DSA ASSIGNMENT 3

Name: Varun Vivek Pai
USN: 01FE23BEC126
Roll No.: 332
Div: C

1. You are given a set of integers: [45, 23, 78, 12, 34, 56, 89, 67, 05, 99]. Construct a Binary Search Tree (BST) by inserting these elements in the given order. Subsequently, demonstrate the step-by-step process for:
   a) Deleting the node with value 34 (a leaf node).
   b) Deleting the node with value 78 (a node with two children).
   c) Inserting the value 40 into the modified BST. Draw the state of the BST after each operation.

Soln:

```
#include <stdio.h>

#include <stdlib.h>


typedef struct Node

{

   int data;

   struct Node* left;

   struct Node* right;

} Node;


Node* insert(Node* root, int data)

{

   Node* nw, *cur, *parent;

   nw = (Node*) malloc(sizeof(Node));

   if (nw)

   {

      nw->data = data;
```

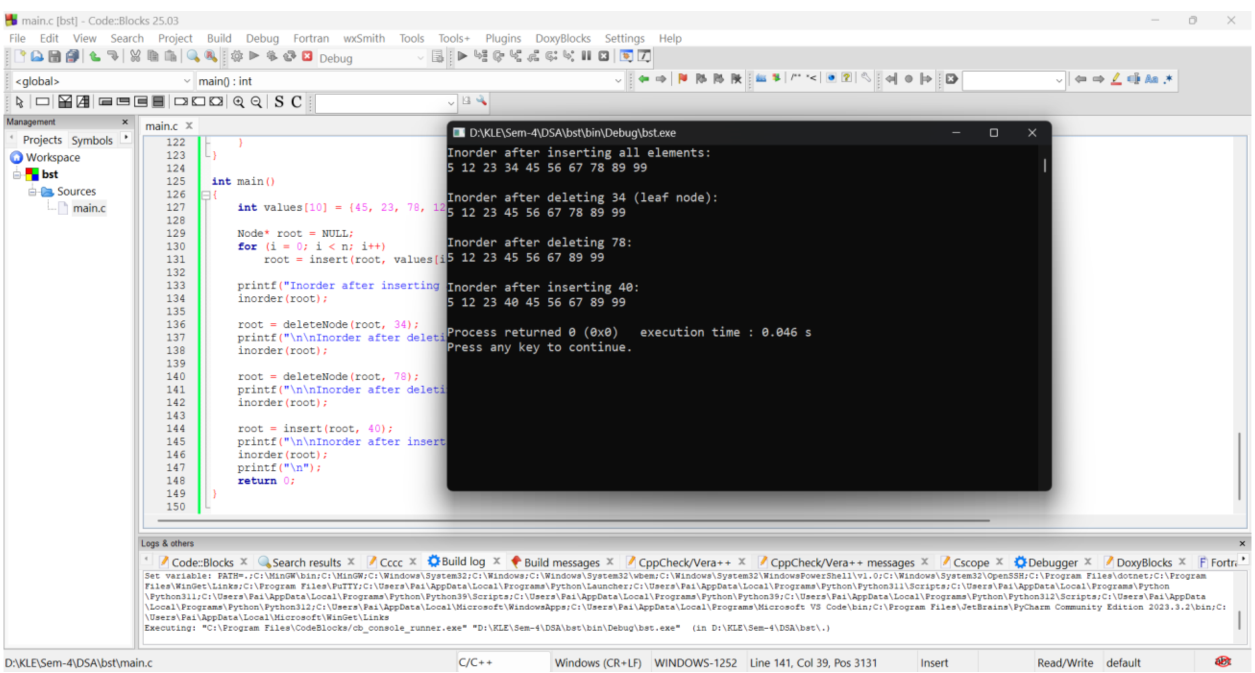
```
        nw->left=nw->right=NULL;
        if (root == NULL)
            return nw;
        for(cur = root, parent = NULL; cur != NULL;)
        {
            parent = cur;
            if (nw->data < cur->data)
                cur = cur->left;
            else
                cur = cur->right;
        }
        if (nw->data < parent->data)
            parent->left = nw;
        else
            parent->right = nw;
    }
    else
    {
        printf("\nNode allocation failed\n");
    }
    return root;
}

Node* deleteNode (Node*root, int data)
{
    struct Node *par, *cur, *temp;
    if (root == NULL)
    {
        printf ("\nEmpty Tree\n");
```

```
    return root;
}
par = NULL;
cur = root;
while (cur != NULL && cur -> data != data)
{
    par = cur;
    if (data < cur -> data)
        cur = cur -> left;
    else
        cur = cur -> right;
}
if (cur == NULL)
{
    printf ("\nNode not found\n");
    return root;
}
if (cur -> left != NULL && cur -> right != NULL)
{
    temp = par = cur;
    cur = cur -> left;
    while (cur -> right != NULL)
    {
        par = cur;
        cur = cur -> right;
    }
    temp -> data = cur -> data;
}
if (cur -> left != NULL && cur -> right == NULL)
```

```
{
   if (cur == root)
      root = root -> left;
   else
   {
      if (cur == par -> left)
         par -> left = cur -> left;
      else
         par -> right = cur -> left;
   }
}
else if (cur -> left == NULL && cur -> right != NULL)
{
   if (cur == root)
      root = root -> right;
   else
   {
      if (cur == par -> left)
         par -> left = cur -> right;
      else
         par -> right = cur -> right;
   }
}
else
{
   if (cur == root)
      root = NULL;
   else
   {
```

```
            if (par -> left == cur)
                par -> left = NULL;
            else
                par -> right = NULL;
        }
    }
    free(cur);
    return root;
}

void inorder(Node* root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main()
{
    int values[10] = {45, 23, 78, 12, 34, 56, 89, 67, 5, 99}, n=10, i;

    Node* root = NULL;
    for (i = 0; i < n; i++)
        root = insert(root, values[i]);

    printf("Inorder after inserting all elements:\n");
```

```
    inorder(root);

    root = deleteNode(root, 34);
    printf("\n\nInorder after deleting 34 (leaf node):\n");
    inorder(root);

    root = deleteNode(root, 78);
    printf("\n\nInorder after deleting 78:\n");
    inorder(root);

    root = insert(root, 40);
    printf("\n\nInorder after inserting 40:\n");
    inorder(root);
    printf("\n");
    return 0;
}
```



2. Consider an initially empty AVL tree. Insert the following sequence of values into the AVL tree, one by one: [10, 20, 30, 25, 28, 05, 08]. For each insertion, clearly show the balance factor of all affected nodes and demonstrate any necessary rotations (single or double) performed to maintain the AVL tree property. Draw the tree after each insertion and subsequent balancing.

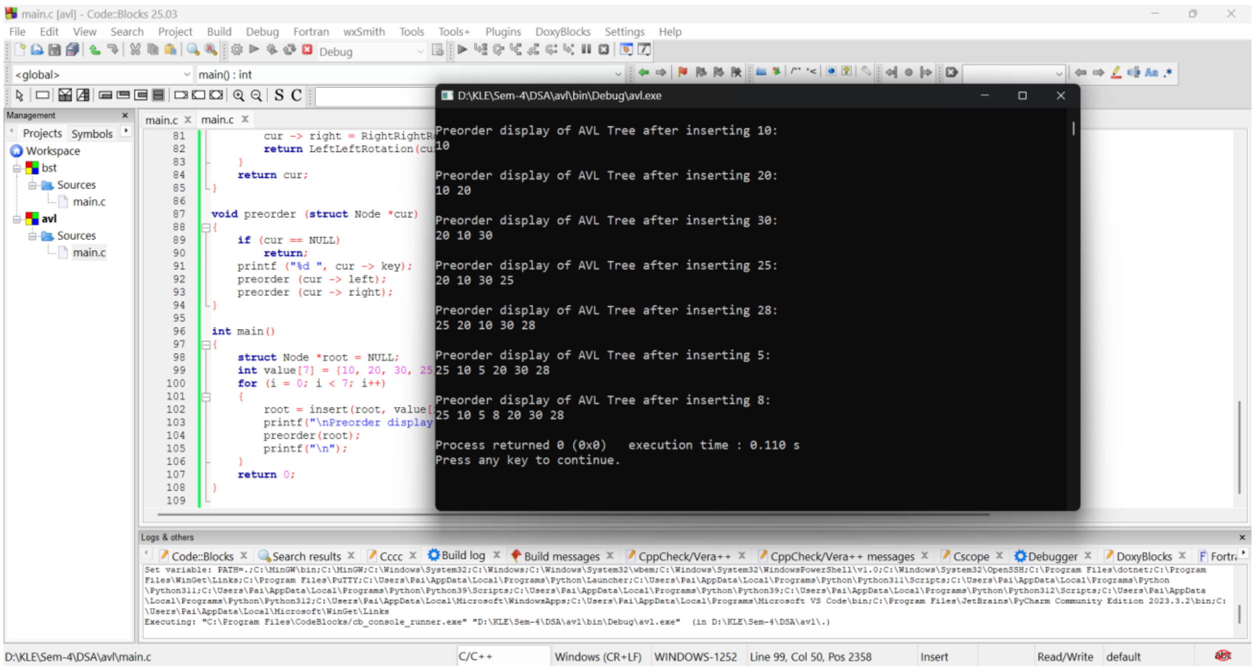
Soln:

```
#include <stdio.h>

#include <stdlib.h>


struct Node

{

   struct Node *left, *right;

   int h, key;

};


int height (struct Node *cur)

{

   if (cur == NULL)

      return 0;

   return cur -> h;

}


struct Node * getNode (int data)

{

   struct Node *nw;

   nw = malloc (sizeof (struct Node));

   nw -> key = data;

   nw -> left = nw -> right = NULL;

   nw -> h = 1;

   return nw;
```

```c
}

int max (int a, int b)
{
   return (a > b) ? a : b;
}

struct Node *LeftLeftRotation (struct Node *x)
{
   struct Node *y;
   y = x -> right;
   x -> right = y -> left;
   y -> left = x;
   x ->  h = max(height(x -> left), height (x -> right)) + 1;
   y ->  h = max(height(y -> left), height (y -> right)) + 1;
   return y;
}

struct Node *RightRightRotation (struct Node *x)
{
   struct Node *y;
   y = x -> left;
   x -> left = y -> right;
   y -> right = x;
   x ->  h = max(height(x -> left), height (x -> right)) + 1;
   y ->  h = max(height(y -> left), height (y -> right)) + 1;
   return y;
}
```

```
struct Node * insert (struct Node *cur, int data)
{
   int bal = 0;
   if (cur == NULL)
      return getNode (data);
   if (data < cur -> key)
      cur -> left = insert (cur -> left, data);
   else if (data > cur -> key)
      cur -> right = insert (cur -> right, data);
   else
   {
      printf ("\nNode already exists\n");
      return cur;
   }
   cur -> h = max(height(cur -> left), height (cur -> right)) + 1;
   bal = height (cur -> left) - height (cur -> right);
   if (bal > 1 && cur -> left -> key)
      return RightRightRotation (cur);
   else if (bal < -1 && data > cur -> right -> key)
      return LeftLeftRotation(cur);
   else if (bal > 1 && data < cur -> left -> key)
   {
      cur -> left = LeftLeftRotation(cur -> left);
      return RightRightRotation(cur);
   }
   else if (bal < -1 && data < cur -> right -> key)
   {
      cur -> right = RightRightRotation (cur -> right);
      return LeftLeftRotation(cur);
```

```
    }
    return cur;
}

void preorder (struct Node *cur)
{
    if (cur == NULL)
        return;
    printf ("%d ", cur -> key);
    preorder (cur -> left);
    preorder (cur -> right);
}

int main()
{
    struct Node *root = NULL;
    int value[7] = {10, 20, 30, 25, 28, 5, 8}, i;
    for (i = 0; i < 7; i++)
    {
        root = insert(root, value[i]);
        printf("\nPreorder display of AVL Tree after inserting %d:\n", value[i]);
        preorder(root);
        printf("\n");
    }
    return 0;
}
```



3. Given the following binary tree:

```
     50
    /  \
   30   70
  / \   / \
 20 40 60 80
```

a) Mention height and depth of the each node.
b) Represent given binary tree using linked list.
c) Perform an In-order traversal and list the nodes visited.
d) Perform a Pre-order traversal and list the nodes visited.
e) Perform a Post-order traversal and list the nodes visited.
f) Discuss a real-world application where each of these traversal methods would be most appropriate.

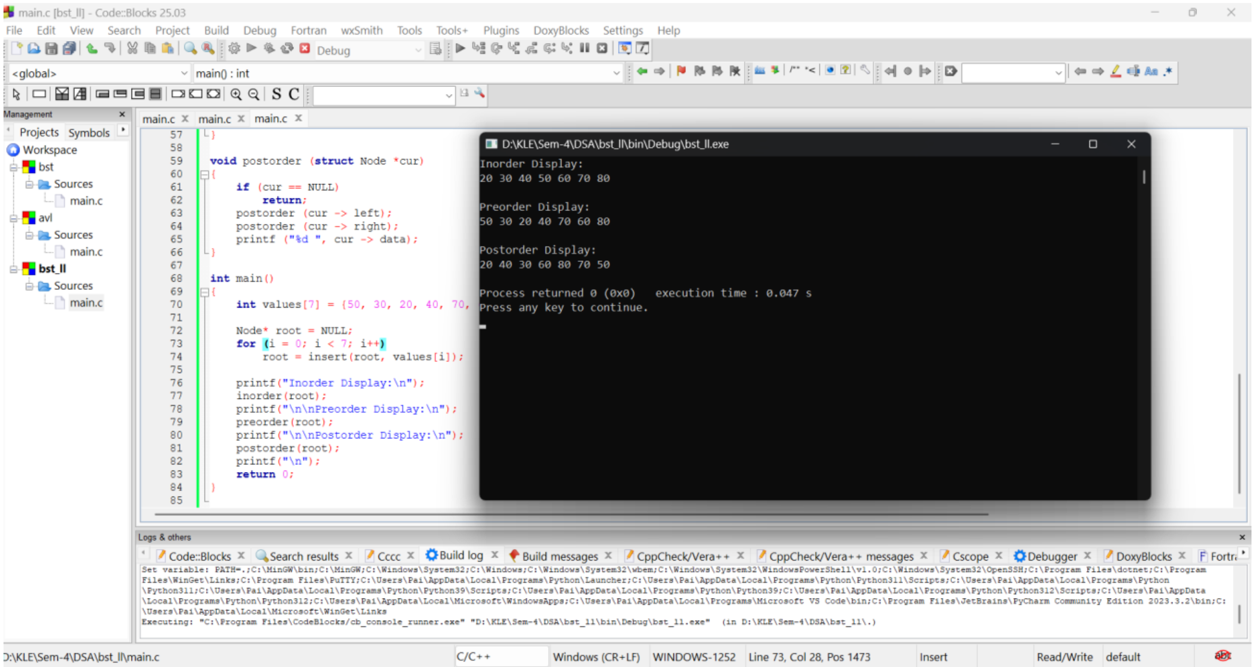
Soln:

```
#include <stdio.h>

#include <stdlib.h>


typedef struct Node

{
```

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

Node* insert(Node* root, int data)
{
    Node* nw, *cur, *parent;
    nw = (Node*) malloc(sizeof(Node));
    if (nw)
    {
        nw->data = data;
        nw->left=nw->right=NULL;
        if (root == NULL)
            return nw;
        for(cur = root, parent = NULL; cur != NULL;)
        {
            parent = cur;
            if (nw->data < cur->data)
                cur = cur->left;
            else
                cur = cur->right;
        }
        if (nw->data < parent->data)
            parent->left = nw;
        else
            parent->right = nw;
    }
    else
```

```
    {
        printf("\nNode allocation failed\n");
    }
    return root;
}

void preorder (struct Node *cur)
{
    if (cur == NULL)
        return;
    printf ("%d ", cur -> data);
    preorder (cur -> left);
    preorder (cur -> right);
}

void inorder (struct Node *cur)
{
    if (cur == NULL)
        return;
    inorder (cur -> left);
    printf ("%d ", cur -> data);
    inorder (cur -> right);
}

void postorder (struct Node *cur)
{
    if (cur == NULL)
        return;
    postorder (cur -> left);
```

```
    postorder (cur -> right);
    printf ("%d ", cur -> data);
}

int main()
{
    int values[7] = {50, 30, 20, 40, 70, 60, 80}, i;

    Node* root = NULL;
    for (i = 0; i < 7; i++)
        root = insert(root, values[i]);

    printf("Inorder Display:\n");
    inorder(root);
    printf("\n\nPreorder Display:\n");
    preorder(root);
    printf("\n\nPostorder Display:\n");
    postorder(root);
    printf("\n");
    return 0;
}
```



4. Consider the following unweighted, undirected graph:

Vertices: A, B, C, D, E, F Edges: (A, B), (A, C), (B, D), (C, E), (D, F), (E, F)

a) Represent this graph using an Adjacency Matrix.
b) Represent this graph using an Adjacency List.
c) Starting from vertex A, perform a Breadth-First Search (BFS) and list the order in which nodes are visited.
d) Starting from vertex A, perform a Depth-First Search (DFS) and list the order in which nodes are visited. (Assume alphabetical order for visiting adjacent unvisited nodes in both traversals).

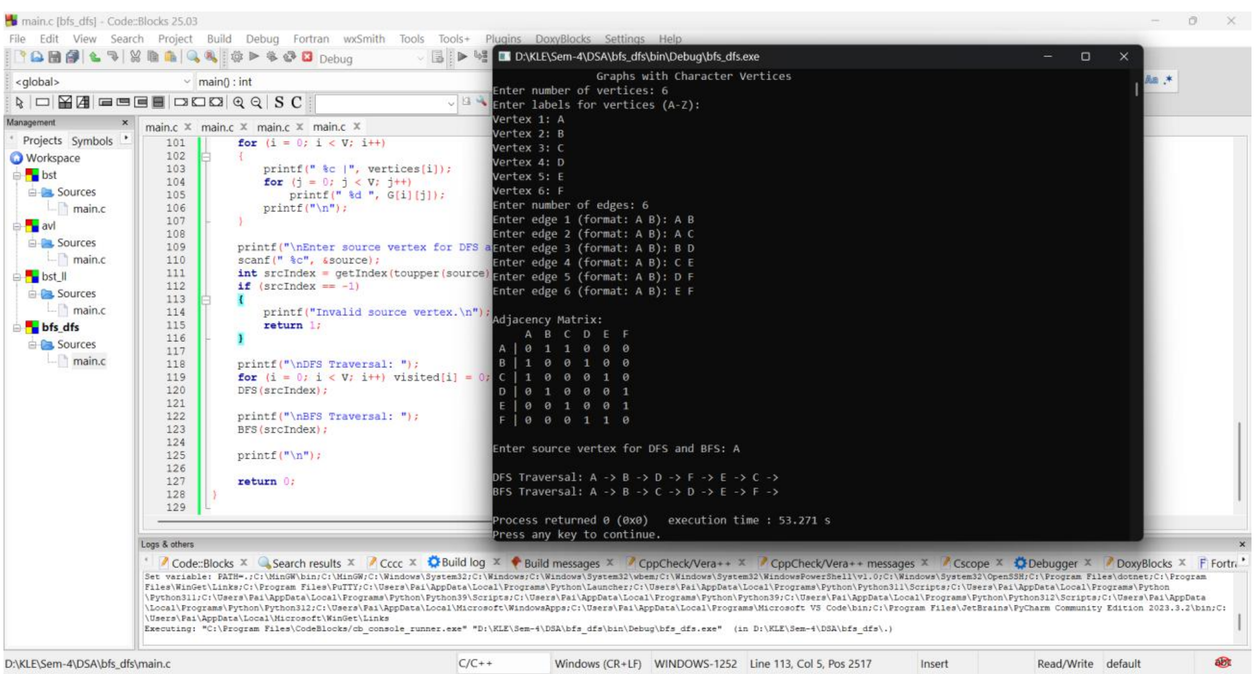
Soln:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define SIZE 20

int V, E, G[SIZE][SIZE], visited[SIZE];
char vertices[SIZE];

int getIndex(char c)
{
   for (int i = 0; i < V; i++)
```

```
    {
        if (vertices[i] == c)
            return i;
    }
    return -1;
}

void DFS(int i)
{
    visited[i] = 1;
    printf("%c -> ", vertices[i]);
    for (int j = 0; j < V; j++)
    {
        if (G[i][j] == 1 && visited[j] == 0)
            DFS(j);
    }
}

void BFS(int start)
{
    int queue[SIZE], front = 0, rear = 0, i, current;
    for (i = 0; i < V; i++)
        visited[i] = 0;

    queue[rear++] = start;
    visited[start] = 1;

    while (front < rear)
    {
        current = queue[front++];
        printf("%c -> ", vertices[current]);
        for (i = 0; i < V; i++)
        {
            if (G[current][i] == 1 && visited[i] == 0)
            {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }
    }
}

int main()
{
    int i, j, idx1, idx2;
    char v1, v2, source;
```

```
printf("\t\tGraphs with Character Vertices\n");
printf("Enter number of vertices: ");
scanf("%d", &V);

printf("Enter labels for vertices (A-Z):\n");
for (i = 0; i < V; i++) {
    printf("Vertex %d: ", i + 1);
    scanf(" %c", &vertices[i]);
    vertices[i] = toupper(vertices[i]);
}
for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
        G[i][j] = 0;

printf("Enter number of edges: ");
scanf("%d", &E);

for (i = 0; i < E; i++)
{
    printf("Enter edge %d (format: A B): ", i + 1);
    scanf(" %c %c", &v1, &v2);
    idx1 = getIndex(toupper(v1));
    idx2 = getIndex(toupper(v2));
    if (idx1 != -1 && idx2 != -1)
    {
        G[idx1][idx2] = 1;
        G[idx2][idx1] = 1;
    }
    else
    {
        printf("Invalid vertex label.\n");
        i--;
    }
}

printf("\nAdjacency Matrix:\n");
printf("    ");
for (i = 0; i < V; i++)
    printf(" %c ", vertices[i]);
printf("\n");
for (i = 0; i < V; i++)
{
    printf(" %c |", vertices[i]);
    for (j = 0; j < V; j++)
        printf(" %d ", G[i][j]);
```

```
        printf("\n");
    }

    printf("\nEnter source vertex for DFS and BFS: ");
    scanf(" %c", &source);
    int srcIndex = getIndex(toupper(source));
    if (srcIndex == -1)
    {
        printf("Invalid source vertex.\n");
        return 1;
    }

    printf("\nDFS Traversal: ");
    for (i = 0; i < V; i++) visited[i] = 0;
    DFS(srcIndex);

    printf("\nBFS Traversal: ");
    BFS(srcIndex);

    printf("\n");

    return 0;
}
```



6. You have a hash table of size 10 (indices 0-9) and the hash function h(k)=kpmod10. Insert the following sequence of keys: [43, 22, 1, 31, 77, 99, 11, 55, 60]. Perform the insertions using the following collision resolution techniques:

a) Chaining (using linked lists for collisions)
b) Linear probing
c) Quadratic probing

For each method, provide the final state of the hash table. In the case of chaining, include the linked lists representing collided elements at each index.

Soln:

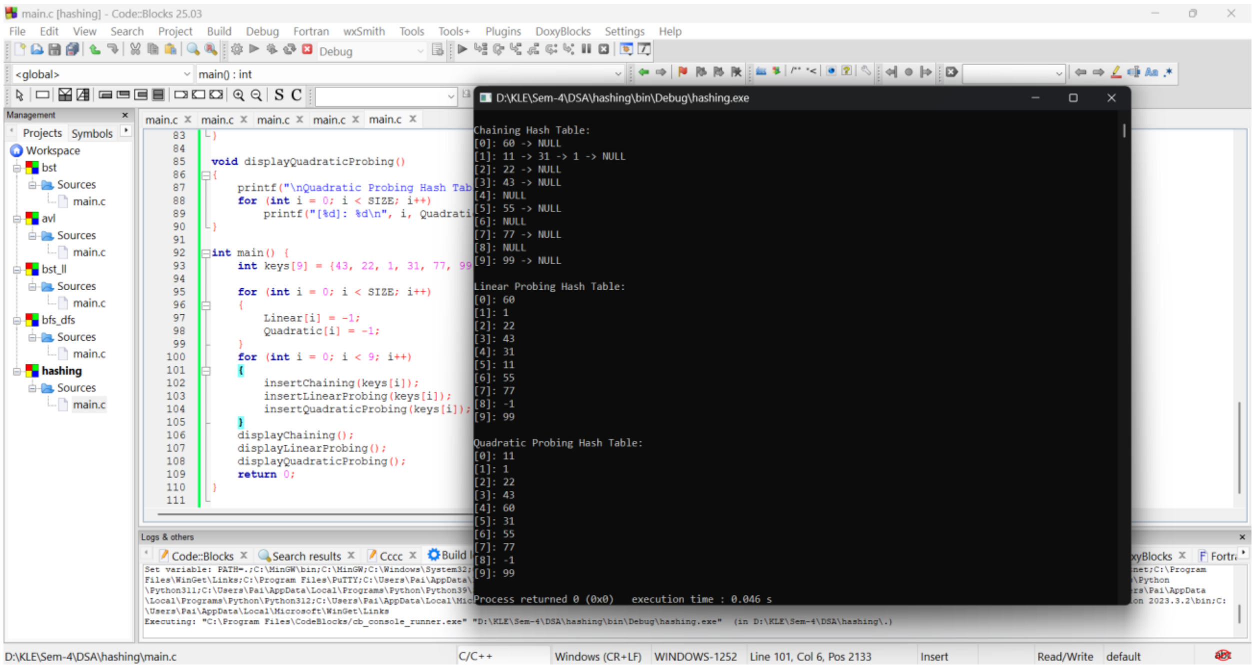
```
#include <stdio.h>

#include <stdlib.h>


#define SIZE 10


typedef struct

{

   int data;

   struct Node* next;

} Node;


Node* hashTableChain[SIZE] = {NULL};

int Linear[SIZE], Quadratic[SIZE];


int hash(int key)

{

   return key % SIZE;

}


void insertChaining(int key)

{
```

```
    int index = hash(key);
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = key;
    newNode->next = hashTableChain[index];
    hashTableChain[index] = newNode;
}

void displayChaining()
{
    Node *temp;
    int i;
    printf("\nChaining Hash Table:\n");
    for (i = 0; i < SIZE; i++)
    {
        printf("[%d]:", i);
        temp = hashTableChain[i];
        while (temp)
        {
            printf(" %d ->", temp->data);
            temp = temp->next;
        }
        printf(" NULL\n");
    }
}

void insertLinearProbing(int key)
{
    int index = hash(key), org = index;
    while (Linear[index] != -1)
```

```
    {
        index = (index + 1) % SIZE;
        if (index == org)
        {
            printf("Hash table full! Could not insert %d in Linear Probing.\n", key);
            return;
        }
    }
    Linear[index] = key;
}

void displayLinearProbing()
{
    printf("\nLinear Probing Hash Table:\n");
    for (int i = 0; i < SIZE; i++)
        printf("[%d]: %d\n", i, Linear[i]);
}

void insertQuadraticProbing(int key)
{
    int index = hash(key), i = 0, newIndex;
    while (i < SIZE)
    {
        newIndex = (index + i*i) % SIZE;
        if (Quadratic[newIndex] == -1)
        {
            Quadratic[newIndex] = key;
            return;
        }
```

```
        i++;
    }
    printf("Hash table full! Could not insert %d in Quadratic Probing.\n", key);
}

void displayQuadraticProbing()
{
    printf("\nQuadratic Probing Hash Table:\n");
    for (int i = 0; i < SIZE; i++)
        printf("[%d]: %d\n", i, Quadratic[i]);
}

int main() {
    int keys[9] = {43, 22, 1, 31, 77, 99, 11, 55, 60};

    for (int i = 0; i < SIZE; i++)
    {
        Linear[i] = -1;
        Quadratic[i] = -1;
    }
    for (int i = 0; i < 9; i++)
    {
        insertChaining(keys[i]);
        insertLinearProbing(keys[i]);
        insertQuadraticProbing(keys[i]);
    }
    displayChaining();
    displayLinearProbing();
    displayQuadraticProbing();
```

```
    return 0;
}
```


main.c [hashing] - Code::Blocks 25.03
D:\KLE\Sem-4\DSA\hashing\bin\Debug\hashing.exe
Chaining Hash Table:
[0]: 60 -> NULL
[1]: 11 -> 31 -> 1 -> NULL
[2]: 22 -> NULL
[3]: 43 -> NULL
[4]: NULL
[5]: 55 -> NULL
[6]: NULL
[7]: 77 -> NULL
[8]: NULL
[9]: 99 -> NULL
Linear Probing Hash Table:
[0]: 60
[1]: 1
[2]: 22
[3]: 43
[4]: 31
[5]: 11
[6]: 55
[7]: 77
[8]: -1
[9]: 99
Quadratic Probing Hash Table:
[0]: 11
[1]: 1
[2]: 22
[3]: 43
[4]: 60
[5]: 31
[6]: 55
[7]: 77
[8]: -1
[9]: 99
Process returned 0 (0x0)   execution time : 0.046 s
D:\KLE\Sem-4\DSA\hashing\main.c