Linked Lists: Deleting Nodes

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Linked Lists: Basic Operations

- Operations Performed on Linked Lists
  - Several operations can be performed on linked lists
    - Add a new node
    - Delete a node
    - Search for a node
    - Counting nodes
    - Modifying nodes
    - and more
  - We will build functions to perform these operations

Linked Lists: Deleting Nodes

- General Approach:
  - You must search for the node that you want to delete (remember, we are using sorted lists)
  - If found, you must delete the node from the list
  - This means that you change the various link pointers
    - The **predecessor** of the deleted node must point to the deleted nodes **successor**
  - Finally, the node must be physically deleted from the heap
    - You must `free` the node

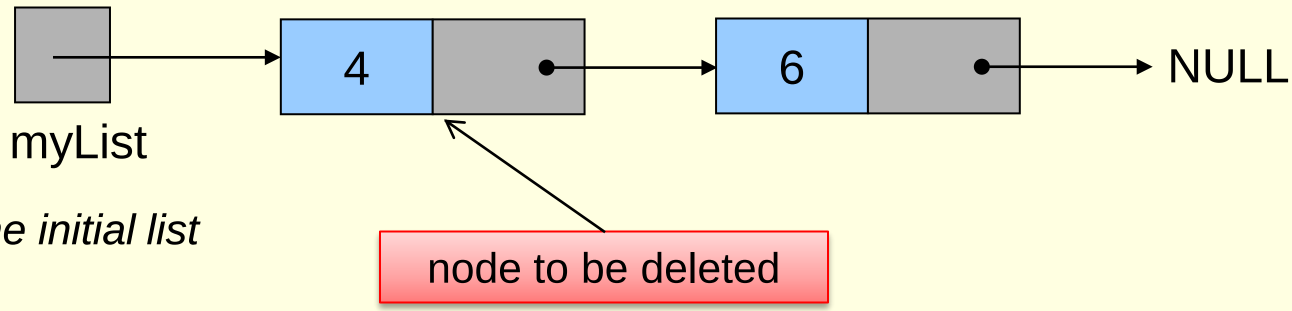
Linked Lists: Deleting Nodes

- General Approach:
  - There are 4 deletion scenarios:

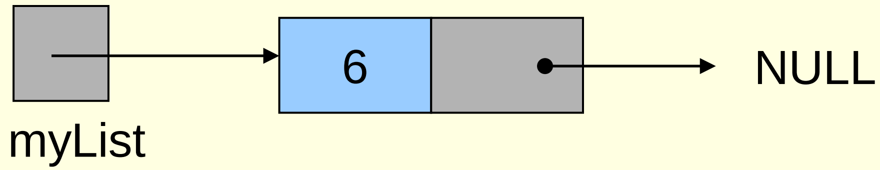
1) Delete the first node of a list
2) Delete any middle node of the list
   - Not the first node or the last node
3) Delete the last node of the list
4) A special case when we delete the only node in the list
   - Causes the resulting list to become empty

Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  1) Delete the first node of a list



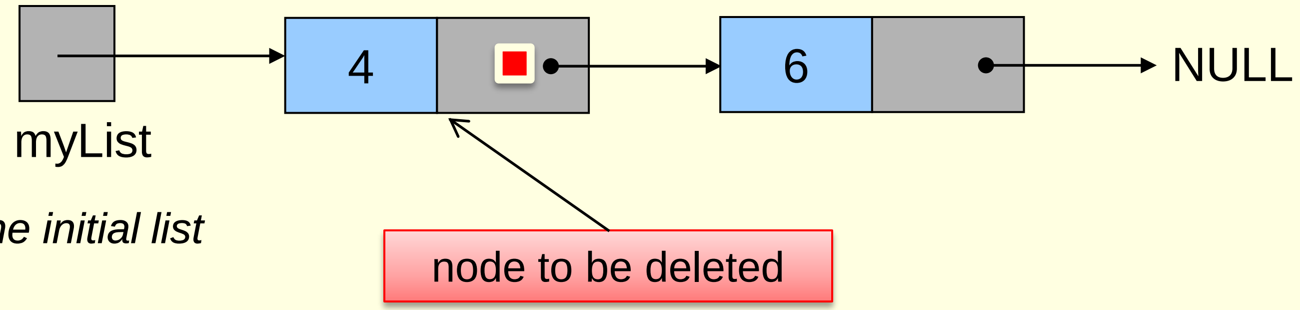
The initial list



The list after deleting the first node

Linked Lists: Deleting Nodes

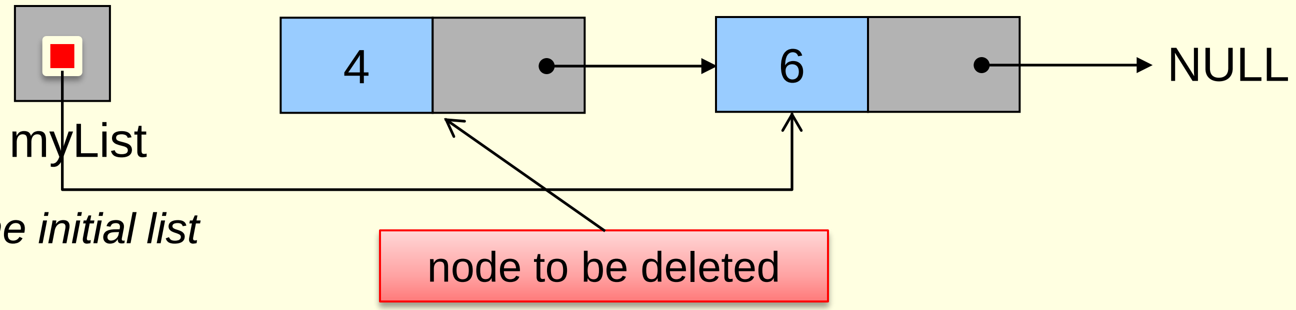
- 4 Cases of Deletion:
  1) Delete the first node of a list
     - Think about how you make this happen:
       - `myList` needs to point to the 2^{nd} node in the list
       - So we save the address of the 2^{nd} node into `myList`
       - Where do we get that address:
         - It is saved in the "`next`" of the first node
       - So we take that address and save it into `myList`



The initial list

Linked Lists: Deleting Nodes

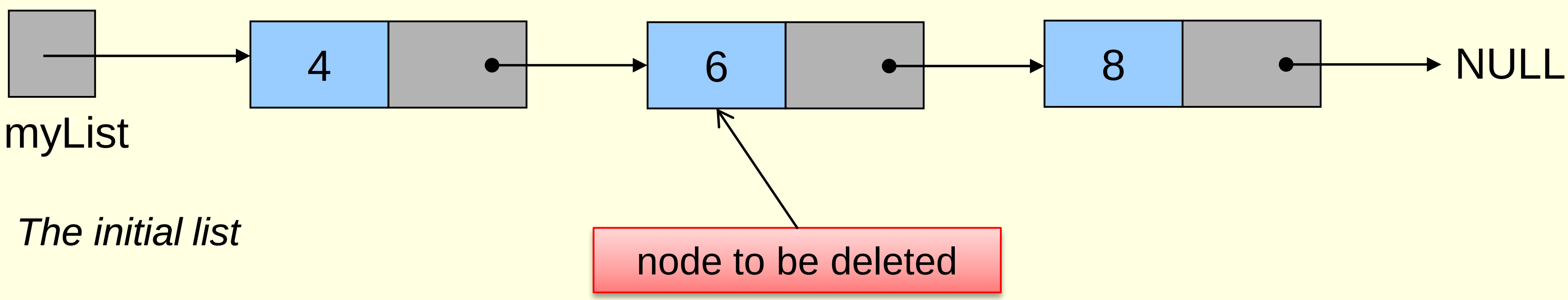
- 4 Cases of Deletion:
  1) Delete the first node of a list
     - Think about how you make this happen:
       - `myList` needs to point to the 2nd node in the list
       - So we save the address of the 2nd node into `myList`
       - Where do we get that address:
         - It is saved in the “`next`” of the first node
       - So we take that address and save it into `myList`
       - Finally, we free the 1st node



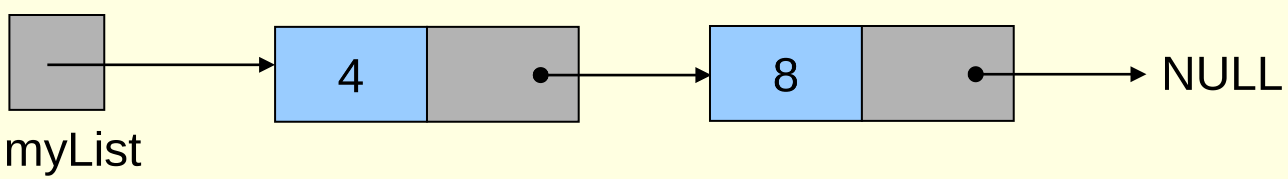
The initial list

Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  2) Delete any middle node of the list



