PLAYBOOK NOTES: ADVANCE LINK LIST PROGRAM

Before we get started:

It is important that you trace through and understand the simple link list program before reading through the advance link list program.

This link list will be like the simple link list program, except all the numbers will be kept in order. When the link list is printed to the screen you will see the numbers printed starting the smallest number and increasing to the largest number. This document will be using these three pointers (all created in the CreateList function) to accomplish this task.

POINTER	DESCRIPTION
pList	Like in the simple link list program, this is the anchor of our link list.
pCurrent	This pointer will be used to look for the location to insert the new node.
pPrev	This pointer will always point to the node that comes before pCurrent.

Since much of the code is the same, we will only be discussing two functions.

FUNCTION	DESCRIPTION
FindLocation	This function will find the location to insert the new node. Updates will be made to pCurrent and pPrev.
InsertNewNode	This function will use all three pointers to insert the new node into the link list.

These functions will be called from the CreateList function. It is worth noting that pCurrent and pPrev need to be initialized to NULL before every call to FindLocation. If we assume that we have already created our new node, we now need to find the location for pointers pCurrent and pPrev.

FUNCTION: FindLocation(linkList* pList, linkList** pCurrent, linkList** pPrev, int value)

First notice the parameters. pList is a single pointer because the list itself will not be changing.

Both pointers pCurrent and pPrev are double pointers because the addresses they point to will be changing.

The last parameter is the value of the new node. This will be used to help determine how far down the list pointers pCurrent and pPrev must traverse.

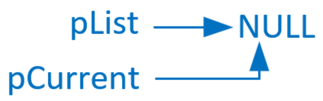
Start by examining the following code:

```
int done = FALSE;

// Assign pCurrent to the list
*pCurrent = pList;
```

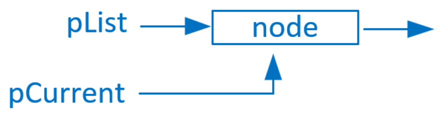
The value for FALSE is 0 and is defined by a #define in the header file. Worth noting, TRUE is defined as 1 in the header file. The value for the variable done will be changed when we find the location to insert the new node. More on the done variable later.

The pointer pCurrent is assigned to the list.

If the pointer pList is equal to NULL (an empty list), then the pointer pCurrent will also be assigned NULL.



If there is at least one node in the list, then the pointer pCurrent will point to the first Node.



The next line of code makes sure there is something in the list.

```
// Check to see if our link list is empty
if (!IsEmpty(pList))
{
```

NOTE: The IsEmpty function will return the pointer passed in equal to NULL.

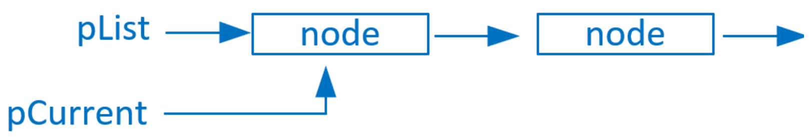
If there is nothing in the list (pList == NULL), then we are done. The pointers pCurrent and pPrev will exit the function equaling NULL. If there is at least one node in the list, then the test for the if-statement will pass and we move on to the loop.

Examine the while loop that is inside the if-statement:

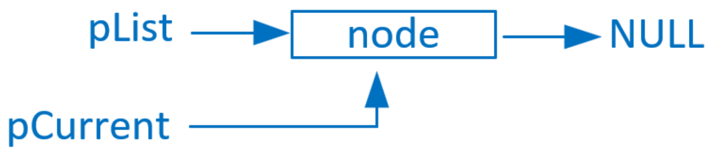
```
while ((*pCurrent)->pNext != NULL && done == FALSE)
{
```

Reaching this point in the code, we know that there is at least on node in the list.

The first test will determine if there at least one more node in the list (as pictured below).



If there is not (as pictured below) then we are done, and the loop will exit.



In this situation, the pointer pPrev will still equal NULL.

The second test (done == false) will flip true inside the loop when we find the location for the new node (more on that later).

