Linked Lists

CSCI-2270

Elizabeth Boese

Linked List ADT

ADT

Linked List

- Series of connected *nodes*
- Each node is a data structure
- Can grow or shrink in size as the program runs

List ADT

- Objects/data
  - A_0, A_1, A_2, … A_{N-1}
  - Size of the List is N
- Operations
  - Up to the designer of a List, for example,
  - `printList()`
  - `makeEmpty()`
  - `Find()`
  - `Insert()`
  - `Remove()`
  - `findKth()`
  - `etc`

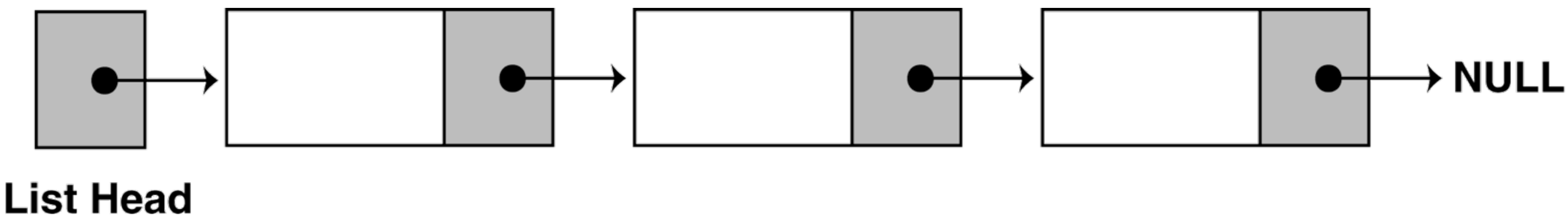
Linked List

Nodes

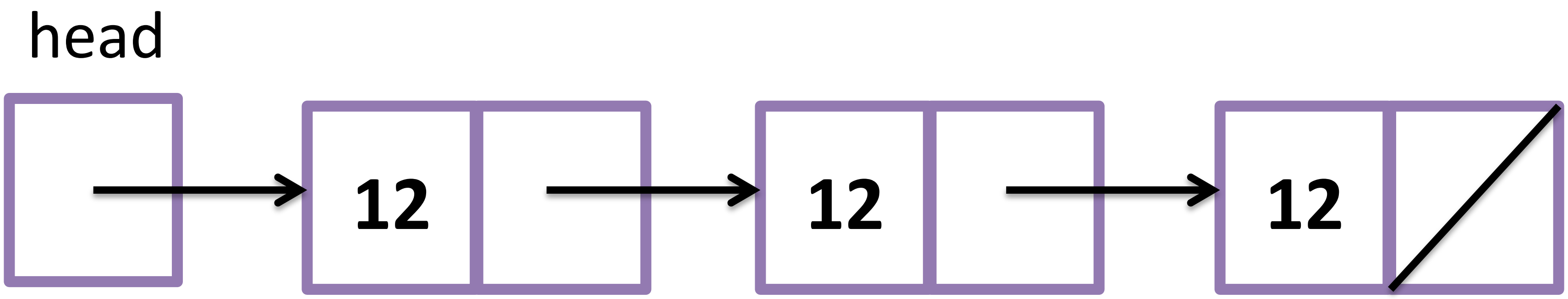
- Data
- Pointer

Linked List

- Called "linked" because each node in the series has a pointer that points to the next node in the list.



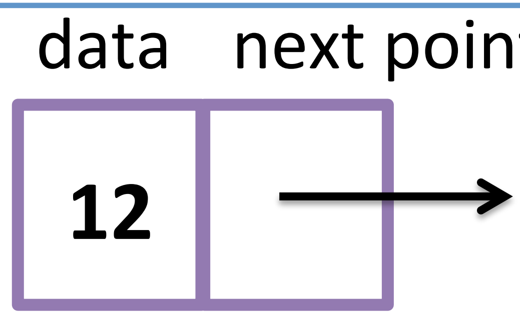
Linked List



Declarations

- Declare a data structure that will be used for the nodes.

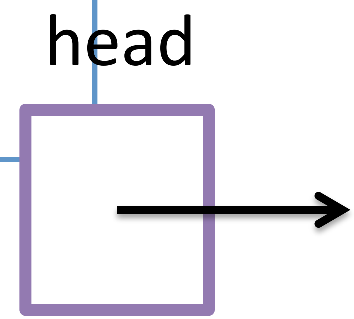
```
struct ListNode
{
      float value;
      struct ListNode *next;
};
```



Declarations

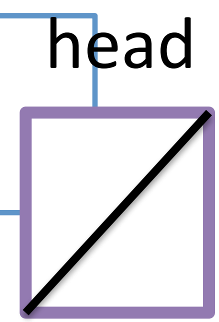
- Declare a pointer to serve as the list head

```
ListNode *head;
```



- To designate an empty list,

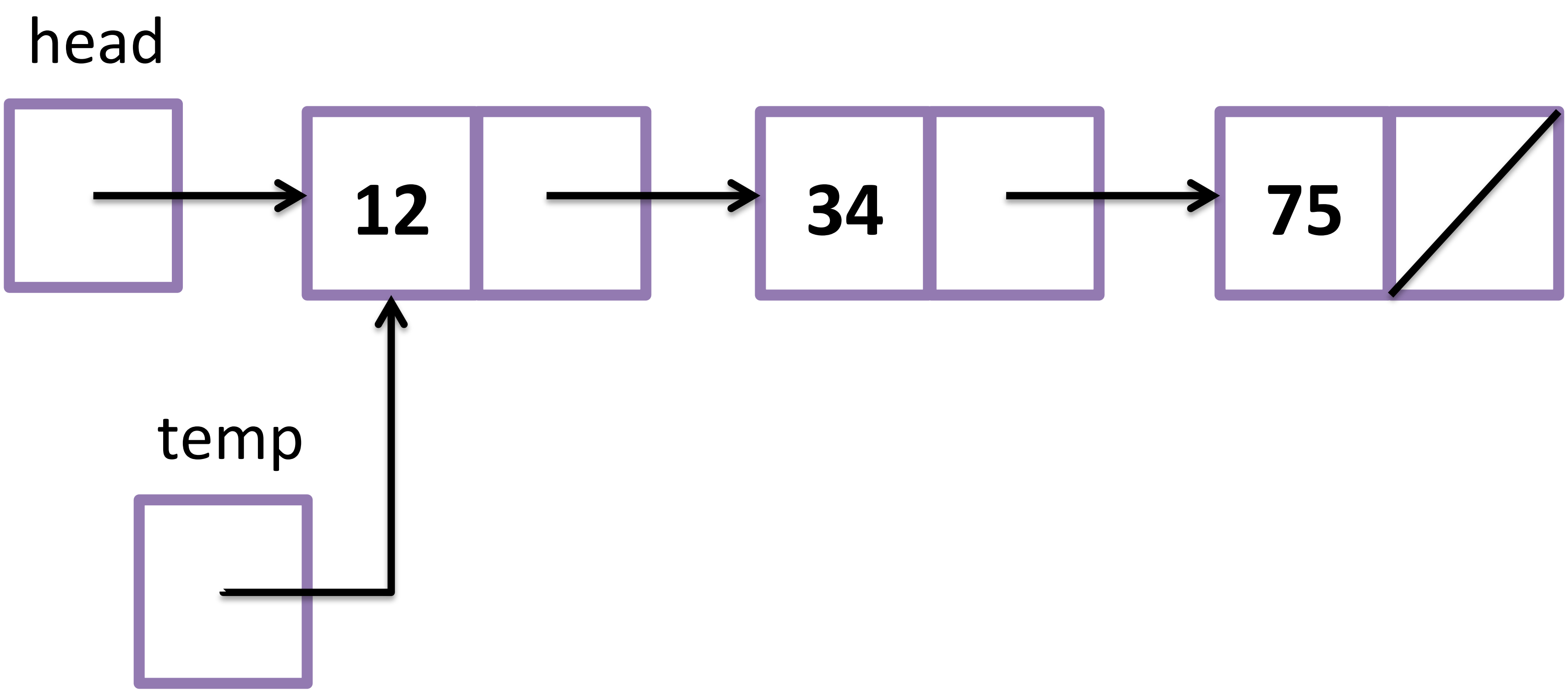
```
head = NULL;
```



Walking through the List

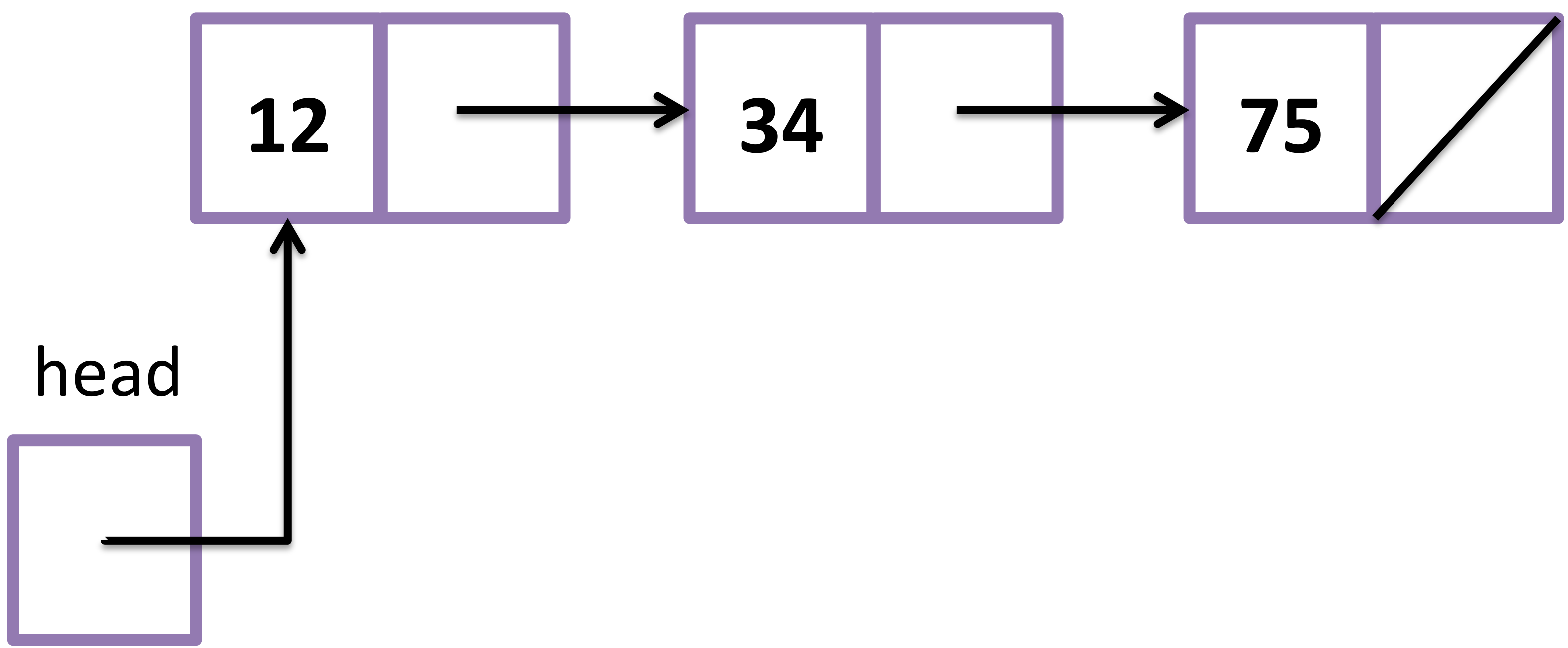
Walking through the List

- Whenever you traverse a linked list, you need a temporary pointer.



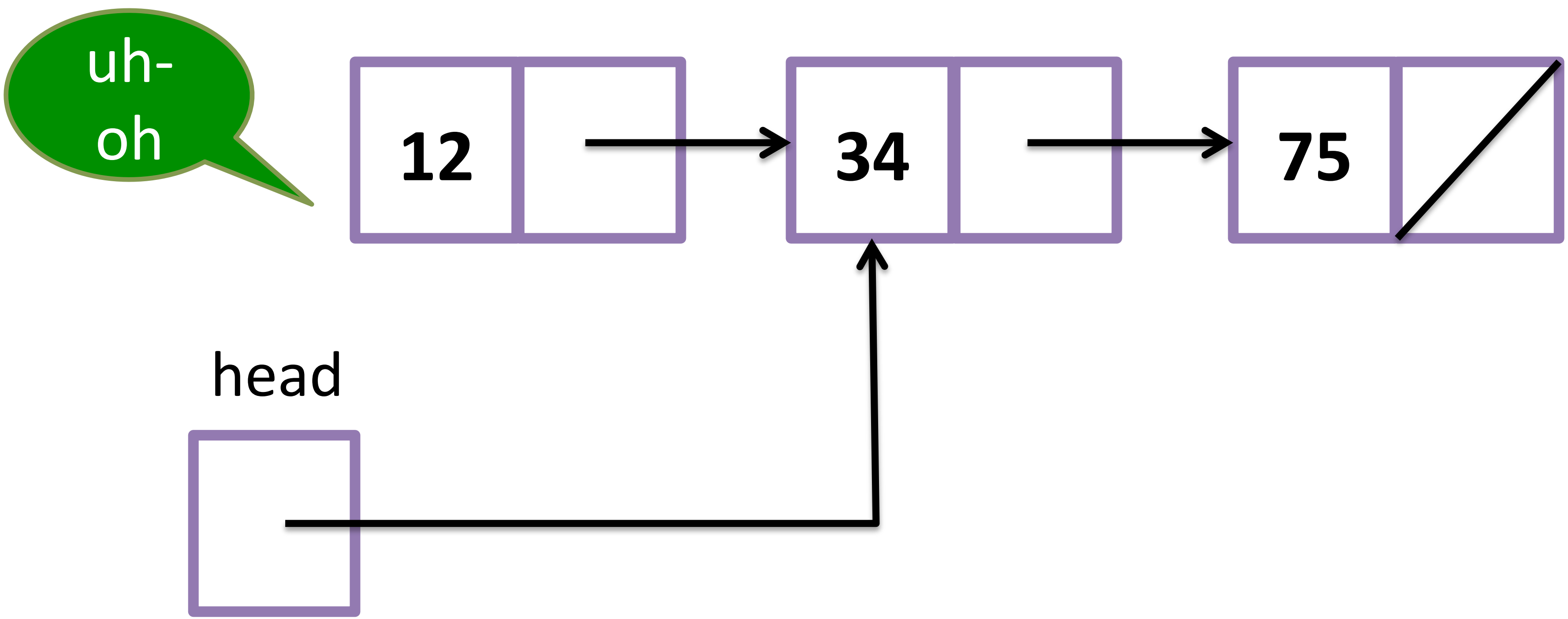
Walking through the List

- Whenever you traverse a linked list, you need a temporary pointer.
  Never use the head pointer or you will lose access to your nodes!



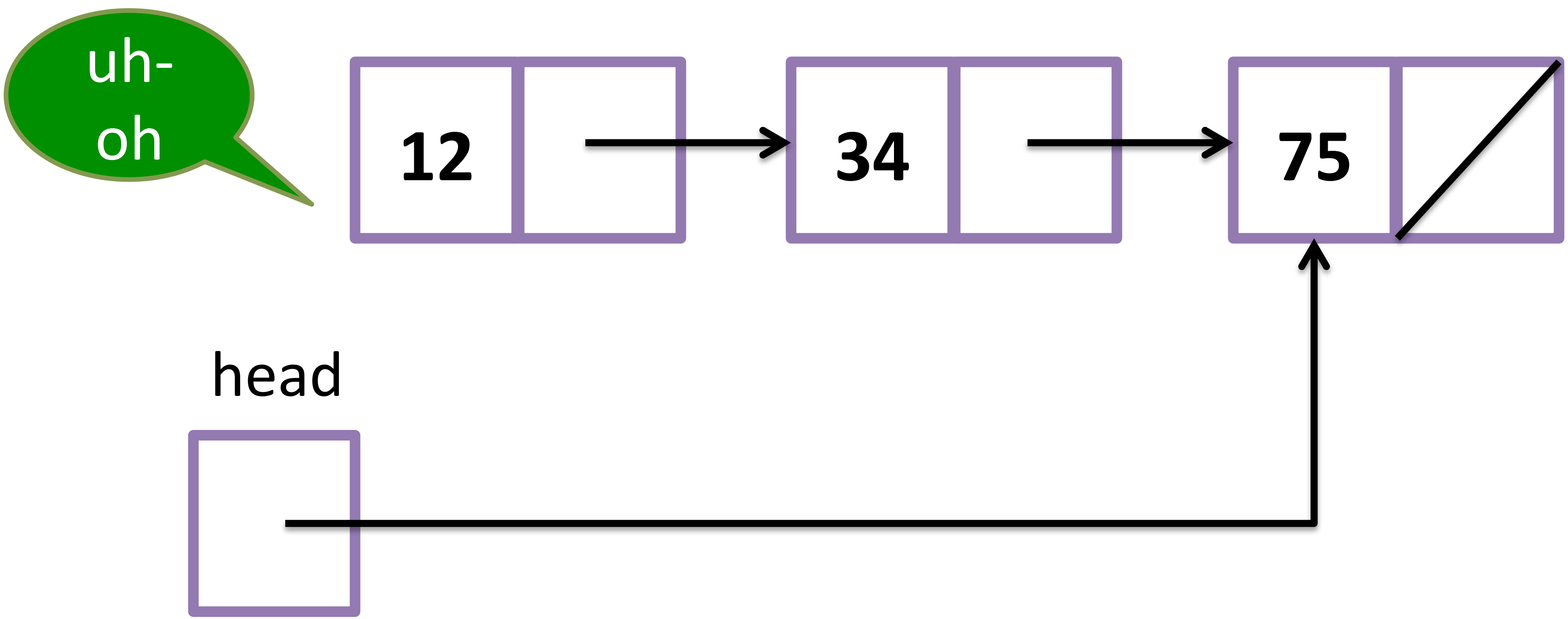
Walking through the List

- Whenever you traverse a linked list, you need a temporary pointer.
  Never use the head pointer or you will lose access to your nodes!



Walking through the List

- Whenever you traverse a linked list, you need a temporary pointer.
  Never use the head pointer or you will lose access to your nodes!



Walk through the List

Assign node pointer to the list head

While node pointer is not NULL

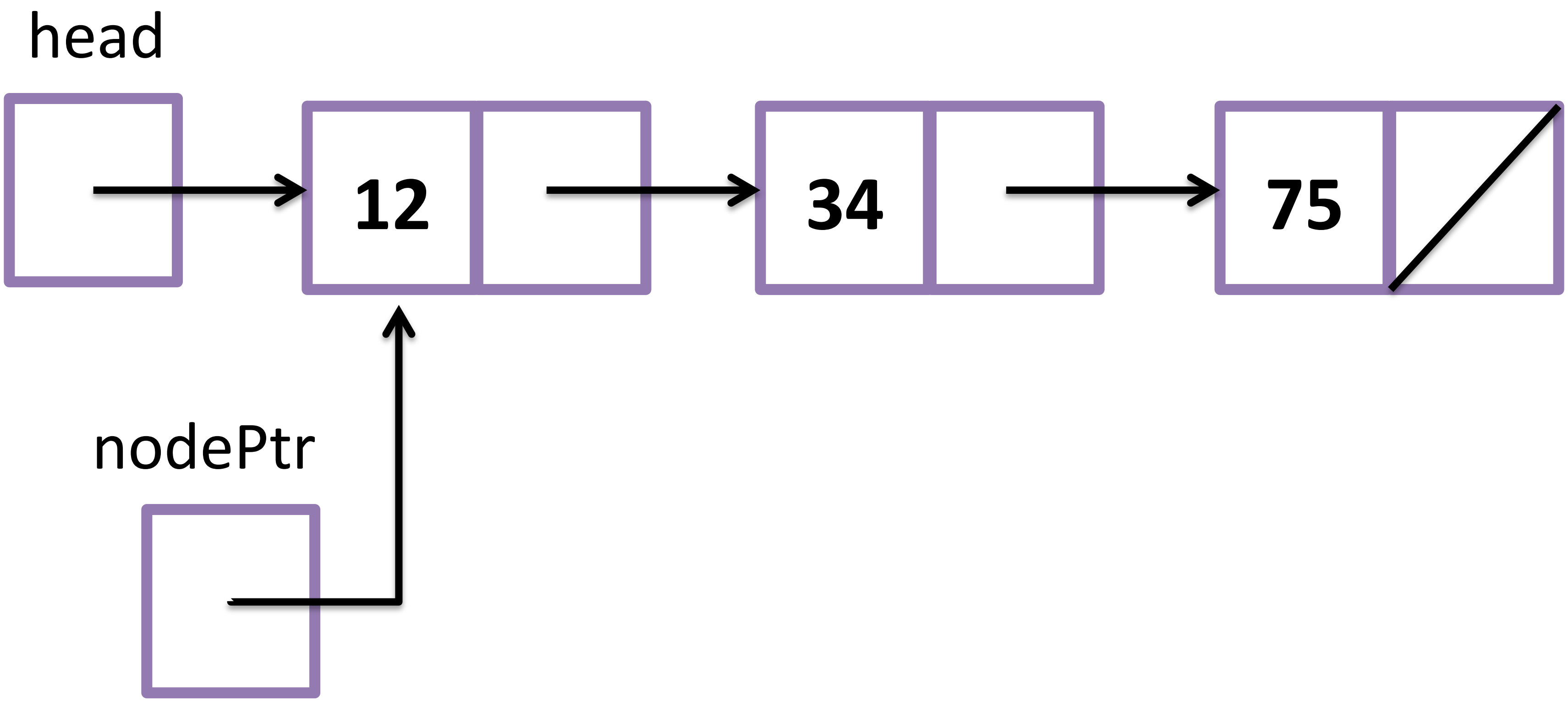
Display the `value` *member of the node*

pointed to by node pointer.

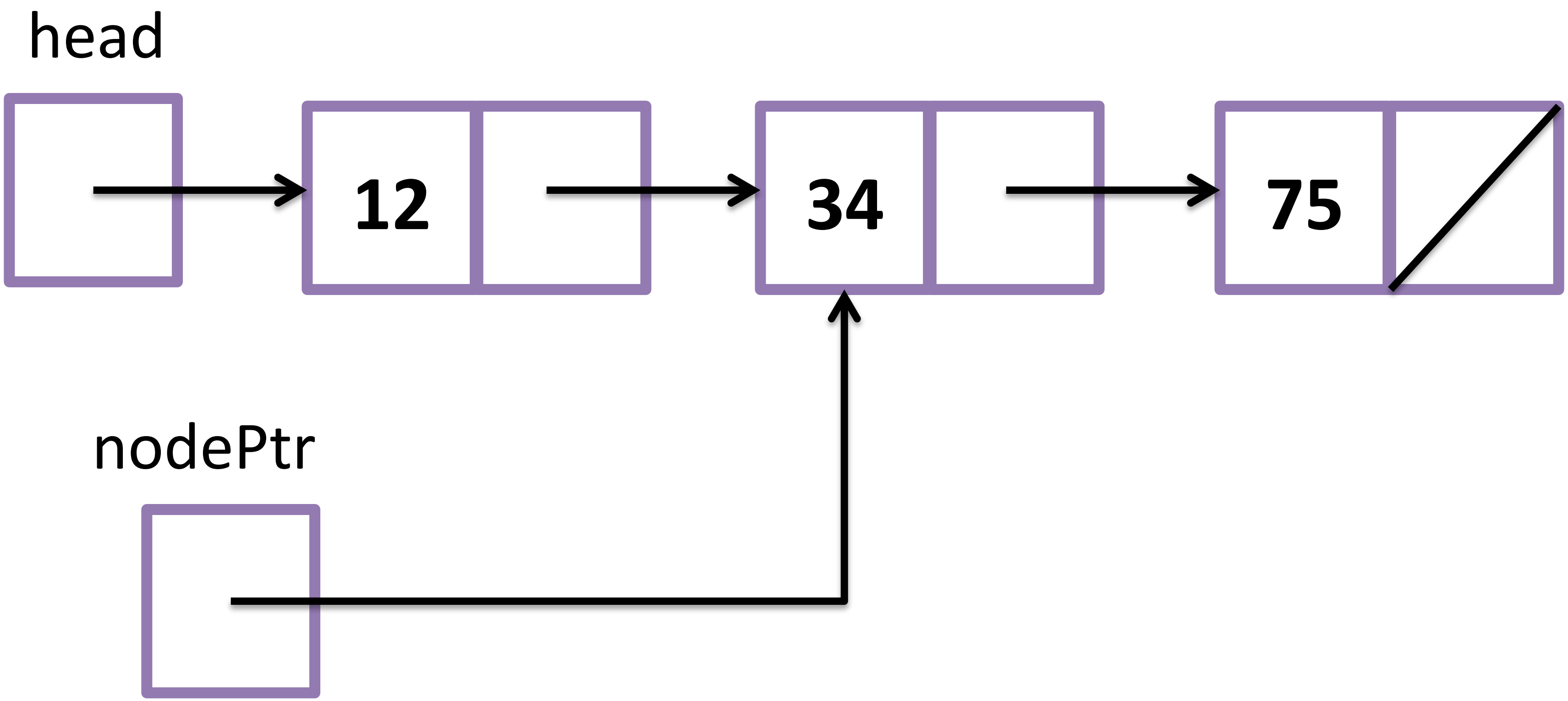
Assign node pointer to its own next node member

End While

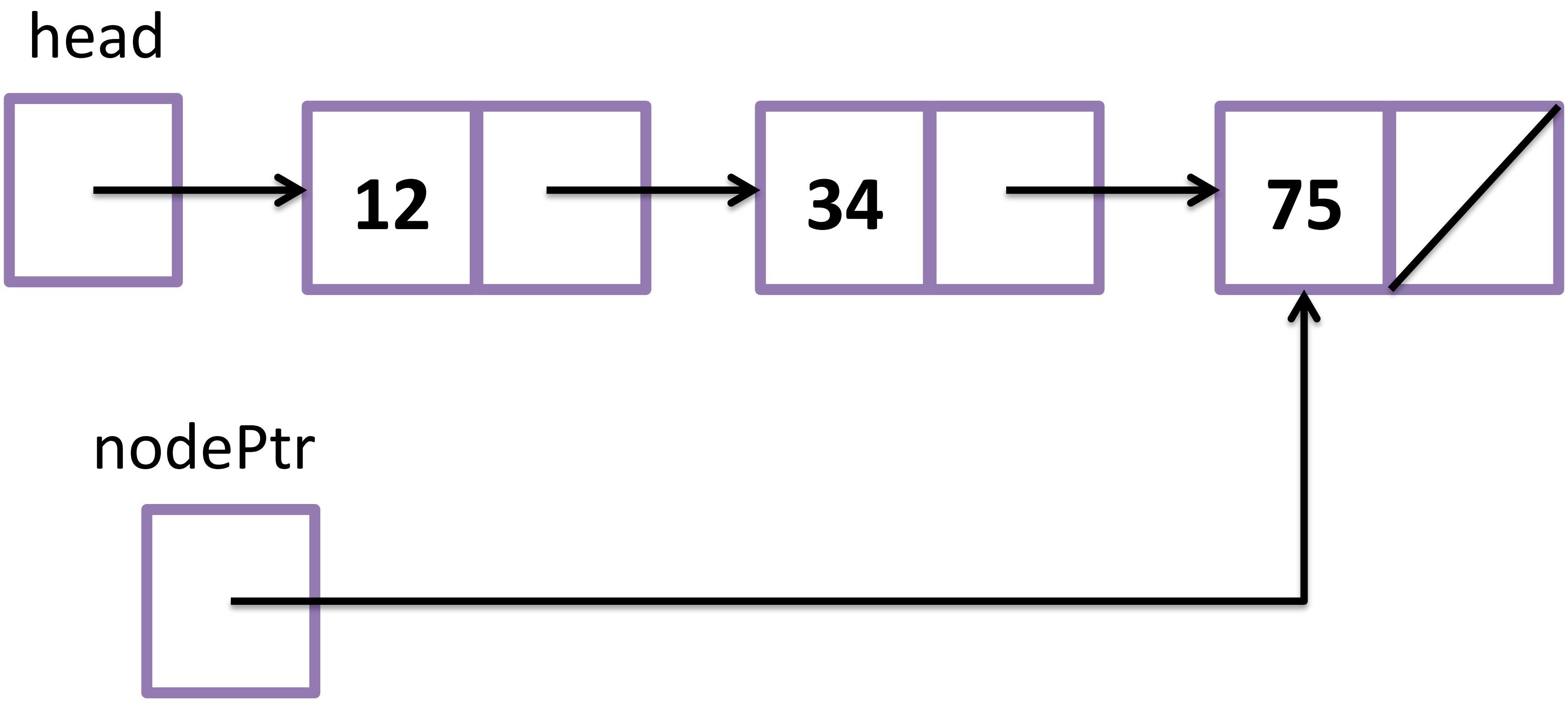
Walk through the List



Walk through the List



Walk through the List



Appending a Node to the List

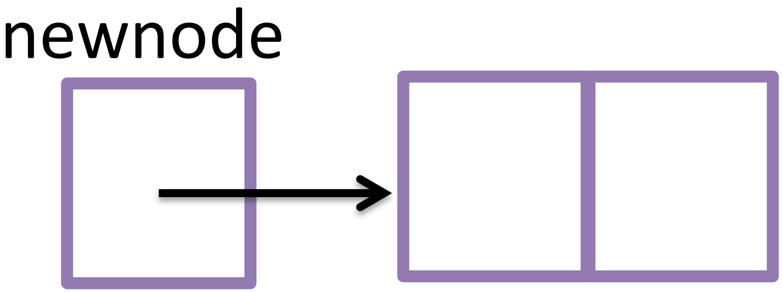
Appending a Node to the List

Append = add the node to the end of the list

Create a new node.
Store data in the new node.
Set next pointer in new node to NULL
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

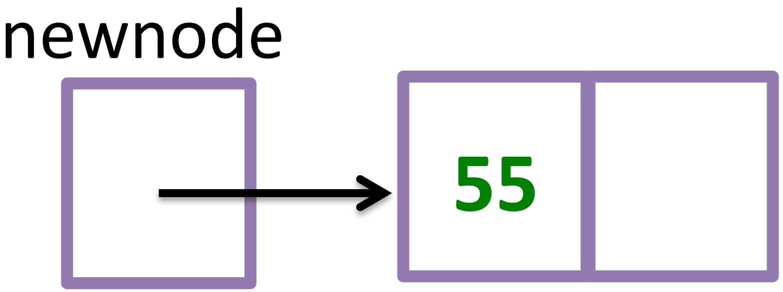
Appending a Node to the List

Create a new node.



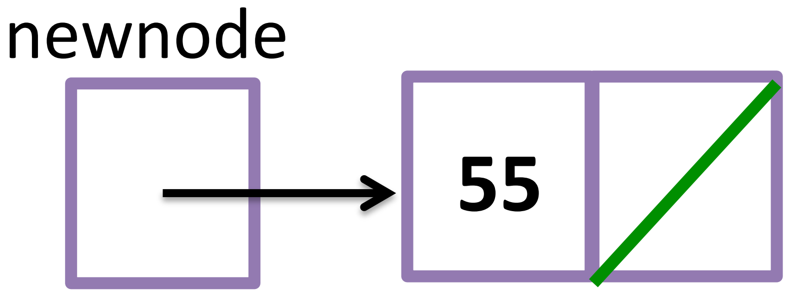
Appending a Node to the List

Store data in the new node.



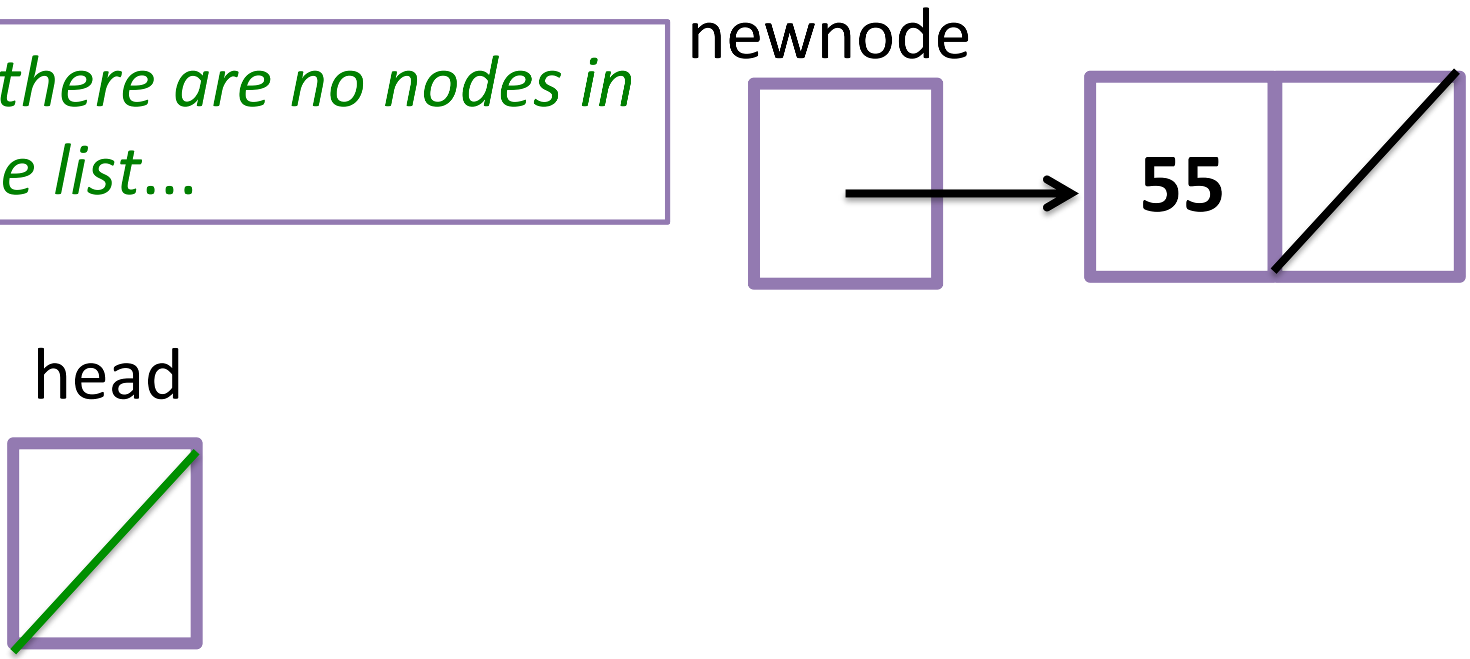
Appending a Node to the List

Set next pointer in new node to NULL



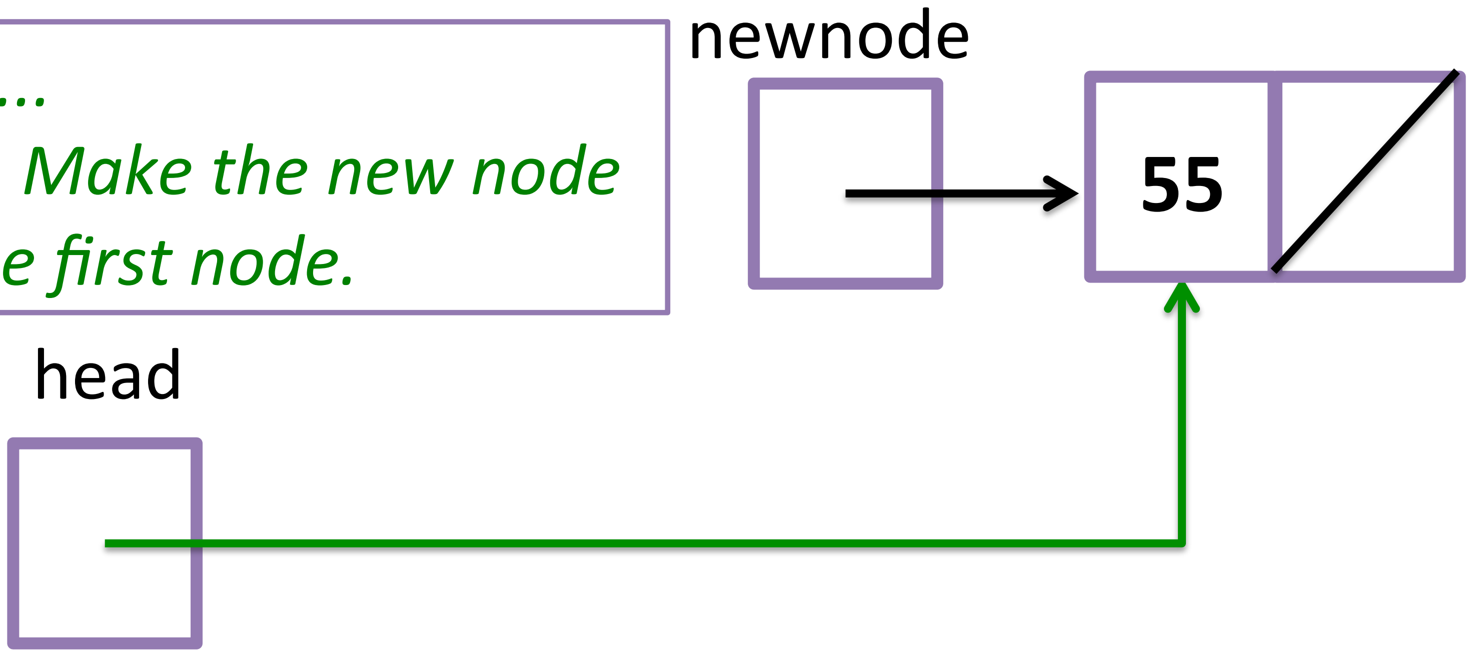
Appending a Node to the List

If there are no nodes in the list...



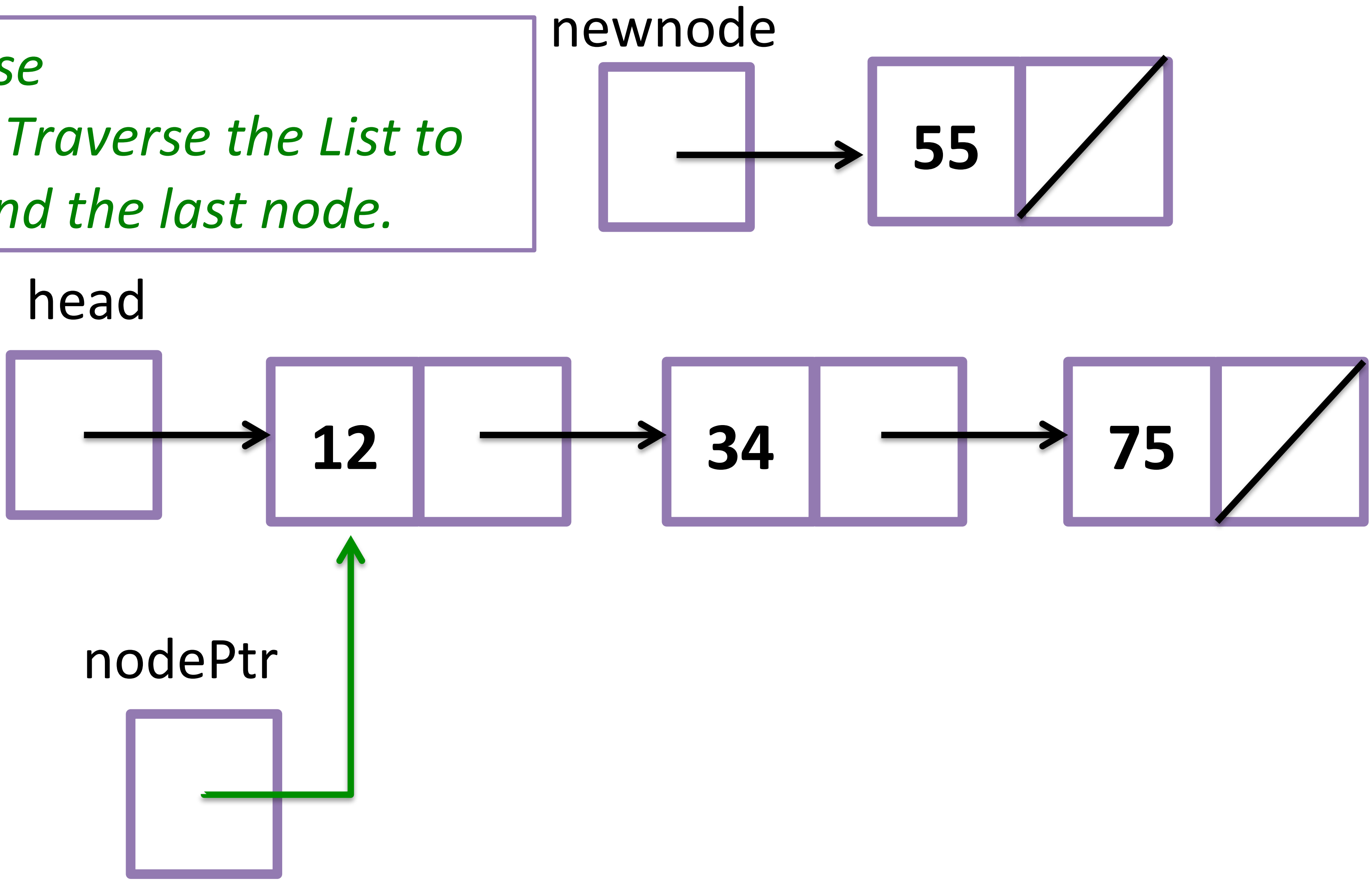
Appending a Node to the List

If ...
Make the new node the first node.



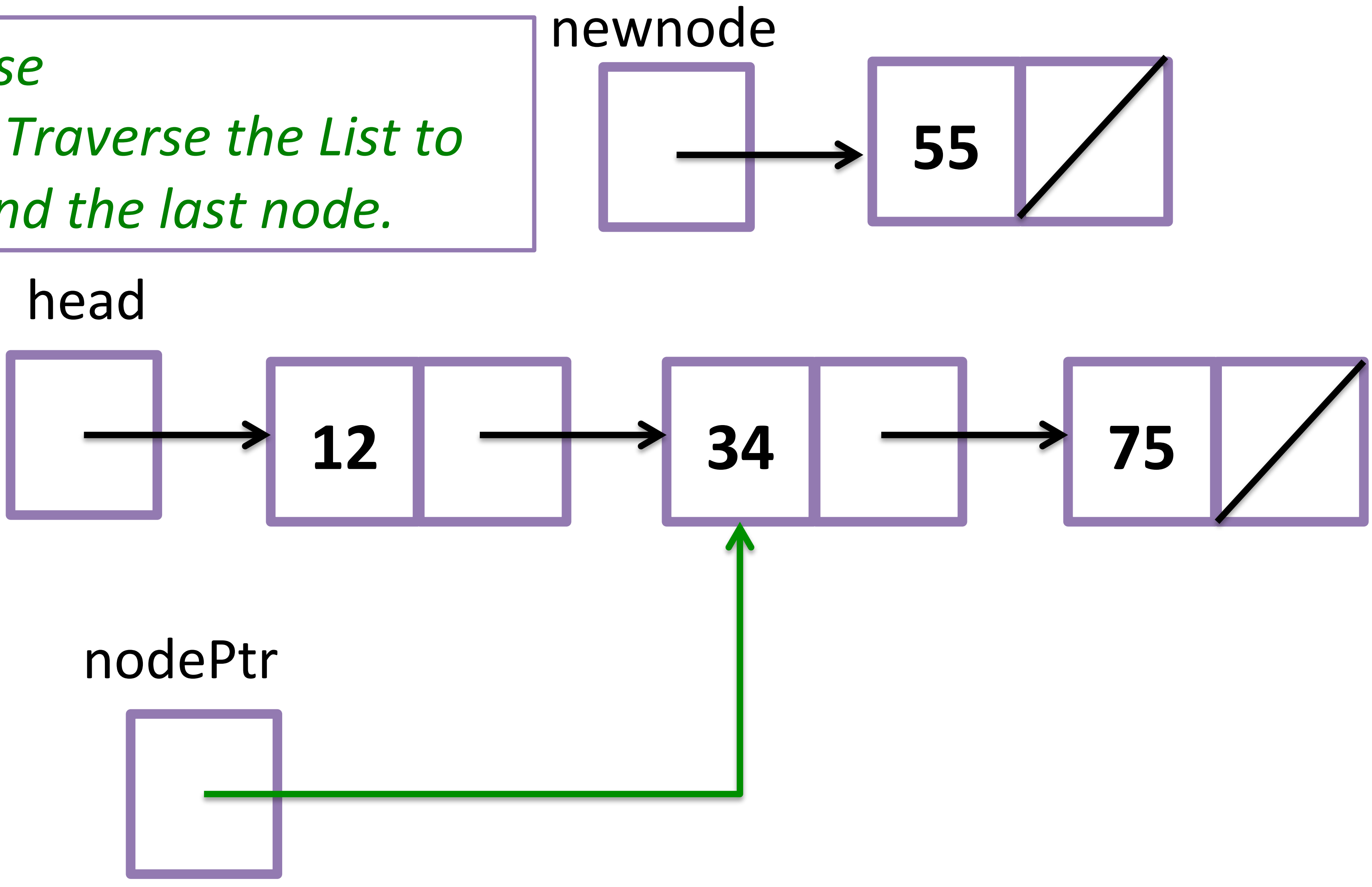
Appending a Node to the List

Else
Traverse the List to
Find the last node.



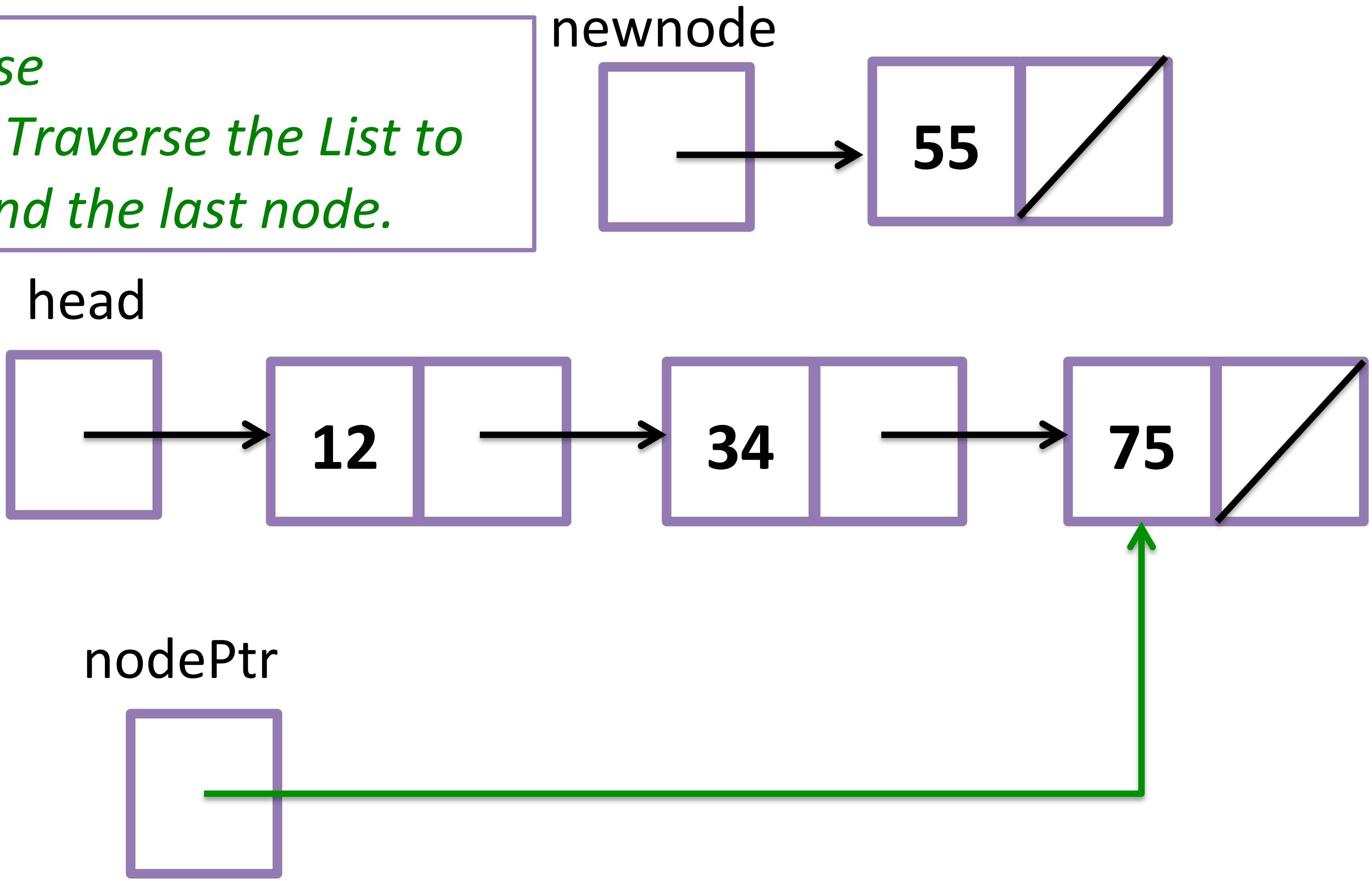
Appending a Node to the List

Else
Traverse the List to
Find the last node.



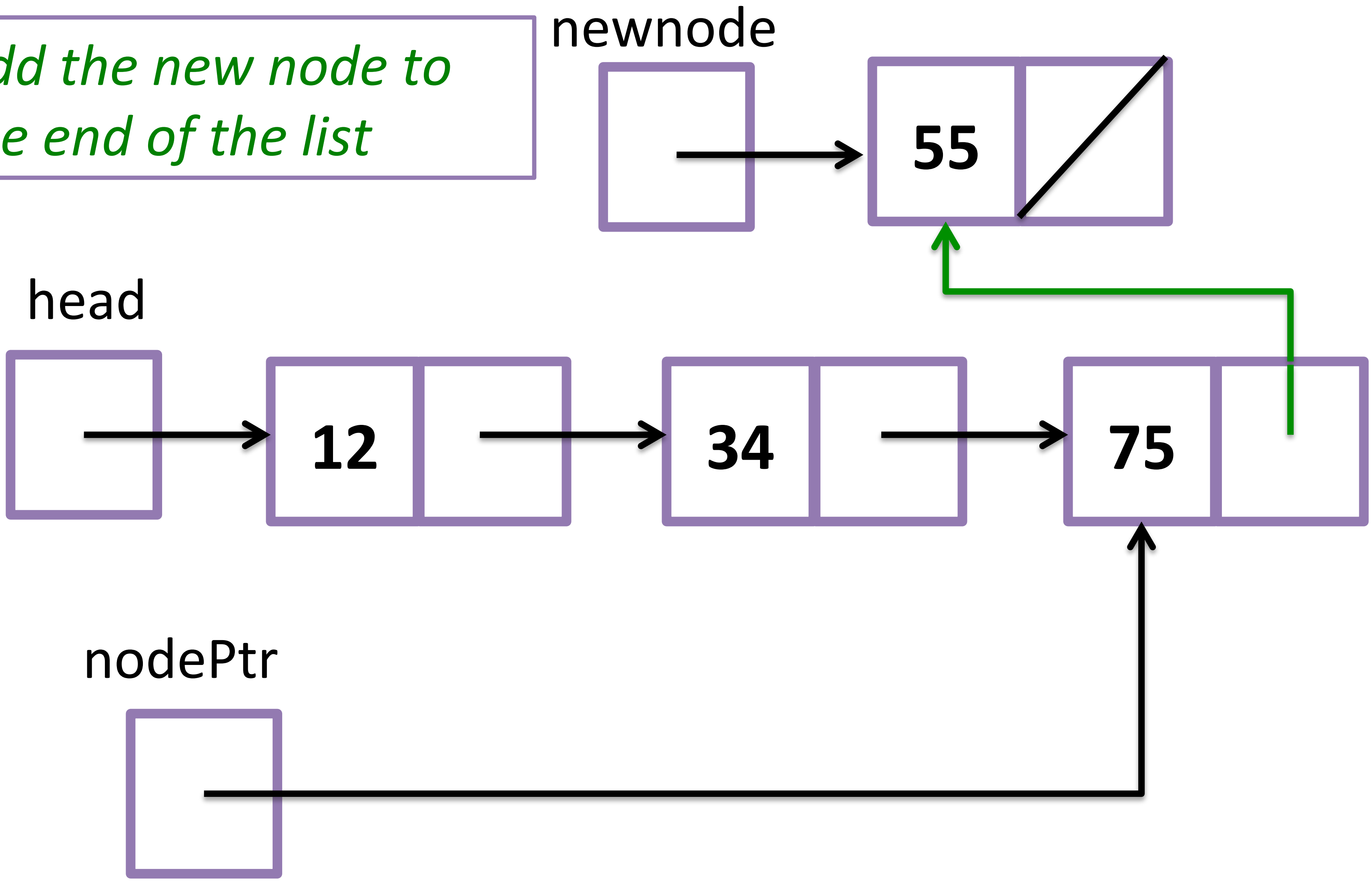
Appending a Node to the List

Else
Traverse the List to
Find the last node.



Appending a Node to the List

Add the new node to the end of the list



Inserting a Node to the List

Insert a Node to the List

Insert = add the node in a particular position 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
  Find the first node whose value is greater than or equal 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 node was found.
End If

Deleting a Node in the List

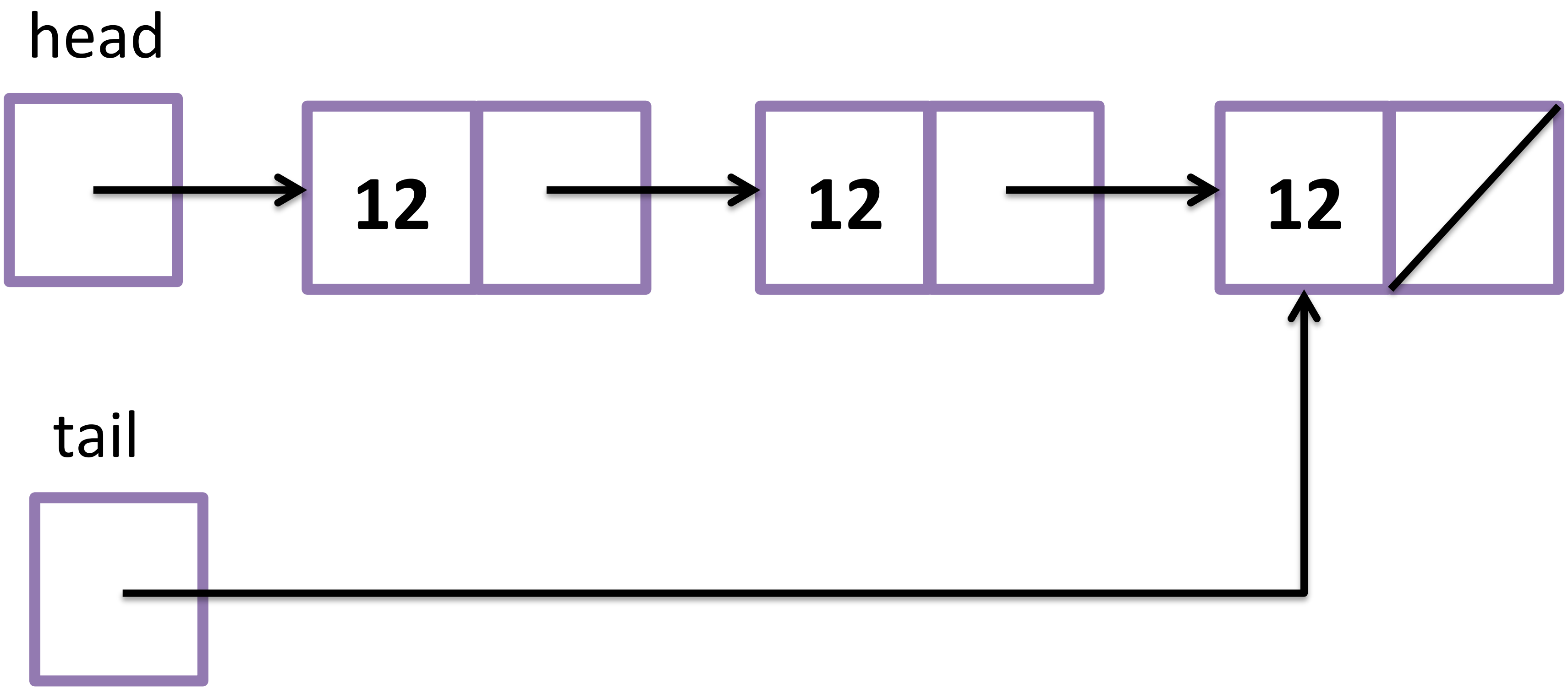
Deleting a Node

- Delete node
  - Remove the node from the list without breaking the links created by the next pointers
  - Delete the node from memory

Destroying the List

- Step through the list
- Delete each node one-by-one

Head and Tail pointers



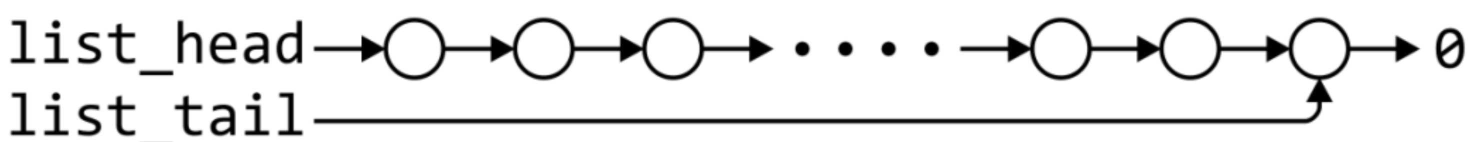
Advantages of Linked Lists over Arrays and `vectors`

- A linked list can easily grow or shrink in size.
- Insertion and deletion of nodes is quicker with linked lists than with vectors.
  - Big O of getting kth element in array? LL?

Singly linked list

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

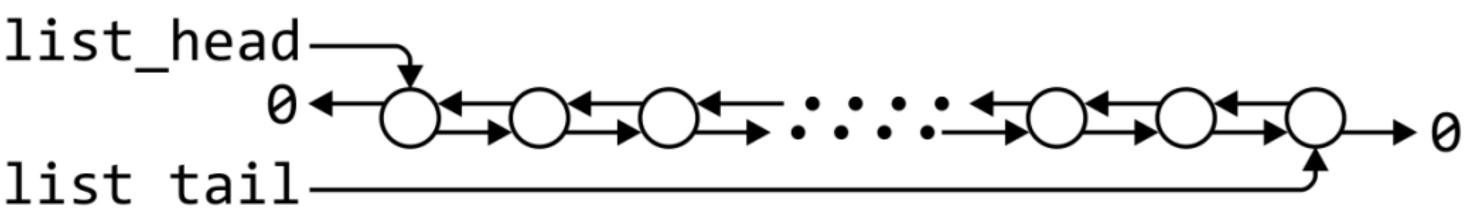
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Doubly linked lists

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$\mathrm{O}(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

* These assume we have already accessed the k^{th} entry—an $\mathrm{O}(n)$ operation



STL

The STL `list` Container

- The **`list`** container, found in the Standard Template Library (STL), is a **template** version of a **doubly linked list**.

- STL **`list`**s can insert elements, or add elements to their front quicker than `vectors` can, because **`list`**s do not have to shift the other elements.

- **`list`**s are also efficient at adding elements at their back because they have a built-in pointer to the last element in the `list` (no traversal required).

A Linked List Template

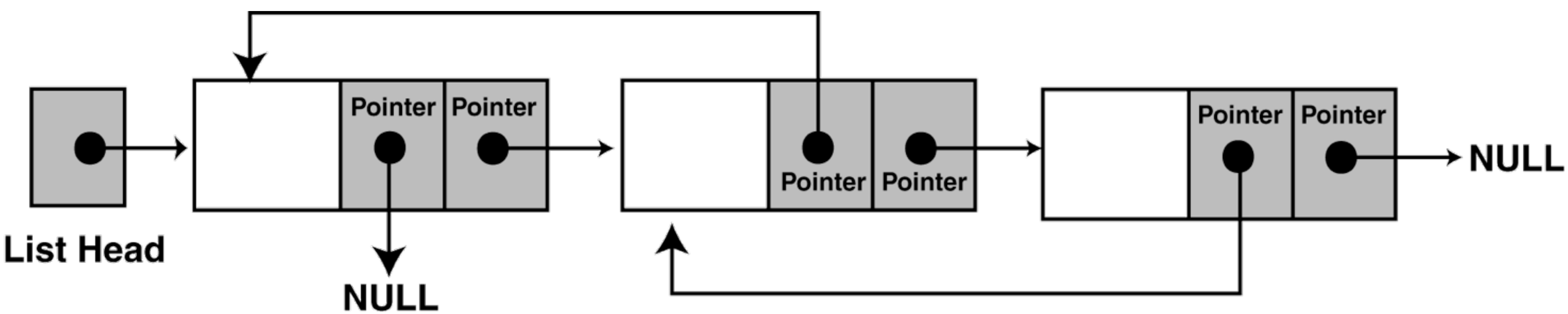
```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

template <class T>
class LinkedList
{
private:
   // Declare a structure for the list
   struct ListNode
   {
      T value;
      struct ListNode *next;
   };

   ListNode *head;   // List head pointer
```

see book for full Linked List template class example

VARIATIONS

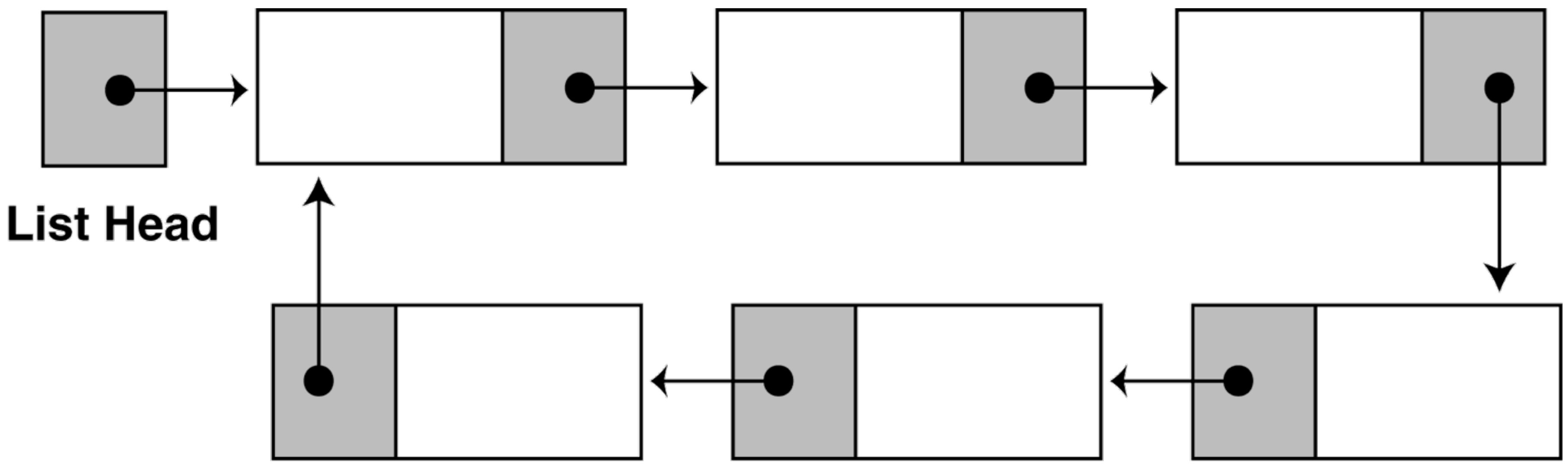
Variations of the Linked List

The Doubly-Linked List



Variations of the Linked List

The Circular Linked List



QUESTIONS TO PONDER

Questions

1. What is Θ for inserting an element at the kth entry in an array? a LL?
2. What is Θ for de-allocating the memory of a singly-linked list?
3. What is Θ for concatenating 2 singly-linked lists? How does it change if you have a tail pointer? What if it is a doubly-linked list?

Questions

1. How does the algorithm differ for deleting a node in the list if there is a tail pointer?
2. How does the implementation of arrays and linked-lists differ?
3. Implement a linked-list. What changes are required if using a tail pointer to make it more efficient in some operations? By making it a doubly-linked list, what changes are required? Which operations are now easier to implement?