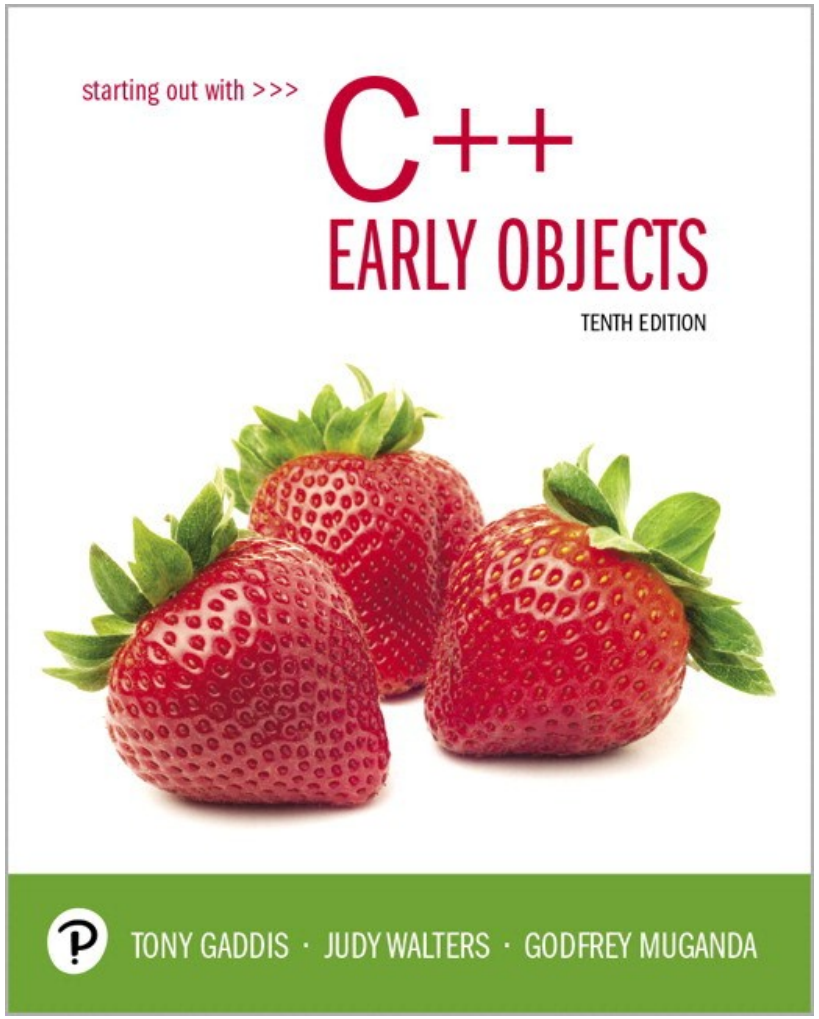
Starting Out with C++ Early Objects

Tenth Edition



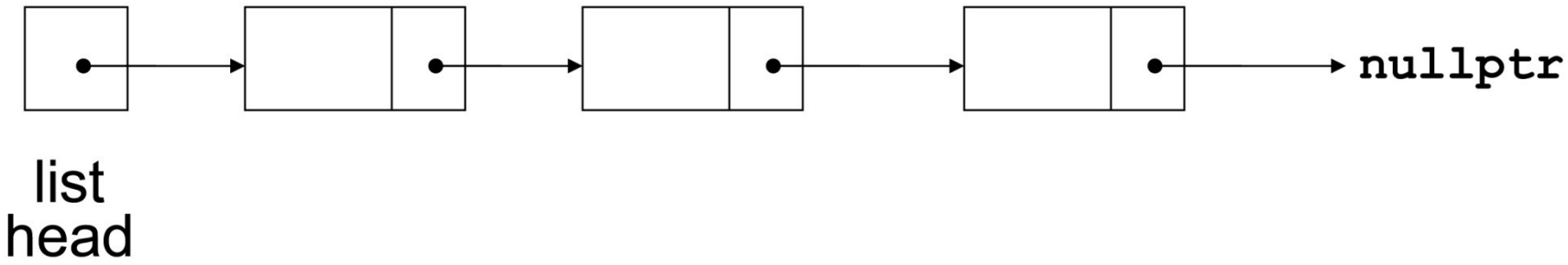
Chapter 18

Linked Lists

Topics

18.1 Introduction to Linked Lists

- Linked list: series of dynamically allocated data structures (nodes) with each node containing a pointer to its successor
- The last node in the list has its successor pointer set to **`nullptr`**

