List of all Data Structures and Algorithms used in the project:

Data Structures used:

1. Binary Search Tree

Note: The binary tree is implemented as a linked list.

Algorithms used in the program:

1. In-Order traversal (Recursive and Iterative)
2. Pre-Order traversal (Recursive and Iterative)
3. Post-Order traversal (Recursive and Iterative)
4. Finding the level of a node in the tree
5. Checking if two nodes are siblings
6. Checking if two nodes are cousins

Function-wise explanation of algorithms used in the program:

A. Recursive In-Order traversal (inOrder()):

1. Check if the root node is NULL.
2. If the root node is not NULL, call the inOrder function with the left child node of the root node as an argument.
3. Print the value of the root node's data.
4. Call the inOrder function with the right child node of the root node as an argument.
5. Repeat steps 1-4 until all nodes in the tree have been visited and printed.

B. Non-recursive In-Order traversal(inOrdernr()):

1. Initialize a pointer current to the root node of the binary tree.
2. Initialize an array stack of size 100 to store the nodes and an integer variable top with value -1.
3. Start a while loop that continues as long as current is not NULL or top is greater than or equal to 0.
4. Within the while loop, start another while loop that continues as long as current is not NULL.
5. Within the second while loop, increment the value of top and store the value of current in the stack at the index top.
6. Update the value of current to the left child of the current node.
7. End the second while loop.
8. Set the value of current to the node at the top of the stack (i.e. stack[top--]), decrement the value of top, and print the data value of the node.
9. Update the value of current to the right child of the current node.
10. End the first while loop.

C. Recursive Pre-Order traversal (preOrder()):

1. Start with the preOrder function, which takes a pointer to the root node of a binary tree as input.

2. If the root node is NULL, return from the function.
3. Print the data value of the root node.
4. Call the preOrder function recursively with the left child of the root node as input.
5. Call the preOrder function recursively with the right child of the root node as input.
6. End the function.

D. Non-Recursive Pre-Order Traversal (preOrdernr()):

1. Initialize a pointer root to the root node of the binary tree.
2. If the root node is NULL, return from the function.
3. Initialize an array stack of size 100 to store the nodes and an integer variable top with value -1.
4. Start an infinite loop.
5. Within the infinite loop, start another while loop that continues as long as root is not NULL.
6. Within the second while loop, print the data value of the root node, increment the value of top, and store the value of root in the stack at the index top.
7. Update the value of root to the left child of the current node.
8. End the second while loop.
9. If top is less than 0, break from the infinite loop.
10. Set the value of root to the node at the top of the stack (i.e. stack[top--]), decrement the value of top, and update the value of root to the right child of the current node.
11. End the infinite loop.

E. Recursive Post-Order Traversal (postOrder()):

1. Start with the postOrder function, which takes a pointer to the root node of a binary tree as input.
2. If the root node is NULL, return from the function.
3. Call the postOrder function recursively with the left child of the root node as input.
4. Call the postOrder function recursively with the right child of the root node as input.
5. Print the data value of the root node.
6. End the function.

F. Non-Recursive Post Order Traversal:

1. Initialize an array "stack" with a maximum size of 100 to store nodes during the traversal, and a variable "top" with a value of -1 to keep track of the top of the stack.
2. Set a variable "cur" equal to the root of the binary tree and a variable "prev" equal to NULL.
3. Start a while loop that continues until "cur" is NULL and "top" is -1.
4. Within the while loop, start another while loop that continues until "cur" is NULL.
   a. Within this inner while loop, push "cur" onto the top of the stack, and set "cur" equal to its left child.
5. Set "cur" equal to the top node on the stack.
6. Check if either "cur->right" is NULL or "cur->right" is equal to "prev". If either of these conditions is true:
   a. Print the value of "cur->data".
   b. Pop the top node from the stack and decrement "top".
   c. Set "prev" equal to "cur".
   d. Set "cur" equal to NULL.
7. If the conditions in step 6 are not met, set "cur" equal to "cur->right".
8. Repeat the process from step 5 until "cur" is NULL and "top" is -1.

G. Function to insert a new node:

1. Start with the insert function, which takes a pointer to a binary tree node and an integer value data as input.
2. If the node is NULL, the function creates a new node with the given data value using the newNode function.
3. If the given data value is less than the data value of the node, the function calls itself recursively with the left child of the node and the data value as input.
4. If the given data value is greater than the data value of the node, the function calls itself recursively with the right child of the node and the data value as input.
5. Return the node.
6. End the function.

H. newNode Function:

It takes an integer value data as input and returns a pointer to a newly created binary tree node. The function does the following:

1. Allocates memory for a new struct Node using the malloc function, which takes the size of the struct Node data structure as input.
2. Assigns the value of data to the data field of the newly created node.
3. Initializes the left and right pointers of the node to NULL to indicate that the node currently has no children.
4. Returns a pointer to the newly created node.

This function can be used to create a new node in a binary tree with a specified data value and with no children.

I. Functions to check whether two given nodes are siblings or not:

a. isSibling():

1. Start with the function isSibling, which takes a pointer to a binary tree node, root, and two integer values x and y as input.
2. If root is NULL, return 0.
3. Check if root has a left child root->left and a right child root->right, and if the data values of the left and right children are equal to x and y respectively, or if the data values of the left and right children are equal to y and x respectively. If either of these conditions is met, return 1 as x and y are siblings.
4. Recursively call isSibling with the left child of root and x, y as input. If the result of this call is 1, return 1 as x and y are siblings.
5. Recursively call isSibling with the right child of root and x, y as input. If the result of this call is 1, return 1 as x and y are siblings.
6. If none of the above conditions is met, return 0 as x and y are not siblings.
7. End the function.

b. areSiblings(): (Boolean Function)

1. If the root is NULL, return false.

2. Check if the left child of the root contains data 'a' and the right child contains data 'b' or vice versa.
3. If step 2 is true, return true.
4. Recursively call the function on the left child and right child of the root, passing 'a' and 'b' as arguments.
5. If either of the calls in step 4 returns true, return true.
6. If none of the above steps return true, return false.

J. Function to find the level of a given node in the binary tree (edepth()):

1. If the root is NULL, return 0.
2. If the data in the root node is equal to x, return the current level.
3. Recursively call the function on the left child of the root, passing x and level + 1 as arguments. Store the result in a variable 'downlevel'.
4. If downlevel is not equal to 0, return downlevel.
5. Recursively call the function on the right child of the root, passing x and level + 1 as arguments. Store the result in the same 'downlevel' variable.
6. Return downlevel.

K. Function to determine whether two given nodes are siblings or not(areCousins()):

1. Get the level of node x by calling the edepth function and passing root, x and 1 as parameters. Store the result in a variable xLevel.
2. Get the level of node y by calling the edepth function and passing root, y and 1 as parameters. Store the result in a variable yLevel.
3. If xLevel is equal to yLevel and the nodes x and y are not siblings (checked by calling the areSiblings function and passing root, x, y as parameters), return 1.
4. If step 3 is not satisfied, return 0.

L. Main Function:

1. Initialize the binary tree with a root node as NULL.
2. Initialize variables for data, choice, x, y, a, b, height, n1, n2.
3. Display a menu of choices to the user and take the input.
4. Using a while loop, continue to take the user's choice until the user selects the exit option (choice 11).
5. Within the while loop, use a switch case to perform the selected action based on the user's choice. The actions are as follows:
   a. Case 1: Insert a new node with the given data into the binary tree.
   b. Case 2: Perform a recursive in-order traversal of the binary tree and display the result.
   c. Case 3: Perform a non-recursive in-order traversal of the binary tree and display the result.
   d. Case 4: Perform a recursive pre-order traversal of the binary tree and display the result.
   e. Case 5: Perform a non-recursive pre-order traversal of the binary tree and display the result.
   f. Case 6: Perform a recursive post-order traversal of the binary tree and display the result.
   g. Case 7: Perform a non-recursive post-order traversal of the binary tree and display the result.
   h. Case 8: Take two nodes as input and check if they are cousins. If they are, display "The two nodes are cousins." If not, display "The two nodes are not cousins."

i. Case 9: Take two nodes as input and check if they are siblings. If they are, display "The two nodes are siblings." If not, display "The two nodes are not siblings."
j. Case 10: Take a node as input and find its level in the binary tree. Display the result.

6. End the program when the user selects the exit option.