Linked List

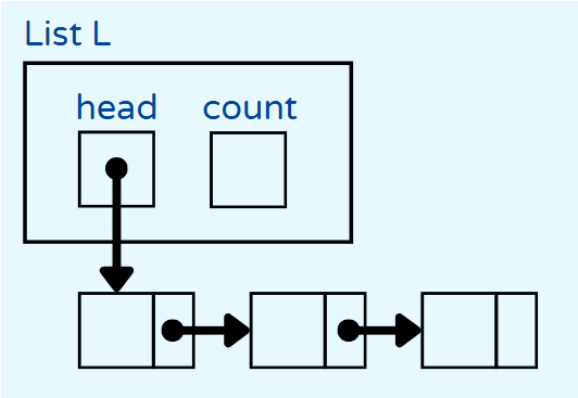
There are 2 variations:

1. **List** is a **Linked List** and accessed by **value**.
2. **List** is a **Linked List** and accessed by **pointer**.

The main difference between the two variations is how the list is handled. In **Variation 1**, the list is **passed by value** (a copy is made), and the modified list must be returned by the function. In **Variation 2**, the list is **passed by reference**, allowing the function to modify the original list directly without needing to return it.

Variation 2

List is a **Linked List** and accessed by **pointer**.




```
typedef struct node {
    int data
    struct node *next;
} Node;

typedef struct {
    Node  *head;
    int count;
} List;
```

Operations	Checklist	Example
`List* initialize();`	☐ Allocate memory for a new List struct using malloc ☐ If the allocation was unsuccessful, return NULL ☐ Set the head pointer of the List to NULL ☐ Set the count of the List to 0 ☐ Return the pointer to the newly created List	`List *L = initialize();`

`void empty(List *list);`	☐ Iterate through the nodes and free each one ☐ Set head to NULL ☐ Set count to 0	

`void insertFirst(List *list, int data);`	☐ Allocate memory for a new node
	☐ Set the data of the new node to the provided data
	☐ Set the next pointer of the new node to the current head of the list
	☐ Update the list's head pointer to point to the new node
	☐ Increment the list's count
		`Before:`
		`head: 2 -> 6 -> 5 -> NULL`
		`count: 3`
		
		`insertFirst(list, 7);`
		
		`After:`
		`head: 7 -> 2 -> 6 -> 5 -> NULL`
		`count: 4`

`void insertLast(List *list, int data);`	☐ Allocate memory for a new node
	☐ Set the data of the new node to the provided data
	☐ Set the next pointer of the new node to NULL
	☐ If the list's head is NULL (the list is empty), set the head to the new node
	☐ If the list is not empty, create a “current” pointer and initialize it with the head
	☐ Traverse the list until current->next is NULL
	☐ Set current->next to the new node
	☐ Increment the list's count
		`Before:`
		`head: 2 -> 6 -> 5 -> NULL`
		`count: 3`
		
		`insertLast(list, 7);`
		
		`After:`
		`head: 2 -> 6 -> 5 -> 7 -> NULL`
		`count: 4`

`void insertPos(List *list, int data, int index);`	☐ Index must be valid (less than list->count)
	☐ If index is 0, call insertFirst()
	☐ If index is equal to list->count, call insertLast()
	☐ Otherwise, allocate memory for a new Node
	☐ Set the data of the new node
	☐ Create a “current” pointer and initialize it
		`Before:`
		`head: 2 -> 6 -> 5 -> NULL`
		`count: 3`
		
		`insertPos(list, 7, 2);`
		
		`After:`
		`head: 2 -> 6 -> 7 -> 5 -> NULL`
		`count: 4`

	to the head ☐ Iterate *index - 1* times to find the node just before the insertion point ☐ Set the new node's next pointer to current->next ☐ Set current->next to the new node ☐ Increment the list's count	

`void deleteStart(List *list);`	☐ Create a “current” pointer and set it to the head
	☐ Update the list's head to current->next
	☐ Free the memory for current
	☐ Decrement the list's count
		`Before:`
		`head: 2 -> 6 -> 5 -> NULL`
		`count: 3`
		
		`deleteStart(list);`
		
		`After:`
		`head: 6 -> 5 -> NULL`
		`count: 2`

`void deleteLast(List *list);`	☐ If the list has only one node, free the head, set the head to NULL, and decrement the count
	☐ Otherwise, create a “current” pointer and initialize it to the list's head
	☐ Use a loop that runs *list->count - 2* times to place current at the second to the last node
	☐ Free the memory of the last node
	☐ Set current->next to NULL
	☐ Decrement the list's count
		`Before:`
		`head: 2 -> 6 -> 5 -> NULL`
		`count: 3`
		
		`deleteLast(list);`
		
		`After:`
		`head: 2 -> 6 -> NULL`
		`count: 2`

`void deletePos(List *list, int index);`	☐ If index is 0, call removeStart()
	☐ Otherwise, create a “current” pointer, and initialize it to the head
	☐ Iterate *index - 1* times to find the node just before the one to be removed
	☐ Create a “temp” pointer and set it to current->next
	☐ Set current->next to temp->next
		`Before:`
		`head: 2 -> 6 -> 5 -> NULL`
		`count: 3`
		
		`deletePos(list, 1);`
		
		`After:`
		`head: 2 -> 5 -> NULL`

	☐ Free the memory for temp ☐ Decrement the list's count	`count: 2`
`int retrieve(List *list, int index);`	☐ Index must be valid (less than list->count) ☐ Create a “current” pointer and initialize it to the head ☐ Iterate *index* times, moving current forward ☐ Return the data from current	
`int locate(List *list, int data);`	☐ If the list's head is NULL, return -1 ☐ Create a “current” pointer and initialize it to the head ☐ Initialize an integer variable “index” to 0 ☐ Iterate as long as current is not NULL ☐ If there's a match, return index ☐ If there's no match, move current to its next node and increment index by 1 ☐ Return the index or -1 if not found	
`void display(List *list);`	☐ Create a “current” pointer and initialize it to the head ☐ Begin a loop that continues as long as current is not NULL ☐ Inside the loop, print the data from current ☐ Move current to current->next	

Note:

For most operations, such as **insert** and **delete**, it is also common to return a **boolean value** representing whether the operation is successful or not.