Linked Lists

Disadvantages of Python Lists:

1. The length of a dynamic array might be longer than the actual number of elements that it stores.
2. Amortized bounds for operations may be unacceptable in real-time systems.
3. Insertions and deletions at interior positions of an array are expensive.

- Both Arrays and linked list keep elements in a certain order
- Array is one "Big Chunk" of memory used.

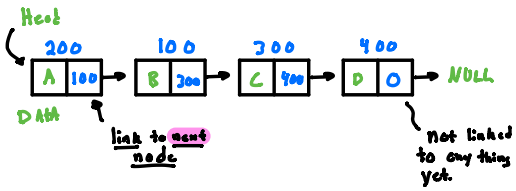
- link list is more distributed representation, in which an object called a "node" is used to allocate for each element, node keeps a reference to its element and the neighbors, to make sure linear order.

Arrays Vs. Linked List

- Lets compare

	Array	Linked List
1) Cost of accessing an element	Constant time - 1(0)	
2.) Memory Requirements	Work on later	
3.) Cost of inserting an element		

Singly Linked Lists



Each Node in a list Consists of 2 parts:

1) pointer
2) data

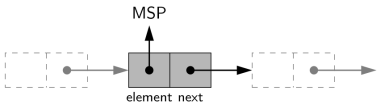
```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize
                          # next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None

# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)
```

- Three Nodes have been created

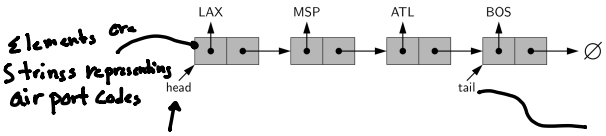
```
'''
Three nodes have been created.
We have references to these three blocks as head,
second and third

llist.head        second              third
     |               |                  |
     |               |                  |
+----+------+     +----+------+     +----+------+
| 1  | None |     | 2  | None |     |  3 | None |
+----+------+     +----+------+     +----+------+
'''

llist.head.next = second; # Link first node with second

'''
Now next of first Node refers to second.  So they
both are linked.

llist.head        second              third
     |               |                  |
     |               |                  |
+----+------+     +----+------+     +----+------+
| 1  |  o-------->| 2  | null |     |  3 | null |
+----+------+     +----+------+     +----+------+
'''

second.next = third; # Link second node with the third node

'''
Now next of second Node refers to third.  So all three
nodes are linked.

llist.head        second              third
     |               |                  |
     |               |                  |
+----+------+     +----+------+     +----+------+
| 1  |  o-------->| 2  |  o-------->|  3 | null |
+----+------+     +----+------+     +----+------+
'''
```

Singly Linked Lists

- Collection of nodes, that form a linear sequence.
- Each node stores a reference to an object that is an Element of the sequence. Each node stores a reference to the next node of the list

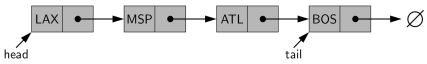


Example of a node instance



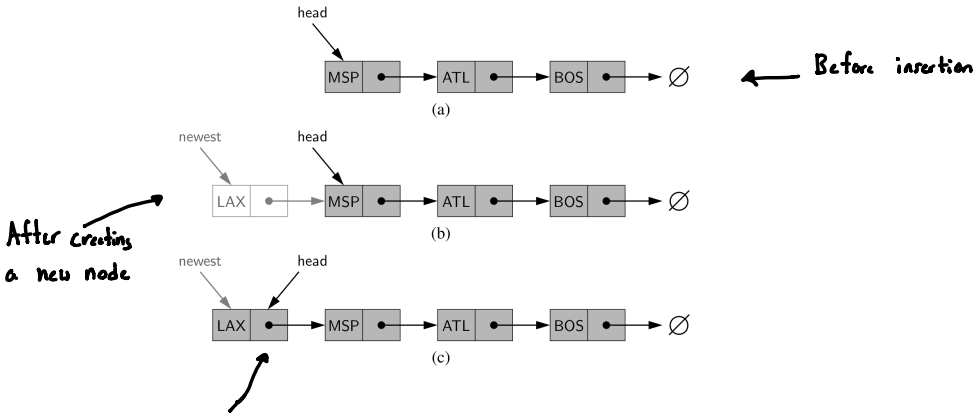
Identify the last node of the list

Identifys the first node of the list



- First and last nodes of a linked list represent the head and tail
- Tail has [none] as its next reference
- Next reference of a node can be called a "pointer" or "link" to another node
- Common for link list instance to keep a count of the nodes

Inserting an Element at the Head of a Singular Linked List

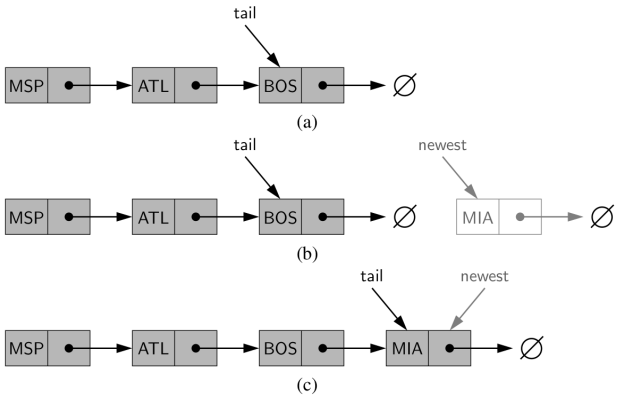


Algorithm add_first(L, e):

```
newest = Node(e)          {create new node instance storing reference to element e}
newest.next = L.head      {set new node's next to reference the old head node}
L.head = newest           {set variable head to reference the new node}
L.size = L.size + 1       {increment the node count}
```

Inserting an Element at the Tail of a Singular Linked List

- Can easily do assuming we have a reference to the tail
- Then set it's pointer to none

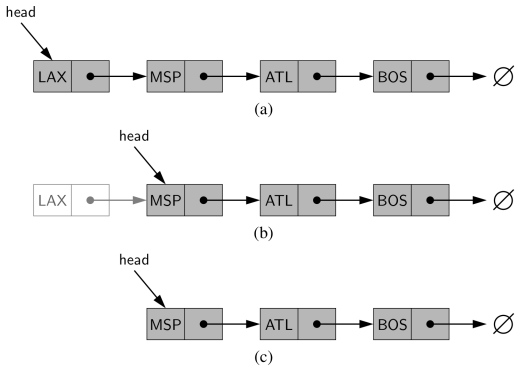


Algorithm add_last(L, e):

```
newest = Node(e)          {create new node instance storing reference to element e}
newest.next = None        {set new node's next to reference the None object}
L.tail.next = newest      {make old tail node point to new node}
L.tail = newest           {set variable tail to reference the new node}
L.size = L.size + 1       {increment the node count}
```

Removing An element from a Singular Linked Array

- Removal of a head is reverse of insertion



Algorithm remove_first(L):
  if L.head is None **then**
    Indicate an error: the list is empty.
  L.head = L.head.next {make head point to next node (or None)}
  L.size = L.size − 1 {decrement the node count}

7.1.1 Implementing a Stack with a Singly Linked List

- Top of the Stack as the head of our list

```
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'          # streamline memory usage

  def __init__(self, element, next):       # initialize node's fields
    self._element = element                # reference to user's element
    self._next = next                      # reference to next node
```

Constructor

Node Only has two instances Variables

Code Fragment 7.4: A lightweight _Node class for a singly linked list.

non-public nested _Node class

```
1  class LinkedStack:
2    """LIFO Stack implementation using a singly linked list for storage."""
3
4    #---------------------- nested _Node class ----------------------
5    class _Node:
6      """Lightweight, nonpublic class for storing a singly linked node."""
7      __slots__ = '_element', '_next'          # streamline memory usage
8
9      def __init__(self, element, next):       # initialize node's fields
10       self._element = element                # reference to user's element
11       self._next = next                      # reference to next node
12
13   #------------------------------ stack methods ------------------------------
14   def __init__(self):
15     """Create an empty stack."""
16     self._head = None                        # reference to the head node
17     self._size = 0                           # number of stack elements
18
19   def __len__(self):
20     """Return the number of elements in the stack."""
21     return self._size
22
23   def is_empty(self):
24     """Return True if the stack is empty."""
25     return self._size == 0
26
27   def push(self, e):
28     """Add element e to the top of the stack."""
29     self._head = self._Node(e, self._head)   # create and link a new node
30     self._size += 1
31
32   def top(self):
33     """Return (but do not remove) the element at the top of the stack.
34
35     Raise Empty exception if the stack is empty.
36     """
37     if self.is_empty():
38       raise Empty('Stack is empty')
39     return self._head._element               # top of stack is at head of list
```

Hidden from the Public.

Each Stack instance maintains two elements:

- _head
- _size ↑ Size of stack

```
40   def pop(self):
41     """Remove and return the element from the top of the stack (i.e., LIFO).
42
43     Raise Empty exception if the stack is empty.
44     """
45     if self.is_empty():
46       raise Empty('Stack is empty')
47     answer = self._head._element
48     self._head = self._head._next            # bypass the former top node
49     self._size -= 1
50     return answer
```

Full Code of our linked Stack

Analysis of our LinkedStack

- At Worst case

Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$