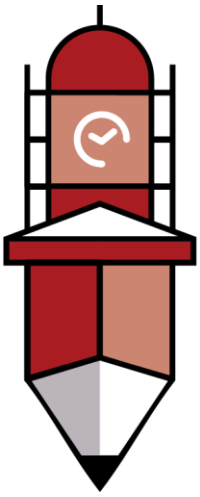

The University of Faisalabad

# Data Structures

# Course Code: CS-216

## Assignment

**Submitted to:**

**Ms. Irsha Qureshi**

**Submitted by:**

**Muhammad Tayyab Imran**

**Registration no:**

**2023-BS-AI-019**

**Department:**

**Computer Science**

# Doubly Linked List

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

## Code:

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

// Define the structure for a doubly linked list node
struct node
{
	int data; // Data stored in the node
	struct node *next; // Pointer to the next node
	struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
	n = new node(); // Allocate memory for a new node
	n->data = data; // Assign the data to the new node

	if(first == NULL) // If the list is empty
	{
		n->next = n->prev = NULL; // Initialize the new node's pointers to NULL
		first = last = n; // Set the new node as the first and last node
	}
	else // If the list is not empty
	{
		n->next = NULL; // Set the new node's next pointer to NULL
		n->prev = last; // Link the new node to the last node
		last->next = n; // Update the last node's next pointer
		last = n; // Update the last pointer to the new node
	}
}

// Function to delete the first node in the linked list
void deleteFirstNode ()
{
	temp = first; // Temporarily store the first node
	first = first->next; // Move the first pointer to the next node
	first->prev = NULL; // Set the previous pointer of the new first node to NULL
	delete(temp); // Free the memory of the old first node
}
```

```
// Function to display all nodes in the linked list
void display()
{
	current = first; // Start from the first node

	while(current != NULL) // Traverse the list until the end
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->next; // Move to the next node
	}
	cout << endl;
}

int main()
{
	// Create nodes in the linked list with the given values
	create(10);
	create(20);
	create(30);
	create(40);
	create(50);

	// Display the linked list before deletion
	cout << "Display before Deletion: ";
	display();

	// Delete the first node in the linked list
	deleteFirstNode();

	// Display the linked list after deletion
	cout << "Display after Deletion: ";
	display();

	return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Display after Deletion: 20 30 40 50
```

.....

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

**Code:**

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

// Define the structure for a doubly linked list node
struct node
{
```

```
        int data; // Data stored in the node
        struct node *next; // Pointer to the next node
        struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
        n = new node(); // Allocate memory for a new node
        n->data = data; // Assign the data to the new node

        if(first == NULL) // If the list is empty
        {
                n->next = n->prev = NULL; // Initialize the new node's pointers to NULL
                first = last = n; // Set the new node as the first and last node
        }
        else // If the list is not empty
        {
                n->next = NULL; // Set the new node's next pointer to NULL
                n->prev = last; // Link the new node to the last node
                last->next = n; // Update the last node's next pointer
                last = n; // Update the last pointer to the new node
        }
}

// Function to delete the last node in the linked list
void deleteLastNode()
{
        temp = last; // Temporarily store the last node
        last = last->prev; // Move the last pointer to the previous node
        last->next = NULL; // Set the next pointer of the new last node to NULL

        delete(temp); // Free the memory of the old last node
}

// Function to display all nodes in the linked list
void display()
{
        current = first; // Start from the first node

        while(current != NULL) // Traverse the list until the end
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->next; // Move to the next node
        }
        cout << endl;
}
```

```
int main()
{
	// Create nodes in the linked list with the given values
	create(10);
	create(20);
	create(30);
	create(40);
	create(50);

	// Display the linked list before deletion
	cout << "Display before Deletion: ";
	display();

	// Delete the last node in the linked list
	deleteLastNode();

	// Display the linked list after deletion
	cout << "Display after Deletion: ";
	display();

	return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Display after Deletion: 10 20 30 40
```

.....

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

**Code:**

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

// Define the structure for a doubly linked list node
struct node
{
	int data; // Data stored in the node
	struct node *next; // Pointer to the next node
	struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
	n = new node(); // Allocate memory for a new node
```

```
        n->data = data; // Assign the data to the new node

        if(first == NULL) // If the list is empty
        {
                n->next = n->prev = NULL; // Initialize the new node's pointers to NULL
                first = last = n; // Set the new node as the first and last node
        }
        else // If the list is not empty
        {
                n->next = NULL; // Set the new node's next pointer to NULL
                n->prev = last; // Link the new node to the last node
                last->next = n; // Update the last node's next pointer
                last = n; // Update the last pointer to the new node
        }
}

// Function to delete a node with a specific value from the list
void deleteByValue(int value)
{
        current = first; // Start from the first node

        while(current != NULL) // Traverse the list
        {
                if(current->data == value) // If the node with the given value is found
                {
                        if(current == first) // Case 1: Deleting the first node
                        {
                                temp = first; // Temporarily store the first node
                                first = first->next; // Update the first pointer to the next
                                node
                                if (first != NULL) // Check if the list is not empty after
                                deletion
                                first->prev = NULL; // Set the previous pointer of the new
                                first node to NULL
                                delete(temp); // Free the memory of the old first node
                        }
                        else if(current == last) // Case 2: Deleting the last node
                        {
                                temp = last; // Temporarily store the last node
                                last = last->prev; // Update the last pointer to the previous
                                node

                                last->next = NULL; // Set the next pointer of the new last
                                node to NULL
                                delete(temp); // Free the memory of the old last node
                        }
                        else // Case 3: Deleting a node in the middle
                        {
                                current->prev->next = current->next; // Update the
                                previous node's next pointer
```

```
                        current->next->prev = current->prev; // Update the next
                        node's previous pointer
                        delete current; // Free the memory of the current node
                    }
                    return; // Exit the function after deletion
                }
                current = current->next; // Move to the next node
            }
            cout<<"Value Not Found!"; // If value to be deleted is not found
            exit(0); // Exit the program
}

// Function to display all nodes in the linked list
void display()
{
            current = first; // Start from the first node

            while(current != NUL; // Traverse the list until the end
            {
                        cout << current->data << " "; // Print the data of the current node
                        current = current->next; // Move to the next node
            }
            cout << endl;
}

int main()
{
            // Create a linked list with the given elements
            create(10);
            create(20);
            create(30);
            create(40);
            create(50);

            // Display the linked list before deletion
            cout << "Display before Deletion: ";
            display();

            // Input the value to be deleted
            int value;
            cout << "Enter the Value to Delete: ";
            cin >> value;

            // Delete the node with the given value
            deleteByValue(value);

            // Display the linked list after deletion
            cout << "Display after Deletion: ";
            display();
```

```
    return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Enter the Value to Delete: 30
Display after Deletion: 10 20 40 50
```

.....

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

**Code:**

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

// Define the structure for a doubly linked list node
struct node
{
    int data; // Data stored in the node
    struct node *next; // Pointer to the next node
    struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
    n = new node(); // Allocate memory for a new node
    n->data = data; // Assign the data to the new node

    if(first == NULL) // If the list is empty
    {
        n->next = n->prev = NULL; // Initialize the new node's pointers to NULL
        first = last = n; // Set the new node as the first and last node
    }
    else // If the list is not empty
    {
        n->next = NULL; // Set the new node's next pointer to NULL
        n->prev = last; // Link the new node to the last node
        last->next = n; // Update the last node's next pointer
        last = n; // Update the last pointer to the new node
    }
}

// Function to delete a node at a specific position in a doubly linked list
void deleteAtSpecificPos(int position)
{
```

```
// Check if the position is invalid or the list is empty
if (position <= 0 || first == nullptr)
{
	cout << "Invalid position or empty list!" << endl;
	return;
}

current = first; // Start from the head of the list
int index = 1;

// Traverse the list to find the node at the specified position
while (current != NULL && index <= position)
{
	if (index == position)
	{
		temp = current; // Store the node to be deleted

		if (current == first)
		{
			// Case: Deleting the first node
			first = current->next; // Update head to the next node
			first->prev = NULL; // Remove backward link from the
			new head
		}
		else if (current == last)
		{
			// Case: Deleting the last node
			last = current->prev; // Update tail to the previous node
			last->next = NULL; // Remove forward link from the new
			tail
		}
		else
		{
			// Case: Deleting a middle node
			current->prev->next = current->next; // Link the previous
			node to the next node
			current->next->prev = current->prev; // Link the next node
			to the previous node
		}

		delete temp; // Delete the node from memory
		return; // Exit the function after deletion
	}

	current = current->next; // Move to the next node
	index++;
}

// If the specified position is not found in the list
cout << "Value Not Found!"; // Print an error message
```

```
        exit(0); // Exit the program
}

// Function to display all nodes in the linked list
void display()
{
        current = first; // Start from the first node
        while(current != NUL; // Traverse the list until the end
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->next; // Move to the next node
        }
        cout << endl;
}

int main()
{
        // Create a linked list with the given elements
        create(10);
        create(20);
        create(30);
        create(40);
        create(50);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Input the position to be deleted
        int position;
        cout << "Enter the Position to Delete: ";
        cin >> position;

        // Delete the node at the specific position
        deleteAtSpecificPos(value);

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
        display();

        return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Enter the Value to Delete: 30
Display after Deletion: 10 20 40 50
```

.....

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

**Code:**

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

// Define the structure for a doubly linked list node
struct node
{
	int data; // Data stored in the node
	struct node *next; // Pointer to the next node
	struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
	n = new node(); // Allocate memory for a new node
	n->data = data; // Assign the data to the new node

	if(first == NULL) // If the list is empty
	{
		n->next = n->prev = NULL; // Initialize the new node's pointers to NULL
		first = last = n; // Set the new node as the first and last node
	}
	else // If the list is not empty
	{
		n->next = NULL; // Set the new node's next pointer to NULL
		n->prev = last; // Link the new node to the last node
		last->next = n; // Update the last node's next pointer
		last = n; // Update the last pointer to the new node
	}
}

// Function to delete a node with a specific value from the list
void deleteAtSpecificPos(int value)
{
	current = first; // Start from the first node

	while(current != NULL) // Traverse the list
	{
		if(current->data == value) // If the node with the given value is found
		{
			if(current == first) // Case 1: Deleting the first node
			{
```

```
                        temp = first; // Temporarily store the first node
                        first = first->next; // Update the first pointer to the next
                        node
                        if (first != NULL) // Check if the list is not empty after
                        deletion
                        first->prev = NULL; // Set the previous pointer of the new
                        first node to NULL
                        delete(temp); // Free the memory of the old first node
                }
                else if(current == last) // Case 2: Deleting the last node
                {
                        temp = last; // Temporarily store the last node
                        last = last->prev; // Update the last pointer to the previous
                        node

                        last->next = NULL; // Set the next pointer of the new last
                        node to NULL
                        delete(temp); // Free the memory of the old last node
                }
                else // Case 3: Deleting a node in the middle
                {
                        current->prev->next = current->next; // Update the
                        previous node's next pointer
                        current->next->prev = current->prev; // Update the next
                        node's previous pointer
                        delete current; // Free the memory of the current node
                }
                return; // Exit the function after deletion
        }
        current = current->next; // Move to the next node
    }
    cout<<"Value Not Found!"; // If value to be deleted is not found
    exit(0); // Exit the program
}

// Function to display all nodes forward in the linked list
void displayForward()
{
    current = first; // Start from the first node

    while(current != NUL; // Traverse the list until the end
    {
        cout << current->data << " "; // Print the data of the current node
        current = current->next; // Move to the next node
    }
    cout << endl;
}

// Function to display all nodes reverse in the linked list
void displayReverse()
```

```
{
	current = first; // Start from the lastt node

	while(current != NUL; // Traverse the list until the end
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->next; // Move to the previous node
	}
	cout << endl;
}

int main()
{
	// Create a linked list with the given elements
	create(10);
	create(20);
	create(30);
	create(40);
	create(50);

	// Display the forward linked list after deletion
	cout << "Display Forward before Deletion: ";
	displayForward();

	// Display the reverse linked list after deletion
	cout << "Display Reverse before Deletion: ";
	displayReverse();

	// Input the value to be deleted
	int value;
	cout << "\nEnter the Value to Delete: ";
	cin >> value;

	// Delete the node with the given value
	deleteByValue(value);

	// Display the forward linked list after deletion
	cout << "\nDisplay Forward after Deletion: ";
	displayForward();

	// Display the reverse linked list after deletion
	cout << "Display Reverse after Deletion: ";
	displayReverse();

	return 0;
}
```

**Output:**

```
Display Forward before Deletion: 10 20 30 40 50
Display Reverse before Deletion: 50 40 30 20 10

Enter the Value to Delete: 30

Display Forward after Deletion: 10 20 40 50
Display Reverse after Deletion: 50 40 20 10
```

.....

# Circular Linked List

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

## Code:

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

// Define the structure for a circular linked list node
struct node
{
	int data; // Data stored in the node
	struct node *link; // Pointer to the next node in the circular linked list
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
	n = new node(); // Allocate memory for a new node
	n->data = data; // Assign the data to the new node

	if(first == NULL) // If the list is empty
	{
		n->link = n; // Point the new node to itself (circular link)
		first = last = n; // Set the new node as both the first and last node
	}
	else // If the list is not empty
	{
		n->link = first; // Point the new node to the first node
		last->link = n; // Update the last node's link to point to the new node
		last = n; // Update the last pointer to the new node
	}
}

// Function to delete the first node in the circular linked list
void deleteFirstNode()
{
	temp = first; // Store the first node in a temporary pointer
	first = first->link; // Update the first pointer to the second node
	last->link = first; // Update the last node's link to point to the new first node
	delete(temp); // Free the memory of the old first node
}
```

```
// Function to display all nodes in the circular linked list
void display()
{
	current = first; // Start from the first node

	do
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link; // Move to the next node
	}
	while(current != first); // Stop when we loop back to the first node
	cout << endl;
}

// Main function
int main()
{
	// Create nodes in the circular linked list with the given values
	create(10);
	create(20);
	create(30);
	create(40);
	create(50);

	// Display the circular linked list before deletion
	cout << "Display before Deletion: ";
	display();

	// Delete the first node in the circular linked list
	deleteFirstNode();

	// Display the circular linked list after deletion
	cout << "Display after Deletion: ";
	display();

	return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Display after Deletion: 20 30 40 50
```

.....

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

**Code:**

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

```
// Define the structure for a circular linked list node
struct node
{
	int data; // Data stored in the node
	struct node *link; // Pointer to the next node in the circular linked list
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
	n = new node(); // Allocate memory for a new node
	n->data = data; // Assign the data to the new node

	if(first == NULL) // If the list is empty
	{
		n->link = n; // Point the new node to itself (circular link)
		first = last = n; // Set the new node as both the first and last node
	}
	else // If the list is not empty
	{
		n->link = first; // Point the new node to the first node
		last->link = n; // Update the last node's link to point to the new node
		last = n; // Update the last pointer to the new node
	}
}

// Function to delete the last node in the circular linked list
void deleteLastNode()
{
	current = first; // Start from the first node

	while(current->link != last) // Traverse the list to find the second-to-last node
	{
		current = current->link;
	}

	delete(last); // Free the memory of the last node
	current->link = first; // Update the second-to-last node's link to point to the first
	node
	last = current; // Update the last pointer to the second-to-last node
}


// Function to display all nodes in the circular linked list
void display()
{
```

```
        current = first; // Start from the first node
        do
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->link; // Move to the next node
        }
        while(current != first); // Stop when we loop back to the first node
        cout << endl;
}

// Main function
int main()
{
        // Create nodes in the linked list with the given values
        create(10);
        create(20);
        create(30);
        create(40);
        create(50);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Delete the last node in the linked list
        deleteLastNode();

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
        display();

        return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Display after Deletion: 10 20 30 40
```

.....

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

**Code:**

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

// Define the structure for a circular linked list node
struct node
{
        int data; // Data stored in the node
        struct node *link; // Pointer to the next node in the circular linked list
```

```
};
// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
	n = new node(); // Allocate memory for a new node
	n->data = data; // Assign the data to the new node

	if(first == NULL) // If the list is empty
	{
		n->link = n; // Point the new node to itself (circular link)
		first = last = n; // Set the new node as both the first and last node
	}
	else // If the list is not empty
	{
		n->link = first; // Point the new node to the first node
		last->link = n; // Update the last node's link to point to the new node
		last = n; // Update the last pointer to the new node
	}
}

// Function to delete a node with a specific value in the circular linked list
void deleteByValue(int value)
{
	struct node *previous = NULL; // Pointer to keep track of the previous node
	current = first; // Start with the first node

	do
	{
		if(current->data == value) // If the current node contains the value
		{
			if(current == first) // If the node to delete is the first node
			{
				first = first->link; // Update the first pointer to the next
				node
				last->link = first; // Maintain the circular structure
			}
			else if(current == last) // If the node to delete is the last node
			{
				previous->link = first; // Update the previous node to link
				to the first
				last = previous; // Update the last pointer to the previous
				node
			}
			else // If the node to delete is in the middle
			{
				previous->link = current->link; // Skip the current node
			}
```

```
                delete(current); // Free the memory of the deleted node
                return; // Exit the function after deletion
            }
            previous = current; // Update the previous pointer
            current = current->link; // Move to the next node
        }
        while(current != first); // Stop when we loop back to the first node

        cout<<"Value Not Found!"; // If value to be deleted is not found
        exit(0); // Exit the program
}

// Function to display all nodes in the circular linked list
void display()
{
        current = first; // Start from the first node

        do
        {
            cout << current->data << " "; // Print the data of the current node
            current = current->link; // Move to the next node
        }
        while(current != first); // Stop when we loop back to the first node
        cout << endl;
}

// Main function
int main()
{
        // Create a linked list with the given elements
        create(10);
        create(20);
        create(30);
        create(40);
        create(50);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Input the value to be deleted
        int value;
        cout << "Enter the Value to Delete: ";
        cin >> value;

        // Delete the node with the given value
        deleteByValue(value);

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
```

```
        display();

        return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Enter the Value to Delete: 30
Display after Deletion: 10 20 40 50
```

.....

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

**Code:**

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

// Define the structure for a circular linked list node
struct node
{
        int data; // Data stored in the node
        struct node *link; // Pointer to the next node in the circular linked list
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
        n = new node(); // Allocate memory for a new node
        n->data = data; // Assign the data to the new node

        if(first == NULL) // If the list is empty
        {
                n->link = n; // Point the new node to itself (circular link)
                first = last = n; // Set the new node as both the first and last node
        }
        else // If the list is not empty
        {
                n->link = first; // Point the new node to the first node
                last->link = n; // Update the last node's link to point to the new node
                last = n; // Update the last pointer to the new node
        }
}

// Function to delete a node at a specific position from the circular linked list
void deleteAtSpecificPos(int position)
```

```
{
        // Check for invalid position or empty list
        if (position <= 0 || first == nullptr)
        {
                cout << "Invalid position or empty list!" << endl;
                return;
        }

        struct node *previous = NULL;
        current = first;
        int index = 1;

        do
        {
                // Check if the current node is at the specified position
                if (index == position)
                {
                        temp = current;

                        if (current == first)
                        {
                                // Case: Deleting the first node
                                first = current->link;
                                last->link = first; // Update last node to point to the new
                                first node
                        }
                        else if (current == last)
                        {
                                // Case: Deleting the last node
                                last = previous;
                                last->link = first; // Update the last node to point to the first
                                node
                        }
                        else
                        {
                                // Case: Deleting a middle node
                                previous->link = current->link;
                        }

                        // Delete the node and return
                        delete temp;
                        return;
                }

        // Move to the next node
        previous = current;
        current = current->link;
        index++;
        }
```

```
        while (current != first && index <= position); // Traverse until the end of the list
        or specified position

        // If position is out of bounds
        cout << "Value Not Found!";
        exit(0);
}

// Function to display all nodes in the circular linked list
void display()
{
        current = first; // Start from the first node

        do
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->link; // Move to the next node
        }
        while(current != first); // Stop when we loop back to the first node
        cout << endl;
}

// Main function
int main()
{
        // Create a linked list with the given elements
        create(10);
        create(20);
        create(30);
        create(40);
        create(50);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Input the position to be deleted
        int position;
        cout << "Enter the Value to Delete: ";
        cin >> position;

        // Delete the node at the specific position
        deleteAtSpecificPos(value);

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
        display();

        return 0;
}
```

**Output:**

```
Display before Deletion: 10 20 30 40 50
Enter the Value to Delete: 30
Display after Deletion: 10 20 40 50
```

.....

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

**Code:**

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

// Define the structure for a circular linked list node
struct node
{
	int data; // Data stored in the node
	struct node *link; // Pointer to the next node in the circular linked list
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
	n = new node(); // Allocate memory for a new node
	n->data = data; // Assign the data to the new node

	if(first == NULL) // If the list is empty
	{
		n->link = n; // Point the new node to itself (circular link)
		first = last = n; // Set the new node as both the first and last node
	}
	else // If the list is not empty
	{
		n->link = first; // Point the new node to the first node
		last->link = n; // Update the last node's link to point to the new node
		last = n; // Update the last pointer to the new node
	}
}

// Function to delete a node with a specific value in the circular linked list
void deleteByValue(int value)
{
	struct node *previous = NULL; // Pointer to keep track of the previous node
	current = first; // Start with the first node

	do
```

```
	{
		if(current->data == value) // If the current node contains the value
		{
			if(current == first) // If the node to delete is the first node
			{
				first = first->link; // Update the first pointer to the next
				node
				last->link = first; // Maintain the circular structure
			}
			else if(current == last) // If the node to delete is the last node
			{
				previous->link = first; // Update the previous node to link
				to the first
				last = previous; // Update the last pointer to the previous
				node
			}
			else // If the node to delete is in the middle
			{
				previous->link = current->link; // Skip the current node
			}
			delete(current); // Free the memory of the deleted node
			return; // Exit the function after deletion
		}
		previous = current; // Update the previous pointer
		current = current->link; // Move to the next node
	}
	while(current != first); // Stop when we loop back to the first node

	cout<<"Value Not Found!"; // If value to be deleted is not found
	exit(0); // Exit the program
}

// Function to display all nodes in the circular linked list
void displayForward()
{
	current = first; // Start from the first node

	do
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link; // Move to the next node
	}
	while(current != first); // Stop when we loop back to the first node
	cout << endl;
}

// Main function
int main()
{
	// Create a linked list with the given elements
```

```
    create(10);
    create(20);
    create(30);
    create(40);
    create(50);

    // Display the linked list before deletion
    cout << "Forward Display before Deletion: ";
    displayForward();

    // Input the value to be deleted
    int value;
    cout << "Enter the Value to Delete: ";
    cin >> value;

    // Delete the node with the given value
    deleteByValue(value);

    // Display the linked list after deletion
    cout << "Forward Display after Deletion: ";
    displayForward();

    return 0;
}
```

**Output:**

```
Forward Display before Deletion: 10 20 30 40 50
Enter the Value to Delete: 30
Forward Display after Deletion: 10 20 40 50
```

.....

# Binary Search Tree

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

## Code:

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

// Structure for node
struct Node
{
	int data; // Value of the node
	struct Node* left = NULL; // Pointer to the left child
	struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
	if (root == NULL) // If empty tree or reaching a leaf node
	{
		root = new Node(); // Create a new node
		root->data = value; // Assign the value
		return root; // Return the new node as the root
	}

	// Recur on the left subtree if the value is smaller
	else if (value < root->data)
	{
		root->left = insert(root->left, value);
	}
	// Recur on the right subtree if the value is larger
	else if (value > root->data)
	{
		root->right = insert(root->right, value);
	}

	return root; // Return the root after insertion
}

// Function to count all nodes in the BST
int countNodes(Node* root)
{
	if (!root)
	{
		return 0;
	}
```

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

// Function for in-order traversal
void inorder(Node* root)
{
        if (root == NULL) // If tree is empty
        {
                return;
        }

        inorder(root->left); // Traverse the left subtree
        cout << root->data << " "; // Print the current node's data
        inorder(root->right); // Traverse the right subtree
}

int main()
{
        Node* root = NULL; // Initialize an empty BST

        // Insert nodes into the BST
        root = insert(root, 19);
        root = insert(root, 33);
        root = insert(root, 52);
        root = insert(root, 28);
        root = insert(root, 56);
        root = insert(root, 16);

        // Display the tree nodes using in-order traversal
        cout << "Display In-Order: ";
        inorder(root);

        // Count and display the number of nodes in the BST
        cout << "\nTotal number of nodes in the BST: " << countNodes(root) << endl;

        return 0;
}
```

**Output:**

```
Display In-Order: 16 19 28 33 52 56
Total number of nodes in the BST: 6
```

.....

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

**Code:**

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

// Structure for node
struct Node
{
	int data; // Value of the node
	struct Node* left = NULL; // Pointer to the left child
	struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
	if (root == NULL) // If tree is empty or reaching a leaf
	{
		root = new Node(); // Create a new node
		root->data = value; // Assign the value to the node
		return root; // Return the new node as the root
	}

	// Recur on the left subtree if the value is smaller
	else if (value < root->data)
	{
			root->left = insert(root->left, value);
	}
	// Recur on the right subtree if the value is larger
	else if (value > root->data)
	{
		root->right = insert(root->right, value);
	}

	return root; // Return the root after insertion
}

// Function for searching
Node* searching(Node* root, int value)
{
	// If tree is empty or value matches the current node
	if (root == NULL || value == root->data)
	{
		return root;
	}
	// Recur on the left subtree if the value is smaller
	else if (value < root->data)
```

```
        {
                return searching(root->left, value);
        }
        // Recur on the right subtree if the value is larger
        else
        {
                return searching(root->right, value);
        }
}

int main()
{
        Node* root = NULL; // Initialize an empty BST

        // Insert nodes into the BST
        root = insert(root, 19);
        root = insert(root, 15);
        root = insert(root, 46);
        root = insert(root, 9);
        root = insert(root, 25);
        root = insert(root, 89);

        // Ask user for the value to search
        int value;
        cout<<"Enter the Value to Search: ";
        cin>>value;

        // Search for a value in the BST
        cout << "Searching Node: ";
        Node* search = searching(root, value); // Search for the value

        // Display search result
        if (search != NULL)
        {
                cout << "Value Exists!";
        }
        else
        {
                cout << "Value Doesn't Exist!";
        }

        return 0;
}
```

**Output:**

```
Enter the Value to Search: 19
Value Exists!
```

.....

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

**Code:**

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

// Structure for node
struct Node
{
	int data; // Value of the node
	struct Node* left = NULL; // Pointer to the left child
	struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
	if (root == NULL) // If tree is empty or reaching a leaf
	{
		root = new Node(); // Create a new node
		root->data = value; // Assign the value to the node
		return root; // Return the new node as the root
	}

	// Recur on the left subtree if the value is smaller
	else if (value < root->data)
	{
		root->left = insert(root->left, value);
	}
	// Recur on the right subtree if the value is larger
	else if (value > root->data)
	{
		root->right = insert(root->right, value);
	}

	return root; // Return the root after insertion
}

// Function for in-order traversal (Left, Root, Right)
void inorder(Node* root)
{
	if (root == NULL) // If tree is empty
	{
		return;
	}

	inorder(root->left); // Traverse the left subtree
	cout << root->data << " "; // Print the current node's data
```

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

// Function for pre-order traversal (Root, Left, Right)
void preorder(Node* root)
{
        if (root == NULL) // If tree is empty
        {
                return;
        }

        cout << root->data << " "; // Print the current node's data
        preorder(root->left); // Traverse the left subtree
        preorder(root->right); // Traverse the right subtree
}

// Function for post-order traversal (Left, Right, Root)
void postorder(Node* root)
{
        if (root == NULL) // If tree is empty
        {
                return;
        }

        postorder(root->left); // Traverse the left subtree
        postorder(root->right); // Traverse the right subtree
        cout << root->data << " "; // Print the current node's data
}

int main()
{
        Node* root = NULL; // Initialize an empty BST

        // Insert nodes into the BST
        root = insert(root, 19);
        root = insert(root, 18);
        root = insert(root, 20);
        root = insert(root, 17);
        root = insert(root, 30);
        root = insert(root, 10);

        // Display the BST using in-order traversal
        cout << "Display In-Order: ";
        inorder(root);

        // Display the BST using pre-order traversal
        cout << "\nDisplay Pre-Order: ";
        preorder(root);

        // Display the BST using post-order traversal
```

```
        cout << "\nDisplay Post-Order: ";
        postorder(root);

        return 0;
}
```

**Output:**

```
Display In-Order: 10 17 18 19 20 30
Display Pre-Order: 19 18 17 10 20 30
Display Post-Order: 10 17 18 30 20 19
```

.....

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

**Code:**

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

// Structure for node
struct Node
{
        int data; // Value of the node
        struct Node* left = NULL; // Pointer to the left child
        struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
        if (root == NULL) // If empty tree or reaching a leaf node
        {
                root = new Node(); // Create a new node
                root->data = value; // Assign the value
                return root; // Return the new node as the root
        }

        // Recur on the left subtree if the value is smaller
        else if (value < root->data)
        {
                root->left = insert(root->left, value);
        }
        // Recur on the right subtree if the value is larger
        else if (value > root->data)
        {
                root->right = insert(root->right, value);
        }

        return root; // Return the root after insertion
}
```

```
// Function for reverse-in-order traversal
void ReverseInOrder(Node* root)
{
	if (root == NULL) // If tree is empty
	{
		return;
	}

	ReverseInOrder(root->right); // Traverse the right subtree
	cout << root->data << " "; // Print the current node's data
	ReverseInOrder(root->left); // Traverse the left subtree
}

int main()
{
	Node* root = NULL; // Initialize an empty BST

	// Insert nodes into the BST
	root = insert(root, 19);
	root = insert(root, 33);
	root = insert(root, 52);
	root = insert(root, 28);
	root = insert(root, 56);
	root = insert(root, 16);

	// Display the tree nodes using reverse-in-order traversal
	cout << "Display Reverse-In-Order: ";
	inorder(root);

	return 0;
}
```

**Output:**

```
Display Reverse-In-Order: 56 52 33 28 19 16
```

.....

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

**Code:**

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

// Structure for node
struct Node
{
	int data; // Value of the node
	int count; // Count of occurrences of the value
	struct Node* left = NULL; // Pointer to the left child
```

```
        struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
        if (root == NULL) // If tree is empty or reaching a leaf
        {
                root = new Node(); // Create a new node
                root->data = value; // Assign the value to the node
                return root; // Return the new node as the root
        }

        // If the value already exists, increment its count
        else if (value == root->data)
        {
                root->count++;
                return root;
        }
        // Recur on the left subtree if the value is smaller
        else if (value < root->data)
        {
                root->left = insert(root->left, value);
        }
        // Recur on the right subtree if the value is larger
        else
        {
                root->right = insert(root->right, value);
        }

        return root; // Return the root after insertion
}

// Function for in-order traversal
void inorder(Node* root)
{
        if (root == NULL) // If tree is empty
        {
                return;
        }

        inorder(root->left); // Traverse the left subtree
        cout << root->data << "(" << root->count << ") "; // Print the data and its count
        inorder(root->right); // Traverse the right subtree
}

int main()
{
        Node* root = NULL; // Initialize an empty BST
```

```
        // Insert nodes into the BST
        root = insert(root, 19);
        root = insert(root, 7);
        root = insert(root, 3);
        root = insert(root, 26);
        root = insert(root, 37);
        root = insert(root, 49);
        root = insert(root, 19); // Duplicate value
        root = insert(root, 7); // Duplicate value
        root = insert(root, 3); // Duplicate value

        // Display the BST in in-order traversal
        cout << "Display In-Order: ";
        inorder(root);

        return 0;
}
```

**Output:**

```
Display In-Order: 3(1) 7(1) 19(1) 26(0) 37(0) 49(0)
```

.....

**Q: 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.**

**Code:**

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

// Structure for Node
struct Node
{
        int data; // Value of the node
        struct Node* left = NULL; // Pointer to the left child
        struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
        if (root == NULL) // If tree is empty or reaching a leaf
        {
                root = new Node(); // Create a new node
                root->data = value; // Assign the value to the node
                return root; // Return the new node as the root
        }

        // Recur on the left subtree if the value is smaller
        else if (value < root->data)
        {
```

```
            root->left =insert(root->left, value);
        }
        // Recur on the right subtree if the value is larger
        else if (value > root->data)
        {
            root->right = insert(root->right, value);
        }

        return root; // Return the root after insertion
}

// Function for in-order successor
Node* getSuccessor(Node* root)
{
        root = root->right; // Start from the right subtree
        while (root != nullptr && root->left != nullptr)
        {
            root = root->left; // Traverse left to find the smallest value
        }
        return root; // Return the in-order successor
}

// Function for Deletion
Node* deletion(Node* root, int value)
{
        if (root == NULL)   // If tree is empty
        {
            return root;
        }

        // Recur on the left subtree if the value is smaller
        if (value < root->data)
        {
            root->left = deletion(root->left, value);
        }
        // Recur on the right subtree if the value is larger
        else if (value > root->data)
        {
            root->right = deletion(root->right, value);
        }
        // Node to be deleted is found
        else
        {
            // Case 1: Node has no children or only one child
            if (root->left == NULL)
            {
                Node* temp = root->right; // Replace with right child
                delete root; // Delete the node
                return temp;
            }
```

```
            else if (root->right == NULL)
            {
                Node* temp = root->left; // Replace with left child
                delete root; // Delete the node
                return temp;
            }

            // Case 2: Node has two children
            Node* temp = getSuccessor(root); // Find in-order successor
            root->data = temp->data; // Replace data with successor's value
            root->right = deletion(root->right, temp->data); // Delete successor
        }
        return root; // Return the updated root
}

// Function for in-order traversal
void inorder(Node* root)
{
        if (root == NULL) // If tree is empty
        {
            return;
        }

        inorder(root->left); // Traverse the left subtree
        cout << root->data << " "; // Print the current node's data
        inorder(root->right); // Traverse the right subtree
}

int main()
{
        Node* root = NULL; // Initialize an empty BST

        // Insert nodes into the BST
        root = insert(root, 19);
        root = insert(root, 33);
        root = insert(root, 52);
        root = insert(root, 28);
        root = insert(root, 56);
        root = insert(root, 16);

        // Display the BST before deletion
        cout << "Before Deletion: ";
        inorder(root);

        // Delete nodes with values 52 and 33
        root = deletion(root, 52);
        root = deletion(root, 33);

        // Display the BST after deletion
        cout << "\nAfter Deletion: ";
```

```
    inorder(root);

    return 0;
}
```

**Output:**

```
Before Deletion: 16 19 28 33 52 56
After Deletion: 16 19 28 56
```

.....