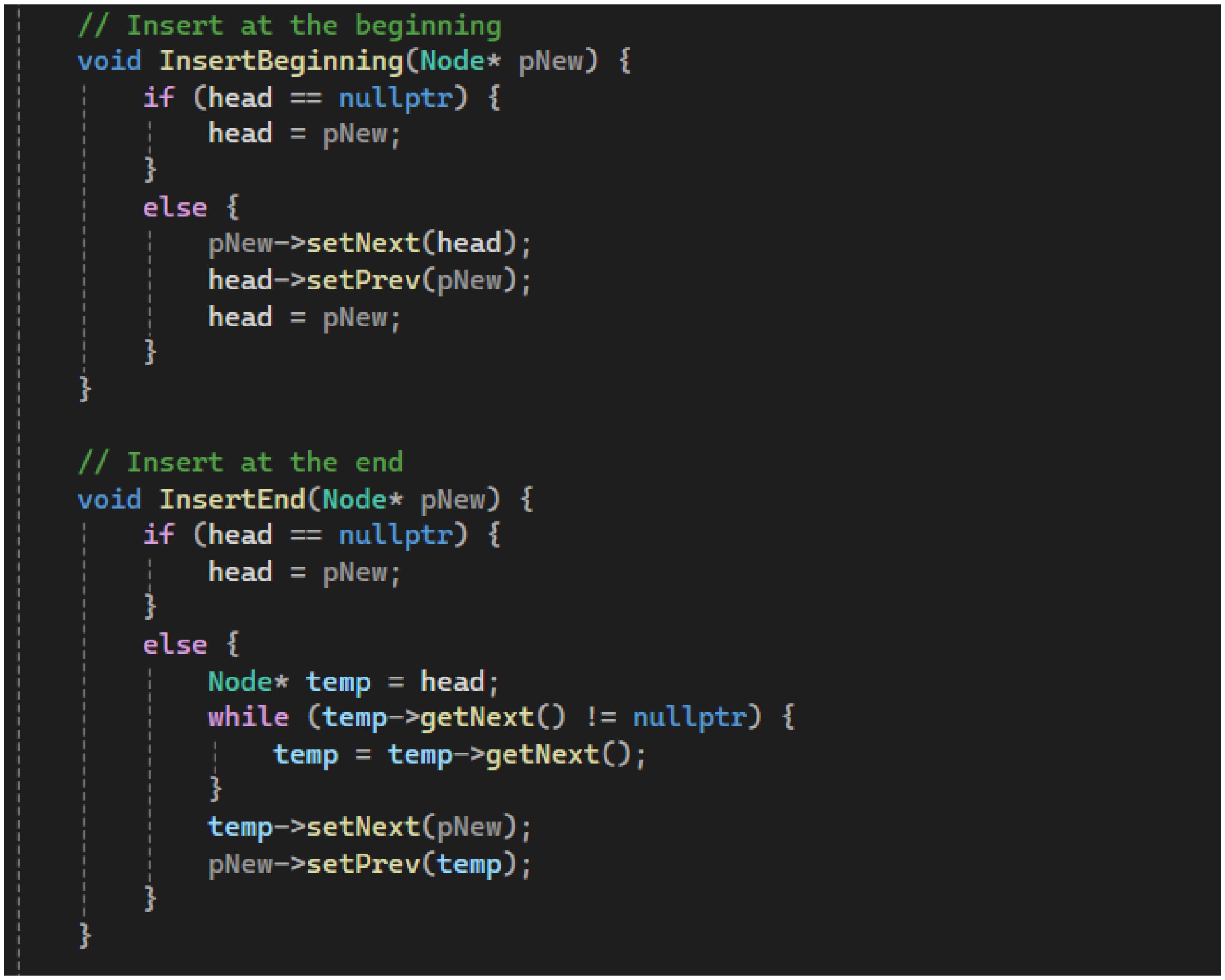
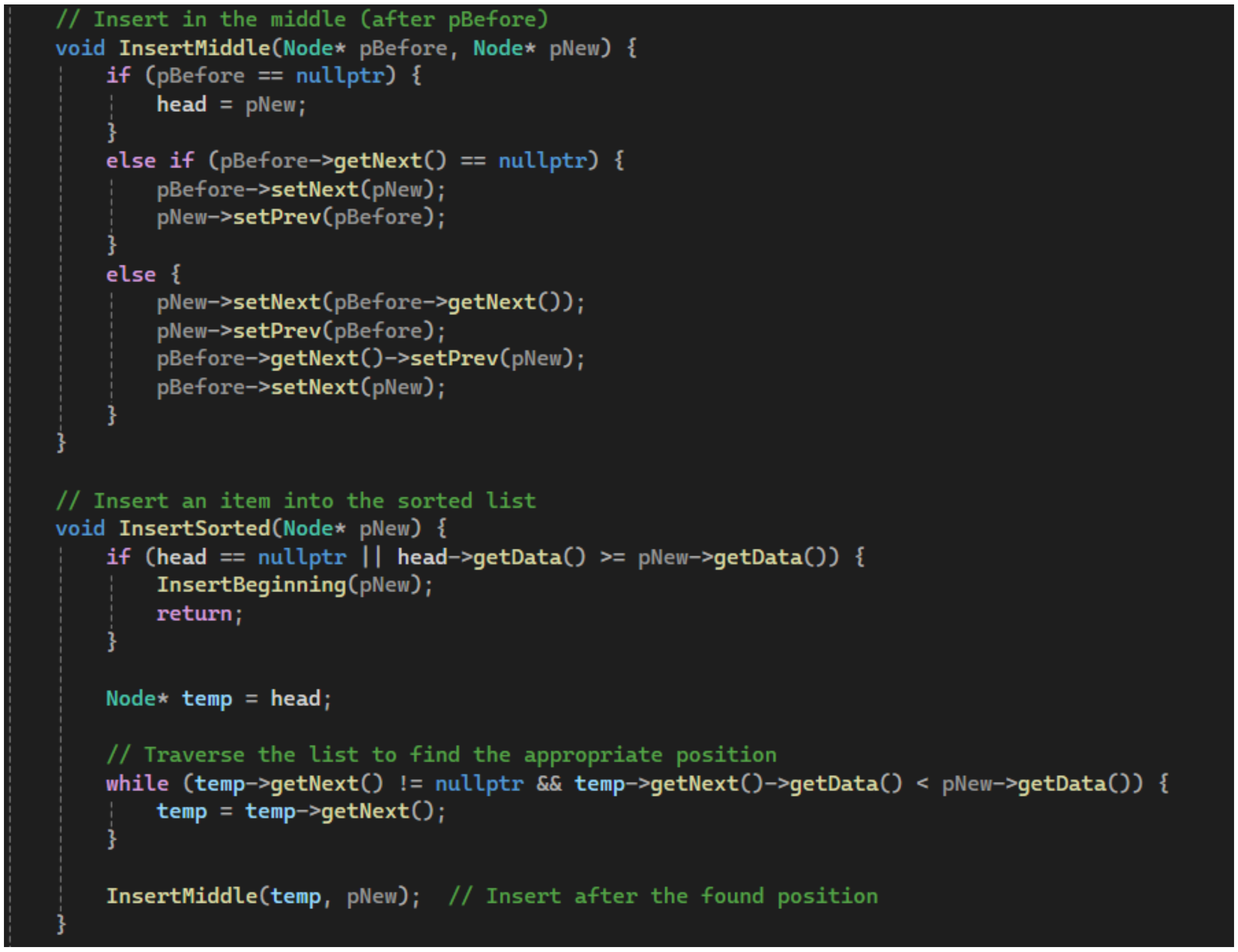
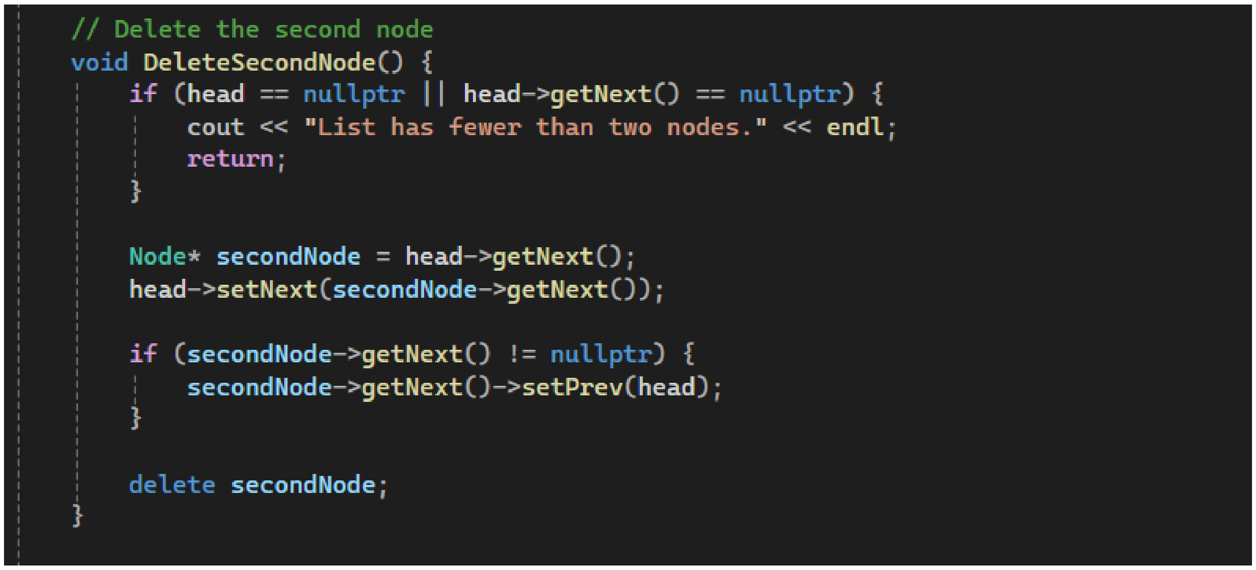
- **QUES 3**
  Doubly



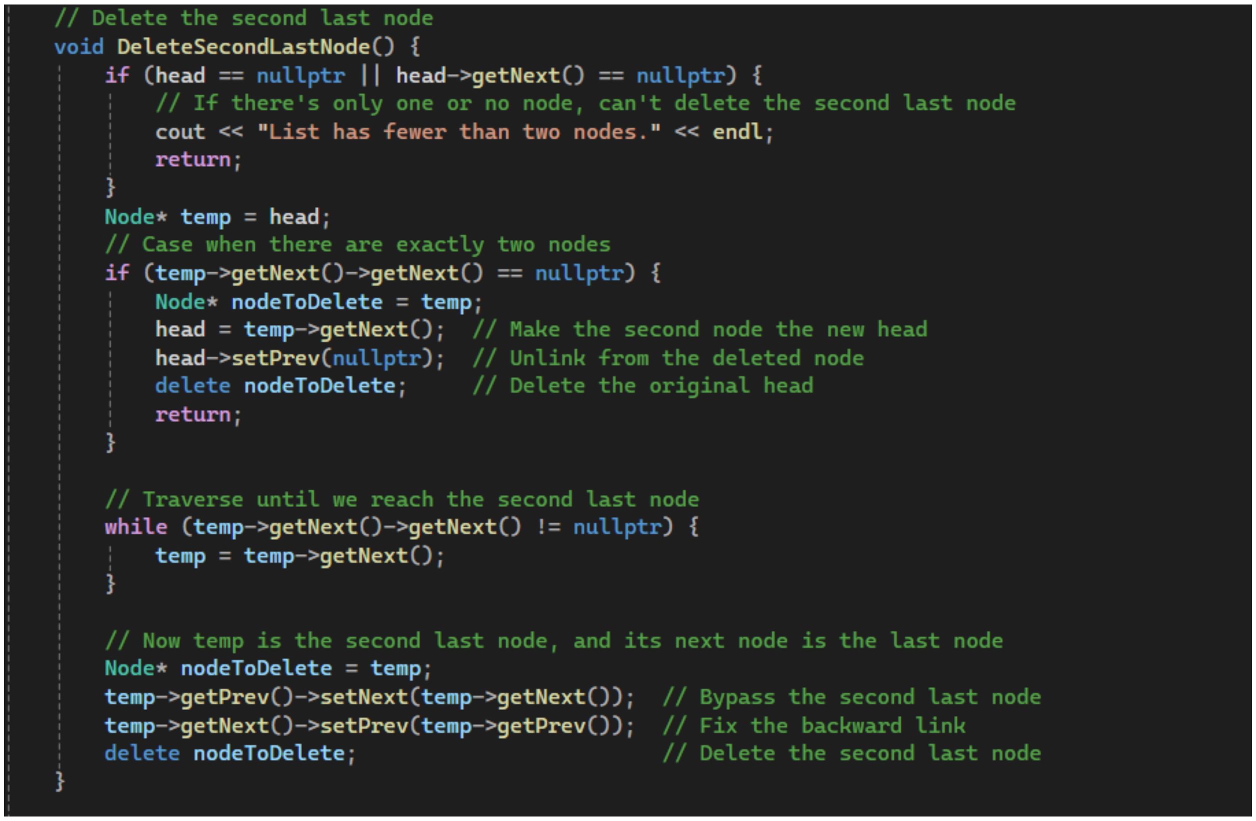
```
// Insert at the beginning
void InsertBeginning(Node* pNew) {
    if (head == nullptr) {
        head = pNew;
    }
    else {
        pNew->setNext(head);
        head->setPrev(pNew);
        head = pNew;
    }
}

// Insert at the end
void InsertEnd(Node* pNew) {
    if (head == nullptr) {
        head = pNew;
    }
    else {
        Node* temp = head;
        while (temp->getNext() != nullptr) {
            temp = temp->getNext();
        }
        temp->setNext(pNew);
        pNew->setPrev(temp);
    }
}
```



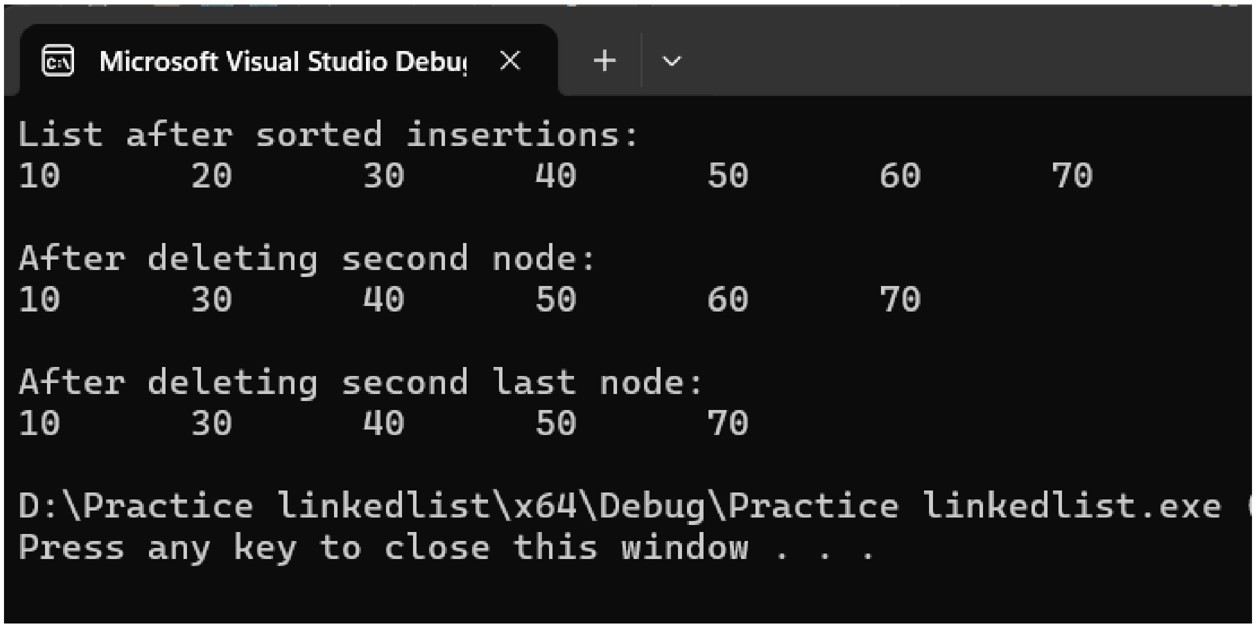
```cpp
// Insert in the middle (after pBefore)
void InsertMiddle(Node* pBefore, Node* pNew) {
    if (pBefore == nullptr) {
        head = pNew;
    }
    else if (pBefore->getNext() == nullptr) {
        pBefore->setNext(pNew);
        pNew->setPrev(pBefore);
    }
    else {
        pNew->setNext(pBefore->getNext());
        pNew->setPrev(pBefore);
        pBefore->getNext()->setPrev(pNew);
        pBefore->setNext(pNew);
    }
}

// Insert an item into the sorted list
void InsertSorted(Node* pNew) {
    if (head == nullptr || head->getData() >= pNew->getData()) {
        InsertBeginning(pNew);
        return;
    }

    Node* temp = head;

    // Traverse the list to find the appropriate position
