**Removing a specific item**

To remove a specific item from the list, either by its index from the beginning of the list or by its value, we will need to go over all the items, continuously looking ahead to find out if we've reached the node before the item we wish to remove. This is because we need to change the location to where the previous node points to as well.

Here is the algorithm:

1. Iterate to the node before the node we wish to delete
2. Save the node we wish to delete in a temporary pointer
3. Set the previous node's next pointer to point to the node after the node we wish to delete
4. Delete the node using the temporary pointer

There are a few edge cases we need to take care of, so make sure we understand the code.

int remove\_by\_index(node\_t \*\* head, int n) {

int i = 0;

int retval = -1;

node\_t \* current = \*head;

node\_t \* temp\_node = NULL;

if (n == 0) {

return pop(head);

}

for (int i = 0; i < n-1; i++) {

if (current->next == NULL) {

return -1;

}

current = current->next;

}

temp\_node = current->next;

retval = temp\_node->val;

current->next = temp\_node->next;

free(temp\_node);

return retval;

}

# [Delete an item from a linked list](http://codereview.stackexchange.com/questions/83659/delete-an-item-from-a-linked-list)

|  |  |
| --- | --- |
| up vote3down vote[favorite](http://codereview.stackexchange.com/questions/83659/delete-an-item-from-a-linked-list) | I wrote this code to delete an item from a linked list. I checked it and it works, but I want to know if the logic could be written simpler or clearer.  typedef struct linked\_list{  int value;  struct linked\_list \* next;  }linked;  int deleteSpecificItem(linked \*\* head, int val)  {  if(!head || !(\*head)) return -1;  linked \* tmp = \*head;  int ret;  linked \* listToDel;  if(tmp->value == val)  {  \*head = (\*head)->next;  ret = tmp->value;  free(tmp);  return ret;  }  while(tmp->next != NULL)  {  if(tmp->next->value == val)  {  listToDel = tmp->next;  tmp->next = tmp->next->next;  ret = listToDel->value;  free(listToDel);  return ret;  }  tmp = tmp->next;  }  return -1;  }  When checking if a pointer is NULL, I can use either !head or head != NULL. What is preferred in companies? |
|  | |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | | 2 |  | | Problem: Your code does not compile. You're missing the definition of linked (which I suspect ought to be called node istead) – [Snowbody](http://codereview.stackexchange.com/users/11647/snowbody" \o "4,904 reputation) [Mar 9 '15 at 15:37](http://codereview.stackexchange.com/questions/83659/delete-an-item-from-a-linked-list#comment150444_83659) |   add a comment |

## 2 Answers

[active](http://codereview.stackexchange.com/questions/83659/delete-an-item-from-a-linked-list?answertab=active#tab-top)[oldest](http://codereview.stackexchange.com/questions/83659/delete-an-item-from-a-linked-list?answertab=oldest#tab-top)[**votes**](http://codereview.stackexchange.com/questions/83659/delete-an-item-from-a-linked-list?answertab=votes#tab-top)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| up vote6down voteaccepted | Apart from the less-than-ideal variable and type names already mentioned you are doubling up on code just to check head first and then separately doing the same again for every other node. The only difference is that you need to modify head if the element to be removed is the first one. This can be condensed into a single loop:  int deleteSpecificItem(linked\*\* head, int val)  {  if (!head || !(\*head)) return -1;  linked\* tmp = \*head;  linked\* prev = NULL;  while (tmp->value != val && tmp->next != NULL)  {  prev = tmp;  temp = temp->next;  }  if (tmp->value == val)  {  if (prev)  {  prev->next = tmp->next;  }  else  {  \*head = tmp->next;  }  free(tmp);  return val;  }  return -1;  }  The way this works is: First we try to find the node which should be removed and we also keep a pointer to the previous node. Once the loop finishes there are several cases to check:   * If the current value is not equal to the one we are looking then it's not in the list and we can return * If the current value is equal to the one we are looking for then tmp points to the node which needs to be removed   + If we have a valid prev then rewire by skipping tmp   + If we don't have a valid prev then apparently head needs to be moved   + Now the current node is unlinked and we can delete it   + We can also return val instead of remembering the value of the current node because we know that they are equal.   Regarding !head vs head == NULL: I tend to prefer !head if it's for simple pointer checks and prev->next == NULL when the expression is more complex. However some companies enforce the style NULL == head because a typo like typing = instead of == will yield in a compile error this way (you can't assign something to a constant) instead of an accidental assignment. I never managed to consistently do it that way (old habits die hard) and to be honest I've never run into a bug due to this (compilers these days spit out warnings for these kind of constructs).   |  |  |  | | --- | --- | --- | | [share](http://codereview.stackexchange.com/a/83677)[improve this answer](http://codereview.stackexchange.com/posts/83677/edit) | [edited Mar 10 '15 at 2:01](http://codereview.stackexchange.com/posts/83677/revisions) | answered Mar 9 '15 at 19:13  [[https://www.gravatar.com/avatar/6a9768c145184d17e1c3839c067d7451?s=32&d=identicon&r=PG](http://codereview.stackexchange.com/users/30346/chriswue)](http://codereview.stackexchange.com/users/30346/chriswue)  [ChrisWue](http://codereview.stackexchange.com/users/30346/chriswue)  **17.5k**32594 | | |
|  | |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  |  | | Great answer! thank you very much! – [Anton.P](http://codereview.stackexchange.com/users/67782/anton-p" \o "143 reputation) [Mar 9 '15 at 19:31](http://codereview.stackexchange.com/questions/83659/delete-an-item-from-a-linked-list#comment150473_83677) |   add a comment | |
| up vote5down vote | | My concerns are with the type name linked and the variable name listToDel. linked is really a node, not a handle to a linked list. Similarly, listToDel is the node that's going to be deleted, not a list that's going to be deleted.  Also, tmp is a very poor variable name; it gives no indication about what it's used for.  You also might be able to save some indirection by changing from using tmp to using curNodeand nextNode. |

## Search Perticular Element : Singly Linked List

**Searching and Traversing for Data in Singly Linked List**  
**Way 1 : This Function only tells that whether data is present or not**

* This function return 1 if data found successfully.
* Returns 0 if data is not present

**int** search(**int** num)

{

**int** flag = 0;

**struct** node \*temp;

temp = start;

**while**(temp!=NULL)

{

**if**(temp->data == num)

**return**(1); //Found

temp = temp->next;

}

**if**(flag == 0)

**return**(0); // Not found

}

**Way 2 : This Function Returns the node.**

* If data or element is not present then return starting node
* Otherwise return the exact node.

**struct** node \* search(**int** num)

{

**int** flag = 0;

**struct** node \*temp;

temp = start;

**while**(temp!=NULL)

{

**if**(temp->data == num)

**return**(temp); //Found

temp = temp->next;

}

**if**(flag == 0)

**return**(start); // Not found

}

# Deleting from a Linked List

1. **Linked List example:**

Here, we will discuss deleting an element with a specific value from a linked list.

Here are some constraints for our example:

* 1. The list holds char's.
  2. Deletion can occur anywhere in the list.
  3. The delete operation will not return the element that we delete, it only need remove it from the list.

Now, recall the basic structure of a singly-linked list:

list

|

v

--------- --------- ---------

| a | --+---> | b | --+---> | c | 0 |

--------- --------- ---------

In other words, there is a node for each element, where nodes consist of a part in which the element is stored and a link to the next node. The last node links to nothing (i.e., there are no nodes after it). Also, there is a link to the beginning of the list.

What do these**links**correspond to in C?

**Answer:** Pointers!

We'll consider our elements and nodes to have the following types:

typedef char elementT;

**...**

typedef struct nodeTag {

elementT element;

struct nodeTag \*next;

} nodeT;

and thus, the pointer to the beginning of the list will be:

nodeT \*list;

**Aside:** Although list is a pointer, we won't use the convention of naming it "listP". That's because we want to be abstract about what "list" is. I.e., users of the list don't really care whether it is implemented as a linked list or array or whatever--when they want to access a list, they just call list functions on this variable.

When the list is empty, what value will the pointer to the beginning have?

**Answer**: There are no nodes to point to, so it will be NULL!

1. **Different cases:**

There are a few steps to deleting a specific element from the list:

* 1. Find the node with the element (if it exists).
  2. Remove that node.
  3. Reconnect the linked list.
  4. Update the link to the beginning (if necessary).

The order in which we perform these steps will depend on how we implement the deletion operation.

**Note:** For simplicity, if there are two elements with the value we want, we'll just remove the first one.

Finding the node in question is a matter of traversing the list and looking at each node's element.

Reconnecting the list once a node is to be removed is more interesting. Let's consider at least 3 cases:

* 1. Removing a node from the beginning.
  2. Removing a node from the middle.
  3. Removing a node from the end.

### Removing from the beginning

When removing the node at the beginning of the list, there is no relinking of nodes to be performed, since the first node has no preceding node. For example, removing node with **a**:

list

|

v

--------- --------- ---------

| a | --+---> | b | --+---> | c | 0 |

--------- --------- ---------

However, we must fix the pointer to the beginning of the list:

list

|

+-------------+

|

v

--------- --------- ---------

| a | --+---> | b | --+---> | c | 0 |

--------- --------- ---------

### Removing from the middle

Removing a node from the middle requires that the preceding node skips over the node being removed. For example, removing the node with **b**:

list

|

v

--------- --------- ---------

| a | --+--+ | b | --+---> | c | 0 |

--------- | --------- ---------

| ^

+----------------+

This means that we need some way to refer to the node**before**the one we want to remove.

