Pointers and Linked Lists

Modified from Section 13.1

Linked Lists

- A linked list is a list of nodes in which each node has a member variable that is a pointer that points to the next node in the list
  - The first node is called the head
  - The pointer variable head, points to the first node
    - The pointer named head is not the head of the list. It points to the head of the list
  - The last node contains a pointer set to NULL
  - If the list is empty, head will be set to NULL

Building a Linked List: The node definition

- Let's begin with a simple node definition:

```
struct Node
{
   int data;
   Node *link;
};

typedef Node* NodePtr;
```

Building a Linked List: Declaring Pointer Variable head

- With the node defined and a type definition to make or code easier to understand, we can declare the pointer variable head:

```
NodePtr head;
```

  - head is a pointer variable that will point to the head node when the node is created

Building a Linked List: Creating the First Node

- To create the first node, the operator new is used to create a new dynamic variable:

```
head = new Node;
```

  - Now head points to the first, and only, node in the list

Building a Linked List: Initializing the Node

- Now that head points to a node, we need to give values to the member variables of the node:

```
head->data = 3;
head->link = NULL;
```

  - Since this node is the last node, the link is set to NULL

Function head_insert

- It would be better to create a function to insert nodes at the head of a list, such as:
  - void head_insert(NodePtr& head, int the_number);
    - The first parameter is a NodePtr parameter that points to the first node in the linked list
    - The first parameter is passed by reference because head's value can be changed if the input is an empty list
    - The second parameter is the number to store in the list
  - head_insert will create a new node for the number
    - The number will be copied to the new node
    - The new node will be inserted in the list as the new head node

Pseudocode for head_insert

- Create a new dynamic variable pointed to by temp_ptr
- Place the data in the new node called *temp_ptr
- Make temp_ptr's link variable point to the head node
- Make the head pointer point to temp_ptr

Translating head_insert to C++

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;  //create the temporary pointer
    temp_ptr = new Node; // create the new node

    temp_ptr->data = the_number; //copy the number

    temp_ptr->link = head; //new node points to first node
    head = temp_ptr;  // head points to new first node
}
```

An Empty List

- A list with nothing in it is called an empty list
- An empty linked list has no head node
- The head pointer of an empty list is NULL

head = NULL;

  - Any functions written to manipulate a linked list should check to see if it works on the empty list

Losing Nodes

- You might be tempted to write head_insert using the head pointer to construct the new node:

