# Linked Lists

Computer Science Department
University of Central Florida

*COP 3502 – Computer Science I*

# Announcements

- Program 1:
  - Due this Wednesday (5/25/11)

- Quiz 0:
  - As with ALL quizzes, the quiz is on Webcourses
  - Available starting at 2 PM today (5/24/11)
  - You have until 11:55 PM tonight (5/24/11) to finish and submit the Quiz.
  - As mentioned on day one, Quiz 0 is simply based on the syllabus and various class policies

# Linked Lists

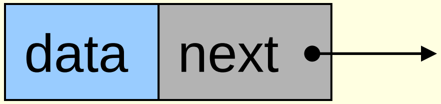
- What are they?
  - Abstraction of a list:  i.e. a sequence of nodes in which each node is linked to the node following it.
- Why not just use an array?
  - Each node in an array is stored, physically, in contiguous spaces in memory
    - Arrays are fixed size (not dynamic)
    - Inserting and deleting elements is difficult
    - In an array of size 100, if we insert an element after the 10th element, all the remaining 90 elements must be shifted.

# Linked Lists

- Why use linked lists?
  - They are dynamic; length can increase or decrease as necessary
  - Each node does not necessarily follow the previous one in memory
  - Insertion and deletion is cheap!
    - Only need to change a few nodes (at most)
- Is there a negative aspect of linked lists?
  - Getting to a particular node may take a large number of operations, as we do not know the address of any individual node

# Linked Lists

- In detail:
  - A linked list is an ordered collection of data
    - Each element (generally called nodes) contains  the location of the next element in the list
    - Each node essentially has two parts:
      - The data part
        - If this was a list of student records, for example, the data here may consist of a name, PID, social security number, address, phone, email, etc.
      - The link part
        - This link is used to chain the nodes together.
        - It simply contains a pointer variable that points to the next node in the linked list
        - Variable is often called "next"



# Linked Lists - Example

struct ll_node a, b, c;

a.data = 1;

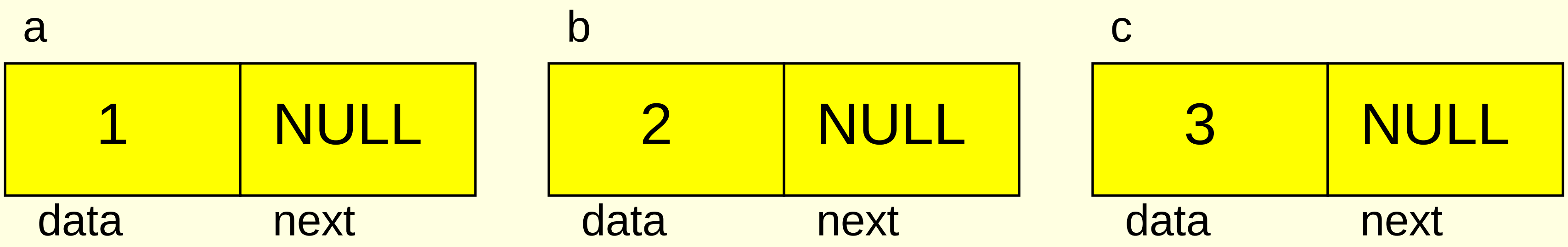
b.data = 2;

c.data = 3;

a.next = b.next = c.next = NULL;

Previous struct declaration:

```
struct ll_node {
        int data;
        struct ll_node *next;
};
```



a

| 1 | NULL |
|---|---|
| data | next |

b

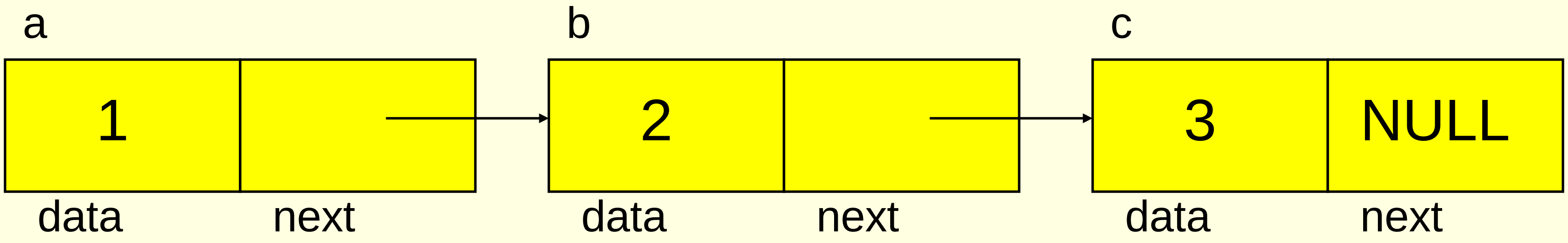
| 2 | NULL |
|---|---|
| data | next |

c

| 3 | NULL |
|---|---|
| data | next |

# Linked Lists – Example (cont.)

```
a.next = &b;
b.next = &c;
a.next -> data   has value 2
a.next -> next -> data   has value 3
b.next -> next -> data   error !!
```

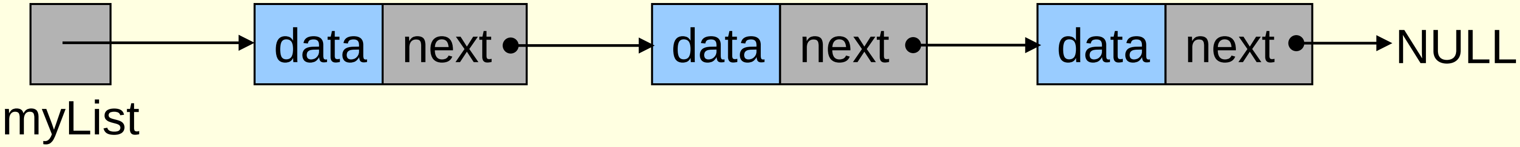


# Linked Lists

- In detail:
  - You can think of each node as a record
    - The first part of the record is all the necessary data
    - The final part of the record is a field that stores a pointer to the next node in the list
  - Head of the list
    - Each node of the list is created dynamically and points to the next node
      - So from the first node, we can get to the second, and so on
    - But how do you reach the first node?
    - You must have a pointer variable that simply points to the first node of the list
      - Simply called whatever you choose to name your list (myList)

# Linked Lists

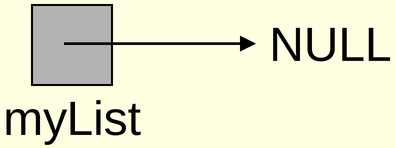
- Example of a Linked List
  - Don’t get confused over the “data” here
    - It could be simply an integer value
    - It could be 20+ separate fields of information storing name, address, phone, email, etc.



*A linked list containing three elements*

# Linked Lists

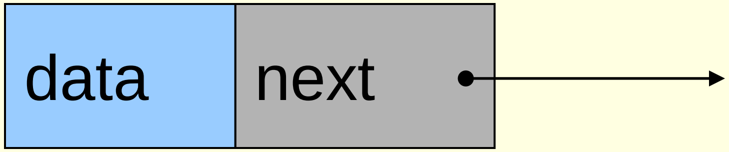
- Example of an Empty Linked List
  - This list is empty
  - There are no nodes (elements)
  - myList simply points to NULL
    - Which signifies an empty list



*An empty linked list*

# Nodes of a Linked List – Examples

- Linked List Nodes
  - Here’s a picture of a single linked list node



*For the sake of ease, data will simply be an int value in this example.

  - Here’s the struct that we would use to define this node

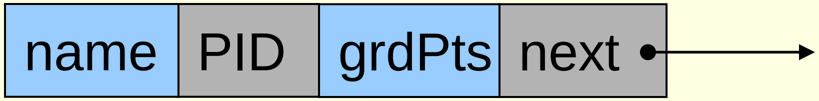
```
struct ll_node {
        int data;
        struct ll_node *next;
};
```

So what is `*next`?
It is a pointer of type `struct ll_node`.
It stores the address to the next node in the list.

# Nodes of a Linked List – Examples

- Linked List Nodes
  - A node with three data fields:

```
struct student_node {
      char name[20];
      char PID[8];
      double grdPts;
      struct student_node *next;
};
```

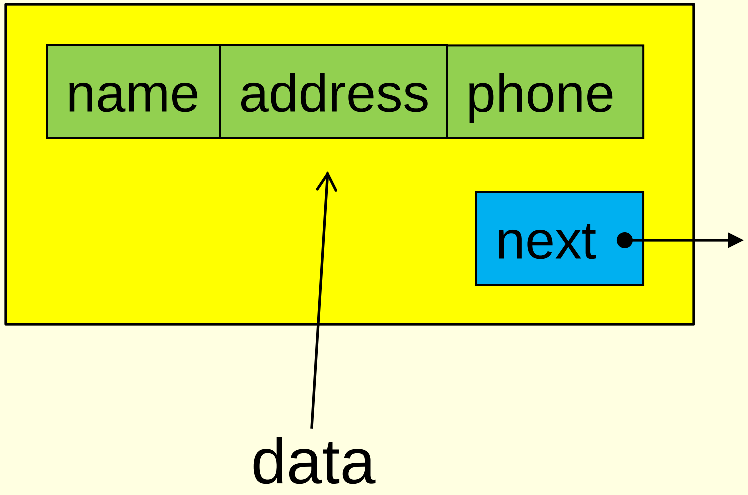


# Nodes of a Linked List – Examples

- Linked List Nodes
  - A `struct` within a node:

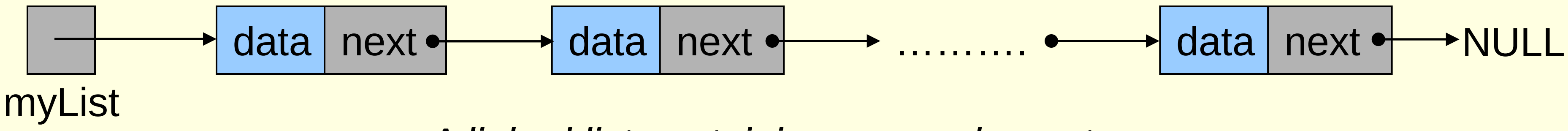
```
struct person{
      char name[20];
      char address[50];
      char phone[10];
};

struct person_node{
      struct person data;
      struct person_node *next;
};
```



# Linked Lists

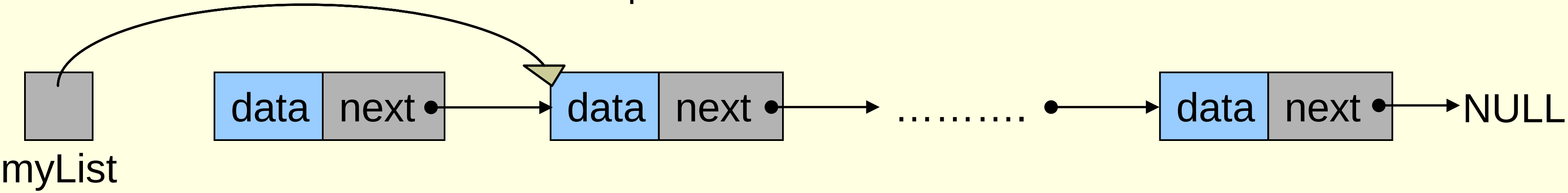
- Accessing Nodes of a Linked List
  - Let’s assume we already have a list created with several nodes
    - Don’t worry how we made this
      - We’ll get to that in a bit
  - We access the list via the head ptr, `myList`
    - How would you move to the $2^{nd}$ node in the list?



*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - One of the most common errors:
    - “moving” the head of the list to point to subsequent nodes
    - Consider if we made `myList` point to the second node
      - Instead of pointing to the first node
    - We would essentially lose access to the first node
      - Cause each node only points to the NEXT node
        - Not the previous one



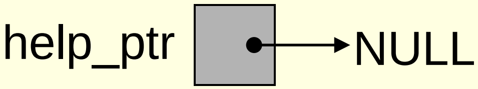
*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - How then do we traverse (walk down) a list?
  - We make a temporary `ll_node` pointer to help us move through the list

```
struct ll_node *help_ptr;
help_ptr = NULL;
```

  - It isn’t good to leave variables uninitialized
    - So we initialize `help_ptr` to NULL



# Linked Lists

- Accessing Nodes of a Linked List
  - We want `help_ptr` to traverse the list
    - So it needs access to the list
    - We use the following line:

```
help_ptr = myList;
```

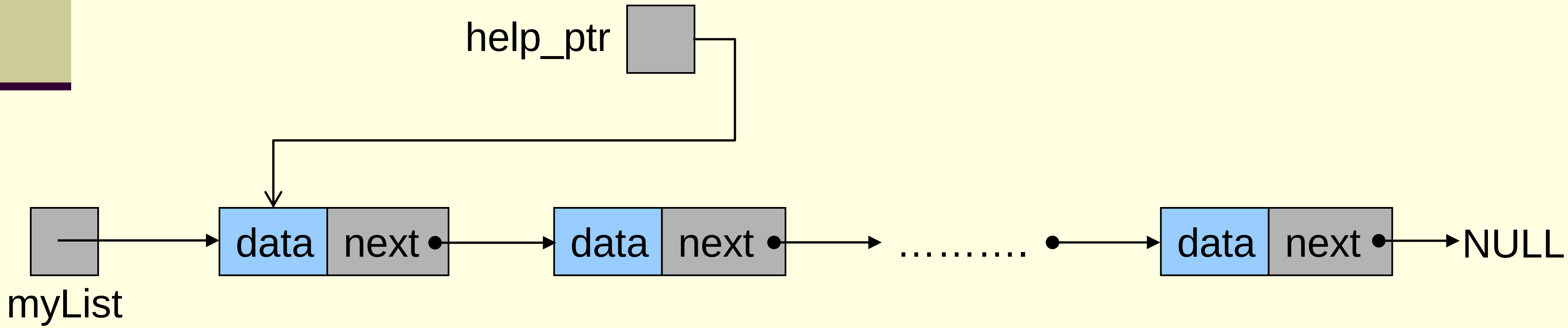
    - Remember that `myList` is a pointer of type `ll_node`
    - Also, `help_ptr` is a pointer of type `ll_node`
    - So this line basically says:
      - Take the address that is saved in `myList` (where `myList` points to)
      - And save that address into `help_ptr` (make `help_ptr` point to the same place)

# Linked Lists

- Accessing Nodes of a Linked List
  - We want `help_ptr` to traverse the list
    - So it needs access to the list
    - We use the following line:

```
help_ptr = myList;
```

  - Here's what our picture looks like now:

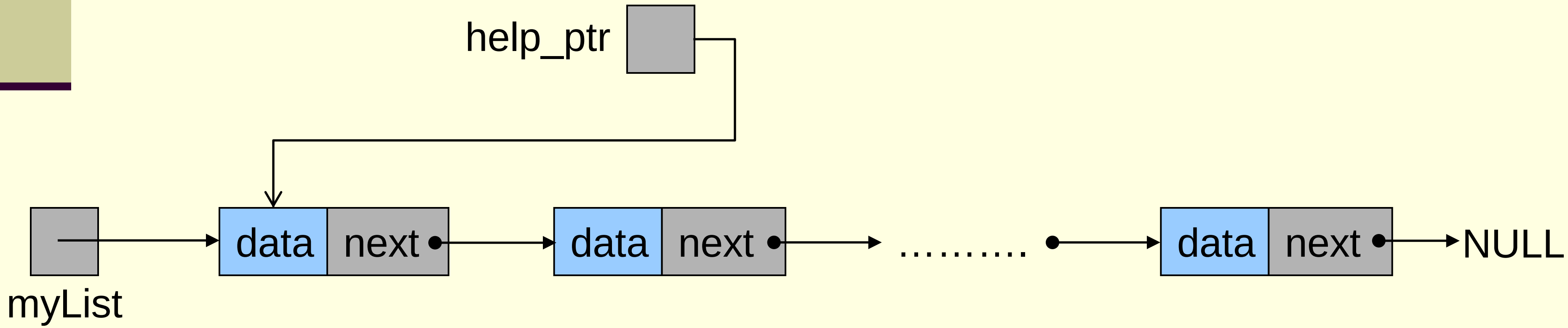


*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - Now, here's how we could access the "data" field of the first node in the list:

```
(*myList).data    OR   (*help_ptr).data
myList->data      OR   help_ptr->data
```



*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - Now, here's how we could access the "data" field of the first node in the list:

```
(*myList).data    OR   (*help_ptr).data
myList->data      OR   help_ptr->data
```

  - Few things to notice here:
    - Both of these expressions refer to the same exact `data` variable
      - since `myList` and `help_ptr` are pointing to the same exact node of the linked list
    - We use the dot operator to refer to a field within the record, as learned with structs in COP 3223

# Linked Lists

- Accessing Nodes of a Linked List
  - Now, here's how we could access the "data" field of the first node in the list:

```
(*myList).data    OR   (*help_ptr).data
myList->data      OR   help_ptr->data
```

  - Few things to notice here:
    - Remember that `myList` and `help_ptr` are NOT actual nodes. They are NOT of type `ll_node`.
      - Rather, they are both POINTERS of type `ll_node`
    - Therefore, in order to access the first node, we MUST dereference the pointer using the * symbol
      - Notice that `myList.data` is syntactically incorrect because `myList` is NOT of type `ll_node`

# Linked Lists

- Accessing Nodes of a Linked List
  - Now, here's how we could access the "data" field of the first node in the list:

```
(*myList).data   OR   (*help_ptr).data
```

**`myList->data       OR   help_ptr->data`**

  - Few things to notice here:
    - Finally, notice that the arrow, ->, provides a valid, alternative syntax
    - Most people find it easier to type
      - `help_ptr->data`
      - instead of
      - `(*help_ptr).data`

# Linked Lists

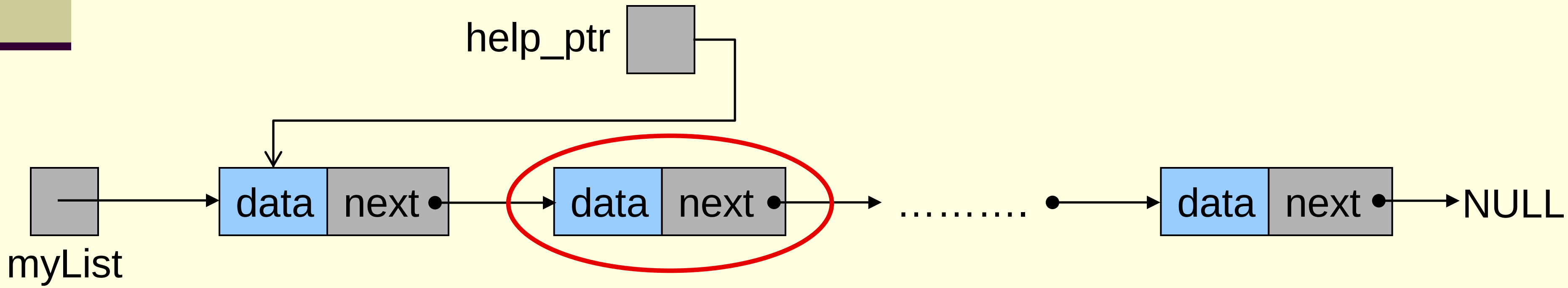
- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - We can use `help_ptr` to traverse the list pointed to by `myList`
    - Here would be the instruction to walk one node over:

```
help_ptr = help_ptr->next;
```

    - Note that the syntax here is correct
      - Why?
      - Cuz both sides of the assignment statement are pointers to struct `ll_node`
    - Let's now examine this statement in detail
      - And how it changes our picture

# Linked Lists

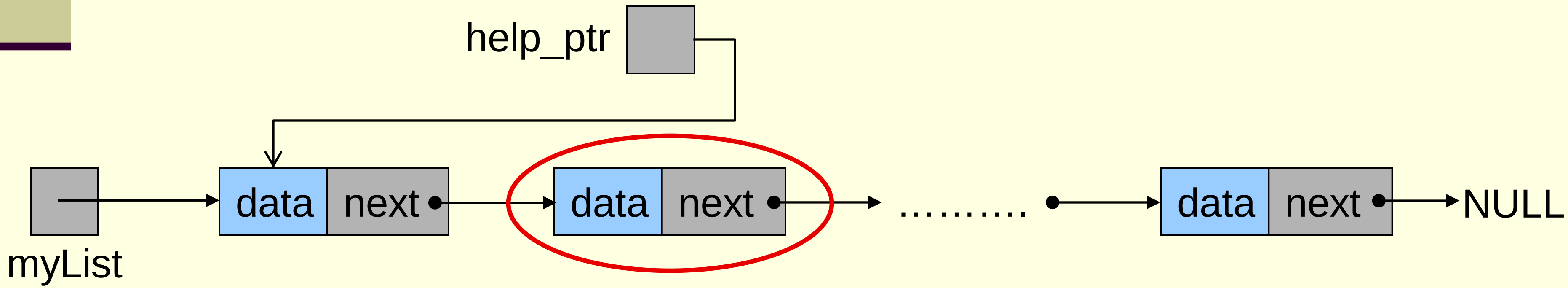
- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - Here's our before picture:
    - Remember, what is the goal here?
      - We want `help_ptr` to point to the second node in the list
    - The question is:
      - How do we accomplish this?



*A linked list containing many elements*

# Linked Lists

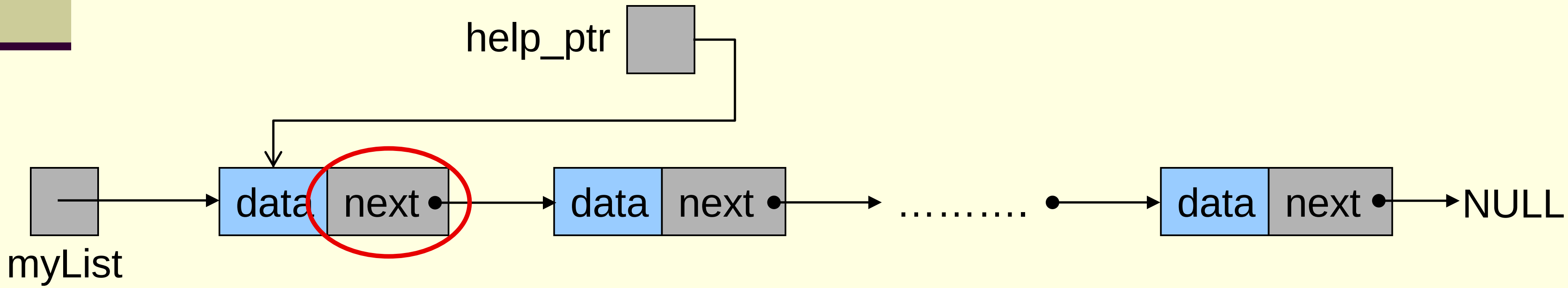
- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - Think:
      - That second node is located somewhere in memory
        - It has an address
      - Currently where is that address saved?
      - In other words, locate the pointer that is pointing to the second node



*A linked list containing many elements*

# Linked Lists

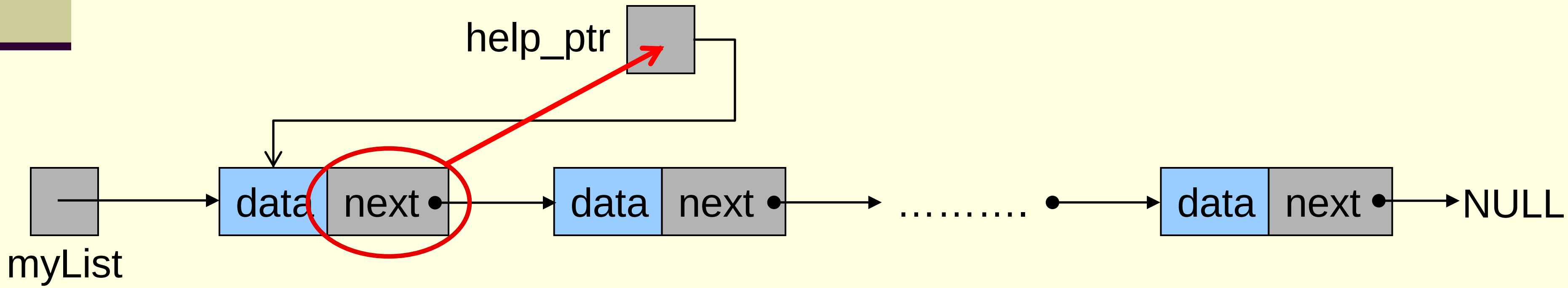
- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - Think:
      - The “next” pointer, of the first node, is currently pointing to the second node
      - And what is a pointer?  An address!
      - So the address of the second node is currently saved in the “next” pointer of the first node



*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - Remember:
      - We want `help_ptr` to point to the second node
      - So we need to take the address that is stored in the "<u>next</u>" of the <u>first node</u> and save it into `help_ptr`
      - This will make help_ptr point to the <u>second node</u>



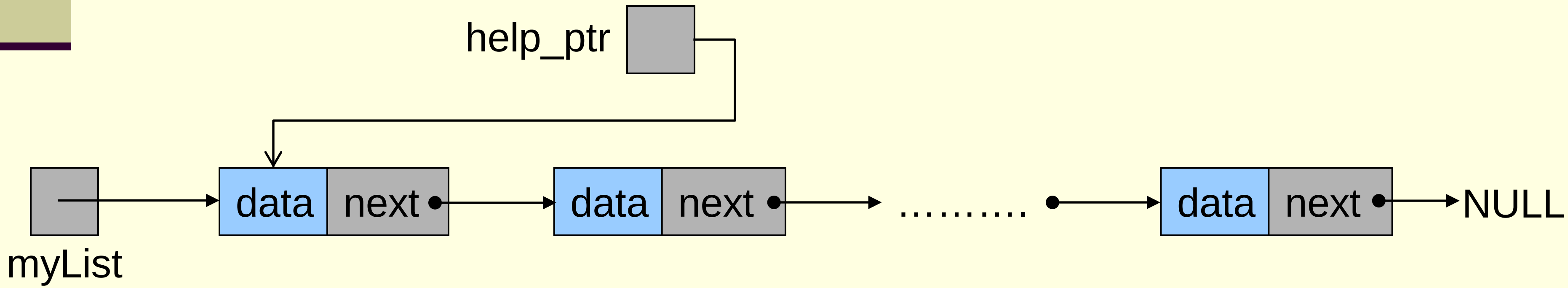
*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - Again, here's the instruction that does this:

```
help_ptr = help_ptr->next;
```

    - Here's how that statement changes our picture:



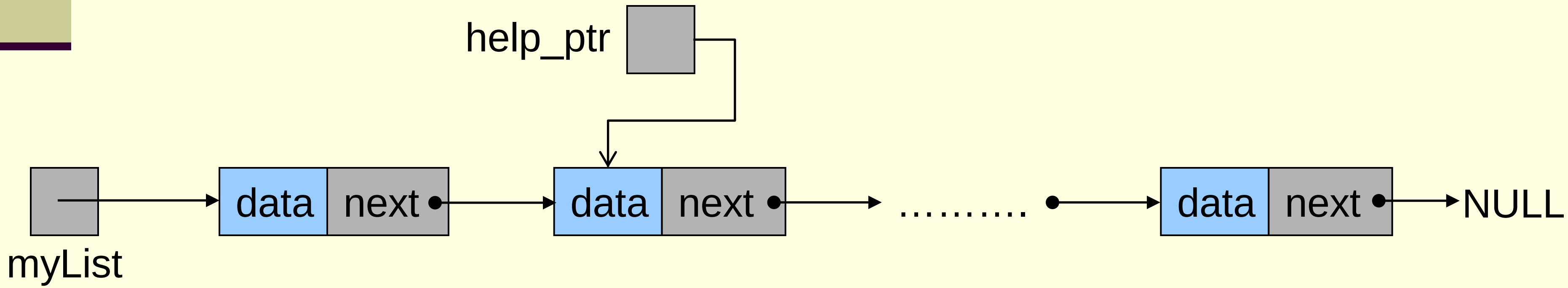
*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - Again, here's the instruction that does this:

```
help_ptr = help_ptr->next;
```

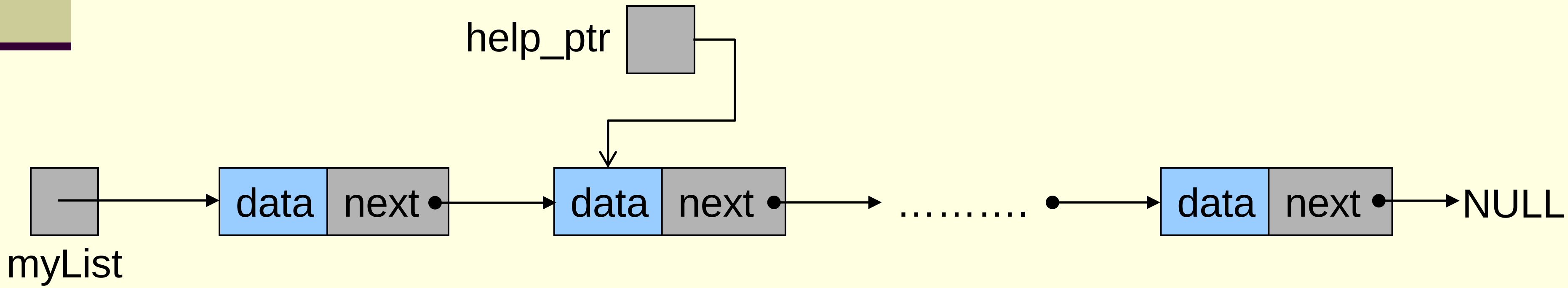
    - Here's how that statement changes our picture:



*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - Now we could refer to the data field of the second node as: `help_ptr->data`
    - We can also repeatedly use `help_ptr` in this fashion to iterate through the list



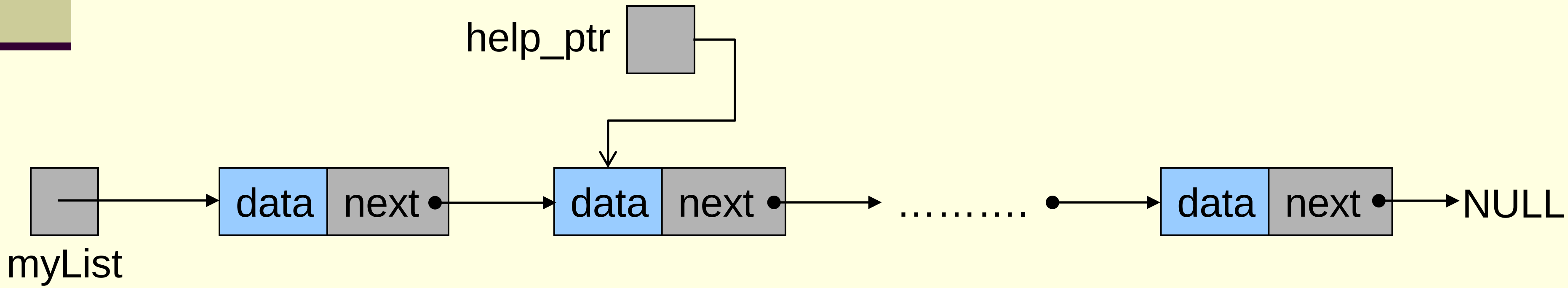
*A linked list containing many elements*

# Linked Lists

- Accessing Nodes of a Linked List
  - Traversing (moving through) the list
    - We could also modify the values in the list with a statement like:

```
help_ptr->data = 10;
```

    - This saves 10 into the data field of the second node
    - This sort of manipulation is handy for "editing" lists



*A linked list containing many elements*

# Brief Interlude:  Human Stupidity

# Traversing Linked Lists

- Traverse and Print out data of a linked list
  - **Assume** that `myList` is already pointing to a valid linked list of nodes of type `ll_node`
  - Here’s the code to **Traverse** a linked list:

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
       printf("%d ", help_ptr->data);
       help_ptr = help_ptr->next;
}
```

  - Let’s take a closer look

# Traversing Linked Lists

- Traverse and Print out data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;
```

  - Let’s take a closer look:
    - We start by making our `help_ptr`
    - `myList` is the pointer that points to our actual list
    - So we save this value into `help_ptr`
      - Which we use to traverse the list

# Traversing Linked Lists

- Traverse and Print out data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
```

  - Let’s take a closer look:
    - The while statement simply makes sure that we are pointing to a valid node
    - Because if `help_ptr` is NULL, we have reached the end of the list

# Traversing Linked Lists

- Traverse and Print out data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
      printf("%d ", help_ptr->data);
```

- Let’s take a closer look:
  - So while `help_ptr` is not NULL
    - Meaning, we are at a valid node of the linked list
    - We print out that particular node’s “data” field

# Traversing Linked Lists

- Traverse and Print out data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
       printf("%d ", help_ptr->data);
       help_ptr = help_ptr->next;
}
```

- Let’s take a closer look:
  - So while `help_ptr` is not NULL
    - We then move `help_ptr` over to the next node in the list
    - This is the SAME line of code that we went over in detail on earlier slides

# Traversing Linked Lists

- Traverse and Print out data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
	printf("%d ", help_ptr->data);
	help_ptr = help_ptr->next;
}
```

- Let’s take a closer look:
  - So while `help_ptr` is not NULL
    - We basically print a node’s data and then traverse down the list one step
    - We do this again, and again, and again, and …

# Traversing Linked Lists

- Traverse and Print out data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
      printf("%d ", help_ptr->data);
      help_ptr = help_ptr->next;
}
```

  - Let’s take a closer look:
    - At some point, we will reach the last node
      - The “next” value of that node will point to **NULL**
      - Which will get saved into `help_ptr`
      - Which will kick us out of this while loop

# Traversing Linked Lists

- Traverse and Print out data of a linked list
  - Food for thought:
    - Would the following code be valid if we didn't use the helper pointer node, `help_ptr`?
    - Yes, it would be valid
      - And it would traverse the list just fine

```
// myList is already pointing to
// a valid list

while (myList != NULL) {
      printf("%d ", myList->data);
      myList = myList->next;
}
```

# Traversing Linked Lists

- Traverse and Print out data of a linked list
  - Food for thought:
    - But what is the negative aspect of doing this?
      - In other words, why do we want to use `help_ptr`?
    - Once this while loop finishes, `myList` is pointing to NULL!  We've effectively LOST OUR LIST!

```
// myList is already pointing to
// a valid list

while (myList != NULL) {
      printf("%d ", myList->data);
      myList = myList->next;
}
```

# Traversing Linked Lists

- Traverse and Print out data of a linked list
  - Remember:
    - When traversing linked lists, you ALWAYS want to use a helper pointer
    - NEVER use the head of the list for this purpose
    - This allows you to maintain the integrity of the list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
      printf("%d ", help_ptr->data);
      help_ptr = help_ptr->next;
}
```

# Traversing Linked Lists

- Traverse and Modifying data of a linked list
  - **Assume** that myList is already pointing to a valid linked list of nodes of type ll_node
    - This struct (ll_node) was defined earlier
  - Let’s say we want to add “10” to the data field of all nodes.  Here’s the code to do this:

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
      help_ptr->data += 10;
      help_ptr = help_ptr->next;
}
```

# Traversing Linked Lists

- Traverse and Modifying data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
       help_ptr->data += 10;
       help_ptr = help_ptr->next;
}
```

- Let’s take a closer look:
  - This works just like the last example
  - Instead of printing out the data field of each node
  - We are modifying each data field
    - Simply adding 10 to whatever is already stored in it

# Traversing Linked Lists

- Traverse and Modifying data of a linked list

```
struct ll_node *help_ptr;
help_ptr = myList;

while (help_ptr != NULL) {
      help_ptr->data += 10;
      help_ptr = help_ptr->next;
}
```

  - Let’s take a closer look:
    - We then traverse the list with the second instruction of the while loop
    - When we reach the end of the list, `help_ptr->next` will be NULL, and we will exit the loop

# Linked Lists - Operations

- There are several basic operations that need to be performed on linked lists:

1. Add a node.
2. Delete a node.
3. Search for a node.

- For each of these, you need to know how to traverse the list from the previous slides

- Next time, we will go over Adding nodes to a list

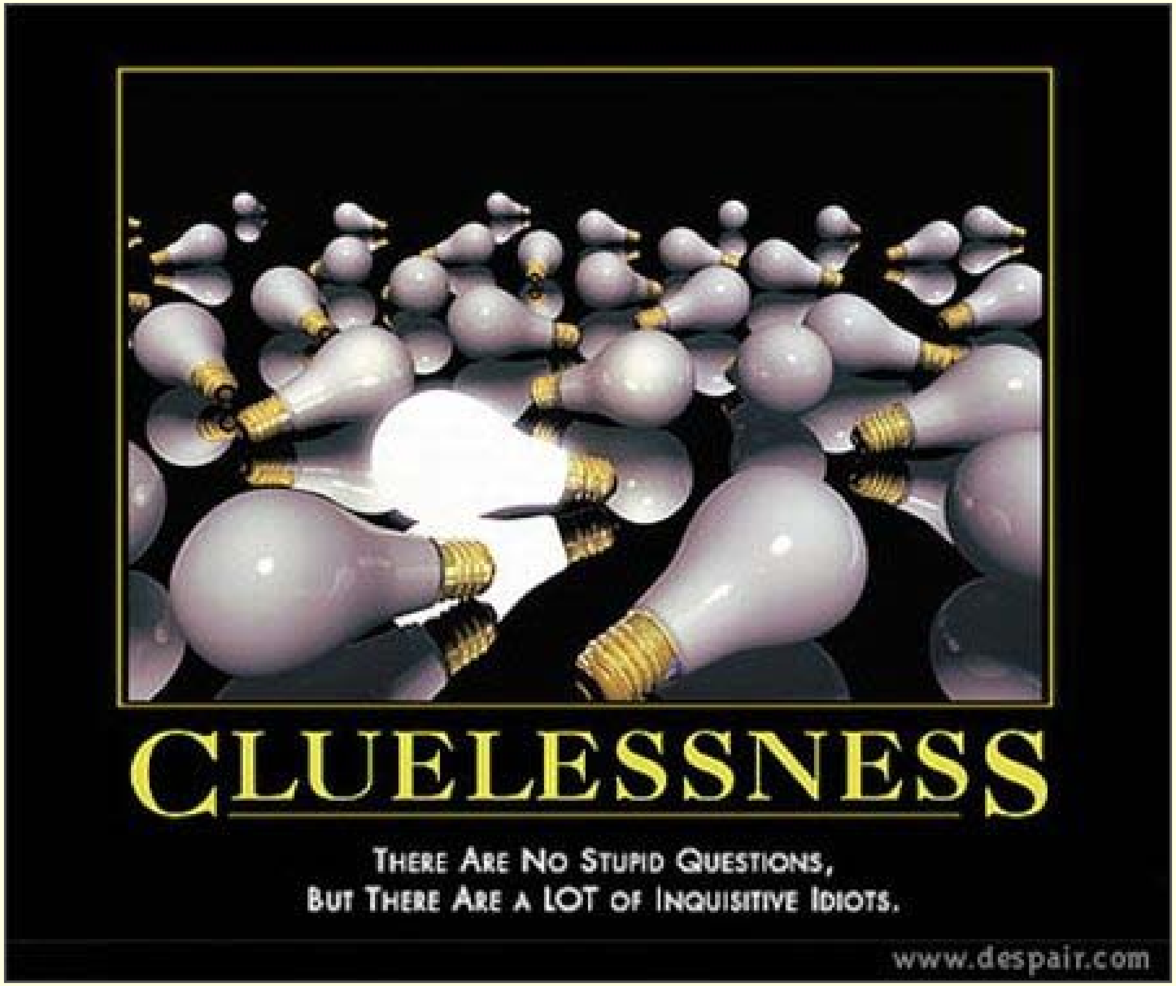
# Linked Lists

WASN'T

THAT

AMAZING!

# Daily Demotivator


CLUELESSNESS
There Are No Stupid Questions,
But There Are a LOT of Inquisitive Idiots.
www.despair.com

# Linked Lists

Computer Science Department
University of Central Florida

*COP 3502 – Computer Science I*