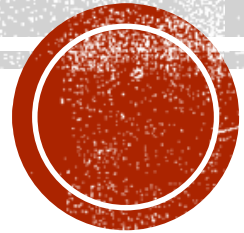
LINKED LISTS



INSERTING A NODE

- Using the `listNode` structure again, the pseudocode on the next slide shows an algorithm for finding a new node's proper position in the list and inserting there.
- The algorithm assumes the nodes in the list are already in order.

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.

The code for the traversal algorithm is shown below. (As before, `num` holds the value being inserted into the list.)

```
// Initialize nodePtr to head of list
nodePtr = head;

// Skip all nodes whose value member is less
// than num.
while (nodePtr != NULL && nodePtr->value < num)
{
      previousNode = nodePtr;
      nodePtr = nodePtr->next;
}
```

The entire `insertNode` function begins on the next slide.

```
void FloatList::insertNode(float num)
{
        ListNode *newNode, *nodePtr, *previousNode;

        // Allocate a new node & store Num
        newNode = new ListNode;
        newNode->value = num;

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

                // Skip all nodes whose value member is less
                // than num.
                while (nodePtr != NULL && nodePtr->value < num)
                {
                        previousNode = nodePtr;
                        nodePtr = nodePtr->next;
                }
```

Continued on next slide…

Continued from previous slide.

```
                // If the new mode is to be the 1st in the list,
                // insert it before all other nodes.
                if (!previousNode)
                {
                      head = newNode;
                      newNode->next = nodePtr;
                }
                else
                {
                      previousNode->next = newNode;
                      newNode->next = nodePtr;
                }
          }
}
```

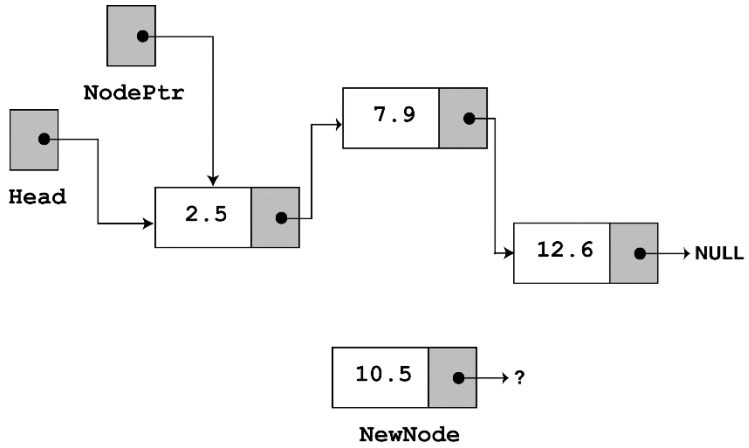
PROGRAM 17-3

```
// This program calls the displayList member function.
// The function traverses the linked list displaying
// the value stored in each node.
#include <iostream.h>
#include "FloatList.h”

void main(void)
{
        FloatList list;

        // Build the list
        list.appendNode(2.5);
        list.appendNode(7.9);
        list.appendNode(12.6);

        // Insert a node in the middle
        // of the list.
        list.insertNode(10.5);

        // Dispay the list
        list.displayList();
}
```

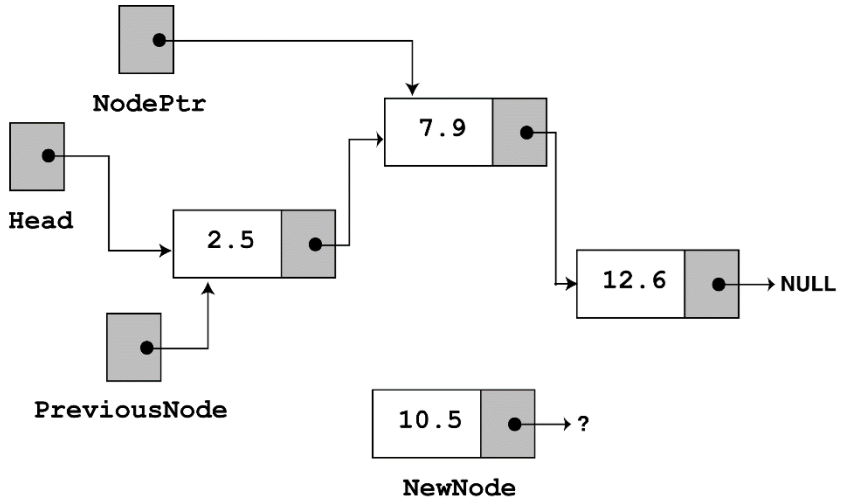
PROGRAM 17-3 OUTPUT

```
2.5
7.9
10.5
12.6
```

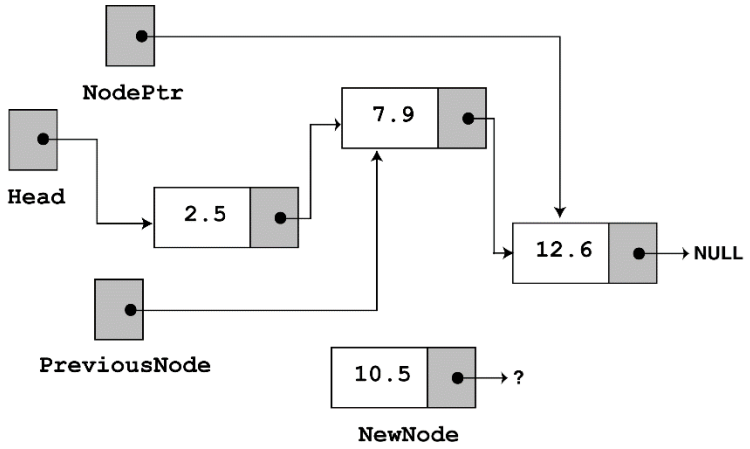
In `insertNode`, a new node is created and the function argument is copied to its `value` member. Since the list already has nodes stored in it, the `else` part of the `if` statement will execute. It begins by assigning `nodePtr` to head.



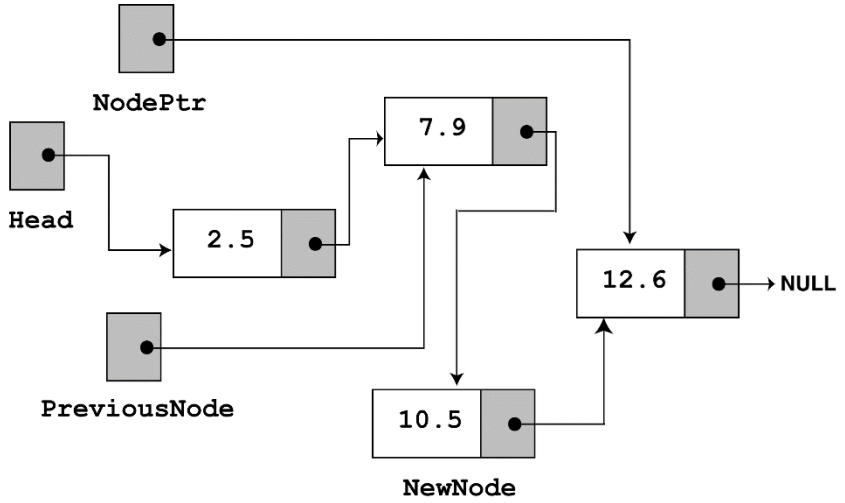
Since `nodePtr` is not NULL and `nodePtr->value` is less than `num`, the `while` loop will iterate. During the iteration, `previousNode` will be made to point to the node that `nodePtr` is pointing to. `nodePtr` will then be advanced to point to the next node.



Once again, the loop performs its test. Since `nodePtr` is not NULL and `nodePtr->value` is less than `num`, the loop will iterate a second time. During the second iteration, both `previousNode` and `nodePtr` are advanced by one node in the list.



This time, the loop's test will fail because `nodePtr` is not less than `num`. The statements after the loop will execute, which cause `previousNode->next` to point to `newNode`, and `newNode->next` to point to `nodePtr`.



If you follow the links, from the `head` pointer to the NULL, you will see that the nodes are stored in the order of their `value` members.

