DS ASSIGNMENT (032)

SUBMITTED TO MAM IRSHA

Data Structures

Final assignment

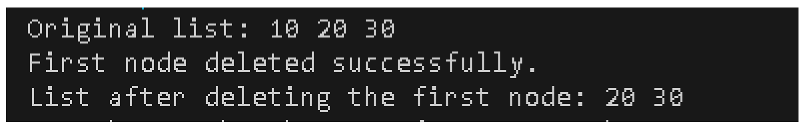
Q1: Write a program to delete the first node in a doubly linked list.

```
#include <iostream>
using namespace std;
struct Node
{
    int data;
    Node *prev;
    Node *next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};
// Function to delete the first node
void deleteFirstNode(Node *&head)
{
    if (head == nullptr)
    { // Check if the list is empty
        cout << "The list is empty. No node to delete." << endl;
        return;
    }
```

```
    Node *temp = head; // Store the current head
    head = head->next; // Move head to the next node

    if (head != nullptr)
    {
        head->prev = nullptr; // Update the new head's previous pointer
    }

    delete temp; // Free the memory of the old head
    cout << "First node deleted successfully." << endl;
}

// Function to display the list
void displayList(Node *head)
{
    Node *temp = head;
    while (temp != nullptr)
    {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to append a new node at the end
void appendNode(Node *&head, int value)
```

```
{
   Node *newNode = new Node(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()
{
   Node *head = nullptr;

   // Append some nodes to the list
   appendNode(head, 10);
   appendNode(head, 20);
   appendNode(head, 30);
```

```
    cout << "Original list: ";
    displayList(head);

    // Delete the first node
    deleteFirstNode(head);

    cout << "List after deleting the first node: ";
    displayList(head);

    return 0;
}
```



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

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

// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
```

```
};

// Function to delete the last node
void deleteLastNode(Node*& head) {
  if (head == nullptr) { // Check if the list is empty
    cout << "The list is empty. No node to delete." << endl;
    return;
  }

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

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

  // Update the second last node's next pointer
  temp->prev->next = nullptr;
  delete temp; // Free the memory of the last node
```

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

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to append a new node at the end
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(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() {
    Node* head = nullptr;

    // Append some nodes to the list
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);

    cout << "Original list: ";
    displayList(head);

    // Delete the last node
    deleteLastNode(head);

    cout << "List after deleting the last node: ";
    displayList(head);

    return 0;
```

```
Original list: 10 20 30
Last node deleted successfully.
List after deleting the last node: 10 20
```

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

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

// Node structure
struct Node {
  int data;
  Node* prev;
  Node* next;

  Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

// Function to delete a node by its value
void deleteNodeByValue(Node*& head, int value) {
  if (head == nullptr) { // Check if the list is empty
    cout << "The list is empty. No node to delete." << endl;
    return;
  }

  Node* temp = head;

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

```
    temp = temp->next;
  }

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

  // If the node to be deleted is the head
  if (temp == head) {
    head = head->next;
    if (head != nullptr) {
      head->prev = nullptr;
    }
    delete temp;
    cout << "Node with value " << value << " deleted from the head." << endl;
    return;
  }

  // If the node to be deleted is the last node
  if (temp->next == nullptr) {
    temp->prev->next = nullptr;
    delete temp;
    cout << "Node with value " << value << " deleted from the tail." << endl;
    return;
  }
```

```
    // If the node to be deleted is in the middle
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
    delete temp;
    cout << "Node with value " << value << " deleted from the middle." << endl;
}

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to append a new node at the end
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(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() {
    Node* head = nullptr;

    // Append some nodes to the list
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);

    cout << "Original list: ";
    displayList(head);

    // Delete a node by value
    deleteNodeByValue(head, 20);

    cout << "List after deleting node with value 20: ";
    displayList(head);
```

```
    // Try deleting a non-existing value
    deleteNodeByValue(head, 50);

    return 0;
```

```
Original list: 10 20 30 40
Node with value 20 deleted from the middle.
List after deleting node with value 20: 10 30 40
Node with value 50 not found.
```

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;

// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

// Function to delete a node at a specific position
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) { // Check if the list is empty
```

```
    cout << "The list is empty. No node to delete." << endl;
    return;
}

if (position < 1) { // Invalid position
    cout << "Invalid position. Position must be greater than or equal to 1." << endl;
    return;
}

Node* temp = head;

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

if (temp == nullptr) { // Position is out of bounds
    cout << "Position out of bounds. No node found at position " << position << "." << endl;
    return;
}

// If the node to be deleted is the head
if (temp == head) {
    head = head->next;
    if (head != nullptr) {
        head->prev = nullptr;
```

```
        }
        delete temp;
        cout << "Node at position " << position << " deleted from the head." << endl;
        return;
    }

    // If the node to be deleted is the last node
    if (temp->next == nullptr) {
        temp->prev->next = nullptr;
        delete temp;
        cout << "Node at position " << position << " deleted from the tail." << endl;
        return;
    }

    // If the node to be deleted is in the middle
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
    delete temp;
    cout << "Node at position " << position << " deleted from the middle." << endl;
}

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
```

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

// Function to append a new node at the end
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(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() {
    Node* head = nullptr;

    // Append some nodes to the list
```

```
    appendNode(head, 10);

    appendNode(head, 20);

    appendNode(head, 30);

    appendNode(head, 40);


    cout << "Original list: ";

    displayList(head);

    // Delete node at position 2

    deleteNodeAtPosition(head, 2);

    cout << "List after deleting node at position 2: ";

    displayList(head);

    // Delete node at position 1 (head)

    deleteNodeAtPosition(head, 1);

    cout << "List after deleting node at position 1: ";

    displayList(head);

    // Delete node at an invalid position

    deleteNodeAtPosition(head, 10);

    return 0;

}
```

```
--interpreter=mi
Original list: 10 20 30 40
Node at position 2 deleted from the middle.
List after deleting node at position 2: 10 30 40
Node at position 1 deleted from the head.
List after deleting node at position 1: 30 40
Position out of bounds. No node found at position 10.
```

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

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

// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

// Function to append a new node at the end
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }

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

```
    }

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

// Function to delete a node at a specific position
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) { // Check if the list is empty
        cout << "The list is empty. No node to delete." << endl;
        return;
    }

    if (position < 1) { // Invalid position
        cout << "Invalid position. Position must be greater than or equal to 1." << endl;
        return;
    }

    Node* temp = head;

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

    if (temp == nullptr) { // Position is out of bounds
```

```
        cout << "Position out of bounds. No node found at position " << position << "." << endl;
        return;
    }

    // If the node to be deleted is the head
    if (temp == head) {
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
        delete temp;
        cout << "Node at position " << position << " deleted from the head." << endl;
        return;
    }

    // If the node to be deleted is the last node
    if (temp->next == nullptr) {
        temp->prev->next = nullptr;
        delete temp;
        cout << "Node at position " << position << " deleted from the tail." << endl;
        return;
    }

    // If the node to be deleted is in the middle
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
```

```
    delete temp;
    cout << "Node at position " << position << " deleted from the middle." << endl;
}

// Function for forward traversal
void forwardTraversal(Node* head) {
    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;
}

// Function for reverse traversal
void reverseTraversal(Node* head) {
    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 from the tail to the head
    cout << "Reverse Traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}

// Main function
int main() {
    Node* head = nullptr;

    // Append some nodes to the list
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);
```

```
    cout << "Original list:" << endl;
    forwardTraversal(head);
    reverseTraversal(head);

    // Delete node at position 2
    deleteNodeAtPosition(head, 2);

    cout << "\nList after deleting node at position 2:" << endl;
    forwardTraversal(head);
    reverseTraversal(head);
    // Delete node at position 1 (head)
    deleteNodeAtPosition(head, 1);
    cout << "\nList after deleting node at position 1:" << endl;
    forwardTraversal(head);
    reverseTraversal(head);
    // Delete node at an invalid position
    deleteNodeAtPosition(head, 10);
    return 0;
}
```

```
List after deleting node at position 2:
Forward Traversal: 10 30 40
Reverse Traversal: 40 30 10
Node at position 1 deleted from the head.

List after deleting node at position 1:
Forward Traversal: 30 40
Reverse Traversal: 40 30
Position out of bounds. No node found at position 10.
```

Q6: Write a program to delete the first node in a circular linked list.

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

// Node structure
struct Node {
   int data;
   Node* next;

   Node(int value) : data(value), next(nullptr) {}
};

// Function to append a new node to the circular linked list
void appendNode(Node*& head, int value) {
   Node* newNode = new Node(value);
   if (head == nullptr) {
      head = newNode;
      head->next = head; // Point to itself to make it circular
      return;
   }

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

```
    }

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

// Function to delete the first node in a circular linked list
void deleteFirstNode(Node*& head) {
    if (head == nullptr) {
        cout << "The list is empty. No node to delete." << endl;
        return;
    }

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

    // Otherwise, delete the first node
    Node* temp = head;
    Node* last = head;

    // Find the last node
```

```
    while (last->next != head) {
        last = last->next;
    }
    // Update the head and the last node's next pointer
    head = head->next;
    last->next = head;
    delete temp;
    cout << "The first node was deleted." << endl;
}
// Function to display the circular linked list
void displayList(Node* head) {
    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;
}
int main() {
    Node* head = nullptr;
    // Append some nodes to the list
```

```
appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original list:" << endl;

displayList(head);

// Delete the first node

deleteFirstNode(head);

cout << "\nList after deleting the first node:" << endl;

displayList(head);

// Delete the first node again

deleteFirstNode(head);

cout << "\nList after deleting the first node again:" << endl;

displayList(head);

return 0;
```

```
Original list:
Circular Linked List: 10 20 30 40
The first node was deleted.

List after deleting the first node:
Circular Linked List: 20 30 40
The first node was deleted.

List after deleting the first node again:
Circular Linked List: 30 40
```

Q7: How can you delete the last node in a circular linked list? Write the code.

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

// Node structure
struct Node {
   int data;
   Node* next;

   Node(int value) : data(value), next(nullptr) {}
};

// Function to append a new node to the circular linked list
void appendNode(Node*& head, int value) {
   Node* newNode = new Node(value);
   if (head == nullptr) {
      head = newNode;
      head->next = head; // Point to itself to make it circular
      return;
   }

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

```
    }

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

// Function to delete the last node in a circular linked list
void deleteLastNode(Node*& head) {
    if (head == nullptr) {
        cout << "The list is empty. No node to delete." << endl;
        return;
    }

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

    // Otherwise, delete the last node
    Node* temp = head;
    Node* prev = nullptr;

    // Traverse to the last node
```

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

    // Update the second last node's next pointer to point to head
    prev->next = head;

    delete temp;
    cout << "The last node was deleted." << endl;
}

// Function to display the circular linked list
void displayList(Node* head) {
    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() {

   Node* head = nullptr;

   appendNode(head, 10);

   appendNode(head, 20);

   appendNode(head, 30);

   appendNode(head, 40);

   cout << "Original list:" << endl;

   displayList(head);

   // Delete the last node

   deleteLastNode(head);

   cout << "\nList after deleting the last node:" << endl;

   displayList(head);

   // Delete the last node again

   deleteLastNode(head);

   cout << "\nList after deleting the last node again:" << endl;

   displayList(head);

   return 0;

}
```

```
Original list:
Circular Linked List: 10 20 30 40
The last node was deleted.

List after deleting the last node:
Circular Linked List: 10 20 30
The last node was deleted.

