**Problem 1 follows…**  
**Description:** Problem 1 is designed to demonstrate the singly linked list. It consists of functions for initializing, deleting, placing at front/end, swapping and removing data from the Singly LinkedList. There are also get front/index and length functions. AllocationCount will tell how many things are currently dynamically allocated.

**Program Output:**  
Starting demo.

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data has been placed at the front of the LL...

There are now 4 items with an allocation count of 5

Head==> [0] = 4000

[1] = 3000

[2] = 2000

Tail==> [3] = 1000

After the LL has been deleted...

There are now 0 items with an allocation count of 0

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data items have been added at the end of the LL...

There are now 4 items with an allocation count of 5

Head==> [0] = 1000

[1] = 2000

[2] = 3000

Tail==> [3] = 4000

After data has been swapped in the LL...

There are now 4 items with an allocation count of 5

Head==> [0] = 4000

[1] = 3000

[2] = 2000

Tail==> [3] = 1000

A data item has been removed from the front of the LL..

the UserData contained 4000

The allocation count is now 4

A data item has been removed from the front of the LL..

the UserData contained 3000

The allocation count is now 3

After data has been removed from the LL..

There are now 2 items with an allocation count of 3

Head==> [0] = 2000

Tail==> [1] = 1000

After the LL has been deleted...

There are now 0 items with an allocation count of 0

**Answers to Questions:**2.

1. LL\_Init is used to initialize a LinkedList, but also allocate storage for it. Therefore, AllocationCount can be referenced. When malloc() allocates space for LLI\_Ptr, AllocationCount will increase by 1 to keep track of how many things are currently dynamically allocated.
2. LL\_AddToFront updates the head of the list and places a node there. First, it assures that the list is not empty. Then it creates a new node and assigns it to the head of the list. If the rest of the list is empty then it is also assigned to the tail of the list. This is unique from LL\_AddAtEnd. Finally, the number of nodes in the list is updated by +1. The AllocationCount equals 5 after the loop because of the fact that making a node also allocates space for the user's data. There are four nodes being created during this loop and AllocationCount was already 1 from the LL\_Init being called once. So AllocationCount equals 1 + 4 = 5. The output printed data order makes sense because first DemoData[0] will be added as the list's head and tail. Then DemoData[1] will be added as the new head, making the order {2000, 1000}. Then DemoData[2] and DemoData[3] will be added, making the order {4000, 3000, 2000, 1000} with the last item (DemoData[3]) added as the head, or start.
3. LL\_Delete repeatedly gets the front node and deletes until the number of nodes in the list equals 0. While deleting the node, it deallocates its dynamically allocated memory and decreases AllcoationCount by 1. AllocationCount is decreased by 1 for the final time in LL\_Delete so that it equals 0 because the singly linked list will end up equaling NULL.
4. LL\_AddAtEnd updates the tail of the list and places a node there. First, it assures that the list is not empty. Then it creates a new node and assigns it to the tail of the list. If list is empty then it calls LL\_AddToFront instead to assign this new node to both the head and tail. Finally, the number of nodes in the list is updated by +1. The AllocationCount equals 5 after the loop because making a node also allocates space for the user's data. There are four nodes being created during this loop and AllocationCount was already 1 from the LL\_Init being called once. So AllocationCount equals 1 + 4 = 5.
5. The output printed data order makes sense because first DemoData[0] will be added as the list's head and tail because it will be the only node in the LinkedList, which can be referenced as both a head and a tail. Then DemoData[1] will be added as the new tail, making the order {1000, 2000}. Then DemoData[2] and DemoData[3] will be added, making the order {1000, 2000, 3000, 4000} with the last item (DemoData[3]) added as the tail, or end.
6. LL\_GetAtIndex returns the UserData at a specific index. First, it assures that the list is not empty. Then it assures that the index is valid. It starts at the head of the list and moves forward until it gets to the node at said specific index. Then it returns that index's UserData.   
   LL\_SetAtIndex sets the UserData at a specifix index and does not return anything. First, it assures that the list is not empty. Then it assures that the index is valid. It starts at the head of the list and moves forward until it gets to the node at said specific index. Then it updates the node's UserData at that index.
7. LL\_Swap switches the nodes at specific indices to swap the data content itself. First, it assures that the list is not empty. Then it assures that the two indices are valid. If both indices are the same, no action needs to be taken. Otherwise, overwrite specific index one's data with specific index two's data. Then it overwrites specific index two's data with specific index one's data. The output printed data order makes sense because first DemoData[0] will be switched with DemoData[3], making the order {4000, 2000, 3000, 1000}. Then DemoData[1] will be switched with DemoData[2], making the order {4000, 3000, 2000, 1000}. The number of operations (two) are since the LinkedList’s length is 4 nodes long. Switching the first and last nodes as well as the middle two nodes results in a completely swapped LinkedList. What occurs first is, loop = 0 and LL\_Length(LL)- loop - 1 = 3 i.e. switching 0 and 3. Second, loop = 1 and LL\_Length(LL)- loop - 1 = 2 i.e. switching 1 and 2. When loop = 1 it is less than LL\_Length(LL)/2 so the swap function returns to move onto the rest of the main's functionality.
8. Before the last call to LL\_Delete was made, data items were being removed via a loop. Each iteration decreased AllocationCount by 1 until there was an empty LinkedList and AllocationCount equaled 1. LL\_Delete freed the LinkedList which deleted the information structure itself. Then LL\_Delete decreased AllocationCount once more.
9. UserData.h contains a definition of a typedef struct that has one integer field. We would have to change UserData.c if we wanted to our implementations of this struct type to have more fields. For example, we could add a second integer into the struct. Or if we wanted, we could change UserData into a nested struct.
10. LinkedList.h contains C function declarations and definitions. It includes UserData.h so that it may use what code is written in UserData.h. You can see that LinkedList.h uses the UserData typedef struct, so it is necessary to include it in the source file LinkedList.h.
11. SinglyLinkedList.c requires the use of all C function declarations and definitions that is written in LinkedList.h. It also requires the definition of UserData typedef struct written in UserData.h. You can see that SinglyLinkedList.c uses those definitions and functions, so it is necessary to include both .h files in the source file SinglyLinkedList.c. It is important to keep the two files separate instead of copying the content of the header file into the .c file, so that it will not throw errors and become too congested. This way we can include the .h files in many source files without copying it again and again. Editing the header file is much easier than changing all existing copies, which will result in inconsistencies within a program. SinglyLinkedList.c did not need to know that UserData struct contained an int. Instances of structs can be copied from one to another without knowing what their fields are, which is one of the most powerful capabilities associated with structs.