### Removing from the end

Removing a node from the end requires that the preceding node becomes the new end of the list (i.e., points to nothing after it). For example, removing the node with **c**:

list

|

v

--------- --------- ---------

| a | --+---> | b | 0 | | c | 0 |

--------- --------- ---------

Note that the last two cases (middle and end) can be combined by saying that "the node preceding the one to be removed must point where the one to be removed does."

**IMPORTANT NOTE:** When there is only one node, and it is the one to be removed, are we in the case of removing at the **beginning** or **end**?! We should make sure that one of these cases handles removing this single node ELSE we should create a new case.

What we've learned from these pictures is that when we find the node we want to remove, we need some kind of reference to the node preceding it OR some way to backtrack to that node.

In addition, since we need to fix the pointer to the beginning of the list, we'll need some way to get the new beginning pointer out of the function. One way to do so (and the one we'll initially use) is to return the pointer from the deletion function. This means we will have a deletion function as follows:

nodeT \*ListDelete(nodeT \*list, elementT value);

and use it as:

list = ListDelete(list, value);

1. **Recursive implementation:**

One way to implement a deletion function is with recursion (since recursion gives us backtracking). Remember that for a recursive function (i.e., a function that calls itself), we need to decide:

* 1. Base case(s): when to stop recursing.
  2. Recursive part(s): how function will call itself to solve the problem.

The base cases are fairly easy, since we need to stop recursing when:

* 1. We find the matching value.
  2. We search the whole list and don't find a matching value.

To determine what should be the recursive part of the function, we should consider 2 tasks required of this function:

* 1. **Finding the one to remove.**

How will we search the list for the element? Well, the list will be passed as a pointer to a node (recall the prototype--here we've renamed the pointer to currP):

nodeT \*ListDelete(nodeT \*currP, elementT value);

So, if we decide that the node pointed to by currP is not the right node, then we just recurse to examine the next node:

ListDelete(currP->next, value);

and so on, until all nodes have been examined.

**Note:** Arrow (->), of course, is the syntax to use when you want to access a field in a structure using a pointer (like currP) to that structure.

* 1. **Relink the list.**

When we remove a node, the previous node has to skip over the removed node. A way to achieve this with recursion is through pointer reassignment.

Consider the following, where the current call to ListDelete() gets a pointer to the node with **x**:

1. ListDelete(pointer-to-node-with-x, 'y');

--------- --------- ---------

| x | --+---> | y | --+---> | z | --+--->

--------- --------- ---------

^

|

currP (in 1.)

Since **x** is NOT the thing to be removed, another recursive call is made. This next recursive call is going to access the node with **y**:

1. ListDelete(pointer-to-node-with-x, 'y');

2. ListDelete(pointer-to-node-with-y, 'y');

--------- --------- ---------

| x | --+---> | y | --+---> | z | --+--->

--------- --------- ---------

^ ^

| |

currP (in 1.) currP (in 2.)

Remember that when the recursive call on the node with **y** (call 2) is done, we'll be back to the call on the node with **x** (i.e., we backtrack to call 1):

1. ListDelete(pointer-to-node-with-x, 'y');

2. call 2 done, back to 1.

--------- --------- ---------

| x | --+---> | y | --+---> | z | --+--->

--------- --------- ---------

^

|

currP (in 1.)

Now since **y** is the thing to be removed...

If there was some way that this function call on the node with **y** (call 2) could tell us the node with **y** was removed and that it pointed to the node with **z**, then we could fix the node with **x**, having it skip over **y** (to **z**).

Well, it can!! Remember that each recursive call returns a pointer to a node!

Thus, what we'll do is have each recursive call return a pointer to the node to which the previous node should point. Thus, when the node is left alone, we just return a pointer to itself (i.e., "Don't skip over me!"). When the node is removed, we return a pointer to the next node (i.e., "Skip over me!").

So, we use the return value of the recursive call as:

currP->next = ListDelete(currP->next, value);

When the next node is not removed, this pointer reassignment is equivalent to if we had done:

currP->next = currP->next;

i.e., no change. However, if the next node is the one to be removed, it would be like we had done:

currP->next = currP->next->next;

and thus, skips over the removed node.

**Note:** This also works for removing the first node (i.e., it fixes the pointer to the beginning of the list), since the initial call to the function is:

list = ListDelete(list, value);

Again, to complete the function, all the things we'll have to do are:

* 1. Determine when we hit the end of the list.
  2. Determine when we've found the right node.
  3. Deallocate the correct node.
  4. Return what the link in the previous call should point to.

