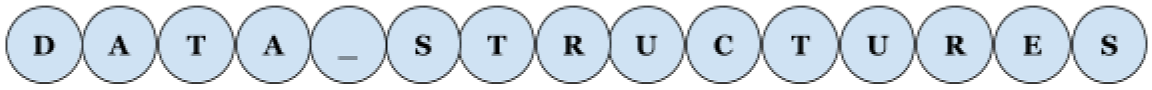
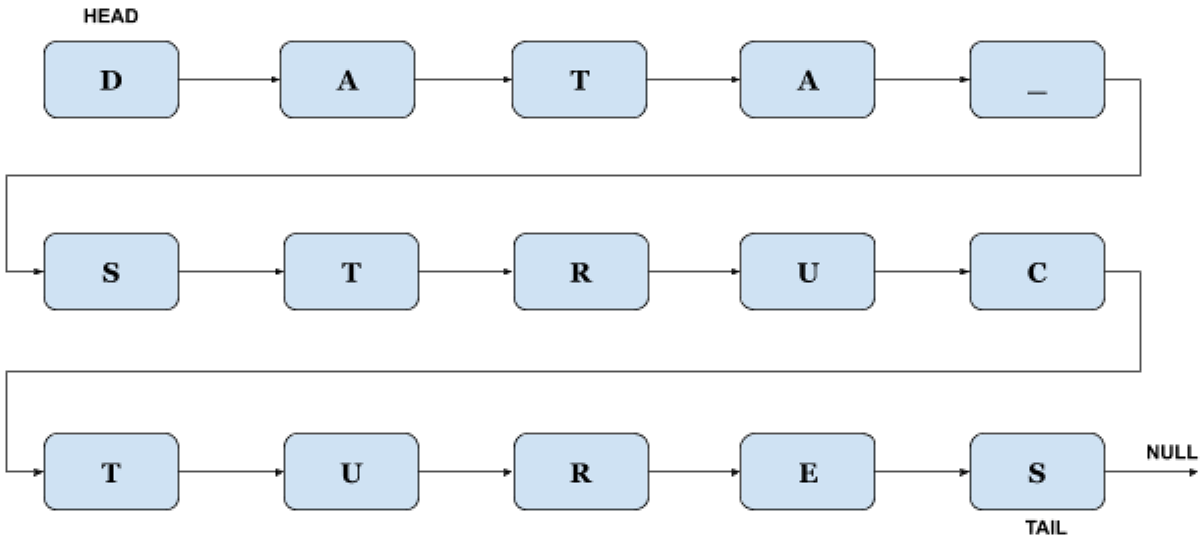
CS261 Data Structures

Assignment 3

v 1.03 (revised 4/21/2020)

Your Very Own Linked List





Contents

General Instructions

1. Your project must be written in Python v3 and submitted to Gradescope before the due date specified in the syllabus.

2. In Gradescope, your code will run through several tests. Any failed tests will provide a brief explanation of testing conditions to help you with troubleshooting. Your goal is to pass all tests.

3. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

4. You are encouraged to develop your own additional tests even though this work won't have to be submitted and won't be graded. Gradescope tests are limited in scope and may not cover all edge cases. Your submission must work on all valid inputs. We reserve the right to test your submission with more tests than Gradescope.

5. Your code must have an appropriate level of comments.

6. You will be provided with a starter "skeleton" code, on which you will build your implementation. Methods defined in skeleton code must retain their names and input / output parameters. Variables defined in skeleton code must also retain their names. We will only test your solution by making calls to methods defined in the skeleton code and by checking values of variables defined in the skeleton code. You can add more functions and variables, as needed.

7. Both the skeleton code and code examples provided in this document are part of assignment requirements. Please read all of them very carefully. They have been carefully selected to demonstrate requirements for each method. Refer to them for the detailed description of expected method behavior, input / output parameters, and handling of edge cases.

8. For each method, you can choose to implement a recursive or iterative solution. When using a recursive solution, be aware of maximum recursion depths on large inputs. We will specify the maximum input size that your solution must handle.

9. If, after reading this entire document, you still have questions about any requirements of this assignment, please post your questions on Piazza.

Part 1 - Summary and Specific Instructions

1. Implement a Singly Linked List class with Deque and Bag ADT interfaces by completing provided skeleton code. Once completed, your implementation will include the following methods:

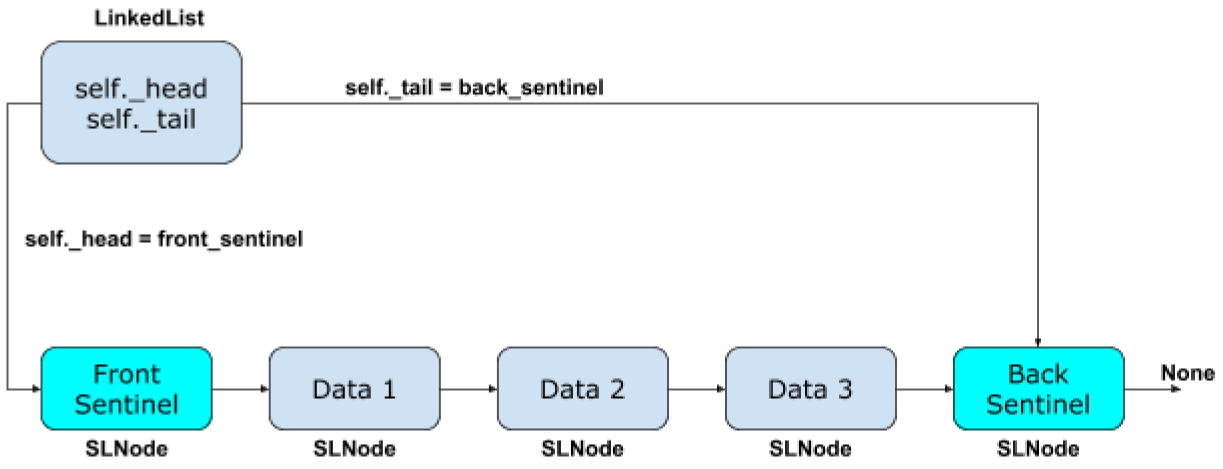
```
add_link_before()
add_front()
add_back()
remove_link()
remove_front()
remove_back()
remove()
get_front()
get_back()
is_empty()
contains()
__str__()
```

2. We will test your implementation only with just integers, floats and strings. Therefore, you do not need to implement your own __eq__, __lt__ and __gt__ methods for this assignment.

3. Number of objects stored in the list at any given time will be between 0 and 900, inclusive.

4. List must allow for duplicate values.

5. This implementation must be done with the use of front and back sentinel nodes.

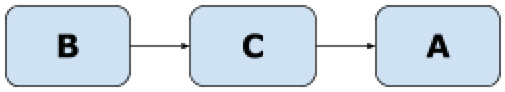


add_link_before(self, data: object, index: int) -> None:

This method adds a new node before the link at the given index. Index 0 refers to the beginning of the list. If an incorrect index position is provided (too low, too high), the method should throw an "Index out of bounds" exception.

Example of how your method may be called:

```
list = LinkedList()
list.add_link_before("A", 0)
list.add_link_before("B", 0)
list.add_link_before("C", 1)
list.add_link_before("D", 20)      # -> Exception: Index out of bounds
```

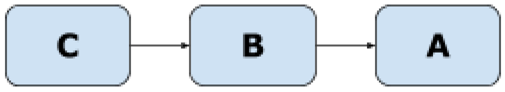


add_front(self, data: object) -> None:

This method adds a new node at the beginning of the list.

Example of how your method may be called:

```
list = LinkedList()
list.add_front("A")
list.add_front("B")
list.add_front("C")
```

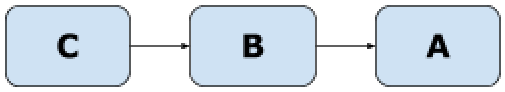


add_back(self, data: object) -> None:

This method adds a new node at the end of the list.

Example of how your method may be called:

```
list = LinkedList()
list.add_back("C")
list.add_back("B")
list.add_back("A")
```



remove_link(self, index: int) -> None:

This method removes the list node given its position in the list. Index 0 refers to the beginning of the list. If an incorrect index position is provided (too low, too high), the method should throw an “Index out of bounds” exception.

Example of how your method may be called:

```
list = LinkedList([1, 2, 3, 4, 5])
list.remove_link(0)
print(list)
list.remove_link(0)
print(list)
list.remove_link(0)
print(list)
list.remove_link(-2)              # -> Exception: Index out of bounds

Output:
[2 -> 3 -> 4 -> 5]
[3 -> 4 -> 5]
[4 -> 5]
```

remove_front(self) -> None:

This method removes the first node from the list. If the list is empty, this method should not remove anything and no exception should be raised.

Example of how your method may be called:

```
list = LinkedList([1, 2])
print(list)
list.remove_front()
list.remove_front()
list.remove_front()
print(list)

Output:
[1 -> 2]
[]
```

remove_back(self) -> None:

This method removes the last node from the list. Behavior is similar to remove_front().

Example of how your method may be called:

```
list = LinkedList()
list.remove_back()
list.add_front("Z")
list.remove_back()
print(list)

Output:
[]
```

remove(self, value: object) -> None:

This method traverses the list from the beginning to the end and removes the first node in the list whose "value" object matches the value to be removed. If there is no node with a matching value in the list, this method should not remove anything and no exception should be raised.

Example of how your method may be called:

```
list = LinkedList([1, 3, 3, 4, 3])
list.remove(3)
print(list)
list.remove(20)
list.remove(3)
list.remove(-2)
print(list)
list.remove(1)
print(list)

Output:
[1 -> 3 -> 4 -> 3]
[1 -> 4 -> 3]
[4 -> 3]
```

get_front(self) -> object:

This method returns data from the first node in the list without removing it.

Example of how your method may be called:

```
list = LinkedList([1, 2])
print(list.get_front())
list.remove_front()
print(list.get_front())
list.remove_back()
print(list.get_front())
```

```
Output:
1
2
None
```

get_back(self) -> object:

This method returns data from the first node in the list without removing it.

Example of how your method may be called:

```
list = LinkedList([1, 2, 3])
list.add_back(4)
print(list.get_back())
list.remove_back()
print(list)
print(list.get_back())
```

```
Output:
4
[1 -> 2 -> 3]
3
```

is_empty(self) -> bool:

This method returns True if the list has no data nodes. Otherwise, it returns False.

Example of how your method may be called:

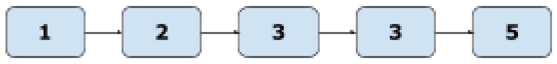
```
list = LinkedList([1, 2])
print(list.is_empty())
list.remove_front()
list.remove_back()
print(list.is_empty())

Output:
False
True
```

contains(self, value: object) -> bool:

This method traverses the list from the beginning to the end and returns True when it finds the first node that matches the "value" object. If no matching nodes are in the list, this method returns False. Empty list does not contain any elements.

Example of how your method may be called:

```
list = LinkedList([1, 2, 3, 3, 5])
```



```
print(list.contains(20))
print(list.contains(3))
list.remove(3)
print(list.contains(5))
list.remove(3)
print(list.contains(3))

Output:
False
True
True
False
```

__str__(self) -> str:

This method returns a human readable string of the list content. Format of the return string is [data1 -> data2 -> data3]. This method has already been implemented for you, no need to change it. You can use it for debugging and testing purposes.

Also, if you have trouble getting started on this assignment, review how __str__ method is implemented in the skeleton code. It will give you some ideas how to traverse a linked list in a forward direction using a while loop. It will also show how to use a sentinel node as a loop exit condition.

Example of how your method may be called:

```
list = LinkedList()
print(list)
list.add_back(10)
print(list)
list.add_front(20)
print(list)
list.add_link_before(40, 1)
print(list)
```

Output:

```
[10]
[20 -> 10]
[20 -> 40 -> 10]
```

Part 2 - Summary and Specific Instructions

1. Implement a Circular Doubly Linked List class with Deque and Bag ADT interfaces by completing provided skeleton code. Once completed, your implementation will include the following methods:

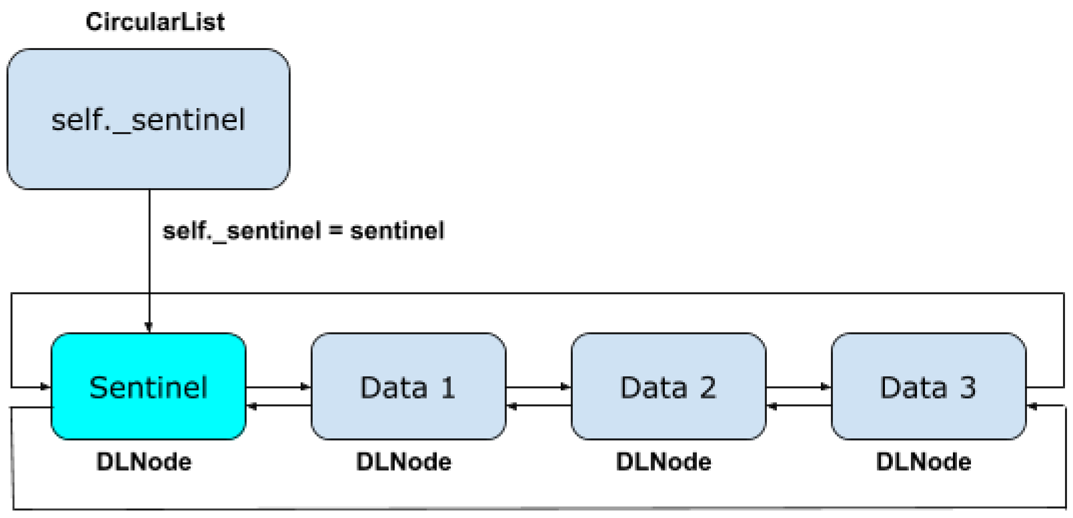
```
add_link_before()
add_front()
add_back()
remove_link()
remove_front()
remove_back()
remove()
get_front()
get_back()
is_empty()
contains()
__str__()
circularListReverse()
```

2. We will test your implementation only with just integers, floats and strings. Therefore, you do not need to implement your own __eq__, __lt__ and __gt__ methods for this assignment.

3. Number of objects stored in the list at any given time will be between 0 and 900, inclusive.

4. List must allow for duplicate values.

5. This implementation must be done with the use of a single sentinel node.



6. All methods except circularListReverse() must work the same way as methods you have implemented for a Singly Linked List in Part 1 of this assignment. Please refer to part 1 of this document for the detailed description of each method and its expected input and output values.

7. When using code examples provided in part 1 of this document, your list must be created from class CircularList, not from LinkedList. Also, note that __str__ method uses different characters for separating nodes (to reflect the fact that this is a Doubly Linked List).

   For example, if a code sample from part 1 says:

   ```
   list = LinkedList([1, 2, 3])
   list.add_back(4)
   print(list)

   Output:
   [1 -> 2 -> 3 -> 4]
   ```

   Then for purposes of part 2 you would change it as follows (also note a slightly different output format):

   ```
   list = CircularList([1, 2, 3])
   list.add_back(4)
   print(list)

   Output:
   [1 <-> 2 <-> 3 <-> 4]
   ```

8. Methods get_front(), get_back(), add_front(), add_back(), remove_front() and remove_back() must be implemented so they have O(1) runtime complexity.

9. If you have troubles getting started on this assignment, review how __str__ method is implemented in the skeleton code. It will give you some ideas how to traverse a doubly linked list in a forward direction using a while loop. It will also show how to use a sentinel node as a loop exit condition.

circularListReverse(self) -> None:

This method reverses the order of the nodes in the list. The reversal must be done "in place" without creating any copies of existing nodes or an entire existing list. Additionally, printing the list in reverse order is not an acceptable solution.

Example of how your method may be called:

```
list = CircularList([1, 2, 3, 3, 4, 5])
print(list)
list.circularListReverse()
print(list)

Output:
[1 <-> 2 <-> 3 <-> 4 <-> 5]
[5 <-> 4 <-> 3 <-> 2 <-> 1]
```

Another example:

```
list = CircularList()
print(list)
list.circularListReverse()
print(list)
list.add_back(2)
list.add_back(3)
list.add_front(1)
list.circularListReverse()
print(list)

Output:
[]
[]
[3 <-> 2 <-> 1]
```