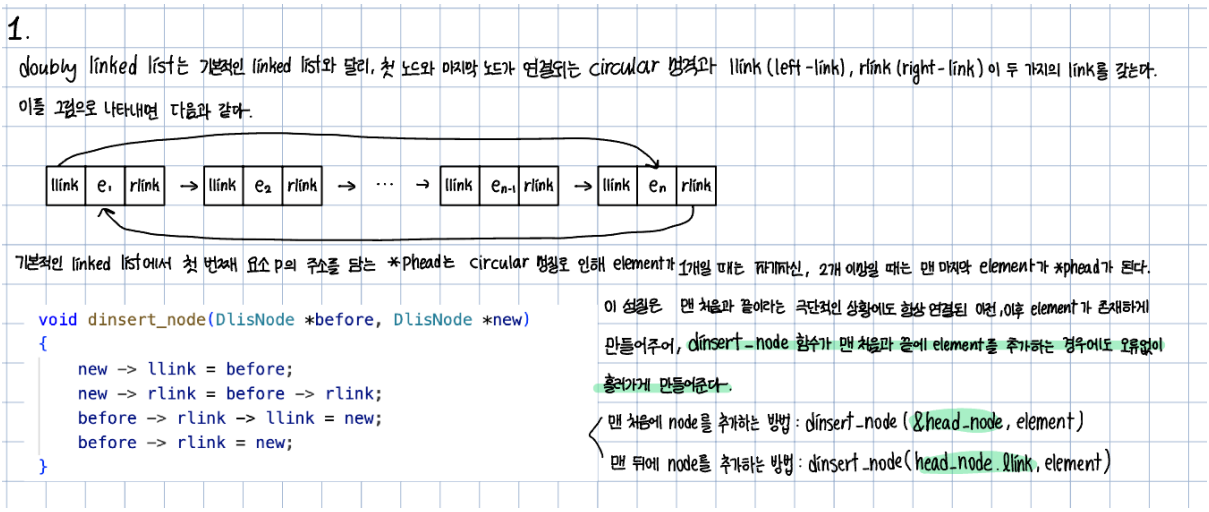
Technical Note

2229027 이지민

1 Hand Writing Note




```
void dinsert_node(DlisNode *before, DlisNode *new)
{
    new -> llink = before;
    new -> rlink = before -> rlink;
    before -> rlink -> llink = new;
    before -> rlink = new;
}
```

The first question asks whether the **dinsert_node** function can insert a new node at the beginning and end of a doubly linked list. As explained earlier, the **dinsert_node** function can indeed insert a new node at any position within the doubly linked list, including at the beginning or end. Therefore, there is no need for additional **dinsert_first_node** or **dinsert_last_node** functions.

Here's a summary of the relevant functions and structures:

- **DlistNode** structure: Represents a node in the doubly linked list.
- **init** function: Initializes a **DlistNode** to set up an empty doubly linked list.
- **display** function: Prints the contents of the doubly linked list in the format **<--- rlink | data | llink --->**.
- **dinsert_node** function: Inserts a new **DlistNode** at the specified position within the list.
- **dremove_node** function: Removes a **DlistNode** from the list.

print

```
<---| 6b36f468 | 9 | 1259240 | --->
<---| 1259260 | 4 | 1259220 | --->
<---| 1259240 | 3 | 1259200 | --->
<---| 1259220 | 2 | 12591e0 | --->
<---| 1259200 | 1 | 12591c0 | --->
<---| 12591e0 | 0 | 1259280 | --->
<---| 12591c0 | 7 | 6b36f468 | --->
```

#2

Code info

This code takes two simple linked lists as input and combines them into a single simple linked list while ensuring that the nodes are arranged in ascending order based on their data values. The following structures and functions are described in the same order as they appear in the code:

1. typedef Element and ListNode structures are defined.
2. The init function initializes a ListNode.
3. The display function prints the simple linked list connected via ListNode.
4. The insert_last function adds a new node to the bottom of the simple linked list.
5. The make_ListNodeC function compares the sizes of node data and creates a new simple linked list C.

A. While loop: This function continues until both A and B have been completely traversed and become NULL.

B. if (A == NULL || B == NULL): If A or B is NULL, it means there are no more remaining nodes in the respective linked list. When A is NULL, the next node from B is added to C, and when B is NULL, the next node from A is added to C.

C. Else: The smaller data between A and B becomes the next data in C.

6. Main:

A. ListNode A, B, and C are allocated dynamic memory, and they are initialized using the init function.

B. For loop and the insert_last function are used to populate ListNode A and B with nodes.

C. The make_ListNodeC function is used to create ListNode C.

D. The display function is used to print ListNode A, B, and C.

Time Complexity:

1. insert_last function: Assuming the lengths of the two lists (A and B) are m and n, respectively, adding elements one by one to each list has a time complexity of O(m + n).
2. make_ListNodeC function: Since all elements in lists A and B are compared and added only once, the time complexity is also O(m + n).

Therefore, the upper bound of the time complexity for this code is O(m + n).

print

A = 1->2->5->10->15->20->25->

B = 3->7->8->15->18->30->

C = 1->2->3->5->7->8->10->15->15->18->20->25->30->

3

Code info

The code you've described is a C program that uses a custom ListType structure to manage a simple linked list. It provides functions for adding, removing, and manipulating nodes in the list. Here's an English explanation of the key components and functions in the code:

Definitions:

element, ListNode, and ListType are defined.

init Function:

Initializes a ListType structure, setting the length to 0 and initializing both head and tail pointers to NULL.

is_empty Function:

Returns 1 if the ListType is empty (head is NULL), otherwise returns 0.

get_node_at Function:

Returns the node at a specified position in the list (0-based index).

insert_node Function:

Inserts a new node after a specified node p. If the list is empty, it sets the new node as the head.

Provides error handling if the list is empty or if p is NULL.

add Function:

Creates a new node with the given data and inserts it into the list at the specified position.

It checks if the position is within bounds (0 to current length) and uses get_node_at to find where to insert the new node.

add_first Function:

Adds a new node at the beginning of the list. If the list is empty, it becomes the head.

add_last Function:

Adds a new node at the end of the list. If the list is empty, it becomes the head.

remove_node Function:

Removes a node from the list. It takes care of updating pointers to maintain the linked list structure.

delete Function:

Deletes a node at the specified position in the list. It internally uses the remove_node function.

delete_first Function:

Deletes the first node in the list.

delete_last Function:

Deletes the last node in the list.

is_in_list Function:

Checks if a specific element exists in the list. It returns true if found and false otherwise.

get_entry Function:

Returns the data of the node at the specified position in the list.

display Function:

Prints the values of all nodes in the list.

main Function:

Initializes list1.

Allocates and inserts nodes into list1.

Deletes a specific node.

Checks if an element is in list1.

Retrieves the value of a node at a specific position.

print
(10 ->20 ->70 ->30 ->40 ->)
(20 ->30 ->)
TRUE
20