LINKED LISTS IN PYTHON

José M. Garrido
Department of Computer Science

January 2016

College of Computing and Software Engineering
Kennesaw State University

© 2015 J. M. Garrido
1 Nodes and Linked Lists

A linked list is a data structure that consists of a chain or sequence of nodes connected in some manner. A node is a relatively smaller data structure that contains data and one or more links that are used to connect the node to one more other nodes. In graphical form, a node may be depicted as a box, which is divided into two types of components:

- A data block that stores one or more data components.
- One or more link components that are references to other nodes.

A simple node has a simple data block and one reference to another node. Figure 1 shows a representation of a simple node. Figure 2 illustrates the general form of a simple linked list in which nodes contain a reference to the next node. Note $H$ is a reference to the first node (the head) of the linked list. The last node (Node 3 in Figure 2) has a link that refers to a black dot to indicate that the node has no connection to any other node and the reference of the node has a value None. When comparing linked lists with arrays, the main differences observed are:

- Linked lists are dynamic in size because they can grow and shrink; arrays are static in size.
- In linked lists, nodes are linked by references and based on many nodes; whereas, an array is a large block of memory with the elements located contiguously.
• The nodes in a linked list are referenced by relationship not by position; to find a data item, always start from the first item (no direct access). Recall that access to the elements in an array is carried out using an index.

![Figure 2: A simple linked list.](image)

Linked lists and arrays are considered low-level data structures. These are used to implement higher-level data structures. Examples of simple higher-level data structures are stacks and queues and each one exhibits a different behavior implemented by an appropriate algorithm. More advanced and complex higher-level data structures are priority queues, trees, graphs, sets, and others.

1.1 Nodes

As mentioned previously, a simple node in a linked list has a data block and a reference that connects it to another node. These nodes can be located anywhere in memory and do not have to be stored contiguously in memory. The following listing shows the Python code with a class definition of a node. Class `Node` includes two attributes: the `data` and the reference `next` to another node. The class also defines two methods, the constructor has one parameter with a default value of `None`.

