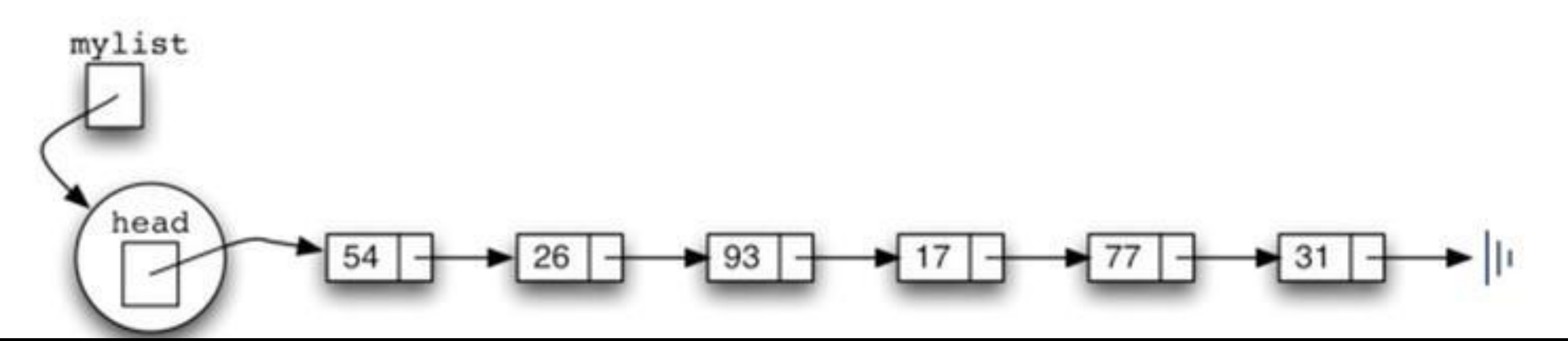
- Keeps elements in order
  - Uses a chain of next pointers
  - Does not have fixed size, proportional to number of elements
- Node
  - Element value Val1
  - Pointer to next node
- Head Pointer
  - A pointer to the header is maintained by the class

Implementation Details

- The Node Class



- The List Class



Basic Linked List Definition

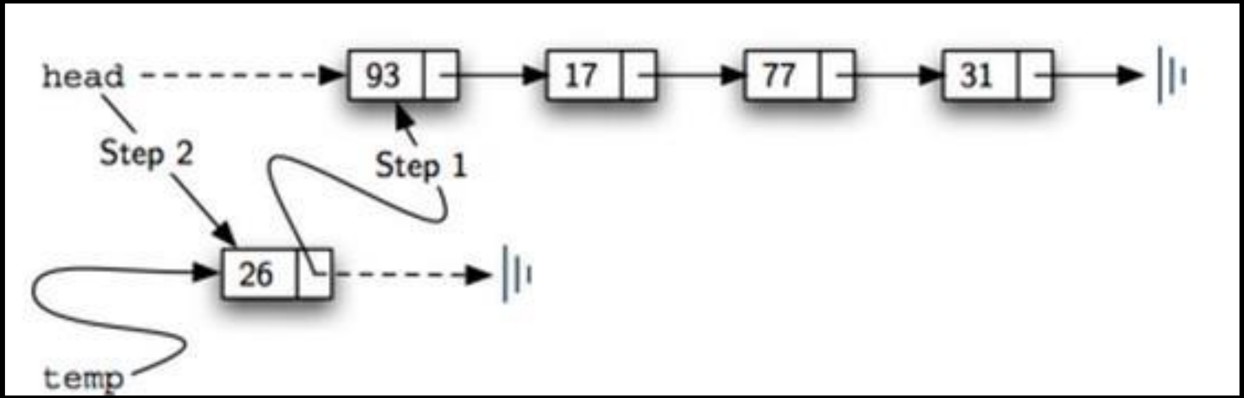
- class Node():
  - element // The data being stored in the node
  - next // A reference to the next node, null for last node, of the type Node
- class List():
  - self.head = None // points to first node of list; null for empty list
  - //this is also known as the head

Linked List - Insertion and Deletion

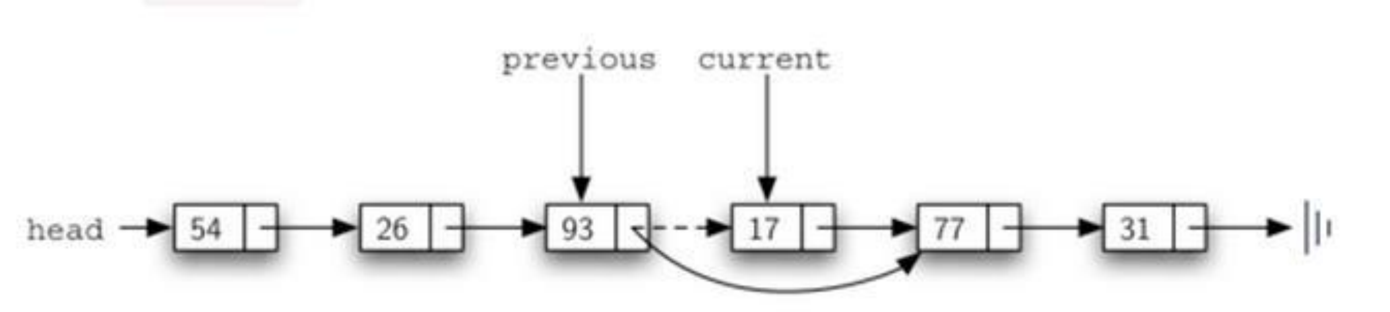
- Insertion can be at head or tail
  - Create new node, and make new node point to head, and make it the new head
  - If using tail pointer, point next of tail to new node, and next of new node to null
- Deletion
  - requires the reorganization of next pointers

The Code

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

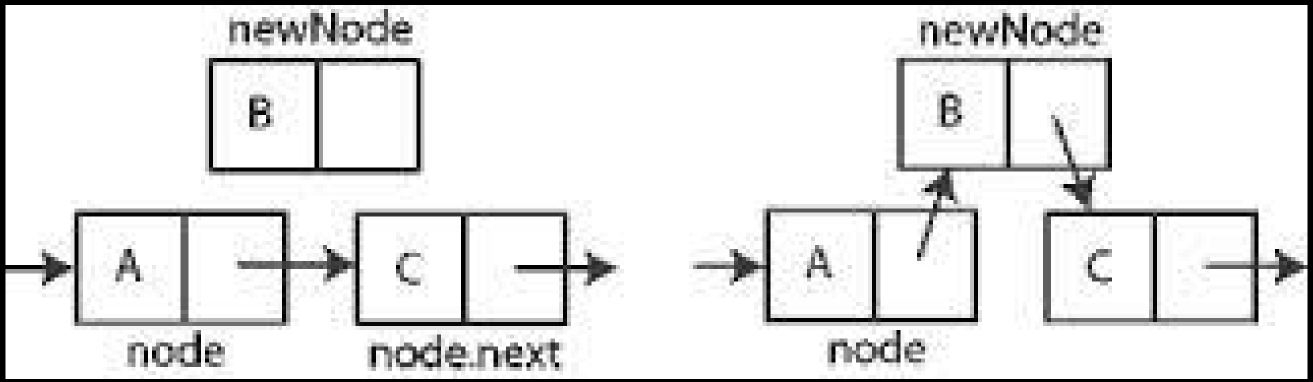


- Deletion
  - Search through the list to find the element (marked as current)
  - `previous.setNext(current.getNext())`


    ```

List ADT: Functions

- Algorithm insertAfter(Node node, Node newNode)
  // insert newNode after node
  newNode.next ← node.next
  node.next ← newNode



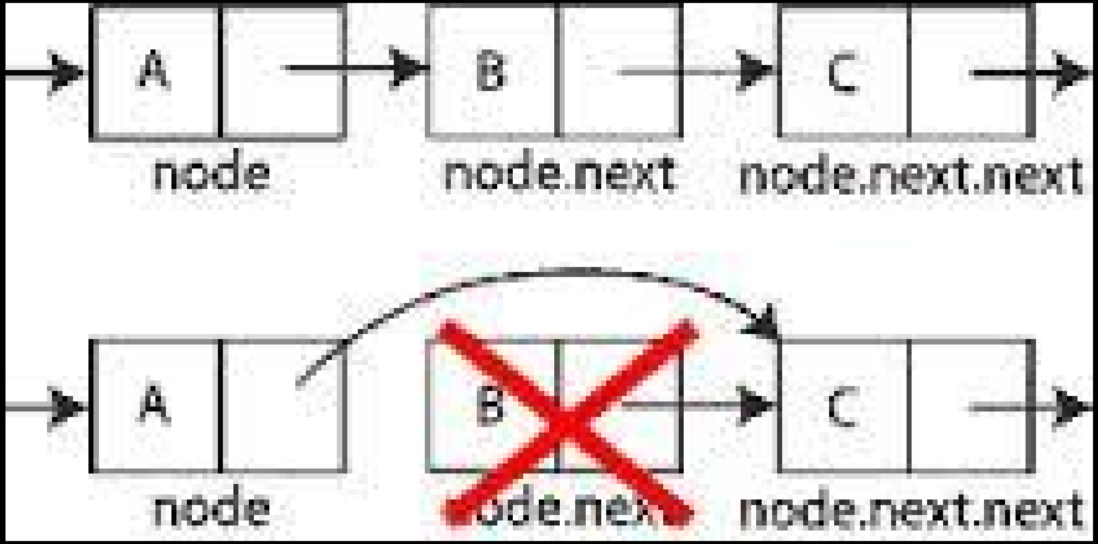
List ADT Functions:

- 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

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

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

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

Quick Summary

- ADT definition
  - Imperative
  - Object Oriented
- Advantages of ADTs
- Stack ADT
- Queue ADT
- Linked List ADT
- Examples
- Exercises

Up Next

Practice.. Practice.. Practice..