Linked Lists

Disedventeges 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. (How to move the array)

- 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 as "node" is used to allocate for each element, node keeps a reference to its element and the neighbors, to make sure linear order.

- Dynamic size
- Ease of insertion and deletion

- Applications of a linked List:

1.) Images are linked with each other

2.) Music Player to switch between music

Drawbacks:

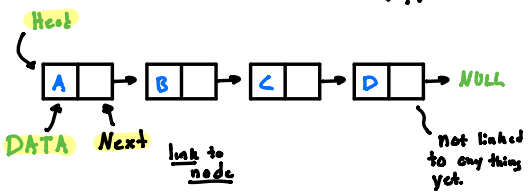
1) Random access is not allowed

2.) Extra memory space for a pointer is required with each element of the List

3.) Not Cache friendly.

Linked Lists Introduction

- A Linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.



- Chain of node objects

Each Node in a list consists of 2 parts:
1) pointer (or reference) to the next node
2) data

Note: Python doesn't have a built-in linked list class

```
# 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.
    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 |
    +----+------+     +----+------+     +----+------+
    '''
```

← Constructor for a Node

~ Empty list

~ Setting the head value to the first node we created in our linked list

← Created 3 Nodes, not linked yet

← Start linking the nodes

← next of first Node now refers to the second

Linked list Traversal

- Traversal, to travel across, over, or through in our case through the linked List.

```python
# A simple Python program for traversal of a linked list

# Node class
class Node:

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


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print (temp.data)
            temp = temp.next

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

    # Start with the empty list
    llist = LinkedList()

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

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

    llist.printList()
```

← Here's where we would assign data.

↑ loop go thru our linked list.

← Call this method/Function to print

Returned:

1
2
3

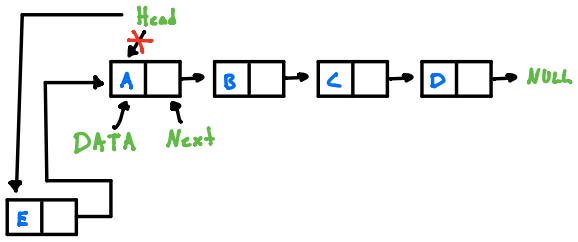
Inserting a node

Methods to insert a new node in the linked list.

1. At the front of the linked list
2. After a given node
3. At the end of the linked list

1. At the front of the linked list

- Newly added node becomes the "new" head of the linked list



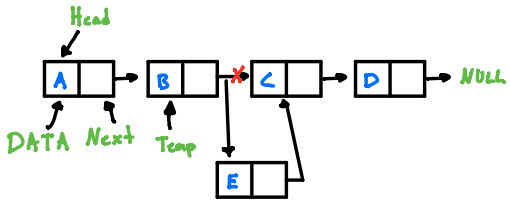
- Lets call the function push(), push must receive a pointer to the head pointer because push must change the head pointer to point to the new node.

4 Steps to add Node at the front:

Time complexity of push() is O(1), as it does constant work

```
# This function is in LinkedList class
# Function to insert a new node at the beginning
def push(self, new_data):

    # 1 & 2: Allocate the Node &
    #        Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

    # 4. Move the head to point to new Node
    self.head = new_node
```

2 Add a node after a given node



- Were given pointer to a node, the new node is inserted after the given node.

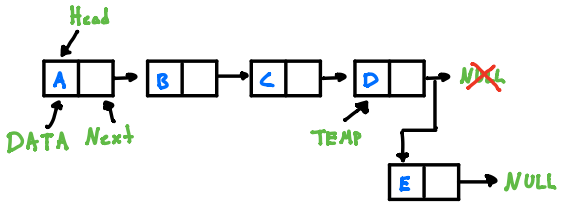
5 Step Process:

```
# This function is in LinkedList class.
# Inserts a new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print("The given previous node must inLinkedList.")
        return

    # 2. Create new node &
    # 3. Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node
```

Time Complexity of insertAfter() is O(1), as it does constant amount of work

Add a node at the end

- New node is added after the last node of the given linked list.
- We have to traverse the list till the end and then change the next of the last node to a new node



6 Step Process:

```
# This function is defined in Linked List class
# Appends a new node at the end.  This method is
#  defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None    ~ Since node will be inserted at the end
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    #    new node as head
    if self.head is None:
         self.head = new_node
         return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

    # 6. Change the next of last node
    last.next =  new_node
```