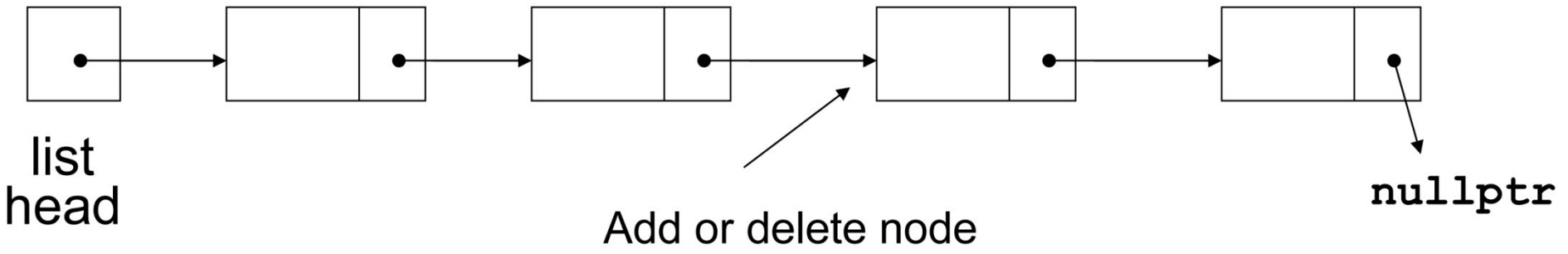


Linked List Terminology

- The node at the beginning is called the head of the list
- The entire list is identified by the pointer to the head node. This pointer is called the list head.

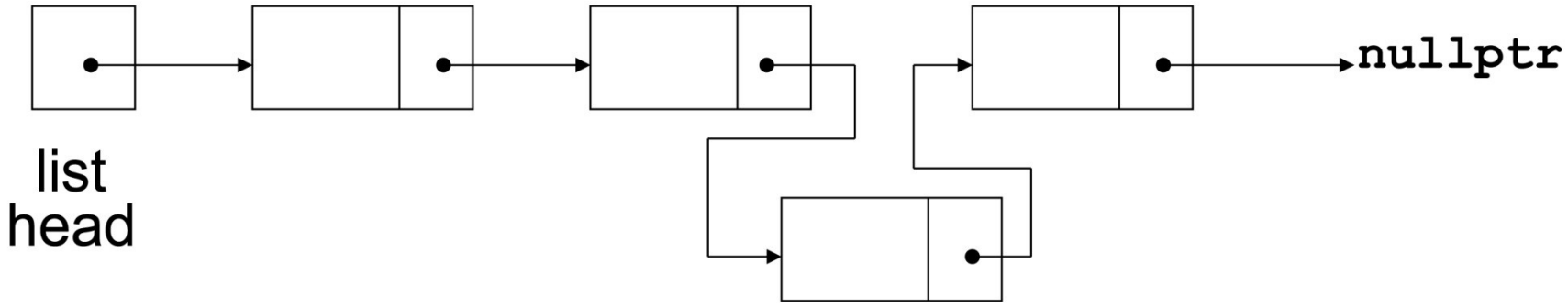
Linked Lists

- Nodes can be added to or removed from the linked list during execution
- Addition or removal of nodes can take place at beginning, end, or middle of the list



Linked Lists vs. Arrays and Vectors

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Unlike vectors, insertion or removal of a node in the middle of the list is very efficient



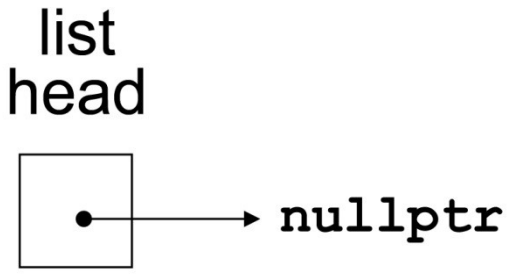
Node Organization

A node contains:

- data: one or more data fields – may be organized as structure, object, etc.
- a pointer that can point to another node

Empty List

- A list with no nodes is called the empty list
- In this case the list head is set to **`nullptr`**



C++ Implementation

Implementation of a node requires a structure containing a pointer to a structure of the same type. This creates a self-referential data structure:

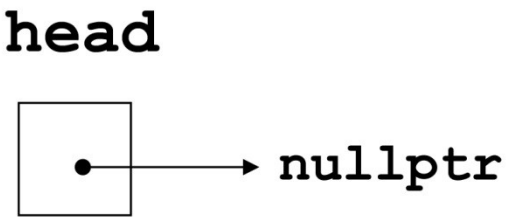
```
struct ListNode
{
   int data;
   ListNode *next;
};
```

Creating an Empty List

- Define a pointer for the head of the list:

```
ListNode *head = nullptr;
```

- Head pointer is initialized to **`nullptr`** to indicate that this is an empty list



C++ Implementation of a Node

Nodes can be equipped with constructors:

```
struct ListNode
{
   int data;
   ListNode *next;
   ListNode(int d, ListNode*
 p=nullptr)
     {data = d; next = p;}
};
```

Building a List from a File of Numbers

```
ListNode *head = NULL;
int val;
while (inFile >> val)
{
   // add new nodes at the head
   head = new ListNode(val, head);

   // Note that assignment is right-to-
   // left.  The present value of head
   // is used when the node is created,
   // then the address of the new node
   // is assigned to head.
};
```

