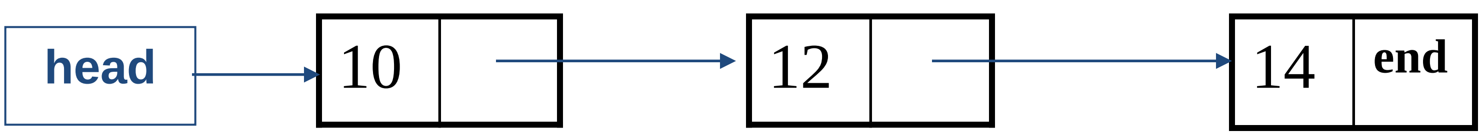
Nodes and Linked Lists

Nodes and Linked Lists

- A linked list is a list that can grow and shrink while the program is running
- A linked list is constructed using pointers
- A linked list often consists of structs or classes that contain a pointer variable connecting them to other dynamic variables
- A linked list can be visualized as items, drawn as boxes, connected to other items by arrows

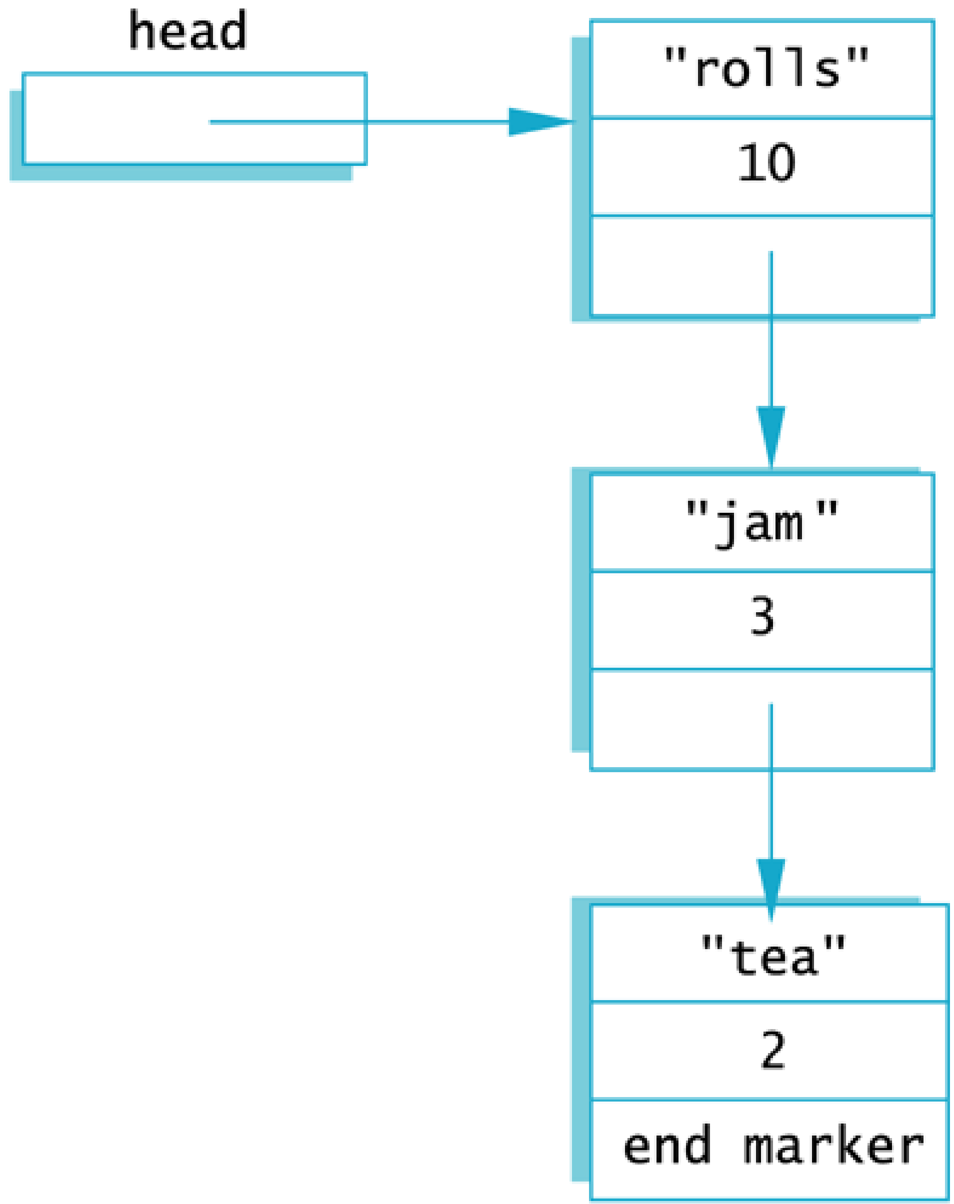


Nodes

- The boxes in the previous drawing represent the **nodes** of a linked list
  - Nodes contain the data item(s) and a pointer that can point to another node of the same type
    - The pointers point to the entire node, not an individual item that might be in the node

- The arrows in the drawing represent pointers

Nodes and Pointers



Implementing Nodes

- Nodes are implemented in C++ as structs or classes
  - **Example**: A structure to store two data items and
    a pointer to another node of the same type,
    along with a type definition might be:

```
struct ListNode
{
    string item;
    int count;
    ListNode* link;
};

typedef ListNode* ListNodePtr;
```

This circular definition is allowed in C++

The head of a List

- The box labeled head, on slide #4, is not a node, but a pointer variable that points to a node

- Pointer variable `head` is declared as:

```
ListNodePtr head;
```

Accessing Items in a Node

- Using the diagram on slide #4, this is one way to change the number in the first node from `10` to `12`:

```
(*head).count = 12;
```

  - `head` is a pointer variable so `*head` is the node that `head` points to
  - The parentheses are necessary because the dot `operator.` has higher precedence than the dereference `operator*`

The Arrow Operator

- The arrow `operator ->` combines the actions of the dereferencing operator * and the dot operator to specify a member of a struct or object pointed to by a pointer

```
(*head).count = 12;
```

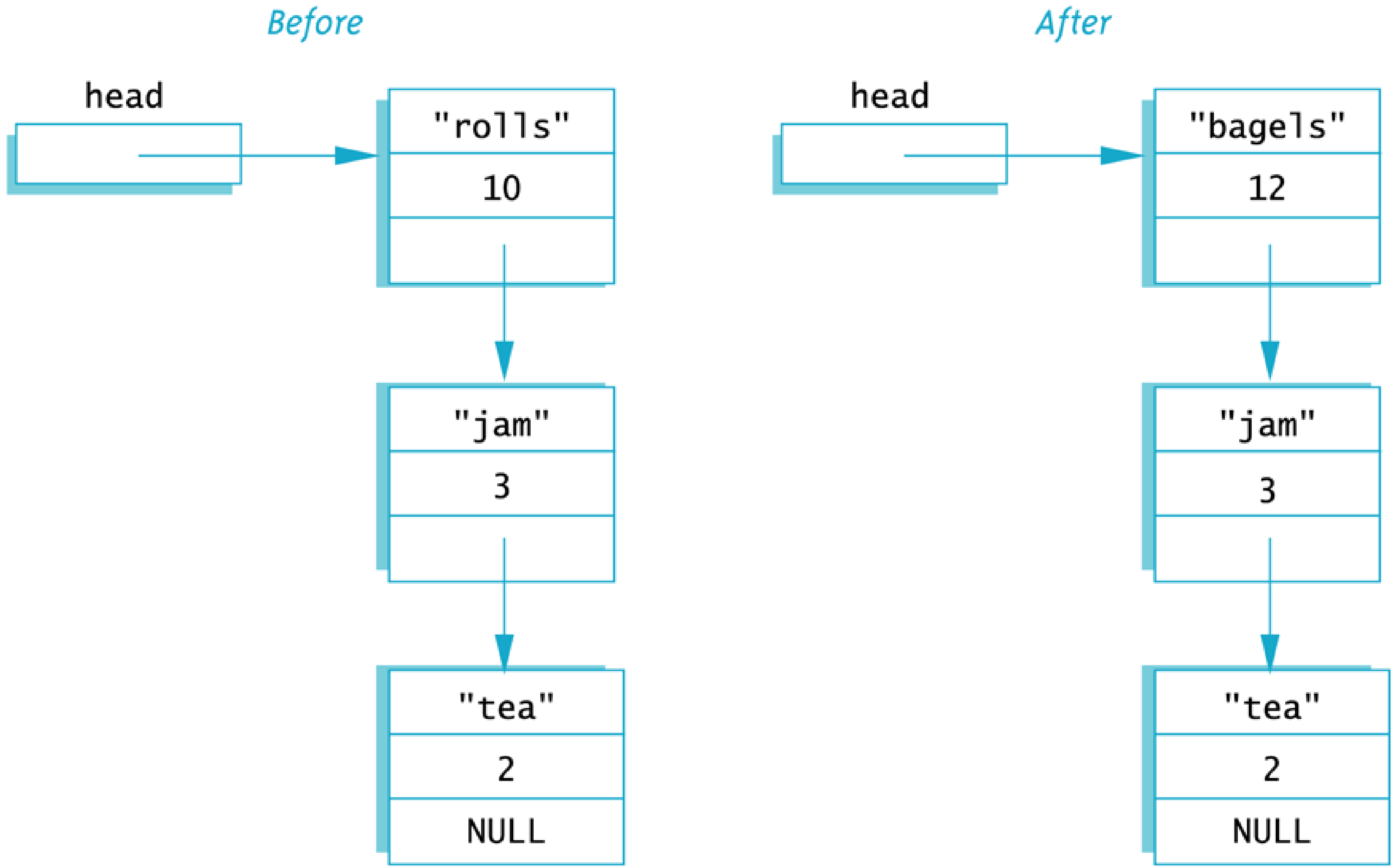
can be written as:

```
head->count = 12;
```

  - The arrow operator is more commonly used

Accessing Node Data

```
head->count = 12;
head->item = "bagels";
```



NULL

- The defined constant `NULL` is used as…
  - An end marker for a linked list
    - A program can step through a list of nodes by following the pointers, but when it finds a node containing `NULL`, it knows it has come to the end of the list
  - The value of a pointer that has nothing to point to
- The value of `NULL` is `0`
- Any pointer can be assigned the value `NULL`:

```
double* there = NULL;
```

To Use `NULL`

- A definition of `NULL` is found in several libraries, including `<iostream>` and `<cstddef>`
- A `using` directive is not needed for `NULL`

`nullptr`

- The fact that the constant NULL is actually the number 0 leads to an ambiguity problem. Consider the overloaded function below:

```
void func(int* p);
void func(int n);
```

- Which function will be invoked if we call `func(NULL)`?
- To avoid this, C++11 has a new constant, **`nullptr`**. It is not the integer zero, but a literal constant used to represent a null pointer.

