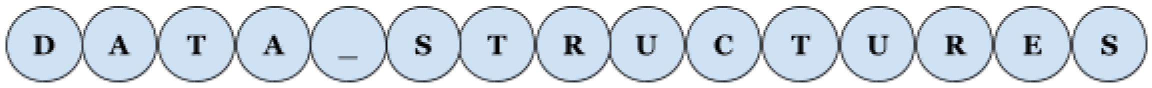
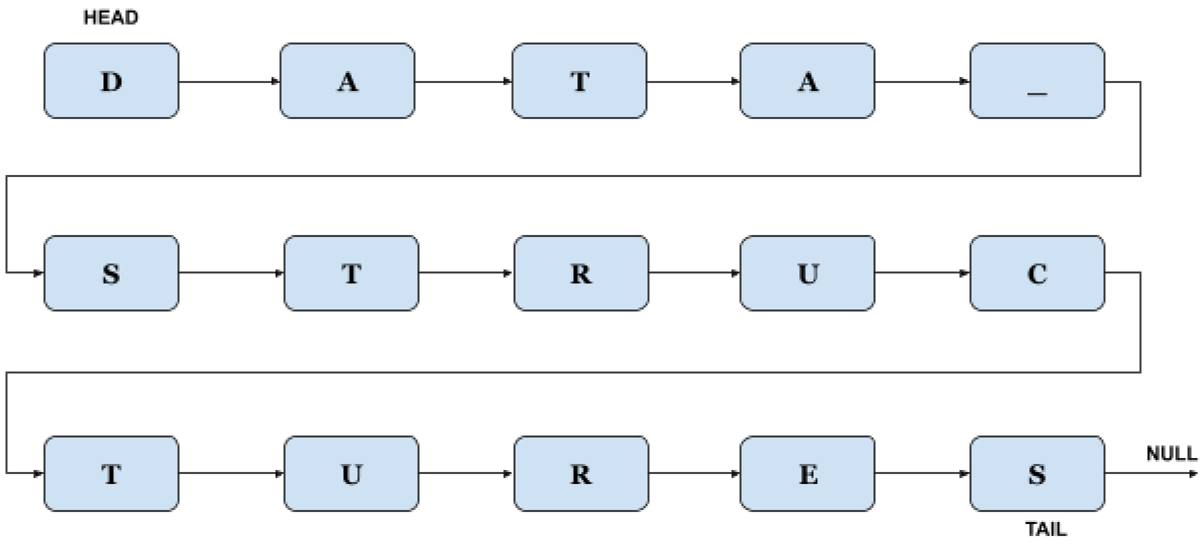
CS261 Data Structures

Assignment 3

v 1.09 (revised 4/28/2021)

Your Very Own Linked List and Binary Search Practice





Contents

General Instructions 3

Part 1 - Singly Linked List

Summary and Specific Instructions 4
add_front() 5
add_back() 5
insert_at_index() 6
remove_front() 7
remove_back() 8
remove_at_index() 9
get_front() 10
get_back() 10
remove() 11
count() 11
slice() 12

Part 2 - Circular Doubly Linked List

Summary and Specific Instructions 13
Solution Performance Checker 15
swap_pairs() 16
reverse() 17
sort() 19
rotate() 20
remove_duplicates() 21
odd_even() 22
add_integer() 23

Part 3 - Binary Search Practice

Summary and Specific Instructions 24
binary_search() 25
binary_search_rotated() 27

General Instructions

1. Programs in this assignment must be written in Python v3 and submitted to Gradescope before the due date specified in the syllabus. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

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. We encourage you to create your own test programs and cases 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.

4. Your code must have an appropriate level of comments. At a minimum, each method should have a descriptive docstring. Additionally, put comments throughout the code to make it easy to follow and understand.

5. You will be provided with a starter "skeleton" code, on which you will build your implementation. Methods defined in the skeleton code must retain their names and input / output parameters. Variables defined in the 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 helper methods and variables as needed.

   However, certains classes and methods cannot be changed in any way. Please see the comments in the skeleton code for guidance. In particular, the content of any methods pre-written for you as part of the skeleton code must not be changed.

6. Both the skeleton code and the code examples provided in this document are part of assignment requirements. They have been carefully selected to demonstrate requirements for each method. Refer to them for a detailed description of expected method behavior, input / output parameters, and the handling of edge cases. Code examples may include assignment requirements not explicitly stated elsewhere.

7. For each method in Part II, you have to implement an iterative solution **(except for Part I of the assignment, where a recursive solution is required)**. 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.

