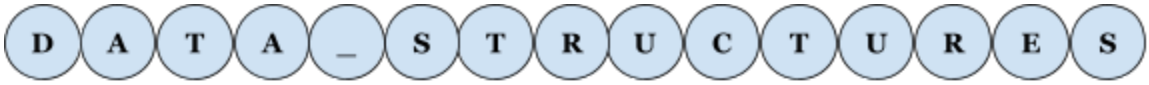
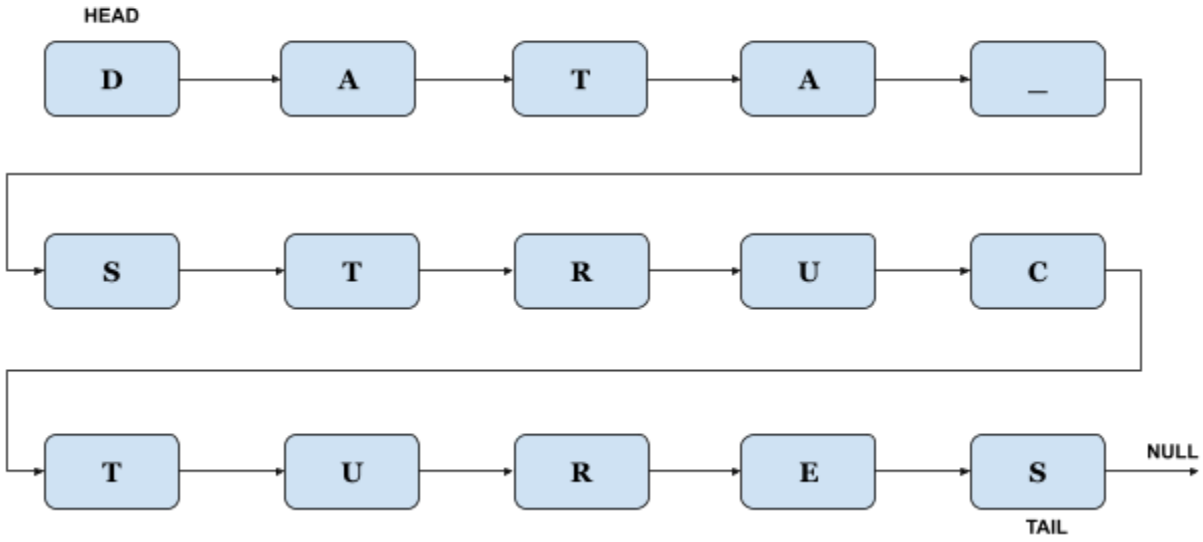
CS261 Data Structures

Assignment 3

Winter 2022

Linked List and ADT Implementation





Contents

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. For full credit you must pass all of the tests.

3. We encourage you to create your own test programs and cases even though those additional tests 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 any non-obvious code.

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. **All methods must be implemented iteratively.**

8. You may not use any imports beyond the ones included in the assignment source code provided.

Part 1 - Summary and Specific Instructions

1. Implement a Singly Linked List data structure with the following methods 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. 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__.

