Assignment 3 - Linked List

Computer Programming 2

Robert Boerwinkle

Spring 2022

1 Introduction

This is an implementation of a linked list written in python. A linked list is a list whose items are not contiguous in memory. Instead, each value has a pointer to the next value. This value / pointer pair is called a node. The behavior of the functions is based on `collections.deque` . Examples will be performed on an simple list: `exlist = LinkedList((5,8,12,2))` . The full code can be found in `program.py` .

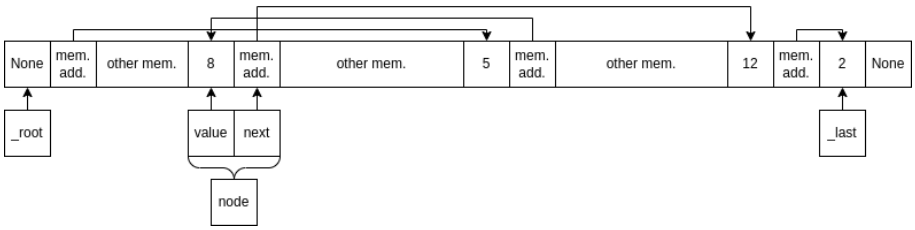

Figure 1: How exlist might look in memory

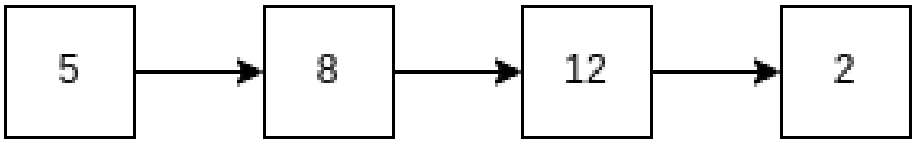

Figure 2: A simpler representation of exlist

Here is the entire `Node` class and the `LinkedList` initializer. The `_last` and `_length` are not necessary, but provide shortcuts for many functions.

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

        def __str__(self):
                return str(self.value)

class LinkedList:
        def __init__(self, values=()):
                self._root = Node(None)
                self._last = self._root
                self._length = 0
                for value in values:
                        self.append(value)
```

2 Built-in / Internal Functions

Iterating through a linked list is somewhat complicated, but very important. Being able to use a `for` loop is a useful ability. To do this, the `LinkedList` needs an `__iter__` function, defined below.

```
def __iter__(self):
        current = self._root
        while True:
                current = current.next
                try:
                        yield current.value
                except AttributeError:
                        break
```

Another important ability is to get the value at an index. This function is what is called when square bracket notation (`exlist[i]`) is used to get values. Setting values is done with the `__setitem__` function, which looks very similar.

```
def __getitem__(self, index):
        index = self._trunci(index, includeLen=False)
        current = self._root
        for i in range(index+1):
                current = current.next
        return current.value
```

3 Final Product Functions

`append` uses the internal `_last` variable. `count` makes use of the `for` loop syntax to iterate through the linked list. These shortcuts, as well as other internal functions allow the rest functions to be similarly simple.

```
def append(self, x):
        self._last.next = Node(x)
        self._last = self._last.next
        self._length += 1

def count(self, x):
        count = 0
        for value in self:
                if x == value:
                        count += 1
        return count
```

The `insert` function is where the internal functions come in handy, and where the most complex manipulation of the linked list happens.

```
def insert(self, i, x):
        i = self._clampi(i)
        inode = self._getnode(i-1)
        inode.next = Node(x, inode.next)
        self._length += 1
        if not inode.next.next:
                self._last = inode.next
```

`_clampi` is another internal function to mimic the indexing patterns of some `collections.deque` functions. `_getnode` works like the `__getitem__` function, except it gets the node itself, not the node's value.

Here is an illustration of `exlist.insert(2, 5)` :

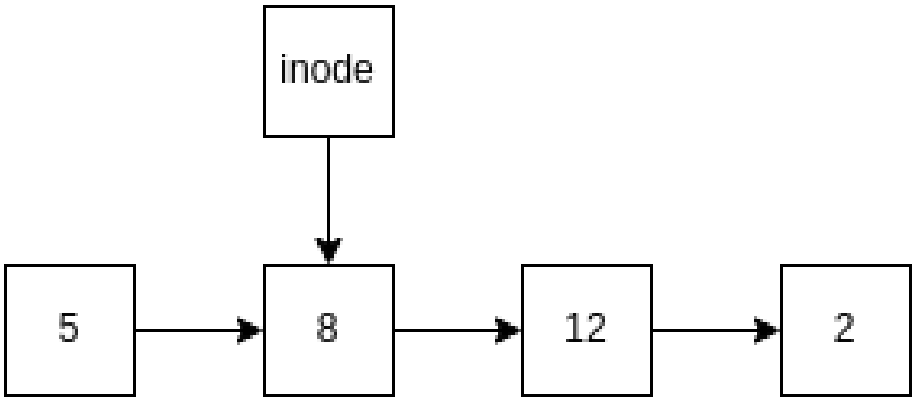

Figure 3: the inode is set to the node before the index; lines 1-2

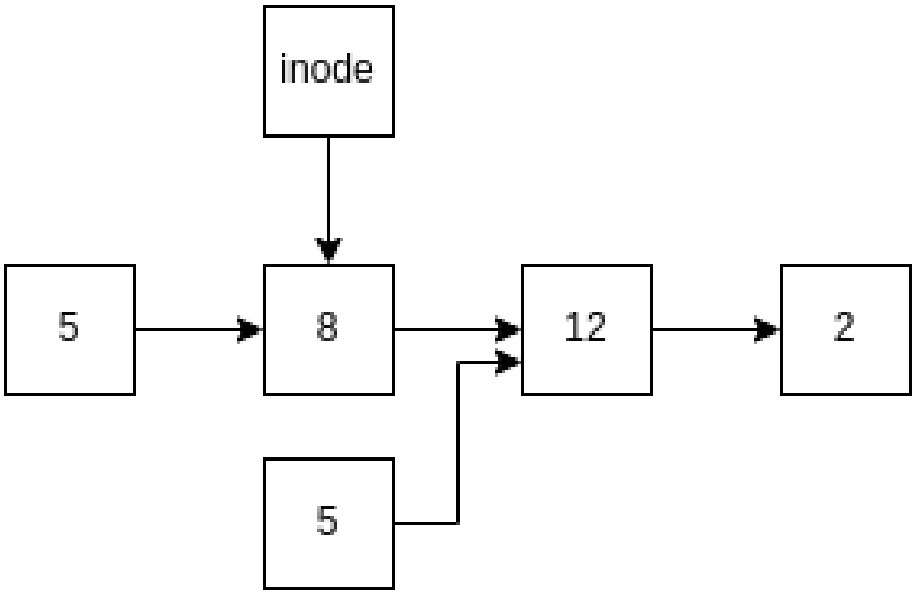

Figure 4: A new node is made whose next is the same as the inode's; second part of line 3

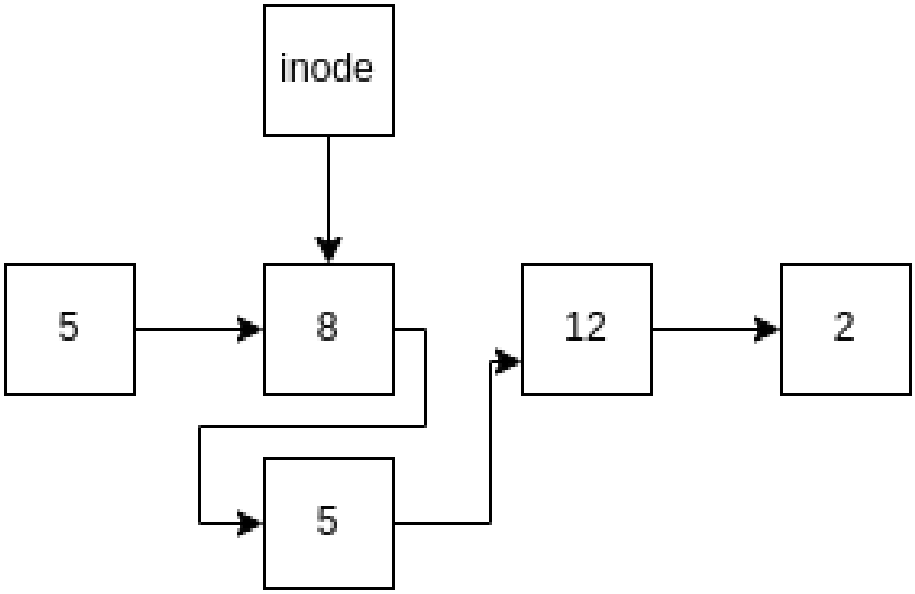

Figure 5: inode's next is set to the new node; first part of line 3

4 Speeds

Speed tests were done on some of the functions to compare to `deque` times. These tests were done on a Raspberry Pi 3 Model B so that background tasks on my personal computer wouldn't affect times. As a result though, all tests ran out of memory before they hit the 5-minute alloted time (except Linked List Pop). The speed tests, as well as tests to ensure the list behaviour mimics `deque` can be found in `test.py` .