**Problem 2 follows…**  
**Description:** Problem 2 is designed to demonstrate the singly linked list with other kinds of data besides integers to be read from a file. It consists of functions for initializing, deleting, placing at front/end, swapping and removing data from the Singly LinkedList. There are also get front/index and length functions. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code (LinkedListTester.c):**

//

// SinglyLinkedListTester

//

// This is a simple demonstration of the singly linked list functions.

// It demos the abilities to:

// Make a linked list - uses call to LL\_Init()

// Add items to the front of the list - uses call to LL\_AddToFront()

// Add items to the end of the list - uses call to LL\_AddToEnd()

// Get whatever item is at the front of the list, optionally deleting it

// - uses call to LL\_GetFront with option to delete or retain the data

// Treat the list like an array, getting or an item by specifying the

// index of the item (0 is the front) - uses call to LL\_GetAtIndex()

// Whenever we want to see how many items are inside the list, we call

// LL\_Length() to return the item count.

// This code has been "overly documented" so that it serves as a learning

// piece of code. Take the time to read and understand it!

// we use printf from stdio.h

#include <stdio.h>

// we use the definition of NULL and the declarations for

// malloc() and free() from stdlib.h

#include <stdlib.h>

// we use strcpy from string.h

#include <string.h>

// we use assert from assert.h

#include <assert.h>

// we use the linked list, so include its functions that we can call

#include "LinkedList.h"

// we use UserData when we call the list functions

#include "UserData.h"

// PrintLL is a local function that we can call to print out a message (msg) and then

// the list contents (theLL). So we can see how many things are allocated as we proceed,

// it will also print out the number of things allocated

static void PrintLL (char msg[], LLInfoPtr theLL);

// PrintLLitem is a local function that we can call to print out a message (msg) and

// a UserData item. So we can see how many things are allocated as we proceed,

// it will also print out the number of things allocated

static void PrintLLItem (char msg[], UserData D);