Part 1 - Summary and Specific Instructions

1. Implement Deque and Bag ADT interfaces with a Singly Linked List data structure by completing the skeleton code provided in the file `sll.py`. Once completed, your implementation will include the following methods:

```
add_front(), add_back()
insert_at_index()
remove_front(), remove_back()
remove_at_index()
get_front(), get_back()
remove()
count()
slice()
```

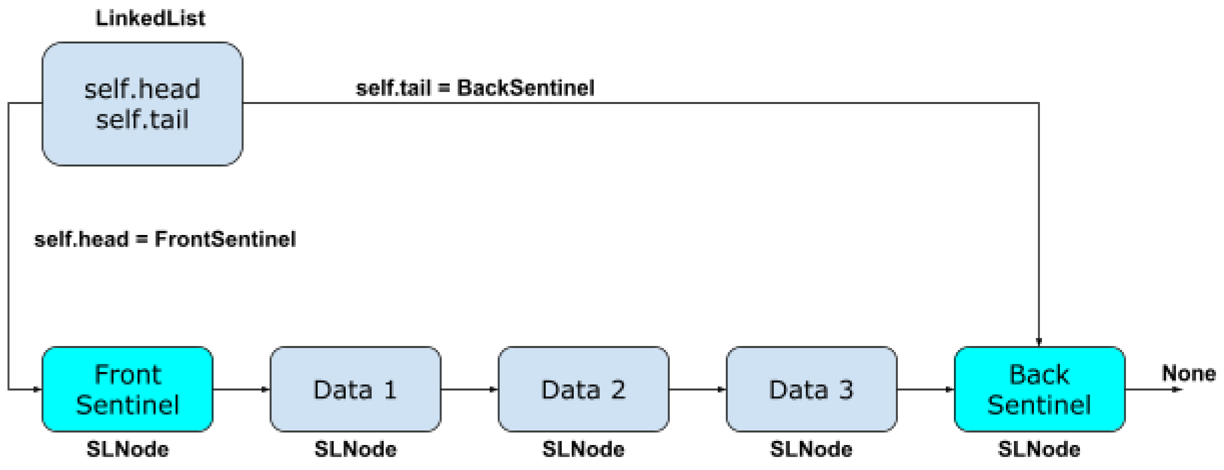
2. We will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have correct implementations of methods __eq__, __lt__, __gt__, __ge__, __le__, and __str__.

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

4. Variables in the SLNode and LinkedList classes are not marked as private. For this portion of the assignment, you are allowed to access and change their values directly. You are not required to write getter or setter methods for them.

5. RESTRICTIONS: You are not allowed to use ANY built-in Python data structures or their methods. **All implementations in this part of the assignment must be done recursively. You are not allowed to use while- or for- loops in this part of the assignment.**

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



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

This method adds a new node at the beginning of the list (right after the front sentinel).

Example #1:

```
lst = LinkedList()
print(lst)
lst.add_front('A')
lst.add_front('B')
lst.add_front('C')
print(lst)
```

Output:

```
SLL[]
SLL[C -> B -> A]
```

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

This method adds a new node at the end of the list (right before the back sentinel).

Example #1:

```
lst = LinkedList()
print(lst)
lst.add_back('C')
lst.add_back('B')
lst.add_back('A')
print(lst)
```

Output:

```
SLL[]
SLL[C -> B -> A]
```

insert_at_index(self, index: int, value: object) -> None:

This method adds a new value at the specified index position in the linked list. Index 0 refers to the beginning of the list (right after the front sentinel).

If the provided index is invalid, the method raises a custom "SLLException". Code for the exception is provided in the skeleton file. If the linked list contains N nodes (not including sentinel nodes in this count), valid indices for this method are [0, N] inclusive.

Example #1:

```
lst = LinkedList()
test_cases = [(0, 'A'), (0, 'B'), (1, 'C'), (3, 'D'), (-1, 'E'), (5, 'F')]
for index, value in test_cases:
    print('Insert of', value, 'at', index, ': ', end='')
    try:
        lst.insert_at_index(index, value)
        print(lst)
    except Exception as e:
        print(type(e))
```

Output:

```
Insert of A at 0 : SLL [A]
Insert of B at 0 : SLL [B -> A]
Insert of C at 1 : SLL [B -> C -> A]
Insert of D at 3 : SLL [B -> C -> A -> D]
Insert of E at -1 : <class '__main__.SLLException'>
Insert of F at 5 : <class '__main__.SLLException'>
```

remove_front(self) -> None:

This method removes the first node from the list. If the list is empty, the method raises a custom “SLLException”. Code for the exception is provided in the skeleton file.

Example #1:
```
lst = LinkedList([1, 2])
print(lst)
for i in range(3):
    try:
        lst.remove_front()
        print('Successful removal', lst)
    except Exception as e:
        print(type(e))
```

Output:
```
SLL[1 -> 2]
Successful removal SLL[2]
Successful removal SLL[]
<class '__main__.SLLException'>
```

remove_back(self) -> None:

This method removes the last node from the list. If the list is empty, the method raises a custom “SLLException”. Code for the exception is provided in the skeleton file.

Example #1:

```
lst = LinkedList()
try:
    lst.remove_back()
except Exception as e:
    print(type(e))
lst.add_front('Z')
lst.remove_back()
print(lst)
lst.add_front('Y')
lst.add_back('Z')
lst.add_front('X')
print(lst)
lst.remove_back()
print(lst)
```

Output:

```
<class '__main__.SLLException'>
SLL []
SLL [X -> Y -> Z]
SLL [X -> Y]
```

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

This method removes a node from the list given its index. Index 0 refers to the beginning of the list (right after the front sentinel.