List after deleting the last node again:
Circular Linked List: 10 20
```

Q8: Write a function to delete a node by its value in a circular linked list.

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

// Node structure
struct Node {
   int data;
   Node* next;

   Node(int value) : data(value), next(nullptr) {}
};

// Function to append a new node to the circular linked list
void appendNode(Node*& head, int value) {
   Node* newNode = new Node(value);
   if (head == nullptr) {
      head = newNode;
      head->next = head; // Point to itself to make it circular
      return;
   }

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

```
    }

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

// Function to delete a node by its value in a circular linked list
void deleteNodeByValue(Node*& head, int value) {
    if (head == nullptr) {
        cout << "The list is empty. No node to delete." << endl;
        return;
    }

    Node* temp = head;
    Node* prev = nullptr;

    // Case 1: The node to be deleted is the head node
    if (head->data == value) {
        // If there's only one node in the list
        if (head->next == head) {
            delete head;
            head = nullptr;
            cout << "The node with value " << value << " was deleted. The list is now empty." <<
endl;
            return;
        }
```

```
    // Otherwise, find the last node to update its next pointer
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = head->next;
    Node* toDelete = head;
    head = head->next;
    delete toDelete;
    cout << "The node with value " << value << " was deleted from the head." << endl;
    return;
}

// Case 2: The node to be deleted is not the head node
do {
    prev = temp;
    temp = temp->next;

    if (temp->data == value) {
        prev->next = temp->next;
        delete temp;
        cout << "The node with value " << value << " was deleted." << endl;
        return;
    }
} while (temp != head);
```

```
    // If the node with the given value is not found
    cout << "The node with value " << value << " was not found." << endl;
}

// Function to display the circular linked list
void displayList(Node* head) {
    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() {
    Node* head = nullptr;

    // Append some nodes to the list
    appendNode(head, 10);
    appendNode(head, 20);
```

```
    appendNode(head, 30);
    appendNode(head, 40);
    cout << "Original list:" << endl;
    displayList(head);
    // Delete a node by value
    deleteNodeByValue(head, 20);
    cout << "\nList after deleting node with value 20:" << endl;
    displayList(head);
    deleteNodeByValue(head, 10);
    cout << "\nList after deleting node with value 10:" << endl;
    displayList(head);
    // Attempt to delete a non-existent value
    deleteNodeByValue(head, 50);
    cout << "\nList after attempting to delete node with value 50:" << endl;
    displayList(head);
    return 0;
}
```

```
List after deleting node with value 20:
Circular Linked List: 10 30 40
The node with value 10 was deleted from the head.

List after deleting node with value 10:
Circular Linked List: 30 40
The node with value 50 was not found.

List after attempting to delete node with value 50:
Circular Linked List: 30 40
```

Q9: How will you delete a node at a specific position in a circular linked list?

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

// Node structure
struct Node {
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

// Function to append a new node to the circular linked list
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        head->next = head; // Point to itself to make it circular
        return;
    }

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

```
    }

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

// Function to delete a node at a specific position in a circular linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "The list is empty. No node to delete." << endl;
        return;
    }

    Node* temp = head;
    Node* prev = nullptr;

    // Case 1: If position is 0, delete the head node
    if (position == 0) {
        if (head->next == head) { // Only one node in the list
            delete head;
            head = nullptr;
            cout << "The node at position 0 (head) was deleted. The list is now empty." << endl;
            return;
        }

        // Find the last node to update its next pointer
```

```
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = head->next;
    Node* toDelete = head;
    head = head->next;
    delete toDelete;
    cout << "The node at position 0 (head) was deleted." << endl;
    return;
}

// Case 2: Traverse to the position to delete the node
int count = 0;
do {
    prev = temp;
    temp = temp->next;
    count++;
    if (count == position) {
        prev->next = temp->next;
        delete temp;
        cout << "The node at position " << position << " was deleted." << endl;
        return;
    }
} while (temp != head);

// If position is greater than the length of the list
```

```
    cout << "Position " << position << " is invalid." << endl;
}

// Function to display the circular linked list
void displayList(Node* head) {
    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() {
    Node* head = nullptr;

    // Append some nodes to the list
    appendNode(head, 10);
    appendNode(head, 20);
```

```
appendNode(head, 30);

appendNode(head, 40);

cout << "Original list:" << endl;

displayList(head);

// Delete node at a specific position

deleteNodeAtPosition(head, 2); // Deleting node at position 2 (third node)

cout << "\nList after deleting node at position 2:" << endl;

displayList(head);

deleteNodeAtPosition(head, 0);

cout << "\nList after deleting node at position 0 (head):" << endl;

displayList(head);

deleteNodeAtPosition(head, 5); // Invalid position

cout << "\nList after attempting to delete node at position 5:" << endl;

displayList(head);

return 0;
```

```
Original list:
Circular Linked List: 10 20 30 40
The node at position 2 was deleted.

List after deleting node at position 2:
Circular Linked List: 10 20 40
The node at position 0 (head) was deleted.

List after deleting node at position 0 (head):
Circular Linked List: 20 40
Position 5 is invalid.

List after attempting to delete node at position 5:
Circular Linked List: 20 40
```

```
}
```

Q10: Write a program to show forward traversal after deleting a node in a circular linked list.

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

// Node structure
struct Node {
   int data;
   Node* next;

   Node(int value) : data(value), next(nullptr) {}
};

// Function to append a new node to the circular linked list
void appendNode(Node*& head, int value) {
   Node* newNode = new Node(value);
   if (head == nullptr) {
      head = newNode;
      head->next = head; // Point to itself to make it circular
      return;
   }

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

```
    }

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

// Function to delete a node by position in a circular linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "The list is empty. No node to delete." << endl;
        return;
    }

    Node* temp = head;
    Node* prev = nullptr;

    // Case 1: If position is 0, delete the head node
    if (position == 0) {
        if (head->next == head) { // Only one node in the list
            delete head;
            head = nullptr;
            cout << "The node at position 0 (head) was deleted. The list is now empty." << endl;
            return;
        }

        // Find the last node to update its next pointer
```

```
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = head->next;
        Node* toDelete = head;
        head = head->next;
        delete toDelete;
        cout << "The node at position 0 (head) was deleted." << endl;
        return;
    }

    // Case 2: Traverse to the position to delete the node
    int count = 0;
    do {
        prev = temp;
        temp = temp->next;
        count++;
        if (count == position) {
            prev->next = temp->next;
            delete temp;
            cout << "The node at position " << position << " was deleted." << endl;
            return;
        }
    } while (temp != head);

    // If position is greater than the length of the list
```

```
    cout << "Position " << position << " is invalid." << endl;
}

// Function to display the circular linked list (forward traversal)
void displayList(Node* head) {
    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() {
    Node* head = nullptr;

    // Append some nodes to the list
    appendNode(head, 10);
    appendNode(head, 20);
```

```
    appendNode(head, 30);
    appendNode(head, 40);
    cout << "Original list:" << endl;
    displayList(head);
    // Delete node at position 2 (third node)
    deleteNodeAtPosition(head, 2);
    cout << "\nList after deleting node at position 2:" << endl;
    displayList(head);
    deleteNodeAtPosition(head, 0);
    cout << "\nList after deleting node at position 0 (head):" << endl;
    displayList(head);
    // Attempt to delete a non-existent position (invalid position)
    deleteNodeAtPosition(head, 5); // Invalid position
    cout << "\nList after attempting to delete node at position 5:" << endl;
    displayList(head);
    return 0;
}
```

```
Original list:
Circular Linked List: 10 20 30 40
The node at position 2 was deleted.

List after deleting node at position 2:
Circular Linked List: 10 20 40
The node at position 0 (head) was deleted.

List after deleting node at position 0 (head):
Circular Linked List: 20 40
Position 5 is invalid.

List after attempting to delete node at position 5:
Circular Linked List: 20 40
```

Q11: Write a program to count all the nodes in a binary search tree.

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

// Define the structure for a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node in the binary search tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }
```

```
    return root;
}

// Function to count the nodes in the binary search tree
int countNodes(Node* root) {
    if (root == nullptr) {
        return 0; // Base case: no node, return 0
    }

    // Recursively count nodes in the left and right subtrees, and add 1 for the current node
    return 1 + countNodes(root->left) + countNodes(root->right);
}

// Function to perform an in-order traversal (optional for demonstration)
void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}
// Main function
int main() {
    Node* root = nullptr;
```

```
    root = insert(root, 50);

    root = insert(root, 30);

    root = insert(root, 20);

    root = insert(root, 40);

    root = insert(root, 70);

    root = insert(root, 60);

    root = insert(root, 80);

    // In-order traversal (optional)

    cout << "In-order Traversal of the BST: ";

    inorderTraversal(root);

    cout << endl;

    // Count and display the total number of nodes

    int nodeCount = countNodes(root);

    cout << "Total number of nodes in the BST: " << nodeCount << endl;

    return 0;

}
```

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

Q12: How can you search for a specific value in a binary search tree? Write the code.

```
#include <iostream>

using namespace std;


// Define the structure for a node in the binary search tree
```

```
struct Node {
   int data;
   Node* left;
   Node* right;

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

// Function to insert a node in the binary search tree
Node* insert(Node* root, int value) {
   if (root == nullptr) {
      return new Node(value);
   }

   if (value < root->data) {
      root->left = insert(root->left, value);
   } else {
      root->right = insert(root->right, value);
   }

   return root;
}

// Function to search for a specific value in the binary search tree
bool search(Node* root, int target) {
   if (root == nullptr) {
```

```
        return false; // Base case: reached a null node, value not found
    }

    // If the value matches the root's data
    if (root->data == target) {
        return true;
    }

    // If the target value is smaller, search in the left subtree
    if (target < root->data) {
        return search(root->left, target);
    }

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

// Function to perform an in-order traversal (optional for demonstration)
void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}
```

```
// Main function
int main() {
    Node* root = nullptr;
    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
    // In-order traversal (optional)
    cout << "In-order Traversal of the BST: ";
    inorderTraversal(root);
    cout << endl;
    // Search for specific values
    int target1 = 40;
    int target2 = 90;
    if (search(root, target1)) {
        cout << "Found " << target1 << " in the BST." << endl;
    } else {
        cout << target1 << " not found in the BST." << endl;
    }
    if (search(root, target2)) {
        cout << "Found " << target2 << " in the BST." << endl;
    } else {
```

```
        cout << target2 << " not found in the BST." << endl;
    }
    return 0;
}
```

```
In-order Traversal of the BST: 20 30 40 50 60 70 80
Found 40 in the BST.
90 not found in the BST.
```

Q13: Write code to traverse a binary search tree in in-order, pre-order, and postorder.

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

// Define the structure for a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node in the binary search tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
```

```
        return new Node(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }

    return root;
}

// In-order Traversal: Left, Node, Right
void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);   // Traverse left subtree
        cout << root->data << " ";      // Visit current node
        inorderTraversal(root->right);  // Traverse right subtree
    }
}

// Pre-order Traversal: Node, Left, Right
void preorderTraversal(Node* root) {
    if (root != nullptr) {
        cout << root->data << " ";      // Visit current node
        preorderTraversal(root->left);  // Traverse left subtree
```

```
        preorderTraversal(root->right); // Traverse right subtree
    }
}

// Post-order Traversal: Left, Right, Node
void postorderTraversal(Node* root) {
    if (root != nullptr) {
        postorderTraversal(root->left);   // Traverse left subtree
        postorderTraversal(root->right);  // Traverse right subtree
        cout << root->data << " ";       // Visit current node
    }
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
```

```
    // In-order Traversal
    cout << "In-order Traversal: ";
    inorderTraversal(root);
    cout << endl;

    // Pre-order Traversal
    cout << "Pre-order Traversal: ";
    preorderTraversal(root);
    cout << endl;

    // Post-order Traversal
    cout << "Post-order Traversal: ";
    postorderTraversal(root);
    cout << endl;

    return 0;
}
```

```
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
```

Q14: 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 for a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node in the binary search tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
```

```
    }

    return root;
}

// Reverse In-order Traversal: Right, Node, Left
void reverseInorderTraversal(Node* root) {
    if (root != nullptr) {
        reverseInorderTraversal(root->right);  // Traverse right subtree
        cout << root->data << " ";             // Visit current node
        reverseInorderTraversal(root->left);   // Traverse left subtree
    }
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
```

```
    // Reverse In-order Traversal
    cout << "Reverse In-order Traversal: ";
    reverseInorderTraversal(root);
    cout << endl;

    return 0;
}
```

```
Reverse In-order Traversal: 80 70 60 50 40 30 20
```

Q15: Write a program to check if there are duplicate values in a binary search tree.

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

// Define the structure for a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
```

```
};

// Function to insert a node in the binary search tree
Node* insert(Node* root, int value) {
   if (root == nullptr) {
      return new Node(value);
   }

   if (value < root->data) {
      root->left = insert(root->left, value);
   } else if (value > root->data) {
      root->right = insert(root->right, value);
   }

   return root;
}

// Function to perform in-order traversal and check for duplicates
bool checkDuplicates(Node* root, set<int>& visited) {
   if (root == nullptr) {
      return false;
   }

   // Traverse left subtree
   if (checkDuplicates(root->left, visited)) {
      return true; // Duplicate found in the left subtree
```

```
    }

    // Check if the current node's value is already visited
    if (visited.find(root->data) != visited.end()) {
        return true; // Duplicate found
    }

    // Mark the current node's value as visited
    visited.insert(root->data);

    // Traverse right subtree
    return checkDuplicates(root->right, visited);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
```

```
    // Insert a duplicate value to test
    root = insert(root, 40);  // Duplicate value for testing

    set<int> visited; // Set to store visited values

    // Check for duplicates in the BST
    if (checkDuplicates(root, visited)) {
        cout << "Duplicate values found in the BST." << endl;
    } else {
        cout << "No duplicate values found in the BST." << endl;
    }

    return 0;
}
```

```
Duplicate values found in the BST.
```

Q16: 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 for a node in the binary search tree
struct Node {
```

```
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node in the binary search tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }
    return root;
}

// Function to find the minimum value node in a given tree
Node* findMin(Node* root) {
    while (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;  // Node to be deleted not found
    }

    // If the value is smaller than the root, it is in the left subtree
    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    }
    // If the value is larger than the root, it is in the right subtree
    else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    }
    // If the value is the same as the root's data, this is the node to be deleted
    else {
        // Case 1: Node has no children (leaf node)
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
            return nullptr;
        }
        // Case 2: Node has one child
        else if (root->left == nullptr) {
```

```
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        // Case 3: Node has two children
        else {
            // Find the in-order successor (smallest node in the right subtree)
            Node* temp = findMin(root->right);
            // Replace the current node's data with the in-order successor's data
            root->data = temp->data;
            // Delete the in-order successor
            root->right = deleteNode(root->right, temp->data);
        }
    }

    return root;
}

// In-order traversal to display the tree
void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
```

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

// Main function to demonstrate deleting nodes in a BST
int main() {
    Node* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    cout << "Original tree (In-order traversal): ";
    inorderTraversal(root);
    cout << endl;

    // Delete a leaf node (20)
    root = deleteNode(root, 20);
    cout << "After deleting leaf node 20: ";
    inorderTraversal(root);
```

```
    cout << endl;

    // Delete a node with one child (30)
    root = deleteNode(root, 30);
    cout << "After deleting node with one child (30): ";
    inorderTraversal(root);
    cout << endl;

    // Delete a node with two children (50)
    root = deleteNode(root, 50);
    cout << "After deleting node with two children (50): ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

```
Original tree (In-order traversal): 20 30 40 50 60 70 80
After deleting leaf node 20: 30 40 50 60 70 80
After deleting node with one child (30): 40 50 60 70 80
After deleting node with two children (50): 40 60 70 80
```