Linked Lists: Inserting Nodes

Kumkum Saxena

Review of Linked Lists

- What is a linked list?
  - Sequence of nodes chained together
    - Data part
    - Link part (points to next node in the chain)
  - Need a head pointer to point to the front of list
    - Called `myList` or `Head` or whatever you want
      - It's goal in life is just to point to the head of the list
  - Need a helper pointer to point to traverse list
    - `help_ptr`
      - We then save the value stored in `myList` into `help_ptr`, thus allowing `help_ptr` to also point to the front of the list
      - And we can nod use `help_ptr` to traverse the list

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;
}
```

  - Review previous slides for more info on this

Inserting Nodes

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: Basic Operations

- Design Approach
  - Before going further, we must understand the design approach of our **functions**
    - Functions that change the contents of lists (insertion and deletion) will return the list head pointer
      - Why?
      - Let's say we have a list with 4 nodes and we are adding a new node
        - For the sake of this example, let's say that based on its key value, the new node must be inserted at the beginning of the list
      - As a result, the address of the front of the list (address of the first node) has now changed! (within the scope of the func.)
      - So we **must return the newly updated head of the list**
      - Now, **`myList`**, in main will know to point to the new 1st node

Linked Lists: Basic Operations

- Design Approach
  - Before going further, we must understand the design approach of our **functions**
    - Functions that change the contents of lists (insertion and deletion) will return the list head pointer
    - Here's an example of when this happens (insertion):
      - `myList = insertNewNode(...);`
    - So if the head of the list changes within the `insertNewNode` function
      - The function **MUST** return the updated head pointer
      - `myList` will be updated accordingly
    - If the head pointer doesn't change within the function:
      - `myList` is simply reset to its original address

Linked Lists: Basic Operations

- Design Approach
  - Before going further, we must understand the design approach of our **functions**
    - Functions that do **not** change the contents of the list return values that are consistent with their purpose
      - Example: a function to locate a node will return an integer (1 perhaps) to indicate whether or not the node was found
      - Example: a function to determine the number of nodes in the list will most likely return an integer count
    - Finally, functions that process the entire list, such as printing the list, will usually simply return void

Linked Lists: Basic Operations

- Linked List Order
  - Linked Lists are **linear** structures
    - They should always have some type of order
      - This could be chronological order: order of arrival and insertion into the list
      - **Key-based order**: lexical ordering based on some key value of the data items (alphabetical by last name, or ordered by NID, etc)
    - Key based lists are the most common
      - New nodes are added to the linked list based on the lexical ordering of key values
    - We will focus these slides on how to insert into a sorted list (key-based list)

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - There are four main steps involved here:
  1) Allocate memory for the new node
  2) Determine the insertion point
     - You need to locate the new node's predecessor
       - Basically, we need to find the node that comes before where you want to insert the new node
  3) Point the new node to its successor
     - To the node that will come after it, once inserted
  4) Point the predecessor node to the new node

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - Step 2 is to **determine the insertion point**
    - For this, we need the location of the new node's predecessor
    - The are four possibilities:
      1) The list is empty. Therefore, there is no predecessor, and new node will become the first node.
      2) The new node is to be inserted at the beginning of the list, so again, there is no predecessor node.
      3) The new node is the last node of the list. So its predecessor was the previous last node
      4) The new node is inserted at some arbitrary point, which is neither the first node or the last node. Meaning, the new node will go somewhere in the middle of the list.

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - We mentioned (few slides back) that we will use sorted linked lists
    - Meaning, when we insert, we must insert a new node into the correct position
  - But for now, **for the sake of ease**:
    - We will simply assume that nodes are added to the **front** of the list
    - We will use a function to add nodes
    - The function must then return the new HEAD of the list
      - It will be a new head right? Of course! Cuz we just added a node at the front. This changes what the head pointer will point to!

Linked Lists: Insertion

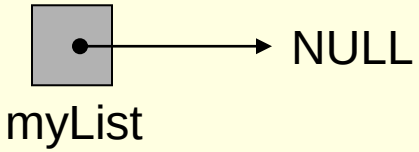
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:
    - There are two scenarios:

    1) The list could be empty
       - Meaning, the head pointer, myList, simply points to NULL
    2) Or there are existing nodes already in the list

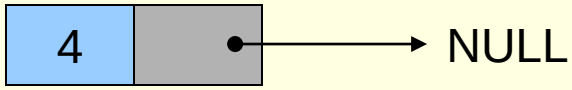
    - Let's look at both of these scenarios...

Linked Lists: Insertion

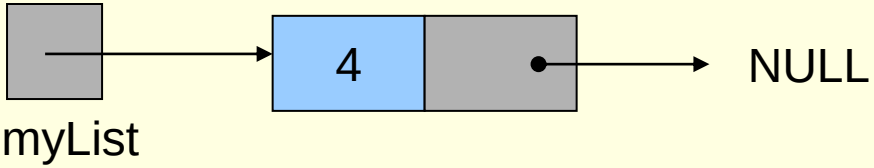
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:
    - There are two scenarios:
    1) Insertion into an Empty List



An empty linked list



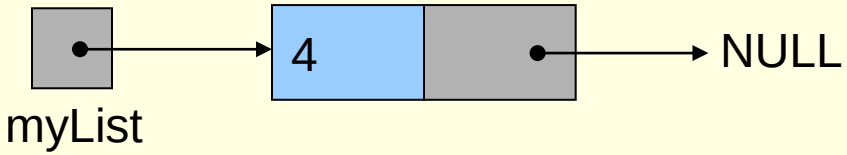
The new node to be inserted



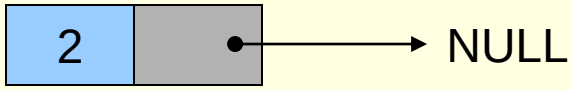
A list after the insertion of the new node

Linked Lists: Insertion

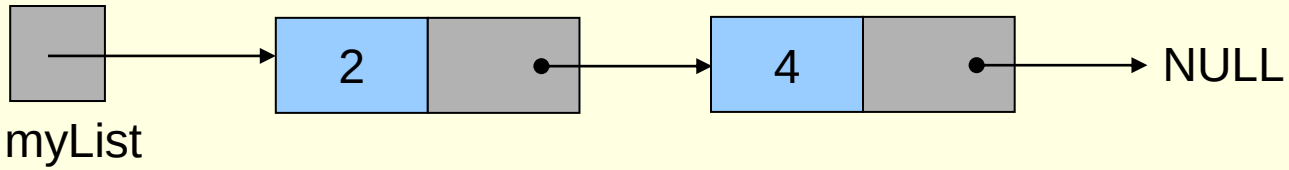
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:
    - There are two scenarios:
    2) Insertion at the Head of an existing list



The initial liked list



The new node to be inserted



A list after the insertion of the new node

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:
    - There are two scenarios:
    1) Insertion into an Empty List
    2) Insertion at the Head of an existing list

  - Let's now look at the code for this in detail…

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - First, here's a sample node:

  - And here is its respective `struct` skeleton:

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

Adding Nodes to Front (code)