DELETING A NODE

- Deleting a node from a linked list requires two steps:
  - Remove the node from the list without breaking the links created by the next pointers
  - Deleting the node from memory
- The `deleteNode` function begins on the next slide.

```
void FloatList::deleteNode(float num)
{
      ListNode *nodePtr, *previousNode;

      // If the list is empty, do nothing.
      if (!head)
            return;

      // Determine if the first node is the one.
      if (head->value == num)
      {
            nodePtr = head->next;
            delete head;
            head = nodePtr;
      }
```

Continued on next slide…

Continued from previous slide.

```
        else
        {
                // Initialize nodePtr to head of list
                nodePtr = head;

                // Skip all nodes whose value member is
                // not equal to num.
                while (nodePtr != NULL && nodePtr->value != num)
                {
                        previousNode = nodePtr;
                        nodePtr = nodePtr->next;
                }

                // Link the previous node to the node after
                // nodePtr, then delete nodePtr.
                previousNode->next = nodePtr->next;
                delete nodePtr;
        }
}
```

PROGRAM 17-4

```
// This program demonstrates the deleteNode member function
#include <iostream.h>
#include "FloatList.h"

void main(void)
{
        FloatList list;

        // Build the list
        list.appendNode(2.5);
        list.appendNode(7.9);
        list.appendNode(12.6);
        cout << "Here are the initial values:\n";
        list.displayList();
        cout << endl;

        cout << "Now deleting the node in the middle.\n";
        cout << "Here are the nodes left.\n";
        list.deleteNode(7.9);
        list.displayList();
        cout << endl;
```

Continued on next slide…

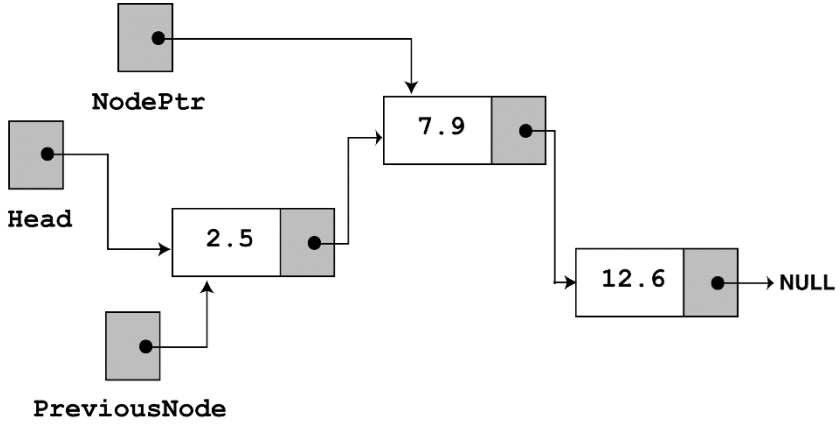
Continued from previous slide.

```
        cout << "Now deleting the last node.\n";
        cout << "Here are the nodes left.\n";
        list.deleteNode(12.6);
        list.displayList();
        cout << endl;

        cout << "Now deleting the only remaining node.\n";
        cout << "Here are the nodes left.\n";
        list.deleteNode(2.5);
        list.displayList();
}
```

Program Output

```
Here are the initial values:
2.5
7.9
12.6

Now deleting the node in the middle.
Here are the nodes left.
2.5
12.6

Now deleting the last node.
Here are the nodes left.
2.5

Now deleting the only remaining node.
Here are the nodes left.
```

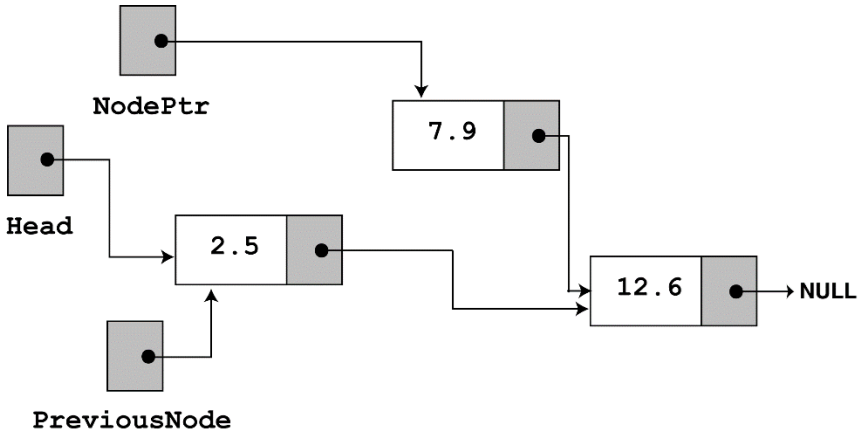
Look at the `else` part of the second `if` statement. This is where the function will perform its action since the list is not empty, and the first node does not contain the value 7.9. Just like `insertNode`, this function uses `nodePtr` and `previousNode` to traverse the list. The while loop terminates when the value 7.9 is located. At this point, the list and the other pointers will be in the state depicted in the figure below.