    while (temp->getNext() != nullptr && temp->getNext()->getData() < pNew->getData()) {
        temp = temp->getNext();
    }

    InsertMiddle(temp, pNew);  // Insert after the found position
}
```



```cpp
// Delete the second node
void DeleteSecondNode() {
    if (head == nullptr || head->getNext() == nullptr) {
        cout << "List has fewer than two nodes." << endl;
        return;
    }

    Node* secondNode = head->getNext();
    head->setNext(secondNode->getNext());

    if (secondNode->getNext() != nullptr) {
        secondNode->getNext()->setPrev(head);
    }

    delete secondNode;
}
```



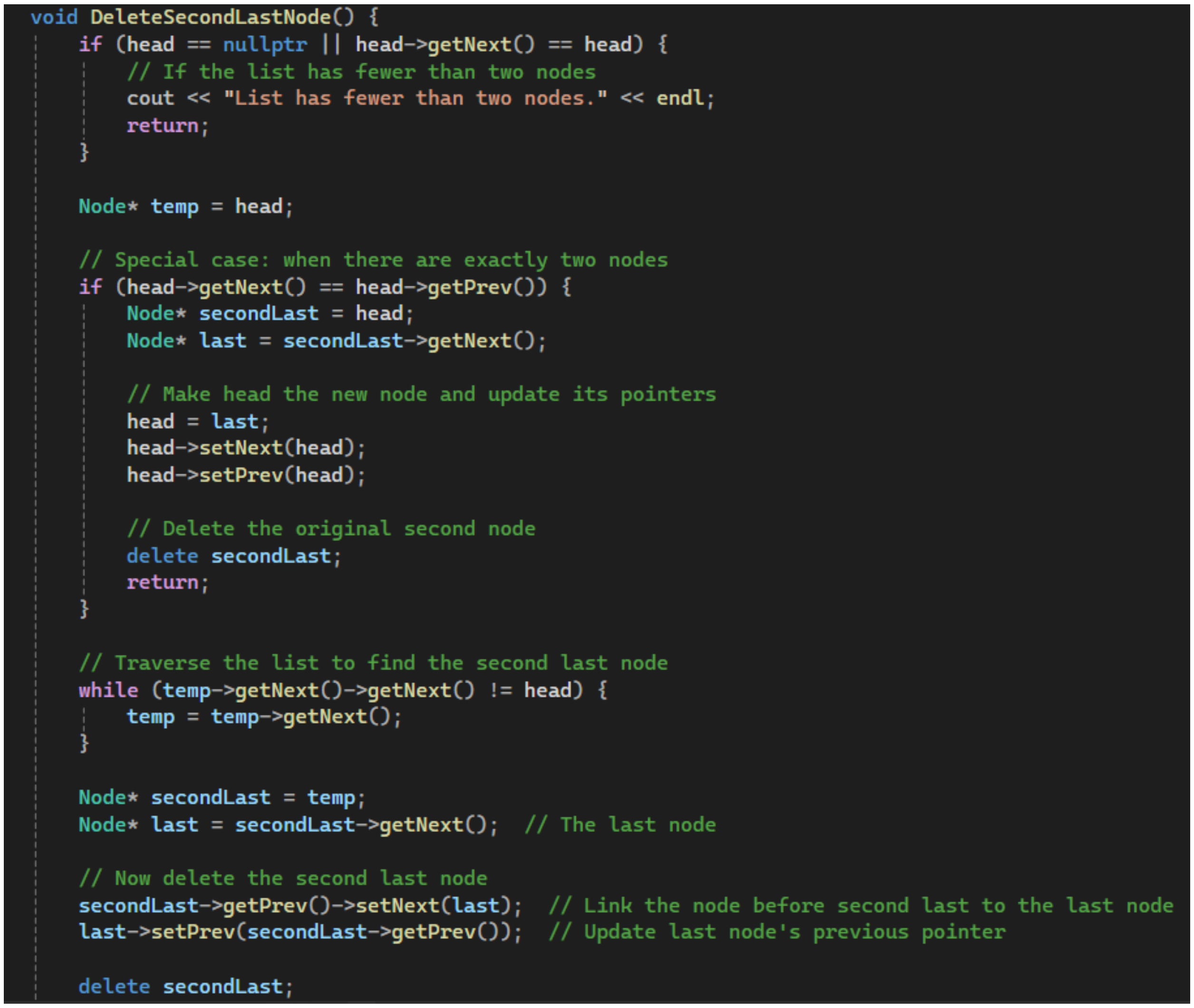
```cpp
// Delete the second last node
void DeleteSecondLastNode() {