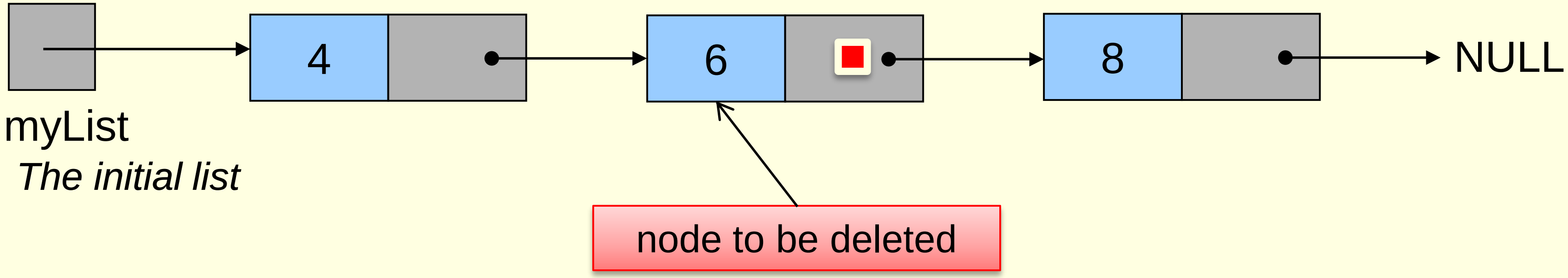
The initial list



The list after deletion has occurred

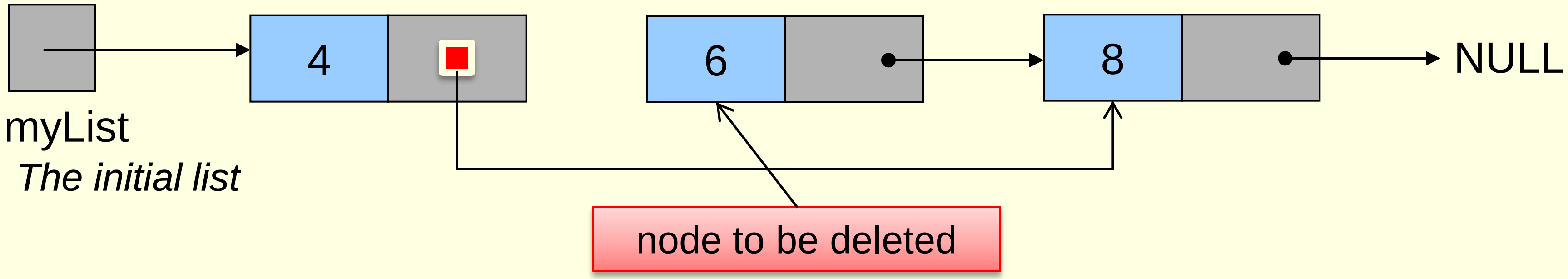
Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  2) Delete any middle node of the list
     - Think about how you make this happen:
       - Node # 4 (with 4 as data) needs to point to Node # 8
       - So we save the address of Node #8 into "`next`" of Node # 4
       - Where do we get the address of Node #8?
         - It is saved in the "`next`" of Node # 6!
       - So we take that address and save it to the "`next`" of Node # 4



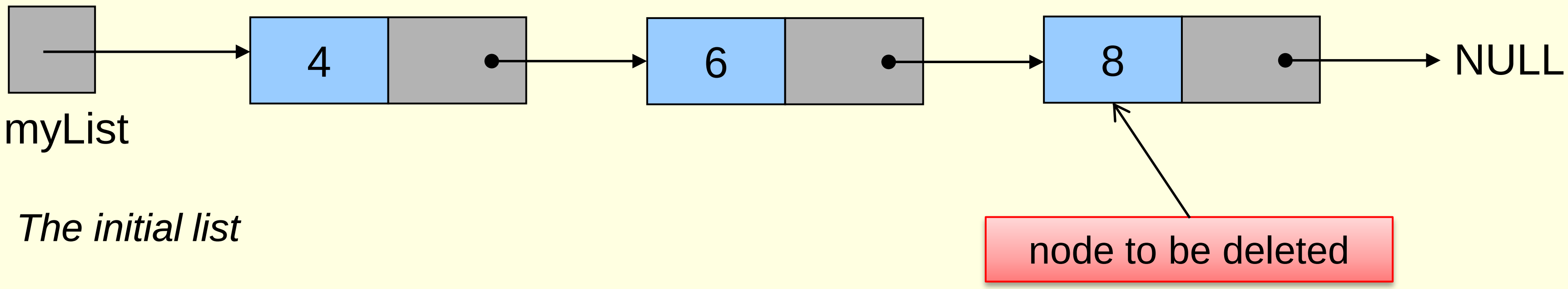
Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  2) Delete any middle node of the list
     - Think about how you make this happen:
       - Node # 4 (with 4 as data) needs to point to Node # 8
       - So we save the address of Node #8 into "`next`" of Node # 4
       - Where do we get the address of Node #8?
         - It is saved in the "`next`" of Node # 6!
       - So we take that address and save it to the "`next`" of Node # 4
       - Finally, we `free` Node # 6

