Task 7.1

# Student Name: Peter Stacey

# Student ID: 219011171

Question 1

Draw a series of figures demonstrating the insertion of the values:

20, 9, 3, 7, 5, 8, 25, 30, 15, 6, 17

A picture containing light, computer, traffic, sitting

Description automatically generated

A picture containing computer, sitting, dark, computer

Description automatically generated

A picture containing sitting, dark, light, computer

Description automatically generated

A close up of a logo

Description automatically generated

A picture containing sitting, dark, computer, light

Description automatically generated

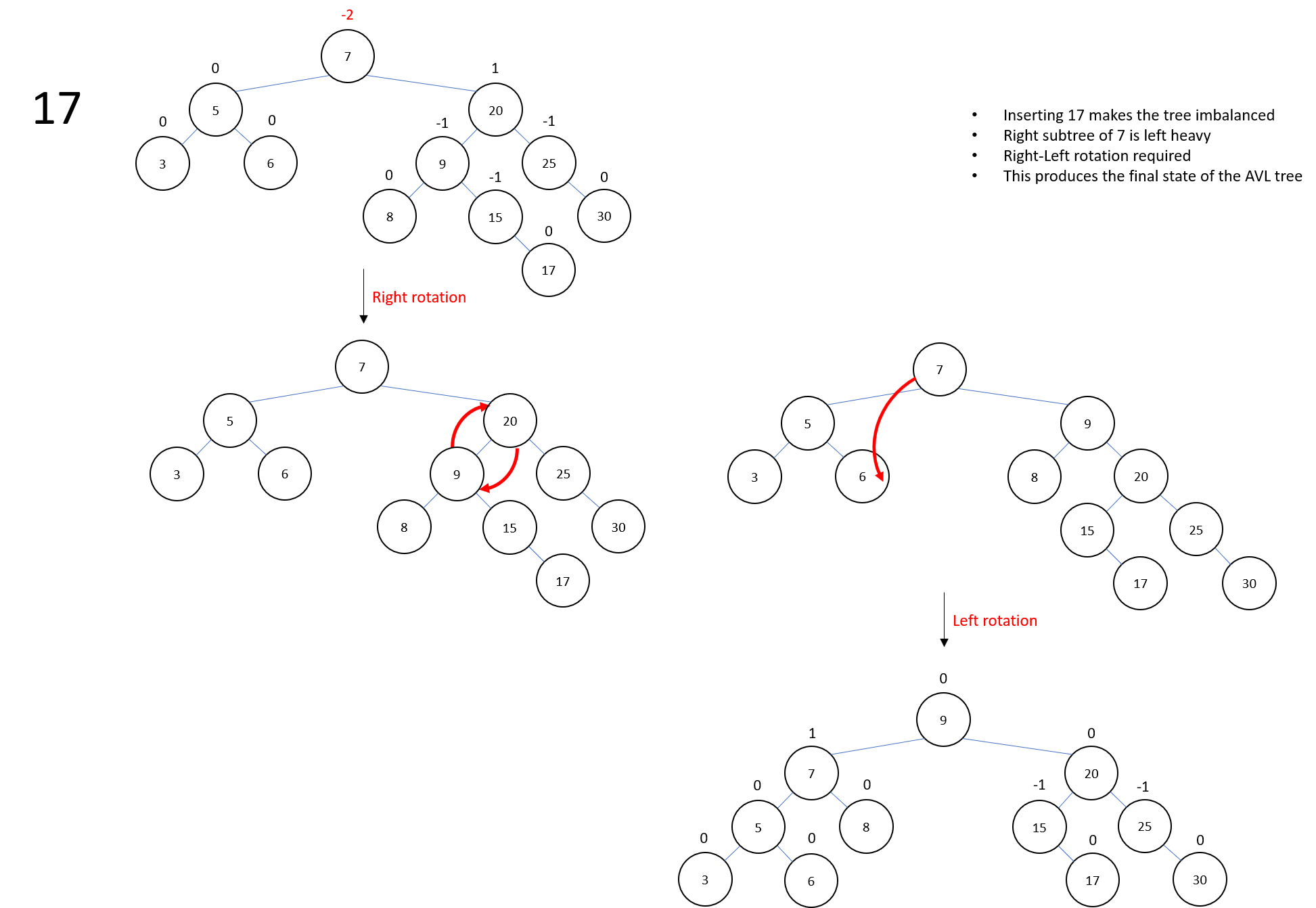
A picture containing dark, sitting, light, computer

Description automatically generated

A picture containing indoor, sitting, computer, dark

Description automatically generated

­



To confirm my results, I wrote an AVL Tree class in C# and tester to print the operations:

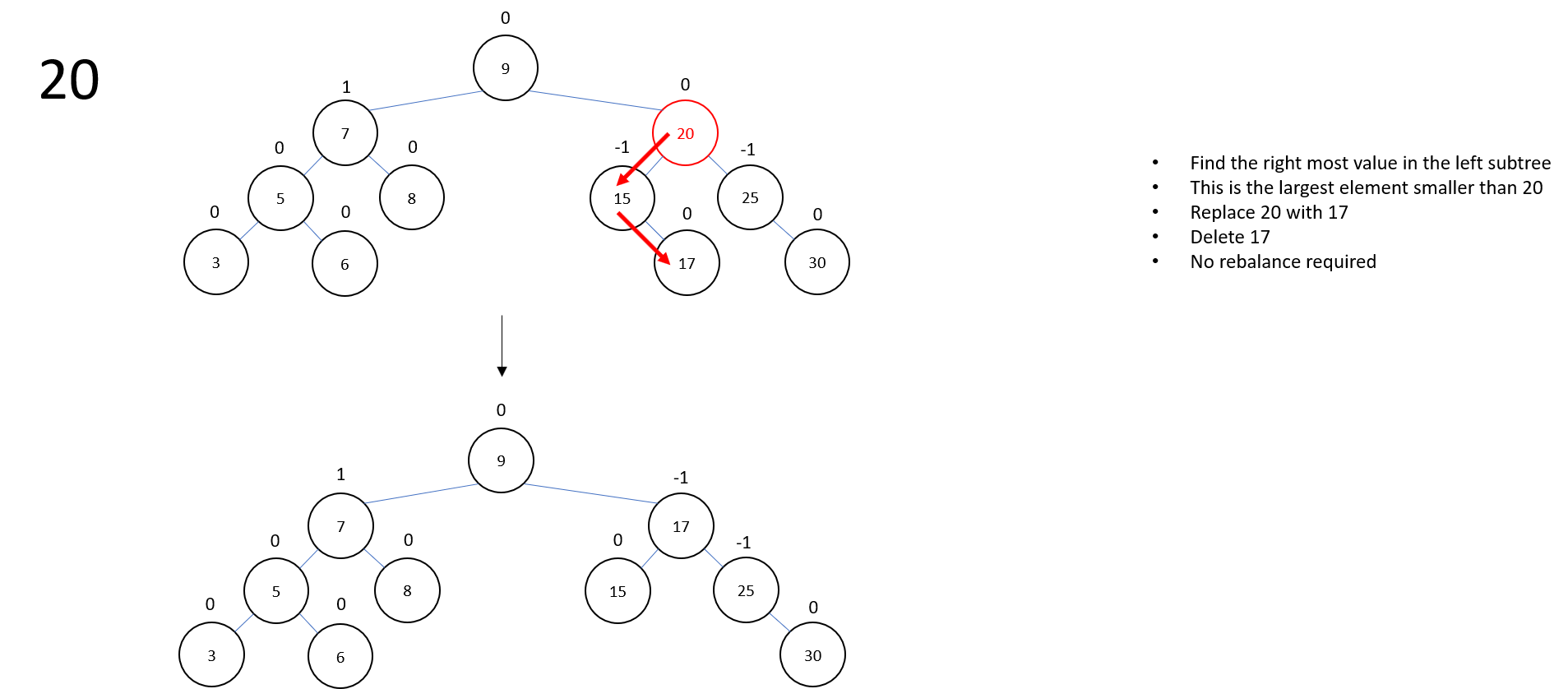
A screenshot of a computer

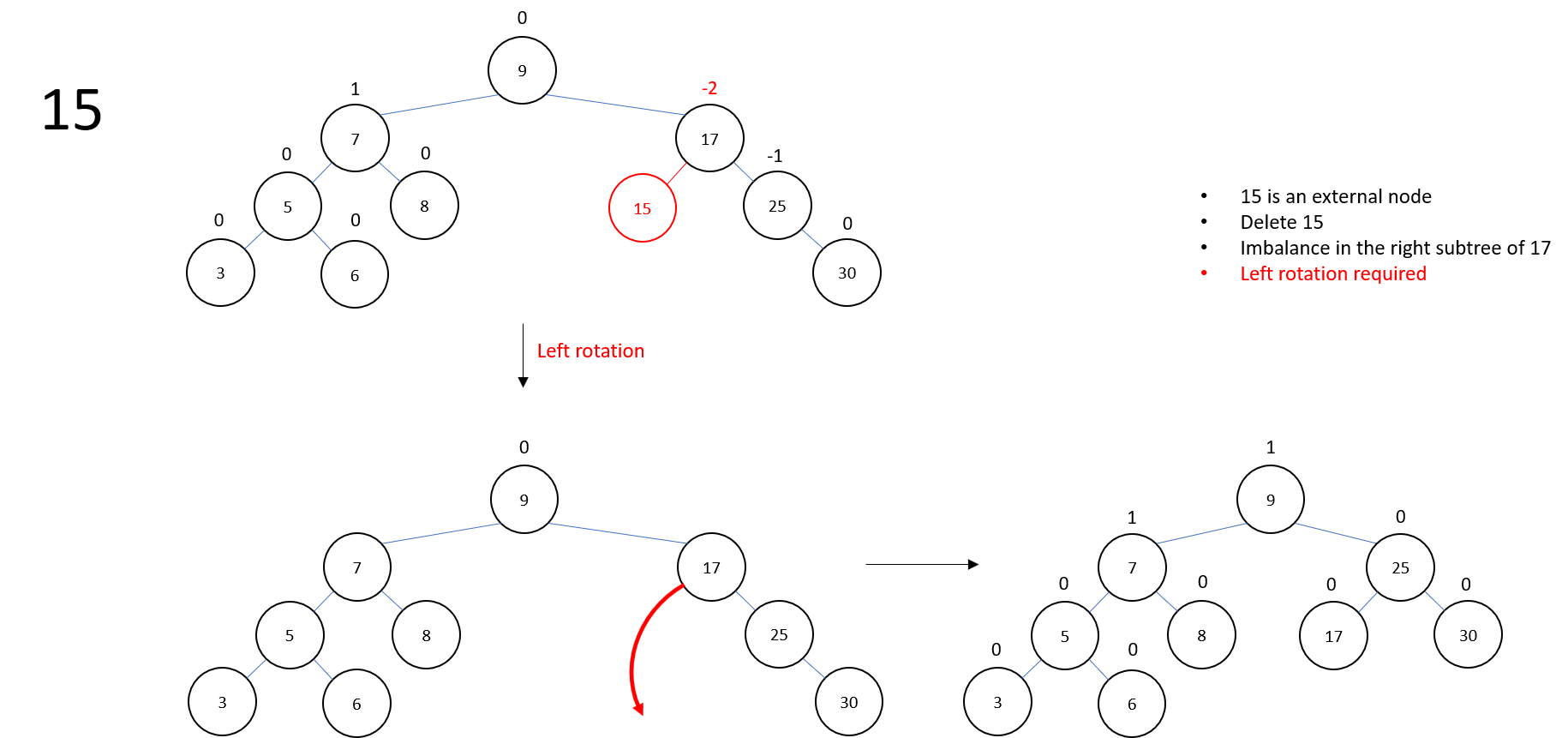
Description automatically generated

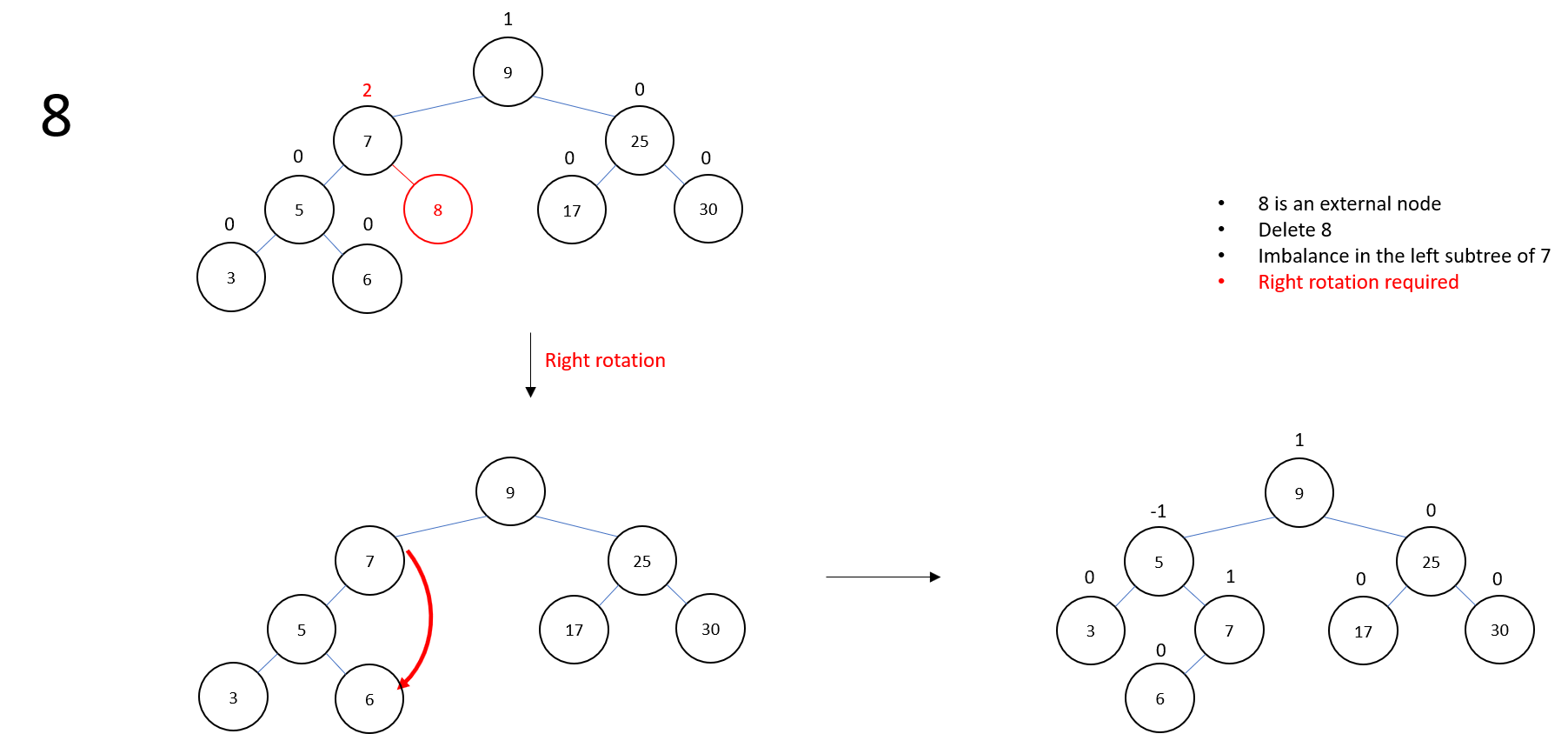
Question 2

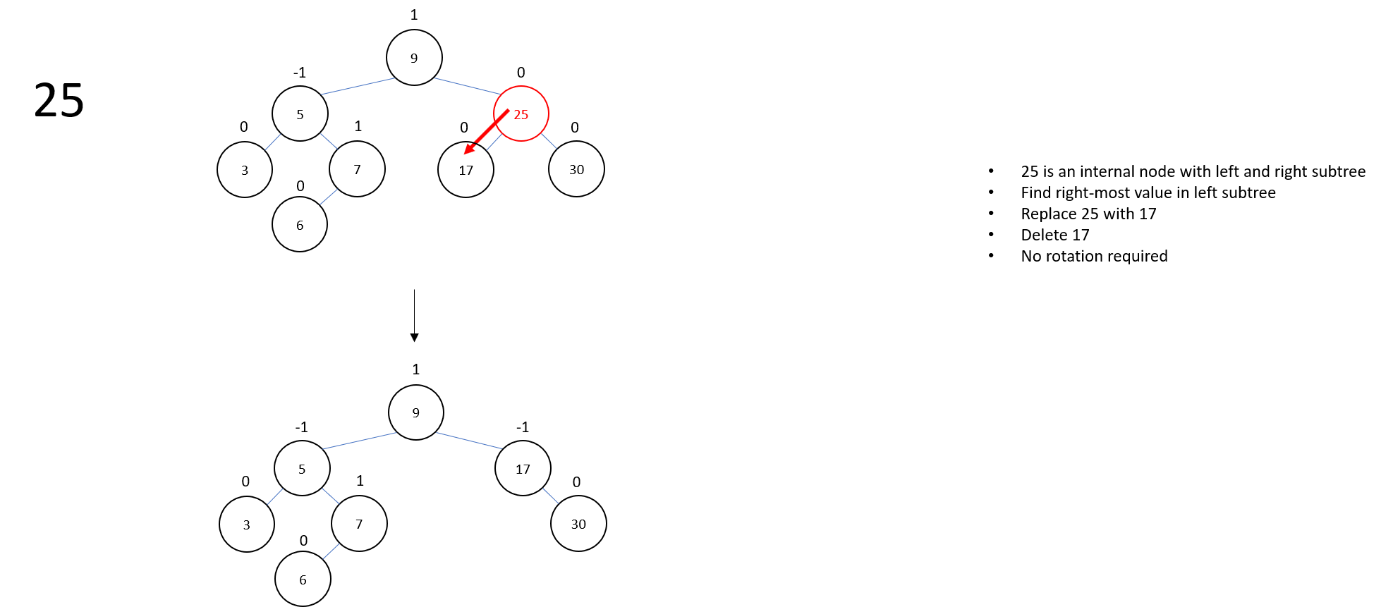
Now, draw a series of figures showing the deletion of the values:

20, 15, 8, 25, 30, 9, 17, 5, 6, 3, 7









A picture containing computer, sitting, computer, dark

Description automatically generated

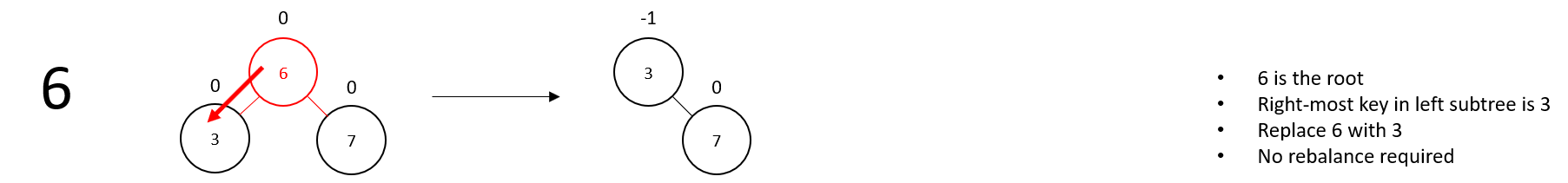
A picture containing computer, computer, sky, star

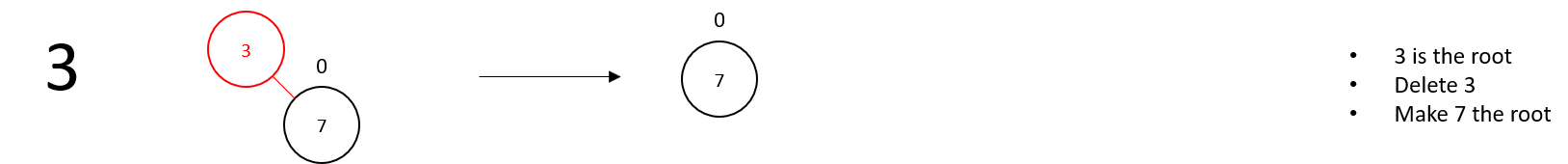
Description automatically generated

A picture containing object, sitting, clock, meter

Description automatically generated

A picture containing light, sitting, dark, computer

Description automatically generated



A close up of a logo

Description automatically generated

Question 3

What order should we insert the elements ሼ1, 2, … ,7ሽ into an empty AVL tree so that we do not have to perform any rotations on it?

Any of the following orders will not require any rotation:

4, 2, 6, 1, 7, 3, 5

4, 6, 2, 1, 7, 3, 5

4, 2, 6, 7, 1, 3, 5

4, 6, 2, 7, 1, 3, 5

4, 2, 6, 1, 7, 5, 3

4, 6, 2, 1, 7, 5, 3

4, 2, 6, 7, 1, 5, 3

4, 6, 2, 7, 1, 5, 3

4, 2, 6, 7, 3, 5, 1

6, 4, 2, 7, 3, 5, 1

4, 2, 6, 7, 3, 1, 5

4, 2, 6, 7, 3, 5, 1

4, 2, 6, 5, 3, 7, 1

4, 2, 6, 5, 3, 1, 7

4, 6, 2, 5, 3, 7, 1

4, 6, 2, 5, 3, 1, 7

3, 2, 5, 1, 6, 4, 7

3, 5, 2, 1, 6, 4, 7

3, 2, 5, 6, 1, 4, 7

3, 5, 2, 6, 1, 4, 7

3, 2, 5, 1, 4, 6, 7

3, 5, 2, 1, 4, 6, 7

3, 2, 5, 4, 1, 6, 7

3, 5, 2, 4, 1, 6, 7

5, 3, 6, 2, 7, 4, 1

5, 6, 3, 2, 7, 4, 1

5, 3, 6, 7, 2, 4, 1

5, 3, 6, 7, 4, 2, 1

5, 3, 6, 4, 7, 2, 1

5, 6, 3, 4, 7, 2, 1

5, 3, 6, 2, 7, 1, 4

5, 6, 3, 2, 7, 1, 4

**Attachment 1**

AVLTree.cs

<https://github.com/pscompsci/SIT221_Library/blob/master/Task_7_1/AVLTree.cs>

|  |
| --- |
|  |
|  | using System;  using System.Collections.Generic; |
|  | using System.Diagnostics.Contracts; |
|  |  |
|  | #pragma warning disable 693 |
|  |  |
|  | namespace Task\_7\_1 |
|  | { |
|  | public enum DisplayMethod |
|  | { |
|  | InOrder, |
|  | PreOrder, |
|  | PostOrder, |
|  | BreadthFirst |
|  | } |
|  |  |
|  | public class AVLTree<T> where T : IComparable<T> |
|  | { |
|  | public class Node<T> : INode<T> where T : IComparable<T> |
|  | { |
|  | public T Key { get; set; } |
|  | public Node<T> Left { get; set; } |
|  | public Node<T> Right { get; set; } |
|  | public Node(T key) |
|  | { |
|  | Key = key; |
|  | } |
|  | } |
|  |  |
|  | public Node<T> Root { get; set; } |
|  | public int Count { get; set; } |
|  | #if DEBUG |
|  | private List<KeyValuePair<string, T>> \_rotations; |
|  | #endif |
|  | public AVLTree() |
|  | { |
|  | #if DEBUG |
|  | \_rotations = new List<KeyValuePair<string, T>>(); |
|  | #endif |
|  | } |
|  |  |
|  | public void Insert(T value) |
|  | { |
|  | #if DEBUG |
|  | \_rotations.Clear(); |
|  | Console.ForegroundColor = ConsoleColor.Red; |
|  | Console.WriteLine("Inserting {0}", value); |
|  | Console.ResetColor(); |
|  | #endif |
|  | Node<T> node = new Node<T>(value); |
|  | if (Root is null) |
|  | { |
|  | Root = node; |
|  | } |
|  | else |
|  | { |
|  | Root = RecursiveInsert(Root, node); |
|  | } |
|  | #if DEBUG |
|  | Console.WriteLine("Inserted"); |
|  | if (\_rotations.Count > 0) |
|  | { |
|  | foreach(var pair in \_rotations) |
|  | { |
|  | Console.WriteLine("Rotated {0} on {1}", pair.Key, pair.Value); |
|  | } |
|  | Console.WriteLine(); |
|  | } |
|  | Console.Write("{0,-30}", "Current BreadthFirst Order:"); |
|  | Display(DisplayMethod.BreadthFirst); |
|  | #endif |
|  | Count++; |
|  | } |
|  |  |
|  | private Node<T> RecursiveInsert(Node<T> current, Node<T> node) |
|  | { |
|  | if (current is null) |
|  | { |
|  | current = node; |
|  | return current; |
|  | } |
|  | #if DEBUG |
|  | Console.WriteLine("Current Node: {0}", current.Key); |
|  | #endif |
|  | if (node.Key.CompareTo(current.Key) < 0) |
|  | { |
|  | current.Left = RecursiveInsert(current.Left, node); |
|  | current = BalanceTree(current); |
|  | } |
|  | else if (node.Key.CompareTo(current.Key) > 0) |
|  | { |
|  | current.Right = RecursiveInsert(current.Right, node); |
|  | current = BalanceTree(current); |
|  | } |
|  | return current; |
|  | } |
|  | private Node<T> BalanceTree(Node<T> node) |
|  | { |
|  | int balanceFactor = BalanceFactor(node); |
|  | if (balanceFactor > 1) |
|  | { |
|  | if (BalanceFactor(node.Left) > 0) |
|  | { |
|  | node = RotateLeft(node); |
|  | } |
|  | else |
|  | { |
|  | node = RotateLeftRight(node); |
|  | } |
|  | } |
|  | else if (balanceFactor < -1) |
|  | { |
|  | if (BalanceFactor(node.Right) > 0) |
|  | { |
|  | node = RotateRightLeft(node); |
|  | } |
|  | else |
|  | { |
|  | node = RotateRight(node); |
|  | } |
|  | } |
|  | return node; |
|  | } |
|  | public void Remove(T value) |
|  | { |
|  | #if DEBUG |
|  | Console.ForegroundColor = ConsoleColor.Green; |
|  | Console.WriteLine("Removing {0}", value); |
|  | Console.ResetColor(); |
|  | #endif |
|  | Root = Remove(Root, value); |
|  | #if DEBUG |
|  | Console.WriteLine("Removed"); |
|  | Console.Write("{0,-30}", "Current BreadthFirst Order:"); |
|  | Display(DisplayMethod.BreadthFirst); |
|  | #endif |
|  | } |
|  |  |
|  | private Node<T> Remove(Node<T> node, T value) |
|  | { |
|  | Node<T> parent; |
|  | if (node is null) return null; |
|  | else |
|  | { |
|  | if (value.CompareTo(node.Key) < 0) |
|  | { |
|  | node.Left = Remove(node.Left, value); |
|  | if (BalanceFactor(node) == -2) |
|  | { |
|  | if (BalanceFactor(node.Right) <= 0) |
|  | { |
|  | node = RotateRight(node); |
|  | } |
|  | else |
|  | { |
|  | node = RotateRightLeft(node); |
|  | } |
|  | } |
|  | } |
|  | else if (value.CompareTo(node.Key) > 0) |
|  | { |
|  | node.Right = Remove(node.Right, value); |
|  | if (BalanceFactor(node) == 2) |
|  | { |
|  | if (BalanceFactor(node.Left) >= 0) |
|  | { |
|  | node = RotateLeft(node); |
|  | } |
|  | else |
|  | { |
|  | node = RotateLeftRight(node); |
|  | } |
|  | } |
|  | } |
|  | else |
|  | { |
|  | if (node.Right != null) |
|  | { |
|  | parent = node.Right; |
|  | while (parent.Left != null) |
|  | { |
|  | parent = parent.Left; |
|  | } |
|  | node.Key = parent.Key; |
|  | node.Right = Remove(node.Right, parent.Key); |
|  | if (BalanceFactor(node) == 2) |
|  | { |
|  | if (BalanceFactor(node.Left) >= 0) |
|  | { |
|  | node = RotateLeft(node); |
|  | } |
|  | else |
|  | { |
|  | node = RotateLeftRight(node); |
|  | } |
|  | } |
|  | } |
|  | else |
|  | { |
|  | return node.Left; |
|  | } |
|  | } |
|  | } |
|  | return node; |
|  | } |
|  |  |
|  | public Node<T> Find(T value) |
