Merge Sorted Nodes:

ErrorCode nodeDestroy(Node to\_destroy)

{

    if(to\_destroy == NULL)

    {

        return SUCCESS;

    }

    Node tmp;

    while(to\_destroy != NULL)

    {

        tmp = to\_destroy;

        to\_destroy = to\_destroy->next;

        free(tmp);

    }

    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;

    }

    if(merged\_out == NULL)

    {

        return NULL\_ARGUMENT;

    }

    Node\* starter\_merged = merged\_out;

    int len1 = getListLength(list1);

    int len2 = getListLength(list2);

    while(len1 > 0 && len2 > 0)

    {

        assert(list1 != NULL && list2 != NULL);

        (\*merged\_out)->next = malloc(sizeof(\*(\*merged\_out)));

        if((\*merged\_out)->next == NULL)

        {

            nodeDestroy(\*starter\_merged);

            return MEMORY\_ERROR;

        }

        if(list1->x >= list2->x)

        {

            (\*merged\_out)->x = list2->x;

            len2--;

            list2 = list2->next;

            merged\_out = &((\*merged\_out)->next);

        }

        else if(list1->x < list2->x)

        {

            (\*merged\_out)->x = list1->x;

            len1--;

            list1 = list1->next;

            merged\_out = &((\*merged\_out)->next);

        }

    }

    while(len1 > 0)

    {

            assert(list1 != NULL);

            (\*merged\_out)->x = list1->x;

            len1--;

            list1 = list1->next;

            if(len1 > 0)

            {

                (\*merged\_out)->next = malloc(sizeof(\*(\*merged\_out)));

                if((\*merged\_out)->next == NULL)

                {

                    nodeDestroy(\*starter\_merged);

                    return MEMORY\_ERROR;

                }

                merged\_out = &((\*merged\_out)->next);

            }

    }

    while(len2 > 0)

    {

        assert(list2 != NULL);

        (\*merged\_out)->x = list2->x;

        len2--;

        list2 = list2->next;

        if(len2 > 0)

        {

            (\*merged\_out)->next = malloc(sizeof(\*(\*merged\_out)));

            if((\*merged\_out)->next == NULL)

            {

                nodeDestroy(\*starter\_merged);

                return MEMORY\_ERROR;

            }

            merged\_out = &((\*merged\_out)->next);

        }

    }

    (\*merged\_out)->next = NULL;

    return SUCCESS;

}

Convention & Coding errors:

char \*stringDuplicator(char \*s, int times) { //Convention error - unconventional "s" shortcut  for string/str

                           //Convention error - bad function name - it is not a verb.

                                  //Coding error - s must be of type const char\*

    assert(!s);

    assert(times > 0);

    int LEN = strlen(s); //Convention error - variable name is in capital letters - "LEN"

    char \*out = malloc(LEN \* times);

    assert(out);

//Coding error - memory allocation must be checked during runtime and not as an assert

    for (int i = 0; i < times; i++) {

    out = out + LEN; //Coding error - does not start at the initial 0th index, leaving behind  empty memory cells

    strcpy(out, s); //Coding error - at the last loop, will encounter a segmentation error

for leaving the bounds of the malloc

                    //Convention error - no indentation!

    }

    return out; //Coding error - returns wrong address to string, only returns the last

duplication (outside the malloc allocation...)

}

Fixed code:

char \*duplicateString(const char \*str, int times)

{

    assert(!str);

    assert(times > 0);

    int len = strlen(str);

    char \*out = malloc(len \* times);

    if (out == NULL)

    {

        return NULL;

    }

    for (int i = 0; i < times; i++)

    {

        strcpy(out +(i\*len), str);

    }

    return out;

}