If the provided index is invalid, the method raises a custom “SLLException”. Code for the exception is provided in the skeleton file. If the list contains N elements (not including sentinel nodes in this count), valid indices for this method are [0, N - 1] inclusive.

Example #1:

```
lst = LinkedList([1, 2, 3, 4, 5, 6])
print(lst)
for index in [0, 0, 0, 2, 2, -2]:
    print('Removed at index:', index, ': ', end='')
    try:
        lst.remove_at_index(index)
        print(lst)
    except Exception as e:
        print(type(e))
print(lst)
```

Output:

```
SLL [1 -> 2 -> 3 -> 4 -> 5 -> 6]
Removed at index: 0 : SLL [2 -> 3 -> 4 -> 5 -> 6]
Removed at index: 0 : SLL [3 -> 4 -> 5 -> 6]
Removed at index: 0 : SLL [4 -> 5 -> 6]
Removed at index: 2 : SLL [4 -> 5]
Removed at index: 2 : <class '__main__.SLLException'>
Removed at index: -2 : <class '__main__.SLLException'>
SLL [4 -> 5]
```

get_front(self) -> object:

This method returns the value from the first node in the list without removing it. If the list is empty, the method raises a custom "SLLException". Code for the exception is provided in the skeleton file.

Example #1:
```
lst = LinkedList(['A', 'B'])
print(lst.get_front())
print(lst.get_front())
lst.remove_front()
print(lst.get_front())
lst.remove_back()
try:
    print(lst.get_front())
except Exception as e:
    print(type(e))
```

Output:
```
A
A
B
<class '__main__.SLLException'>
```

get_back(self) -> object:

This method returns the value from the last node in the list without removing it. If the list is empty, the method raises a custom "SLLException". Code for the exception is provided in the skeleton file.

Example #1:
```
lst = LinkedList([1, 2, 3])
lst.add_back(4)
print(lst.get_back())
lst.remove_back()
print(lst)
print(lst.get_back())
```

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

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

This method traverses the list from the beginning to the end and removes the first node in the list that matches the provided "value" object. The method returns True if some node was actually removed from the list. Otherwise, it returns False.

Example #1:
```
lst = LinkedList([1, 2, 3, 1, 2, 3, 1, 2, 3])
print(lst)
for value in [7, 3, 3, 3, 3]:
    print(lst.remove(value), lst.length(), lst)
```

Output:
```
SLL [1 -> 2 -> 3 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
False 9 SLL [1 -> 2 -> 3 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
True 8 SLL [1 -> 2 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
True 7 SLL [1 -> 2 -> 1 -> 2 -> 1 -> 2 -> 3]
True 6 SLL [1 -> 2 -> 1 -> 2 -> 1 -> 2]
False 6 SLL [1 -> 2 -> 1 -> 2 -> 1 -> 2]
```

count(self, value: object) -> int:

This method counts the number of elements in the list that match the provided "value" object.

Example #1:
```
lst = LinkedList([1, 2, 3, 1, 2, 2])
print(lst, lst.count(1), lst.count(2), lst.count(3), lst.count(4))
```

Output:
```
SLL [1 -> 2 -> 3 -> 1 -> 2 -> 2] 2 3 1 0
```

slice(self, start_index: int, size: int) -> object:

This method returns a new LinkedList object that contains the requested number of nodes from the original list starting with the node located at the requested start index. If the original list contains N nodes, a valid `start_index` is in range [0, N - 1] inclusive. Runtime complexity of your implementation must be O(N).

If the provided start index is invalid or if there are not enough nodes between the start index and the end of the list to make a slice of the requested size, this method raises a custom "SLLException". Code for the exception is provided in the skeleton file.

Example #1:

```
lst = LinkedList([1, 2, 3, 4, 5, 6, 7, 8, 9])
ll_slice = lst.slice(1, 3)
print(lst, ll_slice, sep="\n")
ll_slice.remove_at_index(0)
print(lst, ll_slice, sep="\n")
```

Output:

```
SLL [1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9]
SLL [2 -> 3 -> 4]
SLL [1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9]
SLL [3 -> 4]
```

Example #2:

```
lst = LinkedList([10, 11, 12, 13, 14, 15, 16])
print("SOURCE:", lst)
slices = [(0, 7), (-1, 7), (0, 8), (2, 3), (5, 0), (5, 3), (6, 1)]
for index, size in slices:
    print("Slice", index, "/", size, end="")
    try:
        print(" --- OK: ", lst.slice(index, size))
    except:
        print(" --- exception occurred.")
```

Output:

```
SOURCE: SLL [10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16]
Slice 0 / 7 --- OK:  SLL [10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16]
Slice -1 / 7 --- exception occurred.
Slice 0 / 8 --- exception occurred.
Slice 2 / 3 --- OK:  SLL [12 -> 13 -> 14]
Slice 5 / 0 --- OK:  SLL []
Slice 5 / 3 --- exception occurred.
Slice 6 / 1 --- OK:  SLL [16]
```

