Assingment
Double

Q no 1:
Code:

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

// Define a Node of the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) : data(val), prev(NULL), next(NULL) {}
};

// Function to delete the first node of a doubly linked list
void deleteFirstNode(Node*& head) {
    if (head == NULL) { // If the list is empty
        cout << "The list is already empty." << endl;
        return;
    }

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

    if (head != NULL) { // If the list is not empty after deletion
        head->prev = NULL;
    }

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

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

// Function to insert a node at the end of the doubly linked list
```

```cpp
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);

    if (head == NULL) { // If the list is empty
        head = newNode;
        return;
    }

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

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

int main() {
    Node* head = NULL;

    // Add nodes to the doubly linked list
    appendNode(head, 40);
    appendNode(head, 43);
    appendNode(head, 45);

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

    // Delete the first node
    deleteFirstNode(head);

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

    return 0;
}
```

Output:

```
Original list: 40 43 45
First node deleted successfully.
List after deleting the first node: 43 45

--------------------------------
Process exited after 1.813 seconds with return value 0
Press any key to continue . . .
```

Q no 2.

Code:

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

// Define a Node of the doubly linked list
struct Node {
   int data;
   Node* prev;
   Node* next;

   Node(int val) : data(val), prev(NULL), next(NULL) {}
};

// Function to delete a node by its value in a doubly linked list
void deleteNodeByValue(Node*& head, int value) {
   if (head == NULL) { // If the list is empty
      cout << "The list is empty." << endl;
      return;
   }

   Node* temp = head;

   // Search for the node with the specified value
   while (temp != NULL && temp->data != value) {
      temp = temp->next;
   }

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

```
    // Node with the value found
    if (temp == head) { // If it's the head node
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
    } else if (temp->next == NULL) { // If it's the last node
        temp->prev->next = NULL;
    } else { // If it's a middle node
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }

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

// Function to display the doubly linked list
void displayList(Node* head) {
    Node* temp = head;
    if (temp == NULL) {
        cout << "The list is empty." << endl;
        return;
    }
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to insert a node at the end of the doubly linked list
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);

    if (head == NULL) { // If the list is empty
        head = newNode;
        return;
    }

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

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

```
}

int main() {
    Node* head = NULL;

    // Add nodes to the doubly linked list
    appendNode(head, 100);
    appendNode(head, 200);
    appendNode(head, 300);
    appendNode(head, 400);

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

    // Delete a node by value
    deleteNodeByValue(head, 200);

    cout << "List after deleting the node with value 200: ";
    displayList(head);

    // Try deleting a non-existent value
    deleteNodeByValue(head, 500);

    return 0;
}
```

Output:

```
C:\Users\AS\Desktop\123\Nev

Original list: 100 200 300 400
Node with value 200 deleted successfully.
List after deleting the node with value 200: 100 300 400
Value 500 not found in the list.

--------------------------------
Process exited after 1.943 seconds with return value 0
Press any key to continue . . .
```

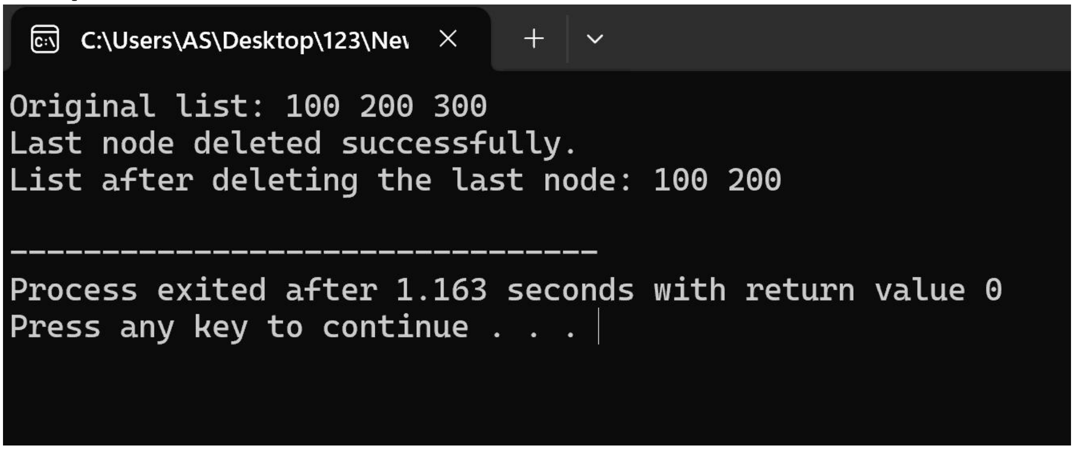
Q no 3 :

Code:

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

// Define a Node of the doubly linked list
struct Node {
   int data;
   Node* prev;
   Node* next;

   Node(int val) : data(val), prev(NULL), next(NULL) {}
};

// Function to delete the last node of a doubly linked list
void deleteLastNode(Node*& head) {
   if (head == NULL) { // If the list is empty
      cout << "The list is already empty." << endl;
      return;
   }

   if (head->next == NULL) { // If the list has only one node
      delete head;
      head = NULL;
      cout << "Last node deleted successfully." << endl;
      return;
   }

   Node* temp = head;

   // Traverse to the last node
   while (temp->next != NULL) {
      temp = temp->next;
   }

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

   delete temp; // Free the memory of the last node
   cout << "Last node deleted successfully." << endl;
}

// Function to display the doubly linked list
void displayList(Node* head) {
   Node* temp = head;
   while (temp != NULL) {
```

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

// Function to insert a node at the end of the doubly linked list
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);

    if (head == NULL) { // If the list is empty
        head = newNode;
        return;
    }

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

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

int main() {
    Node* head = NULL;

    // Add nodes to the doubly linked list
    appendNode(head, 100);
    appendNode(head, 200);
    appendNode(head, 300);

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

    // Delete the last node
    deleteLastNode(head);

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

    return 0;
}
```

Output:



Q no 4:

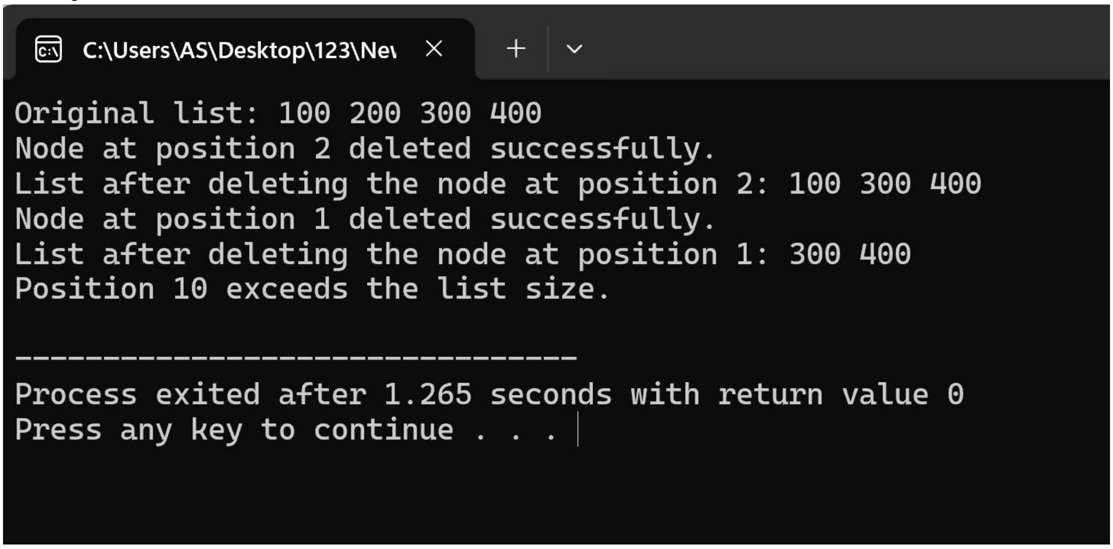
Code:

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

// Define a Node of the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) : data(val), prev(NULL), next(NULL) {}
};

// Function to delete a node at a specific position in a doubly linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == NULL) { // If the list is empty
        cout << "The list is empty." << endl;
        return;
    }

    if (position <= 0) { // Invalid position
        cout << "Invalid position. Position should be greater than 0." << endl;
        return;
    }

    Node* temp = head;
    int currentIndex = 1;

    // Traverse the list to find the node at the specified position
    while (temp != NULL && currentIndex < position) {
        temp = temp->next;
        currentIndex++;
    }
```

```
    if (temp == NULL) { // Position exceeds the size of the list
        cout << "Position " << position << " exceeds the list size." << endl;
        return;
    }

    // If the node to be deleted is the head
    if (temp == head) {
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
    } else if (temp->next == NULL) { // If it's the last node
        temp->prev->next = NULL;
    } else { // If it's a middle node
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }

    delete temp; // Free the memory of the node
    cout << "Node at position " << position << " deleted successfully." << endl;
}

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

// Function to insert a node at the end of the doubly linked list
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);

    if (head == NULL) { // If the list is empty
        head = newNode;
        return;
    }

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

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

int main() {
    Node* head = NULL;

    // Add nodes to the doubly linked list
    appendNode(head, 100);
    appendNode(head, 200);
    appendNode(head, 300);
    appendNode(head, 400);

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

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

    cout << "List after deleting the node at position 2: ";
    displayList(head);

    // Delete the head node
    deleteNodeAtPosition(head, 1);

    cout << "List after deleting the node at position 1: ";
    displayList(head);

    // Try deleting at an invalid position
    deleteNodeAtPosition(head, 10);

    return 0;
}
```

Output:



Q no 5:

Code:

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

// Define a Node of the doubly linked list
struct Node {
   int data;
   Node* prev;
   Node* next;

   Node(int val) : data(val), prev(NULL), next(NULL) {}
};

// Function to perform forward traversal of the doubly linked list
void forwardTraversal(Node* head) {
   cout << "Forward Traversal: ";
   Node* temp = head;
   while (temp != NULL) {
      cout << temp->data << " ";
      temp = temp->next;
   }
   cout << endl;
}

// Function to perform reverse traversal of the doubly linked list
void reverseTraversal(Node* head) {
   if (head == NULL) { // If the list is empty
      cout << "Reverse Traversal: List is empty." << endl;
```

```
        return;
    }

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

    // Traverse backward from the last node
    cout << "Reverse Traversal: ";
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}

// Function to append a node to the end of the doubly linked list
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);

    if (head == NULL) { // If the list is empty
        head = newNode;
        return;
    }

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

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

// Function to delete a node at a specific position in the doubly linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == NULL) { // If the list is empty
        cout << "The list is empty." << endl;
        return;
    }

    if (position <= 0) { // Invalid position
        cout << "Invalid position. Position should be greater than 0." << endl;
        return;
    }
```

```
    Node* temp = head;
    int currentIndex = 1;

    // Traverse to the node at the specified position
    while (temp != NULL && currentIndex < position) {
        temp = temp->next;
        currentIndex++;
    }

    if (temp == NULL) { // Position exceeds the list size
        cout << "Position " << position << " exceeds the list size." << endl;
        return;
    }

    if (temp == head) { // Deleting the head node
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
    } else if (temp->next == NULL) { // Deleting the last node
        temp->prev->next = NULL;
    } else { // Deleting a middle node
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }

    delete temp; // Free the memory of the node
    cout << "Node at position " << position << " deleted successfully." << endl;
}

// Main function to demonstrate forward and reverse traversal
int main() {
    Node* head = NULL;

    // Add nodes to the doubly linked list
    appendNode(head, 100);
    appendNode(head, 200);
    appendNode(head, 300);
    appendNode(head, 400);

    // Perform forward and reverse traversal before deletion
    forwardTraversal(head);
    reverseTraversal(head);

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

```
    // Perform forward and reverse traversal after deletion
    forwardTraversal(head);
    reverseTraversal(head);

    return 0;
}
```

Output:

```
C:\Users\AS\Desktop\123\Nev

Forward Traversal: 100 200 300 400
Reverse Traversal: 400 300 200 100
Node at position 2 deleted successfully.
Forward Traversal: 100 300 400
Reverse Traversal: 400 300 100

--------------------------------
Process exited after 1.366 seconds with return value 0
Press any key to continue . . .
```

Single Cirular:

Q no 1:

Code:

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

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

// Function to delete a node at a specific position in a circular linked list
void deleteNodeAtPosition(Node*& head, int position) {
   if (head == NULL) { // List is empty
      cout << "List is empty. Nothing to delete." << endl;
      return;
   }

   // If the position is 0, delete the head node
   if (position == 0) {
      // If there's only one node
      if (head->next == head) {
         delete head;
         head = NULL;
      } else {
         Node* last = head;
         while (last->next != head) { // Find the last node
            last = last->next;
         }
         Node* temp = head;
         head = head->next; // Move the head pointer
         last->next = head; // Adjust the last node's next pointer
         delete temp; // Delete the old head
      }
      return;
   }

   Node* current = head;
   Node* previous = NULL;
   int count = 0;

   // Traverse to the desired position
   while (current->next != head && count < position) {
      previous = current;
```

```
        current = current->next;
        count++;
    }

    // If position is out of bounds
    if (current->next == head && count < position) {
        cout << "Position out of bounds." << endl;
        return;
    }

    // Delete the node
    previous->next = current->next;
    delete current;
}

// Function to insert a node at the end of the circular linked list
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
        return;
    }

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

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

// Function to display the circular linked list
void display(Node* head) {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }

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

```
}

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

    // Insert some nodes
    insert(head, 100);
    insert(head, 200);
    insert(head, 300);
    insert(head, 400);

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

    // Delete node at position 2
    deleteNodeAtPosition(head, 2);
    cout << "After deleting node at position 2: ";
    display(head);

    // Try to delete node at an invalid position
    deleteNodeAtPosition(head, 5);
    cout << "After attempting to delete node at position 5: ";
    display(head);

    return 0;
}
```

Output:

```
C:\Users\AS\Desktop\123\Nev

Original list: 100 200 300 400
After deleting node at position 2: 100 200 400
Position out of bounds.
After attempting to delete node at position 5: 100 200 400

--------------------------------
Process exited after 1.305 seconds with return value 0
Press any key to continue . . .
```

Q no 2:
Code:
```
#include <iostream>
using namespace std;

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

// Function to delete a node by its value in a circular linked list
void deleteNodeByValue(Node*& head, int value) {
    if (head == NULL) { // List is empty
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    Node* current = head;
    Node* previous = NULL;

    // Case 1: The node to be deleted is the only node in the list
    if (head->data == value && head->next == head) {
        delete head;
        head = NULL;
        return;
    }

    // Case 2: The node to be deleted is the head node
    if (head->data == value) {
        // Find the last node
        while (current->next != head) {
            current = current->next;
        }

        Node* temp = head;
        head = head->next;
        current->next = head;
        delete temp;
        return;
    }

    // Case 3: The node to be deleted is in the middle or end of the list
    do {
        previous = current;
        current = current->next;
```

```
        if (current->data == value) {
            previous->next = current->next;
            delete current;
            return;
        }
    } while (current != head);

    // If the value was not found
    cout << "Value " << value << " not found in the list." << endl;
}

// Function to insert a node at the end of the circular linked list
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
        return;
    }

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

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

// Function to display the circular linked list
void display(Node* head) {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }

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

// Main function
```

```
int main() {
    Node* head = NULL;

    // Insert some nodes
    insert(head, 100);
    insert(head, 200);
    insert(head, 300);
    insert(head, 400);

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

    // Delete a node by value
    deleteNodeByValue(head, 200);

    cout << "After deleting node with value 200: ";
    display(head);

    // Try to delete a node not in the list
    deleteNodeByValue(head, 500);

    cout << "After attempting to delete node with value 500: ";
    display(head);

    return 0;
}
```

Output:

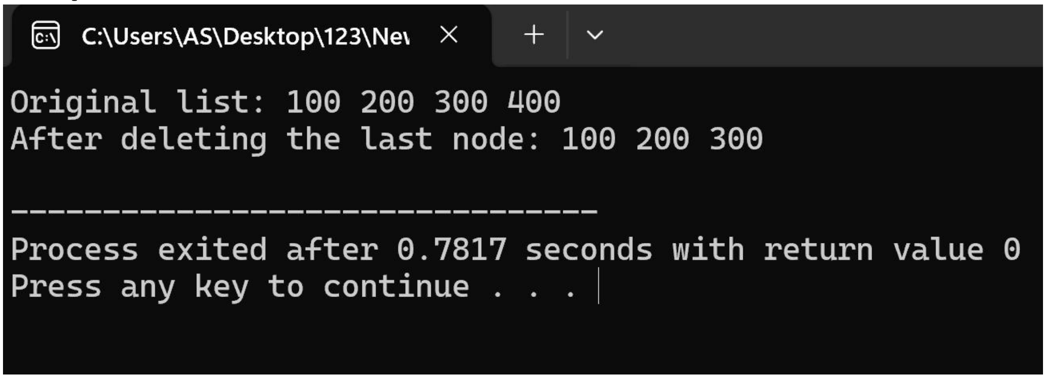
```
C:\Users\AS\Desktop\123\Nev

Original list: 100 200 300 400
After deleting node with value 200: 100 300 400
Value 500 not found in the list.
After attempting to delete node with value 500: 100 300 400

--------------------------------
Process exited after 1.286 seconds with return value 0
Press any key to continue . . .
```

Q no 3:
Code:

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

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

// Function to delete the last node of a circular linked list
void deleteLastNode(Node*& head) {
    if (head == NULL) { // List is empty
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    if (head->next == head) { // Only one node in the list
        delete head;
        head = NULL;
        return;
    }

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

    // Adjust pointers and delete the last node
    Node* last = current->next;
    current->next = head;
    delete last;
}

// Function to insert a node at the end of the circular linked list
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
        return;
    }
```

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

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

// Function to display the circular linked list
void display(Node* head) {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }

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

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

    // Insert some nodes
    insert(head, 100);
    insert(head, 200);
    insert(head, 300);
    insert(head, 400);

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

    // Delete the last node
    deleteLastNode(head);

    cout << "After deleting the last node: ";
    display(head);

    return 0;
}
```

Output:



Q no 4:

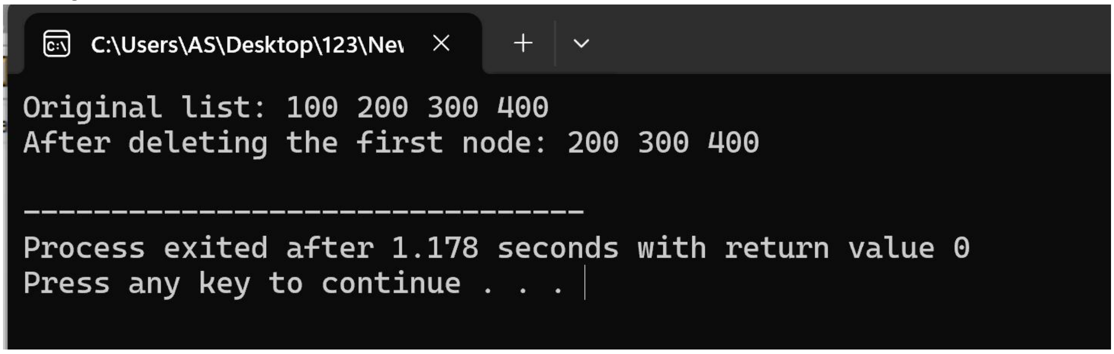
Code:

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

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

// Function to delete the first node of a circular linked list
void deleteFirstNode(Node*& head) {
    if (head == NULL) { // List is empty
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    if (head->next == head) { // Only one node in the list
        delete head;
        head = NULL;
        return;
    }

    // Find the last node in the list
    Node* last = head;
    while (last->next != head) {
        last = last->next;
    }

    // Point the last node to the second node
    Node* temp = head;
    head = head->next;
    last->next = head;

    // Delete the first node
```

```
    delete temp;
}

// Function to insert a node at the end of the circular linked list
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
        return;
    }

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

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

// Function to display the circular linked list
void display(Node* head) {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }

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

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

    // Insert some nodes
    insert(head, 100);
    insert(head, 200);
    insert(head, 300);
    insert(head, 400);
```

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

    // Delete the first node
    deleteFirstNode(head);

    cout << "After deleting the first node: ";
    display(head);

    return 0;
}
```

Output:



```
Original list: 100 200 300 400
After deleting the first node: 200 300 400

--------------------------------
Process exited after 1.178 seconds with return value 0
Press any key to continue . . .
```

Q no 5:

Code:

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

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

// Function to delete a node at a specific position in a circular linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == NULL) { // List is empty
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    // If the position is 0, delete the head node
    if (position == 0) {
```

```
        // If there's only one node
        if (head->next == head) {
            delete head;
            head = NULL;
        } else {
            Node* last = head;
            while (last->next != head) { // Find the last node
                last = last->next;
            }
            Node* temp = head;
            head = head->next; // Move the head pointer
            last->next = head; // Adjust the last node's next pointer
            delete temp; // Delete the old head
        }
        return;
    }

    Node* current = head;
    Node* previous = NULL;
    int count = 0;

    // Traverse to the desired position
    while (current->next != head && count < position) {
        previous = current;
        current = current->next;
        count++;
    }

    // If position is out of bounds
    if (current->next == head && count < position) {
        cout << "Position out of bounds." << endl;
        return;
    }

    // Delete the node
    previous->next = current->next;
    delete current;
}

// Function to insert a node at the end of the circular linked list
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
        return;
```

```
    }

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

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

// Function to display the circular linked list in forward traversal
void display(Node* head) {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }

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

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

    // Insert some nodes
    insert(head, 100);
    insert(head, 200);
    insert(head, 300);
    insert(head, 400);

    co;ut << "Original list: ";
    display(head);

    // Delete node at position 2
    deleteNodeAtPosition(head, 2);
    cout << "After deleting node at position 2: ";
    display(head);

    // Delete node at position 0 (head node)
    deleteNodeAtPosition(head, 0);
    cout << "Forward Traversal: ";
```

```
    display(head);

    return 0;
}
```

Output:

```
C:\Users\AS\Desktop\123\Nev

Original list: 100 200 300 400
After deleting node at position 2: 100 200 400
Forward Traversal: 200 400

--------------------------------
Process exited after 0.9451 seconds with return value 0
Press any key to continue . . .
```

Tree:

Q no 1:

Code:

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

// Node structure for Binary Search Tree
struct Node {
   int data;
   Node* left;
   Node* right;

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

// Function to insert a node in the Binary Search Tree
Node* insert(Node* root, int value) {
   if (root == NULL) {
      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 count the nodes in the Binary Search Tree
int countNodes(Node* root) {
   if (root == NULL) {
      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 and print the tree
void inorderTraversal(Node* root) {
```

```
    if (root != NULL) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}

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

    // Insert nodes into the BST
    root = insert(root, 100);
    root = insert(root, 50);
    root = insert(root, 150);
    root = insert(root, 25);
    root = insert(root, 75);
    root = insert(root, 125);
    root = insert(root, 175);

    cout << "In-order traversal of the Binary Search Tree: ";
    inorderTraversal(root);
    cout << endl;

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

    return 0;
}
```

Output:

```
C:\Users\AS\Desktop\123\Nev

In-order traversal of the Binary Search Tree: 25 50 75 100 125 150 175
Total number of nodes in the BST: 7

--------------------------------
Process exited after 1.379 seconds with return value 0
Press any key to continue . . .
```

Q no 2:
Code:
```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

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

// Function to delete a node
Node* deleteNode(Node* root, int key) {
    if (root == NULL) return root;

    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == NULL) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        // Node with two children: Get the inorder successor (smallest in the right
subtree)
```

```
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Function to perform an in-order traversal of the BST
void inorderTraversal(Node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

// Main function to test deletion
int main() {
    Node* root = new Node(10);
    root->left = new Node(5);
    root->right = new Node(15);
    root->left->left = new Node(3);
    root->left->right = new Node(7);
    root->right->left = new Node(12);
    root->right->right = new Node(18);

    cout << "Original BST (In-order Traversal): ";
    inorderTraversal(root);
    cout << endl;

    cout << "Deleting value 5." << endl;
    root = deleteNode(root, 5);

    cout << "BST after deleting value 5 (In-order Traversal): ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:

```
C:\Users\AS\Desktop\123\Nev

Original BST (In-order Traversal): 3 5 7 10 12 15 18
Deleting value 5.
BST after deleting value 5 (In-order Traversal): 3 7 10 12 15 18

--------------------------------
Process exited after 1.134 seconds with return value 0
Press any key to continue . . .
```

Q no 3:

Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

// Function to insert a value (with duplicate check)
Node* insert(Node* root, int val) {
    if (root == NULL) {
        return new Node(val);
    }
    if (val < root->data) {
        root->left = insert(root->left, val);
    } else if (val > root->data) {
        root->right = insert(root->right, val);
    } else {
        cout << "Duplicate value " << val << " not allowed." << endl;
    }
    return root;
}
```

```cpp
// Function to perform an in-order traversal of the BST
void inorderTraversal(Node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

// Main function to test duplication handling
int main() {
    Node* root = NULL;

    cout << "Inserting values into the BST..." << endl;
    root = insert(root, 10);
    root = insert(root, 5);
    root = insert(root, 15);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 12);
    root = insert(root, 10); // Attempt to insert a duplicate value
    root = insert(root, 18);

    cout << "\nIn-order traversal of the BST: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:

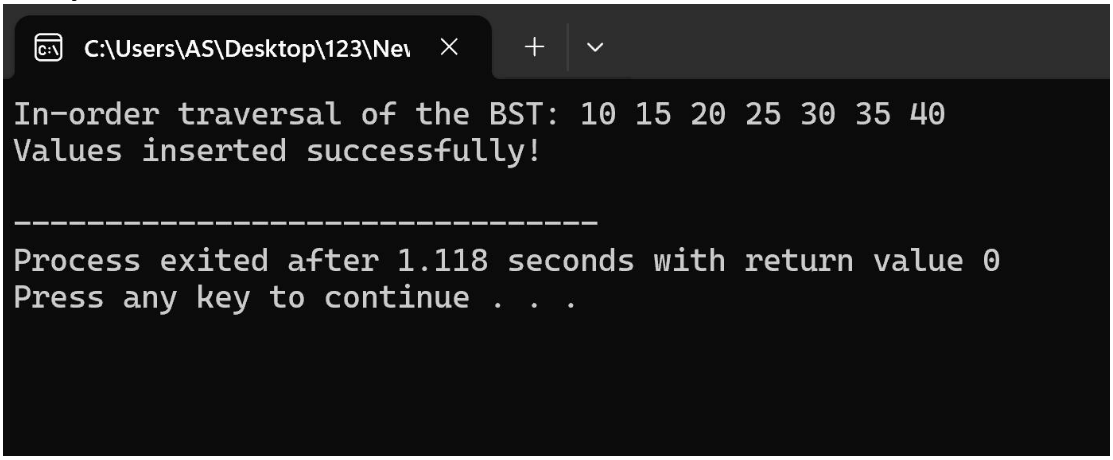
```
C:\Users\AS\Desktop\123\Ne

Inserting values into the BST...
Duplicate value 10 not allowed.

In-order traversal of the BST: 3 5 7 10 12 15 18

--------------------------------
Process exited after 0.995 seconds with return value 0
Press any key to continue . . .
```

Q no 4:
Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to initialize a new node
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

// Function to perform in-order traversal
void inorderTraversal(Node* root) {
    if (root == NULL) return; // Base case: if the node is NULL, return
    inorderTraversal(root->left);  // Traverse the left subtree
    cout << root->data << " ";     // Visit the current node
    inorderTraversal(root->right); // Traverse the right subtree
}

// Function to insert a new value into the BST
Node* insertNode(Node* root, int val) {
    if (root == NULL) {
        return new Node(val); // Create and return a new node if root is NULL
    }
    if (val < root->data) {
        root->left = insertNode(root->left, val); // Insert into the left subtree
    } else if (val > root->data) {
        root->right = insertNode(root->right, val); // Insert into the right subtree
    }
    return root; // Return the unchanged root
}

// Main function to test insertion and traversal
int main() {
    Node* root = NULL; // Initialize the root of the BST

    // Insert new values into the BST
    root = insertNode(root, 25);
```

```
    root = insertNode(root, 15);
    root = insertNode(root, 35);
    root = insertNode(root, 10);
    root = insertNode(root, 20);
    root = insertNode(root, 30);
    root = insertNode(root, 40);

    // Perform an in-order traversal
    cout << "In-order traversal of the BST: ";
    inorderTraversal(root);
    cout << endl;

    cout << "Values inserted successfully!" << endl;

    return 0;
}
```

Output:



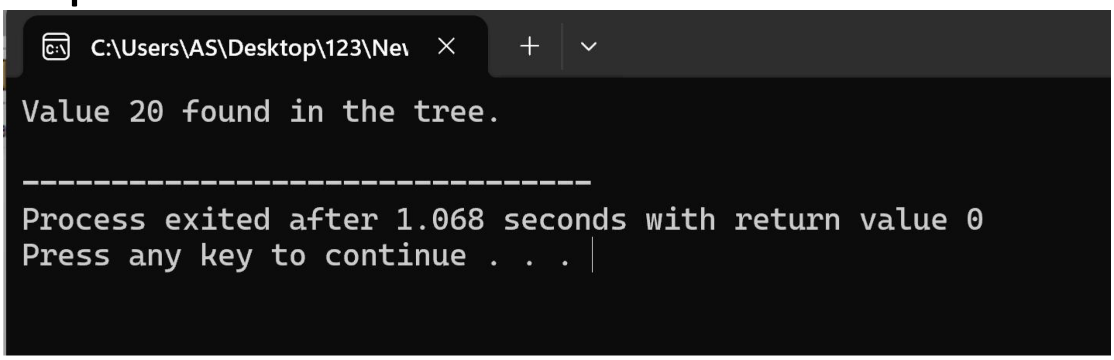
Q no 5:

Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
```

```
};

// Function to search for a value
bool search(Node* root, int key) {
   if (root == NULL) return false;       // If root is NULL, key not found
   if (root->data == key) return true;   // If key matches the current node
   if (key < root->data)                 // If key is smaller, search the left subtree
      return search(root->left, key);
   return search(root->right, key);      // If key is greater, search the right subtree
}

// Main function to test searching
int main() {
   // Create a tree
   Node* root = new Node(25);
   root->left = new Node(15);
   root->right = new Node(35);
   root->left->left = new Node(10);
   root->left->right = new Node(20);
   root->right->left = new Node(30);
   root->right->right = new Node(40);

   // Value to search
   int searchKey = 20;
   if (search(root, searchKey)) {
      cout << "Value " << searchKey << " found in the tree." << endl;
   } else {
      cout << "Value " << searchKey << " not found in the tree." << endl;
   }

   return 0;
}
```

Output:



Q no 6:

Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

// Traversal functions
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

void preorder(Node* root) {
    if (root == NULL) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

// Main function to test traversals
int main() {
    // Create tree with different values
    Node* root = new Node(25);
    root->left = new Node(15);
    root->right = new Node(35);
    root->left->left = new Node(10);
```

```
    root->left->right = new Node(20);
    root->right->left = new Node(30);
    root->right->right = new Node(40);

    // Perform traversals
    cout << "Inorder: ";
    inorder(root);
    cout << endl;

    cout << "Preorder: ";
    preorder(root);
    cout << endl;

    cout << "Postorder: ";
    postorder(root);
    cout << endl;

    return 0;
}
```

Output: