Activity No. 3	
LINKED LISTS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/27/24**
Section: CPE21S4	**Date Submitted: 9/27/24**
Name(s): Dominic Joseph P. Virtucio	**Instructor: Maam Sayo**
6. Output	

Screenshot

```
C/C++
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    ListTraversal(head);

    return 0;
}
```

```
Output
/tmp/wfuC1PPjnK.o
C P E 0 1 0
```

Discussion

The output of the program will print the characters stored in each node of the linked list, which are: C P E 0 1 0. This is because each node contains a single character and is linked to the next node, forming a chain of values. The program traverses the linked list starting from head and prints each character until it reaches the last node, which points to nullptr.

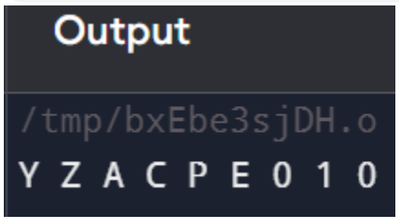
Table 3-1. Output of Initial/Simple Implementation

Operation	Screenshot

Traversal	C/C++
	`#include <iostream>`
	
	`struct Node {`
	`  char data;`
	`  Node* next;`
	`};`
	
	`void insertAtEnd(Node*& head, char newData) {`
	`  Node* newNode = new Node;`
	`  newNode->data = newData;`
	`  newNode->next = nullptr;`
	
	`  if (head == nullptr) {`
	`  head = newNode;`
	`  } else {`
	`  Node* temp = head;`
	`  while (temp->next != nullptr) {`
	`  temp = temp->next;`
	`  }`
	`  temp->next = newNode;`
	`  }`
	`}`
	
	`void ListTraversal(Node* n) {`
	`  while (n != nullptr) {`
	`  std::cout << n->data << " ";`
	`  n = n->next;`
	`  }`
	`  std::cout << std::endl;`
	`}`
	
	`int main() {`
	`  Node* head = nullptr;`
	
	`  insertAtEnd(head, 'C');`
	`  insertAtEnd(head, 'P');`
	`  insertAtEnd(head, 'E');`
	`  insertAtEnd(head, '0');`
	`  insertAtEnd(head, '1');`
	`  insertAtEnd(head, '0');`
	
	`  ListTraversal(head);`
	
	`  return 0;`
	`}`
	
	Output:
	Output
	/tmp/HeI3urN9kb.o
	C P E 0 1 0

Insertion at head	C/C++
	`#include <iostream>`
	
	`struct Node {`
	`  char data;`
	`  Node* next;`
	`};`
	
	`void insertAtEnd(Node*& head, char newData) {`
	`  Node* newNode = new Node;`
	`  newNode->data = newData;`
	`  newNode->next = nullptr;`
	
	`  if (head == nullptr) {`
	`  head = newNode;`
	`  } else {`
	`  Node* temp = head;`
	`  while (temp->next != nullptr) {`
	`  temp = temp->next;`
	`  }`
	`  temp->next = newNode;`
	`  }`
	`}`
	
	`void insertAtHead(Node*& head, char newData) {`
	`  Node* newNode = new Node;`
	`  newNode->data = newData;`
	`  newNode->next = head;`
	`  head = newNode;`
	`}`

```
void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAtHead(head, 'H');
    insertAtHead(head, 'A');

    ListTraversal(head);

    return 0;
}
```

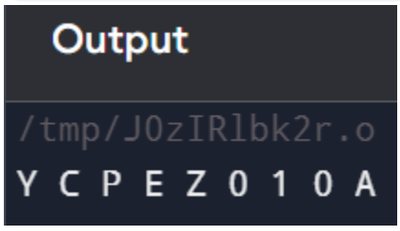


Insertion at any part of the list

C/C++

```
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void insertAtPosition(Node*& head, char newData, int position) {
    Node* newNode = new Node;
    newNode->data = newData;

    if (position == 0) {
        newNode->next = head;
        head = newNode;
    } else {
        Node* temp = head;
        for (int i = 0; i < position - 1 && temp != nullptr; i++) {
            temp = temp->next;
        }

        if (temp != nullptr) {
            newNode->next = temp->next;
            temp->next = newNode;
        } else {
            std::cout << "Position out of bounds" << std::endl;
            delete newNode;
        }
    }
```

```
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAtPosition(head, 'Z', 3);
    insertAtPosition(head, 'Y', 0);
    insertAtPosition(head, 'A', 8);

    ListTraversal(head);

    return 0;
}
```

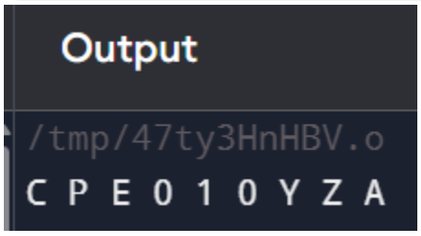


Insertion at the end

C/C++

```
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void insertAtPosition(Node*& head, char newData, int position) {
    Node* newNode = new Node;
    newNode->data = newData;

    if (position == 0) {
        newNode->next = head;
        head = newNode;
    } else {
        Node* temp = head;
```

```
        for (int i = 0; i < position - 1 && temp != nullptr; i++) {
            temp = temp->next;
        }

        if (temp != nullptr) {
            newNode->next = temp->next;
            temp->next = newNode;
        } else {
            std::cout << "Position out of bounds" << std::endl;
            delete newNode;
        }
    }
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAtPosition(head, 'Y', 6);
    insertAtPosition(head, 'Z', 7);
    insertAtPosition(head, 'A', 8);

    ListTraversal(head);

    return 0;
}
```



Deletion of a node

```
C/C++
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
```

```
    }
}

void deleteNode(Node*& head, char value) {
    if (head == nullptr) return;

    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* current = head;
    while (current->next != nullptr && current->next->data != value) {
        current = current->next;
    }

    if (current->next != nullptr) {
        Node* temp = current->next;
        current->next = current->next->next;
        delete temp;
    }
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    std::cout << "List before deletion: ";
    ListTraversal(head);

    deleteNode(head, 'E');
    deleteNode(head, '0');
    deleteNode(head, 'C');

    std::cout << "List after deletion: ";
    ListTraversal(head);

    return 0;
}
```

```
Output

/tmp/XdBXpdDMzJ.o
List before deletion: C P E 0 1 0
List after deletion: P 1 0
```

Table 3-2. Code for the List Operations

a.	**Source Code**	

C/C++

```
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    ListTraversal(head);

    return 0;
}
```

Console

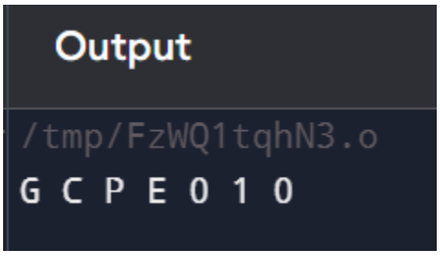
b. **Source Code**

C/C++

```
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
```

```
        temp->next = newNode;
    }
}

void insertAtHead(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAtHead(head, 'G');
    ListTraversal(head);

    return 0;
}
```

Console

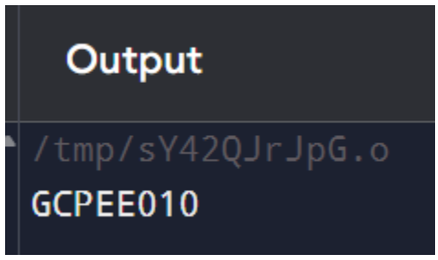


c. **Source Code**

```
C/C++
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void insertAfter(Node*& head, char prevData, char newData) {
    Node* temp = head;
    while (temp != nullptr && temp->data != prevData) {
        temp = temp->next;
    }
    if (temp != nullptr) {
        Node* newNode = new Node;
        newNode->data = newData;
        newNode->next = temp->next;
        temp->next = newNode;
    }
```

```
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data;
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'G');
    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAfter(head, 'P', 'E');
    insertAfter(head, 'P', 'E');

    ListTraversal(head);

    return 0;
}
```

Console



d. **Source Code**

```
C/C++
#include <iostream>

struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void insertAfter(Node*& head, char prevData, char newData) {
    Node* temp = head;
    while (temp != nullptr && temp->data != prevData) {
        temp = temp->next;
    }
    if (temp != nullptr) {
        Node* newNode = new Node;
```

```
        newNode->data = newData;
        newNode->next = temp->next;
        temp->next = newNode;
    }
}

void deleteNode(Node*& head, char key) {
    if (head == nullptr) return;

    if (head->data == key) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* current = head;
    Node* prev = nullptr;
    while (current != nullptr && current->data != key) {
        prev = current;
        current = current->next;
    }

    if (current == nullptr) return;

    prev->next = current->next;
    delete current;
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data;
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'G');
    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAfter(head, 'P', 'E');
    insertAfter(head, 'P', 'E');

    deleteNode(head, 'C');

    ListTraversal(head);

    return 0;
}
```

Console

e. **Source Code**

```
C/C++
#include <iostream>
```

```
struct Node {
    char data;
    Node* next;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void insertAfter(Node*& head, char prevData, char newData) {
    Node* temp = head;
    while (temp != nullptr && temp->data != prevData) {
        temp = temp->next;
    }
    if (temp != nullptr) {
        Node* newNode = new Node;
        newNode->data = newData;
        newNode->next = temp->next;
        temp->next = newNode;
    }
}

void deleteNode(Node*& head, char key) {
    if (head == nullptr) return;

    if (head->data == key) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* current = head;
    Node* prev = nullptr;
    while (current != nullptr && current->data != key) {
        prev = current;
        current = current->next;
    }

    if (current == nullptr) return;

    prev->next = current->next;
    delete current;
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data;
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'G');
    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
```

```
    insertAtEnd(head, '0');

    insertAfter(head, 'P', 'E');
    insertAfter(head, 'P', 'E');

    deleteNode(head, 'C');
    deleteNode(head, 'P');

    ListTraversal(head);

    return 0;
}
```

Console

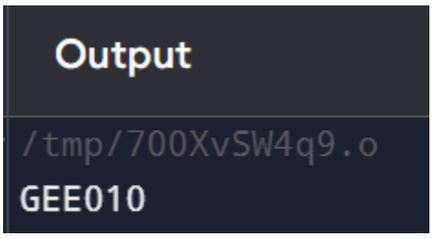


Table 3-3. Code and Analysis for Singly Linked Lists

Screenshots	Analysis
C/C++ `#include <iostream>`	The fundamental distinction between singly linked and doubly linked lists is their structure and traversal capabilities. In a singly linked list, each node holds simply

```
struct Node {
    char data;
    Node* next;
    Node* prev;

    Node(char newData) : data(newData), next(nullptr),
prev(nullptr) {}
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node(newData);

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

void ListTraversalReverse(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->prev;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    std::cout << "Traversal from head to tail: ";
    ListTraversal(head);

    Node* temp = head;
    while (temp && temp->next != nullptr) {
        temp = temp->next;
    }

    std::cout << "Traversal from tail to head: ";
    ListTraversalReverse(temp);

    return 0;
}
```

a pointer to the next node, simplifying operations like insertion and deletion while limiting traversal to one direction. Doubly linked lists, on the other hand, feature nodes that contain pointers to both the next and previous nodes, allowing for bidirectional traversal and simplifying certain operations, but at the expense of increased memory usage due to the additional pointers.

```
C/C++
#include <iostream>

struct Node {
    char data;
```

In my initial singly linked list implementation, I described the node structure as struct Node { char data; Node* next; }. This meant that each node could only point to the next node in the list, which limited my ability to go

```
    Node* next;
    Node* prev;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;
    newNode->prev = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

void insertAtHead(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = head;
    newNode->prev = nullptr;

    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

void ListReverseTraversal(Node* n) {
    if (n == nullptr) return;

    while (n->next != nullptr) {
        n = n->next;
    }

    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->prev;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAtHead(head, 'G');

    std::cout << "List: ";
    ListTraversal(head);

    std::cout << "Reverse List: ";
    ListReverseTraversal(head);
```

backwards. When I moved to a doubly linked list structure, I added a prev pointer: struct Node { char data; Node* next; Node* prev; }. This update allowed me to move through the list in both ways. I discovered that this bidirectional functionality makes it easier to manage the list and simplifies activities that need backward traversal.

```
    return 0;
}
```

b.

C/C++

```
#include <iostream>

struct Node {
    char data;
    Node* next;
    Node* prev;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;
    newNode->prev = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

void insertAfter(Node*& head, char prevData, char newData)
{
    Node* temp = head;
    while (temp != nullptr && temp->data != prevData) {
        temp = temp->next;
    }
    if (temp != nullptr) {
        Node* newNode = new Node;
        newNode->data = newData;
        newNode->next = temp->next;
        newNode->prev = temp;

        if (temp->next != nullptr) {
            temp->next->prev = newNode;
        }

        temp->next = newNode;
    }
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data << " ";
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'G');
    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');
```

```
    insertAfter(head, 'P', 'E');
    insertAfter(head, 'P', 'E');

    ListTraversal(head);

    return 0;
}
```

c.

C/C++

```
#include <iostream>

struct Node {
    char data;
    Node* next;
    Node* prev;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;
    newNode->prev = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

void insertAfter(Node*& head, char prevData, char newData)
{
    Node* temp = head;
    while (temp != nullptr && temp->data != prevData) {
        temp = temp->next;
    }
    if (temp != nullptr) {
        Node* newNode = new Node;
        newNode->data = newData;
        newNode->next = temp->next;
        newNode->prev = temp;

        if (temp->next != nullptr) {
            temp->next->prev = newNode;
        }

        temp->next = newNode;
    }
}

void deleteNode(Node*& head, char key) {
    if (head == nullptr) return;

    if (head->data == key) {
        Node* temp = head;
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
        delete temp;
        return;
    }

    Node* current = head;
```

When I implemented insertion in my singly linked list, I discovered that inserting nodes at any location was inefficient because I could only change the 'next' pointer. Transitioning to a doubly linked list enabled me to use both 'next' and 'prev' pointers, making it easy to insert nodes at any time. This new solution kept the list structure intact by guaranteeing that all links were properly changed. I discovered that handling numerous pointers improves the flexibility and efficiency of data operations, hence simplifying the insertion process overall.

```
    while (current != nullptr && current->data != key) {
        current = current->next;
    }

    if (current == nullptr) return;

    if (current->next != nullptr) {
        current->next->prev = current->prev;
    }
    if (current->prev != nullptr) {
        current->prev->next = current->next;
    }
    delete current;
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data;
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'G');
    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAfter(head, 'P', 'E');
    insertAfter(head, 'P', 'E');

    deleteNode(head, 'C');

    ListTraversal(head);

    return 0;
}
```

d.

```
C/C++
#include <iostream>

struct Node {
    char data;
    Node* next;
    Node* prev;
};

void insertAtEnd(Node*& head, char newData) {
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;
    newNode->prev = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
```

