```c
#include <stdbool.h>
#include <stdlib.h>
#include "dry_tester_h.h"

//checks if new node is not null.
#define CHECK_NULL_ARGUMENT \
    do { \
        if (new_node == NULL) { \
            *merged_out = NULL; \
            return MEMORY_ERROR; \
        } \
    } \
    while(0)
//adds the new node to the merged list and moves the pointer to the next
element
#define ADD_AND_UPDATE_PTR(ptr) \
    do { \
        if (addNode(new_node, current_node, ptr->x) != SUCCESS) { \
            return NULL_ARGUMENT; \
        } \
        current_node = new_node; \
        ptr = ptr->next; \
    } \
    while(0)


/*************************************************************************
typedef struct node_t {
    int x;
    struct node_t *next;
} *Node;

typedef enum {
    SUCCESS=0,
    MEMORY_ERROR,
    EMPTY_LIST,
    UNSORTED_LIST,
    NULL_ARGUMENT,
} ErrorCode;

int getListLength(Node list); // not used
bool isListSorted(Node list);
ErrorCode mergeSortedLists(Node list1, Node list2, Node *merged_out);

/************************************************************************

// creates a new node
Node createNode() {
    Node ptr = malloc(sizeof(*ptr));
    if(!ptr) {
        return NULL;
    }
    ptr->x = 0;
    ptr->next = NULL;
    return ptr;
}
```

```c
// add a new node to the last node in the merged list + updates the number
of the new node.
ErrorCode addNode(Node new_node, Node last_node, int x) {
    if (last_node == NULL || new_node == NULL) {
        return NULL_ARGUMENT;
    }
    new_node->x = x;
    last_node->next = new_node;
    return SUCCESS;
}

ErrorCode mergeSortedLists(Node list1, Node list2, Node *merged_out)
{
    if(list1 == NULL || list2 == NULL){
        return EMPTY_LIST;
    }
    if(!isListSorted(list1) || !isListSorted(list2)){
        return UNSORTED_LIST;
    }
    Node list1_ptr = list1;
    Node list2_ptr = list2;
    *merged_out = createNode();
    if (*merged_out == NULL) {
        return MEMORY_ERROR;
    }
    if(list1_ptr->x <= list2_ptr->x) { //first element insertion
        (*merged_out)->x = list1_ptr->x;
        list1_ptr = list1_ptr->next;
    }
    else {
        (*merged_out)->x  = list2_ptr->x;
        list2_ptr = list2_ptr->next;
    }
    Node current_node = *merged_out;
    while(list1_ptr && list2_ptr) {
        Node new_node = createNode();
        CHECK_NULL_ARGUMENT;
        if (list1_ptr->x <= list2_ptr->x) {
            ADD_AND_UPDATE_PTR(list1_ptr);
        } else {
            ADD_AND_UPDATE_PTR(list2_ptr);
        }
    }
    while(list1_ptr != NULL) {
        Node new_node = createNode();
        ADD_AND_UPDATE_PTR(list1_ptr);
    }
    while(list2_ptr != NULL) {
        Node new_node = createNode();
        ADD_AND_UPDATE_PTR(list2_ptr);
    }
    return SUCCESS;
}
```