Linked Lists

- The diagram on slide #9 depicts a linked list
- **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`

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

- Now, we can declare the pointer variable `head`:

```
NodePtr head;
```

  or:

```
Node* 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 (so far), 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 num);
```

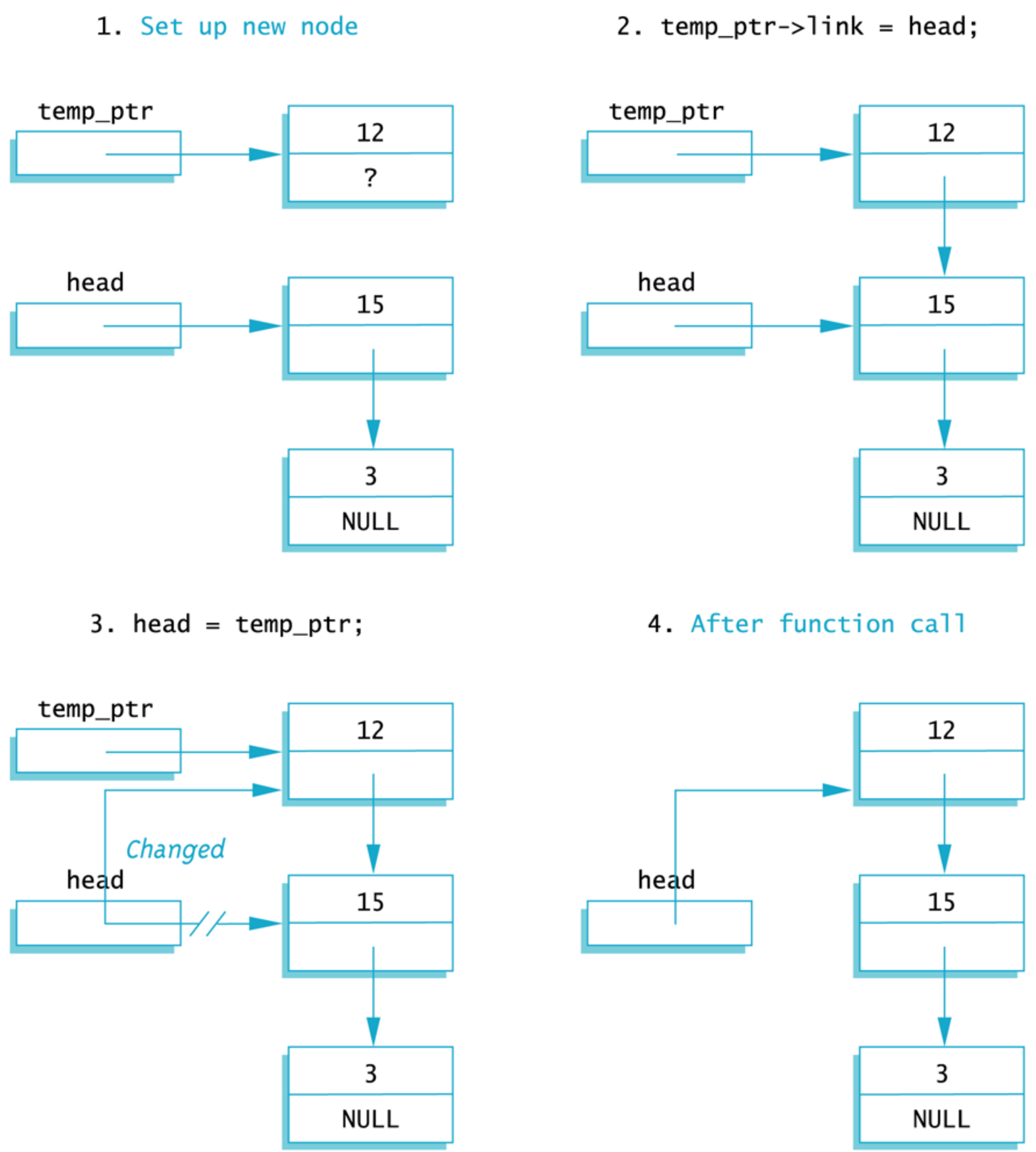
  - The first parameter is a `NodePtr` parameter that points to the first node in the linked list
  - The second parameter is the number/data to store in the list

- `head_insert` will create a new node for the `num`
  - The number will be copied to the `data` field in 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`

Adding a Node to a Linked List



Implementation for `head_insert`

```
void head_insert(NodePtr& head, int the_number);
//Precondition: The pointer variable head points to
//the head of a linked list.
//Postcondition: A new node containing the_number
//has been added at the head of the linked list.
```