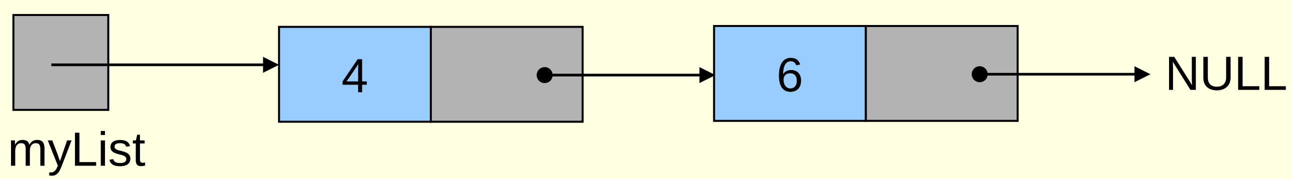


Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  3) Delete the last node of the list



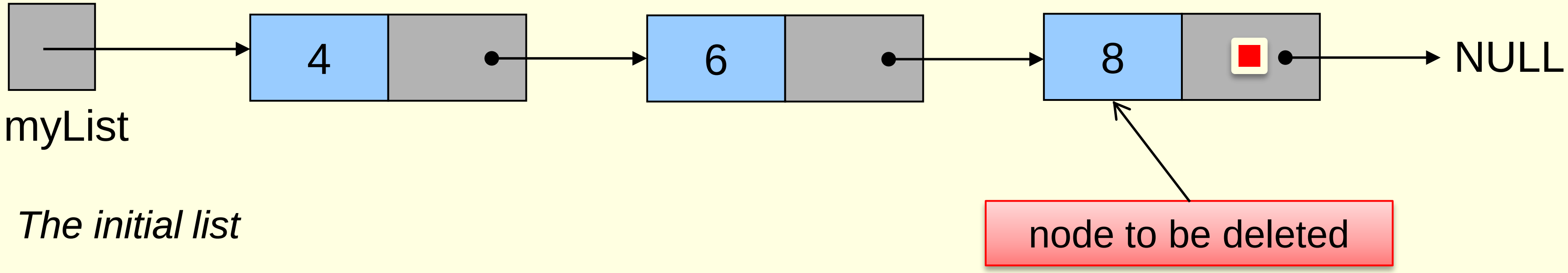
The initial list



The list after deletion has occurred

Linked Lists: Deleting Nodes

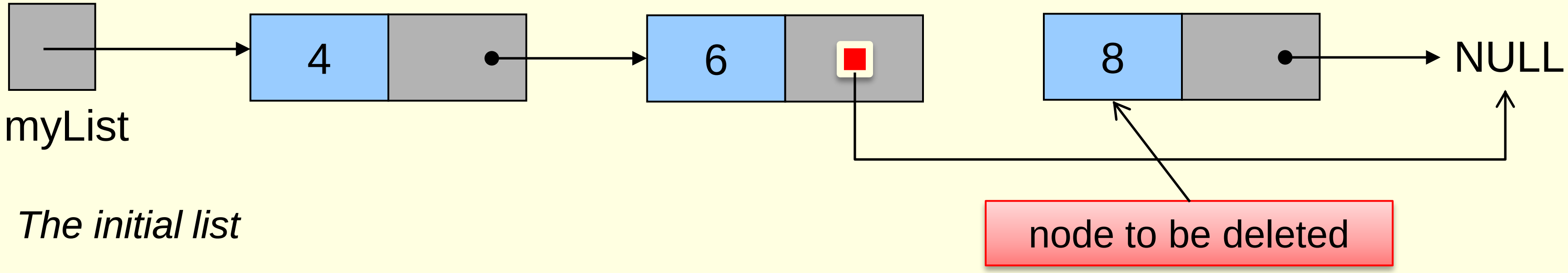
- 4 Cases of Deletion:
  3) Delete the last node of the list
     - Think about how you make this happen:
       - We simply need to save `NULL` to the "`next`" of Node # 6
         - This bypasses Node # 8
       - Where is `NULL` currently saved?
         - In the "`next`" of Node # 8
       - So take that value (`NULL`) and save into the "`next`" of Node #6



The initial list

Linked Lists: Deleting Nodes

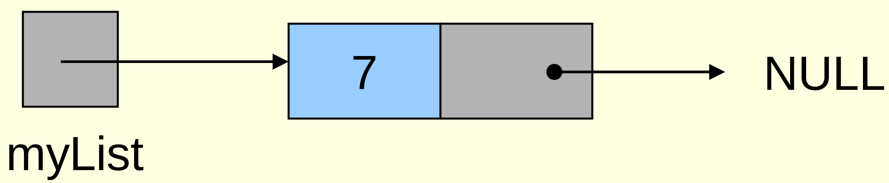
- 4 Cases of Deletion:
  3) Delete the last node of the list
     - Think about how you make this happen:
       - We simply need to save NULL to the "next" of Node # 6
         - This bypasses Node # 8
       - Where is NULL currently saved?
         - In the "next" of Node # 8
       - So take that value (NULL) and save into the "next" of Node #6
       - Finally, we free Node # 8



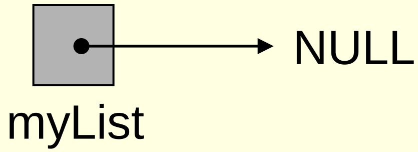
The initial list

Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  4) A special case when we delete the only node in the list



The initial list

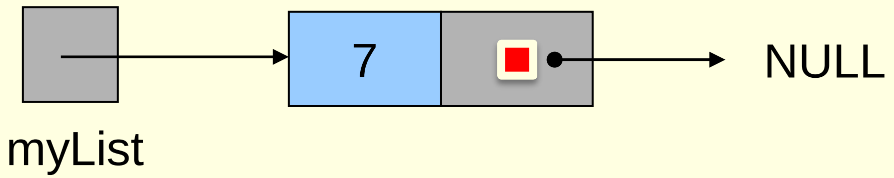


The list after deleting the only node.

This is a special case only in the sense that the head pointer value, which is returned to the function, will be NULL instead of pointing to a valid node.

Linked Lists: Deleting Nodes

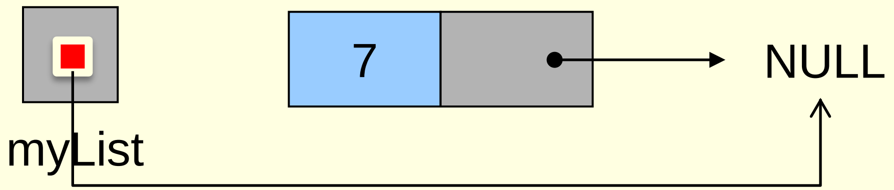
- 4 Cases of Deletion:
  4) Special case: deleting the only node in the list
     - Think about how you make this happen:
       - We simply need to save `NULL` into `myList`
         - This bypasses Node # 7
       - Where is `NULL` currently saved?
         - In the "`next`" of Node # 7
       - So take that value (`NULL`) and save into `myList`



The initial list

Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  4) Special case: deleting the only node in the list
     - Think about how you make this happen:
       - We simply need to save `NULL` into `myList`
         - This bypasses Node # 7
       - Where is `NULL` currently saved?
         - In the “`next`” of Node # 7
       - So take that value (`NULL`) and save into `myList`
       - Finally, we `free` Node # 7



The initial list

Brief Interlude: Human Stupidity

Deleting Nodes (code)

```
// Function Prototype:
struct ll_node* delete(struct ll_node *list, int target) ;

int main( ) {
        int number = 0;
        // We assume that we already created a valid list (myList).
        // There are several nodes already in myList.
        // This is just a cheesy while loop to call delete function
        while(number!= -1) {
                // Get the next number.
                printf("Enter data that you wish to delete:  ");
                scanf("%d", &number);

                // Delete node from linked list if number is not -1.
                if (number !=-1)
                        myList = delete(myList, number);
        }
        return 1;
}
```

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
                while (help_ptr->next != NULL) {
                        if (help_ptr->next->data == target) {
                                node2delete = help_ptr->next;
                                help_ptr->next = help_ptr->next->next;
                                free(node2delete);
                                return list
                        }
                        help_ptr = help_ptr->next;
                }
        }
        return list;
}
```

Now let's look at this code in detail.

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int value) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
```

- **In detail:**
  - We make two pointers of type `ll_node`:
    - `help_ptr` and `node2delete`
    - We should all know what `help_ptr` is for
      - Traversing our list
    - `node2delete` will be used later in the program
      - When deleting from the middle or end of a list
        - `node2delete` will be used to point to the node we want to delete
        - We can then `free` it accordingly
  - We then save `list` into `help_ptr`
    - Remember, `list` points to the first node of the list
    - We take the address that is stored in `list` and save into `help_ptr`
      - Thus making `help_ptr` also point to the same first node

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
```

- **<u>In detail:</u>**
  - We can only delete a node if there are nodes in the list!
  - Right.?.
  - So if there are no nodes in the list, there is nothing to delete
  - That's what this line checks for
  - if `help_ptr` does equal `NULL`, then the list is empty
  - So:
    - The ONLY time we delete (enter into this IF statement) is when:
    - `help_ptr != NULL`
    - Meaning, there are node(s) in the list

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
```

- **In detail:**
  - Examine this `IF` statement
    - At this point, `help_ptr` is pointing to the front of the list
    - So this says, if our target is found within this first node
      - Execute the 3 lines within this `IF` statement
    - So this if statement is specifically checking if we are deleting the FIRST node in the list

Deleting Nodes (code)

