1. Define the Node structure

```
struct Node {
   int data;       // The data stored in the node
   Node* next;     // Pointer to the next node

   // Constructor to initialize a new node with given data
   Node(int data) {
       this->data = data;
       this->next = nullptr;  // Initially, the next pointer is
null
   }
};
```

2. Appending a node

```
void append(Node*& head, int data) {
   Node* newNode = new Node(data);  // Create a new node with
the given data
   if (head == nullptr) {
       head = newNode;  // If the list is empty, set head to the
new node
   } else {
       Node* temp = head;
       while (temp->next != nullptr) {  // Traverse until the
last node
           temp = temp->next;
       }
       temp->next = newNode;  // Set the last node's next
pointer to the new node
   }
}
```

3. Displaying the list

```
void display(Node* head) {
   Node* temp = head;
   while (temp != nullptr) {
       cout << temp->data << " -> ";  // Print the current
node's data
       temp = temp->next;  // Move to the next node
```

```
    }
    cout << "nullptr" << endl;   // Indicate the end of the list
}
```

4. Deleting a node by value

```
void deleteValue(Node*& head, int value) {
    if (head == nullptr) return;   // If the list is empty,
there's nothing to delete

    // If the node to delete is the head
    if (head->data == value) {
        Node* temp = head;
        head = head->next;   // Move the head to the next node
        delete temp;   // Delete the old head
        return;
    }

    // Traverse the list to find the node to delete
    Node* temp = head;
    while (temp->next != nullptr && temp->next->data != value) {
        temp = temp->next;
    }

    // If the value is not found in the list
    if (temp->next == nullptr) {
        cout << "Value not found!" << endl;
        return;
    }

    // Delete the node
    Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;   // Skip over the node to
delete
    delete nodeToDelete;   // Free the memory of the deleted node
}
```

5. Cleaning up the list

```
void deleteList(Node*& head) {
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;  // Move the head to the next node
        delete temp;  // Delete the old head
    }
}
```

6. Main function

```
int main() {
    Node* head = nullptr;  // Initialize the head of the list as 
nullptr

    // Append some values to the list
    append(head, 10);
    append(head, 20);
    append(head, 30);

    // Display the list
    cout << "List after appending values: ";
    display(head);

    // Delete a node by value
    deleteValue(head, 20);
    cout << "List after deleting 20: ";
    display(head);

    // Try deleting a non-existing value
    deleteValue(head, 40);

    // Clean up and delete the entire list
    deleteList(head);

    return 0;
}
```

Refactor to Template-based

1. Define the Node structure

```
template <typename T>
```

```
struct Node {
   T data;        // Data can be of any type (e.g., int, float,
etc.)
   Node* next;    // Pointer to the next node

   Node(T data) {  // Constructor to initialize node with given
data
       this->data = data;
       this->next = nullptr;
   }
};
```

2. Appending a node

```
template <typename T>
void append(Node<T>*& head, T data) {
   Node<T>* newNode = new Node<T>(data);  // Create a new node
with the given data
   if (head == nullptr) {
       head = newNode;  // If the list is empty, set the head to
the new node
   } else {
       Node<T>* temp = head;
       while (temp->next != nullptr) {  // Traverse to the last
node
           temp = temp->next;
       }
       temp->next = newNode;  // Set the last node's next
pointer to the new node
   }
}
```

3. Displaying the list

```
template <typename T>
void display(Node<T>* head) {
   Node<T>* temp = head;
   while (temp != nullptr) {
        std::cout << temp->data << " -> ";  // Print the data of
the current node
        temp = temp->next;  // Move to the next node
    }
    std::cout << "nullptr" << std::endl;  // Indicate the end of
the list
}
```

4. Deleting a node by value

```
template <typename T>
void deleteValue(Node<T>*& head, T value) {
    if (head == nullptr) return;  // If the list is empty,
nothing to delete

    if (head->data == value) {  // If the value to delete is the
head
        Node<T>* temp = head;
        head = head->next;  // Move the head to the next node
        delete temp;  // Delete the old head
        return;
    }

    Node<T>* temp = head;
    while (temp->next != nullptr && temp->next->data != value) {
// Traverse to find the node
        temp = temp->next;
    }

    if (temp->next == nullptr) {  // If the value is not found
        std::cout << "Value not found!" << std::endl;
        return;
    }

    Node<T>* nodeToDelete = temp->next;
    temp->next = temp->next->next;  // Skip the node to delete
    delete nodeToDelete;  // Delete the node
}
```

5. Deleting the entire list

```
template <typename T>
void deleteList(Node<T>*& head) {
   while (head != nullptr) {
       Node<T>* temp = head;
       head = head->next;   // Move the head to the next node
       delete temp;   // Delete the old head
   }
}
```

6. Main function

```
int main() {
   // Test with int type
   Node<int>* intHead = nullptr;
   append(intHead, 10);
   append(intHead, 20);
   append(intHead, 30);
   cout << "Integer List: ";
   display(intHead);
   deleteValue(intHead, 20);
   cout << "Integer List after deleting 20: ";
   display(intHead);

   // Test with float type
   Node<float>* floatHead = nullptr;
   append(floatHead, 10.5f);
   append(floatHead, 20.5f);
   append(floatHead, 30.5f);
   cout << "Float List: ";
   display(floatHead);
   deleteValue(floatHead, 20.5f);
   cout << "Float List after deleting 20.5: ";
   display(floatHead);

   // Test with double type
   Node<double>* doubleHead = nullptr;
   append(doubleHead, 10.123);
   append(doubleHead, 20.456);
```

```
    append(doubleHead, 30.789);
    cout << "Double List: ";
    display(doubleHead);
    deleteValue(doubleHead, 20.456);
    cout << "Double List after deleting 20.456: ";
    display(doubleHead);

    // Clean up
    deleteList(intHead);
    deleteList(floatHead);
    deleteList(doubleHead);

    return 0;
}
```