**Data Structure\_2071035 Lee Somin**

**Technical Report – ListADT.cpp**

*Theorical Explanation of Functions in ‘ListADT.cpp’*

**#define TURE, #define FALSE**

This definition is for using Boolean form when judging whether element is in the list or not, since C++ doesn’t contain Boolean library itself.

**typedef element**

This typedef defines the data type of the elements of list.

**typedef struct ListNode**

This structure contains the actual information of node, the data and the link.

**typedef struct ListType**

This structure contains the pointer to the head of the list, pointer to the tail of the list, and the length of the list. This is for accessing to the node of head and tail easily, and for determining a list from another.

**error**

Input: const char\* message

Return: non

This function is for printing out the error message and halting the program after error.

**init**

Input: ListType\* list

Return: non

This function initializes the list delivered by input address. The initialization occurs by putting NULL to the headpointer and the tailpointer of the list and setting the length of the list to 0.

**isempty**

input: ListType\* list

Return: int 0 / int 1

This function determines whether the list is empty by looking at the headpointer of the list. If the list is empty, the headpointer would point to NULL.

**get\_length**

input: ListType\* list

return: list -> length

This function returns the length of the given list.

**get\_node\_at**

input: ListType\* list, int pos

return: ListNode\* tmp\_node / ListNode\* NULL

This function follows the list by for statement by updating the tmp\_node by its link. When it is repeated with the number of ‘pos’, the function returns the present ‘tmp\_node’ which is the node of ‘pos’th location in the list. When the position input gets out of the range of non-negative integer, the function returns NULL because there is no node in the location.

**insert\_node**

input: Listnode\*\* phead, ListNode\* p, ListNode\* new\_node

return: non

This function is made to insert the ‘new\_node’ after the preceding node ‘p’. When the headpointer ‘\*phead’ is pointing to NULL, the list is empty, so new node should be the last node and its link is updated to NULL. Then, the headpointer is updated to the address of the ‘new\_node’. When the list has elements, there are two cases. One is that given preceding node is NULL. In this case, the ‘new\_node’ should be inserted to the head of the list, so its link is updated to the address of original first node, and the headpointer is updated to the address of the new node. In second case, there is no special feature, so new node’s link is updated to the link of preceding node, and the link of preceding node is updated to new node.

**add**

input: ListType\* list, int position, element data

return: non

This function is made to inert the new node into the location wanted in input ‘int position’. When the position is within the length of the list, it allocates memory to a new node and assigns the data to ‘node->data’. ‘p’ is initialized with the preceding node from ‘position’, by ‘get\_node\_at()’. Then, the node is inserted to the position through ‘insert\_node’ and the length of the list increases.

**add\_first**

input: ListType\* list, element data

return: non

This function is for inserting the new node at the beginning of the list. When the memory is allocated to the new node, the data of the ‘node’ is updated to the input ‘element data’. When the tail of the list is NULL, it means that the list is empty, so the function updates the tail to the new node created in the function. Then, it inserts node at the beginning of the list by giving the preceding node input as NULL to ‘insert\_node()’. Then the length of the list increases.

**add\_last**

input: ListType\* list, element data

return: non

This function is for inserting a new node at the end of the list. When the memory is allocated to the new node, and if there is no memory allocation error, the data of the ‘node’ is updated to the input ‘element data’. Then the new node is added to the last of the list and the list’s tail is updated to the new node since the new node is now the last node. Then the length of the list increases.

**remove\_node**

input: ListNode\*\* phead, ListNode\* p, ListNode\* removed

return: non

This function was made to remove the node after the preceding node ‘p’. If the head pointer if pointing to NULL, the list is blank, so print the error message. Otherwise, if the preceding node is NULL, it means the node to be removed is the very first node of the list. So, it updates the headpointer of the list to the link of the removed node. Then the function deallocates the memory of removed node. If the preceding node is a general node, the link of the preceding node is updated to the link of removed node, and the memory is deallocated from the removed node.

**delete\_node (originally, ‘delete’, but it was in the preserved statement of C++ )**

input: ListType\* list, int pos

return: non

This function is for deleting the node in the position ‘int pos’. When the list is not empty and the location of the node wanted to be deleted is within the length of the list, the function finds the preceding node by using the ‘get\_node\_at()’. Then, it gets the node that should be removed in similar way. Finally, it removes node with ‘remove\_node()’ and the length of the list decreases.

**delete\_first**

input: ListType\* list

return: non

This function is for deleting the first node of the list. If the list length is 1, the deletion would make it to blank list. So, the tail is updated to NULL. Then, the first node is removed by delivering NULL to ‘remove\_node()’ as the preceding node of node to be removed. Then the length of the list is decreased by 1.

**delete\_last**

input: ListType\* list

return: non

This function is for deleting the last node of the list. The function finds the preceding node of tail by ‘get\_node\_at()’ and removes the last node by ‘remove\_node()’. Then the address of the tail is updated with the preceding node. Finally, the length of the list decreases.

**get\_entry**

input: ListType\* list, int pos

return: element p->data

When the position ‘int pos’ goes out of range of list length, the function prints out the error message. Otherwise, it gets node in position with ‘get\_node\_at()’ and returns the data of it.

**display**

input: ListType\* list

return: non

This function follows the nodes from the head and prints out the data of the list by for statement until the length of the list.

**is\_in\_list**

input: ListType\* list, element item

return: int TRUE / int FALSE

This function is for determining whether the ‘element item’ exists in the given list. After ‘p’ is initialized to the head of the list, the while statement loops and follows the links until the data was found, or until the end of the list was met. If p is NULL(when the data was not in the list), FALSE is returned, and if the data was found, TRUE is returned.

**main()**

In main function, list1 is defined and initialized by ‘init()’, and the insertion of 20 at first, 30 at last, 10 at first, 40 at last, and 70 at the position2 is occurred sequentially. Then the list is displayed, and the deletion of node at position 2, at first, at last is executed sequentially. Then the list is displayed. Finally it determines if data 20 is in the list, prints the return, and prints the 0th node’s data.