
Microchip University

Linked Lists Lab Manual



Linked Lists Lab Manual

Table of Contents

- **LAB1** **Page 4**
 - Linked Lists with Dynamic Memory Allocation

- **LAB2** **Page 10**
 - Linked Lists without Dynamic Memory Allocation

Linked Lists

LAB1 – Linked Lists with Dynamic Memory Allocation

Tasks

In this section you will learn:

- How to create a linked list
- How to search the linked list
- How to add nodes to the linked list
- How to remove nodes from the linked list



Open the Linked_List project by selecting 'File -> OpenProject' then navigating to the folder where the "Linked_List" project is located. Double click on "main.c" under the 'Source Files' folder in the Projects window to open the file.

Code Analysis and Code Execution

The linked list node structure is declared along with the pointers that will be used in the program:

```
//Create the node structure for each element in the linked list
struct Node
{
    int val;
    struct Node *nextPtr;
};

//Create two pointers that will be used to search the linked list and add or delete nodes
struct Node *currentPointer;
struct Node *previousPointer;

//create a variable that will be used as a flag to denote that the search found a node that
// we were looking for
uint8_t nodeFound;
```

The structure *Node* contains an integer *val* and a pointer **nextPtr*. *val* is the value that the node contains and **nextPtr* points to the next node in the linked list. Two pointers are created that point to the created nodes in the program. *currentPointer* and *previousPointer* are used to hold node positions as the linked list is searched in the code. *nodeFound* is created as a flag to signify that a node was found. In an actual code implementation, this flag would not be required since it would be easier to have a search function that would return a NULL if a requested node value was not found. However, this example is in one function to make the code easier to understand.

Now some linked list nodes and pointers to the linked list are created:

```

//Create structure pointer to four instances of nodes in the linked list. The only required
// nodes to be created are "headNode" to signify the first node in the linked list and "newNode"
// that will be used to manipulate the items in the list. The "secondNode" and "thirdNode"
// are created just for demonstration purposes to allow for some initial created nodes and make
// it easier to understand the linked list concept.
    struct Node* headNode = NULL;
    struct Node* secondNode = NULL;
    struct Node* thirdNode = NULL;
    struct Node* newNode = NULL;

//allocate 3 nodes in the heap
    headNode = (struct Node*)malloc(sizeof(struct Node));
    secondNode = (struct Node*)malloc(sizeof(struct Node));
    thirdNode = (struct Node*)malloc(sizeof(struct Node));

//initialize all nodes
    headNode -> val = 2;
    headNode -> nextPtr = secondNode;
    secondNode -> val = 3;
    secondNode -> nextPtr = thirdNode;
    thirdNode -> val = 5;
    thirdNode -> nextPtr = NULL;

```

headNode is a pointer to the first node in the list. *secondNode* and *thirdNode* are created to show values and pointers in nodes at fixed positions. In an actual application, these nodes would not be created but are shown here to demonstrate the operation of the linked list. These nodes are allocated memory positions. Note that this allocation occurs during runtime. Finally, each node is given a value and the respective pointers are set equal to the following node. Again, in a real application, only the *headNode* needs to be declared and the list can be built from that node.

A new node is now declared and will be used to demonstrate how this node can be inserted into the list:

```

//create a new node with a value of 4. It is not placed in the list yet.
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> val = 4;

```

The pointers are both initialized to point to the head node and the *nodeFound* flag is cleared:

```

//add the node in numerical order in the linked list. Initialize the node pointers.
    currentPointer = headNode;
    previousPointer = headNode;
    nodeFound = 0;