next, the following statement executes.

```
previousNode->next = nodePtr->next;
```

The statement above causes the links in the list to bypass the node that nodePtr points to. Although the node still exists in memory, this removes it from the list.



The last statement uses the `delete` operator to complete the total deletion of the node.

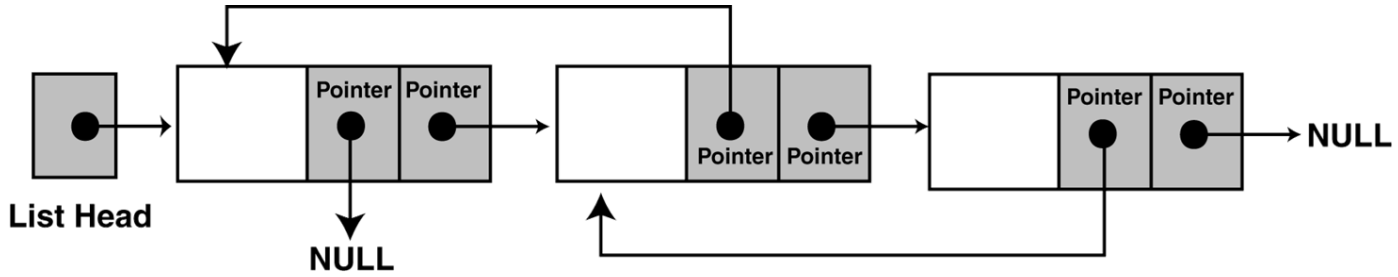
DESTROYING THE LIST

- The class's destructor should release all the memory used by the list.
- It does so by stepping through the list, deleting each node one-by-one. The code is shown on the next slide.

```
FloatList::~FloatList(void)
{
        ListNode *nodePtr, *nextNode;


        nodePtr = head;
        while (nodePtr != NULL)
        {
                nextNode = nodePtr->next;
                delete nodePtr;
                nodePtr = nextNode;
        }
}
```

Notice the use of `nextNode` instead of `previousNode`. The `nextNode` pointer is used to hold the position of the next node in the list, so it will be available after the node pointed to by `nodePtr` is deleted.

DOUBLY-LINKED LIST



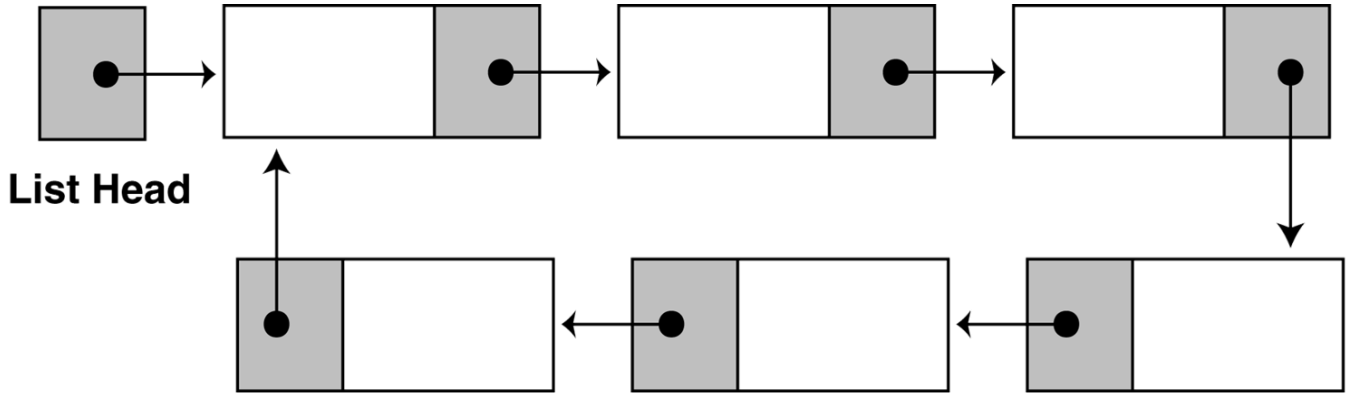
Advantages:

- Convenient to traverse the list backwards.
- Simplifies insertion and deletion because you no longer have to refer to the previous node.

Disadvantage:

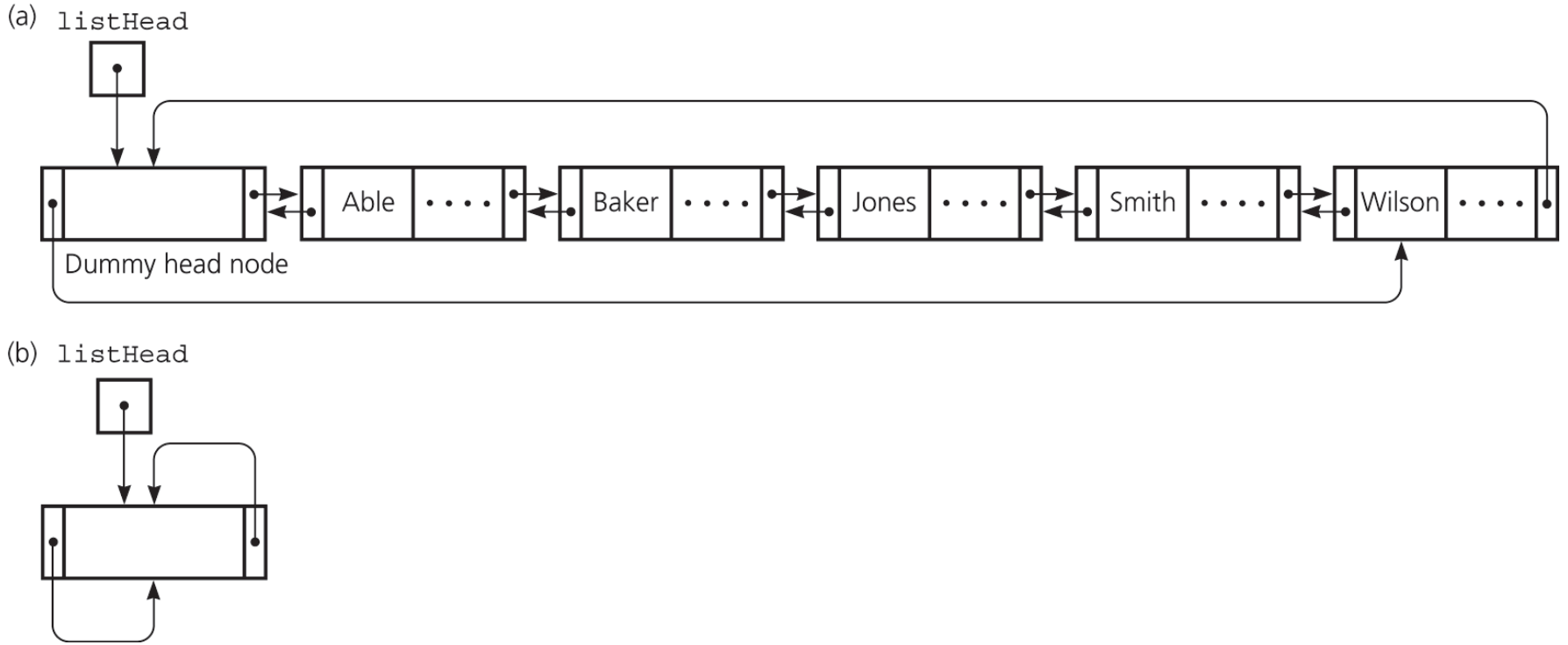
- Increase in space requirements.

CIRCULAR LINKED LIST



- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*

CIRCULAR DOUBLY LINKED LIST



- Circular doubly linked list
  - `prev` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions