Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Doubly Linked List

Q1: Write a program to delete the first node in a doubly linked list.

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

struct Node {
    int data;
    Node* prev = NULL;
    Node* next = NULL;
};

Node* head = nullptr;

void delFNode() {
    if (head == nullptr) {
        cout << "Empty!" << endl;
        return;
    }

    Node* temp = head;
    head = head->next;

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

    delete temp;
    cout << "First node deleted." << endl;
}

void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
    Node* temp = head;
    cout << "Doubly Linked List: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

```
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->prev = nullptr;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
        return;
    }

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

int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);

    printList();

    delFNode();
    printList();

    return 0;
}
```

Output:

```
Doubly Linked List: 10 20 30
First node deleted.
Doubly Linked List: 20 30

--------------------------------
Process exited after 0.4427 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Q2: How can you delete the last node in a doubly linked list? Write the code.

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

struct Node {
    int data;
    Node* prev = nullptr;
    Node* next = nullptr;
};

Node* head = nullptr; // Global head pointer

void delLastNode() {
    if (head == nullptr) { // If the list is empty
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    if (head->next == nullptr) { // If there's only one node
        delete head;
        head = nullptr;
        cout << "The last node (and only node) was deleted." << endl;
        return;
    }

    // Traverse to the last node
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    // Update the previous node's next pointer
    temp->prev->next = nullptr;

    delete temp; // Delete the last node
    cout << "Last node deleted." << endl;
}

// Function to print the doubly linked list
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
```

```
    cout << "Doubly Linked List: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to append a node to the end of the list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

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

int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);

    cout << "Initial list:" << endl;
    printList();

    cout << "Deleting the last node..." << endl;
    delLastNode();
    printList();

    cout << "Deleting the last node again..." << endl;
    delLastNode();
    printList();

    cout << "Deleting the last node again..." << endl;
    delLastNode();
    printList();

    return 0;
}
```

Output:

```
Initial list:
Doubly Linked List: 10 20 30
Deleting the last node...
Last node deleted.
Doubly Linked List: 10 20
Deleting the last node again...
Last node deleted.
Doubly Linked List: 10
Deleting the last node again...
The last node (and only node) was deleted.
The list is empty.

--------------------------------
Process exited after 0.4158 seconds with return value 0
Press any key to continue . . .
```

Q3: Write code to delete a node by its value in a doubly linked list.

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

struct Node {
    int data;
    Node* prev = nullptr;
    Node* next = nullptr;
};

Node* head = nullptr; // Global head pointer

// Function to delete a node by value
void deleteNodeByValue(int value) {
    if (head == nullptr) { // If the list is empty
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    Node* temp = head;

    // Traverse the list to find the node with the given value
    while (temp != nullptr && temp->data != value) {
        temp = temp->next;
    }

    if (temp == nullptr) { // Node with the given value not found
        cout << "Value " << value << " not found in the list." << endl;
        return;
    }

    // Node is the head
    if (temp == head) {
```

```
        head = temp->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
    }
    // Node is in the middle or end
    else {
        if (temp->prev != nullptr) {
            temp->prev->next = temp->next;
        }
        if (temp->next != nullptr) {
            temp->next->prev = temp->prev;
        }
    }

    delete temp; // Free memory
    cout << "Node with value " << value << " deleted." << endl;
}

// Function to print the doubly linked list
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Doubly Linked List: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to append a node to the end of the list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        return;
    }

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

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

int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);

    cout << "Initial list:" << endl;
    printList();

    cout << "Deleting node with value 20..." << endl;
    deleteNodeByValue(20);
    printList();

    cout << "Deleting node with value 40..." << endl;
    deleteNodeByValue(40);
    printList();

    cout << "Deleting node with value 10..." << endl;
    deleteNodeByValue(10);
    printList();

    cout << "Deleting node with value 50 (not in list)..." << endl;
    deleteNodeByValue(50);
    printList();

    return 0;
}
```

Output:

```
Initial list:
Doubly Linked List: 10 20 30 40
Deleting node with value 20...
Node with value 20 deleted.
Doubly Linked List: 10 30 40
Deleting node with value 40...
Node with value 40 deleted.
Doubly Linked List: 10 30
Deleting node with value 10...
Node with value 10 deleted.
Doubly Linked List: 30
Deleting node with value 50 (not in list)...
Value 50 not found in the list.
Doubly Linked List: 30

--------------------------------
Process exited after 0.4129 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Q4: How would you delete a node at a specific position in a doubly linked list? Show it in code.

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

struct Node {
    int data;
    Node* prev = nullptr;
    Node* next = nullptr;
};

Node* head = nullptr; // Global head pointer

// Function to delete a node at a specific position
void deleteNodeAtPosition(int position) {
    if (head == nullptr) { // Check if the list is empty
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    if (position <= 0) { // Invalid position
        cout << "Invalid position. Please provide a position greater than
0." << endl;
        return;
    }

    Node* temp = head;

    // Traverse to the node at the given position
    for (int i = 1; temp != nullptr && i < position; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) { // Position is beyond the length of the list
        cout << "Position " << position << " is out of range." << endl;
        return;
    }

    // If the node to delete is the head
    if (temp == head) {
        head = temp->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
    }
    // If the node to delete is in the middle or end
```

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

    delete temp; // Delete the node
    cout << "Node at position " << position << " deleted." << endl;
}

// Function to print the doubly linked list
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Doubly Linked List: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to append a node to the end of the list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        return;
    }

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

```
int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);

    cout << "Initial list:" << endl;
    printList();

    cout << "Deleting node at position 2..." << endl;
    deleteNodeAtPosition(2);
    printList();

    cout << "Deleting node at position 3..." << endl;
    deleteNodeAtPosition(3);
    printList();

    cout << "Deleting node at position 1..." << endl;
    deleteNodeAtPosition(1);
    printList();

    cout << "Deleting node at position 5 (out of range)..." << endl;
    deleteNodeAtPosition(5);
    printList();

    return 0;
}
```

Output:

```
Initial list:
Doubly Linked List: 10 20 30 40
Deleting node at position 2...
Node at position 2 deleted.
Doubly Linked List: 10 30 40
Deleting node at position 3...
Node at position 3 deleted.
Doubly Linked List: 10 30
Deleting node at position 1...
Node at position 1 deleted.
Doubly Linked List: 30
Deleting node at position 5 (out of range)...
Position 5 is out of range.
Doubly Linked List: 30

--------------------------------
Process exited after 0.486 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

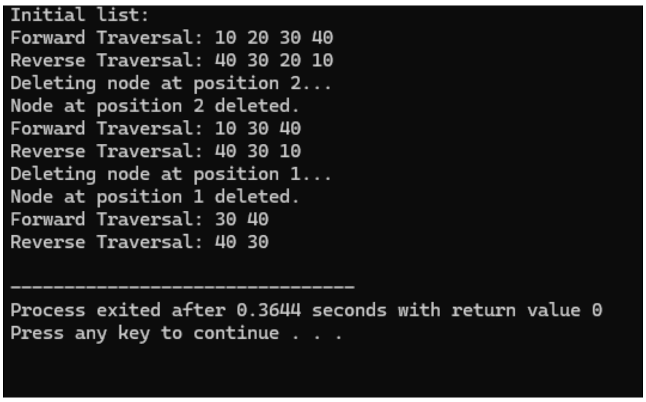
Q5: After deleting a node, how will you write the forward and reverse traversal functions?