Traversing a Linked List 1 of 2

- List traversal visits each node in a linked list to display contents, validate data, etc.
- Basic process of traversal:

set a pointer to the head pointer

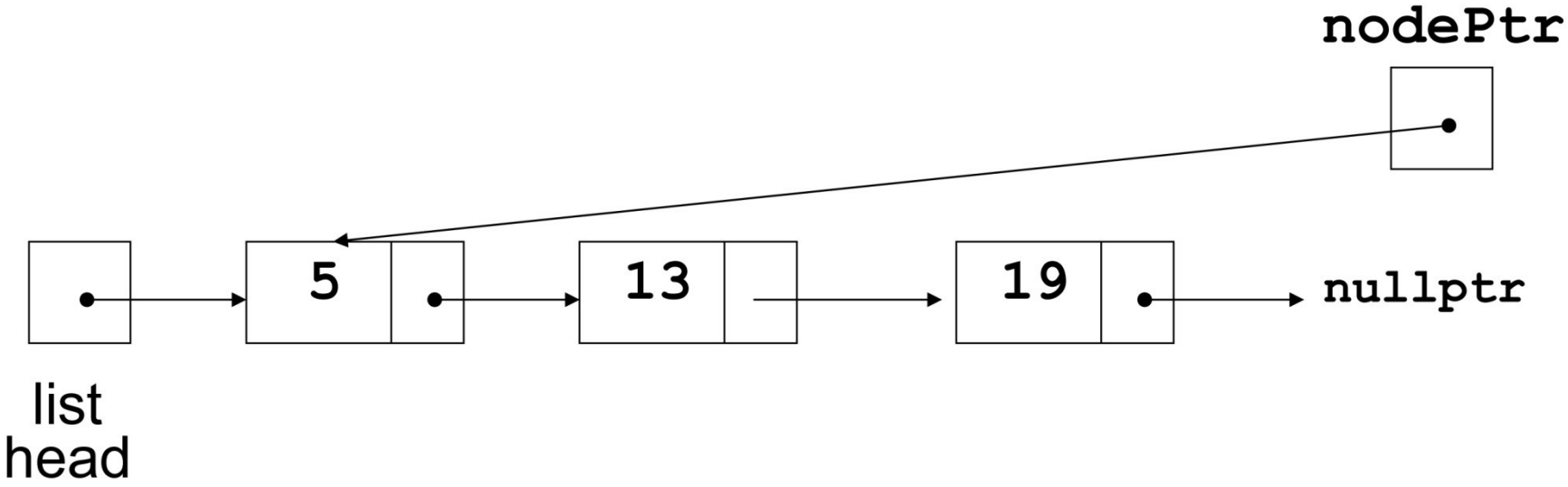
while pointer is not **`nullptr`**

process data

set pointer to the successor of the current node

end while

Traversing a Linked List 2 of 2



`nodePtr` points to the node containing **`5`**, then the node containing **`13`**, then the node containing **`19`**, then points to **`nullptr`**, and the list traversal stops

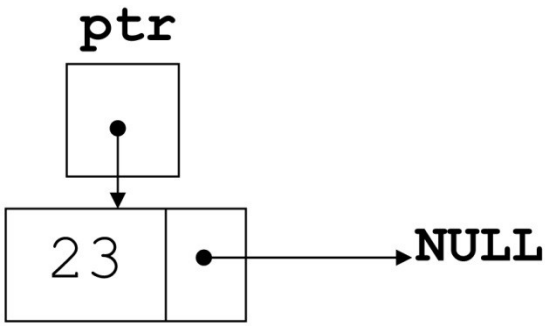
17.8 Linked List Operations

Basic operations:

- add a node to the list
- Delete/remove a node from the list
- traverse the linked list
- delete/destroy the linked list

Creating a Node

```
ListNode *ptr;
int num = 23;
ptr = new ListNode(num);
```



Adding an Item 1 of 2

To add an item to the end of the list:

- If the list is empty, set **`head`** to a new node containing the item

```
head = new ListNode(num);
```

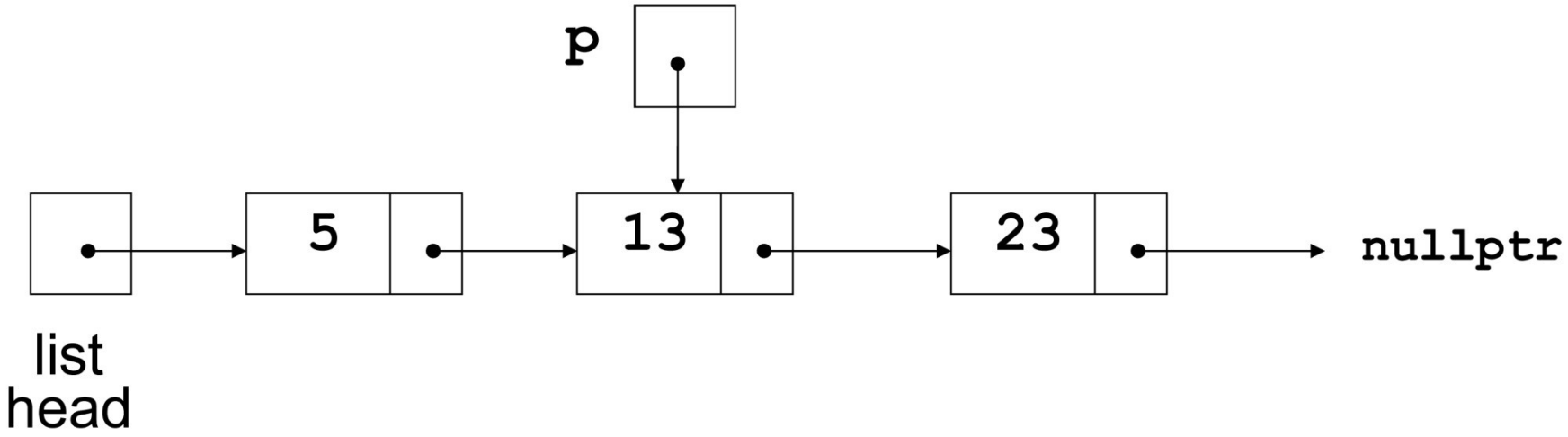
- If the list is not empty,
  - move a pointer **`p`** from head to the last node using

```
while(p->next != NULL)
    p = p->next;
```

  - then add a new node containing the item

```
p->next = new ListNode(num);
```

Adding an Item 2 of 2



List originally has nodes with 5 and 13.

`p` locates the last node, then a node with a

new item, 23, is added

Maintaining a Sorted List

- You may want to keep the nodes in a linked list in order according to their data fields.
- In this case, adding new nodes to the beginning or the end will not work.
- Here are two possibilities (ascending order):
  - The add point is at the head of the list (because the item at the head is already greater than the item being added, or because the list is empty
  - The add point is after an existing node in a non-empty list

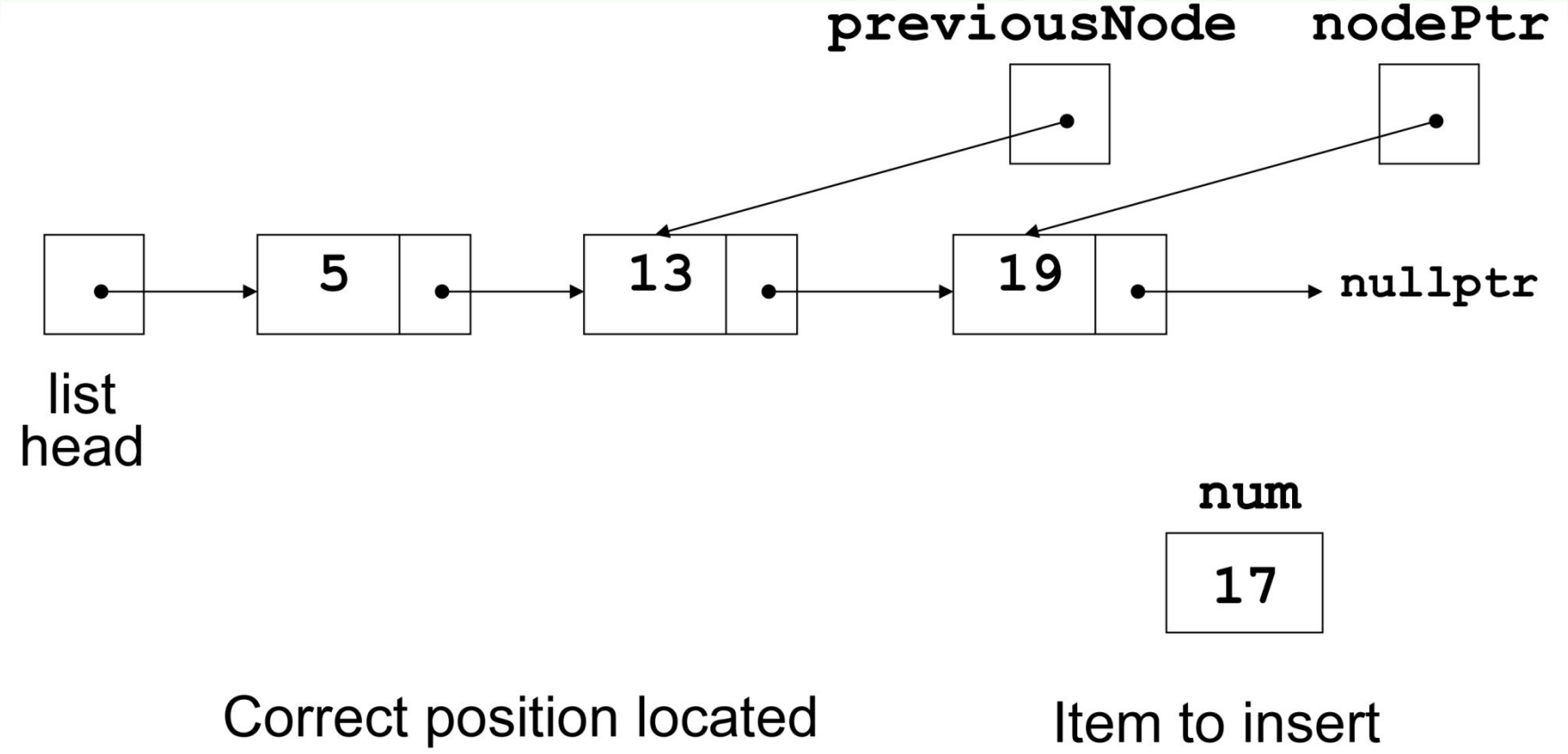
Adding a Node at the Head of a List

- Test to see if
  - head pointer is **`nullptr`**, or
  - node value pointed at by head is greater than value to be inserted
- You must test in this order: the results are unpredictable if the second test is attempted on an empty list
- Create new node, set its next pointer to head, then point head to it

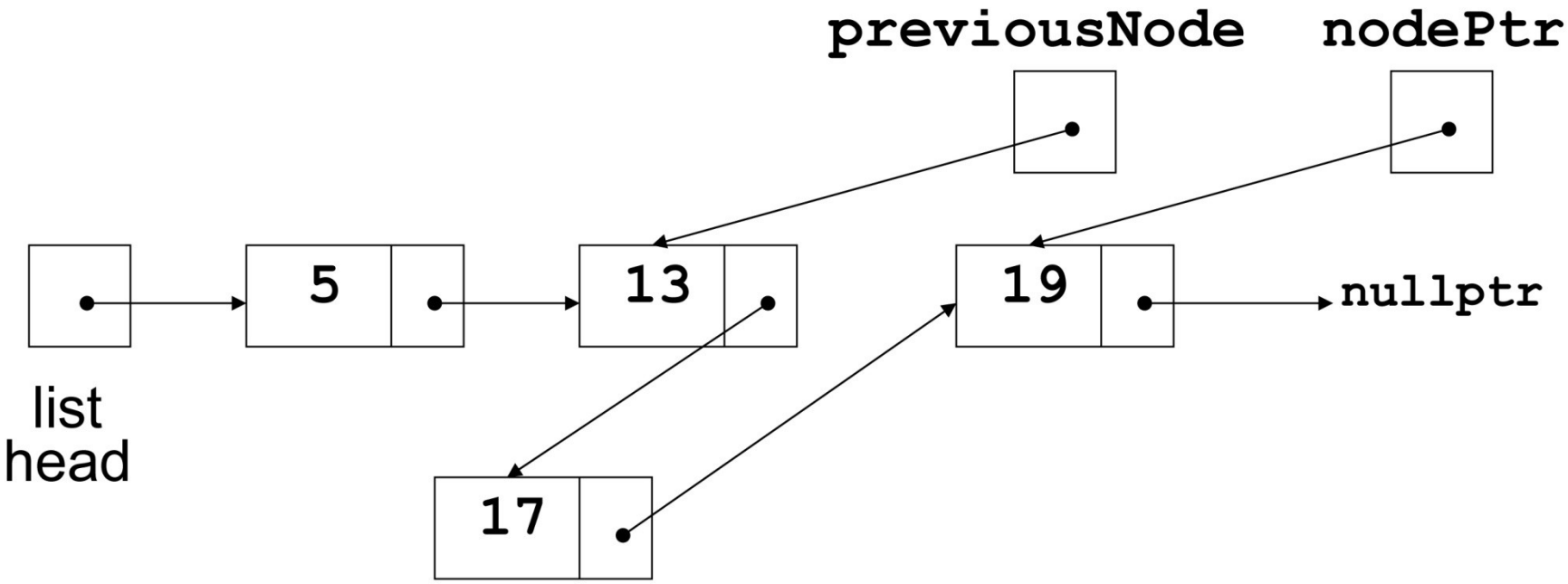
Inserting a Node after Head in a List

- This requires two pointers to traverse the list:
  - a pointer to locate the node with data value greater than that of node to be inserted, or to locate the end of the list
  - a pointer to 'trail behind' one node, to point to node before the point of insertion
- The new node is inserted between the nodes pointed at by these pointers

Inserting a Node into a Linked List 1 of 2



Inserting a Node into a Linked List 2 of 2



New node created and inserted in order in the linked list

Destroying a Linked List

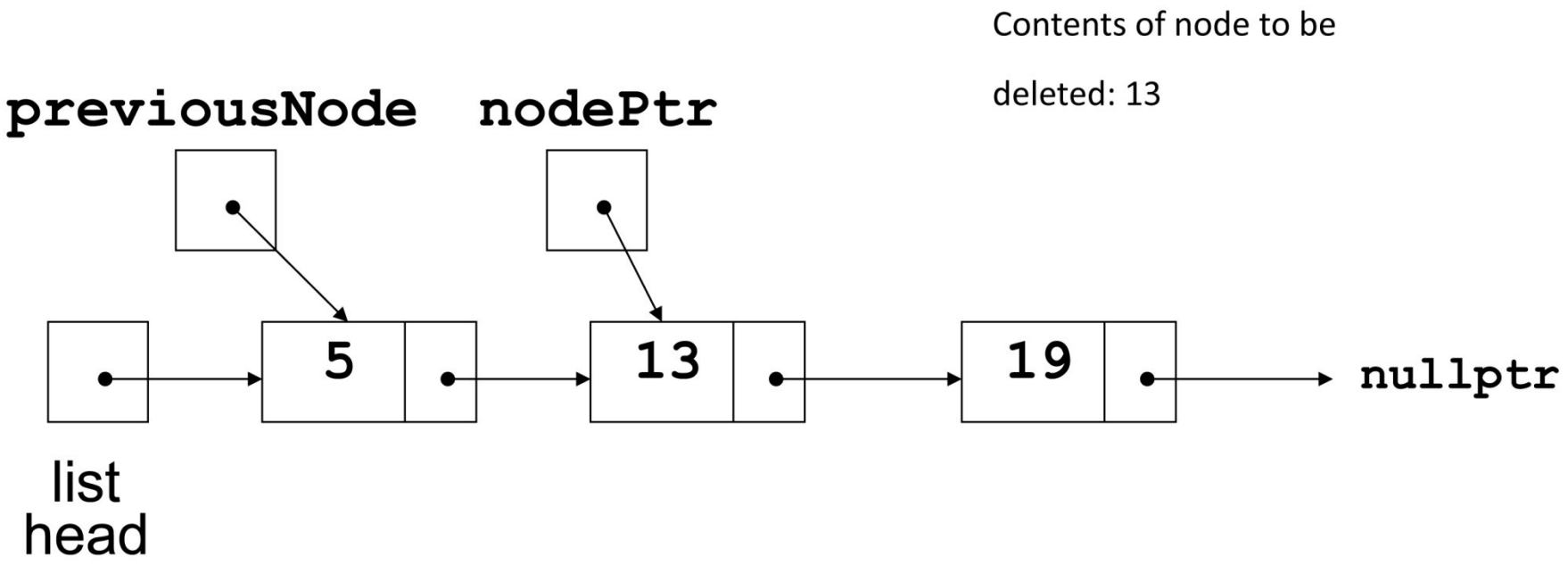
- Done using the destructor for the linked list class.
- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - Free the node's memory
- Finally, set the list head to **`nullptr`**

Removing an Element

1) Find the node to be removed
2) Remove it from the list
3) Delete the memory associated with the node

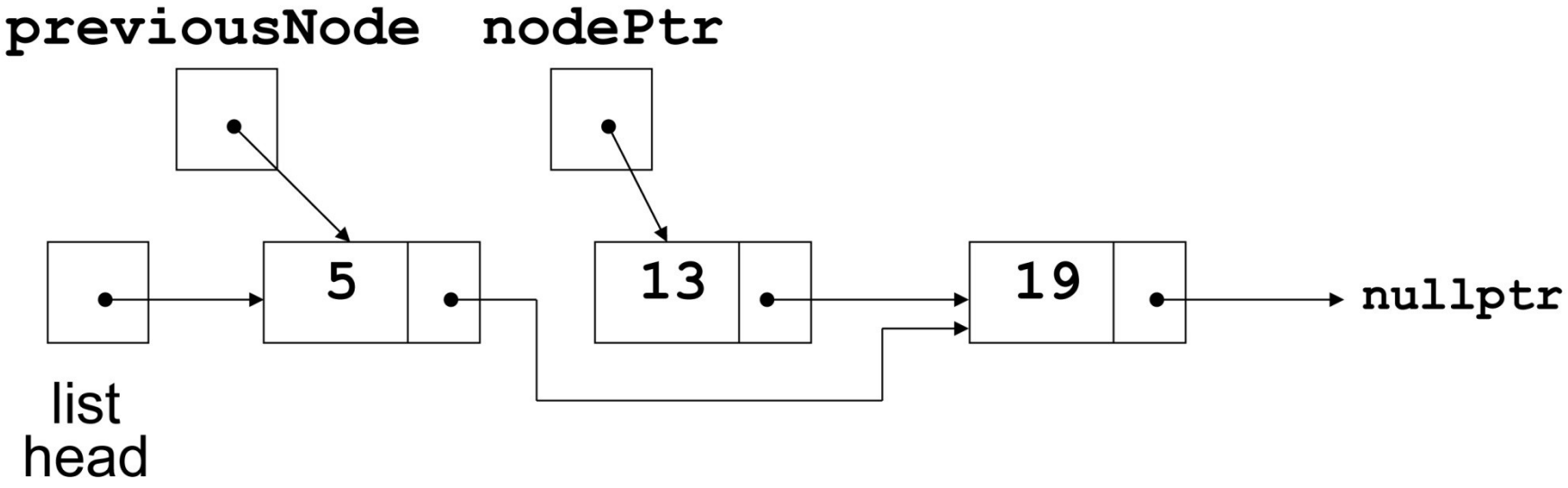
This requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Deleting a Node 1 of 3



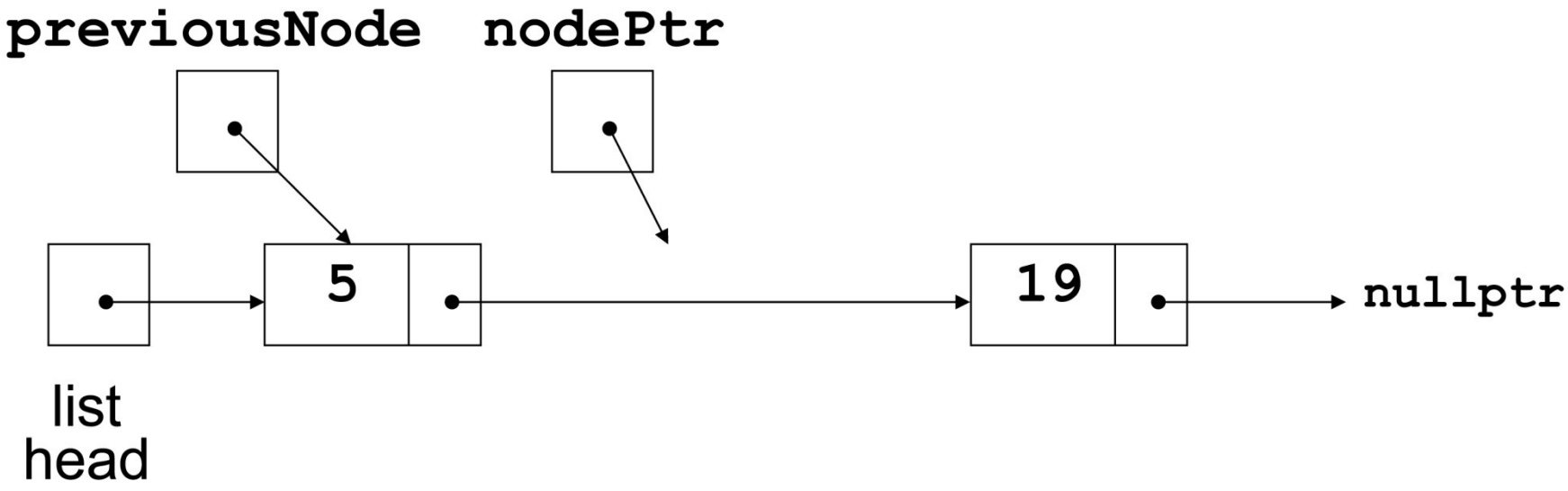
Locating the node containing 13

Deleting a Node 2 of 3



Adjusting pointer around the node to be deleted

Deleting a Node 3 of 3



Linked list after deleting the node containing **13**

18.3 A Linked List Template

- A linked list template can be written by replacing the type of the data in the node with a type parameter, say T.
- If you are defining the linked list as a class template, then all member functions must be function templates
- Implementation assumes use with data types that support comparison: == and <=

18.4 Recursive Linked List Operations

- A non-empty linked list consists of a head node followed by the rest of the nodes
- The rest of the nodes form a linked list that is called the tail of the original list
- Many linked list operations can be broken down into the smaller problems by processing the head of the list and then recursively operating on the tail of the list

Find the Size of a Linked List 1 of 2

To find the size (number of elements) of a list

- If the list is empty, the length is 0 (base case)
- If the list is not empty, find the length of the tail and then add 1 to obtain the length of the original list

Find the Size of a Linked List 2 of 2

To find the size of a list:

```
int size(ListNode *myList)
{
   if (myList == NULL) return 0;
   else
   return 1 + size(myList->next);
}
```

Display the Contents of a Linked List

Using recursion to display a list:

```
void displayList(ListNode *myList)
{
   if (myList != NULL)
   {
      cout << myList->data << " ";
      displayList(myList->next);
   }
}
```

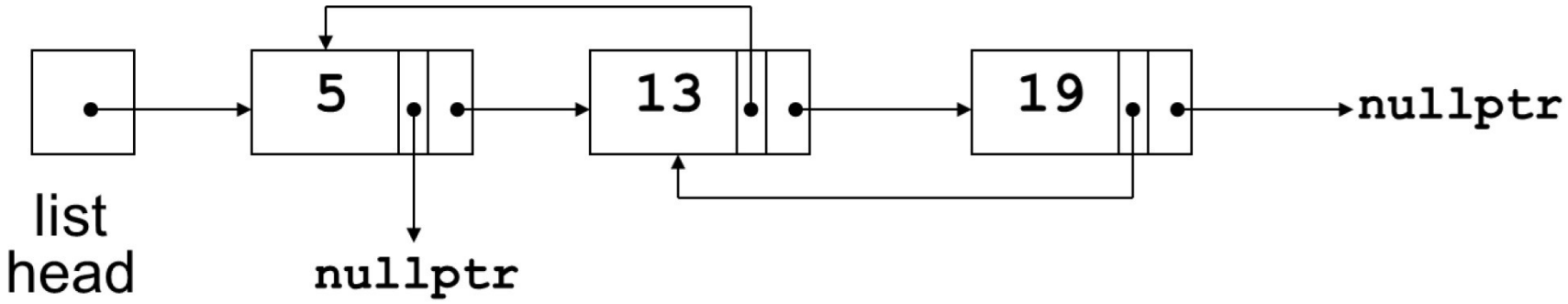
Other Recursive Linked List Operations

- Add and remove operations can be written to use recursion
- General design considerations:
  - The base case is often when the list is empty
  - The recursive case often involves the use of the tail of the list (*i.e.,* the list without the head). Since the tail has one fewer entry than the list that was passed in to this call, the recursion eventually stops.

18.5 Variations of the Linked List 1 of 2

Other linked list organizations:

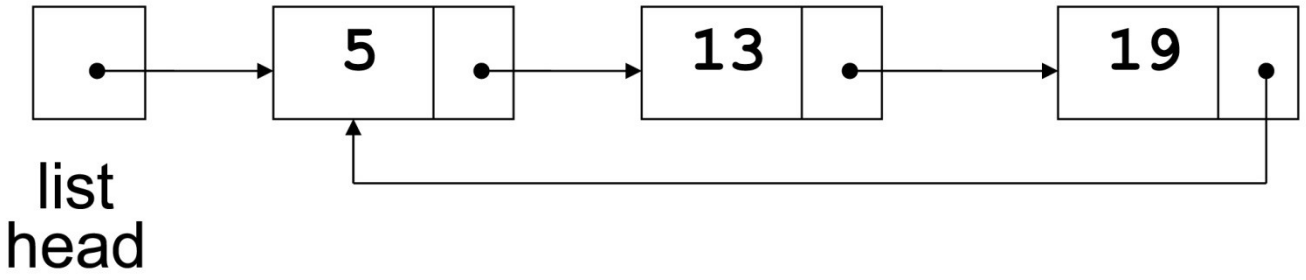
- doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



Variations of the Linked List 2 of 2

Other linked list organizations:

- circular linked list: This is a singly-linked list in which the last node in the list points back to the first node in the list, not to **`nullptr`**



Smart Pointers and Linked Lists

- Smart pointers are generally helpful in preventing common problems such as memory leaks, inadvertent deletion, and multiple deletion.
- They may not be useful in implementing linked lists:
  - Unique pointers cannot be used due to ownership
  - Shared pointers add the overhead of shared groups
  - Shared pointers in a circularly-linked list may lead to memory leaks when trying to destroy a list

18.6 The STL `list` and `forward_list` Containers

Sequence containers in the STL with underlying linked lists in their implementation

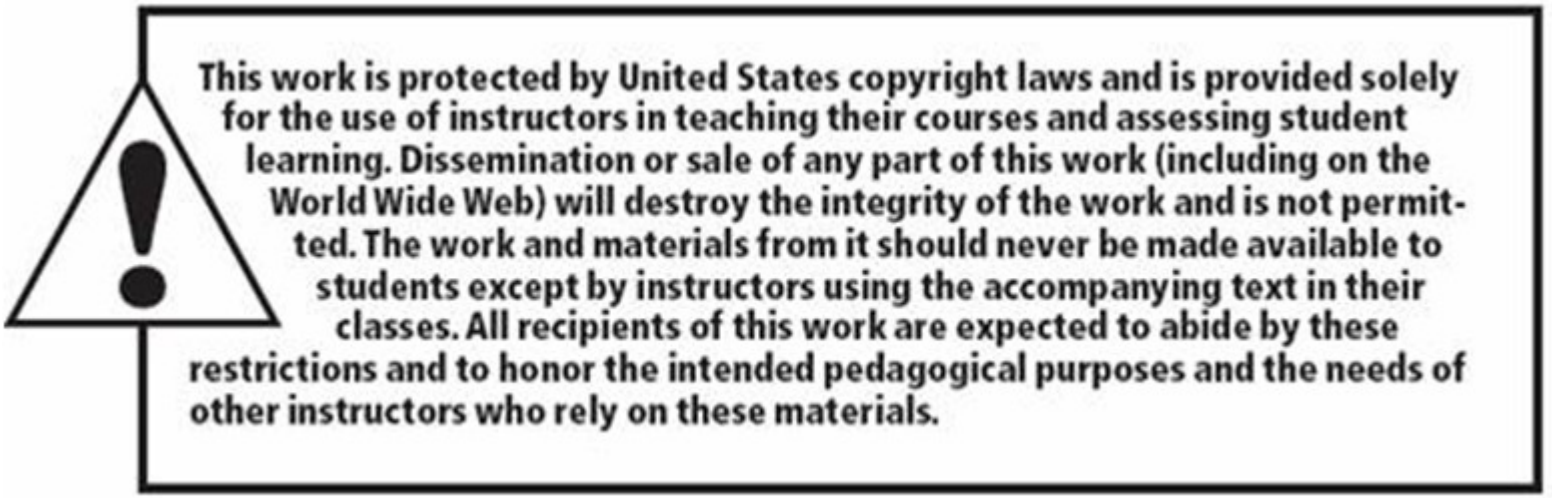
- **`list`**: uses and underlying doubly-linked list
- **`forward_list`**: uses a singly-linked list

Unlike other sequence containers, cannot go directly to an element via an index. Access is sequential from the first to the last element.

list and forward_list Containers

- Constructors: default, 2 fill constructors (fill with zero or with a given value), copy constructor, range constructor.
- Lots of member methods, many already seen in Chapter 17.
- Use online references to learn about the small differences in member methods between the two containers.

Copyright



Pearson