// AllocationCount is declared for global use in the list code itself. We extern it so that

// we can see how the allocations are inceeasing or decreasing.

extern int AllocationCount;

//function to populate the array

int getData(char fileName[], UserData DemoData[], int numItems) {

// initialize variables for reading a file

int count;

int importanceTemp;

int numItemsTemp = 0;

char taskNameTemp[80];

// attempt to open the file

FILE \* inputFile = fopen(fileName, "r");

// exit if the file did not open

assert(inputFile != NULL);

// the file opened, so proceed and process its contents

// read records from the file, assigning each record in the array

for (int i = 0; i < numItems; i++) {

count = fscanf(inputFile, "%d%s", &importanceTemp, taskNameTemp);

if (count == 2) {

DemoData[i].importance = importanceTemp;

strcpy(DemoData[i].taskName, taskNameTemp);

numItemsTemp++;

}

}

//return the number of items in the array

return numItemsTemp;

}

int main(int argc, const char \* argv[]) {

// initialize variables to pass into function for reading a file

char fileName[] = "testData.txt";

// array set to fixed amount of 7

UserData DemoData[7];

//initialize variables used for reading a file

int numItems;

//prompt user for number of people to get data for

printf("Give me a number from 1 to 10: ");

scanf("%d", &numItems);

//check that number is within range

if (numItems < 1 || numItems > 10) {

printf("Number is outside the range of 1 to 10. Exiting Program\n");

exit(0);

}

//populate array data and retrieve number of array items filled

int NumDemoDataItems = getData(fileName, DemoData, numItems);

// make a LinkedList to play with

LLInfoPtr LL = LL\_Init();

PrintLL ("Starting demo.\nAfter the LL has been initialized...", LL);

// add some data to the list at the front

for (int loop=0; loop < NumDemoDataItems; loop++)

{

UserData D;

D = DemoData[loop];

LL\_AddAtFront (LL, D);

}

PrintLL ("After data has been placed at the front of the LL...", LL);

// delete the list

LL = LL\_Delete(LL);

PrintLL ("After the LL has been deleted...", LL);

// create another one

LL = LL\_Init();

PrintLL ("After the LL has been initialized...", LL);

// add some data to the list at the end

for (int loop = 0; loop < NumDemoDataItems; loop++)

{

UserData D;

D = DemoData[loop];

LL\_AddAtEnd(LL, D);

}

// print out the data in the LL

PrintLL ("After data items have been added at the end of the LL...", LL);

// swap the items in the LL

for (int loop = 0; loop < LL\_Length(LL)/2; loop++)

LL\_Swap(LL, loop, LL\_Length(LL)- loop - 1);

// print out the number of items in the LL

PrintLL ("After data has been swapped in the LL...", LL);

// Check out the ability to get a few items from the front of the LL

// deleting the items

for (int loop = 0; loop < 2; loop++)

if (LL\_Length(LL) > 0)

{

UserData D = LL\_GetFront(LL, DELETE\_NODE);

PrintLLItem ("A data item has been removed from the front of the LL.. ", D);

}

PrintLL ("After data has been removed from the LL..", LL);

LL = LL\_Delete(LL);

PrintLL ("After the LL has been deleted...", LL);

return 0;

}

// function PrintLL is called to print out a message, followed by the contents of the list

// To determine the items to print, it uses the LL\_Length function to get the list size

// and then calls the LL\_GetAtIndex function to read the UserData for each node in the list.

void PrintLL (char msg[], LLInfoPtr theLL)

{

printf ("%s\nThere are now %d items with an allocation count of %d\n",

msg, LL\_Length(theLL), AllocationCount);

for (int loop = 0; loop < LL\_Length(theLL); loop++)

{

UserData D = LL\_GetAtIndex(theLL, loop);

if (loop == 0)

printf ("Head==> [%d] = %d %s\n", loop, D.importance, D.taskName);

else if (loop == LL\_Length(theLL)-1)

printf ("Tail==> [%d] = %d %s\n", loop, D.importance, D.taskName);

else printf (" [%d] = %d %s\n", loop, D.importance, D.taskName);

}

}

// function PrintLLItem is called to print out a message, followed by the contents of a

// UserData.

// After logging the data content, it logs the number of remaining allocations.

void PrintLLItem (char msg[], UserData D)

{

printf ("%s \nthe UserData contained %d %s\n", msg, D.importance, D.taskName);

printf ("The allocation count is now %d\n", AllocationCount);

}

**Program Code (UserData.h):**

#ifndef USERDATA\_H\_INCLUDED

#define USERDATA\_H\_INCLUDED

// The UserData struct defines what each node in the LL

// contains.

//

// User data in each node contains an integer

typedef struct {

int importance;

char taskName[80];

} UserData, \*UserDataPtr;

#endif // USERDATA\_H\_INCLUDED

**Data File:**  
1 task1

2 task2A

3 task3A

2 task2B

4 task4A

4 task4B

3 task3B

**Program Output (3):**

Give me a number from 1 to 10: 3

Starting demo.

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data has been placed at the front of the LL...

There are now 3 items with an allocation count of 4

Head==> [0] = 3 task3A

[1] = 2 task2A

Tail==> [2] = 1 task1

After the LL has been deleted...

There are now 0 items with an allocation count of 0

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data items have been added at the end of the LL...

There are now 3 items with an allocation count of 4

Head==> [0] = 1 task1

[1] = 2 task2A

Tail==> [2] = 3 task3A

After data has been swapped in the LL...

There are now 3 items with an allocation count of 4

Head==> [0] = 3 task3A

[1] = 2 task2A

Tail==> [2] = 1 task1

A data item has been removed from the front of the LL..

the UserData contained 3 task3A

The allocation count is now 3

A data item has been removed from the front of the LL..

the UserData contained 2 task2A

The allocation count is now 2

After data has been removed from the LL..

There are now 1 items with an allocation count of 2

Head==> [0] = 1 task1

After the LL has been deleted...

There are now 0 items with an allocation count of 0

**Program Output (6):**

Give me a number from 1 to 10: 6

Starting demo.

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data has been placed at the front of the LL...

There are now 6 items with an allocation count of 7

Head==> [0] = 4 task4B

[1] = 4 task4A

[2] = 2 task2B

[3] = 3 task3A

[4] = 2 task2A

Tail==> [5] = 1 task1

After the LL has been deleted...

There are now 0 items with an allocation count of 0

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data items have been added at the end of the LL...

There are now 6 items with an allocation count of 7

Head==> [0] = 1 task1

[1] = 2 task2A

[2] = 3 task3A

[3] = 2 task2B

[4] = 4 task4A

Tail==> [5] = 4 task4B

After data has been swapped in the LL...

There are now 6 items with an allocation count of 7

Head==> [0] = 4 task4B

[1] = 4 task4A

[2] = 2 task2B

[3] = 3 task3A

[4] = 2 task2A

Tail==> [5] = 1 task1

A data item has been removed from the front of the LL..

the UserData contained 4 task4B

The allocation count is now 6

A data item has been removed from the front of the LL..

the UserData contained 4 task4A

The allocation count is now 5

After data has been removed from the LL..

There are now 4 items with an allocation count of 5

Head==> [0] = 2 task2B

[1] = 3 task3A

[2] = 2 task2A

Tail==> [3] = 1 task1

After the LL has been deleted...

There are now 0 items with an allocation count of 0

**Program Output (10):**

Give me a number from 1 to 10: 10

Starting demo.

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data has been placed at the front of the LL...

There are now 7 items with an allocation count of 8

Head==> [0] = 3 task3B

[1] = 4 task4B

[2] = 4 task4A

[3] = 2 task2B

[4] = 3 task3A

[5] = 2 task2A

Tail==> [6] = 1 task1

After the LL has been deleted...

There are now 0 items with an allocation count of 0

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data items have been added at the end of the LL...

There are now 7 items with an allocation count of 8

Head==> [0] = 1 task1

[1] = 2 task2A

[2] = 3 task3A

[3] = 2 task2B

[4] = 4 task4A

[5] = 4 task4B

Tail==> [6] = 3 task3B

After data has been swapped in the LL...

There are now 7 items with an allocation count of 8

Head==> [0] = 3 task3B

[1] = 4 task4B

[2] = 4 task4A

[3] = 2 task2B

[4] = 3 task3A

[5] = 2 task2A

Tail==> [6] = 1 task1

A data item has been removed from the front of the LL..

the UserData contained 3 task3B

The allocation count is now 7

A data item has been removed from the front of the LL..

the UserData contained 4 task4B

The allocation count is now 6

After data has been removed from the LL..