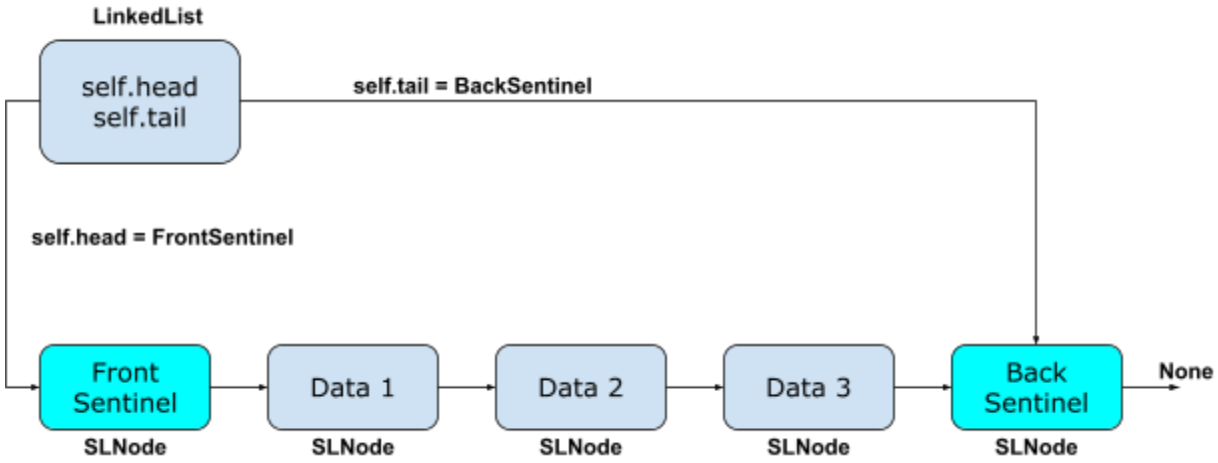
3. The number of objects stored in the list at any given time will be between 0 and 900 inclusive.

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

5. Variables in the SLNode (provided in its own file, as it will be used later in Parts 4 and 5) 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.

6. RESTRICTIONS: You are not allowed to use ANY built-in Python data structures or their methods.

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



insert_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.insert_front('A')
lst.insert_front('B')
lst.insert_front('C')
print(lst)
```

Output:

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

insert_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.insert_back('C')
lst.insert_back('B')
lst.insert_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_at_index(self, index: int) -> None:

This method removes the node at the specified index from the 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 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]
```

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 a 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
```

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

This method returns a Boolean value based on whether or not the provided "value" object is in the list.

Example #1:

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

Output:

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

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. The original list cannot be modified. The 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 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 data storage for your Stack ADT.

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

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

3. 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__.

4. 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.

5. 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 (like self.size, self.capacity, and self.data in Assignment 2). 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. It must be implemented with O(1) amortized runtime complexity. If the stack is empty, the method raises a custom "StackException". The code for the exception is provided in the skeleton file.

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. It must be implemented with O(1) runtime complexity. If the stack is empty, the method raises a custom "StackException". The code for the exception is provided in the skeleton file.

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 by completing the provided skeleton code in the file `queue_da.py`. You will use the Dynamic Array data structure that you have implemented in Assignment 2 as the underlying data storage for your Queue ADT.

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

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

3. 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__.

4. 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.

5. 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 (like self.size, self.capacity, and self.data in Assignment 2). 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 from at the beginning of the queue. It must be implemented with O(N) runtime complexity. If the queue is empty, the method raises a custom "QueueException". The code for the exception is provided in the skeleton file.

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: 5 elements. [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. It must be implemented with O(1) runtime complexity. If the queue is empty, the method raises a custom "QueueException". The code for the exception is provided in the skeleton file.

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]
```

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 data storage for your Stack ADT. Be sure to review the Exploration on Stacks as needed for an example.

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

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

3. 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__.

4. 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.

5. 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.

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. It must be implemented with O(1) runtime complexity. If the stack is empty, the method raises a custom "StackException". The code for the exception is provided in the skeleton file.

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. It must be implemented with O(1) runtime complexity. If the stack is empty, the method raises a custom "StackException". The code for the exception is provided in the skeleton file.

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 data storage for your Queue ADT. Be sure to review the Exploration on Queues as needed for an example.

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

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

3. 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__.

4. 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.

5. 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.

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)
print(q)
```

Output:

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

dequeue(self) -> object:

This method removes and returns the value from the beginning of the queue. It must be implemented with O(1) runtime complexity. If the queue is empty, the method raises a custom "QueueException". The code for the exception is provided in the skeleton file.

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. It must be implemented with O(1) runtime complexity. If the queue is empty, the method raises a custom "QueueException". The code for the exception is provided in the skeleton file.

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]
```