```

The search algorithm is now executed. This code searches through the linked list, starting at the head node. The *newNode* value is 4 and this node will be placed in numerical order in the linked list. The list is searched for a value greater than 4, then the new node is placed in the list just before the first value that is found that is greater than 4. When a node with a value greater than 4 is found, the pointer in *newNode* is set equal to the current pointer so that it points to the node with the greater value. The previous pointer is set to point to the new node. If the value in the new node is less than the head node value, then the *headNode* location is set equal to the *newNode* location. This search continues until the previous pointer value is not NULL which signifies that the end of the list has been reached.

```
while(previousPointer -> nextPtr != NULL)
{
    if((currentPointer -> val) > (newNode -> val))
    {
        nodeFound = 1;
        newNode -> nextPtr = currentPointer;

        if(currentPointer != headNode)
            previousPointer -> nextPtr = newNode;

        else
            headNode = newNode;

        break;
    }

    previousPointer = currentPointer;
    currentPointer = currentPointer -> nextPtr;
}
```

If the search does not find any values that are greater than the value in *newNode*, the loop exits without the *nodeFound* flag being set. The following code checks if this flag is set. If it is not, the *previousPointer* that points to the last node in the list is now set to point to the *newNode*. The pointer in *newNode* is set to point to NULL since it is now the last node.

```
//Check if a node was found. If not, put the value at the end of the list
if(!nodeFound)
{
    previousPointer -> nextPtr = newNode;
    newNode -> nextPtr = NULL;
    nodeFound = 0;
}


nodeFound = 0;
```

Set a breakpoint at the start of the node search:

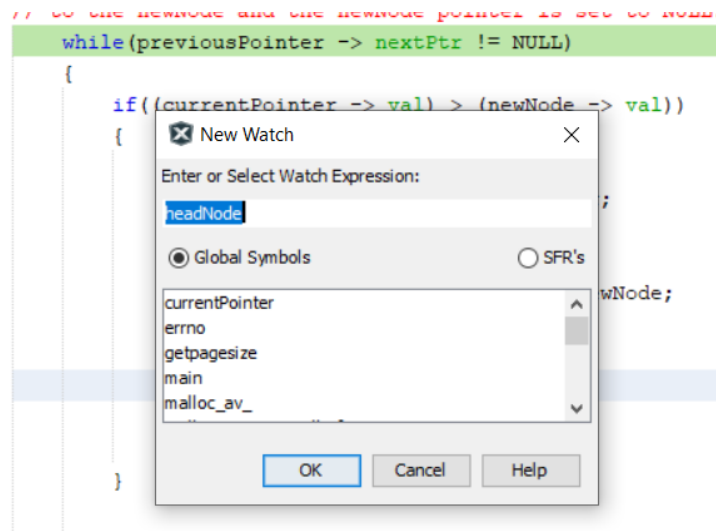
```

119 //If the value to be placed is less than the first location in the list, the newNode pointer
120 // is set to be the headPointer. If the value to be placed is greater than the last location,
121 // this search will fall through without placing the node in the list. nodeFound is used to
122 // signify that a node was placed. If this value is 0, then the previousPointer is set to point
123 // to the newNode and the newNode pointer is set to NULL.
124 while(previousPointer -> nextPtr != NULL)
125 {
126     if((currentPointer -> val) > (newNode -> val))
127     {
128         nodeFound = 1;
129         newNode -> nextPtr = currentPointer;

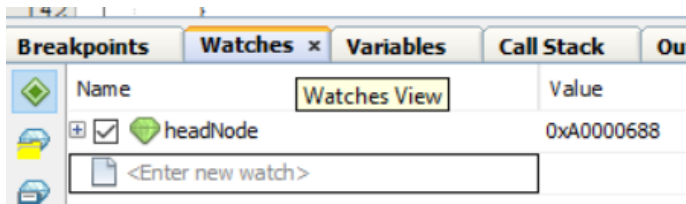
```

Now click on the 'Debug and Run' icon  to start the debugger. The program will start execution and stop.

