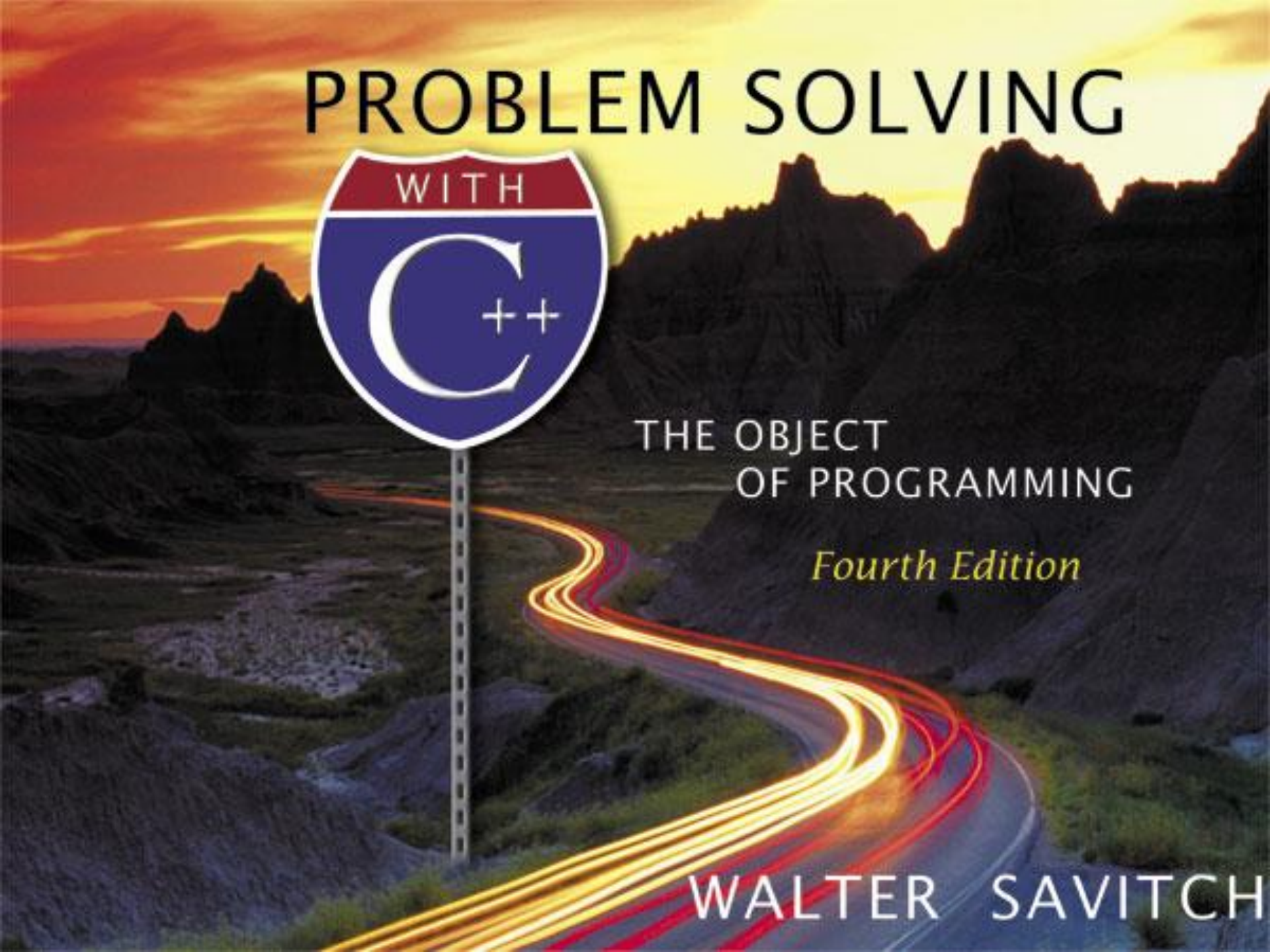

PROBLEM SOLVING
WITH
C++
THE OBJECT
OF PROGRAMMING
Fourth Edition
WALTER SAVITCH

Chapter 15

Pointers and Linked Lists

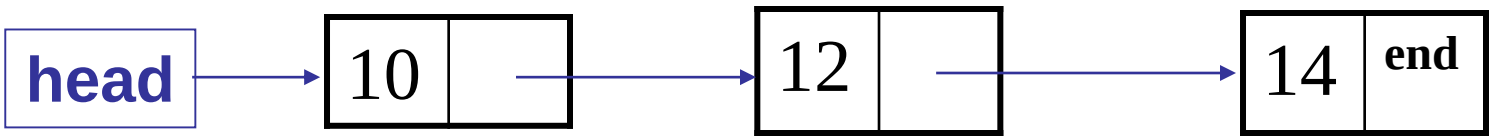
Created by David Mann, North Idaho College

Overview

- Nodes and Linked Lists(15.1)
- A Linked List Application (15.2)

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

Display 15.1

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, in display 15.1, 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 of 15.1, 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

Display 15.2

Linked Lists

- The diagram in Display 15.2 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

- 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 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

Display 15.3

Translating head_insert to C++

- The pseudocode for head_insert can be written in C++ using these lines in place of the lines of pseudocode:
  - 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

