Assignment 3 – MSCI 240

Name: Karla Castro

Student ID#: 20745522

1. **Introduction: A brief summary of the whole report**

   This report describes a high-level description and the simplified complexity time analysis for the methods in the LinkedList class.

2. **A brief description and overview of the class**

   Class name: LinkedList

   The LinkedList class is a data structure that will manage a linked list of ListNode objects. The class has 1 private field and 1 public field, and different methods that are helpful to manipulate and retrieve information from the linked list. The methods include 4 accessors, 4 mutators, and 1 sorting function.

   The bubble sort algorithm was used because is a straightforward and easy to understand algorithm. The implementation is concise and did not require a lot of lines of code. It was also adaptable to sorting linked lists from the array list one.

   Class Structure:

   1. **Constructor:**
      - **__init__(self):** Creates a new empty linked list and initialized private fields:
        - self.head (public field) : head of the linked list.
        - self._length (private field): length of the list.
   2. **Accessors:**
      - **def length(self) -> int**: Returns the number of elements stores in the linked list, representing the length of the linked list.
        - Returns: self._length.
      - **def contains(self, value, map_to_key=lambda x: x) -> bool**: This method checks if a specific value or mapped key belongs to the linked list.

This function calls the find_index_byvalue method, which returns the index of a value if in the liked list.

  - Returns **True** if value is in linked list.
  - Returns **False** if value is not in linked list.

- **def find_index_byvalue(self, value, map_to_key=lambda x: x) -> int**: This method iterates through the linked list and returns the index of the value if this or its mapped key is in the linked list.
  - Returns the first index of a provided value or its mapped key in the linked list.
  - Returns -1 if not found.
- **def get(self, k: int) -> object**: This method returns the element stored at an specified index k in the linked list. For this, the method iterates through the linked list by until current reaches the kth node**.**
  - Raises an **IndexError** if the index is out of range.
  - Returns current.data (element at index k)

3. **Mutators:**

- **def set(self, k: int, value: object)**: This method sets the value of the kth node in the linked list to the provided new value.
  - Raises an **IndexError** if the index is out of range.
  - Raises an **IndexError** if k is out of range.
  - Sets the data of the kth node to the value.
- **def append(self, value: object)**: Adds a new ListNode with the provided (obj) to the end of the linked list, storing the provided value.
  - Creates a new list node: ListNode(obj)
  - If the list is empty, it sets the new node as the head of the list.
  - If the list is not empty, it iterates through the list until it reaches the last node and sets the next node to the new node.
  - It increments the length of the list.

- **def insert(self, k: int, value: object)**: Inserts a new list node with the provided value:object as the kth node in the linked list, shifting subsequent values rightward.
    - Raises an **IndexError** if the index is out of range.
    - If inserting at index 0, it sets the new node as the head and adjusts the next pointer.
    - If index not at 0, it iterates through the list to the node before the specified index and inserts the new node.
    - Increments the length of the list
- **def remove(self, k: int) -> object**: Removes the kth list node in the linked list and returns its value.
    - Raises an **IndexError** if the index is out of range.
    - If removing at index 0, it sets the head to the next node.
    - If not at index 0, iterates through the list to the node before the specified index and adjusts the next pointer to skip the node being removed.
    - Reduces the length of the list.
    - Returns the removed value

4. **Sorting:**
    - **def sort(self, map_to_key=lambda x: x, descending=False)**: Sorts the list in-place using the bubble sort algorithm.
        - Returns sorted list.

3. Class Implementation Description:

a. Class Diagram

Class	***LinkedList***	
Fields	**Constructor**	head
		_length
Behaviour	**Accessors**	length()
		contains()
		find_index_byvalue()
		get()
	Mutators	set()
		append()

		insert()
		remove()
	Sorting	sort()

4. Simplified Time Analysis

Constructor:

```python
def __init__(self):
    """Create an empty list."""
    self.head = None # empty list with no head
    self._length = 0 # empty list with 0 length
```

- This has 2 statements.
  - 2 assigning variables.
- Total:
  - **T(n) = 2**
  - **Big O = O(1)**
- The constructor has constant time complexity O(1) since the algorithm is only assigning variables.

Accessors:

- **def length(self) -> int**:

```python
def length(self) -> int:
    """Return number of elements stored in the linked list."""
    return self._length
```

- This has only 1 returning statement.
- Total:
  - **T(n) = 1**
  - **Big O = O(1)**
- The length method has constant time complexity O(1) since the algorithm is only returning the length.

- **def contains(self, value, map_to_key=lambda x: x) -> bool**:

```python
def contains(self, value, map_to_key=lambda x : x) -> bool:
    """ contains
        Returns True if value is in the list, False otherwise

        map_to_key: an optional mapping function from the stored
            object type to the key to compare with.
    """
    # Returns True if the value is in the list
    return self.find_index_byvalue(value, map_to_key=map_to_key) >= 0
```

  - This has 1 statement:
    - Calls find_index_byvalue method:
      - Big O = O(n, k) = k*n = O(n2)
  - Total:
    - **$T(n, k) = 1 + (2k+5)n$**
    - **Big O = $O(n, k) = k*n = O(n^2)$**
  - The contains method has complexity **$O(n^2)$** which is a quadratic time complexity.

- **def find_index_byvalue(self, value, map_to_key=lambda x: x) -> int**:

```python
def find_index_byvalue(self, value, map_to_key=lambda x : x) -> int:
    """ find_index_byvalue
        Returns the first index of value, or -1 if not found

        value: the value to find as an object,
            or the key to compare to (if using map_to_key)

        map_to_key: an optional mapping function from the stored
            object type to the key to compare with.
    """
    # for loop to iterate through the linked list
    for i in range(self.length()):
        # checks if the value is in the list
        # compares the value to the key of the current node
        if map_to_key(self.get(i)) == value:
            # returns the index of the value
            return i
    return -1
```

- This has:
  - 1 for loop statement (n):
    - Calls length () method with T(n) = 1
    - One if statement that calls the map_to_key method and get accessor:
      - map_to_key: 1 statement
      - get(): 2k + 2
      - 1 return statement for returning the index
- Total:
  - T(n, k) = (1+1+2k+2+1)n
  - **T(n, k) = (2k+5)n**
  - **Big O = O(n, k) = k*n = O(n^2)**
- The contains method has complexity **O(n^2)** which is a quadratic time complexity given the get method was called inside a for loop.

- **def get(self, k: int) -> object**:

```python
def get(self, k : int) -> object:
    """Return element at index k."""
    # check if k is in range
    if not 0 <= k < self._length:
        # raise an error if k is out of range
        raise IndexError("Index k out of range")
    # set current to reference the head of the linked list
    current = self.head
    # iterate through the list until current reaches the kth node
    for i in range (k):
        # check if current is not None and the node is valid at the current position
        if current is not None:
            # set current to reference the next node
            current = current.next
        else:
            # raise an error if current is None
            raise IndexError("Index out of range")
    # returns the data stored in the kth node
    return current.data
```

- This has:

- 1 statement setting current to reference the head
- 1 for loop (k)
  - 1 if statement checking current is not None
    - 1 statement setting current to reference the next node
- 1 statement returning the data stored in kth node

- Total:
  - T(k) = 1 + 2k + 1
  - **T(k) = 2k + 2**
  - **Big O = O(k)**
- The get statement has a linear time complexity, where k is a constant.

Mutators:

- **def set(self, k: int, value: object)**:

```python
def set(self, k, value):
    """Set element at index k"""
    # check if k is in range
    if not 0 <= k < self._length:
        # raise an error if k is out of range
        raise IndexError("Index k our of range")
    # set current to reference the head of the linked list
    current = self.head
    # iterate through the list until current reaches the kth node
    for i in range(k):
        # Update current to point to the next node in the list
        current = current.next
        # check if current index is out of range by checking if it is None
        if current is None:
            raise IndexError("Index out of range")
    # set the data of the kth node to the value
    current.data = value
```

- This has:
  - 1 statement setting current to reference the head
  - 1 for loop (k) to iterate through the list until kth node
    - 1 statement updating current to the next node
  - 1 statement setting the data of the kth node to the value

- Total:
  - $T(k) = 1 + k + 1$
  - **$T(k) = k + 2$**
  - **Big O = $O(k)$**
- The set statement has a linear time complexity, where k is a constant.

- **def append(self, value: object)**:

```
def append(self, obj):
    """Add object to end of the array."""
    # create a new list node with obj
    new_list_node = ListNode(obj)
    if not self.head:
        # if the list is empty, set the new node as the head
        self.head = new_list_node
    else:
        # if the list is not empty, set current to the head
        current = self.head
        while current.next:
            # iterate until the last node is reached
            current = current.next
        # set the next node to reference the new node
        current.next = new_list_node
    # increment the length of the list
    self._length += 1
```

- This has:
  - 1 statement creating a new list node
    - Creates ListNode instance:
      - 2 statements assigning value and reference
  - 1 if statement
    - 1 statement setting the new node as the head
  - 1 else statement
    - 1 statement setting current to the head
    - 1 while statement with $\log_2(n)$
      - 1 statement
    - 1 statement setting the next node to the reference new node
  - 1 statement incrementing length

