ESKISEHIR TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EEM 480 HOMEWORK 2

ALGORITHM AND COMPLEXİTY

Selin Akalın
40444602194

Purpose

The purpose of this assignment is to learn how to use double lists and put what we know into practice. In this assignment, we need to create a LinkedList class that uses a node class called DLNode. There are a number of methods listed in the HW2Interface interface, and we need to code these methods within the LinkedList class.

In summary, this assignment aims to evaluate our ability to work with linked lists, understand and apply the specified methods, handle errors appropriately.

Algorithm

Insert(int newElement, int pos):

1. Check if the position (`pos`) is valid. If not, throw a `LinkedListException`.
2. Create a new `DLNode` with the specified element (`newElement`).
3. If `pos` is 0, insert the new node at the beginning of the list:

    - Set `newNode.right` to the current `head`.
    - Set `newNode.left` to `null`.
    - If `head` is not `null`, set `head.left` to `newNode`.
    - Update `head` to `newNode`.

4. If `pos` is not 0, traverse the list to find the node at position `pos - 1`:

    - Move through the list while keeping track of the current position.
    - If the end of the list is reached before reaching the desired position, throw a `LinkedListException`.
    - Insert the new node between the found node and its right neighbor.

Delete(int pos):

1. Check if the position (`pos`) is valid. If not, throw a `LinkedListException`.
2. If `pos` is 0, delete the first node:

    - Store the value of the current head.
    - Update `head` to the right neighbor.
    - If the new head is not `null`, set its `left` pointer to `null`.
    - Return the stored value.

3. If `pos` is not 0, traverse the list to find the node at position `pos`:
    - Move through the list while keeping track of the current position.
    - If the end of the list is reached before reaching the desired position, throw a `LinkedListException`.
    - Store the value of the found node.
    - Update the pointers of the left and right neighbors to skip the found node.
    - Return the stored value.

ReverseLink():

1. Initialize three pointers: `current`, `next`, and `left`.
2. Start with `ref` pointing to the head.
3. While `ref` is not `null`:

    - Set `next` to the right neighbor of `ref`.
    - Set the right pointer of `ref` to `left`.
    - Set the left pointer of `ref` to `next`.
    - Move `left` and `ref` one step forward in the list.

SquashL():

1. Start from the head and iterate through the list.
2. For each unique element encountered, count the number of contiguous occurrences.
3. Replace the contiguous occurrences with a tuple containing the element and its count.

OplashL():

The `OplashL` method is essentially the reverse of the `SquashL` method.

1. Start from the head and iterate through the list.
2. For each tuple encountered, replicate the element according to its count.

Output() and ROutput():

Simply traverse the list either from the head to the end (for `Output()`) or from the end to the head (for `ROutput()`) and print each element.

Referances

[1] https://www.geeksforgeeks.org/delete-doubly-linked-list-node-given-position/

[2] https://www.geeksforgeeks.org/introduction-and-insertion-in-a-doubly-linked-list/

[3] https://www.geeksforgeeks.org/reverse-a-doubly-linked-list/

[4] https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_algorithm.htm

[5] https://medium.com/@singhamritpal49/doubly-linked-list-20b7e45bb37