Display 15.4

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 15.5

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 15.6

Pseudocode for search

1. Make pointer variable **here** point to the head node
2. 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
   }
3. 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->next;
 }
 if (here->data = = target)
    return here;
 else
    return NULL;
```

Check for last node

Searching an Empty List

- Our search algorithm 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
  - A refined search function that handles an empty list is shown in **Display 15.7**

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);

Display 15.8

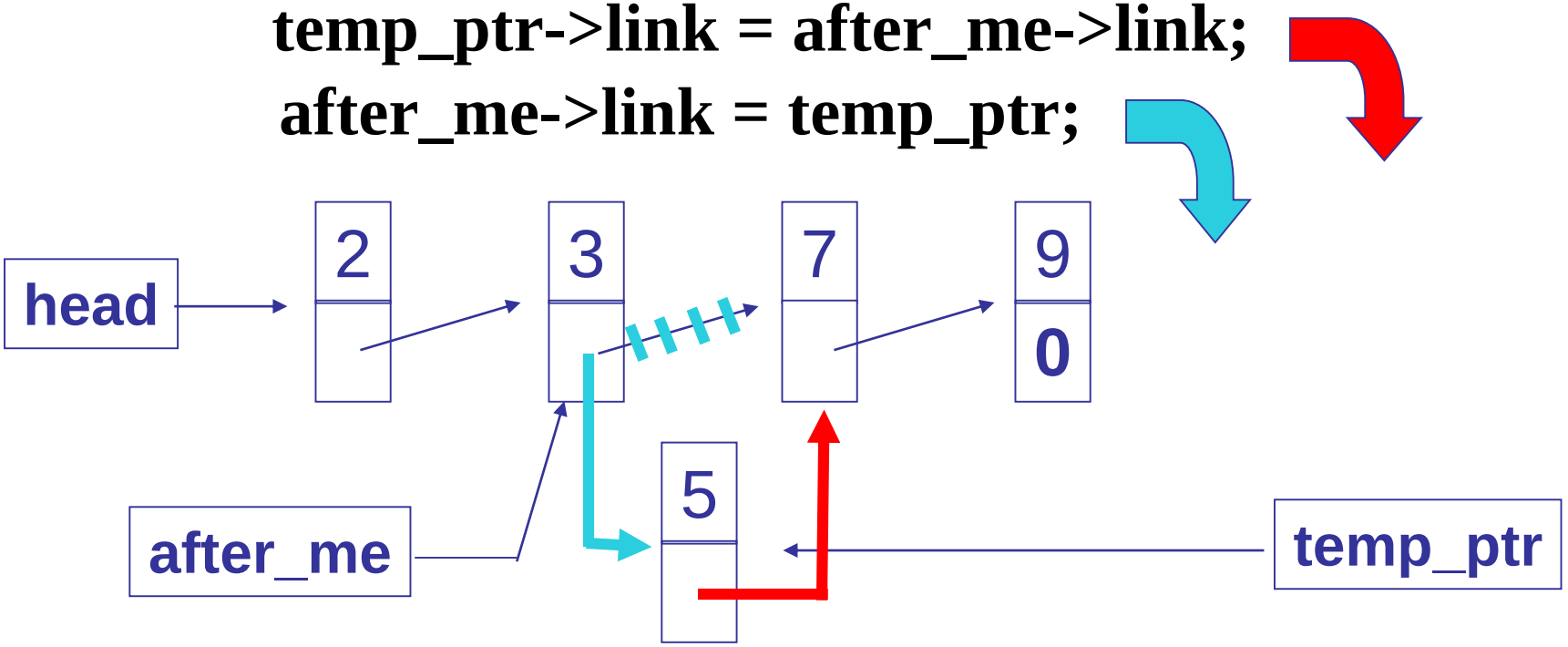
Inserting the New Node

- 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 loose the rest of the list!

- The complete insert function is shown in **Display 15.9**

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

