Beginners Guide to Algorithms and Data Structures

Python

Abinash Mohanty[1]

August 15, 2017

[1]https://www.linkedin.com/in/abinashm89/

Dedicated to Family & Friends.

Acknowledgements

- A special word of thanks goes to geeksforgeeks[1] (for the awesome database of problem sets and solutions.).
- I'm deeply indebted my parents, colleagues and friends for their support and encouragement.

[1] `http://www.geeksforgeeks.org/`

Contents

Chapter 1

Singly Linked Lists

1.1 Introduction

Linked List is a linear data structure. However, contents of a linked list are not stored in contiguous memory locations. Instead, each element stores the address of the subsequent element. Each time a new element is added to the list, memory needed is allocated on the heap (No need to specify the size at the start). However, the disadvantage is that random accessing elements using index is not possible. Python code for a singly linked list and its node is:

```
class listNode:
    def __init__(self, data):
        self.next = None
        self.data = data

class LinkedList:
    def __init__(self):
        self.head = None
```

1.1.1 Iterating over a list

Since the elements are not stored in continuous memory locations, we cannot access them randomly using index. The only way to iterate over a singly linked list is to start from a node and keep going to the next nodes using the next pointers. The following code demonstrates this by printing all the elements in a linked list:

```
def printList(self):
    tmp = self.head
    myStr = ''
    while(tmp):
        print tmp.data
```

```
        tmp = tmp.next
```

1.2 Insertion

1.2.1 Insert at the start.

This is simple, create a new node, point head of the list to the new node and point the next of the new node to the former head.

```
def push(self, data):
    newNode = listNode(data)
    newNode.next = self.head
    self.head = newNode
```

1.2.2 Insert after a given node.

First find the given node, create a new node and then insert it (change the next pointer of the given node to point to the newly created node and point the next pointer of the new node to the former next node of the given node).

```
def insertAfter(self, prev, data):
    if prev is None:
        print 'Node should be a part of the list'
        return
    newNode = listNode(data)
    newNode.next = prev.next
    prev.next = newNode
```

1.2.3 Insert at the end

Iterate the list starting from the head to the end and then insert the new node there. We need to handle the case when the list is empty.

```
def append(self, data):
    newNode = listNode(data)
    if self.head is None:
        self.head = newNode
    last = self.head
    while(last.next):
        last = last.next
    last.next = newNode
```

1.3 Deletion

1.3.1 Delete the node with the given key

Deleting a node from a linked list is similar to insertion. Find the specified node while keeping track of its previous node. Then point the next pointer of the previous node to the next pointer of the node to be deleted. Delete the node.

```
def deleteNode(self, key):
    tmp = self.head
    if (tmp is not None) and (self.head.data == key):
        self.head = tmp.next
        tmp = None
        return
    while tmp is not None:
        if tmp.data == key:
            break
        prev = tmp
        tmp = tmp.next
    if tmp == None:
        return
    prev.next = tmp.next
    tmp = None
```

1.3.2 Delete the node at given position

Deleting a node from a linked list based on the position is similar to what we did earlier. Find the node previous to the specified node. The corner cases to be addressed are when the position is larger than the size of the list, list is null etc.

```
def deleteNode_position(self, position):
    if self.head == None:
        return
    tmp = self.head
    if position == 0:
        self.head = tmp.next
        tmp = None
    for i in range(position-1):
        tmp = tmp.next
        if tmp == None:
            break
    if (tmp == None) or (tmp.next == None):
        return
```

```
    next = tmp.next.next
    tmp.next = None
    tmp.next = next
```

1.4 Problems

I recommend solving the problem first before moving on to the solution.

1.4.1 Count number of nodes

Write a function to count number of nodes in a given singly linked list. Implement it both in iterative and recursive fashion.

Iterative:

```
def getCount(self):
    count = 0
    tmp = self.head
    while(tmp):
        count += 1
        tmp = tmp.next
    return count
```

Recursive:

```
def getCount_recursive(self, node):
    if node is None:
        return 0
    return 1 + self.getCount_recursive(node.next)

def getCount(self):
    tmp = self.head
    n = self.getCount_recursive(tmp)
    return n
```

1.4.2 Search an element in a Linked List

Write a function that searches a given key 'x' in a given singly linked list. The function should return true if x is present in linked list and false otherwise. Implement it both in iterative and recursive fashion.

```
def search(self, key):
    tmp = self.head
    while (tmp):
        if tmp.data == key:
```

```
            return True
        tmp = tmp.next
    return False
```

1.4.3 Swap nodes in a linked list without swapping data

Given a linked list and two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields.

```
def swapNodes(self, X, Y):
    if X == Y:
        return
    currX = self.head
    prevX = None
    while(currX != None and currX.data != X):
        prevX = currX
        currX = currX.next
    currY = self.head
    prevY = None
    while(currY != None and currY.data != Y):
        prevY = currY
        currY = currY.next
    if currX == None or currY == None:
        return
    if prevX == None:
        self.head = currY
    else:
        prevX.next = currY
    if prevY == None:
        self.head = currX
    else:
        prevY.next = currX
    tmp = currX.next
    currX.next = currY.next
    currY.next = tmp
```

1.4.4 Get Nth node in a Linked List

Write a getNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position.

1.4.5 Nth node from the end of a Linked List

Given a Linked List and a number n, write a function that returns the value at the n'th node from end of the Linked List.

1.4.6 Print the middle of a given linked list

Given a singly linked list, find middle of the linked list. For example, if given linked list is 1->2->3->4->5 then output should be 3. If there are even nodes, then there would be two middle nodes, we need to print second middle element.

1.4.7 Reverse a linked list

Given pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

1.4.8 Detect loop in a linked list

Given a linked list, check if the the linked list has loop or not.

1.4.9 Merge two sorted linked lists

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists.