CS256

Chapter 17: Linked List

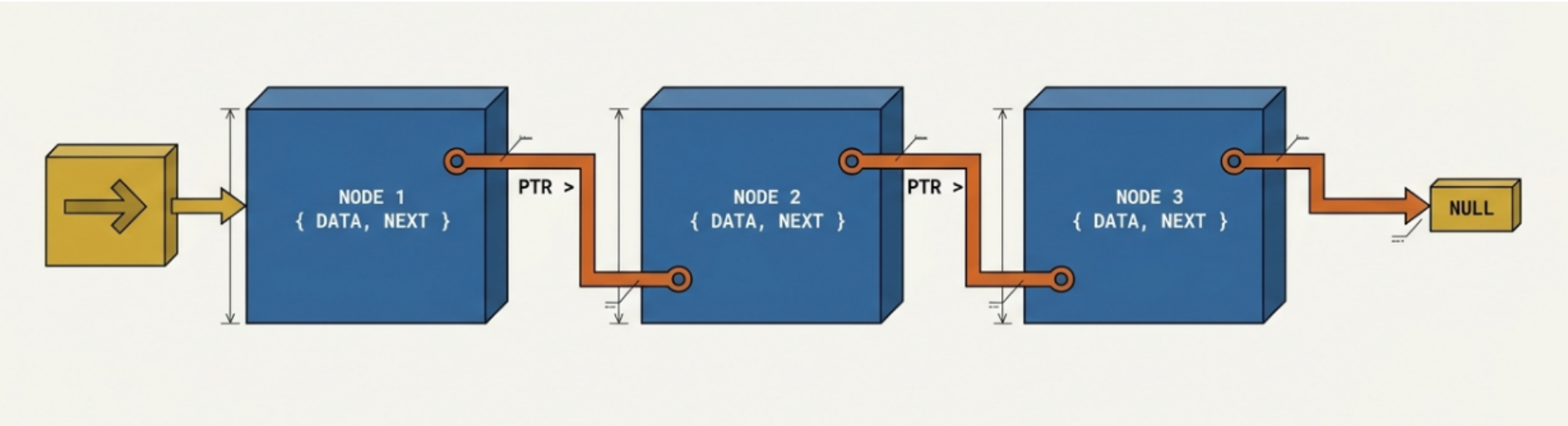
Hanoi University of Science and Technology

Slide contents are mainly based on the following documents:

- Tony Gaddis, Starting out with C++, Eighth Edition, Pearson, 2015
- Tony Gaddis, Power Points Slides:
  `https://www.umsl.edu/~antognolij/gaddis.html`

Linked List

Linked List: set of data structures (nodes) that contain references to other data structures



Data structures (nodes) can be **added to or removed** from the linked list during execution

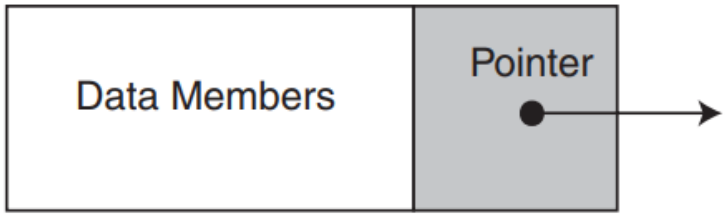
Linked List, Array, Vector

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily

Node Organization

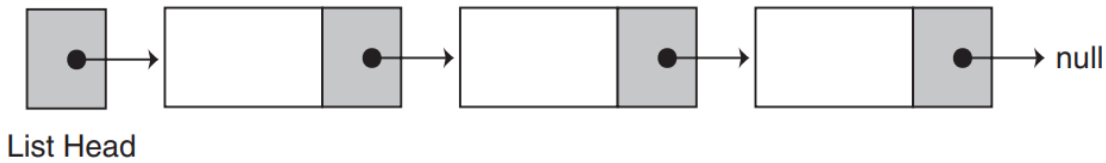
A node contains:

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



Linked List Organization

- Linked list contains 0 or more nodes
- Has a list head to point to first node
- Last node points to `NULL`



- Empty list:
  - If a list currently contains 0 nodes, it is the empty list
  - In this case the list head points to `NULL`

Declaring a Node

```
struct ListNode
{
    double value;             // The value in this node
    struct ListNode *next;    // To point to the next node
};
```

Defining a Linked List

- Define a pointer for the head of the list

```
ListNode *head = NULL;
```

- Head pointer initialized to NULL to indicate an empty list
- NULL Pointer:
  - Should always be tested for before using a pointer:

```
ListNode *p;
while (p != NULL) ...
```

  - Can also test the pointer itself:

```
while (p) ... // same meaning as above
```

Traverse a List

```
Assign List head to node pointer.
While node pointer is not null
    Display the value member of the node pointed to by node pointer
    Assign node pointer to its own next member.
End While.
```

Traverse a List

```
ListNode *nodePtr; // To move through the list

// Position nodePtr at the head of the list.
nodePtr = head;

// While nodePtr points to a node, traverse
// the list.
while (nodePtr)
{
    // Display the value in this node.
    cout << nodePtr->value << endl;

    // Move to the next node.
    nodePtr = nodePtr->next;
}
```

Appending a Node

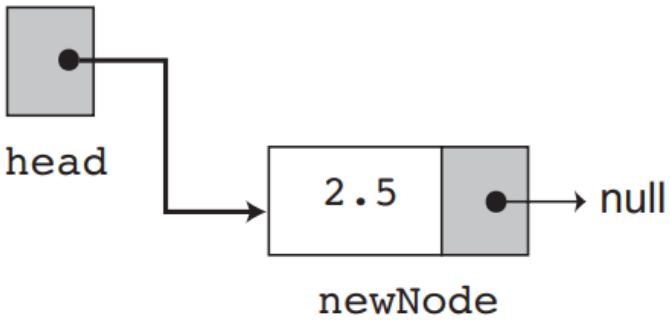
Appending a Node: Add a node to the end of the list

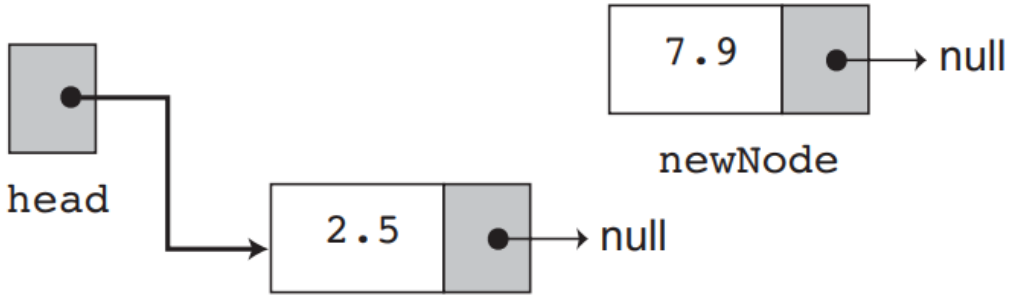
```
Create a new node.
Store data in the new node.
If there are no nodes in the list
    Make the new node the first node.
Else
    Traverse the list to find the last node.
    Add the new node to the end of the list.
End If.
```

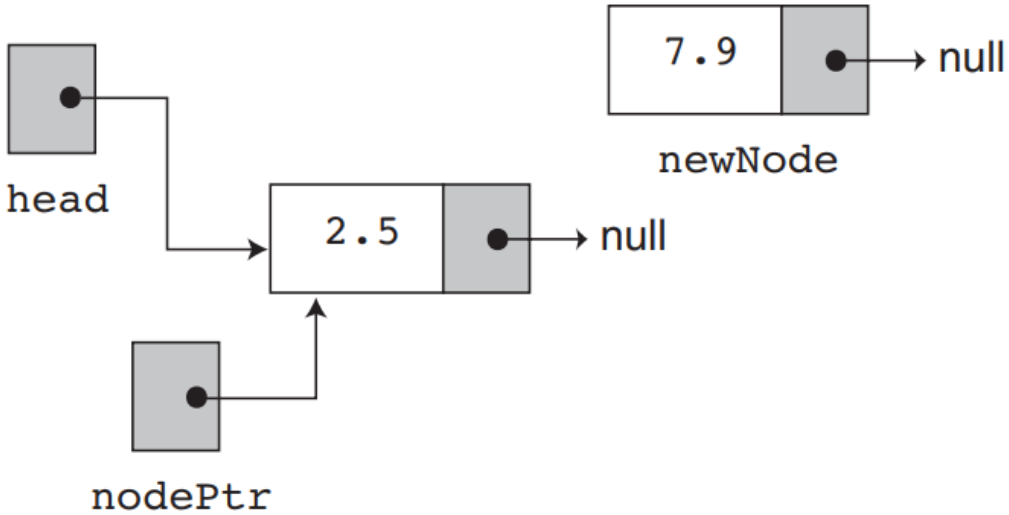
Appending a List

```
ListNode *newNode;  // To point to a new node
ListNode *nodePtr;  // To move through the list

// Allocate a new node and store num there.
newNode = new ListNode;
newNode->value = num;
newNode->next = nullptr;

// If there are no nodes in the list
if (!head) // make newNode the first node.
    head = newNode;
else // Otherwise, insert newNode at end.
{
    nodePtr = head; // Initialize nodePtr to head of list.

    while (nodePtr->next)  // Find the last node in the list.
        nodePtr = nodePtr->next;

    nodePtr->next = newNode; // Insert newNode as the last node
}
```

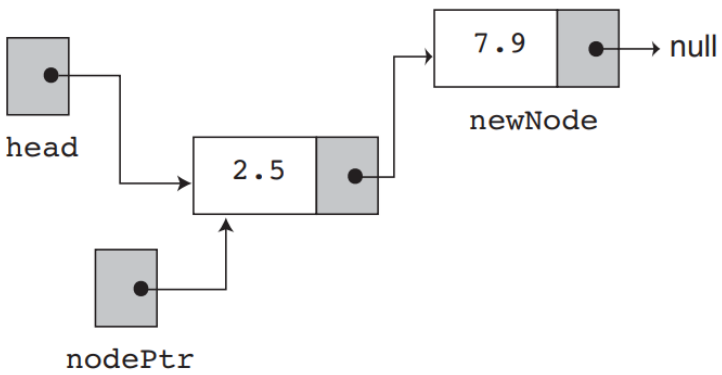
Appending a Node

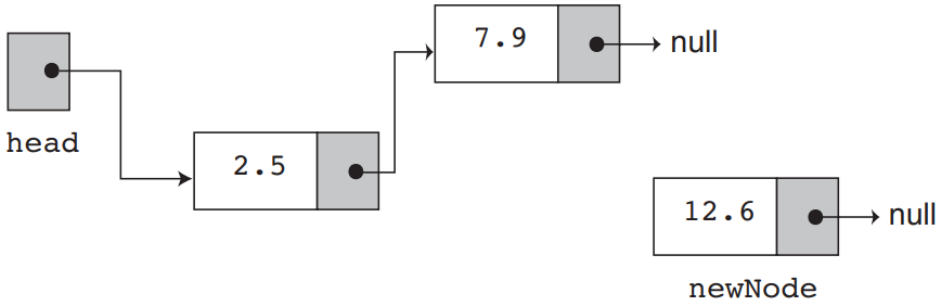


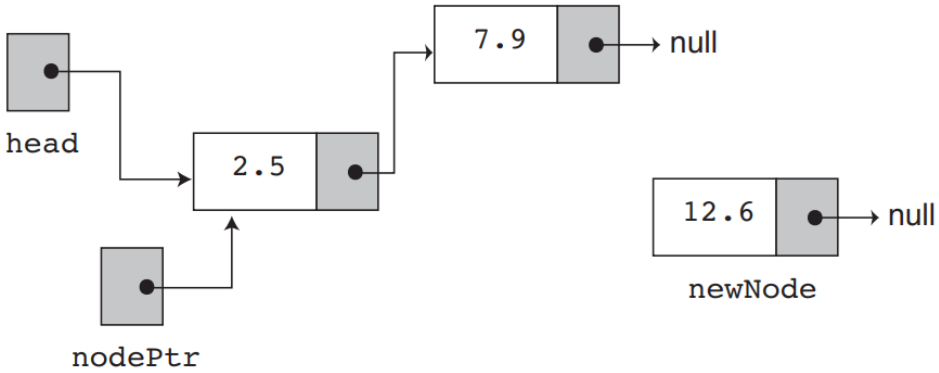




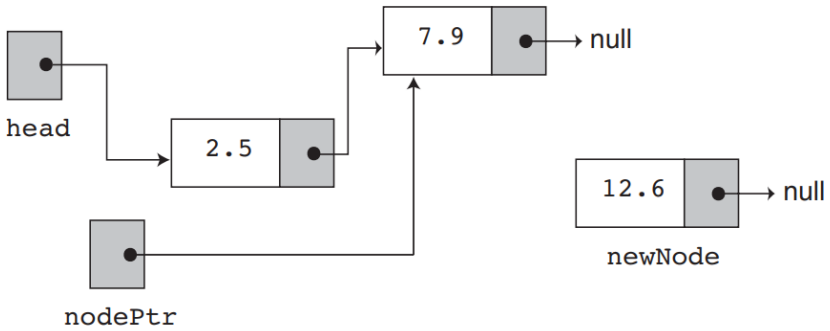
Appending a Node

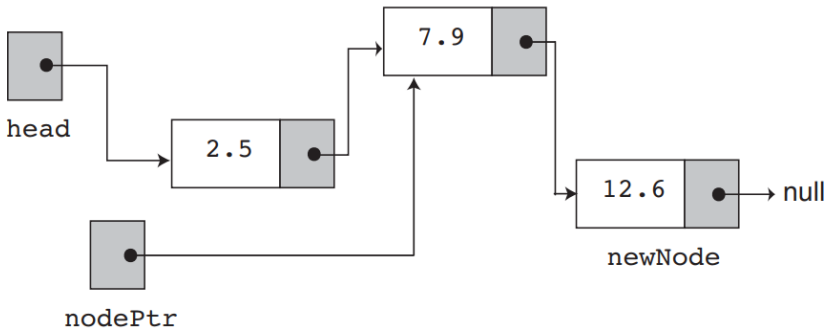






Appending a Node





Inserting a node

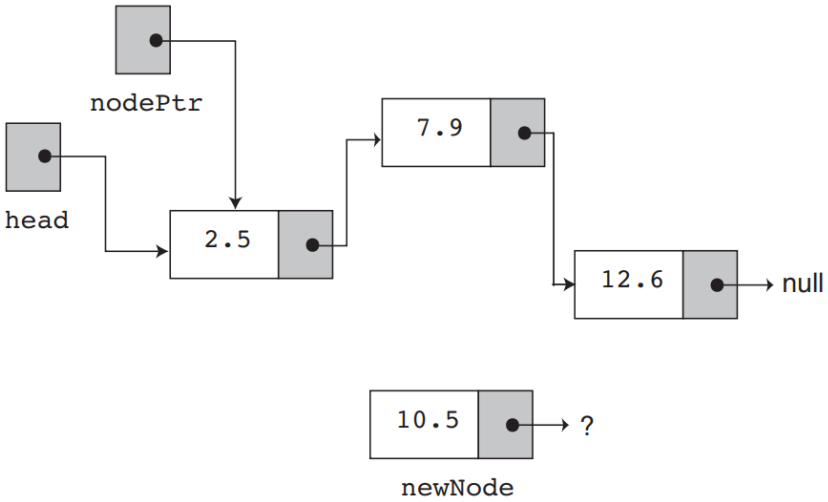
Assumption: the nodes in the list are already in order

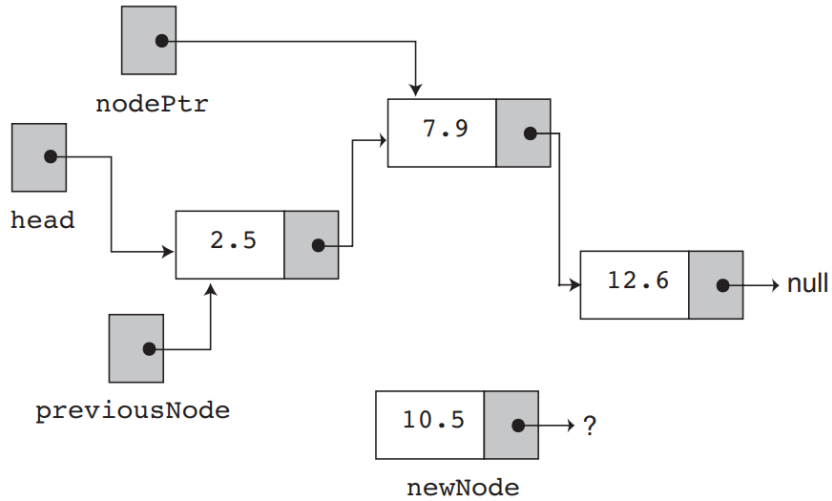
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

Inserting a node

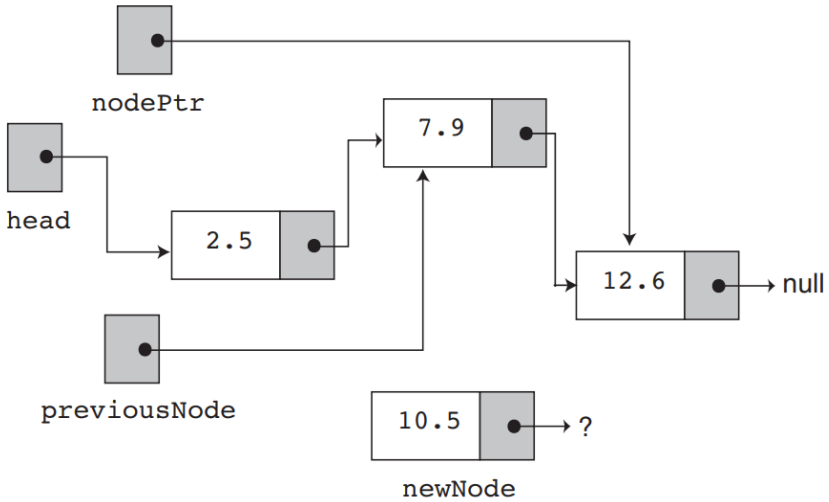
```
Create a new node.
Store data in the new node.
If there are no nodes in the list
    Make the new node the first node.
Else
    Find the first node whose value is greater than or equal to the new
        value, or the end of the list (whichever is first).
    Insert the new node before the found node, or at the end of the list
        if no such node was found.
End If.
```

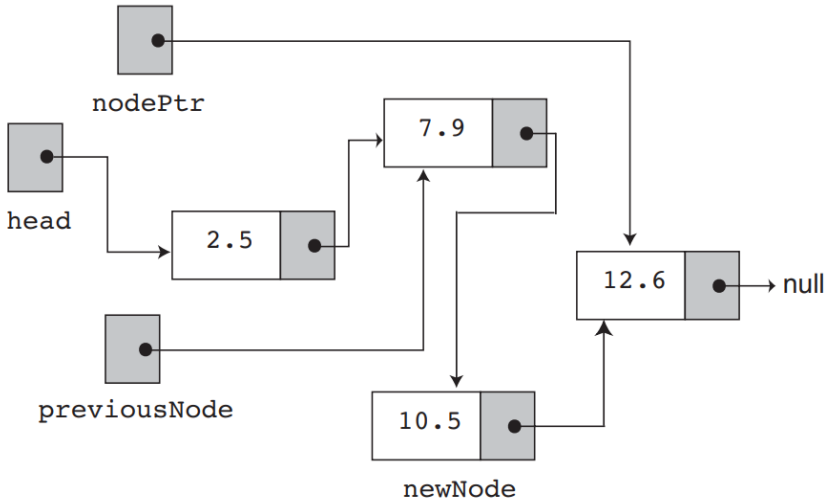
Inserting a node





Inserting a node

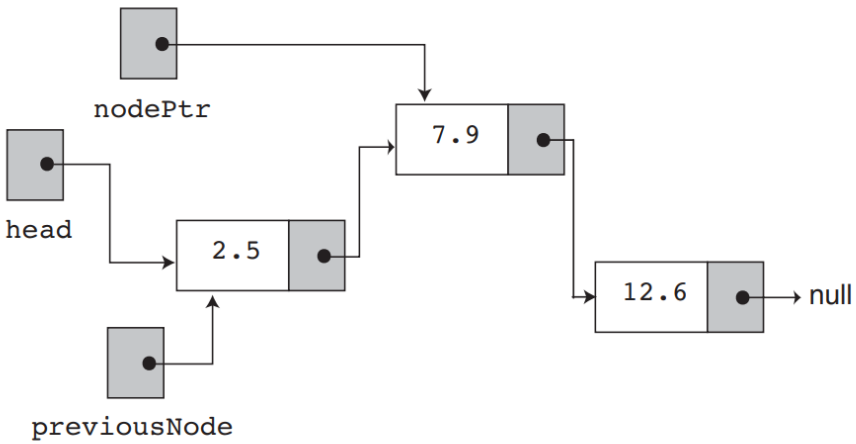


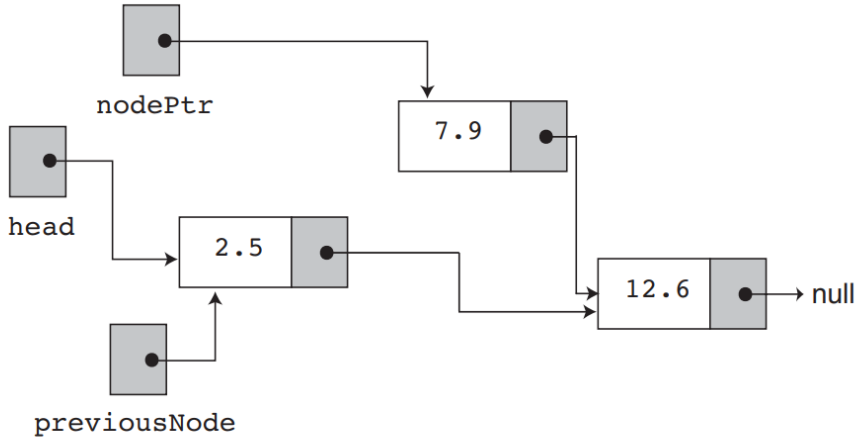


Deleting a Node

- If list uses dynamic memory, then delete node from memory
- 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





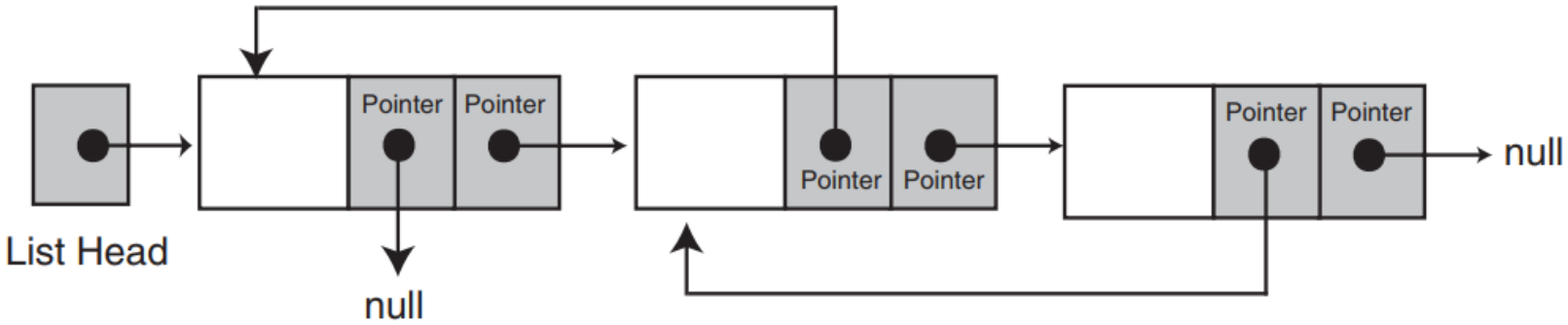
Destroying a Linked List

Must remove all nodes used in the list:

1. To do this, use list traversal to visit each node
2. For each node,
   – Unlink the node from the list
   – If the list uses dynamic memory, then free the node's memory
3. Set the list head to NULL

Variations of the Linked List

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



- circular linked list: the last node in the list points back to the first node in the list, not to NULL

s

The STL `list` Container

- `#include <list>`
- See Table 17.1

Exercises

Challenging problems: 4, 5, 6 + 7.
Additional exercises:

- Merge two sorted linked lists.
- Delete all occurrences of a given key in a linked list
- Given a linked list and positions m and n, reverse the linked list from position m to n
- Delete N nodes after M nodes of a linked list