            - length accessor with $T(n) = 1$
    - Total:
        - **$T(n) = 10 + log_2(n)$**
        - **Big O = $O(log_2(n))$**
    - The append method has a **$O(log_2(n))$** that represents logarithmic time complexity. This is given for the while loop statement.
- **def insert(self, k: int, value: object)**:

```
def insert(self, k, value):
    """Insert value at index k, shifting subsequent values rightward."""
    # NOTE: 0 <= k <= n is okay
    # TODO: stub
    # check if k is in range
    if k < 0 or k > self._length:
        raise IndexError("Index out of range")
    # create a new node
    new_node = ListNode(value)
    # check first index
    if k == 0:
        # sets the new node to reference the head
        new_node.next = self.head
        # adjust the next pointer
        self.head = new_node
    else:
        current = self.head
        for i in range(k -1):
            # iterate through the list to the node before the kth node
            current = current.next
        # set the next node of the new node to the next node of the current node
        new_node.next = current.next
        # set the next node to the new node
        current.next = new_node
    # increment the length of the list
    self._length += 1
```

    - This has:
        - 1 statement creating a new list node
            - Creates ListNode instance:
                - 2 statements assigning value and reference
        - 1 else statement:

        - 1 statement setting current to next
        - 1 for loop (k-1)
            - 1 statement setting current to next
        - 2 statements
    - 1 statement incrementing length
        - 1 statement from length()
- Total:
    - T(k) = 5 + k + 4
    - **T(k) = k + 9**
    - **Big O = O(k)**
- The insert statement has a linear time complexity, where k is a constant.

- **def remove(self, k: int) -> object**:

```python
def remove(self, k : int) -> object:
    """Remove the value at index k, returning it
    """
    # check if k is in range
    if k < 0 or k >= self._length:
        # raise an error if k is out of range
        raise IndexError("Index out of range")
    if k == 0:
        # update head when removing at index 0
        removed_value = self.head.data
        # sets head to reference the next node
        self.head = self.head.next

    else:
        # set current to the head
        current = self.head
        for i in range(k - 1):
            # iterate through the list to the node before the kth node
            current = current.next
        # set the removed value to the data of the kth node
        removed_value = current.next.data
        current.next = current.next.next
    # reduce the length of the list
    self._length -= 1
    return removed_value
```

- This has:

        - 1 else statement:
            - 1 statement setting current to next
            - 1 for loop (k-1)
                - 1 statement setting current to next
            - 2 statements
        - 1 statement incrementing length
            - 1 statement from length()
        - 1 return statement
    - Total:
        - T(k) = 2 + k + 6
        - **T(k) = k + 8**
        - **Big O = O(k)**
    - The remove statement has a linear time complexity, where k is a constant.

Sorting:

- **def sort(self, map_to_key=lambda x: x, descending=False)**:

```python
# check if list is length 1 or 0
if self._length <= 1:
    return

    # Iterate through each element in the list
for i in range(self._length):
    current = self.head
    # Iterate trhough the unsorted part of the list
    for j in range(0, self._length - i - 1):
        # Create key for the current element in the unsorted part
        key1 = map_to_key(self.get(j))
        # Create key for next element in the unsorted part
        key2 = map_to_key(self.get(j+1))
        # Swap if the keys are not in order
        # Compare keys based on the specified order (ascending or descending).
        if ((key1 > key2) if not descending else (key1 < key2)):
            # Swap the elements
            temp = current.data
            current.data = current.next.data
            current.next.data = temp
        current = current.next
```

    - This has:
        - 1 for loop (n)
            - 1 statement from length()
            - 1 statement

                - 1 for loop (n)
                    - 1 statement from length()
                    - 1 statement creating key 1
                        - 1 statement from map_to_key
                        - 4k + 5 from get()
                    - 1 statement creating key 2
                        - 1 statement from map_to_key
                        - 2k + 2 from get()
                    - 1 if statement
                        - 3 statements
                - 1 statement assigning current_next to current
            - Total:
                - T(n, k) = 2n + (3+1+4k+5+1+1+4k+5+1+1+1)n
                - **T(n, k) = 2n + (19+8k)n**
                - **Big O = O(n, k) = O(n^2)**
- To calculate the time complexity, it will depend on the product of the time complexities of the outer loop, the inner loop, and the key mapping and swap. In the worst-case time complexity of the provided sorting algorithm is (**O(n^2)**