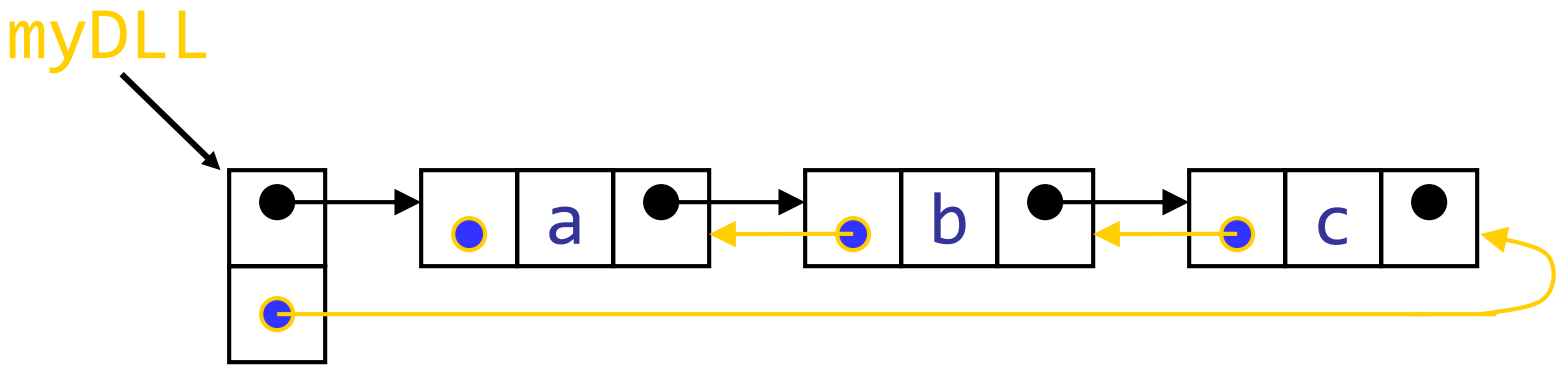
Removing a Node

- 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 15.10

Doubly-linked lists

- Here is a doubly-linked list (DLL):



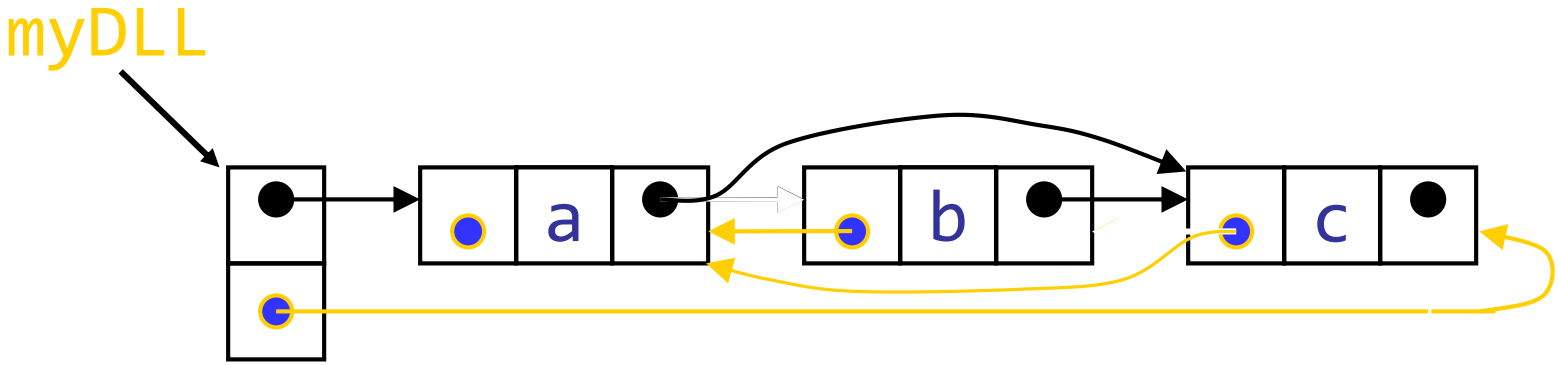
- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



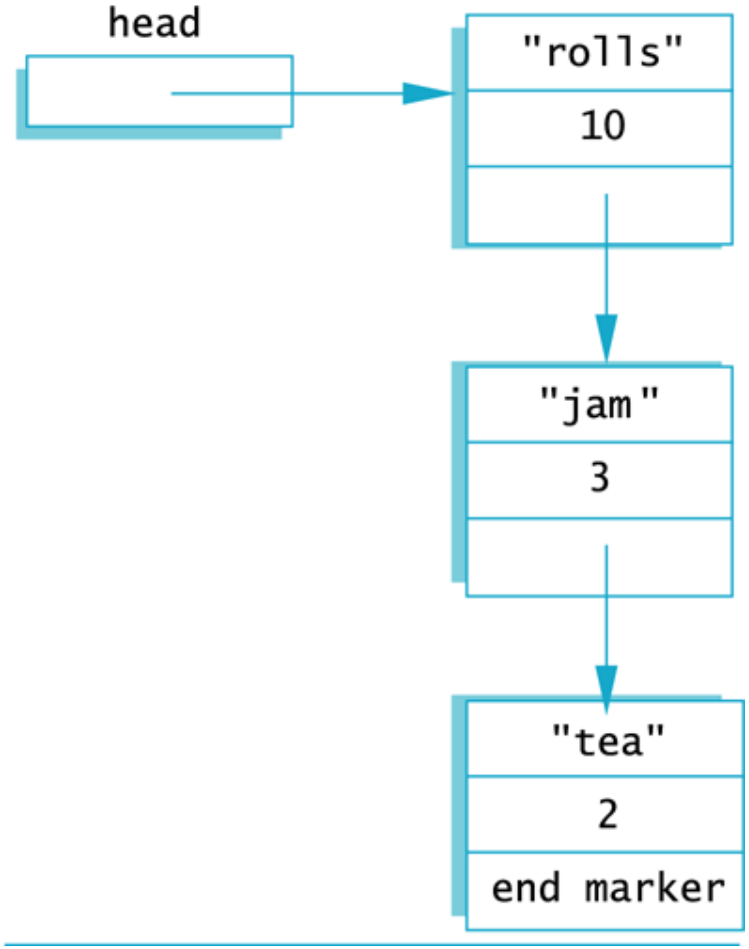
- We don't have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

Linked list nowadays!

- Google:
- Why you should never, ever, EVER use linked-list

Display 15.1

