# Linked List Implementation

by

**Brent Orlina**

Computer Programming II

Nicholas Seward

# Implementation

A `linked list` consists of a collection of `nodes`, storing a value and a pointer either one or two pointers.

A singly linked list only points to its next `node`, while a doubly linked list points both to its next and previous node.

Linked lists allow for fast and efficient insertion and removal in both ends of the list. However, singly linked lists only allow for this efficiency to happen when inserting and removing at the beginning of the list.

An implementation of a singly linked list is benchmarked against Python's `list` and `deque`

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

class LinkedList:
  def __init__(self):
    self.node_count = 0
    self.head = None
    self.tail = None
```

# Methods Implemented

The methods that were implemented include:

- `append(value)`
- `insert(index, value)`
- `extend(list)`
- `pop(index)`
- `remove(value)`
- `count(value)`
- `index(value)`
- `reverse()`

Arguably, the three most important methods are `append()`, `insert()` and `pop()`, which are the three methods that were later benchmarked.

```python
class LinkedList:
  def __init__(self):
    self.node_count = 0
    self.head = None
    self.tail = None

  def append(value):
    pass

  def insert(index, value):
    pass

  def extend(other):
    pass

  def pop(index):
    pass

  def remove(value):
    pass

  def reverse():
    pass
```

# Benchmarking Methods

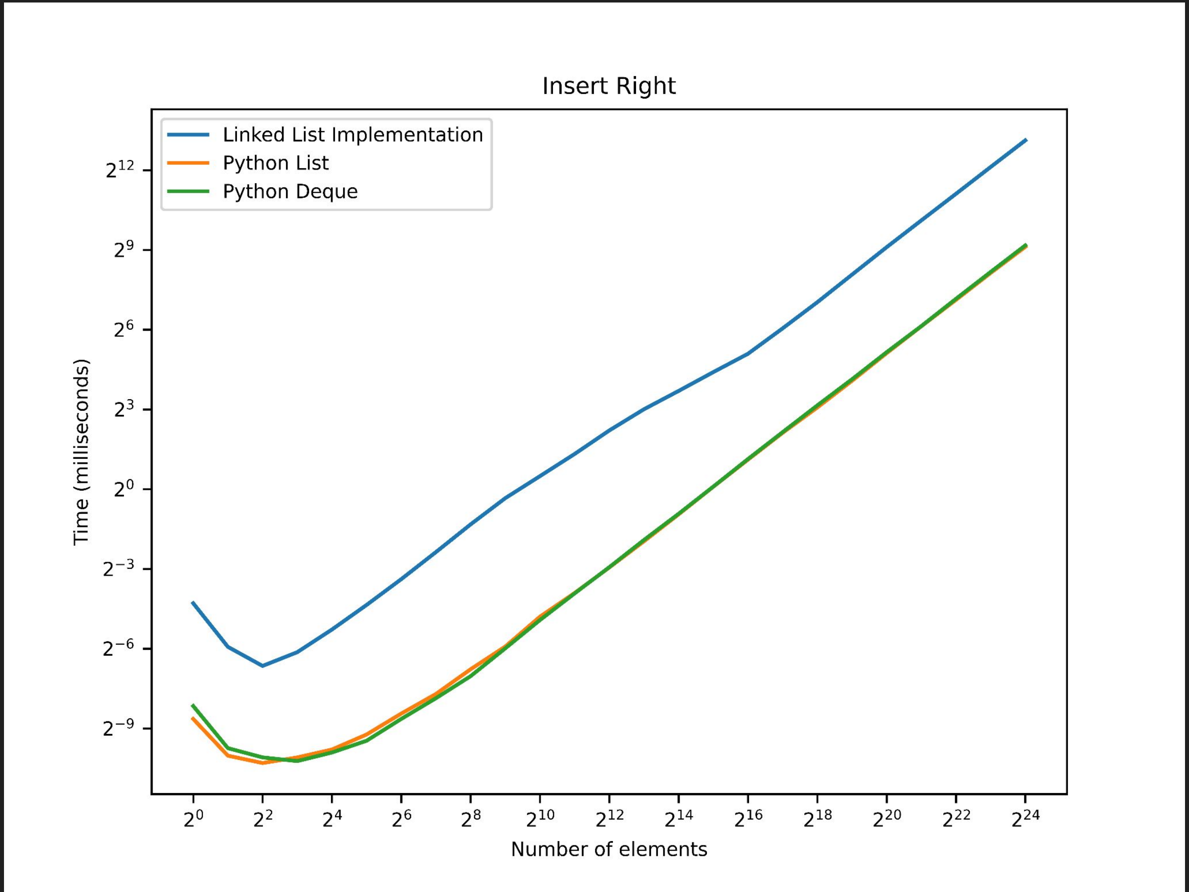
The methods `append()`, `insert()`, and `pop()` were benchmarked against the respective methods of Python's `list` and `deque` classes. Benchmarking the `insert()` method was done by inserting elements at the beginning of the list.

Each method was benchmarked from `2^0` to `2^24` number of elements. Although some methods in the implementation could not reach `2^24` as it exceeded a time limit.

The built-in Python library `timeit` was used as the benchmarking tool for timing each method's performance.

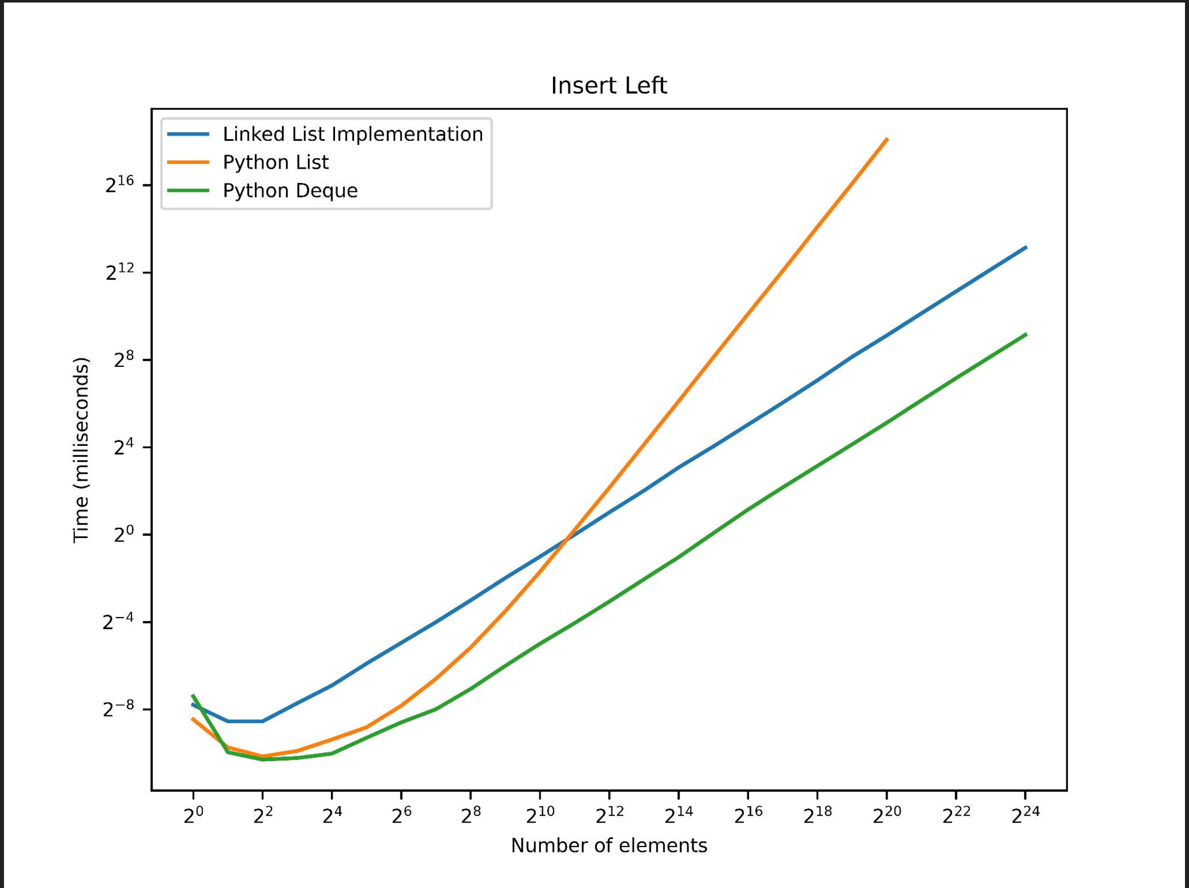
```python
# Benchmarking inserting elements from the left

from collections import deque
from timeit import Timer

# Linked List
for n in range(25):
  L = LinkedList()
  time = Timer(lambda: L.insert(0, 1))
             .repeat(repeat=1, number=2**n)

# Python List
for n in range(25):
  L = list()
  time = Timer(lambda: L.insert(0, 1))
             .repeat(repeat=1, number=2**n)

