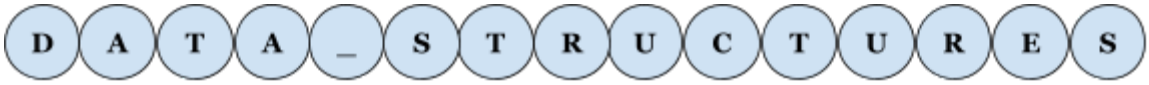
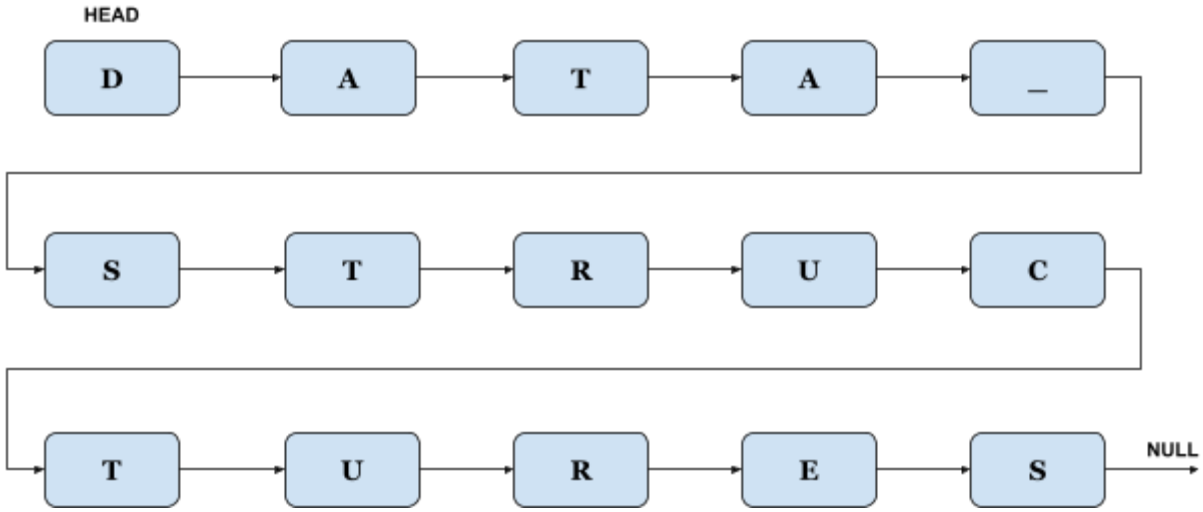
CS261 Data Structures

Assignment 3

Fall 2023

Linked List and ADT Implementation





Contents

General Instructions

1. Programs in this assignment must be written in Python 3 and submitted to Gradescope before the due date specified on Canvas and in the Course Schedule. You may resubmit your code as many times as necessary before the due date (this is encouraged). Gradescope allows you to choose which submission will be graded. **We will grade the currently activated submission.**

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. You earn points as you pass tests. There are no hidden Gradescope tests and you will know your maximum possible score with each submission.

3. We encourage you to create your own test cases (such as with Python's **unittest** unit testing framework, which is more powerful and efficient than "print statement debugging"), even though this work doesn't have to be submitted and won't be graded. Gradescope tests are limited in scope and may not cover all edge cases. We reserve the right to evaluate your submission with additional methods beyond Gradescope.

4. Unless indicated otherwise, we will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have the correct implementation of:

    a. "rich comparison" methods:

        i. `__eq__()`

        ii. `__lt__()`

        iii. `__gt__()`

        iv. `__ge__()`

        v. `__le__()`

    b. `__str__()`

5. **Your code must have an appropriate level of comments.** At minimum, each method must have a descriptive docstring. Additionally, write comments throughout your code to make it easy to follow and understand any non-obvious code. However, be mindful of the amount of comments in your code. Cluttering comments negatively impact readability and are discouraged. If your code includes an excessive amount of comments, consider refactoring. Please refer to "Styling Your Code" from the "Coding Guides and Tips - Style and Debugging" module for specifics.

6. 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. Custom exceptions are defined in the skeleton code to be used as instructed**.** 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 are allowed to:

   - add more helper methods and variables, as needed
   - add optional default parameters to method definitions
   - modify or add to the basic testing section within the scope of:

     ```
     if __name__ == "__main__":
     ```

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

   All points will be deducted from methods with the incorrect time complexity.

7. The skeleton code and 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 detailed descriptions of expected method behavior, input/output parameters, and the handling of edge cases. Code examples may include assignment requirements not explicitly stated elsewhere.

8. For each method, **you are required to use an iterative solution.** Recursion is not permitted.

9. **You may not use any imports** beyond the ones included in the assignment source code.

Part 1 - Summary and Specific Instructions

1. Implement 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:

```
insert_front(), insert_back()
insert_at_index(), remove_at_index()
remove()
count()
find()
slice()
```

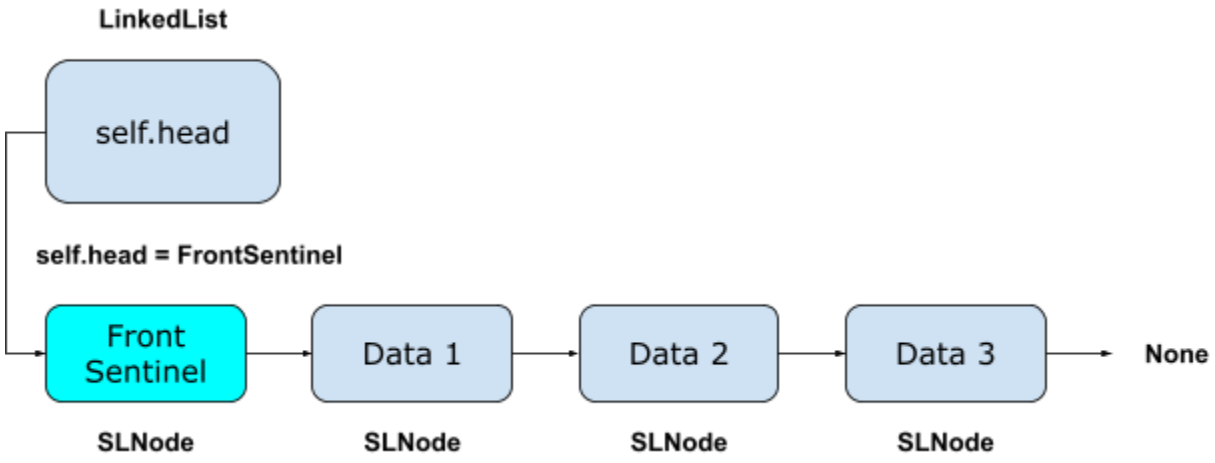
2. The number of objects stored in the list at any given time will be between 0 and 900 inclusive.

3. Make sure that you **include the provided SLNode class in your project**. You do **NOT** upload this class to Gradescope.

4. The variable in the LinkedList class (_head) is marked as **private**, so it may only be accessed/changed directly inside the class. Variables in the SLNode class (provided in its own file, as it will also be used in Parts 4 and 5) are not marked as private. You are allowed to access/change their values directly wherever the SLNode class is used, and 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. Your solutions should not call double underscore ("dunder") methods.

6. This implementation must be done with the use of a **front sentinel** node.



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

This method adds a new node at the beginning of the list (right after the front sentinel). It must be implemented with **O(1)** runtime complexity.

Example #1:

```
test_case = ["A", "B", "C"]
lst = LinkedList()
for case in test_case:
    lst.insert_front(case)
    print(lst)
```

Output:

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

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

This method adds a new node at the end of the list. It must be implemented with **O(N)** runtime complexity.

Example #1:

```
test_case = ["C", "B", "A"]
lst = LinkedList()
for case in test_case:
    lst.insert_back(case)
    print(lst)
```

Output:

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

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

This method inserts 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 must raise a custom **"SLLException"**. If the linked list contains N nodes (the sentinel node is not included in this count), valid indices for this method are [0, N] inclusive. It must be implemented with **O(N)** runtime complexity.

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("Inserted", value, "at index", index, ": ", end="")
        try:
            lst.insert_at_index(index, value)
            print(lst)
        except Exception as e:
            print(type(e))
```

Output:

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

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

This method removes the node at the specified index position from 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 must raise a custom **"SLLException"**. If the list contains N elements (the sentinel node is not included in this count), valid indices for this method are [0, N - 1] inclusive. It must be implemented with **O(N)** runtime complexity.

Example #1:

```
lst = LinkedList([1, 2, 3, 4, 5, 6])
print(f"Initial LinkedList : {lst}")
for index in [0, 2, 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:

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

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

This method traverses the list from the beginning to the end, and removes the first node that matches the provided value. The method returns True if a node was removed from the list. Otherwise, it returns False. It must be implemented with **O(N)** runtime complexity.

Example #1:

```
lst = LinkedList([1, 2, 3, 1, 2, 3, 1, 2, 3])
print(f"Initial LinkedList, Length: {lst.length()}\n  {lst}")
for value in [7, 3, 3, 3, 3]:
    print(f"remove({value}): {lst.remove(value}, Length: {lst.length()}"
          f"\n {lst}")
```

Output:

```
Initial LinkedList, Length: 9
  SLL [1 -> 2 -> 3 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
remove(7): False, Length: 9
  SLL [1 -> 2 -> 3 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
remove(3): True, Length: 8
  SLL [1 -> 2 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
remove(3): True, Length: 7
  SLL [1 -> 2 -> 1 -> 2 -> 1 -> 2 -> 3]
remove(3): True, Length: 6
  SLL [1 -> 2 -> 1 -> 2 -> 1 -> 2]
remove(3): False, Length: 6
  SLL [1 -> 2 -> 1 -> 2 -> 1 -> 2]
```

Example #2:

```
lst = LinkedList([1, 2, 3, 1, 2, 3, 1, 2, 3])
print(f"Initial LinkedList, Length: {lst.length()}\n  {lst}")
for value in [1, 2, 3, 1, 2, 3, 3, 2, 1]:
    print(f"remove({value}): {lst.remove(value), Length: {lst.length()}"
          f"\n {lst}")
```

Output:

```
Initial LinkedList, Length: 9
  SLL [1 -> 2 -> 3 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
remove(1): True, Length: 8
  SLL [2 -> 3 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
remove(2): True, Length: 7
  SLL [3 -> 1 -> 2 -> 3 -> 1 -> 2 -> 3]
remove(3): True, Length: 6
  SLL [1 -> 2 -> 3 -> 1 -> 2 -> 3]
remove(1): True, Length: 5
  SLL [2 -> 3 -> 1 -> 2 -> 3]
remove(2): True, Length: 4
  SLL [3 -> 1 -> 2 -> 3]
remove(3): True, Length: 3
  SLL [1 -> 2 -> 3]
remove(3): True, Length: 2
  SLL [1 -> 2]
remove(2): True, Length: 1
  SLL [1]
remove(1): True, Length: 0
  SLL []
```

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

This method counts the number of elements in the list that match the provided value. The method then returns this number. It must be implemented with **O(N)** runtime complexity.

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
```

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

This method returns a Boolean value based on whether or not the provided value exists in the list. It must be implemented with **O(N)** runtime complexity.

Example #1:

```
lst = LinkedList(["Waldo", "Clark Kent", "Homer", "Santa Claus"])
print(lst)
print(lst.find("Waldo"))
print(lst.find("Superman"))
print(lst.find("Santa Claus"))
```

Output:

```
SLL [Waldo -> Clark Kent -> Homer -> Santa Claus]
True
False
True
```

slice(self, start_index: int, size: int) -> “LinkedList”:

This method returns a new LinkedList 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. The original list cannot be modified. The runtime complexity of your implementation must be **O(N)**.

You are allowed to directly access the variable (_head) of LinkedList objects you create. 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”**.

Example #1:

```
lst = LinkedList([1, 2, 3, 4, 5, 6, 7, 8, 9])
ll_slice = lst.slice(1, 3)
print("Source:", lst)
print("Start: 1 Size: 3 :", ll_slice)
ll_slice.remove_at_index(0)
print("Removed at index 0 :", ll_slice)
```

Output:

```
Source: SLL [1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9]
Start: 1 Size: 3 : SLL [2 -> 3 -> 4]
Removed at index 0 : 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("Start:", index, "Size:", size, end="")
    try:
        print(" :", lst.slice(index, size))
    except:
        print(" : exception occurred.")
```

Output:

```
Source: SLL [10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16]
Start: 0 Size: 7 : SLL [10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16]
Start: -1 Size: 7 : exception occurred.
Start: 0 Size: 8 : exception occurred.
Start: 2 Size: 3 : SLL [12 -> 13 -> 14]
Start: 5 Size: 0 : SLL []
Start: 5 Size: 3 : exception occurred.
Start: 6 Size: 1 : SLL [16]
```

Part 2 - Summary and Specific Instructions

1. Implement a Stack ADT class by completing the provided skeleton code in the file `stack_da.py`. You will use the Dynamic Array data structure that you implemented in Assignment 2 as the underlying storage for your Stack ADT.

2. Your Stack ADT implementation will include the following standard Stack methods:

   ```
   push()
   pop()
   top()
   ```

3. The number of objects stored in the Stack at any given time will be between 0 and 1,000,000 inclusive. The stack must allow for the storage of duplicate objects.

4. **RESTRICTIONS:** You are not allowed to use ANY built-in Python data structures and/or their methods. You must solve this portion of the assignment by importing the DynamicArray class that you wrote in Assignment 2 and using class methods to write your solution.

   You are also not allowed to directly access any variables of the DynamicArray class (self._da._size, self._da._capacity, and self._da._data) or directly call double underscore ("dunder") methods. All work must be done by only using class methods.

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

This method adds a new element to the top of the stack. It must be implemented with **O(1) amortized** runtime complexity.

Example #1:

```
s = Stack()
print(s)
for value in [1, 2, 3, 4, 5]:
    s.push(value)
print(s)
```

Output:

```
STACK: 0 elements. []
STACK: 5 elements. [1, 2, 3, 4, 5]
```

pop(self) -> object:

This method removes the top element from the stack and returns its value. If the stack is empty, the method raises a custom **"StackException"**. It must be implemented with **O(1) amortized** runtime complexity.

Example #1:

```
s = Stack()
try:
    print(s.pop())
except Exception as e:
    print("Exception:", type(e))
for value in [1, 2, 3, 4, 5]:
    s.push(value)
for i in range(6):
    try:
        print(s.pop())
    except Exception as e:
        print("Exception:", type(e))
```

Output:

```
Exception: <class '__main__.StackException'>
5
4
3
2
1
Exception: <class '__main__.StackException'>
```

top(self) -> object:

This method returns the value of the top element of the stack without removing it. If the stack is empty, the method raises a custom **"StackException"**. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
s = Stack()
try:
    s.top()
except Exception as e:
    print("No elements in stack", type(e))
s.push(10)
s.push(20)
print(s)
print(s.top())
print(s.top())
print(s)
```

Output:

```
No elements in stack <class '__main__.StackException'>
STACK: 2 elements. [10, 20]
20
20
STACK: 2 elements. [10, 20]
```

Part 3 - Summary and Specific Instructions

1. Implement a Queue ADT class that utilizes a circular buffer (**as described in the Exploration**) by completing the provided skeleton code in the file `queue_sa.py`. You will use the Static Array data structure from previous assignments as the underlying storage for your Queue ADT.

2. Once completed, your implementation will include the following methods:

   ```
   enqueue()
   dequeue()
   front()
   ```

   The following private helper method in the skeleton code is used by `__str__()` to handle the "wraparound" in the circular buffer. You may find it helpful for your methods:

   ```
   _increment()
   ```

   There is also a suggested (optional and private) helper method in the skeleton code that you may wish to implement, to assist with resizing:

   ```
   _double_queue()
   ```

3. The number of objects stored in the queue at any given time will be between 0 and 1,000,000 inclusive. The queue must allow for the storage of duplicate elements.

4. **RESTRICTIONS:** You are not allowed to use ANY built-in Python data structures and/or their methods. You must solve this portion of the assignment by importing the StaticArray class provided in Assignment 1, and using class methods to write your solution.

   You are also not allowed to directly access any variables of the StaticArray class (self._sa._size and self._sa._data) or directly call double underscore ("dunder") methods. All work must be done by only using class methods.

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

This method adds a new value to the end of the queue. It must be implemented with **O(1) amortized** runtime complexity**.**

Example #1:

```
q = Queue()
print(q)
for value in [1, 2, 3, 4, 5]:
    q.enqueue(value)
print(q)
```

Output:

```
QUEUE: 0 elements. []
QUEUE: 5 elements. [1, 2, 3, 4, 5]
```

dequeue(self) -> object:

This method removes and returns the value at the beginning of the queue. If the queue is empty, the method raises a custom **"QueueException"**. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
q = Queue()
for value in [1, 2, 3, 4, 5]:
    q.enqueue(value)
print(q)
for i in range(q.size() + 1):
    try:
        print(q.dequeue())
    except Exception as e:
        print("No elements in queue", type(e))
for value in [6, 7, 8, 111, 222, 3333, 4444]:
    q.enqueue(value)
print(q)
q.print_underlying_sa()
```

Output:

```
QUEUE: 5 elements. [1, 2, 3, 4, 5]
1
2
3
4
5
No elements in queue <class '__main__.QueueException'>
QUEUE: 7 element(s). [6, 7, 8, 111, 222, 3333, 4444]
STAT_ARR Size: 8 [111, 222, 3333, 4444, 5, 6, 7, 8]
```

front(self) -> object:

This method returns the value of the front element of the queue without removing it. If the queue is empty, the method raises a custom **"QueueException"**. It must be implemented with **O(1)** runtime complexity.
NOTE: The circular buffer tests utilize the `action_and_print()` function to perform various operations and display the results. You are encouraged to review this function's code in the starter file to get a feel for how it works. For an explanation of circular buffers, please read **Exploration: Queues**.

Example #1:

```
q = Queue()
print(q)
for value in ['A', 'B', 'C', 'D']:
    try:
        print(q.front())
    except Exception as e:
        print("No elements in queue", type(e))
    q.enqueue(value)
print(q)
```

Output:

```
QUEUE: 0 elements. []
No elements in queue <class '__main__.QueueException'>
A
A
A
QUEUE: 4 elements. [A, B, C, D]
```

Testing for Circular Buffer

```
print("\n Circular buffer tests: #\n")

q = Queue()
print("# Enqueue: 2, 4, 6, 8")
test_case = [2, 4, 6, 8]
for value in test_case:
    q.enqueue(value)
print(q)
q.print_underyling_sa()
print()
```

Output:

```
# Enqueue: 2, 4, 6, 8
QUEUE: 4 element(s). [2, 4, 6, 8]
STAT_ARR Size: 4 [2, 4, 6, 8]
```

```
action_and_print("# Dequeue a value", q.dequeue, [], q)
```

Output:

```
# Dequeue a value
QUEUE: 3 element(s). [4, 6, 8]
STAT_ARR Size: 4 [2, 4, 6, 8]
```

```
action_and_print("# Enqueue: 10", q.enqueue, [10], q)
```

Output:

```
# Enqueue: 10
QUEUE: 4 element(s). [4, 6, 8, 10]
STAT_ARR Size: 4 [10, 4, 6, 8]
```

```
action_and_print("# Enqueue: 12", q.enqueue, [12], q)
```

Output:

```
# Enqueue: 12
QUEUE: 5 element(s). [4, 6, 8, 10, 12]
STAT_ARR Size: 8 [4, 6, 8, 10, 12, None, None, None]
```

```
print("# Dequeue until empty")
while not q.is_empty():
    q.dequeue
print(q)
q.print_underlying_sa()
print()
```

Output:

```
# Dequeue until empty
QUEUE: 0 element(s). []
STAT_ARR Size: 8 [4, 6, 8, 10, 12, None, None, None]
```

```
action_and_print("# Enqueue: 14, 16, 18", q.enqueue, [14, 16, 18], q)
```

Output:

```
# Enqueue: 14, 16, 18
QUEUE: 3 element(s). [14, 16, 18]
STAT_ARR Size: 8 [4, 6, 8, 10, 12, 14, 16, 18]
```

```
action_and_print("# Enqueue: 20", q.enqueue, [20], q)
```

Output:

```
# Enqueue: 20
QUEUE: 4 element(s). [14, 16, 18, 20]
STAT_ARR Size: 8 [20, 6, 8, 10, 12, 14, 16, 18]
```

```
action_and_print("# Enqueue: 22, 24, 26, 28", q.enqueue, [22, 24, 26, 28], q)
```

Output:

```
# Enqueue: 22, 24, 26, 28
QUEUE: 8 element(s). [14, 16, 18, 20, 22, 24, 26, 28]
STAT_ARR Size: 8 [20, 22, 24, 26, 28, 14, 16, 18]
```

```
action_and_print("# Enqueue: 30", q.enqueue, [30], q)
```

Output:

```
# Enqueue: 30
QUEUE: 9 element(s). [14, 16, 18, 20, 22, 24, 26, 28, 30]
STAT_ARR Size: 16 [14, 16, 18, 20, 22, 24, 26, 28, 30, None, None, None, None,
                   None, None, None]
```

Part 4 - Summary and Specific Instructions

1. Implement a Stack ADT class by completing the provided skeleton code in the file `stack_sll.py`. You will use a chain of **Singly-Linked Nodes** (the provided SLNode) as the underlying storage for your Stack ADT. Be sure to review the Exploration on Stacks for an example.

2. Your Stack ADT implementation will include the following standard Stack methods:

   ```
   push()
   pop()
   top()
   ```

3. The number of objects stored in the Stack at any given time will be between 0 and 1,000,000 inclusive. The stack must allow for the storage of duplicate objects.

4. **RESTRICTIONS:** You are not allowed to use ANY built-in Python data structures and/or their methods. You must solve this portion of the assignment by using the SLNode class provided with this assignment. Your solutions should not call double underscore (“dunder”) methods.

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

This method adds a new element to the top of the stack. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
s = Stack()
print(s)
for value in [1, 2, 3, 4, 5]:
    s.push(value)
print(s)
```

Output:

```
STACK []
STACK [5 -> 4 -> 3 -> 2 -> 1]
```

pop(self) -> object:

This method removes the top element from the stack and returns its value. If the stack is empty, the method raises a custom **"StackException"**. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
s = Stack()
try:
    print(s.pop())
except Exception as e:
    print("Exception:", type(e))
for value in [1, 2, 3, 4, 5]:
    s.push(value)
for i in range(6):
    try:
        print(s.pop())
    except Exception as e:
        print("Exception:", type(e))
```

Output:

```
Exception: <class '__main__.StackException'>
5
4
3
2
1
Exception: <class '__main__.StackException'>
```

top(self) -> object:

This method returns the value of the top element of the stack without removing it. If the stack is empty, the method raises a custom **"StackException"**. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
s = Stack()
try:
    s.top()
except Exception as e:
    print("No elements in stack", type(e))
s.push(10)
s.push(20)
print(s)
print(s.top())
print(s.top())
print(s)
```

Output:

```
No elements in stack <class '__main__.StackException'>
STACK [20 -> 10]
20
20
STACK [20 -> 10]
```

Part 5 - Summary and Specific Instructions

1. Implement a Queue ADT class by completing the provided skeleton code in the file `queue_sll.py`. You will use a chain of **Singly-Linked Nodes** (the provided SLNode) as the underlying storage for your Queue ADT. Be sure to review the Exploration on Queues for an example.

2. Once completed, your implementation will include the following methods:

```
enqueue()
dequeue()
front()
```

3. The number of objects stored in the queue at any given time will be between 0 and 1,000,000 inclusive. The queue must allow for the storage of duplicate elements.

4. **RESTRICTIONS:** You are not allowed to use ANY built-in Python data structures and/or their methods. You must solve this portion of the assignment by using the SLNode class provided with this assignment. Your solutions should not call double underscore (“dunder”) methods.

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

This method adds a new value to the end of the queue. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
q = Queue()
print(q)
for value in [1, 2, 3, 4, 5]:
    q.enqueue(value)m
print(q)
```

Output:

```
QUEUE []
QUEUE [1 -> 2 -> 3 -> 4 -> 5]
```

dequeue(self) -> object:

This method removes and returns the value from the beginning of the queue. If the queue is empty, the method raises a custom **"QueueException"**. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
q = Queue()
for value in [1, 2, 3, 4, 5]:
    q.enqueue(value)
print(q)
for i in range(6):
    try:
        print(q.dequeue())
    except Exception as e:
        print("No elements in queue", type(e))
```

Output:

```
QUEUE [1 -> 2 -> 3 -> 4 -> 5]
1
2
3
4
5
No elements in queue <class '__main__.QueueException'>
```

front(self) -> object:

This method returns the value of the front element of the queue without removing it. If the queue is empty, the method raises a custom **"QueueException"**. It must be implemented with **O(1)** runtime complexity.

Example #1:

```
q = Queue()
print(q)
for value in ['A', 'B', 'C', 'D']:
    try:
        print(q.front())
    except Exception as e:
        print("No elements in queue", type(e))
    q.enqueue(value)
print(q)
```

Output:

```
QUEUE []
No elements in queue <class '__main__.QueueException'>
A
A
A
QUEUE [A -> B -> C -> D]
```