Technical report

[Data structure] Homework 03 – List

2071008 Kim seo yeon

[Homework3_1 Merge]

HW03- List

1. dinsert_node can be used even when inserting a new node in the beginning or at the end of in (circular) doubly Linked list

In case of simple Linked list, the process of insertion is slightly different depending on where the elements are going to be put. (first. last, middle)

It is because of the existence of head `pointer´.

head pointer is also regarded as the member of list.

Head pointer always contain the address of first node.

So Head pointer(pointing first node) should be changed when insertion at first

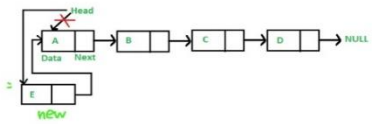
And another reason is that only the last node's link have to be NULL

We determine whether we access to last node or not

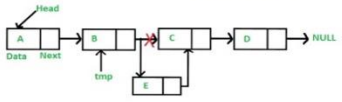
by checking the link of node is Null.

there are 3 situations that can occur when inserting a node.

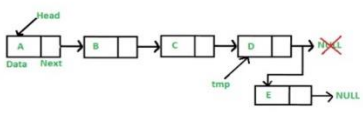
1. insertion at first : The Head should be changed to new node.
The new node have to be connected to next node



2. insertion in middle : There's no need to change the head.
The new node have to be connected to next node.



3. insertion in last : There's no need to change the head.
The link of new node have to be NULL
to indicate that we arrive at last.



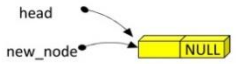
As you can see, in simple Linked list, the process of insertion at each case is unique

So, Insertion can not be generalized to one method in simple Linked list.

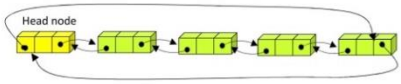
But. In case of circular doubly linked list, the insertion process can be generalized into one method.

It is because there exists head node which has no actual data at the very front of doubly linked list to simplify insertion and deletion

<simple linked list>



<circular doubly linked list>



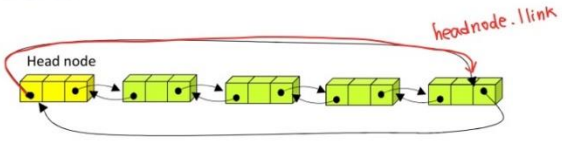
Let's assume you want to insert new node at first in both simple and doubly.

In simple linked list, head pointer should point to new node. but, unlike before, in doubly linked list, we can add new just by connecting new node to head node (the value changes)

The same goes for the insertion at last. In summary, no matter which place (first, middle, last) you want to insert node, the insertion of all cases can be unified into interaction between just node and node.

the practical way to insert both first and last using dinsert_node is as below

```
// Insert 'new_node' into the right of 'before'
void dinsert_node(DlistNode* before, DlistNode* new_node)
{
    new_node->llink = before;
    new_node->rlink = before->rlink;
    before->rlink->llink = new_node;
    before->rlink = new_node;
}
```



The first parameter (DlistNode * before) delivered into function can be different.

1. dinsert_first

```
dinsert_node(&head_node,new_node_first);
```

the address of head node

new node that you want to put in the very front

: when you want to insert node at first, just insert new_node into the right of 'head node'

2. dinsert_last

```
dinsert_node(head_node.llink,new_node_last);
```

the head node's llink

new node that you want to put in the very last

: head_node.llink is the same as the reference to last node.

so this function call means connecting new node next to last node.

[Homework3_2　Merge]

Variable analysis

Before we merge two list, we must create the list.

To make list, we need two structure that store the information about list.

One is ListType and the other is ListNode

Allocating ListType in the main function forms the following data.

List Type	
type	**name**
ListNode*	head
ListNode*	tail
int	length

: List Type stores overall information of the list.

Head, tail, length each contains the address of first node, the address of last node, the number of node in list.