There are now 5 items with an allocation count of 6

Head==> [0] = 4 task4A

[1] = 2 task2B

[2] = 3 task3A

[3] = 2 task2A

Tail==> [4] = 1 task1

After the LL has been deleted...

There are now 0 items with an allocation count of 0

**Problem 3 follows…**  
**Description:** Problem 3 is designed to demonstrate the double linked list with node traversal (forward and backward). It consists of functions for initializing, deleting, placing at front/end, swapping and removing data from the Double LinkedList. There are also get front/index and length functions. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code (copies of code for functions LL\_GetAtIndex and LL\_SetAtIndex in DoubleLinkedList.c):**

/////////////

// LL\_GetAtIndex returns the node user data at the specified index

// in the underlying LL.

// It counts nodes from the end of the LL and assumes the LL

// is a double linked list, returning the UserData at the index

/////////////UserData LL\_GetAtIndex (LLInfoPtr LLI\_Ptr, int FetchIndex)

{

// Make sure the LL exists and the index is valid

assert (LLI\_Ptr != NULL);

assert ((FetchIndex >= 0) && (FetchIndex < LLI\_Ptr->NumNodesInList) );

NodePtr curr;

if (FetchIndex > LLI\_Ptr->NumNodesInList/2) {

// set curr to the end of the list

curr = LLI\_Ptr->Tail;

// move backward until we get to the desired data

int backmoves = LLI\_Ptr->NumNodesInList - FetchIndex - 1;

while (backmoves--)

curr = curr->prev;

} else if (FetchIndex <= LLI\_Ptr->NumNodesInList/2) {

// set curr to the start of the list

curr = LLI\_Ptr->Head;

// move forward until we get to the desired data

int forward = FetchIndex;

while (forward--)

curr = curr->next;

}

// return it

return curr->Data;

}

/////////////

// LL\_SetAtIndex updates the node UserData at the specified index

// in the underlying LL.

// It counts nodes from the back of the LL and assumes the LL

// is a double linked list

// Once the position in the LL has been reached, the data is updated to what

// was provided by the caller.

/////////////

void LL\_SetAtIndex (LLInfoPtr LLI\_Ptr, UserData D, int UpdateIndex)

{

// Make sure the LL exists and the index is valid

assert (LLI\_Ptr != NULL);

assert ((UpdateIndex >= 0) && (UpdateIndex < LLI\_Ptr->NumNodesInList) );

if (UpdateIndex > LLI\_Ptr->NumNodesInList/2) {

// set curr to the end of the list

NodePtr curr = LLI\_Ptr->Tail;

// move backward until we get to the desired data

int backmoves = LLI\_Ptr->NumNodesInList - UpdateIndex - 1;

while (backmoves--)

curr = curr->prev;

// update it

curr->Data = D;

} else if (UpdateIndex <= LLI\_Ptr->NumNodesInList/2) {

// set curr to the start of the list

NodePtr curr = LLI\_Ptr->Head;

// move forward until we get to the desired data

int forward = UpdateIndex;

while (forward--)

curr = curr->next;

// update it

curr->Data = D;

}

}

**Output File:**

Starting demo.

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data has been placed at the front of the LL...

There are now 4 items with an allocation count of 5

Head==> [0] = 4000

[1] = 3000

[2] = 2000

Tail==> [3] = 1000

After the LL has been deleted...

There are now 0 items with an allocation count of 0

After the LL has been initialized...

There are now 0 items with an allocation count of 1

After data items have been added at the end of the LL...

There are now 4 items with an allocation count of 5

Head==> [0] = 1000

[1] = 2000

[2] = 3000

Tail==> [3] = 4000

After data has been swapped in the LL...

There are now 4 items with an allocation count of 5

Head==> [0] = 4000

[1] = 3000

[2] = 2000

Tail==> [3] = 1000

A data item has been removed from the front of the LL..

the UserData contained 4000

The allocation count is now 4

A data item has been removed from the front of the LL..

the UserData contained 3000

The allocation count is now 3

After data has been removed from the LL..

There are now 2 items with an allocation count of 3

Head==> [0] = 2000

Tail==> [1] = 1000

After the LL has been deleted...

There are now 0 items with an allocation count of 0