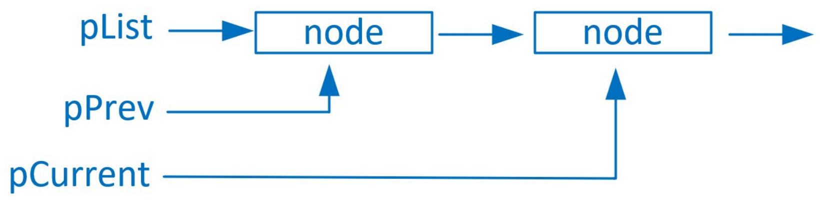
Assuming there are at least two nodes, we now must check the new value against the value of the current node.

```
if (value > (*pCurrent)->record.number)
{
```

If the new value is greater than the value of the current node, then we need to move down the list (See the code inside the if-statement):

```
// Assign previous to current
*pPrev = *pCurrent;

// Move current down one node
*pCurrent = (*pCurrent)->pNext;
```

This code will result in our pointers looking like this:



Going back to the if-statement, if the new value is less than or equal to the value pointed to by pointer pCurrent, then the test fails, and we are done traversing the list.

```
else
{
    // Found where current is less than our new value
    done = TRUE;
```

Regardless of which way the if-statement falls, we go back to the top of the while-loop.

The first test will again check to see if there is another node in the link list.

The second test will check to see if we are done searching.

If either of these test fails, we exit the loop and then exit the function. Otherwise, we continue to traverse down the list.

Once we have established the location for the pointers pPrev and pCurrent we are ready to insert our new node into our link list.

FUNCTION: void InsertNewNode(linkList** pList, linkList** pCurrent, linkList** pPrev, linkList** pTemp)

Notice how all the parameters are all double pointers. This is because there is a potential for each pointer to change the address it is pointing too.

Using our three pointers (pList, pCurrent, pPrev) we will check for five different scenarios.

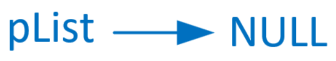
1. The list is empty
2. There is one node in the list - new node is inserted after the first node (This can happen only once)
3. The new node is inserted before the first node – (NEW MIN)
4. More than one node in the list - new node is inserted at the end of the list – (NEW MAX)
5. More than one node in the list - new node is inserted in between the current and previous pointer

FIRST SCENARIO: The list is empty

The first if-statement will check to see if pointer pFirst is equal to NULL:

```
// Check to see if the list is empty
if (IsEmpty(*pList))
{
```

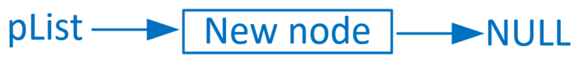
In this scenario, our pList is assigned to NULL (see picture below)



Because the list is empty, the test for the if-statement passes and we use the following code to insert the first node.

```
// Set pTemp to the first node in the list
*pList = *pTemp;
```

The pointer pList now looks like this:

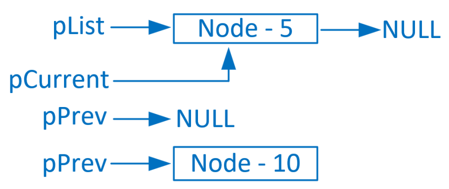


The new node is now inserted into our link list. The remaining parts of the if-statement will be skipped, and the function will exit.

SECOND SCENARIO: There is one node in the list - new node is inserted after the first node

NOTE: This scenario can only happen one time. Once there is more than one node in the list and a node needs to be inserted after the first node, the situation will fall under scenario 5.

We will use some dummy data to help illustrate the scenario. In this scenario, there is one node (with the number equal to 5) in the list. The pointer pCurrent will be equal to pointer pFirst and the pointer pPrev will be equal to NULL. Our pTemp pointer (the new node) will equal a node with the number set to 10.



Looking at the picture, we can see that the pointer pTemp needs to go after the node already in the link list (Remember, we are keeping the numbers in order).

Let's go back to the code:

First, we have the else-if that comes after our first if-statement:

```
// Check to see if their is only one node in the list
else if (*pPrev == NULL)
```

Because pointer pPrev is NULL (see the picture above) we will enter this block of code.

This brings us to an if-statement that will compare the value of the new node to the value pointed to by pointer pCurrent:

```
if ((*pTemp)->record.number > (*pCurrent)->record.number)
{
```

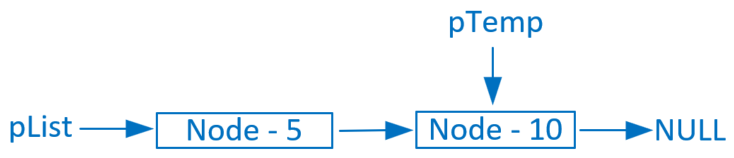
Remember, we are using double pointers, so we have to deference the pointers first.

