**Task No. 1:** Implement Radix and bucket sort. **Radix Sort:**

class Program

{

static void Main(string[] args)

{

int[] radixArray = { 4725, 4586, 1330, 8792, 1594, 5729 };

radixSort(radixArray, 10, 4);

for (int i = 0; i < radixArray.Length ; i++)

{

Console.WriteLine(radixArray[i]);

}

}

public static void radixSort(int[] input, int radix, int width)

{

for (int i = 0; i < width; i++)

{

radixSingleSort(input, i, radix);

}

}

public static void radixSingleSort(int[] input, int position, int radix) {

int numItems = input.Length;

int[] countArray = new int[radix];

foreach (int value in input) {

countArray[getDigit(position, value, radix)]++;

}

// Adjust the count array

for (int j = 1; j < radix; j++) {

countArray[j] += countArray[j - 1];

}

int[] temp = new int[numItems];

for (int tempIndex = numItems - 1; tempIndex >= 0; tempIndex--) {

temp[--countArray[getDigit(position, input[tempIndex], radix)]] =

input[tempIndex];

}

for (int tempIndex = 0; tempIndex < numItems; tempIndex++) {

input[tempIndex] = temp[tempIndex];

}

}

public static int getDigit(int position, int value, int radix)

{

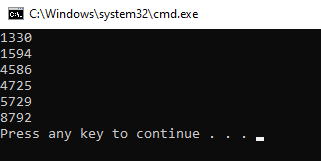
return value / (int)Math.Pow(radix, position) % radix;

}

}

}

**Output:**



**Bucket Sort:**

class Program

{

public static void BucketSort(ref int[] arr)

{

int min = int.MaxValue;

int max = 0;

for (int i = 0; i < arr.Length; i++)

{

if (arr[i] < min)

min = arr[i];

if (arr[i] > max)

max = arr[i];

}

List<int>[] b = new List<int>[max - min + 1];

for (int i = 0; i < b.Length; i++)

{

b[i] = new List<int>();

}

for (int i = 0; i < arr.Length; i++)

{

b[arr[i] - min].Add(arr[i]);

}

int k = 0;

for (int i = 0; i < b.Length; i++)

{

if (b[i].Count > 0)

{

for (int j = 0; j < b[j].Count; j++)

{

arr[k] = b[i][j];

k++;

}

}

}

}

public static void Main()

{

int[] arr = new int[] { 505, 1115, 15, 325, 88, 900, 3, 299 };

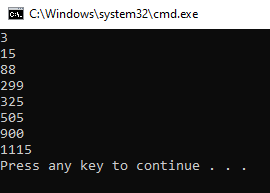
BucketSort(ref arr);

arr.ToList().ForEach(x => Console.WriteLine(x));

}

}

}

**Output:**

**Task No. 2:** Create static tree and perform inorder, preorder and post order traversal. Also search a required node in the tree.

**TreeNode:**

class TreeNode

{

int key;

public TreeNode left, Right;

public TreeNode(int key)

{

this.key = key;

this.left = this.Right = null;

}

public TreeNode getLeft()

{

return this.left;

}

public int GetKey()

{

return this.key;

}

public TreeNode getRight()

{

return this.Right;

}

public void setLeft(int key)

{

this.left = new TreeNode(key); }

public void setRight(int key)

{

this.Right = new TreeNode(key); } }

**BinaryTree:**

class BinaryTree

{

public TreeNode Root;

public BinaryTree()

{

this.Root = null;

}

public BinaryTree(int key)

{

this.Root = new TreeNode(key);

}

public void setRoot(int key)

{

this.Root = new TreeNode(key);

}

public TreeNode getRoot()

{

return this.Root;

}

**InOrder Traversal:**

public void Inorder(TreeNode Node)

{

if (Node == null)

return;

else

{

Inorder(Node.left);

Console.Write(Node.GetKey()+" ");

Inorder(Node.Right);

}

}

**PreOrder Traversal:**

public void Preorder(TreeNode Node)

{

if (Node == null)

return;

else

{

Console.Write(Node.GetKey() + " ");

Preorder(Node.left);

Preorder(Node.Right);

}

}

**PostOrder Traversal:**

public void PostOrder(TreeNode Node)

{

if (Node == null)

return;

else

{

PostOrder(Node.left);

PostOrder(Node.Right);

Console.Write(Node.GetKey() + " "); }

} }

**Function Call:**

BinaryTree tree = new BinaryTree();

tree.setRoot(7);

tree.Root.setLeft(5);

tree.Root.setRight(4);

tree.Root.left.setRight(9);

tree.Root.left.setLeft(20);

tree.Root.Right.setRight(19);

tree.Root.Right.setLeft(26);

System.Console.WriteLine("........InOrder Traversal.......");

tree.Inorder(tree.Root);

System.Console.WriteLine();

System.Console.WriteLine("........PreOrder Traversal.......");

tree.Preorder(tree.Root);

System.Console.WriteLine();

System.Console.WriteLine("........PostOrder Traversal.......");

tree.PostOrder(tree.Root);

**Output:**