Nodes and Pointers

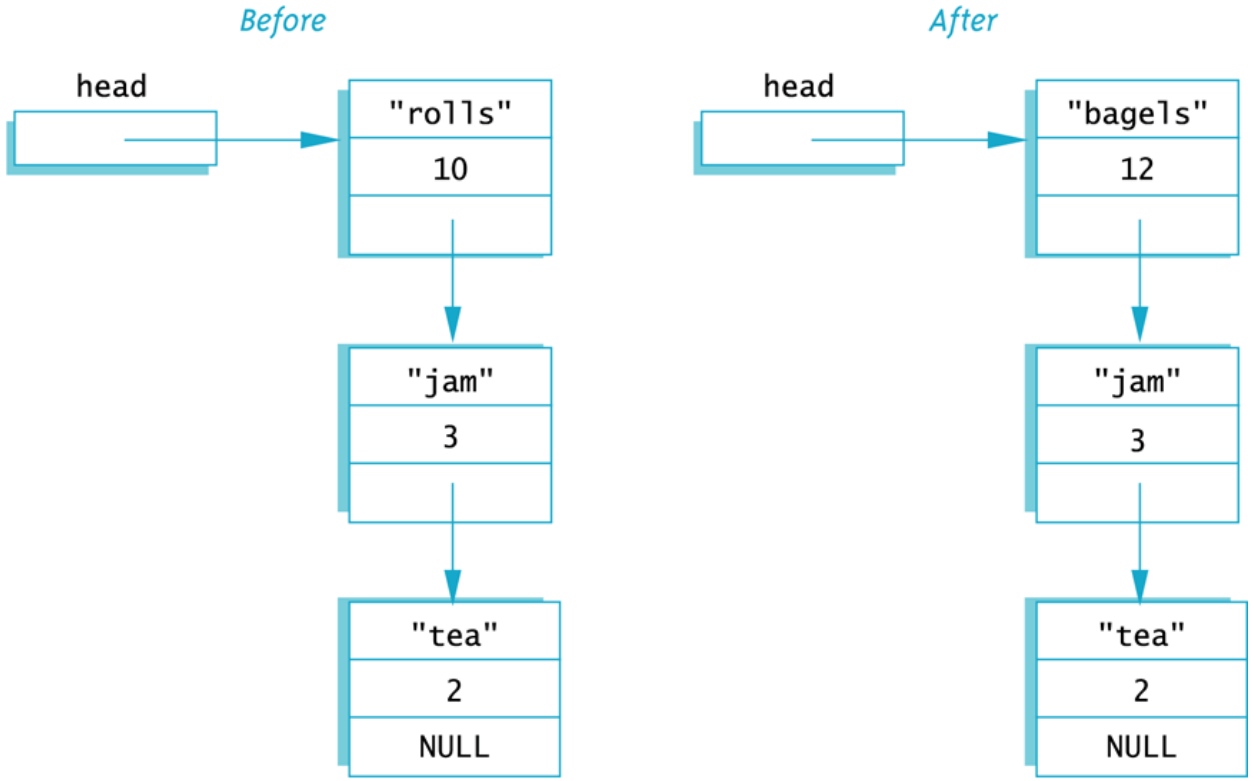


Display 15.2

Back Next

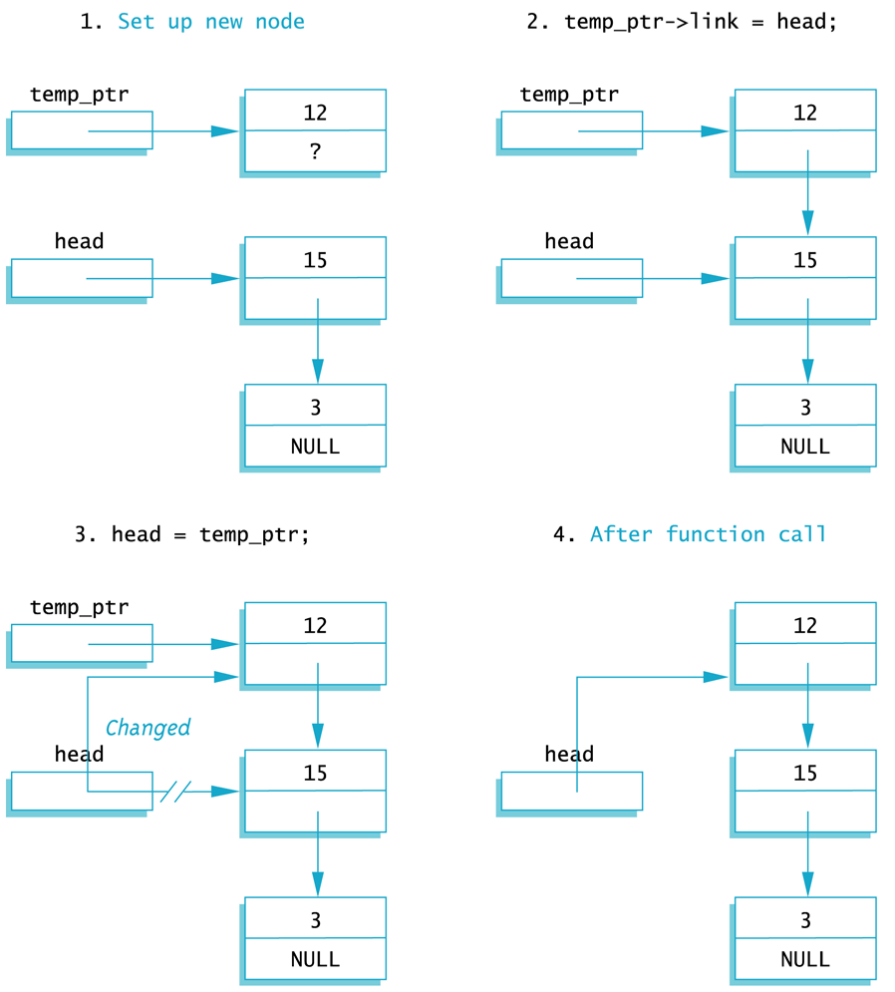
Accessing Node Data

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



Display 15.3

Adding a Node to a Linked List



Display 15.4

Function to Add a Node at the Head of a Linked List

Function Declaration

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

typedef Node* NodePtr;

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.
```

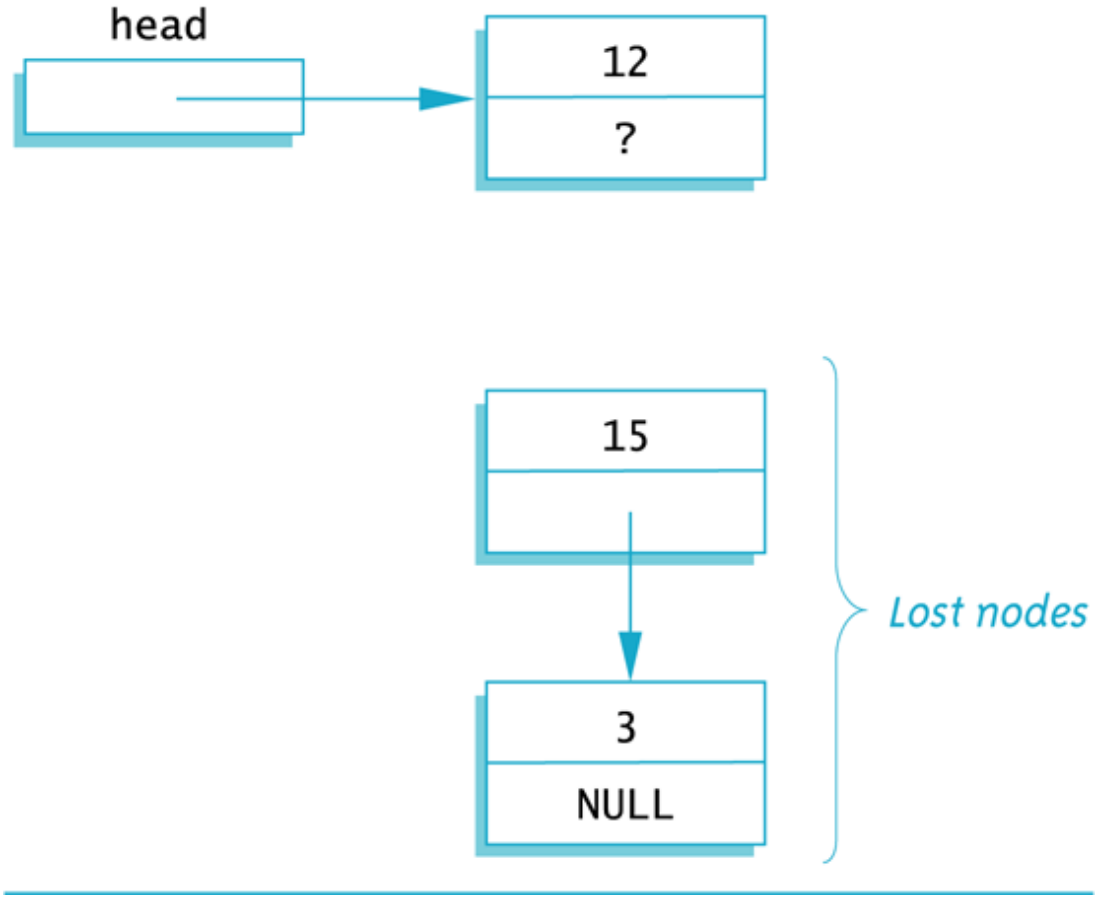
Function Definition

```
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;

}
```