In this scenario, this if-statement is true (10 > 5). So, we enter the block of code to insert the new node at the end of the list.

All we must do in this scenario is assign the next pointer of first node to equal to our new node.

```
// Place the new node after the first node
(*pList)->pNext = *pTemp;
```

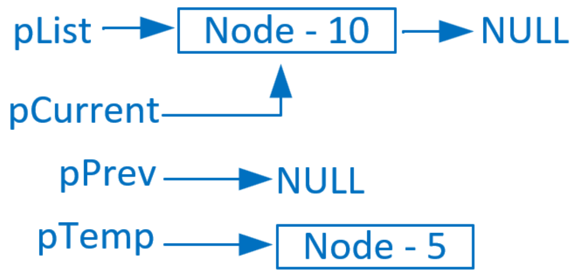
The pointers will now look like this:



The new node is now inserted. The rest of the if-statements are skipped the function will exit.

THIRD SCENARIO: The new node is inserted before the first node – (NEW MIN)

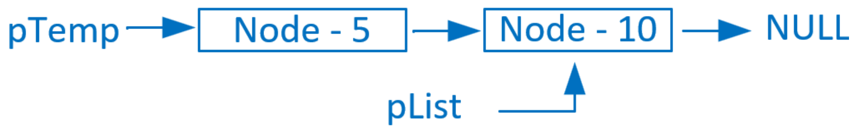
Once again, we will use dummy data to help illustrate the scenario. This time 10 is already in the list and 5 is the number of the new node. To keep the numbers in order, the new node will need to go in front of the node that is already in the link list.



To put the new node in front of the node already in the link list, we will first assign the next pointer from temp to the node already in the list:

```
// Place the new node before the first node
(*pTemp)->pNext = *pList;
```

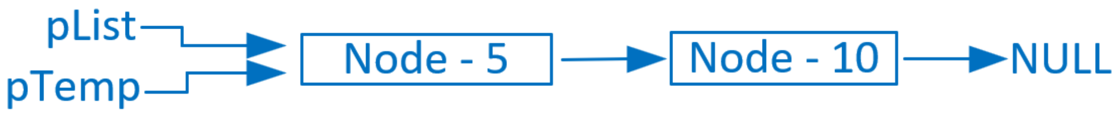
Now our pointers look like this:



Next, we will assign our anchor pList to equal pTemp:

```
*pList = *pTemp;
```

Now our link list looks like this:

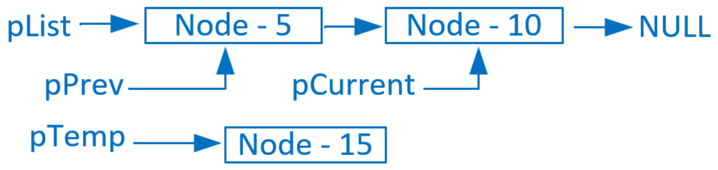


The new node is now inserted. The rest of the if-statements are skipped the function will exit.

FOURTH SCENARIO: new node is inserted at the end of the list – (NEW MAX)

For this scenario, we have already failed the if-statement that checks to see if pPrev is NULL. This means that we can assume that there are at least two nodes in our link list.

For this scenario we will also assume that the pointer pCurrent is pointing to the last node in our link list.
(See picture below)



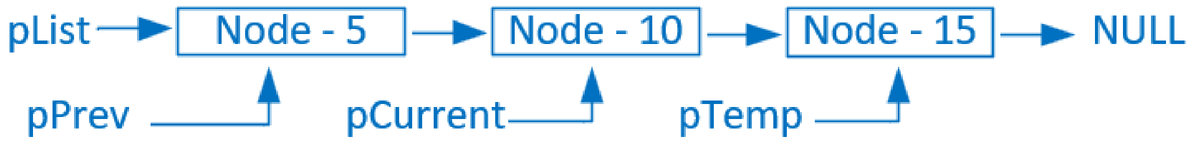
Looking at the picture above you can see that some dummy data has been filled in to help see the scenario. Notice that the pointer pCurrent is pointing to the last node. Also notice that the number pointed to by pTemp is larger than the last number in our link list. These conditions will pass both test of this if-statement:

```
else if ((*pCurrent)->pNext == NULL &&
         (*pTemp)->record.number > (*pCurrent)->record.number)
{
```

