Mamadou Diao Kaba 27070179

Problem 1. Please refer to the class LinkedLDictionary.java
Problem 2.
Asymptotic Analysis for Link list-based Dictionary

Operation	Best cast time cost	Worst case time cost	Average case time cost
clear	O(1)	O(1)	O(1)
insert	O(1)	O(1)	O(1)
remove	O(1)	O(n)	O(n)
removeAny	O(1)	O(1)	O(1)
find	O(1)	O(n)	O(n)
size	O(1)	O(n)	O(n)
Tostring	O(n)	O(n)	O(n)

Problem 3. Please refer to the class DoubleLinkedListDictionary.java
Problem 4.
Asymptotic Analysis for double Linked-list-based Dictionary

Operation	Best cast time cost	Worst case time cost	Average case time cost
clear	O(1)	O(1)	O(1)
insert	O(1)	O(1)	O(1)
remove	O(1)	O(n)	O(n)
removeAny	O(1)	O(1)	O(1)
find	O(1)	O(n)	O(n)
size	O(1)	O(n)	O(n)
Tostring	O(n)	O(n)	O(n)

Singly linked list

1. **clear()**
    - Best Case: O(1)
    - Average Case: O(1)
    - Worst Case: O(1)
    - Explanation: Clearing the dictionary involves setting the **head** of the linked list to **null**, which is a constant time operation.
2. **find(Key k)**
    - Best Case: O(1)
    - Average Case: O(n)
    - Worst Case: O(n)
    - Explanation: In the best case, the key we are looking for is at the head of the linked list, so we find it in constant time. In the average and worst cases, we may need to traverse the linked list to find the key, which takes O(n) time in the worst case (where n is the number of elements in the dictionary).
3. **insert(Key k, E e)**
    - Best Case: O(1)
    - Average Case: O(1)
    - Worst Case: O(1)
    - Explanation: Inserting a key-value pair involves creating a new node and appending it to the end of the linked list. This operation takes constant time because we don't need to search for the key since you're checking for duplicates before insertion.
4. **remove(Key k)**
    - Best Case: O(1)
    - Average Case: O(n)
    - Worst Case: O(n)
    - Explanation: In the best case, the key to remove is at the head of the linked list, so we can remove it in constant time. In the average and worst cases, we may need to traverse the linked list to find the key to remove, which takes O(n) time in the worst case.
5. **removeAny()**
    - Best Case: O(1)
    - Average Case: O(1)
    - Worst Case: O(1)

- Explanation: Removing the "any" element simply involves updating the **head** of the linked list to the next node, which is a constant time operation.

6. **size()**
    - Best Case: O(1)
    - Average Case: O(n)
    - Worst Case: O(n)
    - Explanation: To find the size of the dictionary, we need to traverse the entire linked list and count the number of elements. This takes O(n) time in both the average and worst cases.
7. **toString()**
    - Best Case: O(n)
    - Average Case: O(n)
    - Worst Case: O(n)
    - Explanation: Generating the string representation involves traversing the linked list and appending each key-value pair to the output string. This takes O(n) time because we visit each element once.

Note: In the best-case scenarios for **find**, **insert**, **remove**, and **removeAny**, we assume that the keys are uniformly distributed in the linked list. The worst-case scenarios for these methods occur when we have to traverse the entire linked list to find the key or element to be removed.

Double linked list

1. **clear()**
    - Best Case: O(1)
    - Average Case: O(1)
    - Worst Case: O(1)
    - Explanation: Clearing the dictionary involves setting both **head** and **tail** to **null**, which is a constant time operation.
2. **find(Key k)**
    - Best Case: O(1)
    - Average Case: O(n)
    - Worst Case: O(n)
    - Explanation: In the best case, the key we are looking for is at the head of the linked list, so we find it in constant time. In the average and worst cases, we may need to traverse the linked list to find the key, which takes O(n) time in the worst case (where n is the number of elements in the dictionary).

3. **insert(Key k, E e)**
    - Best Case: O(1)
    - Average Case: O(1)
    - Worst Case: O(1)
    - Explanation: Inserting a key-value pair involves creating a new node and appending it to the end of the double linked list. This operation takes constant time because we don't need to search for the key since you're checking for duplicates before insertion.
4. **remove(Key k)**
    - Best Case: O(1)
    - Average Case: O(n)
    - Worst Case: O(n)
    - Explanation: In the best case, the key to remove is at the head or tail of the linked list, so we can remove it in constant time. In the average and worst cases, we may need to traverse the linked list to find the key to remove, which takes O(n) time in the worst case.
5. **removeAny()**
    - Best Case: O(1)
    - Average Case: O(1)
    - Worst Case: O(1)
    - Explanation: Removing the "any" element involves updating the **tail** of the linked list to the previous node, which is a constant time operation.
6. **size()**
    - Best Case: O(1)
    - Average Case: O(n)
    - Worst Case: O(n)
    - Explanation: To find the size of the dictionary, we need to traverse the entire linked list and count the number of elements. This takes O(n) time in both the average and worst cases.
7. **toString()**
    - Best Case: O(n)
    - Average Case: O(n)
    - Worst Case: O(n)

- Explanation: Generating the string representation involves traversing the linked list and appending each key-value pair to the output string. This takes O(n) time because we visit each element once.

The best-case scenarios for **find**, **insert**, **remove**, and **removeAny** occur when the keys or elements to be operated on are at the head or tail of the linked list. The worst-case scenarios for these methods occur when we have to traverse the entire linked list to find or remove the key or element.