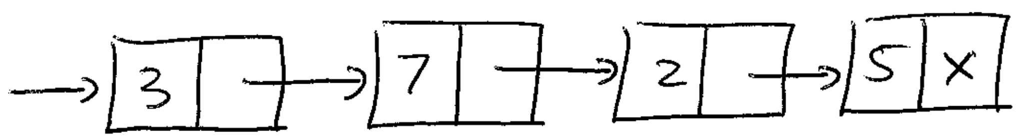
Linked Lists

Linked List: The simplest form of a linked structure. It consists of a chain of data locations called nodes. Each node holds a piece of information AND a link to the next node.

Each node is a struct that contains two fields: the data (which will just be a single integer for our examples) and a pointer to a node.

Here is a picture of what a simple linked list that stores 4 values looks like:



Here is a struct we would use to define a record that stores one of these nodes:

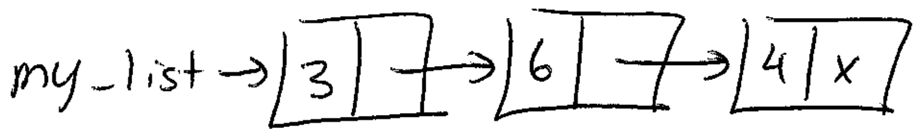
```
struct ll {
  int data;
  struct ll *next;
}
```

Now, to actually use this record we would have to define a variable of type List_Node as follows:

```
struct ll  *my_list;
```

How to access nodes of a linked list

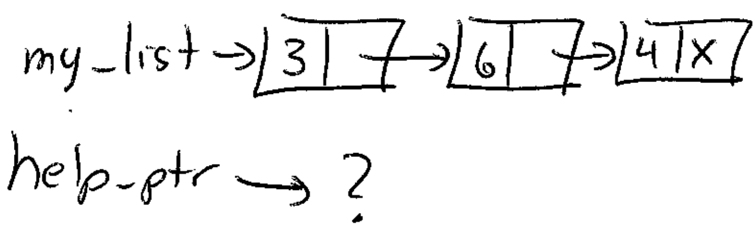
Let's assume we already have my_list **initialized to look like this: (Don't worry how this occurred, we'll go through that in a bit.)**



Now, one of the most common errors dealing with pointers is "moving" the head of the list. Consider if we made my_list **point to the second node we have listed. In this case, we would have NO way to access that data value in the first record. Rather than do this, what we need is a temporary List_Node pointer to help us move through the list. We can define one as follows:**

```
struct ll   *help_ptr;
```

Now our picture looks like this:



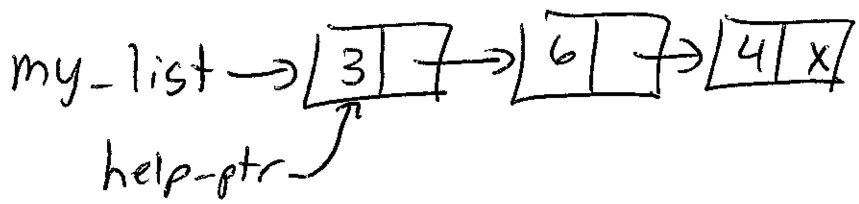
As we can see, help_ptr **is uninitialized. Just as it isn't a good practice to leave variables uninitialized, it isn't good to leave pointers the same way. The default value that a pointer is initialized to is ~~NIL~~ NULL and can be done like the following:**

```
help_ptr = NULL;
```

Now, if we want to use help_ptr to **move around the list pointed to by** my_list, **we could start off with the following line:**

```
help_ptr = my_list;
```

This makes our picture look like:



Now, to access say the data **field of the first record in the list, we could refer to it in either of these ways:**

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

A few things to notice here. First both of these expressions refer to the same exact variable since my_list **and** help_ptr **are pointing to the same exact ll.**

Next, in order to access that first ll through either of the pointers, we MUST dereference the pointer using the * symbol.

Finally, we use the dot operator to refer to a field within the record, as we learned before. Notice that the expression

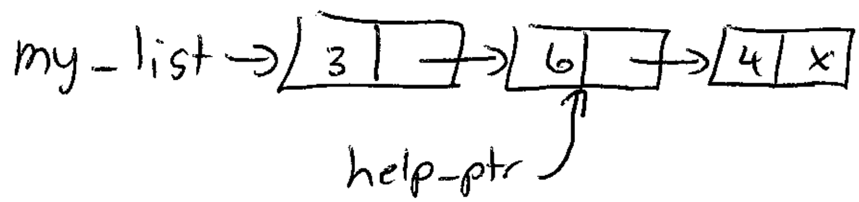
my_list.data

is syntactically incorrect because my_list **IS NOT a of type ll, and we are only allowed to access the data field of a ll. Also notice that the arrow -> provides an valid alternative syntax.**

Consider now using the pointer help_ptr **to traverse the list pointed to by** my_list. **We could do something like**

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

Note that the syntax here is correct because both sides of the assignment statement are pointers to ll's. Here is how this statement would change our picture:



Then, we could refer to the data **field in the second List_Node as:**

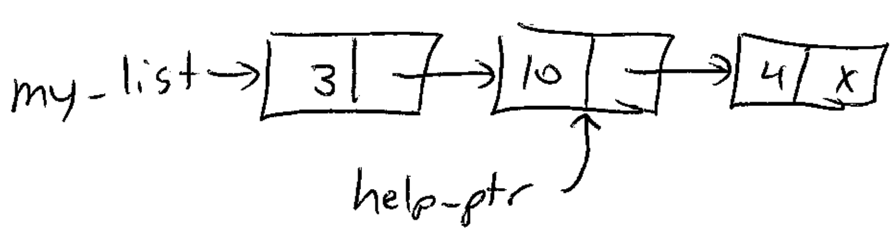
```
help_ptr->data
```

We can repeatedly use help_ptr in this fashion to iterate through this list. We could also modify the values in the list with a statement like:

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

This sort of manipulation will be handy for "editing" lists.

If we did this the new picture is



Applying this to a segment of code that prints out a linked list.

Assume that my_list is already pointing to a valid list of values.

```
struct ll *help_ptr;
help_ptr = my_list;

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

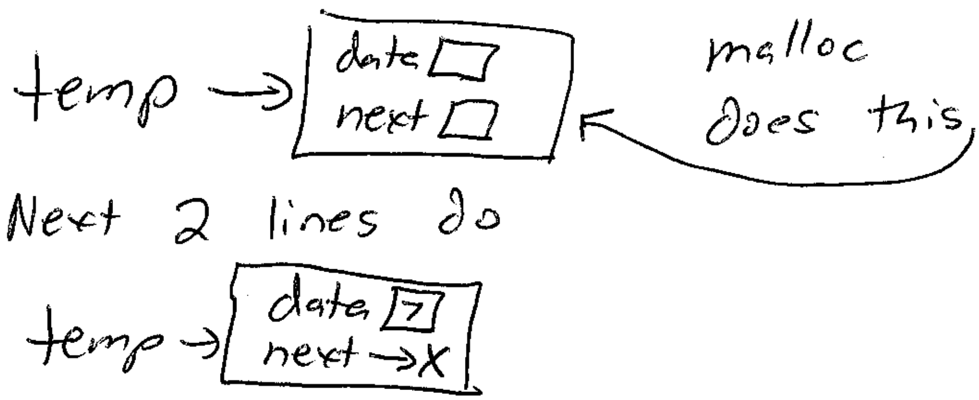
If you carefully look at the linked list code example given to you, you'll see that no temporary pointer is used. Why is this okay?

How to add a node to a list

This is how to create a List_Node to be added to a list:

```
struct ll *temp;
temp = malloc(sizeof(struct ll));
temp->data = 7
temp->next = NULL
```

The picture of this looks like this:



There are a couple of things going on here. Note that temp here is a pointer to a ll, not a ll itself, and that first statement implicitly creates a ll that the pointer temp **is pointing to. Once we do that, all we have to do is dereference our pointer to initialize the newly created ll. By having this capability of created ll's like this on the fly, and then adding them to a list, we have the ability to store information dynamically.**

Now, to finally add this node to an end of a list, assume that the List_Node pointer help_ptr **is pointing to the last node in the list. Then all we have to do to connect the entire list is:**

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

Now, let's go over the other functions in the handout.

<u>insert_front</u>

1) Create a node storing the element to insert.
2) Attach this node to the rest of the list.
3) Return a pointer to this newly created node.

<u>insert_back</u>

1) Create a node storing the element to insert.
2) Use a temporary pointer to iterate to the last node of the list.
3) Attach this last node to the newly created node.
4) Return a pointer to the front of the original list.

In both of these functions: we must check to see if the list we are inserting into is NULL and return accordingly.

<u>insert_inorder</u>

1) Create a node storing the element to insert.
2) Use a temporary pointer to iterate through the list, making sure to stop at the node RIGHT BEFORE(call this x), the insertion needs to be made.
3) Save a pointer to the node RIGHT AFTER(call this y) where the inserted node needs to be placed.
4) Attach x to the newly created node.
5) Attach the newly created node to y.

<u>delete</u>

1) Use a temporary pointer to iterate through the list, stopping at the node RIGHT BEFORE the one storing the value to delete.
2) Store pointers to the node to delete.
3) Patch the node RIGHT BEFORE the deleted node to the one RIGHT AFTER it.
4) Free the memory for the deleted node.