The deleteNode method in my singly linked list implementation simply changed the next pointers. If a node was removed, there may be inconsistencies, particularly in the middle of the list. When I changed this method for the doubly linked list, I discovered that deleting a node updates both the next and prior pointers. This modification secured the list's integrity, preventing potential problems caused by dangling pointers. This experience taught me the value of proper pointer management in linked list operations.

```
}

void insertAfter(Node*& head, char prevData, char newData)
{
    Node* temp = head;
    while (temp != nullptr && temp->data != prevData) {
        temp = temp->next;
    }
    if (temp != nullptr) {
        Node* newNode = new Node;
        newNode->data = newData;
        newNode->next = temp->next;
        newNode->prev = temp;

        if (temp->next != nullptr) {
            temp->next->prev = newNode;
        }

        temp->next = newNode;
    }
}

void deleteNode(Node*& head, char key) {
    if (head == nullptr) return;

    if (head->data == key) {
        Node* temp = head;
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
        delete temp;
        return;
    }

    Node* current = head;
    while (current != nullptr && current->data != key) {
        current = current->next;
    }

    if (current == nullptr) return;

    if (current->next != nullptr) {
        current->next->prev = current->prev;
    }
    if (current->prev != nullptr) {
        current->prev->next = current->next;
    }
    delete current;
}

void ListTraversal(Node* n) {
    while (n != nullptr) {
        std::cout << n->data;
        n = n->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 'G');
    insertAtEnd(head, 'C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    insertAfter(head, 'P', 'E');
    insertAfter(head, 'P', 'E');

    deleteNode(head, 'C');
```

```
    deleteNode(head, 'P');

    ListTraversal(head);

    return 0;
}
```

e.

7. Supplementary Activity

C/C++

```
#include <iostream>
#include <string>
using namespace std;

class Track {
public:
    string name;
    Track* next;
    Track(const string& trackName) : name(trackName), next(nullptr) {}
};

class MusicPlaylist {
private:
    Track* head;

public:
    MusicPlaylist() : head(nullptr) {}

    void addTrack(const string& trackName) {
        Track* newTrack = new Track(trackName);
        if (!head) {
            head = newTrack;
            newTrack->next = head;
        } else {
            Track* current = head;
            while (current->next != head) current = current->next;
            current->next = newTrack;
            newTrack->next = head;
        }
        cout << "Added: " << trackName << endl;
    }

    void deleteTrack(const string& trackName) {
        if (!head) { cout << "Empty playlist." << endl; return; }
        Track *current = head, *prev = nullptr;
        do {
            if (current->name == trackName) {
                if (prev) prev->next = current->next;
                else {
                    Track* last = head;
                    while (last->next != head) last = last->next;
                    last->next = current->next;
                    head = current->next;
                }
                delete current;
                cout << "Removed: " << trackName << endl;
                return;
            }
            prev = current;
            current = current->next;
        } while (current != head);
        cout << "Not found: " << trackName << endl;
    }

    void playAll() {
        if (!head) { cout << "Empty playlist." << endl; return; }
```

```
        Track* current = head;
        do {
            cout << "Playing: " << current->name << endl;
            current = current->next;
        } while (current != head);
    }
};

int main() {
    MusicPlaylist playlist;
    playlist.addTrack("Love Story");
    playlist.addTrack("Shake It Off");
    playlist.addTrack("Blank Space");
    playlist.addTrack("You Belong with Me");
    playlist.addTrack("Cardigan");

    playlist.playAll();
    playlist.deleteTrack("Blank Space");
    playlist.playAll();

    return 0;
}
```

Output

```
/tmp/qVHhNxBflG.o
Added: Love Story
Added: Shake It Off
Added: Blank Space
Added: You Belong with Me
Added: Cardigan
Playing: Love Story
Playing: Shake It Off
Playing: Blank Space
Playing: You Belong with Me
Playing: Cardigan
Removed: Blank Space
Playing: Love Story
Playing: Shake It Off
Playing: You Belong with Me
Playing: Cardigan
```

8. Conclusion

I developed a better grasp of linked lists and their practical applications, especially while creating a circular linked list for a song playlist. The technique entailed developing and implementing functions for managing the linked list, such as adding and removing songs, playing all songs, and navigating between them. Each function necessitated careful analysis of how nodes are connected and how to maintain the circular structure, which strengthened my understanding of pointer manipulation and memory management in C++. The additional activity was useful in demonstrating the actual uses of data structures and giving students hands-on experience with coding and debugging. Although I believe I did quite well, I had problems owing to my lack of solid core knowledge in C++, and I hope to enhance my comprehension of them.

9. Assessment Rubric