Time Complexity of append is O(n) where n is the # of nodes in the linked list. Since there is a loop from head to the end, the function does O(n) work

Full Code:

```
# A complete working Python program to demonstrate all
# insertion methods of linked list

# Node class
class Node:

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


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None


    # Functio to insert a new node at the beginning
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        #        Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head

        # 4. Move the head to point to new Node
        self.head = new_node


    # This function is in LinkedList class. Inserts a
    # new node after the given prev_node. This method is
    # defined inside LinkedList class shown above */
    def insertAfter(self, prev_node, new_data):

        # 1. check if the given prev_node exists
        if prev_node is None:
            print "The given previous node must inLinkedList."
            return

        #  2. create new node &
        #      Put in the data
        new_node = Node(new_data)

        # 4. Make next of new Node as next of prev_node
        new_node.next = prev_node.next

        # 5. make next of prev_node as new_node
        prev_node.next = new_node


    # This function is defined in Linked List class
    # Appends a new node at the end.  This method is
    # defined inside LinkedList class shown above */
    def append(self, new_data):

        # 1. Create a new node
        # 2. Put in the data
        # 3. Set next as None
        new_node = Node(new_data)

        # 4. If the Linked List is empty, then make the
        #    new node as head
        if self.head is None:
            self.head = new_node
            return

        # 5. Else traverse till the last node
        last = self.head
        while (last.next):
            last = last.next

        # 6. Change the next of last node
        last.next =  new_node


    # Utility function to print the linked list
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next
```

Our linKedList Class Contains all the "adding node functions

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

    # Start with the empty list
    llist = LinkedList()

    # Insert 6.  So linked list becomes 6->None
    llist.append(6)

    # Insert 7 at the beginning. So linked list becomes 7->6->None
    llist.push(7);

    # Insert 1 at the beginning. So linked list becomes 1->7->6->None
    llist.push(1);

    # Insert 4 at the end. So linked list becomes 1->7->6->4->None
    llist.append(4)

    # Insert 8, after 7. So linked list becomes 1 -> 7-> 8-> 6-> 4-> None
    llist.insertAfter(llist.head.next, 8)

    print 'Created linked list is:',
    llist.printList()
```

Output:

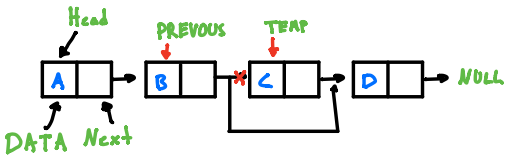
```
Created Linked list is:  1  7  8  6  4
```

Linked List: Deleting a Node

- Given a "key", delete the first occurrence of this key in the linked list

To delete a node from a linked list:

1. Find previous node of the node to be deleted
2. Change the next of the previous node
3. Free memory for the node to be deleted



```python
# Python program to delete a node from linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Given a reference to the head of a list and a key,
    # delete the first occurence of key in linked list
    def deleteNode(self, key):

        # Store head node
        temp = self.head

        # If head node itself holds the key to be deleted
        if (temp is not None):
            if (temp.data == key):
                self.head = temp.next
                temp = None
                return

        # Search for the key to be deleted, keep track of the
        # previous node as we need to change 'prev.next'
        while(temp is not None):
            if temp.data == key:
                break
            prev = temp
            temp = temp.next


      # if key was not present in linked list
      if(temp == None):
          return

      # Unlink the node from linked list
      prev.next = temp.next

      temp = None


    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print (" %d" %(temp.data)),
            temp = temp.next


# Driver program
llist = LinkedList()
llist.push(7)
llist.push(1)
llist.push(3)
llist.push(2)

print ("Created Linked List: ")
llist.printList()
llist.deleteNode(1)
print ("\nLinked List after Deletion of 1:")
llist.printList()
```

Output:

```
Created Linked List:
 2  3  1  7
Linked List after Deletion of 1:
 2  3  7
```

Implementing a Stack with a Singly Linked List

- Top of the Stack as the head of our Linklist ★

```
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
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
```

Class inside our LinkedStack

Hidden from the Public. PRIVATE

Top of our 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()	$O(1)$