Part 2 - Summary and Specific Instructions

1. Implement Deque and Bag ADT interfaces with a circular doubly linked list data structure by completing the skeleton code provided in the file `cdll.py`. Once completed, your implementation will include the following methods:

   Methods identical to Singly Linked List in Part 1:
   ```
   add_front(), add_back()
   insert_at_index()
   remove_front(), remove_back()
   remove_at_index()
   get_front(), get_back()
   remove()
   count()
   ```

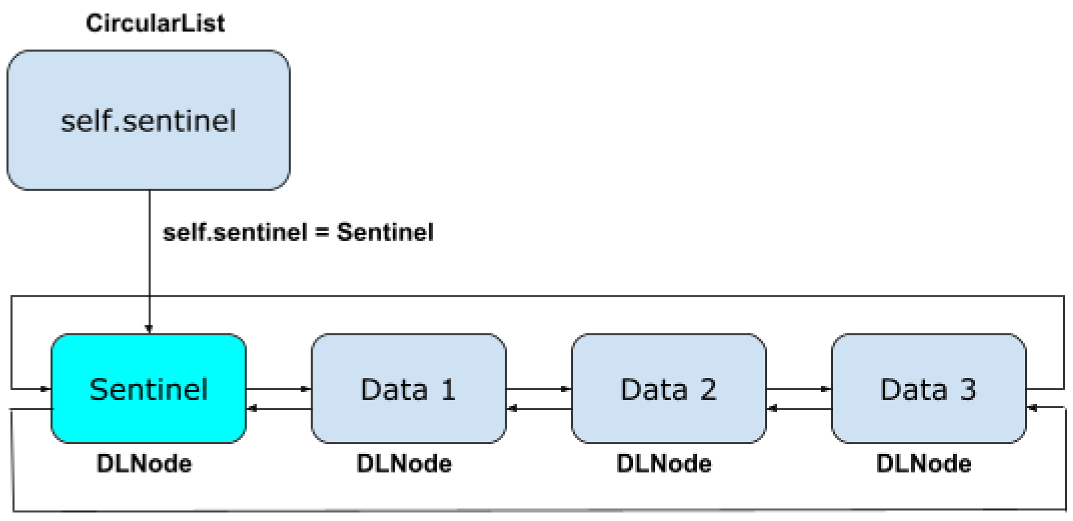
   * Note that you do not need to implement the slice() method.

   Additional methods:
   ```
   swap_pairs(), reverse(), sort()
   rotate(), remove_duplicates()
   odd_even(), add_integer()
   ```

2. We will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have correct implementations of methods __eq__, __lt__, __gt__, __ge__, __le__, and __str__.

3. The number of objects stored in the list at any given time will be between 0 and 1,000,000 inclusive.

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



5. Variables in the DLNode and CircularList classes are not marked as private. For this portion of the assignment, you are allowed to access and change their values directly. You are not required to write getter or setter methods for them.

6. RESTRICTIONS: You are not allowed to use ANY built-in Python data structures or their methods. **For this portion of the assignment, you must implement an iterative solution (no recursive calls).**

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

8. All methods that have identical names to methods in Part 1 should accept the same input and produce the same output as the methods you implemented for a Singly Linked List in Part 1 of this assignment. Please refer to Part 1 of this document for a detailed description of each method and its expected input and output values.

   Note that, in the case where an exception needs to be raised, the name of the exception should be CDLLException, not SLLException. Code for the exception is provided in the skeleton file.

9. When using code examples provided in Part 1 of this document, your list must be created from the class CircularList, not from the class LinkedList. Also, note that the __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:

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

   Output:

   ```
   SLL [1 -> 2 -> 3 -> 4]
   ```

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

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

   Output:

   ```
   CDLL [1 <-> 2 <-> 3 <-> 4]
   ```

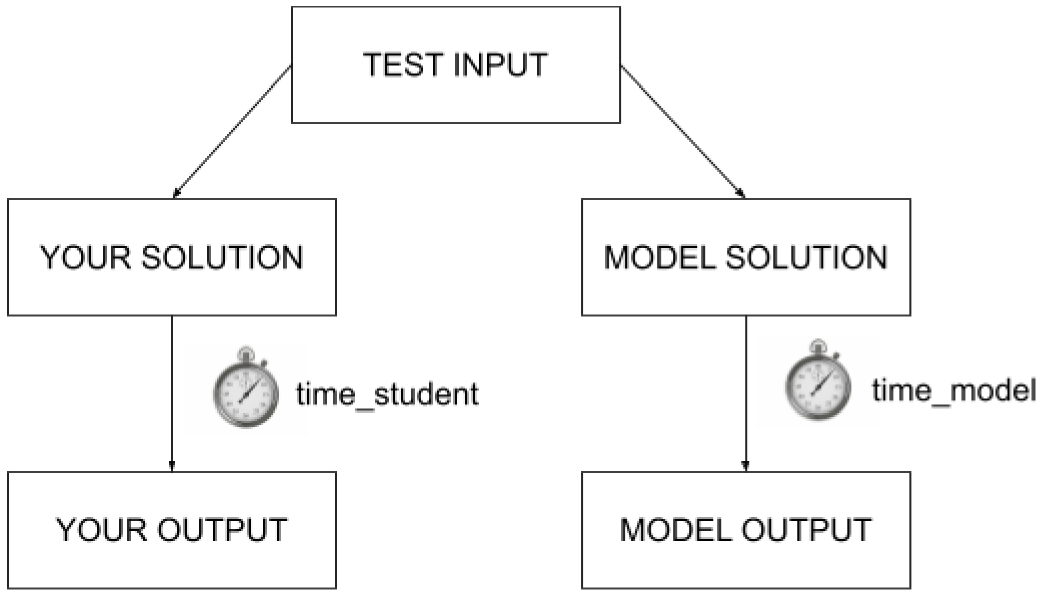
Solution Performance Checker

To earn full points for some methods in this portion of the assignment, you will have to implement a solution that is BOTH correct and efficient. Correct means the solution produces correct output for any given input. Efficient means it does so with some acceptable speed.

Gradescope will measure the efficiency of your solution by taking the same test input and feeding it to both your solution and the model solution. Then Gradescope will measure the time it took both solutions to return the correct output. Execution time of your solution must not be longer than the execution time of the model solution, multiplied by a certain allowance factor.

For example, if the model solution takes 1 second to solve a particular problem and the allowance factor assigned to that problem is 3, then your solution must produce correct output for the same problem within 3 seconds.

The allowance factor varies for each method in this assignment. Please see the description of the methods for more details.



```
if time_student <= time_model * allowance_factor:
    Pass
else:
    Fail
```

swap_pairs(self, index1: int, index2: int) -> None:

This method swaps two nodes given their indices. All work must be done "in place" without creating any new nodes. You are not allowed to change the values of the nodes; the solution must change node pointers. Your solution must have O(N) runtime complexity for finding the pointers to the nodes and O(1) for actually swapping them.

If either of the provided indices is invalid, the method raises a custom "CDLLException". Code for the exception is provided in the skeleton file. If the linked list contains N nodes (not including sentinel nodes in this count), valid indices for this method are [0, N - 1] inclusive.

Solution Performance Checker: The allowance factor for this problem is 4.

Example #1:
```
lst = CircularList([0, 1, 2, 3, 4, 5, 6])
test_cases = ((0, 6), (0, 7), (-1, 6), (1, 5),
              (4, 2), (3, 3), (1, 2), (2, 1))

for i, j in test_cases:
    print('Swap nodes ', i, j, ' ', end='')
    try:
        lst.swap_pairs(i, j)
        print(lst)
    except Exception as e:
        print(type(e))
```

Output:
```
Swap nodes  0 6  CDLL [6 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 0]
Swap nodes  0 7  <class '__main__.CDLLException'>
Swap nodes  -1 6  <class '__main__.CDLLException'>
Swap nodes  1 5  CDLL [6 <-> 5 <-> 2 <-> 3 <-> 4 <-> 1 <-> 0]
Swap nodes  4 2  CDLL [6 <-> 5 <-> 4 <-> 3 <-> 2 <-> 1 <-> 0]
Swap nodes  3 3  CDLL [6 <-> 5 <-> 4 <-> 3 <-> 2 <-> 1 <-> 0]
Swap nodes  1 2  CDLL [6 <-> 4 <-> 5 <-> 3 <-> 2 <-> 1 <-> 0]
Swap nodes  2 1  CDLL [6 <-> 5 <-> 4 <-> 3 <-> 2 <-> 1 <-> 0]
```

reverse(self) -> None:

This method reverses the order of the nodes in the list. All work must be done “in place” without creating any new nodes. You are not allowed to change the values of the nodes; the solution must change node pointers. Your solution must have O(N) runtime complexity.

Solution Performance Checker: The allowance factor for this problem is 10.

Example #1:
```
test_cases = (
    [1, 2, 3, 3, 4, 5],
    [1, 2, 3, 4, 5],
    ['A', 'B', 'C', 'D']
)
for case in test_cases:
    lst = CircularList(case)
    lst.reverse()
    print(lst)
```

Output:
```
CDLL [5 <-> 4 <-> 3 <-> 3 <-> 2 <-> 1]
CDLL [5 <-> 4 <-> 3 <-> 2 <-> 1]
CDLL [D <-> C <-> B <-> A]
```

Example #2:
```
lst = CircularList()
print(lst)
lst.reverse()
print(lst)
lst.add_back(2)
lst.add_back(3)
lst.add_front(1)
lst.reverse()
print(lst)
```

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

Example #3:

```
class Student:
    def __init__(self, name, age):
        self.name, self.age = name, age

    def __eq__(self, other):
        return self.age == other.age

    def __str__(self):
        return str(self.name) + ' ' + str(self.age)

s1, s2 = Student('John', 20), Student('Andy', 20)
lst = CircularList([s1, s2])
print(lst)
lst.reverse()
print(lst)
print(s1 == s2)
```

Output:

```
CDLL [John 20 <-> Andy 20]
CDLL [Andy 20 <-> John 20]
True
```

Example #4:

```
lst = CircularList([1, 'A'])
lst.reverse()
print(lst)
```

Output:

```
CDLL [A <-> 1]
```

sort(self) -> None:

This method sorts the content of the list in non-descending order. All work must be done "in place" without creating any new nodes. You are not allowed to change the values of the nodes; the solution must change node pointers.

You can implement any sort method of your choice. Sorting does not have to be very efficient or fast; a simple insertion or bubble sort will suffice. However, runtime complexity of your implementation cannot be worse than O(N^2). Duplicates in the list can be placed in any relative order in the sorted list (in other words, your sort does not have to be 'stable').

For this method, you may assume that elements stored in the linked list are all of the same type (either all numbers, or strings, or custom objects, but never a mix of these). You do not need to write checks for this condition.

Solution Performance Checker: The allowance factor for this problem is 5.

Example #1:
```
test_cases = (
    [1, 10, 2, 20, 3, 30, 4, 40, 5],
    ['zebra2', 'apple', 'tomato', 'apple', 'zebra1'],
    [(1, 1), (20, 1), (1, 20), (2, 20)]
)
for case in test_cases:
    lst = CircularList(case)
    print(lst)
    lst.sort()
    print(lst)
```

Output:
```
CDLL [1 <-> 10 <-> 2 <-> 20 <-> 3 <-> 30 <-> 4 <-> 40 <-> 5]
CDLL [1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 10 <-> 20 <-> 30 <-> 40]
CDLL [zebra2 <-> apple <-> tomato <-> apple <-> zebra1]
CDLL [apple <-> apple <-> tomato <-> zebra1 <-> zebra2]
CDLL [(1, 1) <-> (20, 1) <-> (1, 20) <-> (2, 20)]
CDLL [(1, 1) <-> (1, 20) <-> (2, 20) <-> (20, 1)]
```

rotate(self, steps: int) -> None:

This method 'rotates' the linked list by shifting the position of its elements right or left `steps` number of times. If `steps` is a positive integer, elements should be rotated right. Otherwise, the elements should be rotated left. All work must be done "in place" without creating any new nodes. You are not allowed to change the values of the nodes; the solution must change node pointers. Please note that the value of the `steps` parameter can be very large (from -10^9 to 10^9). The solution's runtime complexity must be O(N), where N is the length of the list.

Solution Performance Checker: The allowance factor for this problem is 10.

Example #1:

```
source = [_ for _ in range(-20, 20, 7)]
for steps in [1, 2, 0, -1, -2, 28, -100]:
    lst = CircularList(source)
    lst.rotate(steps); print(lst, steps)
```

Output:

```
CDLL [15 <-> -20 <-> -13 <-> -6 <-> 1 <-> 8] 1
CDLL [8 <-> 15 <-> -20 <-> -13 <-> -6 <-> 1] 2
CDLL [-20 <-> -13 <-> -6 <-> 1 <-> 8 <-> 15] 0
CDLL [-13 <-> -6 <-> 1 <-> 8 <-> 15 <-> -20] -1
CDLL [-6 <-> 1 <-> 8 <-> 15 <-> -20 <-> -13] -2
CDLL [-6 <-> 1 <-> 8 <-> 15 <-> -20 <-> -13] 28
CDLL [8 <-> 15 <-> -20 <-> -13 <-> -6 <-> 1] -100
```

Example #2:

```
lst = CircularList([10, 20, 30, 40])
for j in range(-1, 2, 2):
    for _ in range(3):
        lst.rotate(j); print(lst)
```

Output:

```
CDLL [20 <-> 30 <-> 40 <-> 10]
CDLL [30 <-> 40 <-> 10 <-> 20]
CDLL [40 <-> 10 <-> 20 <-> 30]
CDLL [30 <-> 40 <-> 10 <-> 20]
CDLL [20 <-> 30 <-> 40 <-> 10]
CDLL [10 <-> 20 <-> 30 <-> 40]
```

Example #3:

```
lst = CircularList()
lst.rotate(10)
print(lst)
```

Output:

```
CDLL []
```

remove_duplicates(self) -> None:

This method deletes all nodes that have duplicate values from a sorted linked list, leaving only nodes with distinct values. All work must be done “in place” without creating any new nodes. You are not allowed to change the values of the nodes; the solution must change node pointers. Your solution must have O(N) runtime complexity.

You may assume that the list is sorted. You do not need to write checks for this condition. You may also assume that all elements in the list are of the same type and can be compared with each other using the == operator.

Solution Performance Checker: The allowance factor for this problem is 5.

Example #1:

```
test_cases = (
    [1, 2, 3, 4, 5], [1, 1, 1, 1, 1],
    [], [1], [1, 1], [1, 1, 1, 2, 2, 2],
    [0, 1, 1, 2, 3, 3, 4, 5, 5, 6],
    list("abccd"),
    list("005BCDDEEFI")
)
for case in test_cases:
    lst = CircularList(case)
    print('INPUT :', lst)
    lst.remove_duplicates()
    print('OUTPUT:', lst)
```

Output:

```
INPUT : CDLL [1 <-> 2 <-> 3 <-> 4 <-> 5]
OUTPUT: CDLL [1 <-> 2 <-> 3 <-> 4 <-> 5]
INPUT : CDLL [1 <-> 1 <-> 1 <-> 1 <-> 1]
OUTPUT: CDLL []
INPUT : CDLL []
OUTPUT: CDLL []
INPUT : CDLL [1]
OUTPUT: CDLL [1]
INPUT : CDLL [1 <-> 1]
OUTPUT: CDLL []
INPUT : CDLL [1 <-> 1 <-> 1 <-> 2 <-> 2 <-> 2]
OUTPUT: CDLL []
INPUT : CDLL [0 <-> 1 <-> 1 <-> 2 <-> 3 <-> 3 <-> 4 <-> 5 <-> 5 <-> 6]
OUTPUT: CDLL [0 <-> 2 <-> 4 <-> 6]
INPUT : CDLL [a <-> b <-> c <-> c <-> d]
OUTPUT: CDLL [a <-> b <-> d]
INPUT : CDLL [0 <-> 0 <-> 5 <-> B <-> C <-> D <-> D <-> E <-> E <-> F <-> I]
OUTPUT: CDLL [5 <-> B <-> C <-> F <-> I]
```

odd_even(self) -> None:

This method regroups list nodes by first grouping all ODD nodes together followed by all EVEN nodes (here, "odd" and "even" refer to the node position in the list (starting from 1), not the node values). Please see the code examples below for additional details.

All work must be done "in place" without creating any new nodes. You are not allowed to change the values of the nodes; the solution must change node pointers. Your solution must have O(N) runtime complexity.

Solution Performance Checker: The allowance factor for this problem is 10.

Example #1:

```
test_cases = (
    [1, 2, 3, 4, 5], list('ABCDE'),
    [], [100], [100, 200], [100, 200, 300],
    [100, 200, 300, 400],
    [10, 'A', 20, 'B', 30, 'C', 40, 'D', 50, 'E']
)
for case in test_cases:
    lst = CircularList(case)
    print('INPUT :', lst)
    lst.odd_even()
    print('OUTPUT:', lst)
```

Output:

```
INPUT : CDLL [1 <-> 2 <-> 3 <-> 4 <-> 5]
OUTPUT: CDLL [1 <-> 3 <-> 5 <-> 2 <-> 4]
INPUT : CDLL [A <-> B <-> C <-> D <-> E]
OUTPUT: CDLL [A <-> C <-> E <-> B <-> D]
INPUT : CDLL []
OUTPUT: CDLL []
INPUT : CDLL [100]
OUTPUT: CDLL [100]
INPUT : CDLL [100 <-> 200]
OUTPUT: CDLL [100 <-> 200]
INPUT : CDLL [100 <-> 200 <-> 300]
OUTPUT: CDLL [100 <-> 300 <-> 200]
INPUT : CDLL [100 <-> 200 <-> 300 <-> 400]
OUTPUT: CDLL [100 <-> 300 <-> 200 <-> 400]
INPUT : CDLL [10 <-> A <-> 20 <-> B <-> 30 <-> C <-> 40 <-> D <-> 50 <-> E]
OUTPUT: CDLL [10 <-> 20 <-> 30 <-> 40 <-> 50 <-> A <-> B <-> C <-> D <-> E]
```

add_integer(self, num: int) -> None:

Assume the content of the linked list represents a non-negative integer such that each digit is stored in a separate node (you do not need to write checks for this condition).

This method will receive another non-negative integer `num`, add it to the number already stored in the linked list, and then store the result of the addition back into the list nodes, one digit per node. Please see the examples below for more details.

This addition must be done "in place", by changing the values of the existing nodes to the extent that is possible. However, since the result of the addition may have more digits than nodes in the original linked list, you may need to add some new nodes. However, you should only add the minimum number of nodes necessary and not recreate the entire linked list.

For example, if the original list stored the number 123 as [1 <-> 2 <-> 3], and the second integer had a value of 5, no new nodes should be created since the result of the addition can be stored in the same three nodes as [1 <-> 2 <-> 8]. If the original list was [1 <-> 2 <-> 3], and the second integer had a value of 999, you are allowed to create one new node so that the entire result can fit as shown: [1 <-> 1 <-> 2 <-> 2].

Your solution must have $O(N + \log_{10}K)$ runtime complexity, where N is the length of the linked list and K is the integer `num`.

Solution Performance Checker: The allowance factor for this problem is 20.