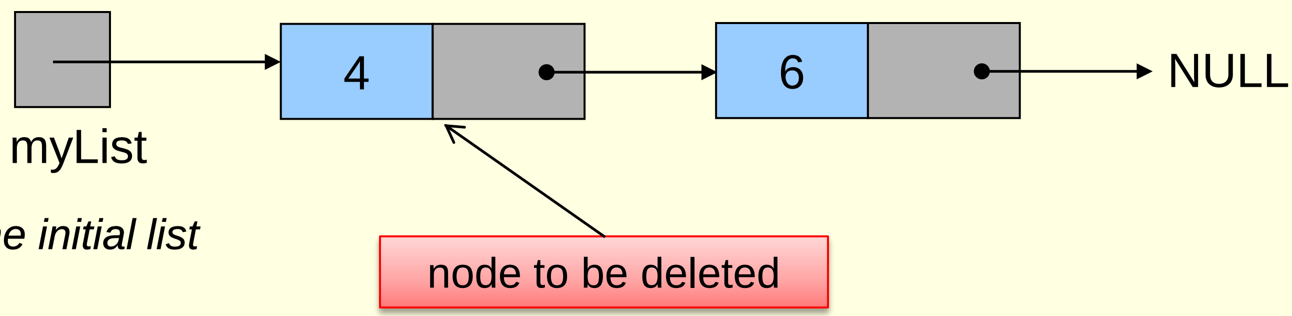
```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
```

- **<u>In detail:</u>**
  - So `IF` this is the case (we are deleting the first node):
    - Take whatever the first node points to and save it into `list`
      - Remember, `help_ptr` is pointing to the first node!
      - Take the address saved in `help_ptr->next` and save into `list`
    - So now, `list` will point to the second node in the list
      - If there were multiple nodes
    - OR `list` will point to `NULL`
      - If the list only had one node

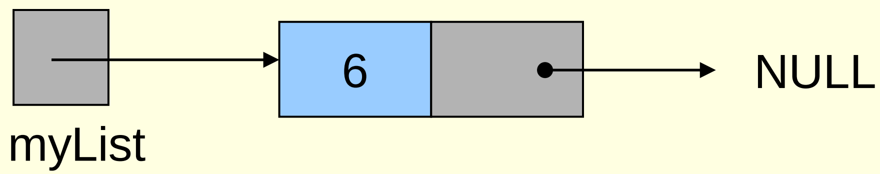
Either way, we effectively bypassed the first node!

Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  1) Delete the first node of a list