# Python Deque
for n in range(25):
  L = deque()
  time = Timer(lambda: L.appendleft(1))
             .repeat(repeat=1, number=2**n)
```

# append() Data



Insert Right
Linked List Implementation
Python List
Python Deque
Time (milliseconds)
$2^{12}$
$2^{9}$
$2^{6}$
$2^{3}$
$2^{0}$
$2^{-3}$
$2^{-6}$
$2^{-9}$
$2^{0}$ $2^{2}$ $2^{4}$ $2^{6}$ $2^{8}$ $2^{10}$ $2^{12}$ $2^{14}$ $2^{16}$ $2^{18}$ $2^{20}$ $2^{22}$ $2^{24}$
Number of elements

# insert() Data


Insert Left
Linked List Implementation
Python List
Python Deque
Time (milliseconds)
$2^{16}$
$2^{12}$
$2^{8}$
$2^{4}$
$2^{0}$
$2^{-4}$
$2^{-8}$
$2^{0}$ $2^{2}$ $2^{4}$ $2^{6}$ $2^{8}$ $2^{10}$ $2^{12}$ $2^{14}$ $2^{16}$ $2^{18}$ $2^{20}$ $2^{22}$ $2^{24}$
Number of elements

# `append()` and `insert()` Results

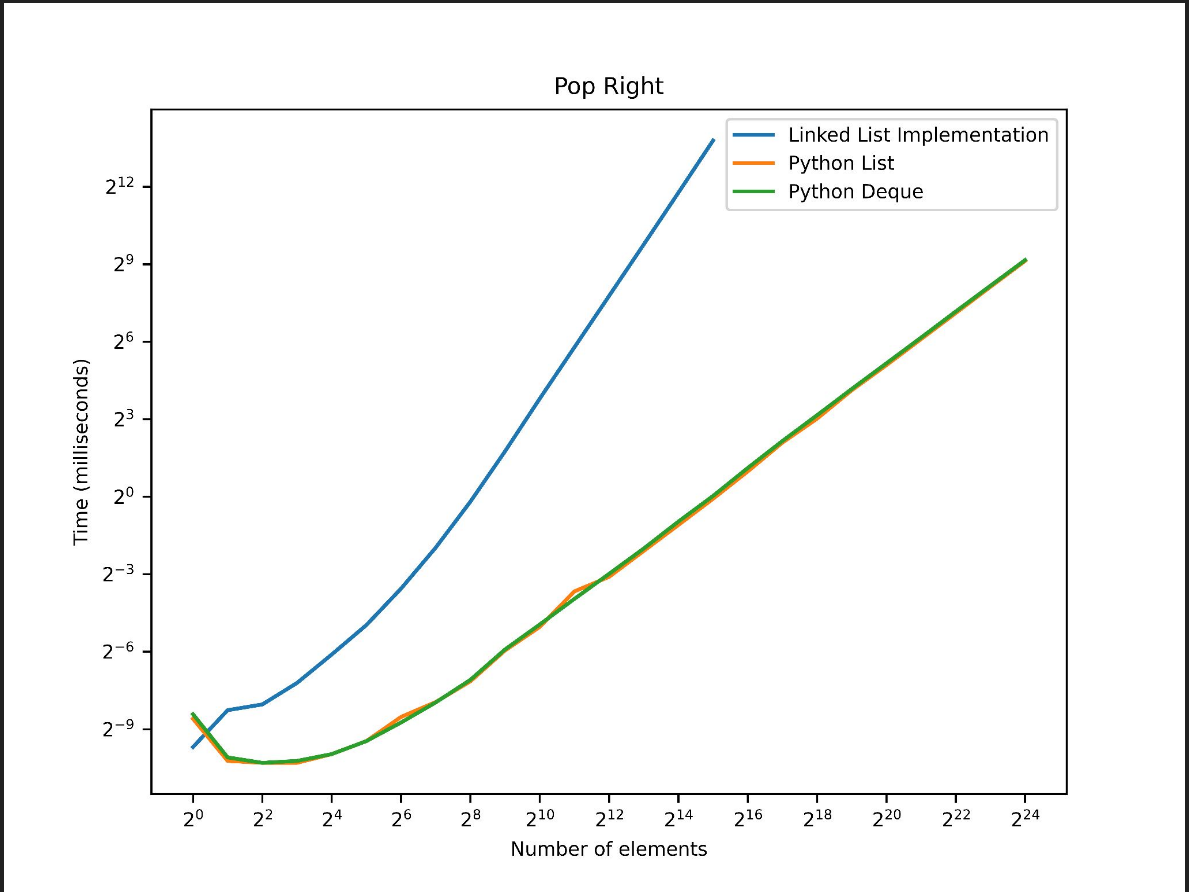
The data shows that the `linked list` implementation is slower compared to Python's `list` and `deque` when using `append()` which had similar speeds.

The `insert()` data shows a similar trend, however, list becomes slower than the linked list at around `2^11` elements.

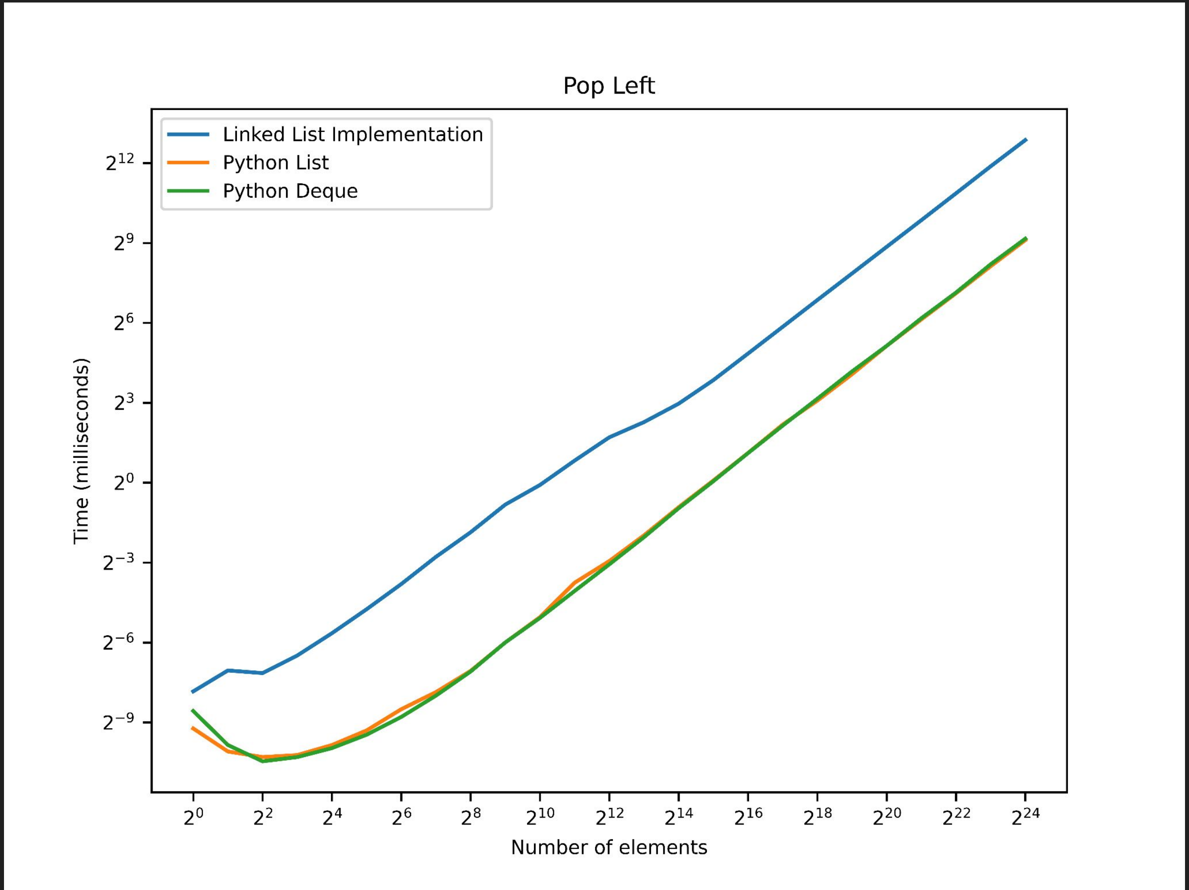
The time complexity of these methods are around `O(1)`, with the exception of list using `insert()`, which is around `O(n)`

All three classes grow similarly at around `O(n)` , with an exception to the `list` using `insert()` at `O(n^2)`.

# pop() Data: Right



# pop() Data: Left



# `pop()` Results

Similar to the `append()` and `insert()` results, the `pop()` data shows the `linked list` implementation were slower than `list` and `deque` which, again, had similar speeds.

The time complexities of `pop()` for `list` and `deque` are `O(1)` and `O(n)` for the `linked list`. However, using `pop()` at the beginning of a `linked list` only takes `O(1)` as it does not have to traverse the list.

All three classes grow similarly at `O(n)` when popping elements at the beginning of the list. However, popping elements at the end of the list causes the `linked list` to grow at `O(n^2)` while `list` and `deque` stay at `O(n)`.

# Conclusion

The `linked list` implementation's `append()`, `insert()`, and `pop()` generally performs worse than Python's `list` and `deque`. These methods have the time complexity of around `O(1)` with the exception of the `pop()` the `linked list` and list's `insert()` which are `O(n)`.

The difference in speed could be explained as both `list` and `deque` are implemented in C, making it inherently faster than a Python implementation. `deque` is also implemented as a doubly linked list, allowing efficiency on both ends of the list.

linked lists can be extremely useful when it's used as a queue. However, re-implementing a linked list in Python is useless as `deque` uses a doubly linked list and is implemented in C, which makes it faster and more efficient.