Example #1:
```
test_cases = (
    ([1, 2, 3], 10456), ([], 25),
    ([2, 0, 9, 0, 7], 108), ([9, 9, 9], 9_999_999))
for list_content, integer in test_cases:
    lst = CircularList(list_content)
    print('INPUT :', lst, 'INTEGER', integer)
    lst.add_integer(integer)
    print('OUTPUT:', lst)
```

Output:
```
INPUT : CDLL [1 <-> 2 <-> 3] INTEGER 10456
OUTPUT: CDLL [1 <-> 0 <-> 5 <-> 7 <-> 9]
INPUT : CDLL [] INTEGER 25
OUTPUT: CDLL [2 <-> 5]
INPUT : CDLL [2 <-> 0 <-> 9 <-> 0 <-> 7] INTEGER 108
OUTPUT: CDLL [2 <-> 1 <-> 0 <-> 1 <-> 5]
INPUT : CDLL [9 <-> 9 <-> 9] INTEGER 9999999
OUTPUT: CDLL [1 <-> 0 <-> 0 <-> 0 <-> 0 <-> 9 <-> 9 <-> 8]
```

Part 3 - Summary and Specific Instructions

There are 2 separate problems in this assignment. For each problem, you will have to write a Python function according to the provided specifications. A "skeleton" code and some basic test cases for each problem are provided in the file `binary_search.py`

RESTRICTIONS: You are NOT allowed to use ANY built-in Python data structures and/or their methods in any of your solutions. This includes built-in Python lists, dictionaries, or anything else. Variables to hold a single value or a tuple holding two/three values are allowed. It is OK to use built-in Python generator functions like `range()`.

You are NOT allowed to directly access any variables of the StaticArray class (like `self._size` or `self._data`). All work must be done using only StaticArray class methods.

binary_search(arr: StaticArray, target: int) -> int:

Write a function that receives a StaticArray and an integer `target` and returns the index of the `target` element if it is present in the array or returns `-1` if it is not. The original array should not be modified.

You may assume that the input array will contain at least one element and that all elements will be integers in the range [-10^9, 10^9]. It is guaranteed that all elements in the input array will be distinct and that the input array will be sorted **in either ascending or descending order**.

Your binary search implementation must have O(logN) runtime complexity.

Example #1:

```
src = (-10, -5, 0, 5, 7, 9, 11)
targets = (7, -10, 11, 0, 8, 1, -100, 100)
arr = StaticArray(len(src))
for i, value in enumerate(src):
    arr[i] = value
print([binary_search(arr, target) for target in targets])
arr._data.reverse()
print([binary_search(arr, target) for target in targets])
```

Output:

```
[4, 0, 6, 2, -1, -1, -1, -1]
[2, 6, 0, 4, -1, -1, -1, -1]
```

Example #2:

```
src = [random.randint(-10 ** 7, 10 ** 7) for _ in range(5_000_000)]
src = sorted(set(src))
arr = StaticArray(len(src))
arr._data = src[:]

# add 20 valid and 20 (likely) invalid targets
targets = [-10 ** 8, 10 ** 8]
targets += [arr[random.randint(0, len(src) - 1)] for _ in range(20)]
targets += [random.randint(-10 ** 7, 10 ** 7) for _ in range(18)]

result, total_time = True, 0
for target in targets:
    total_time -= time.time()
    answer = binary_search(arr, target)
    total_time += time.time()
    result &= arr[answer] == target if target in src else answer == -1
print(result, total_time < 0.5)

arr._data.reverse()
for target in targets:
    total_time -= time.time()
    answer = binary_search(arr, target)
    total_time += time.time()
    result &= arr[answer] == target if target in src else answer == -1
print(result, total_time < 0.5)
```

Output:

```
True True
True True
```

binary_search_rotated(arr: StaticArray, target: int) -> int:

Write a function that receives a StaticArray and an integer `target` and returns the index of the `target` element if it is present in the array or returns `-1` if it is not. The original array should not be modified.

You may assume that the input array will contain at least one element and that all elements will be integers in the range [-10^9, 10^9]. It is guaranteed that all elements in the input array will be distinct.

The input array will be sorted in the ascending order. But, before being passed to your function, **the input array will be rotated an unknown number of steps (either right or left).**

Your binary search implementation must have O(logN) runtime complexity.

Example #1:
```
test_cases = (
    ((6, 8, 12, 20, 0, 2, 5), 0),
    ((6, 8, 12, 20, 0, 2, 5), -1),
    ((1,), 1),
    ((1,), 0),
)
result = []
for src, target in test_cases:
    arr = StaticArray(len(src))
    for i, value in enumerate(src):
        arr[i] = value
    result.append((binary_search_rotated(arr, target)))
print(*result)
```

Output:
```
4 -1 0 -1
```

Example #2:

```
src = [random.randint(-10 ** 7, 10 ** 7) for _ in range(5_000_000)]
src = sorted(set(src))
arr = StaticArray(len(src))
arr._data = src[:]

# add 20 valid and 20 (likely) invalid targets
targets = [-10 ** 8, 10 ** 8]
targets += [arr[random.randint(0, len(src) - 1)] for _ in range(20)]
targets += [random.randint(-10 ** 7, 10 ** 7) for _ in range(18)]

result, total_time = True, 0
for target in targets:
    # rotate arr random number of steps
    pivot = random.randint(0, len(src) - 1)
    arr._data = src[pivot:] + src[:pivot]

    total_time -= time.time()
    answer = binary_search_rotated(arr, target)
    total_time += time.time()
    result &= arr[answer] == target if target in src else answer == -1
print(result, total_time < 0.5)
```

Output:

```
True True
```