PROBLEM SOLUTION APPROACH FOR LDLinkedList

BATUHAN BASAK

1 How To Implement Lazy Deletion

To implement lazy deletion, the list must marked the first removal item. To do this, LDLinkedList has two more references which are **removalNode** and **precedeOfRemovalNode**. The variable **removalNode** is the reference to removal data that list will have removed for next call of method remove. And the **precedeOfRemovalNode** is the preceding node of removal node.

LDLinkedList is holding reference of preceding node of removal node because it is a singly linked list. Thus it can't iterate backwardly.

If remove method called and **removalNode** is null, then it sets **removalNode** to data remove method takes, and **precedeOfRemovalNode** is also setted as reference of preceding node of removal node as expected. If remove method is called when **removalNode** is not null, remove method checks a case before removing both **removalNode** and the other removal data given as parameter.

The case is that whether is **removalNode** and second removal data same data or not. If they are same, the user tries to remove same object twice, function does nothing but just returns removal data. After that remove method determines that whether is removal data head node, left most item on list, or not. If it is, then it removes the **removalNode** and second removal data respectively, **removalNode** and **precedeOfRemovalNode** are setted to null after deletion. If the removal data, the second removal item, is not head node, then remove method gets the reference node of the second removal item before deleting **removalNode**. The reason is that **removalNode** could be left side of the second removal data on the list which makes removing following data of second removal data as second removal data. Because index of second removal data will have been decreased by one after **removalNode** will been removed. Getting reference preceding node of second removal data prevents this problem. After reference of preceding data of second removal data, we can removal **removalNode** and second removal data conveniently.

2 Implementing set, get, add, and remove Methods

2.1 Helper Methods

- getNode(int index) - Finds the reference node at the position *index* in list. The time complexity of getNode is linear time, O(n).
- addFirst(E e) - Adds new data *e* to the head, left most item, in the list. The time complexity of addFirst is constant time, O(1).
- addAfter(Node¡E¿ node, E e) - Adds new item *e* next of the node *node* in the list. The time complexity of addAfter is constant time, O(1).
- removeFirst() - Removes the head node from the list. The time complexity of removeFirst is constant time, O(1).
- removaAfter(Node¡E¿ node) - Removes following node of the *node* given in the list. The time complexity of removeAfter is constant time, O(1).

2.2 Methods

- Set(int index, E e) - Sets new value to the data at position *index.* To get the node at the position *index* it uses helper method getNode. Hence time complexity of getNode is O(n), the time complexity of set method is O(n).
- Get(int index) - Gets the value of data at the position *index* in the list. The get the node at that position, it uses helper method getNode. Hence time complexity of getNode is O(n), the time complexity of set method is O(n).
- Add(int index, E e) - Adds a new data *e* to the position *index.*
  To do this first checks whether index is 0 or not. If index is zero, then uses the helper method addFirst. Else uses method getNode to find preceding node of the positon index, getNode(index - 1), then uses method addAfter to add the new data after the preceding node. Hence the time complexity of getNode is O(n), the time complexity of add method is linear time, O(n).
- Remove(int index) - Removes the data at position *index.* There are two cases running this method as mentioned above earlier. For both cases remove method use helper method getNode. Thus even removing node is constant time, the time complexity of remove method is linear time, O(n).