Since we are placing the new node (pTemp) at the end of the list, all we must do is assign the pointer pCurrent to our new node pTemp.

```
// Add new node to the end of the list
(*pCurrent)->pNext = *pTemp;
```

Our pointers now look like this:



The new node is now inserted. The rest of the if-statements are skipped, and the function will exit.

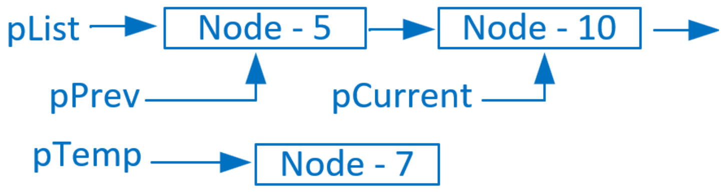
FIFTH SCENARIO: new node is inserted in between the current and previous pointer

In our last scenario we can assume that there are two or more nodes in are list and that the new node will go after the node pointed to by pPrev and before the node pointed to by pCurrent.

Revisit the if-statement from the fourth scenario:

```
else if ((*pCurrent)->pNext == NULL &&
         (*pTemp)->record.number > (*pCurrent)->record.number)
{
```

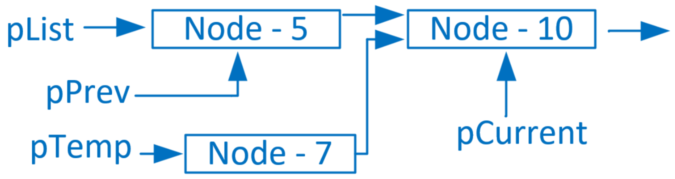
The first test may or may not fail, but the second if-statement will fail. So, our pointers (again with dummy data) look like this:



To place our new node (pointed to by pointer pTemp) in between the pointer pPrev and pointer pCurrent we will first link pointer pTemp to pointer pCurrent:

```
// Add new node before the current node.
(*pTemp)->pNext = *pCurrent;
```

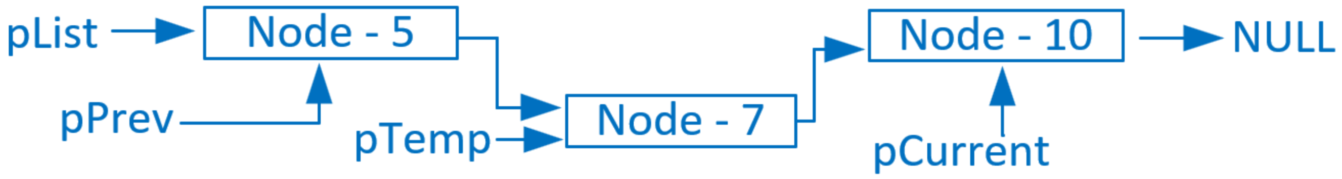
Now our pointers look like this:



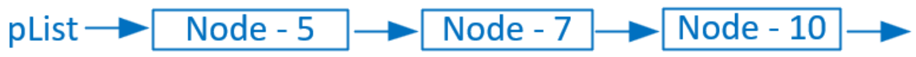
Next, we assign the previous pointer (next) to point to pointer pTemp:

```
(*pPrev)->pNext = *pTemp;
```

Our pointers start to look like this:



Eliminate the pointer pPrev, pointer pTemp and pointer pCurrent from the picture. Then straighten the list out and you get:



The new node is now inserted, and the function exits.

Now that we have finished the function, let's move on to a debugging technique:

DEBUGGING LINK LIST

When writing code that incorporates link list it is a good idea to have a plan on how to debug it.

When developing your code, sometimes it's a good idea to know what is executing.

Let's take our InsertNewNode function. We have five scenarios that we have talked about. Let's see if they all work by inserting a printf statement into each scenario.

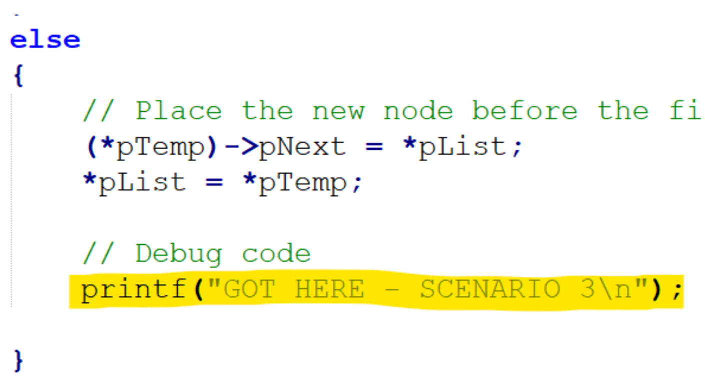
Scenario 1:

```
// Check to see if the list is empty
if (*pList == NULL)
{
    // Set pTemp to the first node in the list
    *pList = *pTemp;

    // Debug code
    printf("GOT HERE - SCENARIO 1\n");

}
```

Scenario 2:

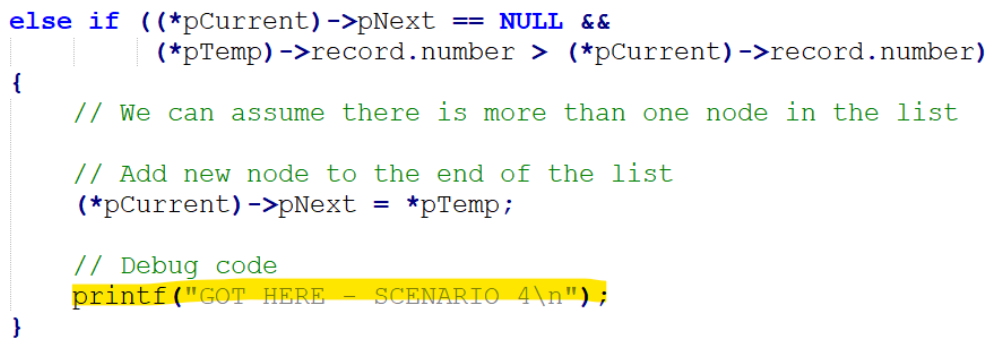
```
// Check to see if their is only one node in the list
else if (*pPrev == NULL)
{
    if ((*pTemp)->record.number > (*pCurrent)->record.number)
    {
        // Place the new node after the first node
        (*pList)->pNext = *pTemp;

        // Debug code
        printf("GOT HERE - SCENARIO 2\n");
    }
```

Scenario 3:



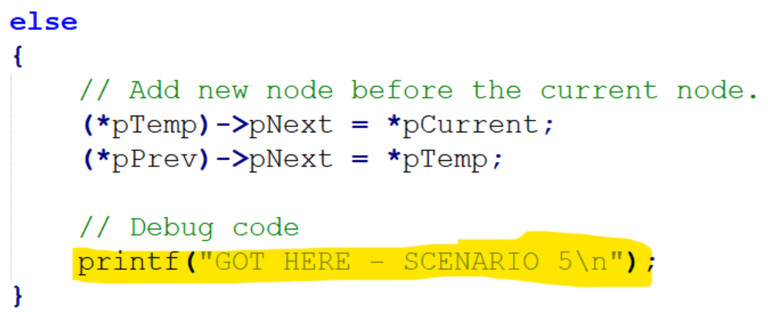
```
else
{
    // Place the new node before the first node
    (*pTemp)->pNext = *pList;
    *pList = *pTemp;

    // Debug code
    printf("GOT HERE - SCENARIO 3\n");

}
```

Scenario 4:



```
else if ((*pCurrent)->pNext == NULL &&
         (*pTemp)->record.number > (*pCurrent)->record.number)
{
    // We can assume there is more than one node in the list

    // Add new node to the end of the list
    (*pCurrent)->pNext = *pTemp;

    // Debug code
    printf("GOT HERE - SCENARIO 4\n");
}
```

Scenario 5:



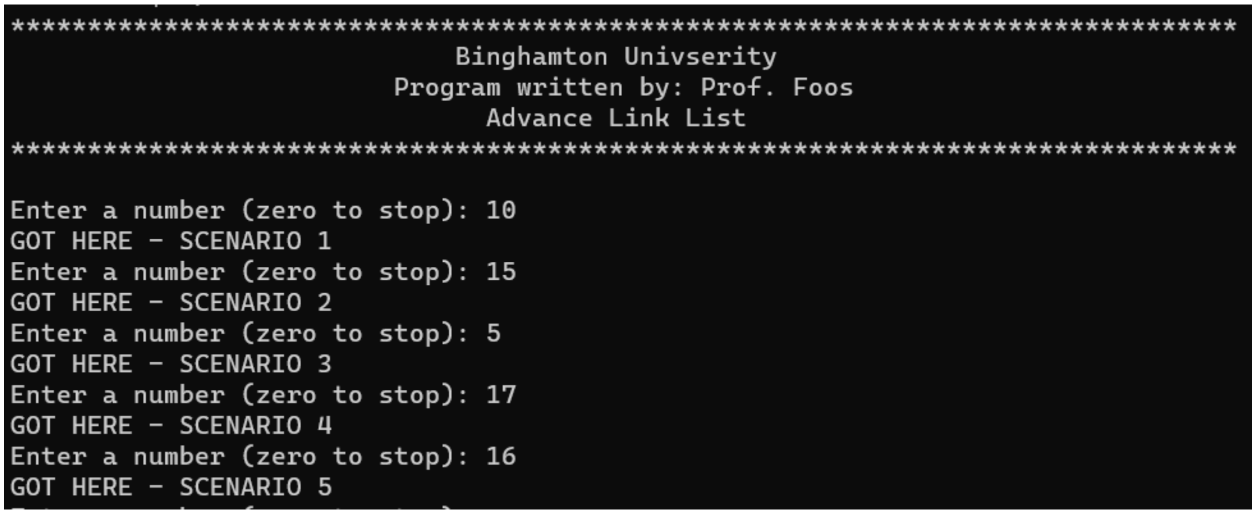
```
else
{
    // Add new node before the current node.
    (*pTemp)->pNext = *pCurrent;
    (*pPrev)->pNext = *pTemp;

    // Debug code
    printf("GOT HERE - SCENARIO 5\n");
}
```

Now when we enter the data, we will know which block of code is being executed.

If we run the program with the following data, we will test each block of code:

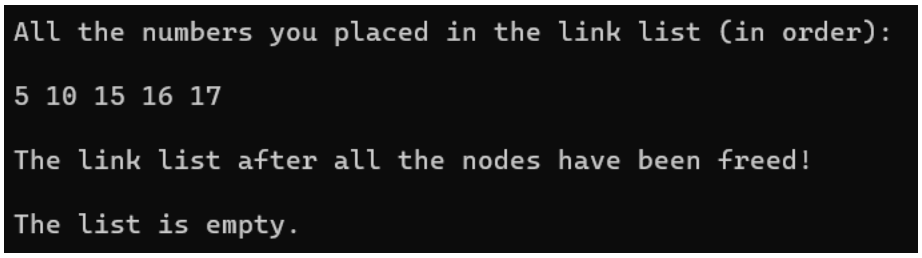
10   15   5   17   16

The following is the output while entering the above data:



```
********************************************************************************
                            Binghamton Univserity
                        Program written by: Prof. Foos
                              Advance Link List
********************************************************************************

Enter a number (zero to stop): 10
GOT HERE - SCENARIO 1
Enter a number (zero to stop): 15
GOT HERE - SCENARIO 2
Enter a number (zero to stop): 5
GOT HERE - SCENARIO 3
Enter a number (zero to stop): 17
GOT HERE - SCENARIO 4
Enter a number (zero to stop): 16
GOT HERE - SCENARIO 5
```

After pressing 0 to stop:



```
All the numbers you placed in the link list (in order):

5 10 15 16 17

The link list after all the nodes have been freed!

The list is empty.
```

So now we know the code works! But what if it doesn't work (and trust me, this didn't work the first time I ran it, and you will run into situations where your code doesn't work too 😊)?

By using printf statements you can determine where code is being executed. You can also print data of certain nodes to see if your nodes are linking correctly.

After you have coded and debugged your code, if you are still stuck, then you ask for help from your UCA or professor.