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

struct Node {
    int data;
    Node* prev = nullptr;
    Node* next = nullptr;
};

Node* head = nullptr; // Global head pointer

// Forward Traversal
void forwardTraversal() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Forward Traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Reverse Traversal
void reverseTraversal() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    // Move to the last node
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    // Traverse backward
    cout << "Reverse Traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
```

```
        temp = temp->prev;
    }
    cout << endl;
}

// Append a new node to the end of the list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        return;
    }

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

// Delete a node at a specific position
void deleteNodeAtPosition(int position) {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    if (position <= 0) {
        cout << "Invalid position. Please provide a position greater than
0." << endl;
        return;
    }

    Node* temp = head;

    // Traverse to the node at the given position
    for (int i = 1; temp != nullptr && i < position; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Position " << position << " is out of range." << endl;
        return;
    }
```

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

    delete temp;
    cout << "Node at position " << position << " deleted." << endl;
}

int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);

    cout << "Initial list:" << endl;
    forwardTraversal();
    reverseTraversal();

    cout << "Deleting node at position 2..." << endl;
    deleteNodeAtPosition(2);
    forwardTraversal();
    reverseTraversal();

    cout << "Deleting node at position 1..." << endl;
    deleteNodeAtPosition(1);
    forwardTraversal();
    reverseTraversal();

    return 0;
}
```

Output:



```
Initial list:
Forward Traversal: 10 20 30 40
Reverse Traversal: 40 30 20 10
Deleting node at position 2...
Node at position 2 deleted.
Forward Traversal: 10 30 40
Reverse Traversal: 40 30 10
Deleting node at position 1...
Node at position 1 deleted.
Forward Traversal: 30 40
Reverse Traversal: 40 30

--------------------------------
Process exited after 0.3644 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Circular Linked List

Q1: Write a program to delete the first node in a circular linked list.

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

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

Node* head = nullptr; // Global head pointer for the circular linked list

// Function to delete the first node in the circular linked list
void deleteFirstNode() {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    // If there's only one node
    if (head->next == head) {
        delete head;
        head = nullptr;
        cout << "First node deleted. The list is now empty." << endl;
        return;
    }

    // Otherwise, find the last node
    Node* last = head;
    while (last->next != head) {
        last = last->next;
    }

    // Update the head and last node's next pointer
    Node* temp = head;
    head = head->next;
    last->next = head;

    delete temp;
    cout << "First node deleted." << endl;
}

// Function to append a node to the circular linked list
void appendNode(int value) {
```

```
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        head->next = head; // Point to itself
        return;
    }

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

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

// Function to print the circular linked list
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Circular Linked List: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

// Main function
int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);

    cout << "Initial list:" << endl;
    printList();

    cout << "Deleting the first node..." << endl;
    deleteFirstNode();
    printList();
```

```
    cout << "Deleting the first node again..." << endl;
    deleteFirstNode();
    printList();

    return 0;
}
```

Output:

```
Initial list:
Circular Linked List: 10 20 30 40
Deleting the first node...
First node deleted.
Circular Linked List: 20 30 40
Deleting the first node again...
First node deleted.
Circular Linked List: 30 40

--------------------------------
Process exited after 0.4458 seconds with return value 0
Press any key to continue . . .
```

Q2: How can you delete the last node in a circular linked list? Write the code.

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

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

Node* head = nullptr; // Global head pointer for the circular linked list

// Function to delete the last node in the circular linked list
void deleteLastNode() {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    // If there's only one node
    if (head->next == head) {
        delete head;
        head = nullptr;
        cout << "Last node deleted. The list is now empty." << endl;
        return;
    }

    // Traverse to find the second-last node
```

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

    // Delete the last node and update the second-last node's next pointer
    Node* lastNode = temp->next;
    temp->next = head;
    delete lastNode;

    cout << "Last node deleted." << endl;
}

// Function to append a node to the circular linked list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        head->next = head; // Point to itself
        return;
    }

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

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

// Function to print the circular linked list
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Circular Linked List: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
```

```
}

// Main function
int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);

    cout << "Initial list:" << endl;
    printList();

    cout << "Deleting the last node..." << endl;
    deleteLastNode();
    printList();

    cout << "Deleting the last node again..." << endl;
    deleteLastNode();
    printList();

    cout << "Deleting the last node again..." << endl;
    deleteLastNode();
    printList();

    cout << "Deleting the last node again..." << endl;
    deleteLastNode();
    printList();

    return 0;
}
```

Output:

```
Initial list:
Circular Linked List: 10 20 30 40
Deleting the last node...
Last node deleted.
Circular Linked List: 10 20 30
Deleting the last node again...
Last node deleted.
Circular Linked List: 10 20
Deleting the last node again...
Last node deleted.
Circular Linked List: 10
Deleting the last node again...
Last node deleted. The list is now empty.
The list is empty.

--------------------------------
Process exited after 0.3925 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

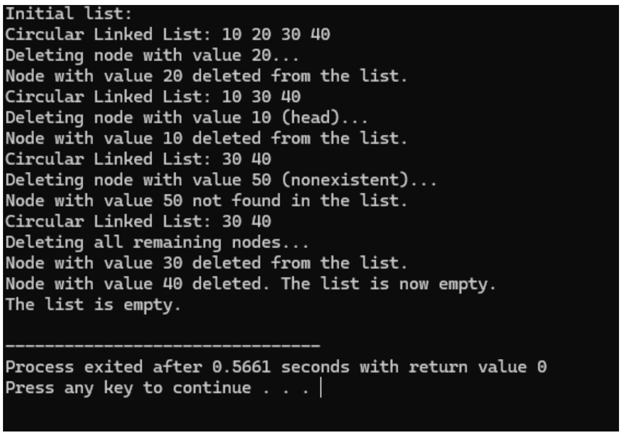
Q3: Write a function to delete a node by its value in a circular linked list.

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

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

Node* head = nullptr; // Global head pointer for the circular linked list

// Function to delete a node by its value in the circular linked list
void deleteNodeByValue(int value) {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    Node* current = head;
    Node* previous = nullptr;

    // Handle the case where the node to delete is the head
    if (head->data == value) {
        // If there's only one node in the list
        if (head->next == head) {
            delete head;
            head = nullptr;
            cout << "Node with value " << value << " deleted. The list is
now empty." << endl;
            return;
        }

        // Otherwise, find the last node
        Node* last = head;
        while (last->next != head) {
            last = last->next;
        }

        // Update head and adjust the circular link
        last->next = head->next;
        Node* temp = head;
        head = head->next;
        delete temp;

        cout << "Node with value " << value << " deleted from the list." <<
endl;
```

```
        return;
    }

    // Traverse the list to find the node to delete
    do {
        previous = current;
        current = current->next;

        if (current->data == value) {
            previous->next = current->next;
            delete current;
            cout << "Node with value " << value << " deleted from the list."
<< endl;
            return;
        }
    } while (current != head);

    cout << "Node with value " << value << " not found in the list." << endl;
}

// Function to append a node to the circular linked list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        head->next = head; // Point to itself
        return;
    }

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

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

// Function to print the circular linked list
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
```

```
    Node* temp = head;
    cout << "Circular Linked List: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

// Main function
int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);

    cout << "Initial list:" << endl;
    printList();

    cout << "Deleting node with value 20..." << endl;
    deleteNodeByValue(20);
    printList();

    cout << "Deleting node with value 10 (head)..." << endl;
    deleteNodeByValue(10);
    printList();

    cout << "Deleting node with value 50 (nonexistent)..." << endl;
    deleteNodeByValue(50);
    printList();

    cout << "Deleting all remaining nodes..." << endl;
    deleteNodeByValue(30);
    deleteNodeByValue(40);
    printList();

    return 0;
}
```

Output:



Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Q4: How will you delete a node at a specific position in a circular linked list? Write code for it

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

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

Node* head = nullptr; // Global head pointer for the circular linked list

// Function to delete a node at a specific position in the circular linked
list
void deleteNodeAtPosition(int position) {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    // If the position is 1 (deleting the head node)
    if (position == 1) {
        Node* temp = head;

        // If there is only one node in the list
        if (head->next == head) {
            head = nullptr;
            delete temp;
            cout << "Node at position 1 deleted. The list is now empty." <<
endl;
            return;
        }

        // Find the last node to update its next pointer
        Node* last = head;
        while (last->next != head) {
            last = last->next;
        }

        // Update head and adjust the circular link
        last->next = head->next;
        head = head->next;
        delete temp;

        cout << "Node at position 1 deleted." << endl;
        return;
```

```
    }

    // Traverse to the node at the given position
    Node* current = head;
    Node* previous = nullptr;
    int count = 1;

    do {
        previous = current;
        current = current->next;
        count++;
    } while (current != head && count < position);

    // If the position is out of range
    if (count < position) {
        cout << "Position out of range. No node deleted." << endl;
        return;
    }

    // Adjust the links and delete the node
    previous->next = current->next;
    delete current;

    cout << "Node at position " << position << " deleted." << endl;
}

// Function to append a node to the circular linked list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        head->next = head; // Point to itself
        return;
    }

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

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

// Function to print the circular linked list
```

```
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Circular Linked List: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

// Main function
int main() {
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);
    appendNode(50);

    cout << "Initial list:" << endl;
    printList();

    cout << "Deleting node at position 3..." << endl;
    deleteNodeAtPosition(3);
    printList();

    cout << "Deleting node at position 1 (head)..." << endl;
    deleteNodeAtPosition(1);
    printList();

    cout << "Deleting node at position 5 (out of range)..." << endl;
    deleteNodeAtPosition(5);
    printList();

    cout << "Deleting all remaining nodes..." << endl;
    deleteNodeAtPosition(1);
    deleteNodeAtPosition(1);
    deleteNodeAtPosition(1);
    printList();

    return 0;
}
```

Output:

```
Initial list:
Circular Linked List: 10 20 30 40 50
Deleting node at position 3...
Node at position 3 deleted.
Circular Linked List: 10 20 40 50
Deleting node at position 1 (head)...
Node at position 1 deleted.
Circular Linked List: 20 40 50
Deleting node at position 5 (out of range)...
Position out of range. No node deleted.
Circular Linked List: 20 40 50
Deleting all remaining nodes...
Node at position 1 deleted.
Node at position 1 deleted.
Node at position 1 deleted. The list is now empty.
The list is empty.

--------------------------------
Process exited after 0.4162 seconds with return value 0
Press any key to continue . . .
```

Q5: Write a program to show forward traversal after deleting a node in a circular linked list.

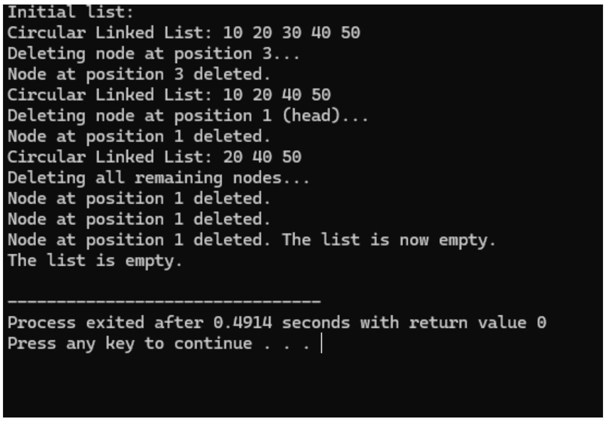
```
#include <iostream>
using namespace std;

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

Node* head = nullptr; // Global head pointer for the circular linked list

// Function to delete a node at a specific position in the circular linked
list
void deleteNodeAtPosition(int position) {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }

    // If the position is 1 (deleting the head node)
    if (position == 1) {
        Node* temp = head;

        // If there is only one node in the list
        if (head->next == head) {
            head = nullptr;
            delete temp;
            cout << "Node at position 1 deleted. The list is now empty." <<
endl;
            return;
        }
```

```
        // Find the last node to update its next pointer
        Node* last = head;
        while (last->next != head) {
            last = last->next;
        }

        // Update head and adjust the circular link
        last->next = head->next;
        head = head->next;
        delete temp;

        cout << "Node at position 1 deleted." << endl;
        return;
    }

    // Traverse to the node at the given position
    Node* current = head;
    Node* previous = nullptr;
    int count = 1;

    do {
        previous = current;
        current = current->next;
        count++;
    } while (current != head && count < position);

    // If the position is out of range
    if (count < position) {
        cout << "Position out of range. No node deleted." << endl;
        return;
    }

    // Adjust the links and delete the node
    previous->next = current->next;
    delete current;

    cout << "Node at position " << position << " deleted." << endl;
}

// Function to append a node to the circular linked list
void appendNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
```

```
        head->next = head; // Point to itself
        return;
    }

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

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

// Function to print the circular linked list (forward traversal)
void printList() {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Circular Linked List: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

// Main function
int main() {
    // Append nodes to the circular linked list
    appendNode(10);
    appendNode(20);
    appendNode(30);
    appendNode(40);
    appendNode(50);

    cout << "Initial list:" << endl;
    printList();

    // Delete node at position 3
    cout << "Deleting node at position 3..." << endl;
    deleteNodeAtPosition(3);
    printList();

    // Delete node at position 1 (head)
```

```
    cout << "Deleting node at position 1 (head)..." << endl;
    deleteNodeAtPosition(1);
    printList();

    // Delete all remaining nodes
    cout << "Deleting all remaining nodes..." << endl;
    deleteNodeAtPosition(1);
    deleteNodeAtPosition(1);
    deleteNodeAtPosition(1);
    printList();

    return 0;
}
```

Output:



```
Initial list:
Circular Linked List: 10 20 30 40 50
Deleting node at position 3...
Node at position 3 deleted.
Circular Linked List: 10 20 40 50
Deleting node at position 1 (head)...
Node at position 1 deleted.
Circular Linked List: 20 40 50
Deleting all remaining nodes...
Node at position 1 deleted.
Node at position 1 deleted.
Node at position 1 deleted. The list is now empty.
The list is empty.

--------------------------------
Process exited after 0.4914 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Binary Search Tree

Q1: Write a program to count all the nodes in a binary search tree.

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

// Define the structure of a node in the BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a node in the BST
Node* insertNode(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value); // Create a new node if the tree is empty
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value); // Insert in the left
subtree
    } else if (value > root->data) {
        root->right = insertNode(root->right, value); // Insert in the right
subtree
    }
    return root;
}

// Function to count all nodes in the BST
int countNodes(Node* root) {
    if (root == nullptr) {
        return 0; // Base case: if the tree is empty, return 0
    }
    // Count the current node + nodes in the left and right subtrees
    return 1 + countNodes(root->left) + countNodes(root->right);
}

// Function to display the BST in-order (for debugging and visualization)
void inOrderTraversal(Node* root) {
```

```
    if (root == nullptr) {
        return;
    }
    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Display the tree (in-order traversal)
    cout << "In-order traversal of the BST: ";
    inOrderTraversal(root);
    cout << endl;

    // Count all nodes in the BST
    int totalNodes = countNodes(root);
    cout << "Total number of nodes in the BST: " << totalNodes << endl;

    return 0;
}
```

Output:

```
In-order traversal of the BST: 20 30 40 50 60 70 80
Total number of nodes in the BST: 7

--------------------------------
Process exited after 0.3332 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Q2: How can you search for a specific value in a binary search tree? Write the code.

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

// Define the structure of a node in the BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a node in the BST
Node* insertNode(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value); // Create a new node if the tree is empty
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value); // Insert in the left
subtree
    } else if (value > root->data) {
        root->right = insertNode(root->right, value); // Insert in the right
subtree
    }
    return root;
}

// Function to search for a specific value in the BST
bool search(Node* root, int value) {
    // If the root is null or the value is found
    if (root == nullptr) {
        return false; // Value not found
    }

    // If the value is equal to the root's data, return true
    if (root->data == value) {
        return true;
    }

    // If the value is smaller, search the left subtree
```

```
    if (value < root->data) {
        return search(root->left, value);
    }

    // If the value is greater, search the right subtree
    return search(root->right, value);
}

// Function to display the BST in-order (for debugging and visualization)
void inOrderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }
    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Display the tree (in-order traversal)
    cout << "In-order traversal of the BST: ";
    inOrderTraversal(root);
    cout << endl;

    // Search for a specific value in the BST
    int valueToSearch = 40;
    if (search(root, valueToSearch)) {
        cout << "Value " << valueToSearch << " found in the BST." << endl;
    } else {
        cout << "Value " << valueToSearch << " not found in the BST." <<
endl;
    }

    valueToSearch = 100;
    if (search(root, valueToSearch)) {
        cout << "Value " << valueToSearch << " found in the BST." << endl;
```

```
    } else {
        cout << "Value " << valueToSearch << " not found in the BST." <<
endl;
    }

    return 0;
}
```

Output:

```
In-order traversal of the BST: 20 30 40 50 60 70 80
Value 40 found in the BST.
Value 100 not found in the BST.

--------------------------------
Process exited after 0.4334 seconds with return value 0
Press any key to continue . . .
```

Q3: Write code to traverse a binary search tree in in-order, pre-order, and post-order.

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

// Define the structure of a node in the BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a node in the BST
Node* insertNode(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value); // Create a new node if the tree is empty
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value); // Insert in the left
subtree
    } else if (value > root->data) {
        root->right = insertNode(root->right, value); // Insert in the
right subtree
    }
    return root;
}
```

```
// In-order traversal: Left, Root, Right
void inOrderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }
    inOrderTraversal(root->left);    // Traverse left subtree
    cout << root->data << " ";       // Visit root
    inOrderTraversal(root->right);   // Traverse right subtree
}

// Pre-order traversal: Root, Left, Right
void preOrderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }
    cout << root->data << " ";       // Visit root
    preOrderTraversal(root->left);   // Traverse left subtree
    preOrderTraversal(root->right);  // Traverse right subtree
}

// Post-order traversal: Left, Right, Root
void postOrderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }
    postOrderTraversal(root->left);  // Traverse left subtree
    postOrderTraversal(root->right); // Traverse right subtree
    cout << root->data << " ";       // Visit root
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Display the tree using different traversal methods
    cout << "In-order traversal: ";
    inOrderTraversal(root);
    cout << endl;
```

```
    cout << "Pre-order traversal: ";
    preOrderTraversal(root);
    cout << endl;

    cout << "Post-order traversal: ";
    postOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:

```
In-order traversal: 20 30 40 50 60 70 80
Pre-order traversal: 50 30 20 40 70 60 80
Post-order traversal: 20 40 30 60 80 70 50

--------------------------------
Process exited after 0.3874 seconds with return value 0
Press any key to continue . . .
```

Q4: How will you write reverse in-order traversal for a binary search tree? Show it in code.

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

// Define the structure of a node in the BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a node in the BST
Node* insertNode(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value); // Create a new node if the tree is empty
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value); // Insert in the left
subtree
    } else if (value > root->data) {
        root->right = insertNode(root->right, value); // Insert in the
right subtree
```

```
    }
    return root;
}

// Reverse in-order traversal: Right, Root, Left
void reverseInOrderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }
    reverseInOrderTraversal(root->right);  // Traverse right subtree
    cout << root->data << " ";              // Visit root
    reverseInOrderTraversal(root->left);   // Traverse left subtree
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Display the tree using reverse in-order traversal
    cout << "Reverse in-order traversal: ";
    reverseInOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:

```
Reverse in-order traversal: 80 70 60 50 40 30 20

--------------------------------
Process exited after 0.3217 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Q5: Write a program to check if there are duplicate values in a binary search tree.

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

// Define the structure of a node in the BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a node in the BST
Node* insertNode(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value); // Create a new node if the tree is empty
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value); // Insert in the left
subtree
    } else if (value > root->data) {
        root->right = insertNode(root->right, value); // Insert in the
right subtree
    }
    return root;
}

// Function to check if there are duplicates in the BST using in-order
traversal
bool checkDuplicates(Node* root, int& prevValue) {
    if (root == nullptr) {
        return false;
    }

    // Check the left subtree
    if (checkDuplicates(root->left, prevValue)) {
        return true;
    }

    // Check if current node is equal to the previous node
    if (root->data == prevValue) {
```

```
        return true; // Duplicate found
    }

    // Update the previous value to the current node's value
    prevValue = root->data;

    // Check the right subtree
    return checkDuplicates(root->right, prevValue);
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Insert a duplicate value
    root = insertNode(root, 40);  // Duplicate node with value 40

    int prevValue = -1;  // Initialize to a value not present in the tree
    if (checkDuplicates(root, prevValue)) {
        cout << "Duplicate values found in the BST!" << endl;
    } else {
        cout << "No duplicate values in the BST." << endl;
    }

    return 0;
}
```

Output:

```
No duplicate values in the BST.

--------------------------------
Process exited after 0.3537 seconds with return value 0
Press any key to continue . . .
```

Name: Wishaq Akbar **Reg #:** 2023-BS-AI-041

Q6: How can you delete a node from a binary search tree? Write code for deleting a leaf, a node with one child, and a node with two children.

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

// Define the structure of a node in the BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a node in the BST
Node* insertNode(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value); // Create a new node if the tree is empty
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value); // Insert in the left
subtree
    } else if (value > root->data) {
        root->right = insertNode(root->right, value); // Insert in the
right subtree
    }
    return root;
}

// Function to find the minimum value node in the tree
Node* findMin(Node* root) {
    while (root && root->left != nullptr) {
        root = root->left;
    }
    return root;
}

// Function to delete a node from the BST
Node* deleteNode(Node* root, int value) {
    if (root == nullptr) {
        return root; // If tree is empty, return NULL
    }
```

```
    // If value to be deleted is smaller than the root's data
    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    }
    // If value to be deleted is greater than the root's data
    else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    }
    // Node to be deleted is found
    else {
        // Case 1: Node has no children (Leaf node)
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
            root = nullptr;
        }
        // Case 2: Node has one child
        else if (root->left == nullptr) {
            Node* temp = root;
            root = root->right;
            delete temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root;
            root = root->left;
            delete temp;
        }
        // Case 3: Node has two children
        else {
            // Find the minimum node in the right subtree (in-order
successor)
            Node* temp = findMin(root->right);

            // Copy the inorder successor's content to this node
            root->data = temp->data;

            // Delete the in-order successor
            root->right = deleteNode(root->right, temp->data);
        }
    }
    return root;
}

// Function to perform in-order traversal
void inOrderTraversal(Node* root) {
    if (root == nullptr) {
        return;
```

```
    }
    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    cout << "In-order traversal before deletion: ";
    inOrderTraversal(root);
    cout << endl;

    // Delete a node (leaf node)
    root = deleteNode(root, 20); // Deleting a leaf node (20)
    cout << "In-order traversal after deleting leaf node 20: ";
    inOrderTraversal(root);
    cout << endl;

    // Delete a node with one child
    root = deleteNode(root, 30); // Deleting a node with one child (30)
    cout << "In-order traversal after deleting node with one child 30: ";
    inOrderTraversal(root);
    cout << endl;

    // Delete a node with two children
    root = deleteNode(root, 50); // Deleting a node with two children (50)
    cout << "In-order traversal after deleting node with two children 50:
";
    inOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:

```
In-order traversal before deletion: 20 30 40 50 60 70 80
In-order traversal after deleting leaf node 20: 30 40 50 60 70 80
In-order traversal after deleting node with one child 30: 40 50 60 70 80
In-order traversal after deleting node with two children 50: 40 60 70 80

--------------------------------
Process exited after 0.3833 seconds with return value 0
Press any key to continue . . .
```