    if (head == nullptr || head->getNext() == nullptr) {
        // If there's only one or no node, can't delete the second last node
        cout << "List has fewer than two nodes." << endl;
        return;
    }
    Node* temp = head;
    // Case when there are exactly two nodes
    if (temp->getNext()->getNext() == nullptr) {
        Node* nodeToDelete = temp;
        head = temp->getNext();  // Make the second node the new head
        head->setPrev(nullptr);  // Unlink from the deleted node
        delete nodeToDelete;     // Delete the original head
        return;
    }

    // Traverse until we reach the second last node
    while (temp->getNext()->getNext() != nullptr) {
        temp = temp->getNext();
    }

    // Now temp is the second last node, and its next node is the last node
    Node* nodeToDelete = temp;
    temp->getPrev()->setNext(temp->getNext());  // Bypass the second last node
    temp->getNext()->setPrev(temp->getPrev());  // Fix the backward link
    delete nodeToDelete;                        // Delete the second last node
}
```



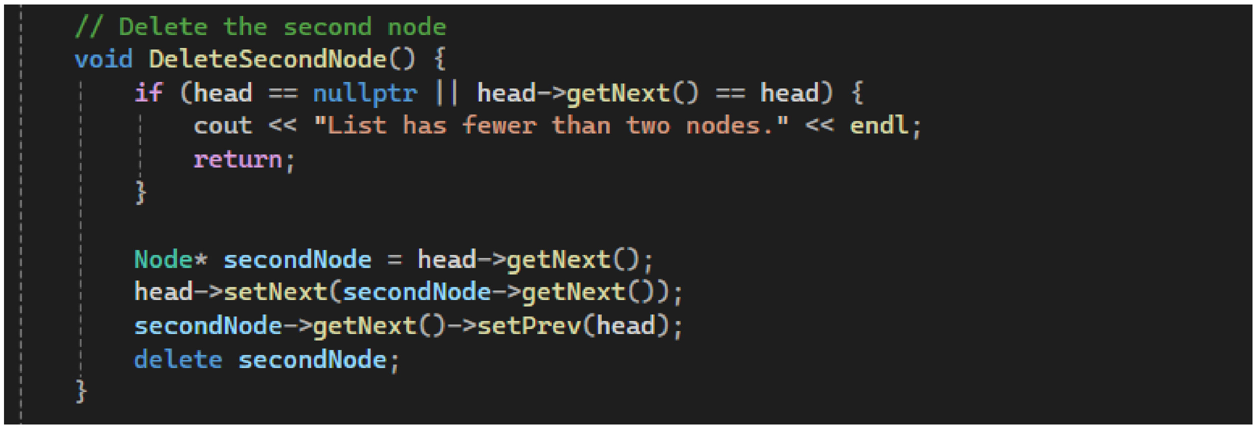
```
List after sorted insertions:
10      20      30      40      50      60      70

After deleting second node:
10      30      40      50      60      70

After deleting second last node:
10      30      40      50      70

D:\Practice linkedlist\x64\Debug\Practice linkedlist.exe
Press any key to close this window . . .
```

- **Ques 4**

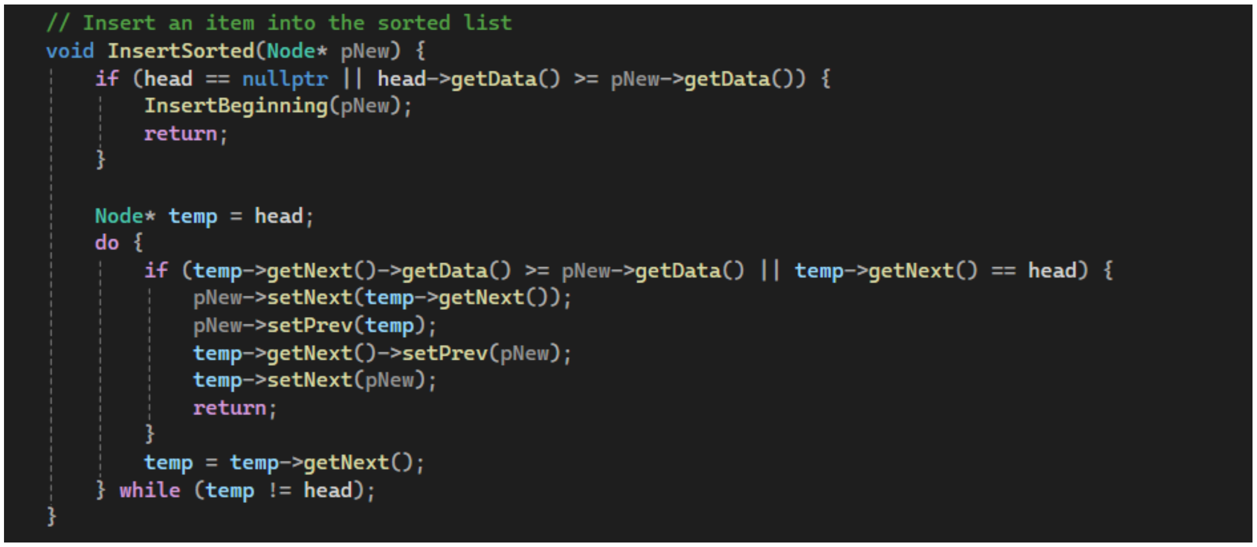
Doubly with circular



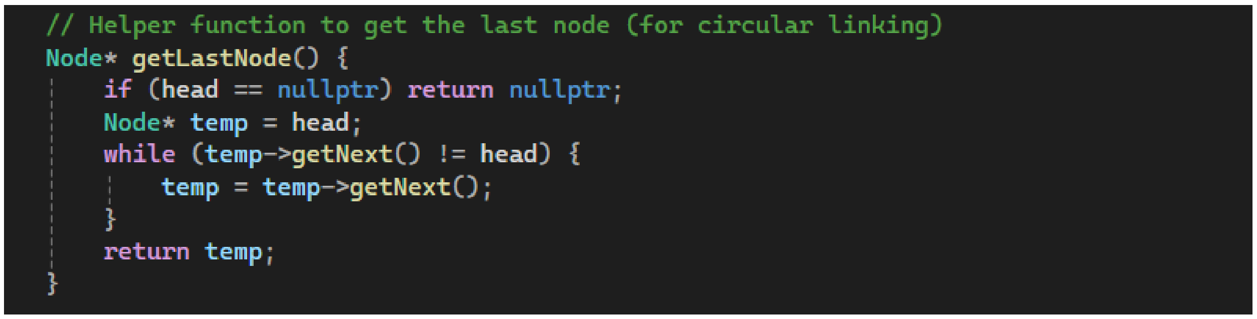
```cpp
void DeleteSecondLastNode() {
    if (head == nullptr || head->getNext() == head) {
        // If the list has fewer than two nodes
        cout << "List has fewer than two nodes." << endl;
        return;
    }

    Node* temp = head;

    // Special case: when there are exactly two nodes
    if (head->getNext() == head->getPrev()) {
        Node* secondLast = head;
        Node* last = secondLast->getNext();

        // Make head the new node and update its pointers
        head = last;
        head->setNext(head);
        head->setPrev(head);

        // Delete the original second node
        delete secondLast;
        return;
    }

    // Traverse the list to find the second last node
    while (temp->getNext()->getNext() != head) {
        temp = temp->getNext();
    }

    Node* secondLast = temp;
    Node* last = secondLast->getNext();  // The last node

    // Now delete the second last node
    secondLast->getPrev()->setNext(last);  // Link the node before second last to the last node
    last->setPrev(secondLast->getPrev());  // Update last node's previous pointer

    delete secondLast;
```



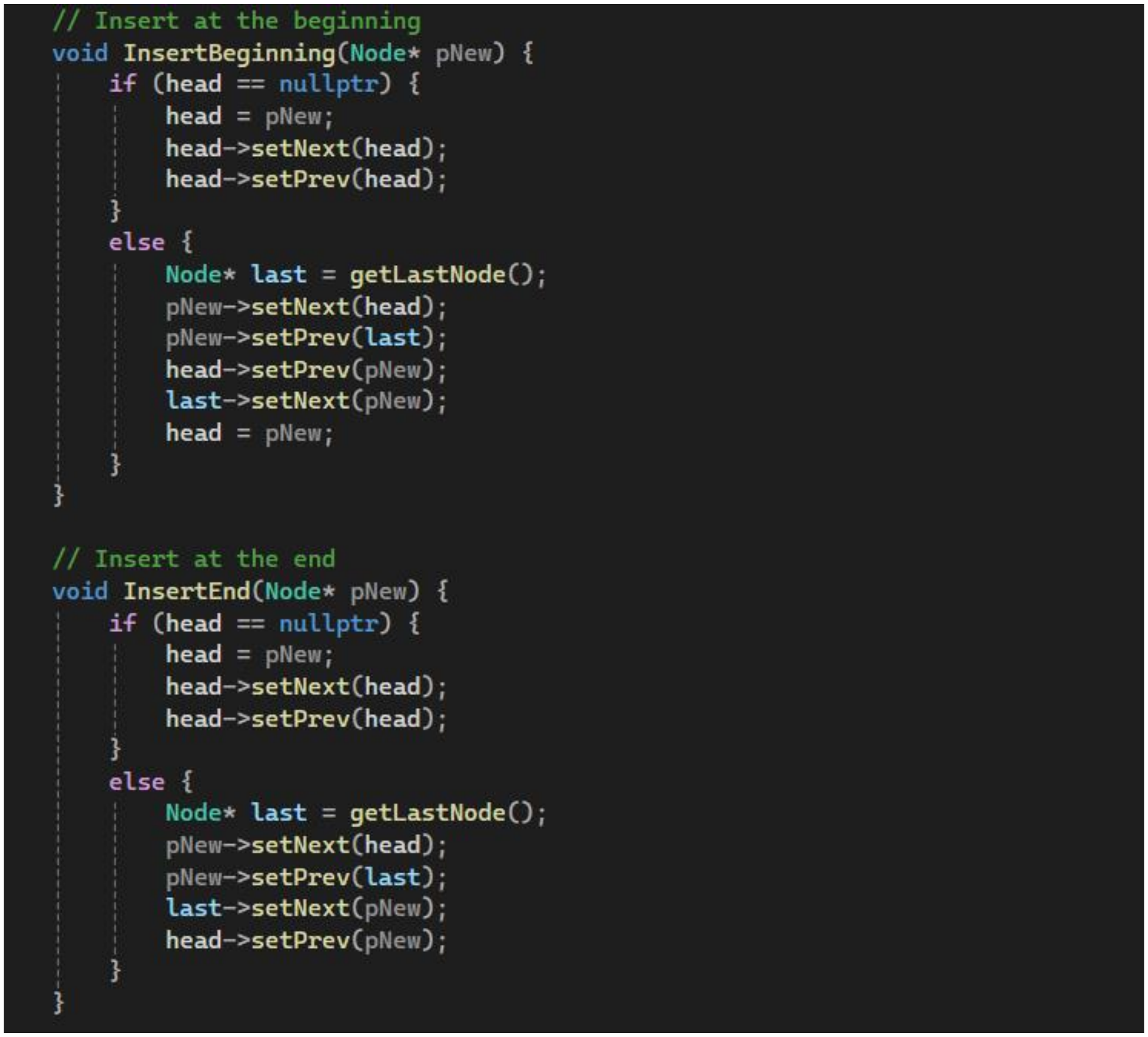
```cpp
    // Delete the second node
    void DeleteSecondNode() {
        if (head == nullptr || head->getNext() == head) {
            cout << "List has fewer than two nodes." << endl;
            return;
        }

        Node* secondNode = head->getNext();
        head->setNext(secondNode->getNext());
        secondNode->getNext()->setPrev(head);
        delete secondNode;
    }
```



```cpp
// Insert an item into the sorted list
void InsertSorted(Node* pNew) {
    if (head == nullptr || head->getData() >= pNew->getData()) {
        InsertBeginning(pNew);
        return;
    }

    Node* temp = head;
    do {
        if (temp->getNext()->getData() >= pNew->getData() || temp->getNext() == head) {
            pNew->setNext(temp->getNext());
            pNew->setPrev(temp);
            temp->getNext()->setPrev(pNew);
            temp->setNext(pNew);
            return;
        }
        temp = temp->getNext();
    } while (temp != head);
}
```

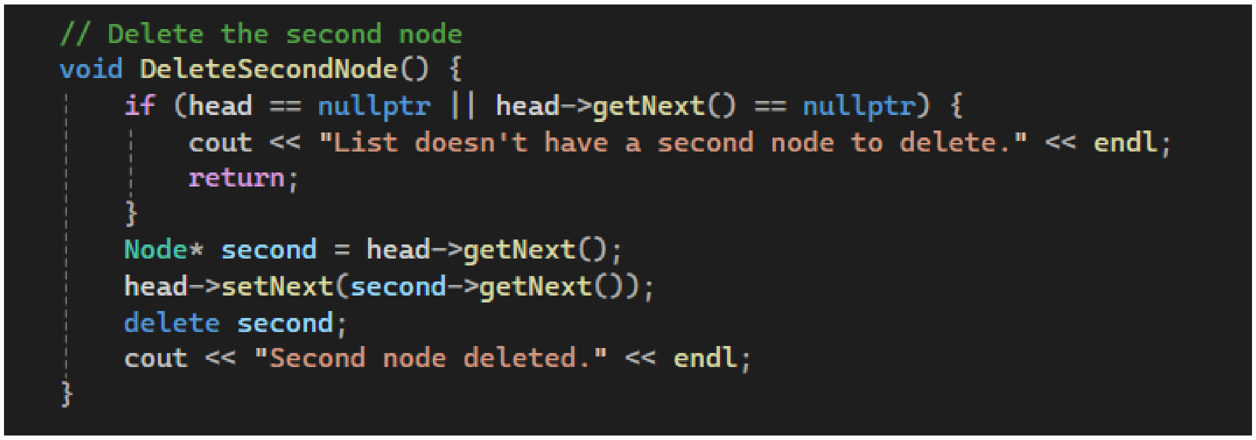


```cpp
// Helper function to get the last node (for circular linking)
Node* getLastNode() {
    if (head == nullptr) return nullptr;
    Node* temp = head;
    while (temp->getNext() != head) {
        temp = temp->getNext();
    }
    return temp;
}
```

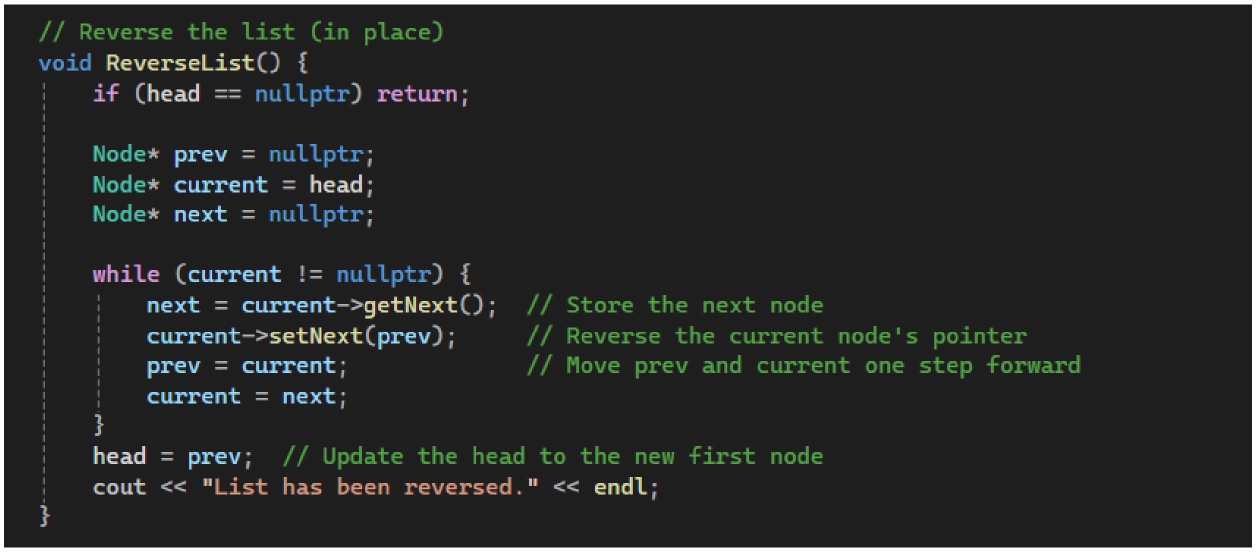


```
// Insert at the beginning
void InsertBeginning(Node* pNew) {
    if (head == nullptr) {
        head = pNew;
        head->setNext(head);
        head->setPrev(head);
    }
    else {
        Node* last = getLastNode();
        pNew->setNext(head);
        pNew->setPrev(last);
        head->setPrev(pNew);
        last->setNext(pNew);
        head = pNew;
    }
}

// Insert at the end
void InsertEnd(Node* pNew) {
    if (head == nullptr) {
        head = pNew;
        head->setNext(head);
        head->setPrev(head);
    }
    else {
        Node* last = getLastNode();
        pNew->setNext(head);
        pNew->setPrev(last);
        last->setNext(pNew);
        head->setPrev(pNew);
    }
}
```

- **Ques 1**



```
// Delete the second node
void DeleteSecondNode() {
    if (head == nullptr || head->getNext() == nullptr) {
        cout << "List doesn't have a second node to delete." << endl;
        return;
    }
    Node* second = head->getNext();
    head->setNext(second->getNext());
    delete second;
    cout << "Second node deleted." << endl;
}
```



```
// Reverse the list (in place)
void ReverseList() {
    if (head == nullptr) return;

    Node* prev = nullptr;
    Node* current = head;
    Node* next = nullptr;

    while (current != nullptr) {
        next = current->getNext();  // Store the next node
        current->setNext(prev);     // Reverse the current node's pointer
        prev = current;             // Move prev and current one step forward
        current = next;
    }
    head = prev;  // Update the head to the new first node
    cout << "List has been reversed." << endl;
}
```