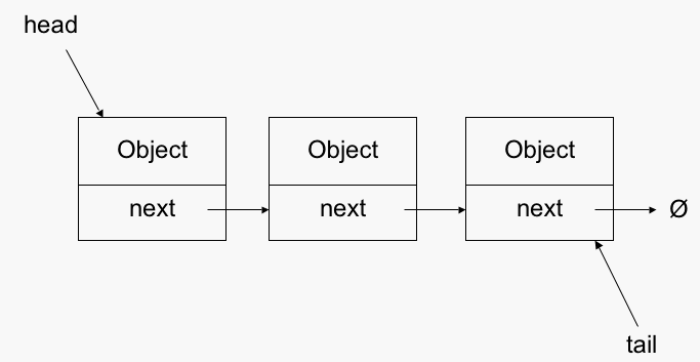
And ListNode, the member of ListType, has information at each node in list.

ListNode	
type	**name**
element	data
ListNode*	link

: ListNode store the information about one node.

data contains the value of this node in element type and link contains the address to next node.

If we allocate List using above two structure, the form of list will be as below



Function analysis

```
ListType a, b, c;
        init(&a);
        init(&b);
        init(&c);
```

: Declare three ListType variable a,b,c . Then, initialize three variables using fuction ‘init’.

```
void init(ListType* list) { //initialize list
		list->length = 0;
		list->head = list->tail = NULL;
}
```

List type a,b,c	
variable	**value**
head	NULL
tail	NULL
length	0

: Then, lists are set to initial state that the list doesn't have any node.

After initialization, we can add node to list using function 'add_last()' and 'inserst_last()'.

```
		int list1[] = { 1,2,5,10,15,20,25 };
		int list2[] = { 3,7,8,15,18,30 };

		for (int i = 0; i < sizeof(list1) / sizeof(int); i++) {
			add_last(&a, list1[i]); //add node with data value of list[i]
		}
		for (int i = 0; i < sizeof(list2) / sizeof(int); i++) {
			add_last(&b, list2[i]); //add node with data value of list[i]

		}
```

: The elements in the array are inserted into the list one by one using function `add_last()`

```
void add_last(ListType* list, element data) {
		ListNode* node = (ListNode*)malloc(sizeof(ListNode)); //create new node
		if (node == NULL) {
			printf("Memory allocation error\n");
			return;
		}
		node->data = data; //data allocation.
		insert_last(list, node); //insert new node at the end of list.
		list->length++; // after inserting new node, the length of list increases by one

}
```

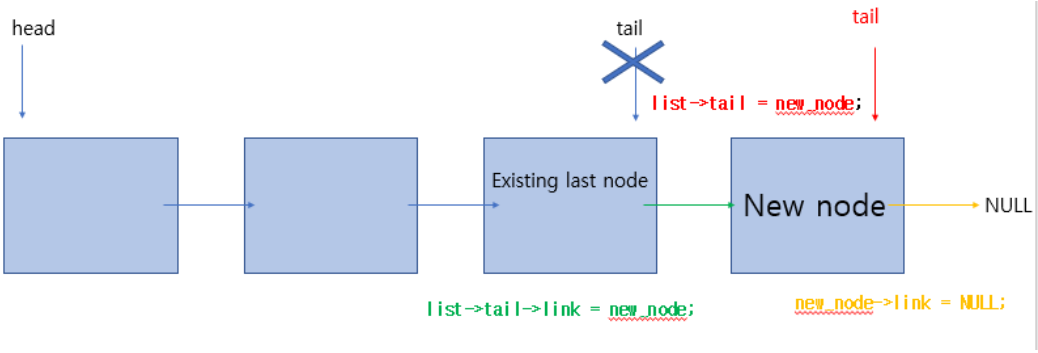
: the function 'add_last()' creates a new node with given data and insert it using function 'insert_last()'.

```
void insert_last(ListType* list, ListNode* new_node) {
		if (list->head == NULL) { //in case of empty list, the head and tail should be renewed
		new_node->link = NULL;
			list->head = new_node;
			list->tail = new_node;
		}
		else {
```

```
                    new_node->link = NULL; // set link of new_node to NULL.
                    list->tail->link = new_node; // change link of existing last node.
                    list->tail = new_node; // tail renewal.

          }
}
```

: In the function insert_last(), it completes total 3 process.