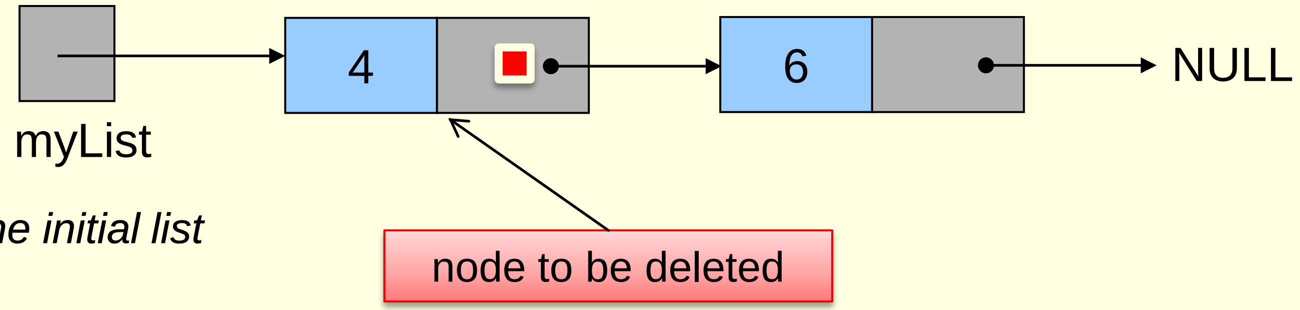
The initial list



The list after deleting the first node

Linked Lists: Deleting Nodes

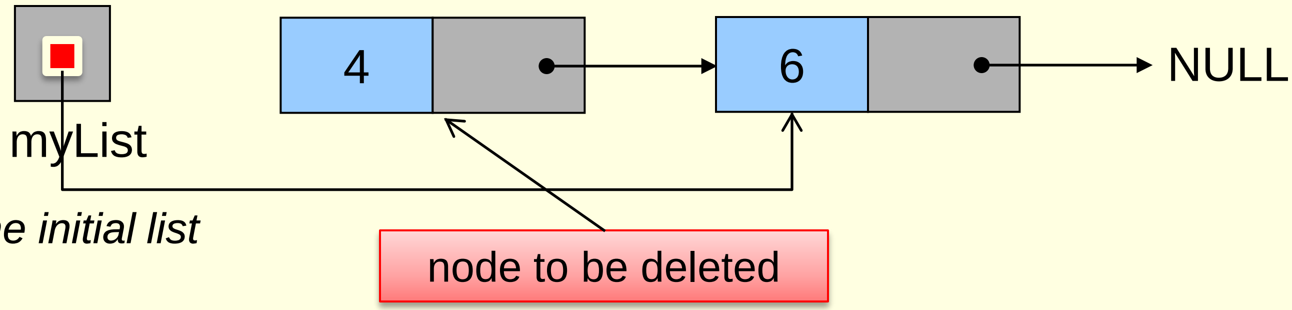
- 4 Cases of Deletion:
  1) Delete the first node of a list
     - Think about how you make this happen:
       - `myList` needs to point to the 2nd node in the list
       - So we save the address of the 2nd node into `myList`
       - Where do we get that address:
         - It is saved in the "`next`" of the first node
       - So we take that address and save it into `myList`



The initial list

Linked Lists: Deleting Nodes

- 4 Cases of Deletion:
  1) Delete the first node of a list
     - Think about how you make this happen:
       - `myList` needs to point to the 2nd node in the list
       - So we save the address of the 2nd node into `myList`
       - Where do we get that address:
         - It is saved in the “`next`” of the first node
       - So we take that address and save it into `myList`
       - Finally, we free the 1st node



The initial list

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
```

- **In detail:**
  - So `IF` this is the case (we are deleting the first node):
    - Take whatever the first node points to and save it into `list`
      - Remember, `help_ptr` is pointing to the first node!
      - Take the address saved in `help_ptr->next` and save into `list`
    - So now, `list` will point to the second node in the list
      - If there were multiple nodes
    - OR `list` will point to `NULL`
      - If the list only had one node

Either way, we effectively bypassed the first node!

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
```

- **In detail:**
  - So `IF` this is the case (we are deleting the first node):
    - Now, think, we just bypassed that first node
      - **But** that first node is still there in memory
    - So we MUST `free` the space allocated to it
      - If you remember, `help_ptr` is still pointing to that first node
      - Although no part of the list is pointing to it
      - We use the `free` command to `free` the space pointed to by `help_ptr`
    - Finally, we return the `list` to main

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
                while (help_ptr->next != NULL) {
```

- **<u>In detail:</u>**
  - The previous IF statement was used to check if the node to be deleted was at the FRONT of the list
  - So now, if we made it this far (to the while loop), we know the node is NOT at the front of the list
  - So we must traverse the list looking for the node to delete
    - And then we delete it!

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
                while (help_ptr->next != NULL) {
```

- **<u>In detail:</u>**
  - Specifically, this `while` loop checks to make sure that the `next` of `help_ptr` is not `NULL`
  - Why?
    - Cause if it is `NULL`, then we've reached the end of the list
  - So we continue this `while` loop possibly all the way to the end of the list

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
                while (help_ptr->next != NULL) {
```

- **<u>In detail:</u>**
  - Additionally, within this `while` loop:
    - We will be checking the data value at one node AFTER where `help_ptr` points to
    - We MUST make sure that `help_ptr->next` does not equal `NULL`
    - Cuz if it does equal `NULL` and we try to check the data of a node that doesn't exist, we will get an error!

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        return list;
                }
                while (help_ptr->next != NULL) {
                        if (help_ptr->next->data == target) {
                                node2delete = help_ptr->next;
                                help_ptr->next = help_ptr->next->next;
                                free(node2delete);
                                return list
                        }
                        help_ptr = help_ptr->next;
                }
```

Now let’s look at this while loop in detail.

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - There are 2 main parts of this `while` loop:
    - The `IF` statement
      - Checks to see if that particular node has the target value
        - Meaning, this is the node we want to delete
      - If found, we delete, we RETURN to main, and we exit the delete function
    - Now, if we do NOT enter the `IF` statement (`target` not found)
      - We step one node over to the next node in the list and continue the loop

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now let's examine the actual `IF` statement:
    - What is obvious is that we are checking if some `data` value is equal to `target`
    - But what `data` value? **<u>Or what node?</u>**
    - `help_ptr->next->data` says to look at the `data` value in the node IMMEDIATELY following the one that `help_ptr` points to

Deleting Nodes (code)

```
      // PREVIOUS CODE WAS HERE
            while (help_ptr->next != NULL) {
                  if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                  }
                  help_ptr = help_ptr->next;
            }
```

- **<u>In detail:</u>**
  - Now let's examine the actual `IF` statement:
    - Example:
      - If `help_ptr` is currently pointing to node # 87
      - Then `help_ptr->next->data` says to look at the `data` value at node # 88.
      - We compare this value to `target`

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now let's examine the actual `IF` statement:
    - So if our `target` is found at node # 12 (for example)
    - Does `help_ptr` point to that node?
      - NO!
    - At that point, `help_ptr` will be pointing to node # 11
    - `help_ptr->next` will be pointing to the node we want to delete

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **In detail:**
  - Now let's examine the actual `IF` statement:
    - So again, the `IF` statement says:
    - `IF` the `data` at the node FOLLOWING the one that `help_ptr` points to is equal to our `target` value
      - Then we enter the `IF` statement and execute those four lines of code

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now look at the code inside the `IF` statement (`target` found)
    - `help_ptr->next` is pointing to the node we want to delete
    - We will need to `free` that memory
    - At fist glance, you may think we could just type
      - `free(help_ptr->next)`
    - Would that work? And if so, what problem arises?

