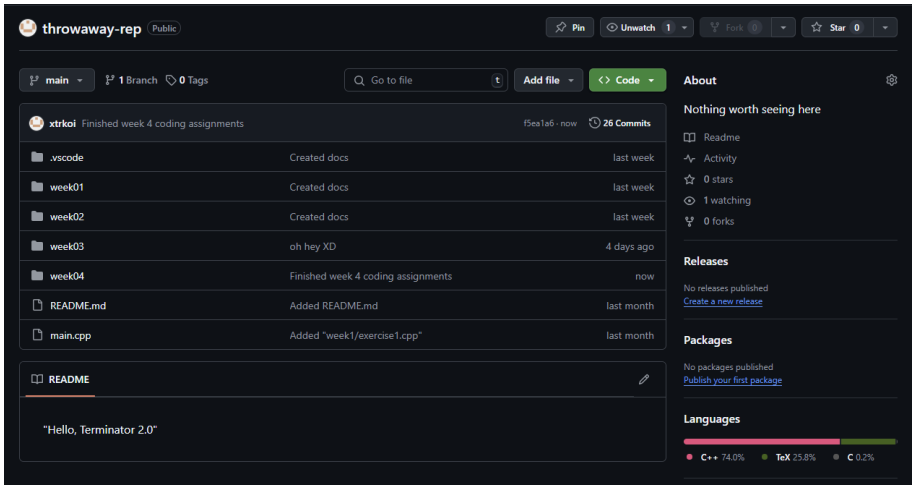
Weekly Homework Report #4

Tran Van Tan Khoi

April 15, 2025

1 About

Typical implementation of some functions of Singly Linked Lists and Doubly Linked Lists. Access the Github repository via this hyperlink.



2 Singly Linked List

Before we begin, there are some auxiliary functions that would help in implementation of the later functions. These include:

1. NODE *pred(List *&L, NODE *p_node): finds the previous node of p_node in the list L (this is especially useful when there is no easy way to access the previous node in singly linked lists).
2. NODE *succ(List *&L, NODE *p_node): finds the next node of p_node in the list L.
3. NODE *find_first_of(List *&L, **int** key): finds the first node in the list L with given key.
4. **bool** remove_node(List *&L, NODE *p_node): remove the node with given pointer.
5. NODE *extract_node(List *&L, **int** index): finds the node with given index.
6. **int** extract_index(List L, NODE p_node): finds the index of given node.

2.1 Create a new node

Creating a new node and returning a pointer to the created node is easily accomplished with the **new** operator. Note that if the allocation fails then **new** simply returns nullptr, which is also what we would return in this case. Checking for allocation failure should be done outside this function.

2.2 Create a new list with a given node

The List class already come with a constructor, simply call List(p_node, p_node) (p_node is an input parameter).

2.3 Insert a node with a given key at the front

Consider two cases: when the list is empty, and when it is not.

When the list is empty, simply attach the p_head and p_tail pointer to the newly created node with given key.

When the list isn't empty, make sure the p_next pointer of the newly created node points the current p_head. Then modify the p_head pointer so it points to the node.

2.4 Insert a node with a given key to the back

A similar approach can be done by considering two cases of whether the list is empty or not.

When the list is empty, simply attach the p_head and p_tail pointer to the newly created node with given key.

When the list isn't empty, change the p_tail 's p_next pointer then change the p_tail pointer itself.

2.5 Remove the first node

We can follow every steps in the front-insertion method in a reverse fashion. We only further need to take into account of the the case where the list is already empty.

2.6 Remove the last node

Same thing, just the opposite.

2.7 Remove all nodes without destroying the list

We assume that we don't deallocate the list itself, only remove all content of the list. There isn't a more straightforward way than removing each node one-by-one.

2.8 Remove the previous node of the node with given key

First, find the node with the given key. Then, remove the node before it. This can be all done with auxiliary functions we implemented beforehand.

2.9 Remove the next node of the node with given key

We do the same thing as before, but removing the node after instead.

2.10 Add a node at a given position

This is where the extract_node comes in handy. Insertion can be handled by determining the nodes before and after the node to be inserted, with special case can be handled by addHead and addTail functions.

2.11 Remove a node at a given position

It is much simpler than adding because we can use the extract_node and remove_node auxiliary functions.

2.12 Add a node before the node with given key

We find the node with the given key then use the function which will add the node at the given position.

2.13 Add a node after the node with given key

Almost the same thing, just slightly different.

2.14 Print all nodes in a list

Traverse the nodes in the list one at a time and print the key of each node, assuming the list doesn't have loops.

2.15 Count the number of nodes in a list

We can traverse the list and increase the count when visiting each node, or find the index of the last node with extract_index auxiliary function.

2.16 Reverse a list by creating a new list

Unlike reversing a linked list in-place, in which we manage some moving pointers, creating a new list which is the reverse of the given list can be easier. Traverse the list from front to back and insert each node to the head of the new list.

2.17 Remove all duplicate nodes so each node in the list is unique

An inefficient solution is to traverse the list and remove each node with same key as the current node which isn't the current node itself.

2.18 Remove all nodes with a given key

An inefficient solution is to repeatedly find a node with the given key and remove it from the list.

3 Doubly Linked List

In a doubly linked list, each node now also has a pointer points to the node preceeding it. This makes insertions and removals a bit more complicated but allows for much faster backward traversal in the list.