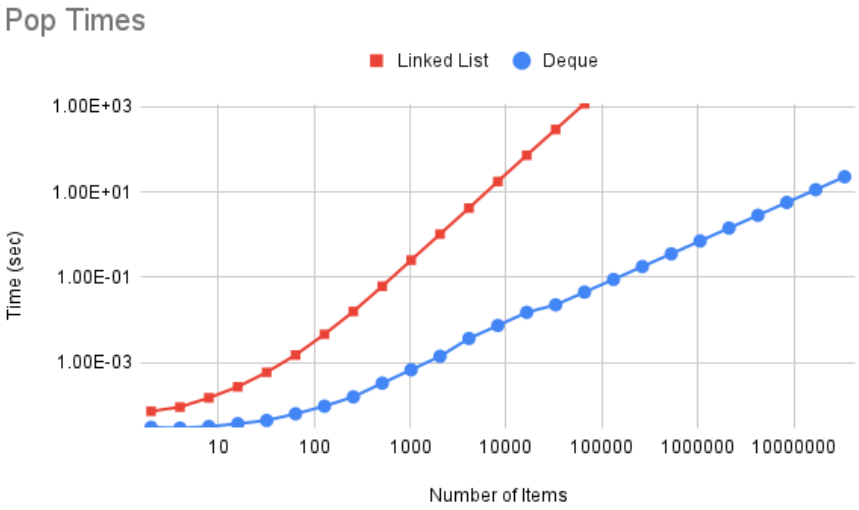

Figure 6: Time it takes to pop all items from a linked list / deque of a given size

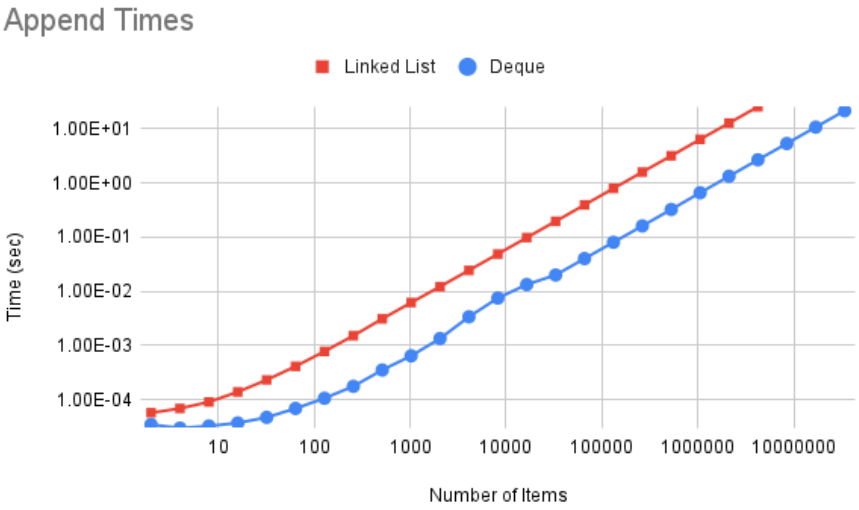

Figure 7: Time it takes to append a given number of items to an empty linked list / deque

The main reason for the discrepancy for `pop` is the fact that it requires the second to last node to be known. A doubly linked list like `collections.deque` makes it easy to get that node, because it keeps track of the beginning / end and each node's next / previous nodes. A singly linked list like `LinkedList` requires stepping all the way through the list.

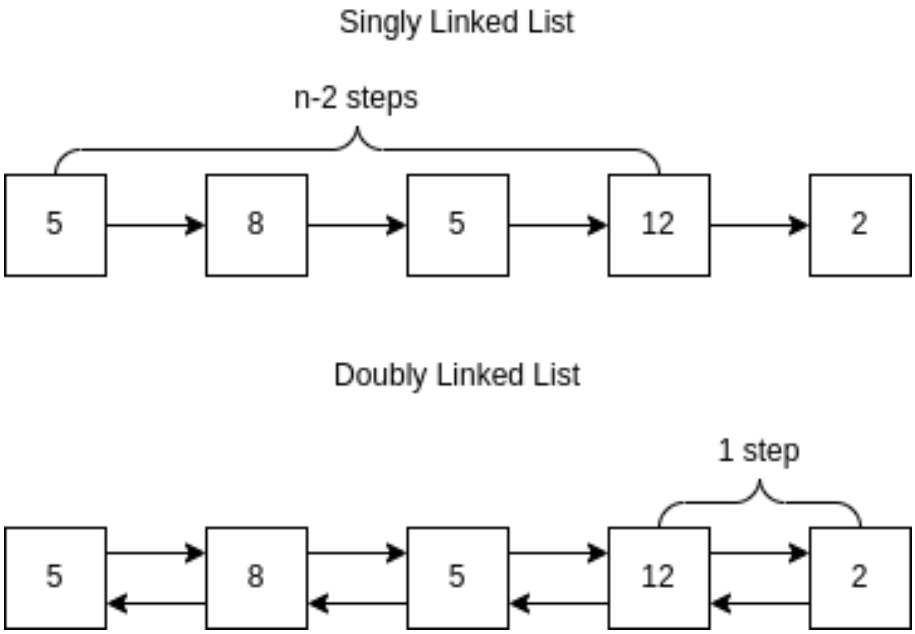

Figure 8: Steps needed to get the node at index -1

5 Conclusion

Linked lists are only useful in certain situations, because of their slow walk to a given index. They are mostly useful when you only want things at the beginning of the list (or either end if the list is doubly linked).

This project involved on manipulating custom objects in python, and code testing / timing. It required integration into python's built-in syntax, automated tests, and automated timing.