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now look at the code inside the `IF` statement (`target` found)
    - If we immediately type `free(help_ptr->next)`
      - That will delete the correct node!
    - BUT, remember, we need to make the connections <u>from the node before it to the node after it</u>
    - ONLY way to reference the node after it is via `help_ptr->next`

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
       while (help_ptr->next != NULL) {
              if (help_ptr->next->data == target) {
                     node2delete = help_ptr->next;
                     help_ptr->next = help_ptr->next->next;
                     free(node2delete);
                     return list
              }
              help_ptr = help_ptr->next;
       }
```

- **In detail:**
  - Now look at the code inside the `IF` statement (`target` found)
    - Example:
      - `help_ptr` points to node # 11
      - `help_ptr->next` points to node # 12 (the node we want to delete)
      - Of course, node # 12 is linked to node # 13
      - And once we delete node # 12, node # 11 must link to node # 13
      - If we go ahead and delete node # 12, what happens?

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now look at the code inside the `IF` statement (`target` found)
    - Example:
      - If we delete node # 12,
      - We will have lost our connection (`next` pointer) to node # 13
        - cuz that pointer is saved in the `next` of node # 12
      - Well why is that a problem?

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now look at the code inside the `IF` statement (`target` found)
    - Example:
      - This is a problem because node # 11 needs to point to node # 13
      - The address of node # 13 is saved in the `next` of node # 12
      - So if we delete node # 12 immediately, we lose that address

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now look at the code inside the `IF` statement (`target` found)
    - So we SAVE the address stored in `help_ptr->next` into the pointer we created earlier, `node2delete`
    - We will `free` that space in a bit
    - BUT first, we need to use that node to refer to the next node in the list (after the one to be deleted)

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **<u>In detail:</u>**
  - Now look at the code inside the `IF` statement (`target` found)
    - Look at the 2nd statement:
      - `help_ptr->next = help_ptr->next->next;`
    - This says, look TWO nodes AFTER where `help_ptr` points to
    - Take the address of that node and save it into `help_ptr->next`
    - What does this effectively do?

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **In detail:**
  - Now look at the code inside the `IF` statement (`target` found)
    - Look at the 2nd statement:
      - For example, say `help_ptr` points to node # 11.
      - Therefore, `help_ptr->next->next` points to node # 13
      - This line says take the address of node # 13 and store it in the `next` of node # 11. **This BYPASSES node # 12.**

Deleting Nodes (code)

```
// PREVIOUS CODE WAS HERE
        while (help_ptr->next != NULL) {
                if (help_ptr->next->data == target) {
                        node2delete = help_ptr->next;
                        help_ptr->next = help_ptr->next->next;
                        free(node2delete);
                        return list
                }
                help_ptr = help_ptr->next;
        }
```

- **In detail:**
  - Now look at the code inside the `IF` statement (`target` found)
    - Now that we're done updating the pointers
      - Meaning we no longer need the to-be-deleted node
    - We `free` the space allocated to that node
    - And finally, we RETURN the head pointer (`list`) to main

Deleting Nodes (code)

```
struct ll_node* delete(struct ll_node *list, int target) {
        struct ll_node *help_ptr, *node2delete;
        help_ptr = list;
        if (help_ptr != NULL) {
                if (help_ptr->data == target) {
                        list = help_ptr->next;
                        free(help_ptr);
                        re
                }
                while (hel
                        if




                        }
                        he
                }
        }
        return list;
}
```

The last possible line to execute is this return list.

When does this execute?

Either:

a) When there are no nodes in the list from the beginning
   - Thus we never even enter the outer IF statement
b) We traversed the ENTIRE list within the while loop and could not find the node to delete

Linked Lists: Basic Operations

- What we've covered thus far:
  - Adding nodes
  - Deleting nodes
  - And in the process of both of these:
    - Searching a list for nodes
    - We did this when we traverse the list searching for our spot to insert/delete
  - Traversing a list
  - Printing a list
  - Guess what?
    - That just about covers it. You are ready for Program #2.

Linked Lists: Deleting Nodes

WASN'T THAT AMAZING!

Daily Demotivator

Linked Lists: Deleting Nodes

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I