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

    def strnode (self):
        print self.data
```

The following example include several Python statements to create objects of class `None`, with the data as an argument a the default value for the reference to the next node. Note that `nd1` is the reference to a new node with the string "Hi there" as its data. Node object `nd2` is created with 24 as a its data.
nd1 = Node("Hi there")
nd2 = Node(24)
nd1.strnode()
nd2.strnode()

1.2 Definition of a Class for Linked Lists

A linked list is an object that creates, references, and manipulates node objects. A set of operations are defined for the linked list and some of these basic are:

- Create an empty linked list
- Create and insert a new node at the front of the linked list
- Insert a new node at the back of the linked list
- Insert a new node at a specified position in the linked list
- Get a copy of the data in the node at the front of the linked list
- Get a copy of the data in the node at a specified position in the linked list
- Remove the node at the front of the linked list
- Remove the node at the back of the linked list
- Remove the node at a specified position in the linked list
- Traverse the list to display all the data in the nodes of the linked list
- Check whether the linked list is empty
- Check whether the linked list is full
- Find a node of the linked list that contains a specified data item

These operations are implemented as methods in class *LinkedList* and it is shown in the following listing and is stored file *linklistc.py*. In addition to these methods, two attributes are defined, *numnodes* and *head*. The the value of the first attribute *numnodes* is the number of nodes in the linked list. The second attribute *head* is a reference to the first node of the linked list. This node is also known as the *head node* because it is the front of the linked list. In an empty list, the value of *numnodes* is zero and the value of *head* is *None*.  

© 2015 J. M. Garrido
class LinkedList:
    def __init__(self):
        self.numnodes = 0
        self.head = None

    def insertFirst(self, data):
        newnode = Node(data)
        newnode.next = self.head
        self.head = newnode
        self.numnodes += 1

    def insertLast(self, data):
        newnode = Node(data)
        newnode.next = None
        if self.head == None:
            self.head = newnode
            return
        lnode = self.head
        while lnode.next != None:
            lnode = lnode.next
        lnode.next = newnode  # new node is now the last node
        self.numnodes += 1

    def remFirst(self):
        cnode = self.head
        self.head = cnode.next  # new head is second node
        cnode.next = None
        del cnode
        self.numnodes -= 1

    def remLast(self):
        lnode = self.head
        while lnode.next != None:
            pnode = lnode
            lnode = lnode.next
        pnode.next = None
        del lnode
        self.numnodes -= 1

    def getFirst(self):
        lnode = self.head
        return lnode.data

    def getLast(self):
        lnode = self.head
        return lnode.data

    def __str__(self):
        return 'Linked List

        Head: {}

        Nodes:
        {}'.format(self.head, self.numnodes)

    def __len__(self):
        return self.numnodes

    def __iter__(self):
        return self

    def __next__(self):
        if self.head == None:
            raise StopIteration
        lnode = self.head
        self.head = lnode.next
        return lnode.data

# Example usage
linked_list = LinkedList()
linked_list.insertFirst(1)
linked_list.insertLast(2)
linked_list.insertFirst(0)
print(linked_list)  # Output: Linked List

© 2015  J. M. Garrido
1.3 Creating and Manipulating a Linked List

To create an empty list, the constructor in class \textit{LinkedList} is invoked as the following example shows. The assignment statement defines \textit{listObj} that now references an empty linked list object.

\begin{verbatim}
listObj = Linkedlist()
\end{verbatim}

Method \textit{empty} checks whether the list is empty by comparing the value of the head reference \textit{head} with \textit{None}. The following example checks the linked list referenced by \textit{listObj} if empty.

\begin{verbatim}
if listObj.empty() == True:
    ...
\end{verbatim}

A node can be inserted to the linked list at the front, at the back, or in any other place specified. Method \textit{insertFirst} creates and inserts a new node at the front of a linked list, given the data for the node. The new node becomes the head or front node of the linked list and the method increments the value of attribute \textit{numnodes}. Figure 3 shows the insertion of a new node to the front of the list.

Assuming that \textit{newData} refers to the data component for a new node, the following example invokes the method that creates and inserts the node:

\begin{verbatim}
llistObj.insertFirst (newData)
\end{verbatim}
Method \textit{getFirst} returns the data in the first node of the linked list. Method \textit{remFirst} is called to remove and delete the node at the front of the linked list. The following example gets the data then removes the first node of the linked list.

\begin{verbatim}
data = listObj.getFirst()
listObj.remFirst()
\end{verbatim}

Method \textit{getLast} returns the data component of the last node in the linked list. Method \textit{remLast} removes the last node of the linked list. The following example gets the data then removes the last node of the linked list.

\begin{verbatim}
data = listobj.getLast()
listObj.remLast()
\end{verbatim}

Simple \textit{traversal} of a linked list involves accessing every node in the linked list by following the links to the next node until the last node. Recall that the link of the last node is \texttt{None}. The following example calls method \textit{print\_llist} that traverses a linked list to display the data of every node.

\begin{verbatim}
listObj.print\_llist()
\end{verbatim}

The following listing shows a Python script that imports class \textit{Node} and class \textit{LinkedList} to create and manipulate a linked list object. The script is stored in file \texttt{testlinklist.py}.

\begin{verbatim}
from linklistc import Node, LinkedList

print "New linked list"
listObj = LinkedList()
listObj.insertFirst("John")
listObj.insertFirst(99)
listObj.insertFirst(45)
listObj.insertLast(78)
\end{verbatim}

© 2015 J. M. Garrido
listObj.insertLast(88)
listObj.insertLast("Mary")
print "Remove first node"
listObj.remFirst()
print "remove last node"
listObj.remLast()
listObj.print_list()

Using the Python interpreter to run the script, produces the following output:

$ python testlinklist.py
New linked list
45
99
John
78
88
Mary
Remove first node
remove last node
99
John
78
88

More flexibility is obtained by including in the class an operation to insert a node at a specified position in the linked list. For example insert a new node after current node 2. Figure 4 illustrates changing the links so that a new node is inserted after node 2. An enhanced implementation of class Node and class LinkedList is stored in file linklist2c.py.

2 Linked Lists with Two Ends

The linked lists discussed previously have only one end, which include a reference to the first node, and this reference is also known as the head of the linked list. In addition to the head node, providing a reference to the last node gives the linked list more flexibility for implementing some of the operations to manipulate linked list objects.

With two ends, a linked list has two references: one to the first node H, also known as the head or front of the list, and a reference to the last node L, also known
Linked Lists

Figure 4: A new node inserted after node 2.

Figure 5: A linked list with two ends.

as the back of the linked list. Figure 5 illustrates a linked list with a head reference $H$ and a back reference $L$.

The class definition of a two-end linked list $TeLinkedList$ includes an additional attribute, the reference to the last node (the last). An object of this class has the ability to directly add a new node to the back of the linked list without traversing it from the front. In a similar manner, the last node of a linked list can be removed without traversing it from the front. The implementation of this class is stored in file $telinklistc.py$.

Linked lists with two ends are very helpful and convenient for implementing higher-level data structures such as stacks and queues.
3 Double-Linked Lists

Linked lists that have nodes with only one link, a reference to the next node, can only traverse the linked list in one direction, starting at the front and toward the back of the list. A second link is included in the definition of the nodes that is a reference to the previous node. Figure 6 shows a linked list with nodes that have two links: a reference to the next node and a reference to the previous node. Such linked lists are known as doubly linked lists.

![Figure 6: A linked list with two links per node.](image)

The following listing of Python statements defines class `DNode` that can be used for creating and manipulating nodes with two links, `next` that references the next node in the linked list and `prev` that references the previous node in the linked list. Class `DNode` and class `DLinkedList` are implemented in module `dlinklistc.py`.

```python
class DNode:
    def __init__(self, data = None):
        self.data = data
        self.next = None
        self.prev = None

    def strnode(self):
        print self.data
```

4 Stacks and Queues Data Structures

More practical data structures are used in problem solving and can be implemented with linked lists or with arrays. The structure and operations of two simple and widely-known higher-level data structures: queues and stacks are discussed here.

© 2015 J. M. Garrido
4.1 Stacks

A stack is a higher-level dynamical data structure that stores a collection of data items, each of which is stored in a node. Each node in a stack includes a data block and one or more links.

A stack has only one end: the top of the stack. The main characteristics of a stack are:

- Nodes can only be inserted at the top of the stack (TOS)
- Nodes can only be removed from the top of the stack
- Nodes are removed in reverse order from that in which they are inserted into the stack. A stack is also known as a last in and first out (LIFO) data structure.

Figure 7: A stack as a dynamical data structure.

Figure 7 shows a stack and the top of the stack as the insertion point and the removal point. A class for stacks includes the following operations:

- `create_stack`, create an empty stack.
- `empty`, returns true if the stack is empty; otherwise returns false.
- `full`, returns true if the stack is full; otherwise returns false.
- `gettop`, returns a copy of the data block at the top of the stack without removing the node from the stack.
Linked Lists

- **pop**, removes the node from the top of the stack.
- **push**, inserts a new node to the top of the stack.
- **getsize**, returns the number of nodes currently in the stack.

The most direct way to implement a stack is with a single-list linked list in which insertions and deletions are performed at the front of the linked list. The two-ended linked list class *TeLinkedList* is used to implement class *Stack*, which is stored in files *stack.py*. The following listing shows the Python source code of class *Stack*.

```python
1 # A simple class for a stack using two-ended Linked List
2 from telinklistc import TeLinkedList
3
4 class Stack:
5    capacity = 100
6    def __init__(self):
7        self.list = TeLinkedList()
8
9    def empty(self):
10       if self.list.numnodes == 0:
11          return True
12       else:
13          return False
14
15    def full(self):
16       if self.list.numnodes == capacity:
17          return True
18       else:
19          return False
20
21    def push(self, data):
22       self.list.insertFirst(data)
23
24    def pop(self):
25       self.list.remFirst()
26
27    def get_top(self):
28       data = self.list.getFirst()
29       return data
30
31    def getSize(self):
32       lsize = self.list.numnodes
33       return lsize
```

© 2015  J. M. Garrido
Linked Lists

```python
34 35 def printStack(self):
36     self.list.print_list()
```

The Python commands that create a stack object and manipulate the stack are included in the following listing and stored in file `teststack.py`.

```python
from stack import Stack

print "New stack"
listObj = Stack()
listObj.push("John")
listObj.push(99)
listObj.push(45)
print "TOS: ", listObj.get_top()
print "Stack empty? ", listObj.empty()
listObj.push(78)
listObj.push(88)
print "TOS: ", listObj.get_top()
listObj.pop()
print "TOS: ", listObj.get_top()
listObj.push(204)
print "TOS: ", listObj.get_top()
listObj.printStack()
```

The following listing shows the Python interpreter running script `teststack.py` and the results produced.

```
$ python teststack.py
New stack
TOS: 45
Stack empty? False
TOS: 88
TOS: 78
TOS: 204
Size of stack: 5
204
78
45
99
John
```
4.2 Queues

A queue is a dynamical data structure that stores a collection of data items or nodes and that has two ends: the head and the tail. The basic restrictions on manipulating a queue are:

- Nodes or data items are inserted at the tail of the queue
- Nodes or data items are removed from the head of the queue
- Nodes or data items are removed in the same order that they were inserted into the queue and is also known as a first in and first out (FIFO) data structure.

Figure 8 illustrates the form of a queue. It shows the insertion point at the tail and the removal point at the head of the queue. The relevant operations for manipulating a queue are:

- `empty`, returns true if the queue is empty; otherwise returns false.
- `full`, returns true if the queue is full; otherwise returns false.
- `getHead`, returns a copy of the data object at the head of the queue without removing the object from the queue.
- `removeHead`, removes the head item from the queue
- `insertTail`, inserts a new data item into the tail of the queue.
- `getsize`, returns the number of data items currently in the queue.

Figure 8: A queue as a dynamical data structure.

Queues can be implemented with single-linked lists, but a good way to implement a queue class is with a linked list with two ends. Class `Queue` is implemented with
class `TeLinkedList`, which has already defined most of the needed operations. The following listing shows the Python source code of class `Queue`, which is stored in file `queue.py`.

```python
# A simple class for a queue using two-ended Linked List
from telinklistc import TeLinkedList

class Queue:
    capacity = 100
    def __init__(self):
        self.list = TeLinkedList()

def empty(self):
    if self.list.numnodes == 0:
        return True
    else:
        return False

def full(self):
    if self.list.numnodes == capacity:
        return True
    else:
        return False

def insertTail(self, data):
    self.list.insertLast(data)

def getHead(self):
    ldata = self.list.getFirst()
    return ldata

def removeHead(self):
    self.list.remFirst()

def getSize(self):
    lsize = self.list.numnodes
    return lsize

def printQueue(self):
    self.list.print_list()
```

The following Python script is used to test class `Queue`. It creates an object of the class and inserts and removes several data items.
from queue import Queue

print "New queue"
listObj = Queue()
listObj.insertTail("John")
print "Head: ", listObj.getHead()
listObj.insertTail(99)
listObj.insertTail(45)
print "Queue empty? ", listObj.empty()
listObj.insertTail(78)
listObj.insertTail(88)
listObj.removeHead()
print "Head: ", listObj.getHead()
listObj.insertTail(204)
print "Size of queue: ", listObj.getSize()
listObj.printQueue()

The following listing shows the Linux shell commands that compile, link, and execute the program. The results produced by the program execution are also shown.

$ python testqueue.py
New queue
Head: John
Queue empty? False
Head: 99
Size of queue: 5
99
45
78
88
204