1. Allocate link of new_node to NULL (because it is last node. It has no next node.)
2. Chage link of existing last node into new_node
3. Tail renewal

In this way, we can insert nodes at the end of the list and make list as below.

List a	**1**	**2**	**5**	**10**	**15**	**25**
List b	**3**	**7**	**8**	**15**	**18**	**30**
List c	**No ListNode.**					

```
merge_list(&a, &b, &c);
```

: We now have to merge the two lists(a,b) into one(c) using function merge_list()

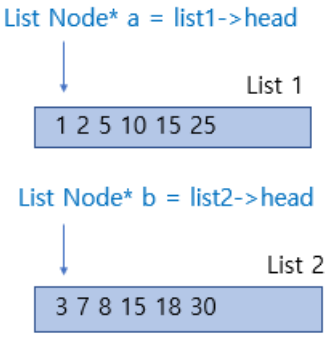
```
void merge_list(ListType* list1, ListType* list2, ListType* result) {
          ListNode* a = list1->head; // make pointer pointing to head of list1
          ListNode* b = list2->head; // make pointer pointing to head of list2

          while (a && b) { //if a or b reaches the end of list, the repetition end.
                    if (a->data < b->data) { //if a->data is smaller than b->data
                              add_last(result, a->data); //add node with a->data into result
                              a = a->link; //move to next node
                    }
                    else { //if b->data is smaller than a->data

                              add_last(result, b->data);//add node with b->data into result
                              b = b->link; //move to next node
                    }
          }
          for (; a != NULL; a = a->link) //insert a remaining data in list a
                    add_last(result, a->data);
```

```
        for (; b != NULL; b = b->link) //insert a remaining data in list b
                add_last(result, b->data);
}
```

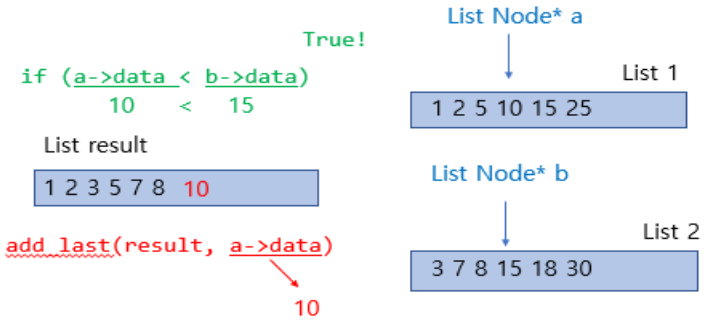
: the process of merge is as below



1. Make pointer pointing to head(first node) of each list. The pointer(List Node *) serves to access all node in list while moving to next node in the list.

2. `while (a && b)`

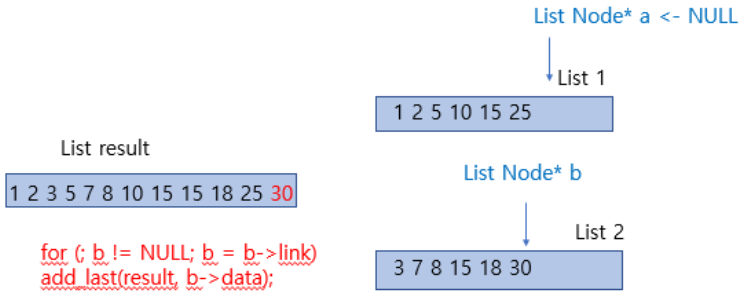
: Perform the loop only while the two pointers do not point to NULL.



: then, start comparing two value pointed by a and b . The idea is to compare the current node of both the lists and add a node with smaller value to the output list

(Because we want to sort result in ascending order, we insert a smaller value among the two .)

3. when a or b points to NULL , which means one of the lists end inserting, we just copy remaining node into result.



: If we say the length of two lists to be merged is M,N , we can simply merge two list with maximum

M+N-1 comparisons. It is because the two list to be merged are already in ascending order. We can simply know what is the smallest value in both list by just comparing the values at the beginning of the each list.

```
printf("c :"); display(&c);
```

: Finally display the result using function 'display()'

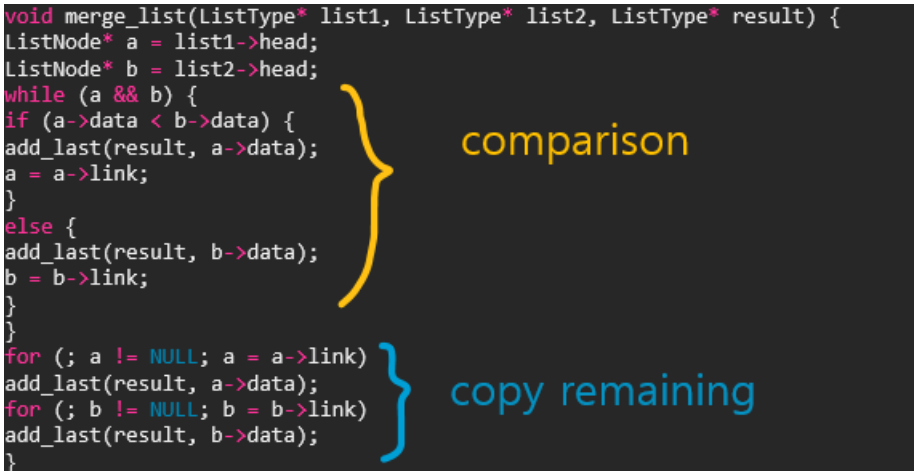
```
void display(ListType* list) { //staring from head to end of list, print out data
        ListNode* node = list->head; // pointer needed to access node's data
        printf("{ ");
        for (int i = 0; i < list->length; i++) {
                printf("%d ", node->data); // access data by pointer
                node = node->link; //move to next node
        }
        printf(" }\n");
}
```

```
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  }
```

---

analyze the time complexity of merge()

we can calculate the time complexity of merge() in terms of two process : comparison , copy remaining



Let's say we only consider the time complexity of comparison. In the worst case, we are traversing through the two lists fully and all elements except the last are compared against another element and moved into merged list, there are at most (m + n - 1) comparisons where m and n are the lengths of the two lists to be merged.

Let's consider the best-case complexity of determining the correct order of the elements in two sorted

sets (without worrying about how they are stored). If the last element in the first set is smaller than the first element in the second set, only a single comparison is needed to determine the correct order. The second set just needs to be "tacked on" to the end of the first set. (Most implementations would start from the front of both sets, but you could write this as a "special case" check at the beginning of the sort.) Since this case takes only a single comparison, regardless of the number of elements in the arrays, determining the correct order in the best-case is O(1).

Now, back to the actual problem where the elements have to be moved into the list.

In addition to comparison , it guaranteed to perform at least m + n operations because we have to add all the nodes into a result list. Even if the comparison ends within O(1), the remaining nodes in list elements still have to be copied to the result. So, while comparison could be done in a constant number of comparisons, the amount of work to move the nodes still yields O(n) in the best case.

In short, Regardless of cases(Best, worst, average), Add_last() is always performed n+m times in total. (inserting by comparison + copy remaining = m+n) So the upperbound is **O(m+n)** and tight bound is also **θ(m+n).**

Answer : O(m+n) , θ(m+n).

.

[Homework 3_3 ListADT]

Variable analysis

Same as 3_2 Merge

Function analysis

1. init

```
void init(ListType* list) { //allocate list as initial state
         list->head = NULL;
         list->tail = NULL;
         list->length = 0;
}
```

:ListType is a structure in which store node of the head and tail ,lenghth of a linked list. Therefore, after declaring the ListType variable, the value must be initialized using this function

2. get_node_at

```
ListNode* get_node_at(ListType* list, int pos) {
	ListNode* tmp_node = list->head; //the pointer using when traversing the list

	if (pos < 0) return NULL; //if pos<0, the corresponding node doesn't exist
	for (int i = 0; i < pos; i++) {
		tmp_node = tmp_node->link; //move to next node until arriving at pos
	}
	return tmp_node;
}
```

:Return the node located in the corresponding pos in the linked list. By using pointer(`ListNode*`), it approaches the node in list sequentially. when executing the loop as much as `pos`, the pointer arrive at the node at position and the node (corresponding pos) comes out.

3. insert_node

```
void insert_node(ListType* list, ListNode* before, ListNode* new_node) {
	if (list->head == NULL) {
		new_node->link = NULL;
		list->head = new_node;
		list->tail = new_node;
	}
	else {
		if (before == NULL) error((char*)"The preceding node cannot be NULL");
		else {

			new_node->link = before->link; //connect new node to next node
			before->link = new_node; //connect before_node to new node
		}
	}
}
```

: when insert node, we need total 2 process.

1. Connect new_node to next node.

   New_node ->link = next node

   The address of next node of the inserted is saved at `before -> link`. So it can be substitued as below.

   New node ->link = before ->link

2. Connect before node to new node.

   Precaution ; the order of two statement can't be changed because we have to make use of `before ->link` as the address of next node of the inserted . we use that value before `before ->link` is updated

4. add_first(ListType *list, element data)

```
void add_first(ListType* list, element data) {

	//create new node with given data
```

```
        ListNode* new_node = (ListNode*)malloc(sizeof(ListNode));
        new_node->data = data;

        if (list->head == NULL) { //when no node exists in the list
                list->head = list->tail = new_node; //only new node exist in the list.
                new_node->link = NULL;

        }
        else {
                new_node->link = list->head; //connect new node to next node
                list->head = new_node; //head renewal

        }
        list->length++;
}
```

:The head is used because we want to insert at the beginning of the linked list.

Fistly, create a new node with given data.

Second, insert it at the beginning of the list.

When the head is NULL: only new node exist in the list. So both head and tail are set to new_node.
When the head is not NULL: `new_node -> link` should be changed into the address of next node. The address of next node is stored at `list -> head`. After reconnection, assign new node to head

5. add_last(ListType *list, element data)

```
void add_last(ListType* list, element data) {
        ListNode* new_node = (ListNode*)malloc(sizeof(ListNode));
        new_node->data = data;
        new_node->link = NULL; //the last node has no next node

        if (list->tail == NULL) { //when no node exists in the list
                list->head = list->tail = new_node; // only new node exist in the list.

        }
        else {
                list->tail->link = new_node; //conncect the before to new node
                list->tail = new_node; //tail renewal
        }
        list->length++;
}
```

:The tail is used because we want to insert at the end of the linked list.

Fistly, create a new node with given data and link of NULL.

Second, insert it at the end of the list.

When the tail is NULL: only new node exists in the list. So, both head and tail are set to new_node.
When the head is not NULL: the existing(previous) last node should be connected to new

node(new last). List->tail = the existing last node. So change the link of the existing last node into new node. After connection, assign new node to tail.

6.add_at

```
void add_at(ListType* list, int position, element data) {
	ListNode* p;

	if ((position >= 1) && (position <= list->length - 1)) { // if you want locate the node in middle

		// make new node with given data
		ListNode* node = (ListNode*)malloc(sizeof(ListNode));
		if (node == NULL) error((char*)"Memory allocation error");
		node->data = data;

		p = get_node_at(list, position - 1); // before node of node at  position
		insert_node(list, p, node);
		list->length++;
	}
	else if (position == 0) { // if you want insert node at first position
		add_first(list, data);
	}
	else if (position == list->length) { // if you want insert node at last position
		add_last(list, data);
	}
}
```

: When insert in middle , Insert the node to the desired position using function insert_node(). At this time, find the node at `position-1` using function `get_node_at()` to find the before node of the inserted. The reason why we find before node is that the function `insert_node` requires the before node of the inserted. If the position value is zero :position = -1 is not possible at get_node_at(). So,add_first() is used when position equals 0 and add_last() is used when position equals length.

7. remove_node

```
void remove_node(ListType* list, ListNode* before, ListNode* removed) {
	if (list->head == NULL) {
		printf("The list is blank.\n");
	}
	else {
		if (before == NULL) printf("The preceding node cannot be NULL.\n");
		else {
			before ->link = removed->link; // connect before node to the next node of the removed
			free(removed);
		}
	}
}
```

: when removing node, we need not only the removed but also the before node of it. The this is because we have to reconnect before node to after node of the removed.

Then, we needs only one process.

before->link = the address of next node of the removed

The address of next node of the removed is stored at `removed->link`. So it can be substituted as below

before->link = removed->link.

And finally, deallocate the memory on the removed.

8. delete_first

```
void delete_first(ListType* list) {
	if (!is_empty(list)) {
		ListNode* removed = list->head; // find first node
		list->head = removed->link; // head renewal
		free(removed);
		list->length--;
	}
}
```

:

9.delete_last

```
void delete_last(ListType* list) {
	if (!is_empty(list)) {
		ListNode* removed = list->tail; // find last node
		ListNode* p = get_node_at(list, list->length - 2); // find before node
of the last node
		list->tail = p; // tail renewal
		p->link = NULL; // disconnect
		free(removed);
		list->length--;
	}
}
```

10. delete_at

```
void delete_at(ListType* list, int pos) {
	if (!is_empty(list) && (pos >= 1) && (pos < list->length - 1)) {
		ListNode* p = get_node_at(list, pos - 1); //find before node of the
removed
		ListNode* removed = get_node_at(list, pos);
		remove_node(list, p, removed);
		list->length--;
	}
	else if (!is_empty(list) && pos == 0) {//if you want remove at first
		delete_first(list);
	}
	else if (!is_empty(list) && pos == list->length - 1) { //if you want remove at
last
		delete_last(list);
	}
}
```

: it removes node in the desired position using function remove_node(). When you want to remove node in the middle of the list , find the node at the `position-1` using `get_node_at()` to find the before node of the removed. The reason why we find before node is that the function `remove_node` requires the before node of the removed. if the position equals 0, however, pos-1 is impossible in get_node_at(). So

in that case, use the delete_first() function instead. Likewise, if the position value equals length-1(=want to remove last node), the tail is not renewed when using `remove_node()` only. so, in that case, use the delete_last() function instead.

11. display

```
void display(ListType* list) {
	ListNode* node = list->head; //start from first node
	printf("(");
	for (int i = 0; i < list->length; i++) {
		printf("%d ", node->data);  //print data in current node
		node = node->link; // move to next node
	}
	printf(")\n");
}
```

:To display the list, we need a node pointer used to traverse the list. Print data in current node while it approaches the all node sequentially. When executing the loop as much as `length of the list`, the pointer arriving at last node and print the last data and the loop is overed.

12. is_in_list

```
bool is_in_list(ListType* list, element item) {
	ListNode* p;
	p = list->head; //start from first node
	while ((p != NULL)) {
		if (p->data == item) //if find the node with given item
			break; //out the loop
		p = p->link; //move to next node
	}
	if (p == NULL)return false;
	else return true;
}
```

: It is a function of checking whether there is an item in the list. To find the data in list, we need a node pointer used to traverse the list. So, we declare a node pointer `p`. To find data starting from the first node, assign p to the head. p is continuously updated to the next node during the iteration with checking whether p has given item. If match, return true.

13. get_entry

```
element get_entry(ListType* list, int pos) {
	ListNode* p;
	if (pos >= list->length)error("position error");
	p = get_node_at(list, pos);
	return p->data;
}
```

it returns the data value in given position. it may be confused with get_node_at().
the difference between get_node_at() and get_entry() is a return type. get_node_at() returns the node pointer and get_entry() returns element at pos.

List ADT can be implemented using functions above.

Result