|  |
| --- |
|  |
| **Technical report** |
| **[Data structure] Homework 03 – List** |

**2071008 Kim seo yeon**

**[Homework3\_1 Merge]**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

**[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.

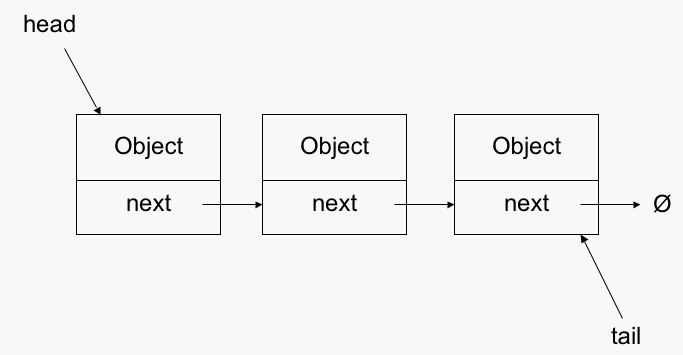
And ListNode, the member of ListType, has information at each node in list.

|  |  |
| --- | --- |
| **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.

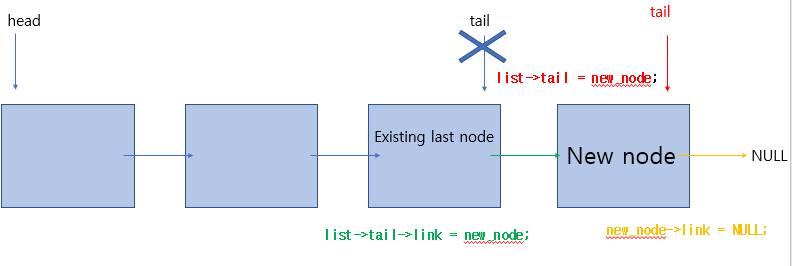
list->tail->link = new\_node; // change link of existing last node.

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

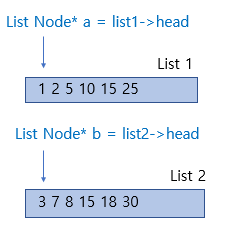
add\_last(result, a->data);

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)

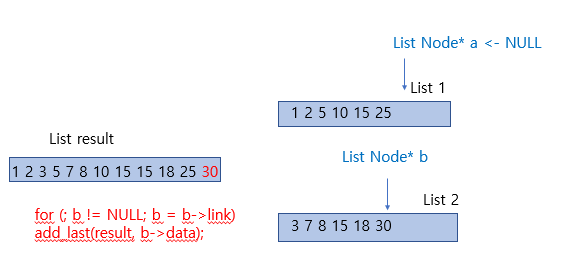
: Perform the loop only while the two pointers do not point to NULL.

텍스트이(가) 표시된 사진

자동 생성된 설명: 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 .)

1. 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");

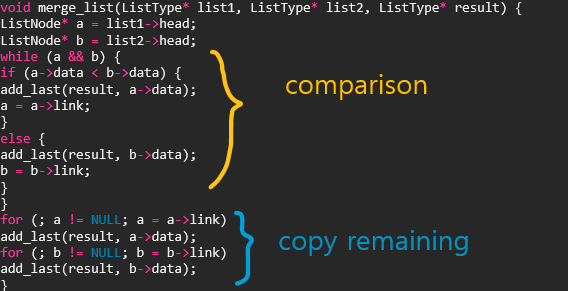
}

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명

**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

1. 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**

**텍스트이(가) 표시된 사진

자동 생성된 설명**