Function Definition

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

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

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

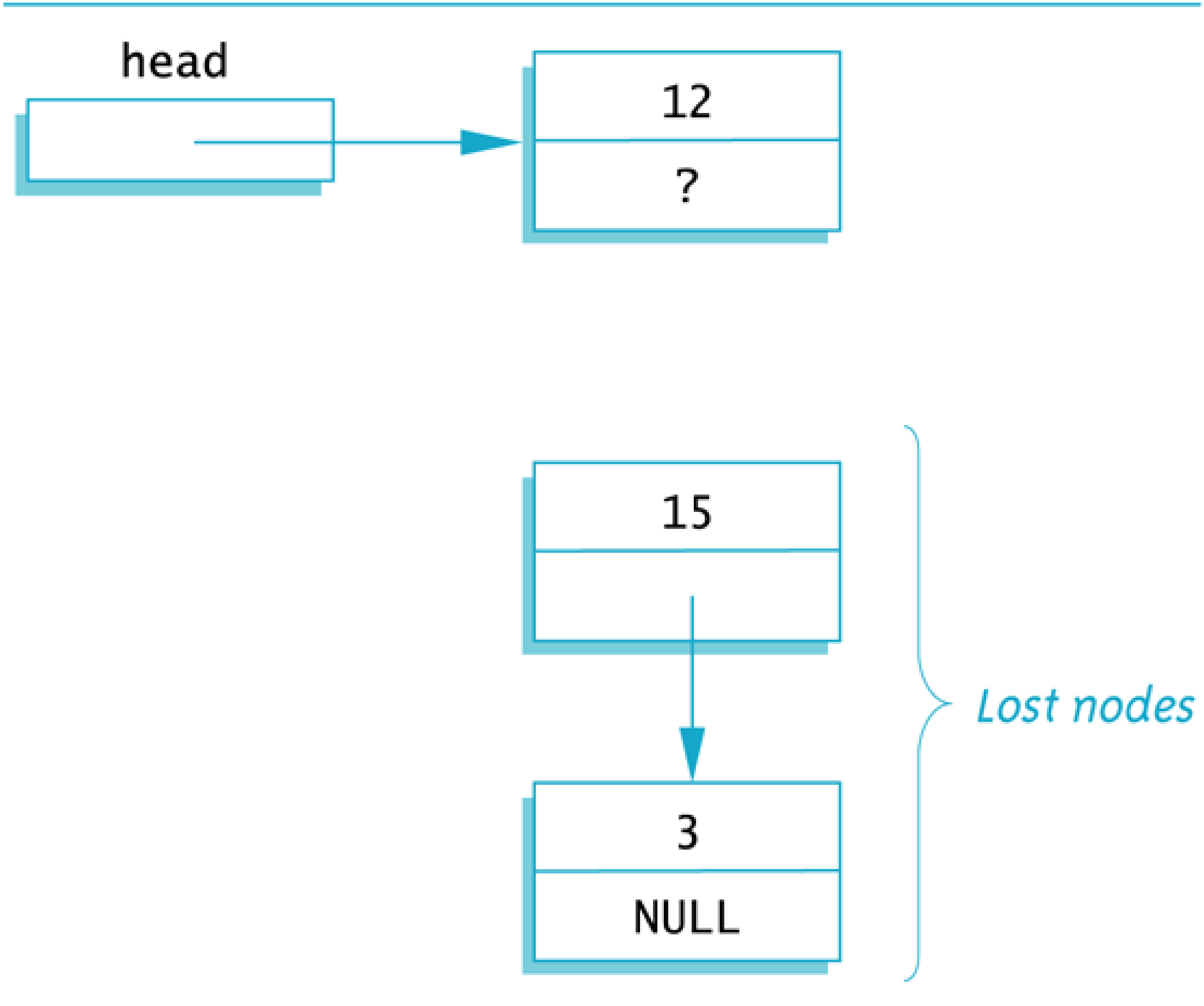
Losing Nodes

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

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

- Problem: the node that `head` used to point to is now lost! (See next slide)

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 heap
- Programs that lose nodes have a memory leak
  - Significant memory leaks can cause system crashes

Searching a Linked List

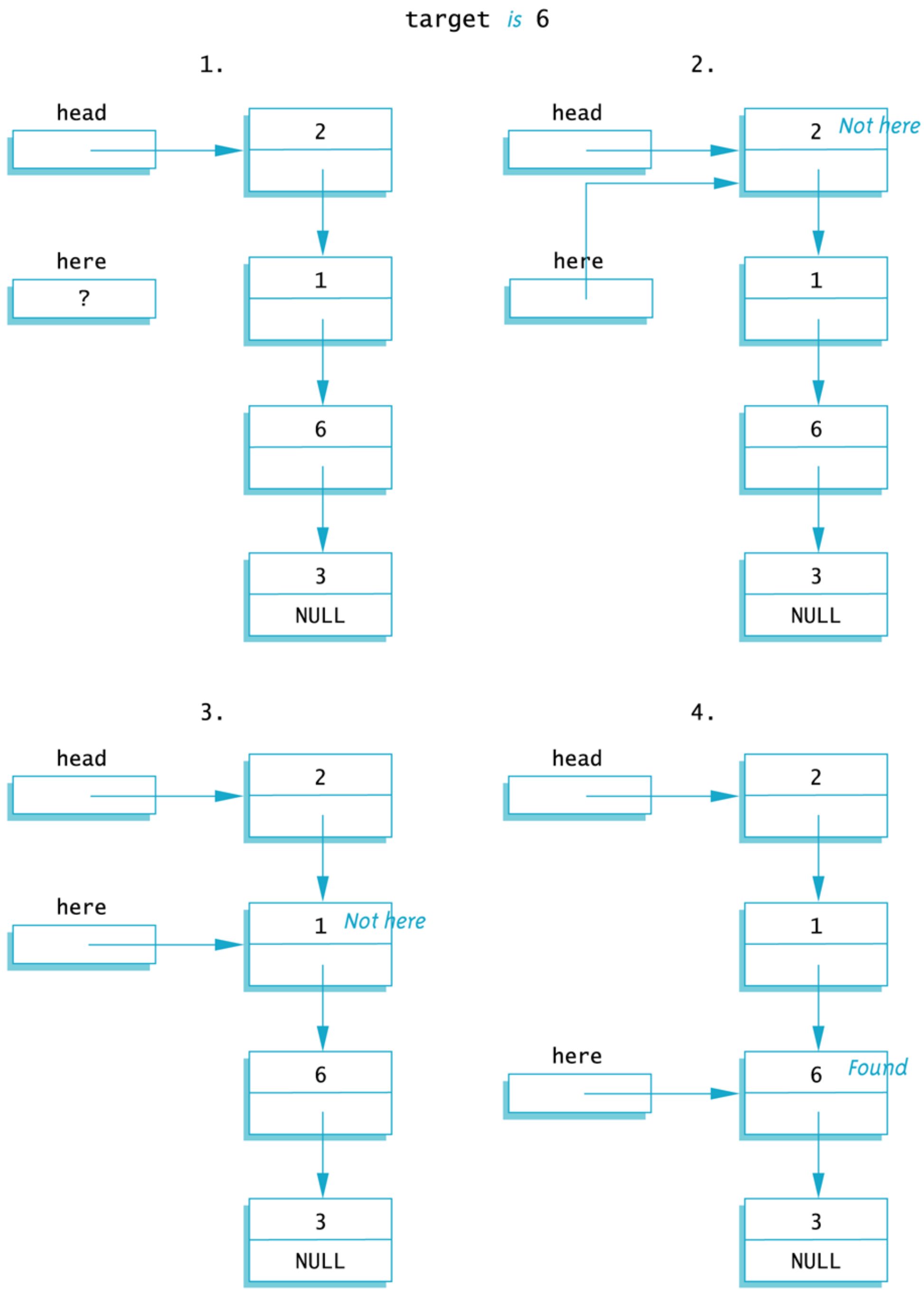
- To design a function that will locate a particular node in a linked list:
  - We want the function to return a pointer to the node so we can use the data if we find it, else return NULL
  - The linked list is one argument to the function
  - The data we wish to find is the other argument
  - This declaration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

```
NodePtr search(NodePtr head, int target);
```

Function `search`

- 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 (visiting each node) following the link

Searching a Linked List



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;
```

C++ implementation?

Searching an Empty List

- Our search pseudocode has a problem
  - 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 our `search` function

Refine our search function that can handle an empty list?

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_Type* iter;
for (iter = Head; iter != NULL; iter = iter->Link)
//perform the action on the node 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 display the contents of a linked list in this way:

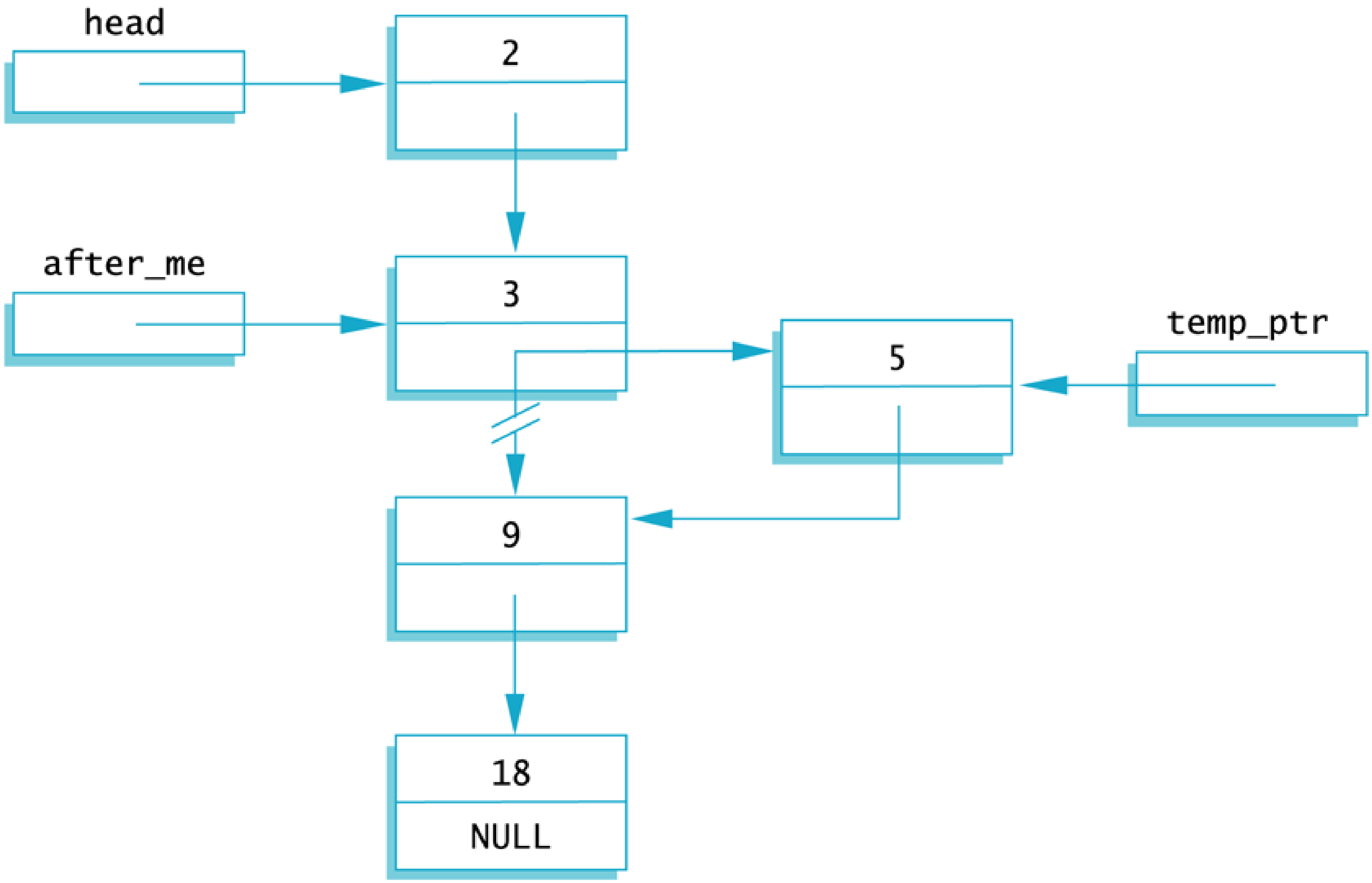
```
NodePtr iter;
for (iter = head; iter != NULL; iter = iter->link)
    cout << (iter->data);
```

`insert`ing 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 as follows to insert the node:

```
void insert (NodePtr after_me, int num);
```

Inserting in the Middle of a Linked List



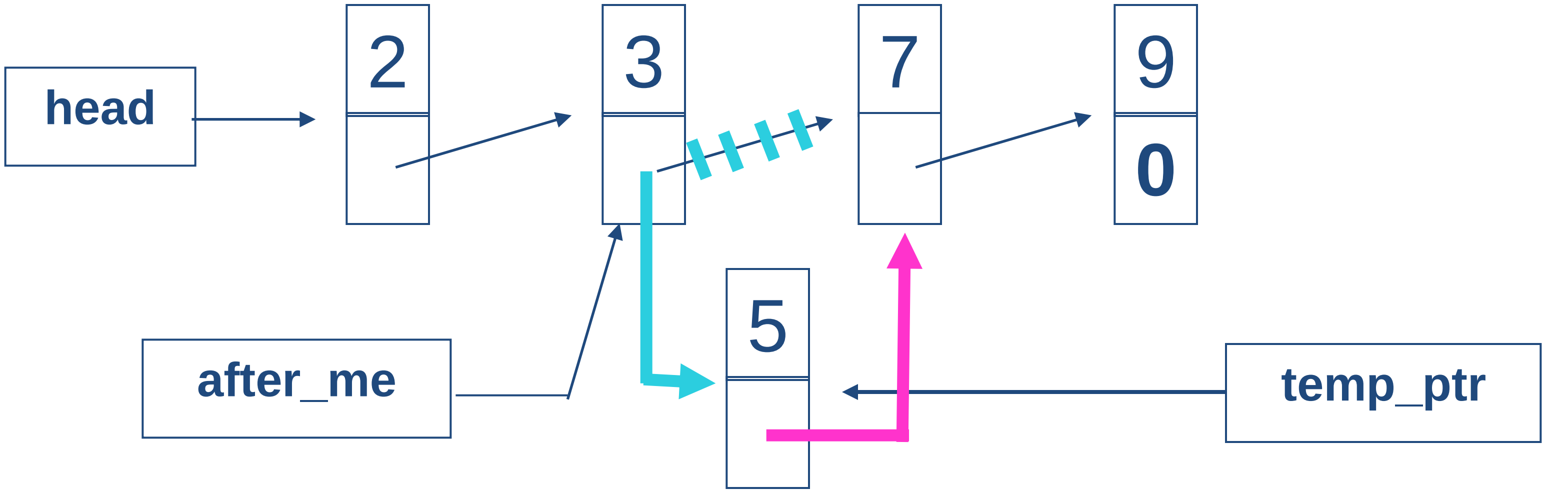
Inserting the New Node

- 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;
```



Caution!

- The order of pointer assignments is critical
  - If we changed `after_me->link` to point to `temp_ptr` first, we would lose the rest of the list!

The implementation of the insert function?

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*

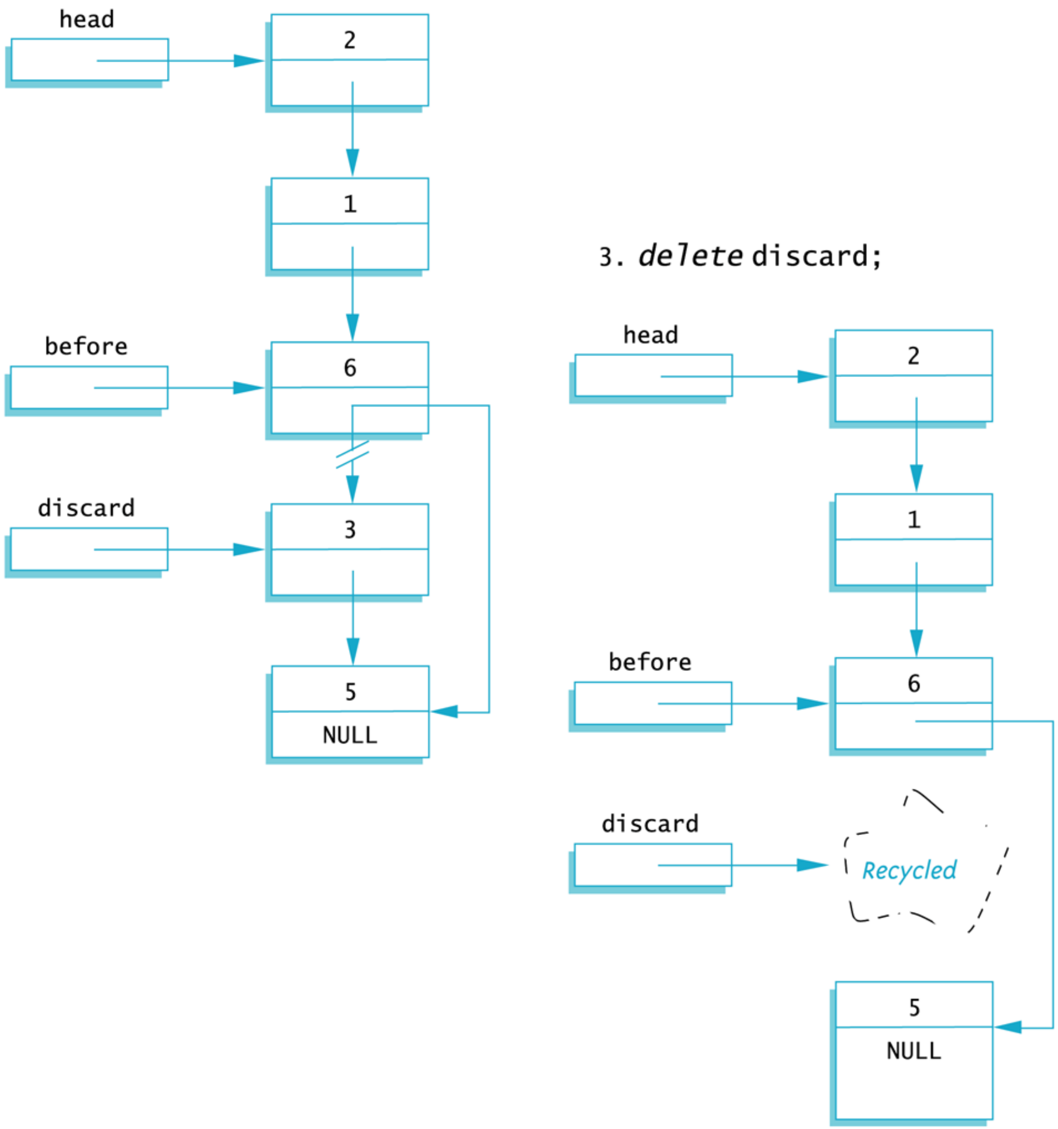
`remove` a Node

- 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 heap: delete `discard`;

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;`



Assignment With Pointers

- If `head1` and `head2` are pointer variables and `head1` points to the head node of a list:

```
head2 = head1;
```

  causes `head2` and `head1` to point to the same list

  – *There is only one list!*

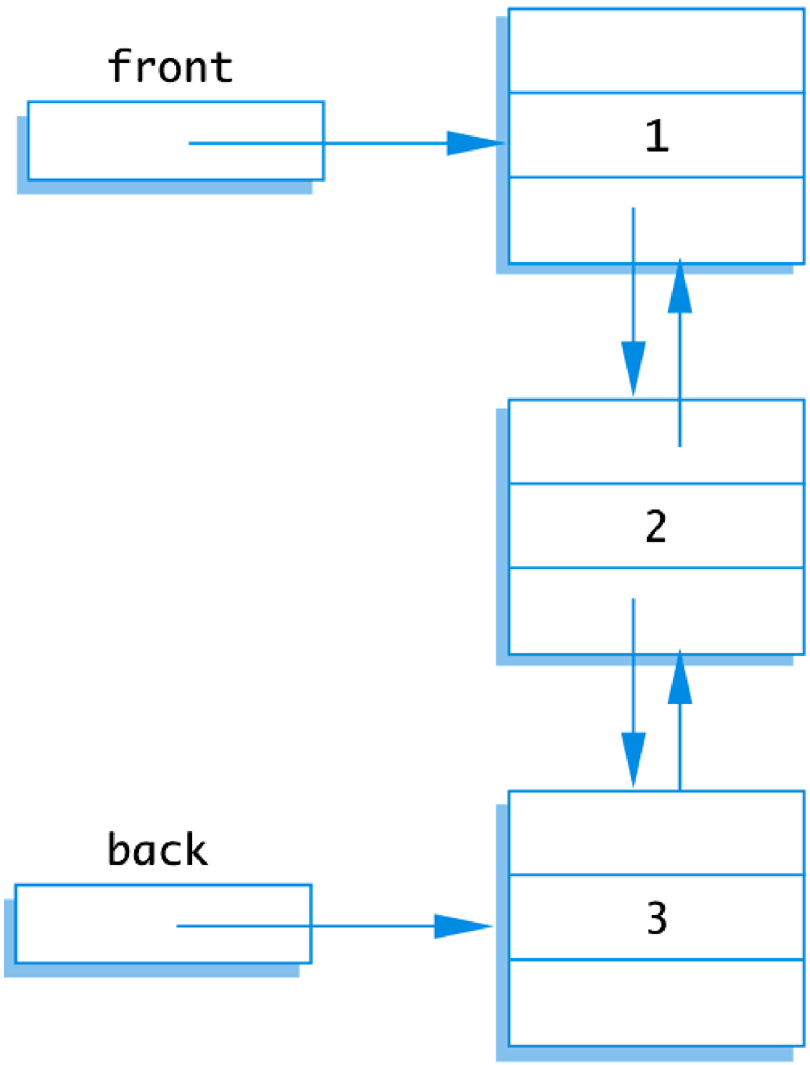
- If you want `head2` to point to a separate copy, you must copy the list node by node,
  or, overload the assignment operator appropriately

Variations on Linked Lists

- Many other data structures can be constructed using nodes and pointers
- **Doubly-Linked List**
  - *Each node has two links, one to the next node and one to the previous node*
  - *Allows easy traversal of the list in both directions*

```
struct Node
{
    int data;
    Node* forward_link;
    Node* back_link;
};
```

A Doubly Linked List



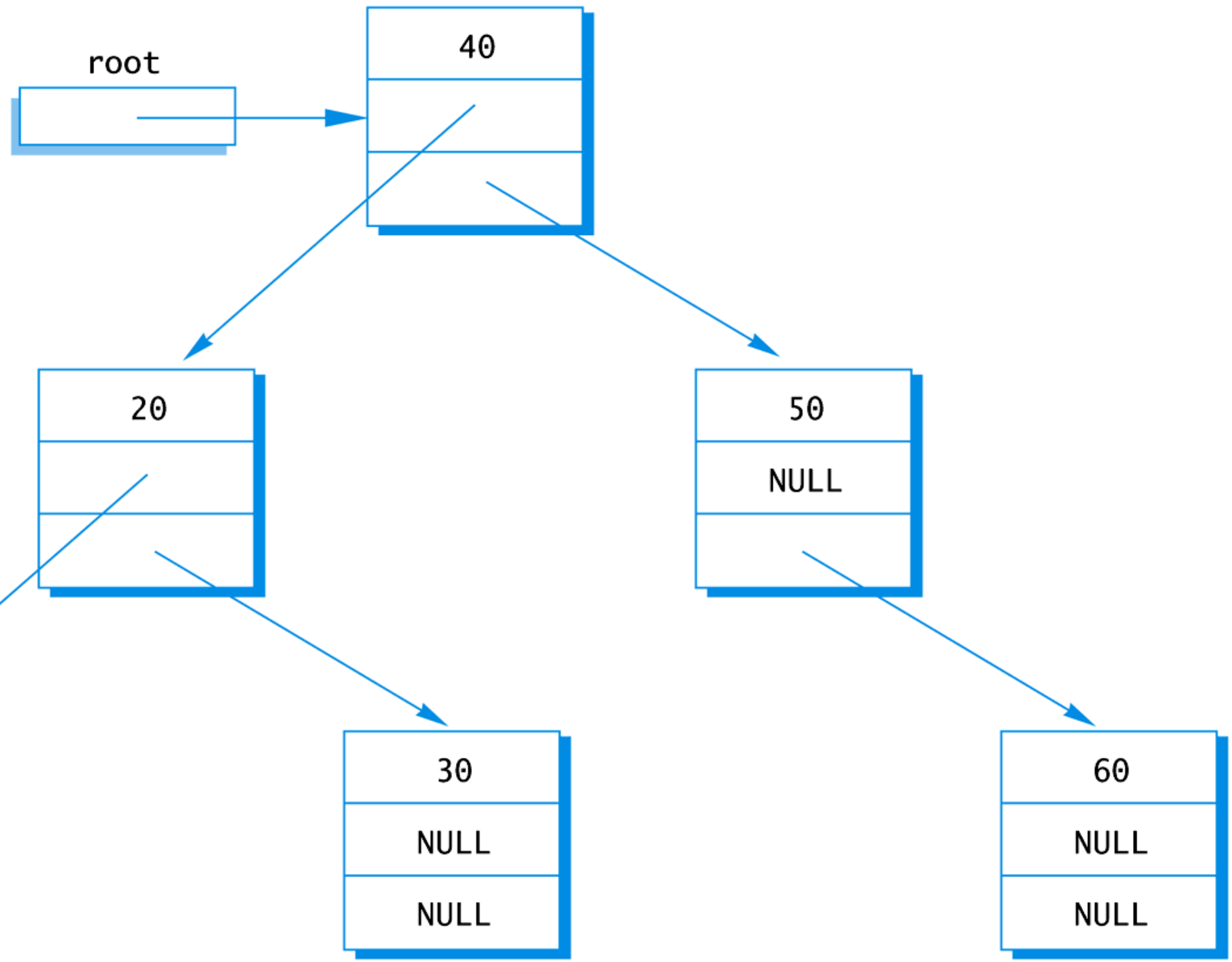
Binary Tree

- A tree is a data structure that looks like an upside-down tree with the root at the top
  - *No cycles*

- In a binary tree each node has at most two links

```
struct TreeNode
{
    int data;
    TreeNode* left_link;
    TreeNode* right_link;
};
```

DISPLAY 13.12 A Binary Tree



Linked List of Classes

- The preceding examples created linked lists of structs. We can also create linked lists using classes.
- Logic to use a class is identical except the syntax of using and defining a class should be substituted in place of that for a struct
- A example `Node` Class declaration is provided on the next slide.

- More about linked lists will be discussed in CMPT225

```
class Node
{
  public:
    Node( );
    Node(int value, Node *next);
    // Constructors to initialize a node

    int getData( ) const;
    // Retrieve value for this node

    Node *getLink( ) const;
    // Retrieve next Node in the list

    void setData(int value);
    // Use to modify the value stored in the list

    void setLink(Node *next);
    // Use to change the reference to the next node

  private:
    int data;
    Node *link;
};
```