Right-click on *headNode* and choose 'New Watch'. Select 'OK' in the 'New Watch' window:



Open the 'Watch' tab to see the *headNode* variable:



Click the '+' in front of this variable to see the members of the structure that *headNode* is pointing to. Continue to expand the pointers within the node to see the full chain of members in the linked list:

Breakpoints	Watches x	Variables	Call Stack	Output	
	Name	Value	Decimal	Address	
	headNode	0xA0000688	2684356232	0xA0000FC8	
	val	0x00000002	2	0xA0000688	
	nextPtr	0xA0000698	2684356248	0xA000068C	
	val	0x00000003	3	0xA0000698	
	nextPtr	0xA00006A8	2684356264	0xA000069C	
	val	0x00000005	5	0xA00006A8	
	nextPtr	0x00000000	0	0xA00006AC	
	<Enter new watch>				

You can see that the chain links nodes with the values 2, 3 and 5.

Now place a breakpoint on the “nodeFound = 0;” line:

151		nodeFound = 0;
153		

Click the green run button . The code will execute past the loop where the number 4 is inserted numerically into the list. Look at the *headNode* variable in the ‘Watch’ window and expand everything within that variable:

Breakpoints	Watches x	Variables	Call Stack	Output	
	Name	Value	Decimal	Address	
	headNode	0xA0000688	2684356232	0xA0000FC8	
	val	0x00000002	2	0xA0000688	
	nextPtr	0xA0000698	2684356248	0xA000068C	
	val	0x00000003	3	0xA0000698	
	nextPtr	0xA00006B8	2684356280	0xA000069C	
	val	0x00000004	4	0xA00006B8	
	nextPtr	0xA00006A8	2684356264	0xA00006BC	
	val	0x00000005	5	0xA00006A8	
	nextPtr	0x00000000	0	0xA00006AC	
	<Enter new watch>				

You can see that the number 4 is inserted in the node between nodes containing 3 and 5.

The next section of code removes a node from the linked list. The pointers are initialized first:

```
currentPointer = headNode;
previousPointer = headNode;
```

The search algorithm is the same as the search algorithm used to place a new node. One function could have been written to perform the search and it could have been called twice in this project. However, the search was written as inline code to make the code easier to follow.

This algorithm simply finds a node and removes it. If it is not the head node or the last node, the *previousPointer* is set to point at the address that the removed node was pointing to. That effectively cuts the removed node out of the list and links the nodes on each side of the cut node. If the removed node is the head node, then the node that the previous node was pointing to is made the head node. If the last node is to be removed, then the node before the last node is changed to point to NULL and becomes the last node in the list.

```
while(previousPointer -> nextPtr != NULL)
{
    if((currentPointer -> val) == 4)
    {
        if(currentPointer == headNode)
        {
            headNode = currentPointer -> nextPtr;
        }

        else if(currentPointer -> nextPtr == NULL)
        {
            previousPointer -> nextPtr = NULL;
        }

        else
        {
            previousPointer -> nextPtr = currentPointer -> nextPtr;
        }

        free(currentPointer);
        break;
    }

    previousPointer = currentPointer;
    currentPointer = currentPointer -> nextPtr;
}
```

Set a breakpoint on the 'while(1)' instruction at the end of the code. Select the green run button. Look at the *headNode* variable again in the watch window and the node with the number 4 value is removed.

You can use this same procedure to add numbers that would be inserted at the beginning or end of the linked list to see the performance.

The remaining portion of the code simply deletes a node and makes sure that the full chain link remains intact. If a node in the middle of the chain is removed, the node before and after it are joined and the removed node is freed from the memory allocation. If a node is removed at the end of the list, then the second-to-last node is made to point to NULL. Finally, if the first node in the chain is removed, the second node becomes the head node. To test this functionality, set the breakpoint at the 'while(1)' instruction at the end of the code. Run the code as before and look at the *headNode* pointer to see that the node with the value of 4 is removed and that all nodes are still linked.

Linked Lists

LAB2 – Linked Lists without Dynamic Memory Allocation

Tasks

In this section you will learn:

- How to create a linked list without dynamically allocating memory
- How to search the linked list
- How to add nodes to the linked list
- How to remove nodes from the linked list



MICROCHIP

Open the Linked_List2 project as we have opened other projects in this manual. Double click on “main.c” under the ‘Source Files’ folder in the Projects window to open the file.