Display 15.5

Back Next

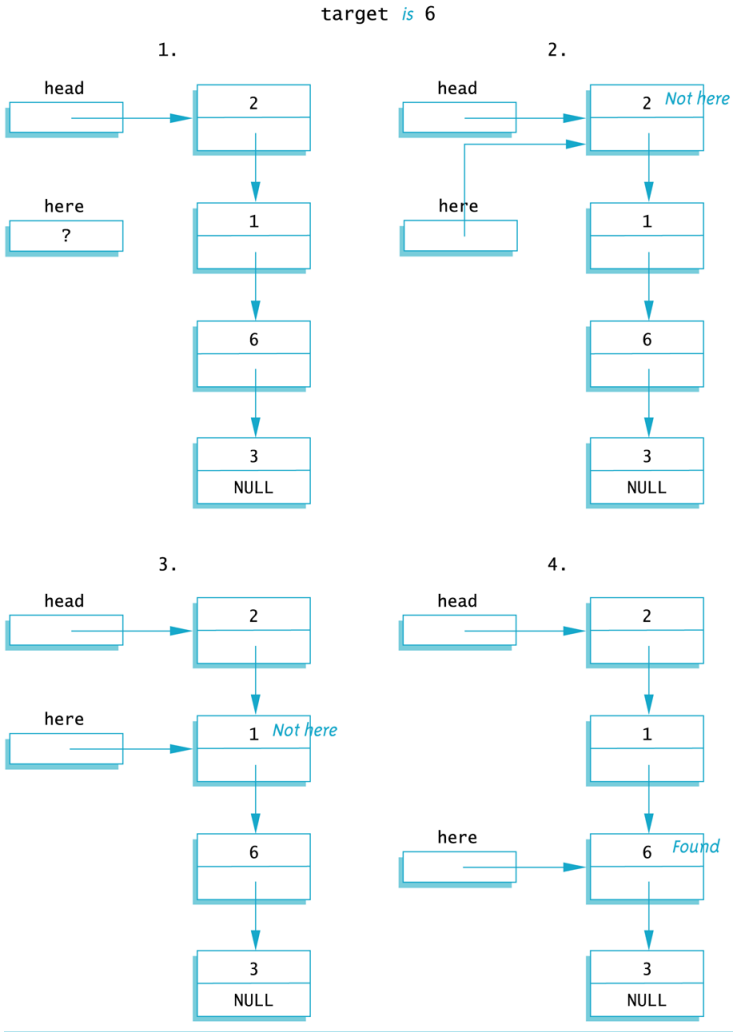
Lost Nodes



Display 15.6

Back Next

Searching a Linked List



Display 15.7

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

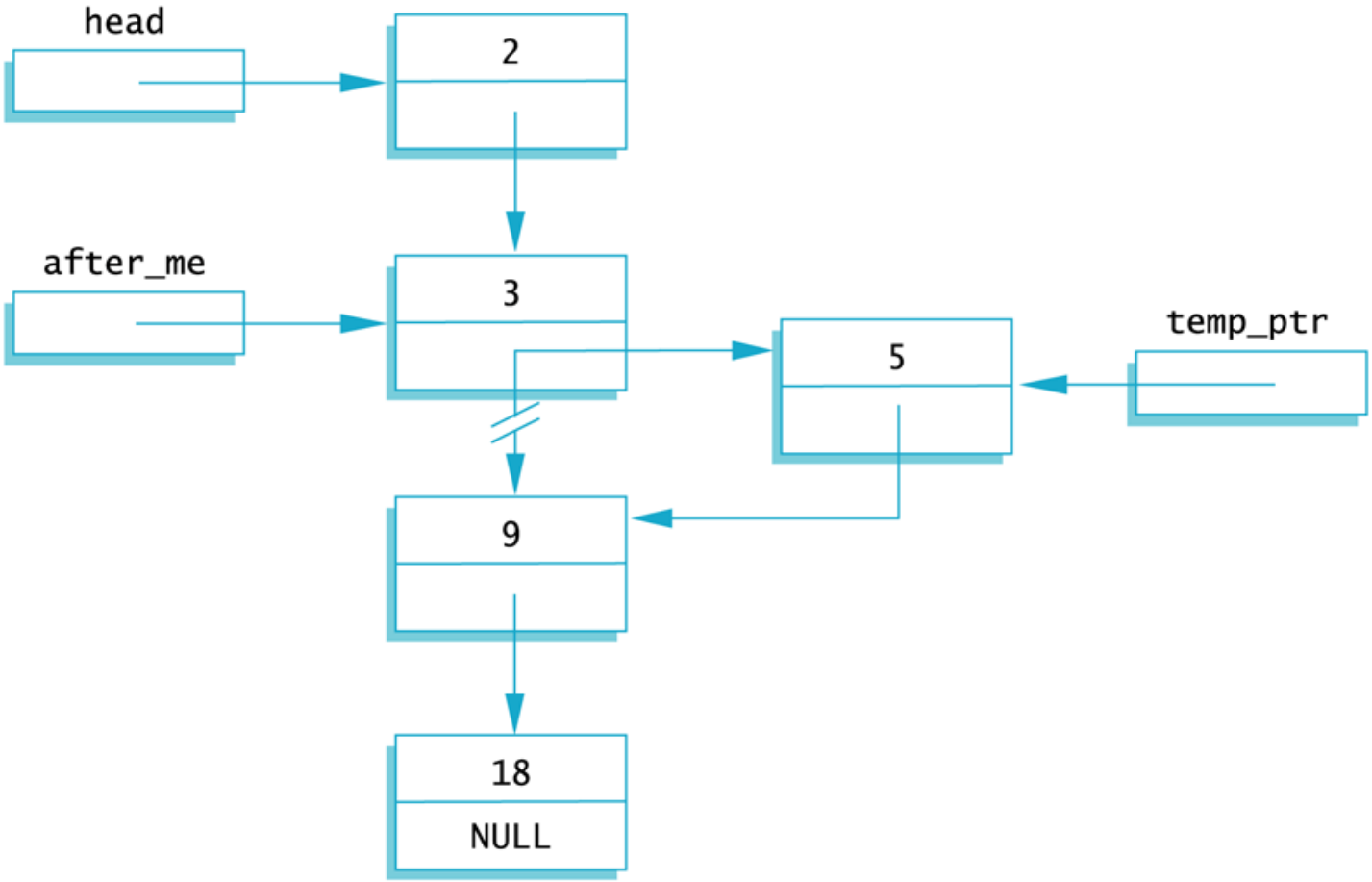
```
//Uses cstddef:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;

    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

Display 15.8

Back Next

Inserting in the Middle of a Linked List



Display 15.9

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

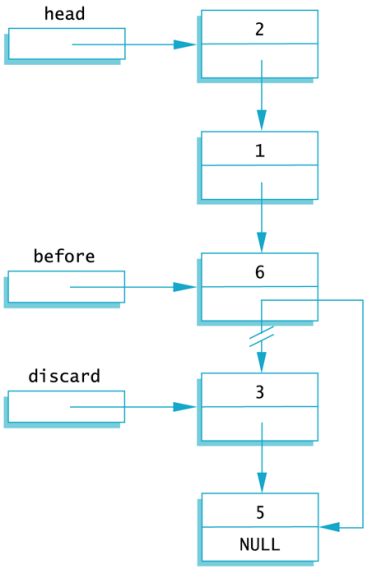
```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

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

Display 15.10

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;`