```
head = new Node;
head->data = the_number;
```

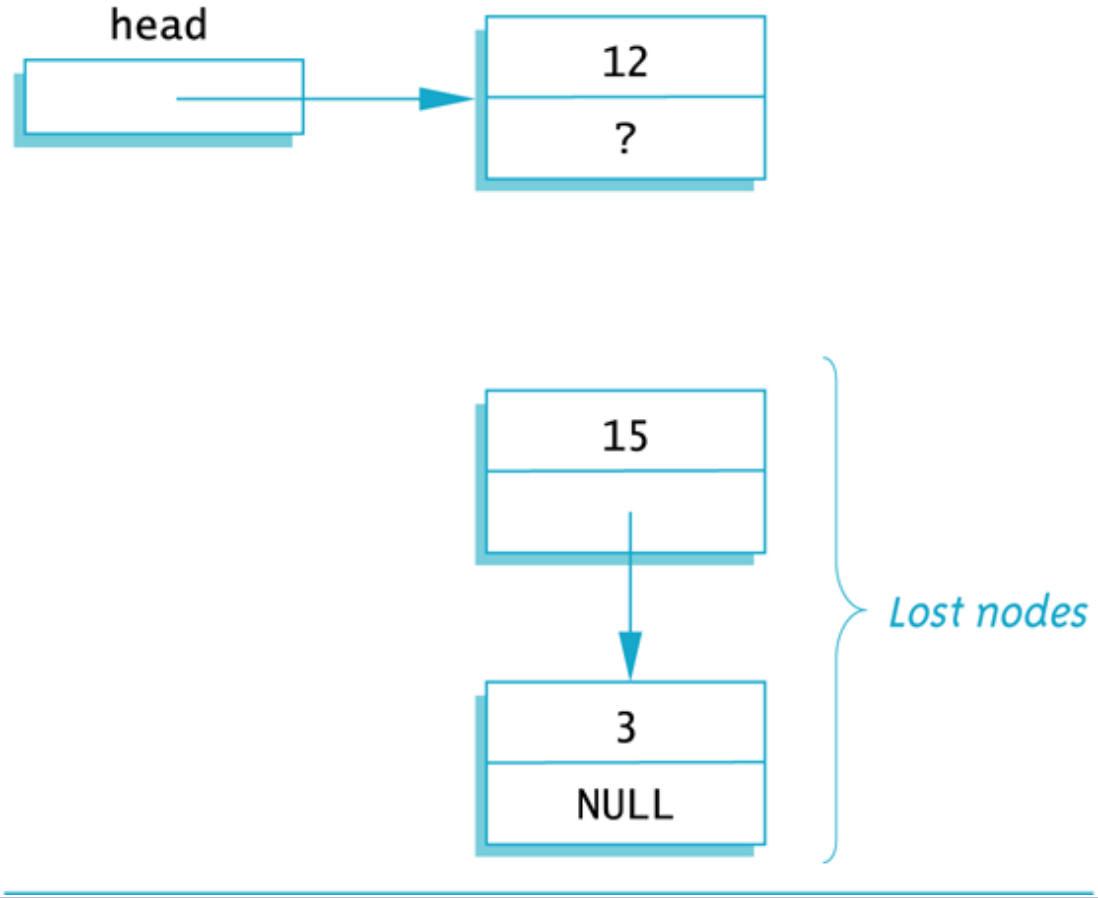
- Now to attach the new node to the list
  - The node that head used to point to is now lost!

Display 13.5

Display 13.5

Back Next

Lost Nodes



Memory Leaks

- Nodes that are lost by assigning their pointers a new address are not accessible any longer
- The program has no way to refer to the nodes and cannot delete them to return their memory to the freestore
- Programs that lose nodes have a memory leak
  - Significant memory leaks can cause system crashes

Searching a Linked List

- To design a function that will locate a particular node in a linked list:
  - We want the function to return a pointer to the node so we can use the data if we find it, else return NULL
  - The linked list is one argument to the function
  - The data we wish to find is the other argument
  - This declaration will work:

    NodePtr search(NodePtr head, int target);

Function search

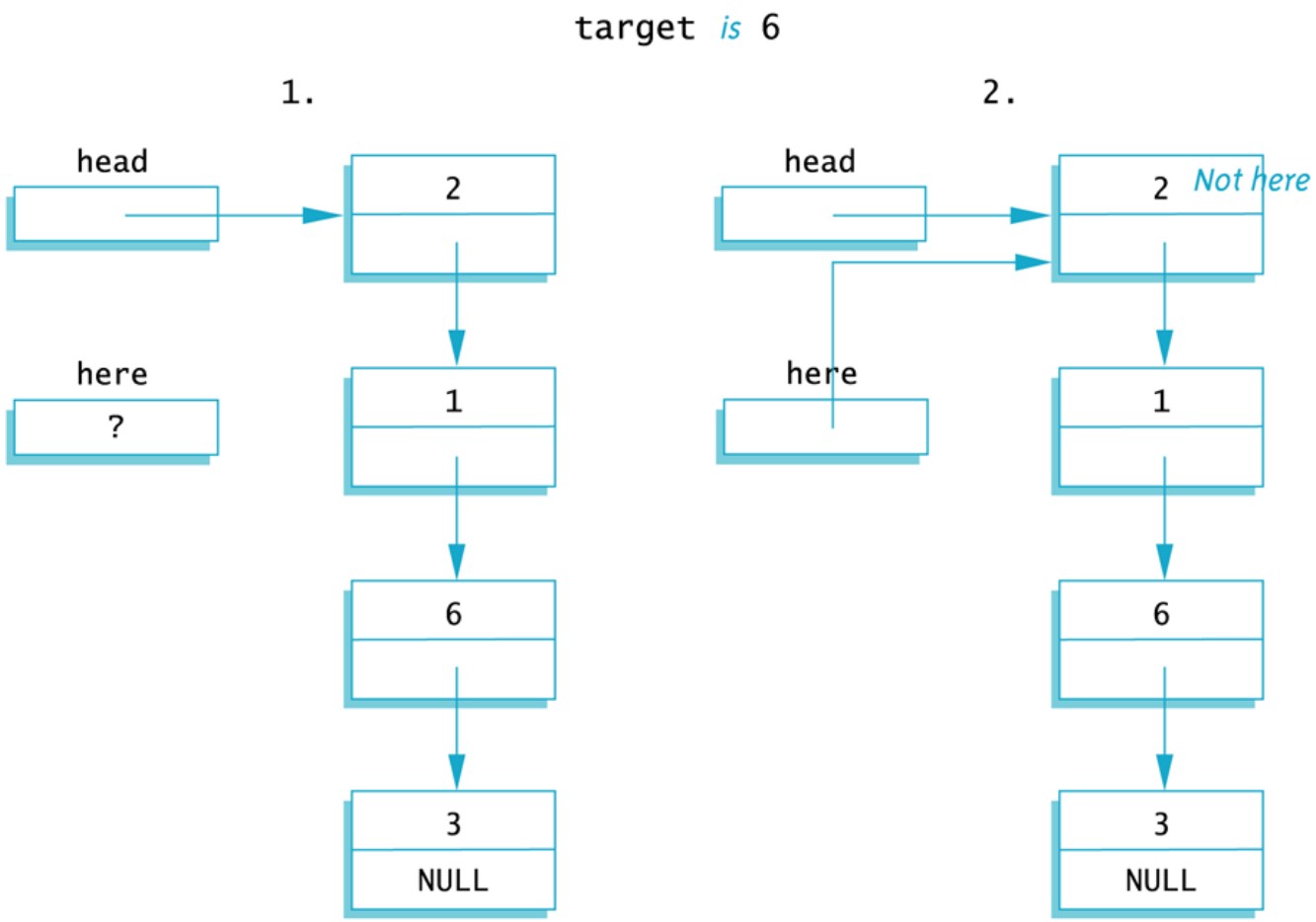
- Refining our function
  - We will use a local pointer variable, named here, to move through the list checking for the target
    - The only way to move around a linked list is to follow pointers
  - We will start with here pointing to the first node and move the pointer from node to node following the pointer out of each node

Display 13.6

Display 13.6

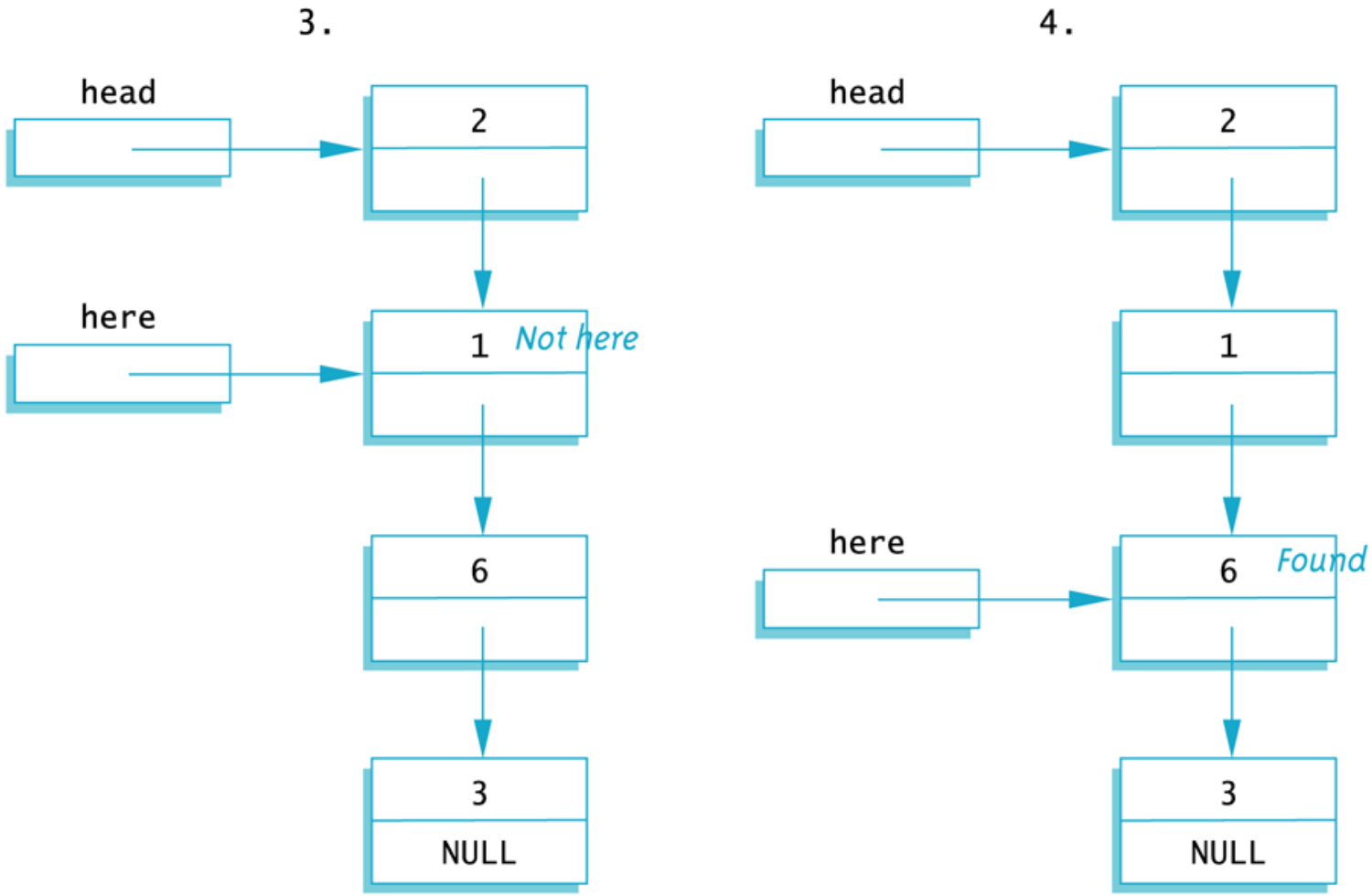
Back Next

Searching a Linked List



Display 13.6

Back Next



Pseudocode for search

- Make pointer variable here point to the head node

- while(here does not point to a node containing target
  AND here does not point to the last node)
  {
  make here point to the next node
  }
  If (here points to a node containing the target)
  return here;
  else
  return NULL;

Moving Through the List

- The pseudocode for search requires that pointer here step through the list
  - How does here follow the pointers from node to node?
  - When here points to a node, here->link is the address of the next node
  - To make here point to the next node, make the assignment:

```
here = here->link;
```

A Refinement of search

- The search function can be refined in this way:

```
here = head;
while (here->data != target && here->link != NULL)
{
     here = here->link;
}
if (here->data == target)
    return here;
else
    return NULL;
```

Check for last node

Searching an Empty List

- Always think about the special cases
  - If the list is empty, here equals NULL before the while loop so…
    - here->data is undefined
    - here->link is undefined
  - The empty list requires a special case in the search function

Display 13.7

Back Next

Function to Locate a Node in a Linked List

Function Declaration

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

NodePtr search(NodePtr head, int target);
//Precondition: The pointer head points to the head of
//a linked list. The pointer variable in the last node
//is NULL. If the list is empty, then head is NULL.
//Returns a pointer that points to the first node that
//contains the target. If no node contains the target,
//the function returns NULL.
```

Function Definition

```
//Uses cstddef:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;

    if (here == NULL)
    {
        return NULL;
    }
    else
    {
        while (here->data != target &&
                                    here->link != NULL)
            here = here->link;

        if (here->data == target)
            return here;
        else
            return NULL;
    }
}
```

Empty list case

Pointers as Iterators

- An iterator is a construct that allows you to cycle through the data items in a data structure to perform an action on each item
  - An iterator can be an object of an iterator class, an array index, or simply a pointer
- A general outline using a pointer as an iterator:

```
Node* iter;
for (iter = head; iter != NULL; iter = iter->link){
    //perform the action on the node that iter points to
}
```

  - Head is a pointer to the head node of the list

Iterator Example

- Using the previous outline of an iterator we can search a linked list in this way:
- Node* search(Node *head, int target){

```
    Node *iter = NULL;
    for (iter = head; iter != NULL; iter = iter->link){
        if (iter->data == target)
            break;
    }
    return iter;
}
```

Removing a Node

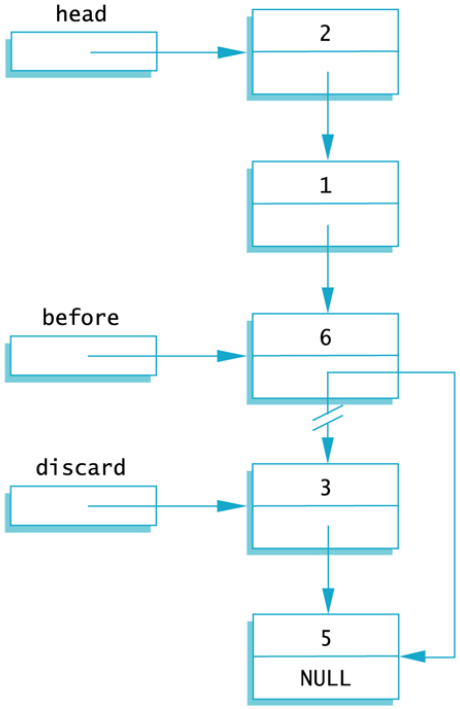
- To remove a node from a linked list
  - Position a pointer, before, to point at the node prior to the node to remove
  - Position a pointer, discard, to point at the node to remove
  - Perform: before->link = discard->link;
    - The node is removed from the list, but is still in memory
  - Return *discard to the freestore:
    delete discard;

Display 13.10

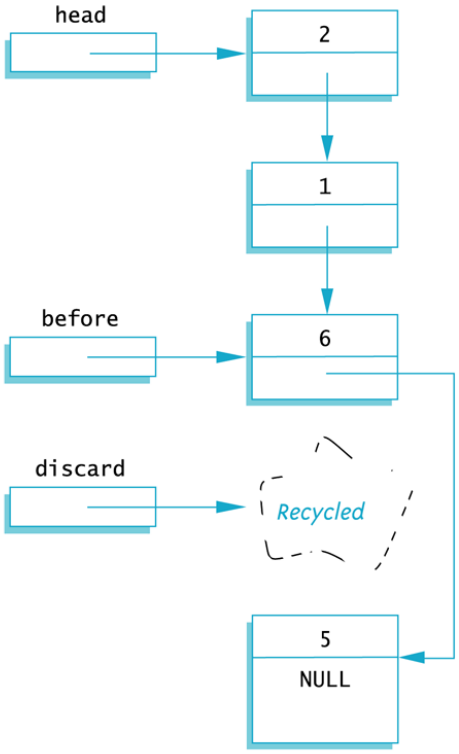
Removing a Node

1. Position the pointer `discard` so that it points to the node to be deleted, and position the pointer `before` so that it points to the node before the one to be deleted.

2. `before->link = discard->link;`



3. *delete* `discard;`



Display 13.10

Back Next

Removing a Node

```
// pre: both before and discard are not NULL, before->link
//        and discard point to the same node, discard points
//        to a dynamic node
// post: the node pointed by discard is removed from the
//          linked list, and it is freed

void remove(Node* before, Node* discard){
    before->link = discard->link;
    delete discard;
}
```

Inserting a Node Inside a List

- To insert a node after a specified node in the linked list:
  - Use another function to obtain a pointer to the node after which the new node will be inserted
    - Call the pointer after_me
  - Use function insert, declared here to insert the node:
    void insert(NodePtr after_me, int the_number);

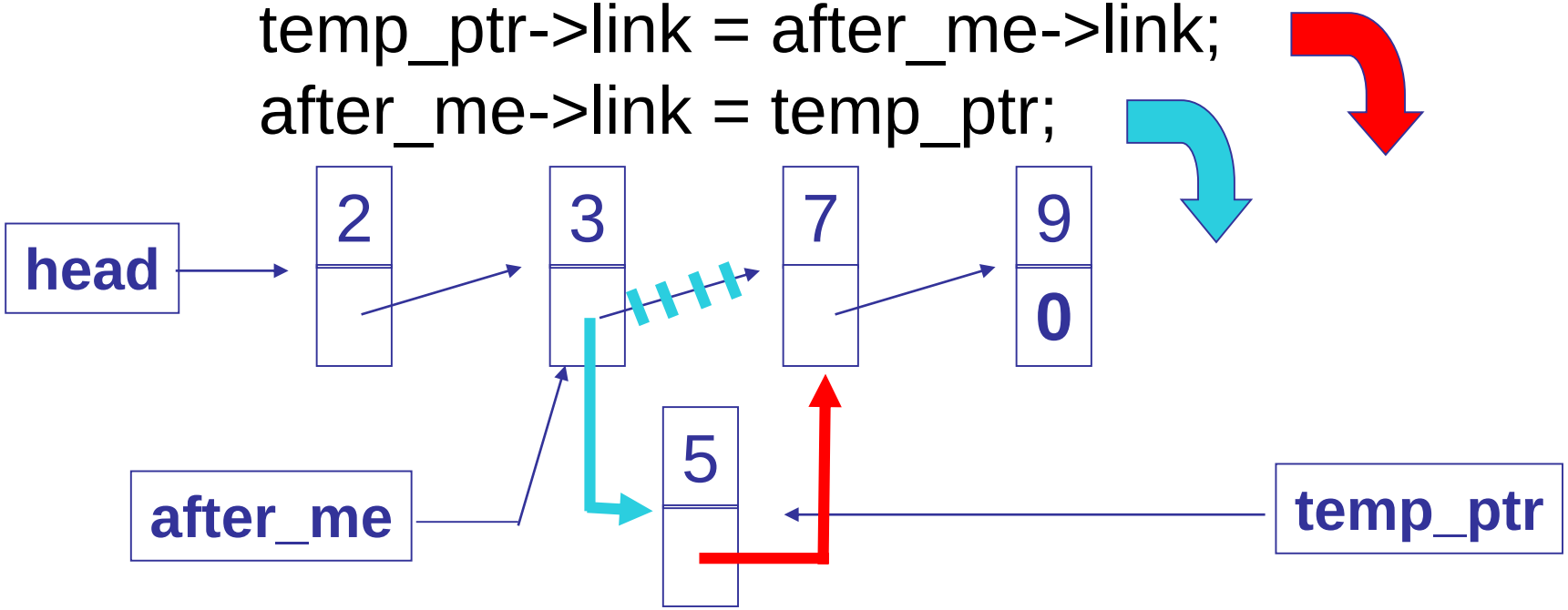
Inserting the New Node

- Function insert creates the new node just as head_insert did
- We do not want our new node at the head of the list however, so…
  - We use the pointer after_me to insert the new node

Inserting the New Node

- This code will accomplish the insertion of the new node, pointed to by temp_ptr, after the node pointed to by after_me:

```
temp_ptr->link = after_me->link;
after_me->link = temp_ptr;
```



Display 13.9

Function to Add a Node in the Middle of a Linked List

Function Declaration

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

void insert(NodePtr after_me, int the_number);
//Precondition: after_me points to a node in a linked
//list.
//Postcondition: A new node containing the_number
//has been added after the node pointed to by after_me.
```

Function Definition

```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```

Function insert Again

- Notice that inserting into a linked list requires that you only change two pointers
  - This is true regardless of the length of the list
  - Using an array for the list would involve copying as many as all of the array elements to new locations to make room for the new item
- Inserting into a linked list is often more efficient than inserting into an array