```
// Function Prototype:
struct ll_node* addToFront(struct ll_node *list, int value) ;

int main( ) {
        int number = 0;
        // We now make our head pointer, myList and initialize to NULL
        struct ll_node *myList = NULL;
        // User enters data for new node (or -1 to exit)
        while(number!= -1) {
                // Get the next number.
                printf("Enter data for next node:  ");
                scanf("%d", &number);

                // Add to linked list if it's not -1.
                if (number !=-1)
                        myList = addToFront(myList, number);
        }
        return 1;
}
```

Adding Nodes to Front (code)

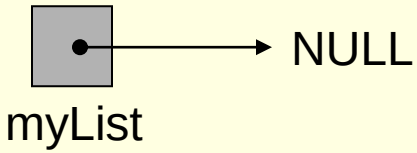
- So we will make this function: `addToFront`
- What are we sending to the function?
- Two parameters:
  1) `myList`, which is the pointer to the head of the list
     - This allows us to access the linked list from the function
  2) And we send over `number`, which will be the data value of the new node
- What does the function return?
- Remember, this function makes a NEW node at the **front** of the list
- We are making a new front of the list
- This means that `myList`, within main, will need to be updated to point to this new first node within of the list
- How do we do that?
- The updated address of the head of the list, within the function, is returned to main and saved into `myList`

Adding Nodes to Front (code)

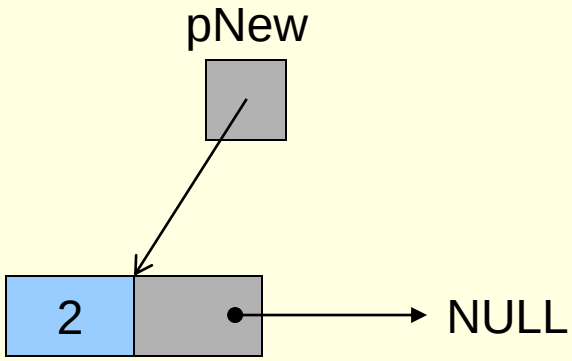
```
struct ll_node* addToFront(struct ll_node *list, int value) {
        // Create the new node and put data (from argument) into it
        struct ll_node * pNew = (struct ll_node *)
                                        malloc(sizeof(struct ll_node));
        pNew->data =  value;
        pNew->next = NULL;
```

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



An empty linked list



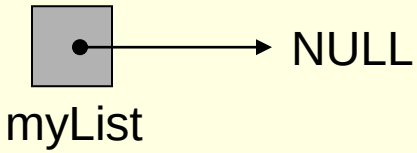
The new node to be inserted

Adding Nodes to Front (code)

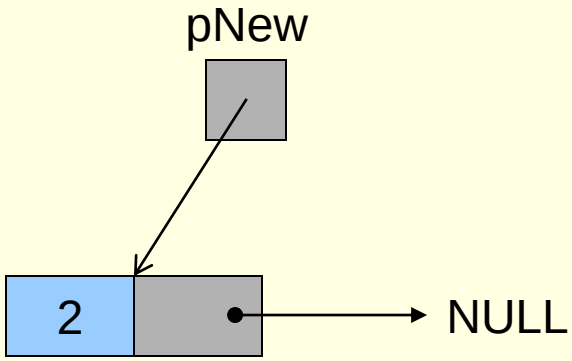
```
struct ll_node* addToFront(struct ll_node *list, int value) {
        // Create the new node and put data (from argument) into it
        struct ll_node * pNew = (struct ll_node *)
                                        malloc(sizeof(struct ll_node));
        pNew->data = value;
        pNew->next = NULL;
        // If the original list is empty, set the original head
        // pointer to point to the new node.
        if(list == NULL)
                list = pNew;
```

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



An empty linked list

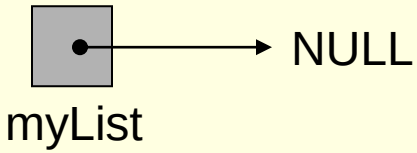


The new node to be inserted

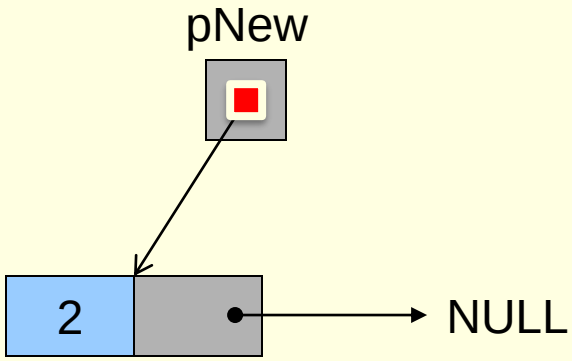
  - If the list is empty (`myList` points to NULL)
    - Take the address of `pNew` and put it into `myList`
    - This makes `myList` now point to the new Node

Linked Lists: Insertion

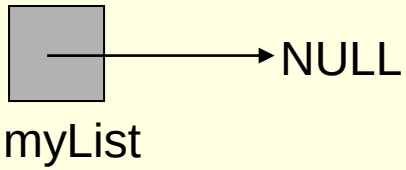
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



An empty linked list



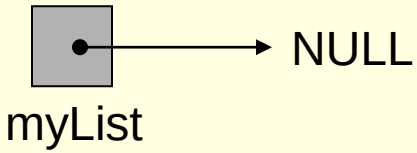
The new node to be inserted



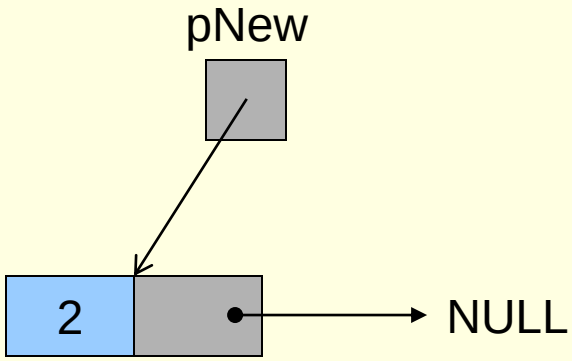
A list after the insertion of the new node

Linked Lists: Insertion

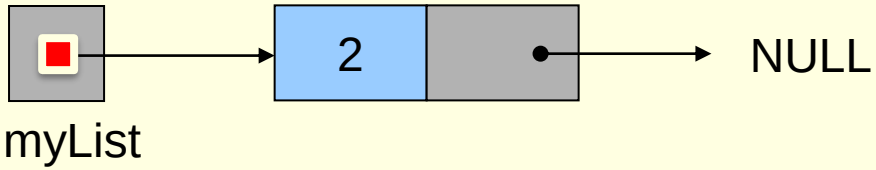
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



An empty linked list



The new node to be inserted



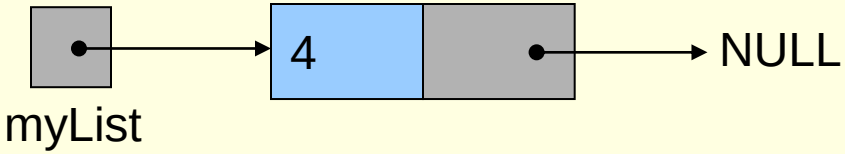
A list after the insertion of the new node

Adding Nodes to Front (code)

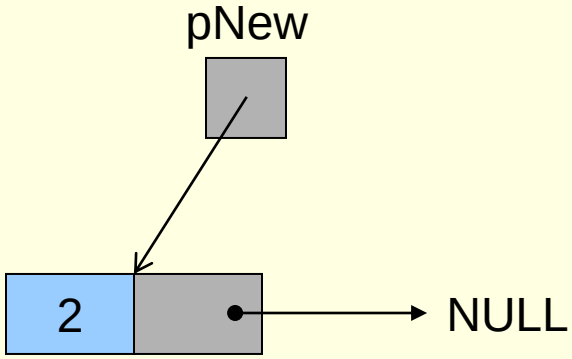
```
struct ll_node* addToFront(struct ll_node *list, int value) {
        // Create the new node and put data (from argument) into it
        struct ll_node * pNew = (struct ll_node *)
                                        malloc(sizeof(struct ll_node));
        pNew->data =  value;
        pNew->next = NULL;
        // If the original list is empty, set the original head
        // pointer to point to the new node.
        if(list == NULL)
                list = pNew;
        // Else, list is currently pointing to a non-empty list.
        else {
                // Point new node to wherever Head pointer pointed to.
                pNew->next  =  list;
```

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



The initial liked list

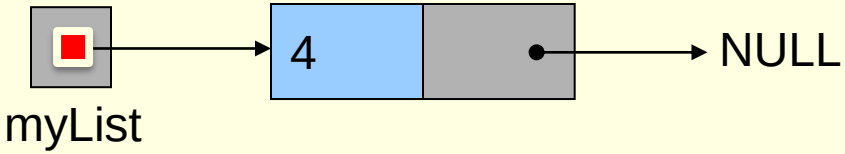


The new node to be inserted

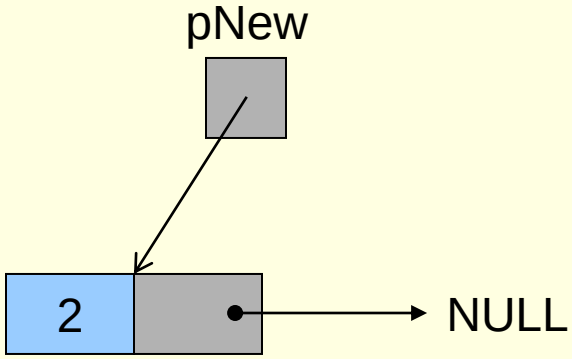
  - ELSE, if the list is non-empty
    - Take the address that `myList` points to and put in the `next` of `pNew`
    - This makes `pNew` point to the (previously) first node

Linked Lists: Insertion

- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



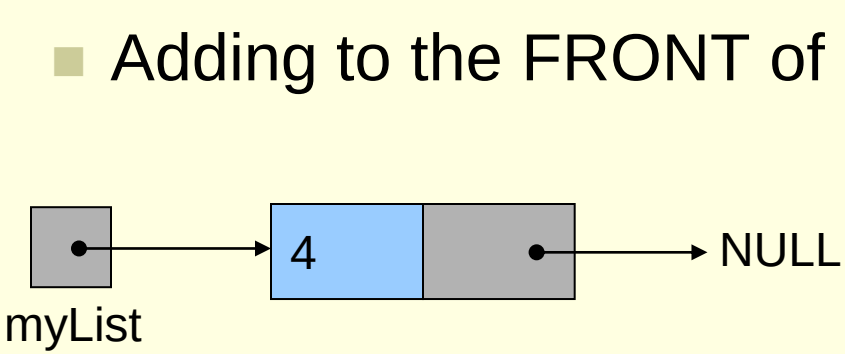
The initial liked list



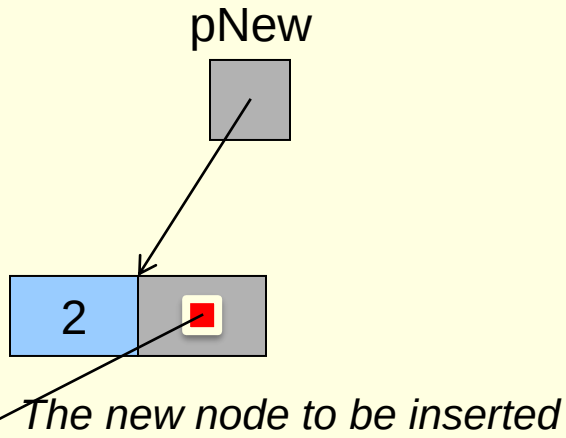
The new node to be inserted

Linked Lists: Insertion

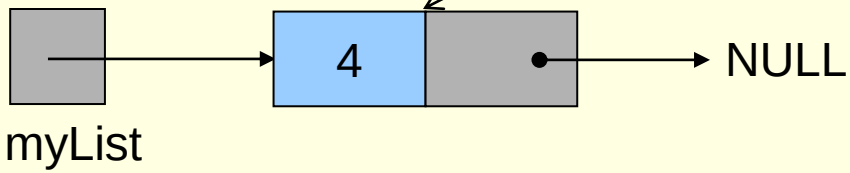
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



The initial liked list



The new node to be inserted



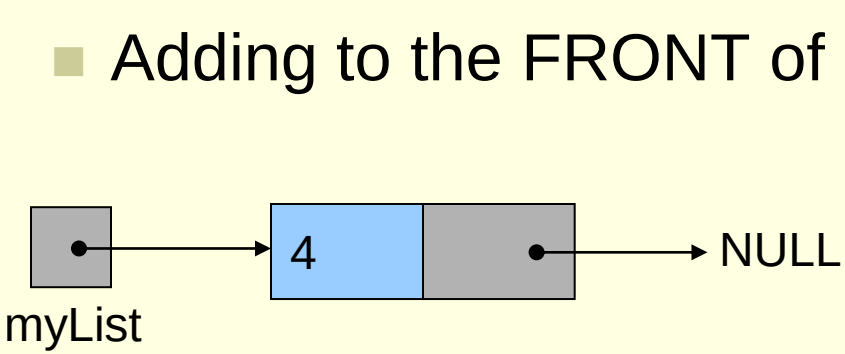
A list after the insertion of the new node

Adding Nodes to Front (code)

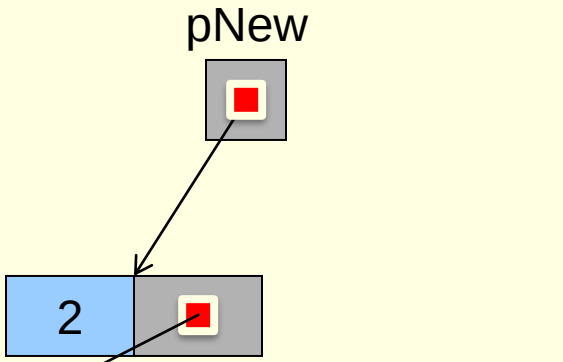
```
struct ll_node* addToFront(struct ll_node *list, int value) {
        // Create the new node and put data (from argument) into it
        struct ll_node * pNew = (struct ll_node *)
                                        malloc(sizeof(struct ll_node));
        pNew->data =  value;
        pNew->next = NULL;
        // If the original list is empty, set the original head
        // pointer to point to the new node.
        if(list == NULL)
                list = pNew;
        // Else, list is currently pointing to a non-empty list.
        else {
                // Point new node to wherever Head pointer pointed to.
                pNew->next  =  list;

                // Now make Head pointer point to the new node.
                list  =  pNew;
        }
        return list;
}
```

Linked Lists: Insertion

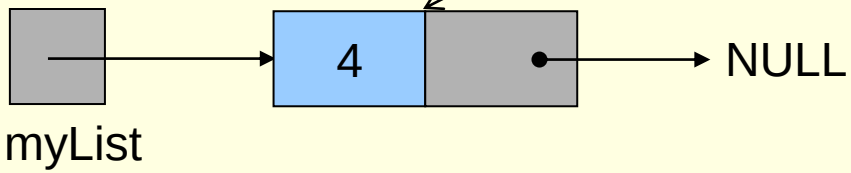
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



The initial liked list



The new node to be inserted

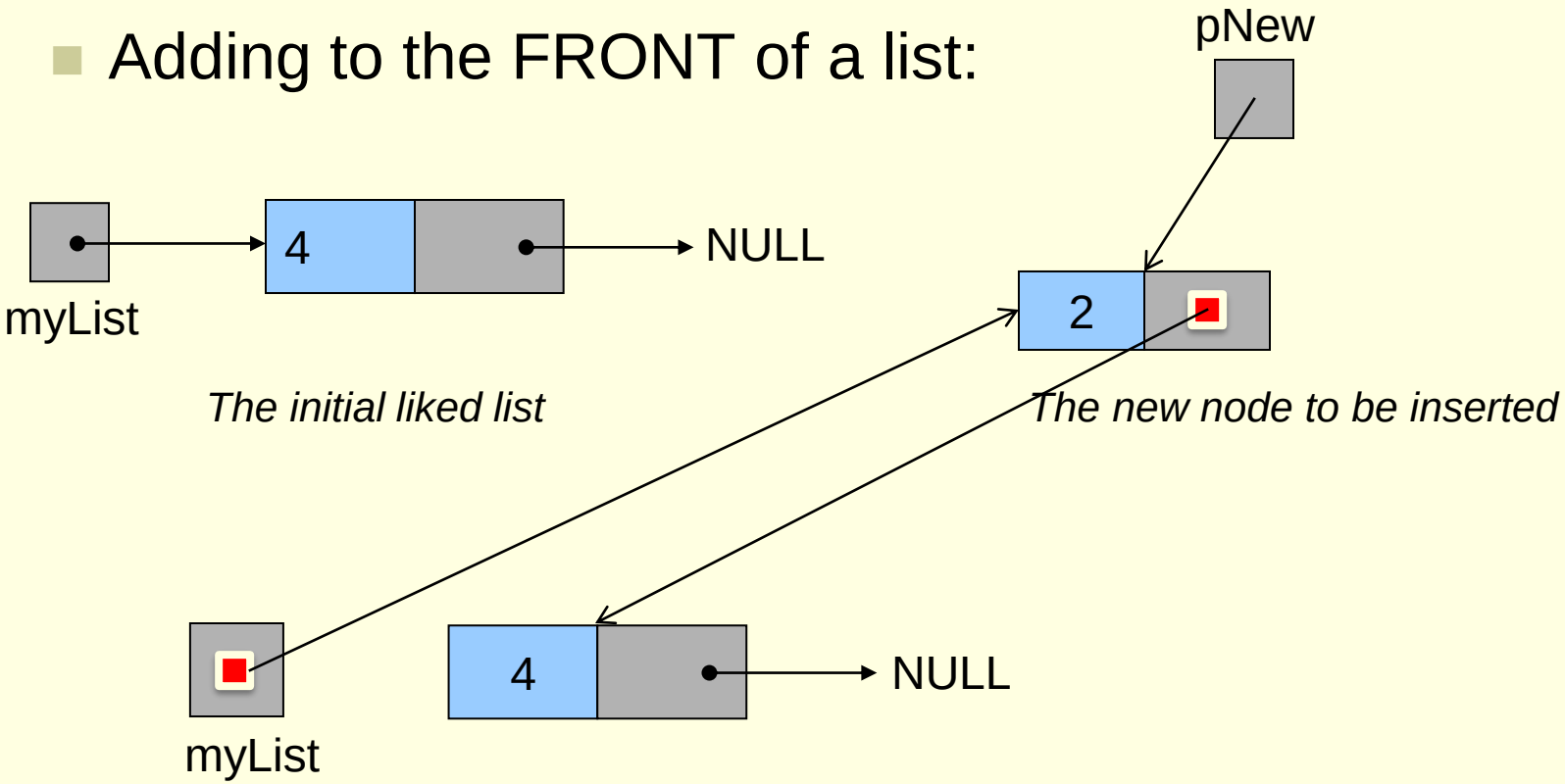


Basically, take the address stored in `pNew` and put it in `myList`

A list after the insertion of the new node

Linked Lists: Insertion

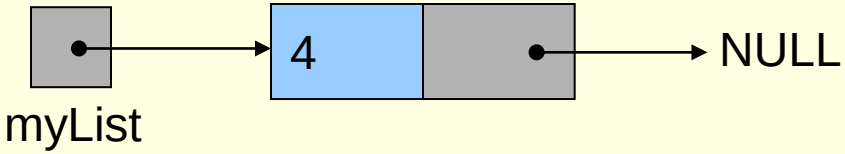
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



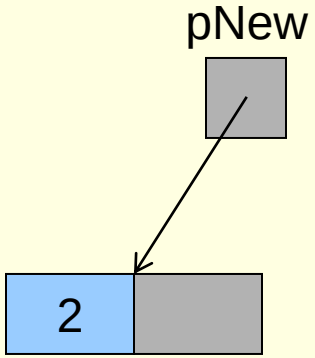
A list after the insertion of the new node

Linked Lists: Insertion

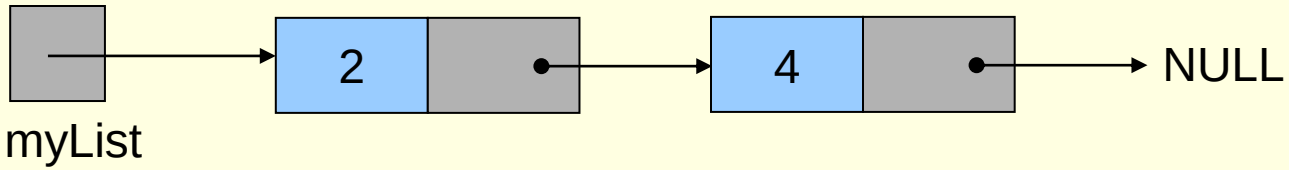
- Adding Nodes to a Linked List
  - Adding to the FRONT of a list:



The initial liked list



The new node to be inserted



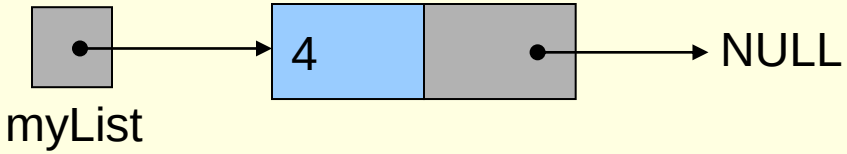
A list after the insertion of the new node

Linked Lists: Insertion

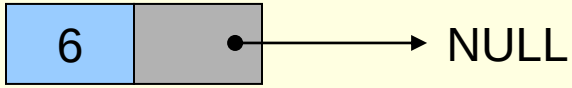
- Adding Nodes to a Linked List
  - Now let's assume that we are adding always to the end of the list
  - The code in main won't change a whole lot
  - But the function to add to the end is a bit different.
  - Can anyone tell us why?
  - Because we need to **traverse** the list in order to arrive at the insertion point (the end of the list)
  - Here's the picture followed by the code:

Linked Lists: Insertion

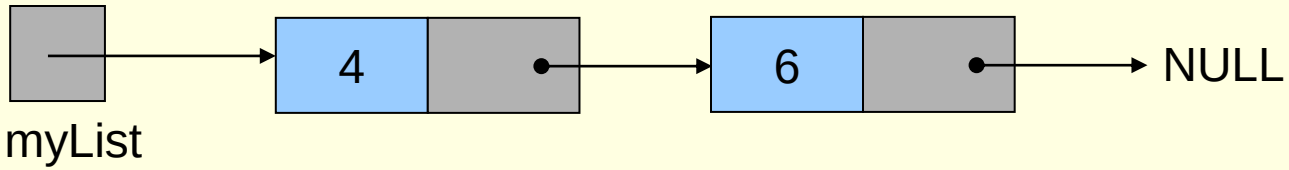
- Adding Nodes to a Linked List
  - Adding to the END of a list



The initial liked list



The new node to be inserted



A list after the insertion of the new node

Adding Nodes to End (code)

```
// Function Prototype:
struct ll_node* addToEnd(struct ll_node *list, int value) ;

int main( ) {
        int number = 0;
         // We now make our head pointer, myList and initialize to NULL
        struct ll_node *myList = NULL;
        // User enters data for new node (or -1 to exit)
        while(number!= -1) {
                // Get the next number.
                printf("Enter data for next node:  ");
                scanf("%d", &number);

                // Add to linked list if it's not -1.
                if (number !=-1)
                        myList = addToEnd(myList, number);
        }
        return 1;
}
```

Adding Nodes to End (code)

```
struct ll_node* addToEnd(struct ll_node *list, int value) {
        // Make helper pointer and store head of list into it
        struct ll_node *help_ptr = list;

        // Create the new node and put data (from argument) into it
        struct ll_node * pNew = (struct ll_node *)
                                        malloc(sizeof(struct ll_node));
        pNew->data = value;
        pNew->next = NULL;
        // If list is empty, pNew becomes the first node
        if (list == NULL)
                return pNew;
```

If the list is empty:

- pNew will be the first (and only) node of the list
- So we have our head pointer, `myList` or `list`
  - And this head pointer MUST point to the new node, `pNew`
  - So we can simply return the address of `pNew` to main
  - Remember, we called this function with:
  - **`myList = addToEnd(myList, number);`**
  - So whatever we return will be saved in `myList` (head pointer)

Adding Nodes to End (code)

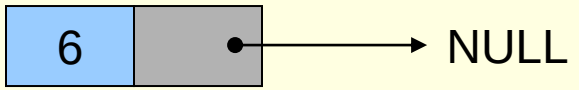
```
struct ll_node* addToEnd(struct ll_node *list, int value) {
        // Make helper pointer and store head of list into it
        struct ll_node *help_ptr = list;

        // Create the new node and put data (from argument) into it
        struct ll_node * pNew = (struct ll_node *)
                                        malloc(sizeof(struct ll_node));
        pNew->data = value;
        pNew->next = NULL;
        // If list is empty, pNew becomes the first node
        if (list == NULL)
                return pNew;
        // Else, traverse the list to arrive a the last node
        while (help_ptr->next != NULL)
                help_ptr = help_ptr->next;
        // Make the last node point to the to-be-inserted node, i.e.
        // put the address of new node into the last node's "next"
        help_ptr->next = pNew;

        // Return a pointer to the modified list
        return list;
}
```

Linked Lists: Insertion

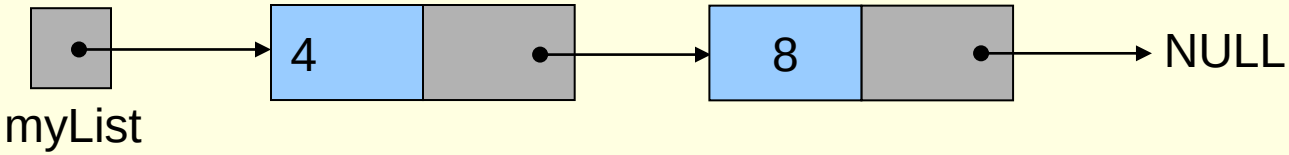
- Adding Nodes to a Linked List
  - Adding to the MIDDLE of a list:
    - Think about what must happen.
    - We are inserting a new node between two other nodes
    - So the new node must now point to its successor node
      - Meaning, it must point to where its predecessor node was pointing to (before insertion of new node)
    - Then the address of the new node must be saved into the “next” of the predecessor node.
      - These two steps maintain the integrity of the list
    - Again, here’s some examples.

Linked Lists: Insertion

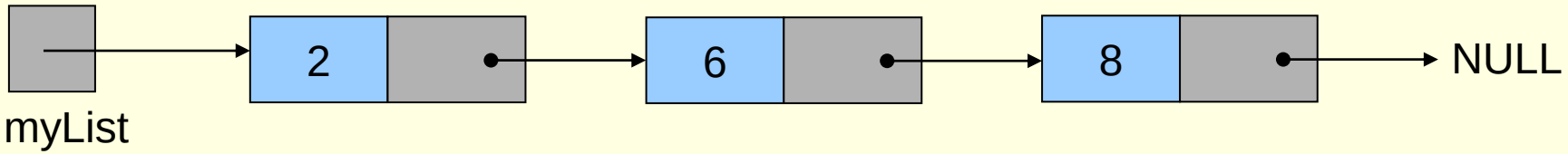
- Adding Nodes to a Linked List
  - Adding to the MIDDLE of a list:



The new node to be inserted

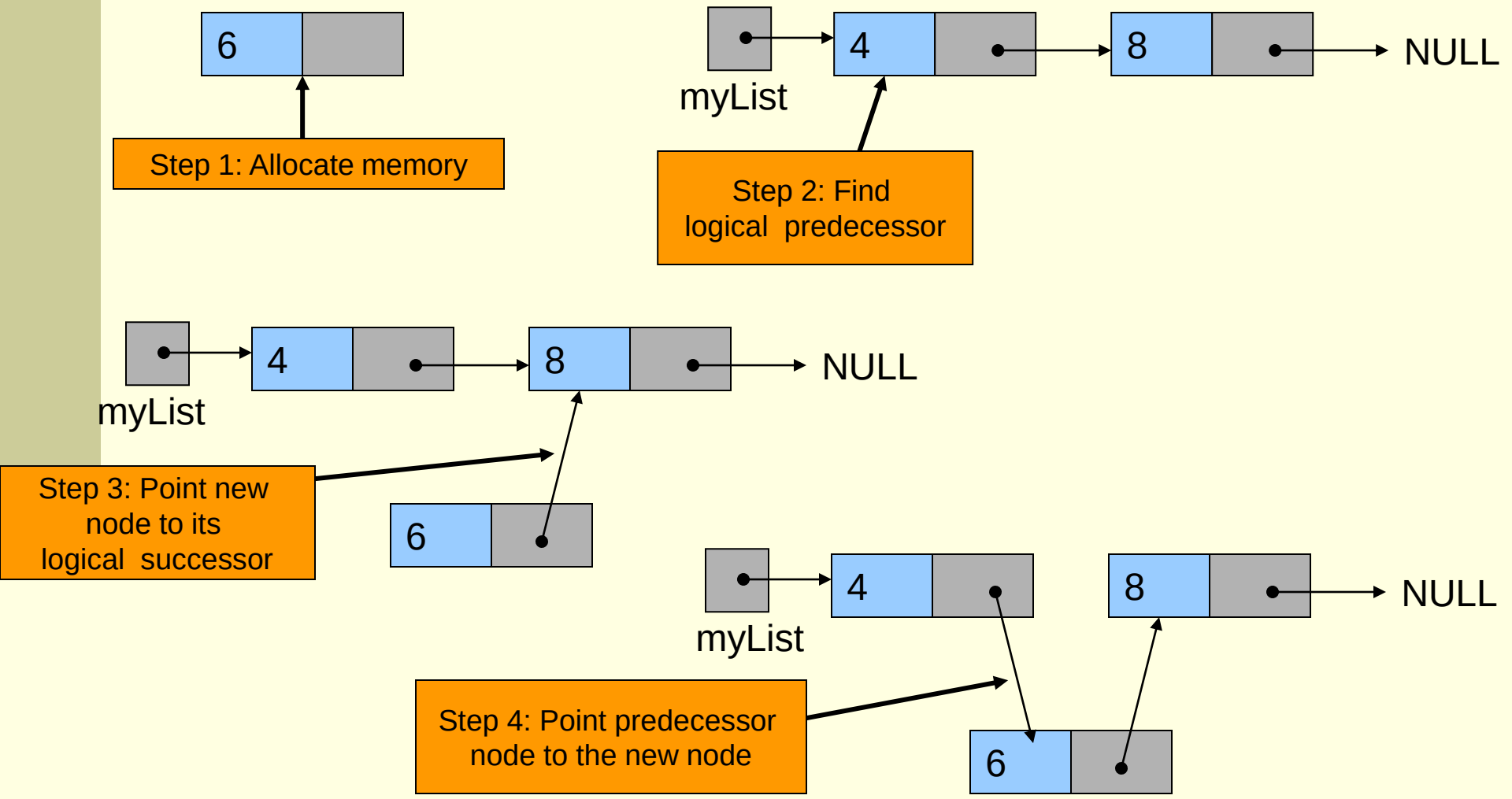


The initial liked list



A list after the insertion of the new node

Linked Lists: Insertion in Detail



Linked Lists: Insertion

- Adding Nodes to a Linked List
  - Adding to the MIDDLE of a list:
    - We MUST **first** point the new node to the successor node
    - THEN, and only then, can we point the predecessor node to the new node
    - Why MUST this happen in this specific order?
    - What if we first point the predecessor node to the new node?
      - What's wrong with that?
    - If we first point the predecessor node to the new node, we will have lost our connection to the successor node
      - Only the predecessor knew where the successor is!

Adding Nodes to Sorted List

- Adding Nodes to a Linked List
  - Adding nodes to a sorted list
    - Now we are hopefully ready to add a new node to any location of a sorted linked list
      - This new node may end up being inserted at the beginning, the middle, or at the end.
    - The following code takes care of **all** possibilities
      - This is the "real" linked list code
      - The previous two sets of code were just teaching examples
    - Again, we are just using an `int` as the `data` item
    - The nodes of the linked list are sorted in ascending order based on the value stored in each node's `data`

Adding Nodes to Sorted List (code)

```
// Function Prototype:
struct ll_node* insert(struct ll_node *list, int value) ;

int main( ) {
        int number = 0;
         // We now make our head pointer, myList and initialize to NULL
        struct ll_node *myList = NULL;
        // User enters data for new node (or -1 to exit)
        while(number!= -1) {
                // Get the next number.
                printf("Enter data for next node:  ");
                scanf("%d", &number);

                // Add to linked list if it's not -1.
                if (number !=-1)
                        myList = insert(myList, number);
        }
        return 1;
}
```

Adding Nodes to Sorted List (code)

```
struct ll_node* insert(struct ll_node *list, int value) {
        // Make helper pointer and store head of list into it
        struct ll_node *help_ptr = list;

        // Create the new node and put data (from argument) into it
        struct ll_node * pNew = (struct ll_node *)
                                        malloc(sizeof(struct ll_node));
        pNew->data = value;
        pNew->next = NULL;
        // First, we take care of Insertion into an empty list
        // OR Insertion at the front of a non-empty list
        if (list == NULL || list->data > value) {
                pNew->next = list;
                list = pNew;
                return list;
        }
```

Let's now look at this IF statement in detail.

Adding Nodes to Sorted List (code)

- Adding nodes to **front** of sorted list
  - Think about it:
    - When do we go inside the IF statement?
      1) If `list` is `NULL`
         - What does this mean?
         - It means the list is currently empty
      2) OR if the `data` at the node that `list` points to is greater than `value`
         - Meaning, the value we want to insert is smaller than the current first node. So, the new node must be placed at the front.

```
if (list == NULL || list->data > value) {
        pNew->next = list;
        list = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to **front** of sorted list
  - Now look at the three lines inside the statement
    - In both scenarios (empty list or insert at front)
      - We MUST take the address that is currently saved in `list` and save it in `pNew->next`
      - What does this do?
        - It makes `pNew` point to whatever `list` was pointing to
        - So if `list` was pointing to node A (the previous first node)
        - Now `pNew` will also point to node A
        - Which makes sense, since `pNew` will be the new first node

```
if (list == NULL || list->data > value) {
        pNew->next = list;
        list = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to **front** of sorted list
  - Now look at the three lines inside the statement
    - In both scenarios (empty list or insert at front)
      - Also, we MUST take the address of `pNew` and save it into `list`
      - This makes `list` point to `pNew` (the new first node)
      - Which makes sense right? `list` is the head pointer!
        - Remember, the only goal in life of the head pointer is to point to the first node!
      - Finally, we return list (head pointer address) to main

```
if (list == NULL || list->data > value) {
        pNew->next = list;
        list = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

```
struct ll_node* addToEnd(struct ll_node *list, int value) {

        // ...
        // CODE BELONGING HERE WAS ON PREVIOUS PAGE
        // ...

        // Continuing the code ...

        // Insert at MIDDLE or END of list
        // We MUST now find the right place to insert
        while(help_ptr->next != NULL && help_ptr->next->data < value)
                help_ptr = help_ptr->next;
```

- While:
  - there are still nodes in the list
  - AND the data value at the node **AFTER** the one that `help_ptr` points to is less than the value of the new node to be inserted
    - Meaning, we haven't reached the insertion spot yet
    - KEEP traversing the list to find insertion spot

Adding Nodes to Sorted List (code)

```
struct ll_node* addToEnd(struct ll_node *list, int value) {

        // ...
        // CODE BELONGING HERE WAS ON PREVIOUS PAGE
        // ...

        // Continuing the code ...

        // Insert at MIDDLE or END of list
        // Find the right place to insert
        while(help_ptr->next != NULL && help_ptr->next->data < value)
                help_ptr = help_ptr->next;
```

- Notice the `&&` instead of `||`
- So **when do we exit the while loop**?
  - If the data in the node AFTER the one that `help_ptr` points to is **greater than or equal to** `value`
    - Meaning, we've found our insertion spot (after `help_ptr`)
  - OR we exit once `help_ptr->next` is `NULL` (reached end of list)

Adding Nodes to Sorted List (code)

```
struct ll_node* addToEnd(struct ll_node *list, int value) {

        // ...
        // CODE BELONGING HERE WAS ON PREVIOUS PAGE
        // ...

        // Continuing the code ...

        // Insert at MIDDLE or END of list
        // Find the right place to insert
        while(help_ptr->next != NULL && help_ptr->next->data < value)
                help_ptr = help_ptr->next;

        // So help_ptr is now pointing to the node right before
        // the spot where we want to insert.
        // Now insert pNew right after the position that help_ptr points to
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```

Let's now look at these last three lines in detail.

Adding Nodes to Sorted List (code)

- Adding nodes to middle or end of sorted list
  - Remember:
    - We just traversed the list with `help_ptr` to find our insertion spot
    - So right now, `help_ptr` is pointing to the node immediately BEFORE our insertion spot.
      - It is pointing to the **predecessor**.
    - Example: if we need to insert at position 12, then `help_ptr` is currently pointing to position 11

```
        // ... previous code was here
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to middle or end of sorted list
  - Now, think about what happens:
    - If we are inserting `pNew` at the **END** of the list
      - `pNew->next` will need to point to NULL
        - Indicating the end of the list
      - So we execute this line of code
        - `pNew->next = help_ptr->next;`
      - Since `help_ptr` was pointing to the last node
        - `help_ptr->next` will have `NULL` in it
        - We save that value into `pNew->next`

```
        // ... previous code was here
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to middle or end of sorted list
  - Now, think about what happens:
    - If we are inserting `pNew` at the **END** of the list
      - Also, the previous last node in the list
        - Which is currently pointed to by `help_ptr`
      - Must now point to the new last node (`pNew`)
      - So we execute this line of code:
        - `help_ptr->next = pNew;`
      - Saves the address of `pNew` into `help_ptr->next`

```
        // ... previous code was here
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to middle or end of sorted list
  - Now, think about what happens:
    - If we are inserting `pNew` at the **END** of the list
      - The connections are now made
      - And we can simply return the head of the list
        - `return list;`

```
        // ... previous code was here
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to middle or end of sorted list
  - Now, think about what happens:
    - If we are inserting `pNew` in the **MIDDLE** of the list
      - `pNew->next` will need to point to the next node
        - The node that will come after it (once `pNew` is inserted)
      - So we execute this line of code
        - `pNew->next = help_ptr->next;`
      - Since `help_ptr` was pointing to the **predecessor**
        - `help_ptr->next` will have the address of the **successor**
        - We save that value into `pNew->next`

```
        // ... previous code was here
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to middle or end of sorted list
  - Now, think about what happens:
    - If we are inserting `pNew` in the **MIDDLE** of the list
      - Also, the predecessor
        - Which is currently pointed to by `help_ptr`
      - Must now point the newly inserted node (`pNew`)
      - So we execute this line of code:
        - `help_ptr->next = pNew;`
      - This saves the address of `pNew` into `help_ptr->next`

```
        // ... previous code was here
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```

Adding Nodes to Sorted List (code)

- Adding nodes to middle or end of sorted list
  - Now, think about what happens:
    - If we are inserting `pNew` in the **MIDDLE** of the list
      - The connections are now made
      - And we can simply return the head of the list
        - `return list;`

```
        // ... previous code was here
        pNew->next = help_ptr->next;
        help_ptr->next = pNew;
        return list;
}
```