Code Analysis and Code Execution

The linked list node structure is declared along with the pointers that will be used in the program:

```
//Create the node structure for each element in the linked list
struct Node
{
    uint8_t locationIsAvailable;
    uint32_t __attribute__((packed)) val;
    struct __attribute__((packed)) Node *nextPtr;
};
```

The structure *Node* contains an integer *val* and a pointer **nextPtr*. *val* is the value that the node contains and **nextPtr* points to the next node in the linked list. Two pointers are created that point to the nodes in the linked list. *currentPointer* and *previousPointer* are used to hold node positions as the linked list is searched in the code (will be initialized shortly). *nodeFound* is created as a flag to signify that a node was found. In an actual code implementation, this flag would not be required since it would be easier to have a search function that would return a NULL if a requested node is not found. However, this example is in one function to make the code easier to understand.

One additional value is added to every node to signify if the node is available for data. This variable is *locationIsAvailable*. This node is also packed to make sure that the values are in contiguous RAM locations so that there is no wasted space. This is not necessary but guarantees that there are no empty RAM locations between the variables in each node.

Memory is now reserved for the linked list using an array of nodes:

```
//reserve memory for 20 nodes. This array is only used for memory allocation.
struct Node listReservedMemory[TOTAL_NODES];
```

The node pointers are set up the same as they were in the linked list project with dynamic memory allocation:

```
//Create two pointers that will be used to search the linked list and add or delete nodes
struct Node *currentPointer;
struct Node *previousPointer;

//create the newNode that will be used to place data within the list
struct Node *newNode;

//create node designation for the first node in the list
struct Node *headNode;

//create a variable that will be used as a flag to denote that the search found a node that
// we were looking for
uint8_t nodeFound;
```

The pointers for the head node and the current position pointer are set up using the array starting address:

```
//establish the linked list first node (listHead) and search pointer
currentPointer = &listReservedMemory[0];
headNode = &listReservedMemory[0];
```

The entire node array is initialized so that the values are 0 and the pointers point to NULL. The *locationIsAvailable* byte is set to 1 to signify that the node is available to use:

```
//initialize all list locations
for(i = 0; i < TOTAL_NODES; i++)
{
    currentPointer->locationIsAvailable = 1;
    currentPointer->val = 0;
    currentPointer->nextPtr = NULL;
    currentPointer++;
}
```

A few nodes are initialized for demonstration purposes:

```
//initialize some nodes to demonstrate the search
currentPointer = &listReservedMemory[0];

headNode -> val = 2;
headNode -> locationIsAvailable = 0;
headNode -> nextPtr = ++currentPointer;

currentPointer -> val = 3;
currentPointer -> locationIsAvailable = 0;
currentPointer -> nextPtr = ++currentPointer;

currentPointer -> val = 5;
currentPointer -> locationIsAvailable = 0;
currentPointer -> nextPtr = NULL;

newNode = NULL;
```

One more routine is required to search the linked list for the first available node. In the previous lab, the dynamic memory location `malloc()` performed this functionality. In this lab, we simply search the linked list to locate a node with *locationIsAvailable* set to 1:

```

//find the first unused location in the list to use as the newNode
// First, reset the currentPointer
currentPointer = headNode;

for(i = 0; i < TOTAL_NODES; i++)
{
    if(currentPointer ->locationIsAvailable)
    {
        newNode = currentPointer;
        newNode -> locationIsAvailable = 0;
        break;
    }

    currentPointer++;
}

```

The setup is complete! The rest of the linked list functionality is identical to the previous linked list example with dynamic memory allocation. The only difference is that when a node is removed from the list, *locationIsAvailable* is set to 1 instead of using the `free()` function.

You can run through the same debug process as you did in the previous lab to see how nodes are added and removed.