Finally, our whole recursive function looks like (we named the pointer currP since it will traverse the list instead of just pointing to the beginning):

nodeT \*ListDelete(nodeT \*currP, elementT value)

{

/\* See if we are at end of list. \*/

if (currP == NULL)

return NULL;

/\*

\* Check to see if current node is one

\* to be deleted.

\*/

if (currP->element == value) {

nodeT \*tempNextP;

/\* Save the next pointer in the node. \*/

tempNextP = currP->next;

/\* Deallocate the node. \*/

free(currP);

/\*

\* Return the NEW pointer to where we

\* were called from. I.e., the pointer

\* the previous call will use to "skip

\* over" the removed node.

\*/

return tempNextP;

}

/\*

\* Check the rest of the list, fixing the next

\* pointer in case the next node is the one

\* removed.

\*/

currP->next = ListDelete(currP->next, value);

/\*

\* Return the pointer to where we were called

\* from. Since we did not remove this node it

\* will be the same.

\*/

return currP;

}

This recursive implementation even works when we try to delete from an empty list (i.e., there are no nodes).

As mentioned, the initial call to this function will be:

list = ListDelete(list, value);

which will make sure that the pointer to the beginning gets fixed if we remove the first node.

1. **Iterative implementation:**

Now that we've seen how to delete a node using recursion, let's look at how to do the same thing iteratively (i.e., with a loop).

Again, we'll need to be able to change the pointer to the list, i.e., list, in the case that the first node in the list is removed. However, instead of passing back a new value for the beginning of the list via the return mechanism, as in:

list = ListDelete(list, value);

we'll pass in the pointer to the beginning of the list by reference (as a pointer to a pointer), so that we can change it in ListDelete() when necessary. Thus, the prototype for our function will be:

void ListDelete(nodeT \*\*listP, elementT value);

**Note:** This time, we pass the address of the "list" variable to the function, so it makes sense to call the parameter that receives that address "listP", since it is a pointer to a "list".

Before, we used recursion to search through the list, but now we will interate (i.e., loop) through it. It's easy to see how to start a pointer at the beginning and move it from one node to the next:

--------- --------- ---------

| x | --+---> | y | --+---> | z | --+--->

--------- --------- ---------

^

|

currP

In other words:

* 1. Start at the beginning:
  2. currP = \*listP

Note that to access the address of the first node, we have to dereference listP with the star (\*) since listP is a "pointer to a pointer to the first node" (and we want to remove one level of pointer-ness to get the "pointer to the first node").

* 1. Advance to the next node when necessary:
  2. currP = currP->next
  3. Stop when there are no more nodes, i.e., make sure that:
  4. currP != NULL

It's easy to combine these into a for loop:

for (currP = \*listP; currP != NULL; currP = currP->next) {

**...**

However, when we find the one to remove, we'll also need a pointer to the previous node:

--------- --------- ---------

| x | --+---> | y | --+---> | z | --+--->

--------- --------- ---------

^ ^

| |

prevP currP

Thus, we also need to maintain a previous pointer at each step of the loop:

for (currP = \*listP;

currP != NULL;

prevP = currP, currP = currP->next) {

(Notice that there are 2 increments separated by a comma.)

To complete the function, we'll have to:

* 1. Have some way to indicate when there is no previous node.  
     Remember that removing the first node is a special case because...
     + It requires no relinking of nodes.
     + We have to update the pointer to the beginning of the list. We don't have to return the new pointer to the beginning (in case it was changed) since we have direct access to it in the function (via*listP*).
  2. Relink the list (i.e., skip over the node to be deleted).
  3. Remove the node.

One implementation of this function might be:

void ListDelete(nodeT \*\*listP, elementT value)

{

nodeT \*currP, \*prevP;

/\* For 1st node, indicate there is no previous. \*/

prevP = NULL;

/\*

\* Visit each node, maintaining a pointer to

\* the previous node we just visited.

\*/

for (currP = \*listP;

currP != NULL;

prevP = currP, currP = currP->next) {

if (currP->element == value) { /\* Found it. \*/

if (prevP == NULL) {

/\* Fix beginning pointer. \*/

\*listP = currP->next;

} else {

/\*

\* Fix previous node's next to

\* skip over the removed node.

\*/

prevP->next = currP->next;

}

/\* Deallocate the node. \*/

free(currP);

/\* Done searching. \*/

return;

}

}

}

We do handle the situation of removing the first node, i.e., when there is no previous. See how we use a previous pointer value of NULL to indicate this and do the right thing.

Does this function handle deletions from empty lists correctly?

Finally, this function will be called as:

ListDelete(&list, value);

Note that the address of the list must be passed.