CSC148-Section:L0301/L0401
Week#4-Wednesday

Instructed by
AbdulAziz Al-Helali
a.alhelali@mail.utoronto.ca
Office hours: Wednesday 11-1, BA2230.

Slides adapted from Professor Danny Heap course material
winter17
Outline

• Linked Lists
  • Implement methods in LinkedList – Cont..
    • delete_front
    • pop_front
    • delete_back
  • Rebuild Stack/Queue using LinkedList
• Test Performance
  • Stack/Queues using Python list vs Stack/Queues using LinkedList
def delete_front(self) -> None:
    
    """
    Delete LinkedListNode self.front from self. 
    Assume self.front is not None
    """

>>> lnk = LinkedList()
>>> lnk.prepend(0)
>>> lnk.prepend(1)
>>> lnk.prepend(2)
>>> lnk.delete_front()
>>> str(lnk.front)
'1 ->0 ->|'
>>> lnk.size
2
>>> lnk.delete_front()
>>> lnk.delete_front()
>>> str(lnk.front)
'None'

"""
delete_front

• Two cases:
  • The list has only one node
  • The list has more than one node
delete_front

• Two cases:
  • The list has only one node
  • The list has more than one node
def delete_front(self) -> None:
    
    if self.size == 1:
        self.front = self.back = None
    else:
        new_front = self.front.next_
        self.front = new_front
        self.size -= 1
```python
def pop_front(self):
    """
    Remove self.front and return its value. Assume self.size >= 1
    
    @param LinkedList self: this LinkedList
    @rtype: object
    
    >>> lnk = LinkedList()
    >>> lnk.append(0)
    >>> lnk.append(1)
    >>> lnk.pop_front()
    0
    """
```

We need this method to implement stack and queues using LinkedList. We can not use delete_front because it returns nothing.
def pop_front(self):
    ""
    ""
    first = self.front.value
    self.delete_front()
    return first
**delete_back**

```python
def delete_back(self) -> None:
    
    """
    Delete LinkedListNode self.back from self.
    Assume self.back is not None
    >>> lnk = LinkedList()
    >>> lnk.prepend(0)
    >>> lnk.prepend(1)
    >>> lnk.prepend(2)
    >>> lnk.delete_back()
    >>> str(lnk.back)
    '1 ->|'
    >>> lnk.size
    2
    >>> lnk.delete_back()
    >>> lnk.delete_back()
    >>> str(lnk.back)
    'None'
    """
```
delete_back

• We need to find the second last node. Walk two references along the list.

```python
prev_node, cur_node = None, lnk.front
# walk along until cur_node is lnk.back
while <some condition>:
    prev_node = cur_node
    cur_node = cur_node.nxt
```
Two cases:

• The list has only one node
• The list has more than one node
delete_back

• Two cases:
  • The list has only one node
  • The list has more than one node

1. If \( \text{size} = 1 \)
   
   ![Diagram of a list with size 1]

2. Otherwise
   
   ![Diagram of a list with more than one node]

   - We need to walk the list to reach this node
   - Very large number
delete_back

def delete_back(self) -> None:
    
    if self.size == 1:
        self.front = self.back = None
    else:
        prev_node = None
        cur_node = self.front
        while cur_node != self.back:
            prev_node = cur_node
            cur_node = cur_node.next_
        self.back = prev_node
        self.back.next_ = None
        self.size -= 1
Rebuild Stack/Queue using LinkedList

- something linked lists do better than Python lists?
  
  Python list:
  \[L=[1,2,3,\ldots,100]\]

  Is \textbf{Remove} from end a \textbf{problem}? 
  Is \textbf{add} to end a \textbf{problem}?

  Is \textbf{Remove} from beginning a \textbf{problem}? 
  Is \textbf{add} to beginning a \textbf{problem}? 

  Will this impact the performance of a Queue or Stack that uses python list?
Rebuild Stack/Queue using LinkedList

• something linked lists do better than lists?
  • Adding to the back and removing from the front

• list-based Queue has a problem: adding or removing will be slow.
Rebuild Stack/Queue using LinkedList

• Do the following changes:
  • **Stack**
    • self._storage = LinkedList()
    • add
      • self._storage.prepend(obj)  # adding to the front
    • Remove
      • return self._storage.pop_front()  # removing to the front
  • **Queue**
    • self._storage = LinkedList()
    • add
      • self._storage.append(obj)  # adding to the back
    • Remove
      • return self._storage.pop_front()  # removing from to the back
Test Performance

```python
def container_cycle(c: Container, r: int) -> str:
    ""
    Add/remove c r times.
    c - Container to add/remove
    r - number of times to add/remove
    ""
    start = time()
    for i in range(r):
        c.add(i)
    start = time()
    for i in range(r):
        # repeatedly add and remove
        c.remove()
        c.add(i)
    print("{} add/remove in {} seconds".format(r, time() - start))
```
Test Performance

![Graph showing performance comparison between different data structures. The x-axis represents size, the y-axis represents time (sec). The graph compares Python List Stack, LL List Stack, Python List Queue, and LL List Queue.](image-url)
Where Can I find the code presented in class

• You can find the full code in the course website under section **MWF2 (L0301) and MWF3 (L0401)**

• with the following file names:
  • linked_list_Wednesday.py
  • stack_as_list.py
  • stack_as_ll.py
  • queue_as_list.py
  • queue_as_ll.py
  • container.py
  • testing_performance.py

• Download them Try different things with them and practice
  • Do not be afraid of doing mistakes