|  | { |
|  | return Find(Root, value); |
|  | } |
|  |  |
|  | private Node<T> Find(Node<T> node, T value) |
|  | { |
|  |  |
|  | if (value.CompareTo(node.Key) < 0) |
|  | { |
|  | if (value.Equals(node.Key)) |
|  | { |
|  | return node; |
|  | } |
|  | else |
|  | return Find(node.Left, value); |
|  | } |
|  | else |
|  | { |
|  | if (value.Equals(node.Key)) |
|  | { |
|  | return node; |
|  | } |
|  | else |
|  | return Find(node.Right, value); |
|  | } |
|  | } |
|  |  |
|  | private int Max(int l, int r) |
|  | { |
|  | return l > r ? l : r; |
|  | } |
|  |  |
|  | private int GetHeight(Node<T> node) |
|  | { |
|  | int height = 0; |
|  | if (node != null) |
|  | { |
|  | int l = GetHeight(node.Left); |
|  | int r = GetHeight(node.Right); |
|  | int m = Max(l, r); |
|  | height = m + 1; |
|  | } |
|  | return height; |
|  | } |
|  | private int BalanceFactor(Node<T> node) |
|  | { |
|  | int l = GetHeight(node.Left); |
|  | int r = GetHeight(node.Right); |
|  | return l - r; |
|  | } |
|  |  |
|  | private Node<T> RotateRight(Node<T> parent) |
|  | { |
|  | #if DEBUG |
|  | \_rotations.Add(new KeyValuePair<string, T>("Left", parent.Key)); |
|  | #endif |
|  | Node<T> pivot = parent.Right; |
|  | parent.Right = pivot.Left; |
|  | pivot.Left = parent; |
|  | return pivot; |
|  | } |
|  |  |
|  | private Node<T> RotateLeft(Node<T> parent) |
|  | { |
|  | #if DEBUG |
|  | \_rotations.Add(new KeyValuePair<string, T>("Right", parent.Key)); |
|  | #endif |
|  | Node<T> pivot = parent.Left; |
|  | parent.Left = pivot.Right; |
|  | pivot.Right = parent; |
|  | return pivot; |
|  | } |
|  |  |
|  | private Node<T> RotateLeftRight(Node<T> parent) |
|  | { |
|  | Node<T> pivot = parent.Left; |
|  | parent.Left = RotateRight(pivot); |
|  | return RotateLeft(parent); |
|  | } |
|  |  |
|  | private Node<T> RotateRightLeft(Node<T> parent) |
|  | { |
|  | Node<T> pivot = parent.Right; |
|  | parent.Right = RotateLeft(pivot); |
|  | return RotateRight(parent); |
|  | } |
|  |  |
|  | public List<T> InOrder() |
|  | { |
|  | List<T> result = new List<T>(); |
|  | InOrder(Root); |
|  |  |
|  | return result; |
|  |  |
|  | void InOrder(Node<T> node) |
|  | { |
|  | if (node.Left != null) |
|  | { |
|  | InOrder(node.Left); |
|  | } |
|  | result.Add(node.Key); |
|  | if (node.Right != null) |
|  | { |
|  | InOrder(node.Right); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | public List<T> PostOrder() |
|  | { |
|  | List<T> result = new List<T>(); |
|  | PostOrder(Root); |
|  |  |
|  | return result; |
|  |  |
|  | void PostOrder(Node<T> node) |
|  | { |
|  | if (node.Left != null) |
|  | { |
|  | PostOrder(node.Left); |
|  | } |
|  | if (node.Right != null) |
|  | { |
|  | PostOrder(node.Right); |
|  | } |
|  | result.Add(node.Key); |
|  | } |
|  | } |
|  |  |
|  | public List<T> PreOrder() |
|  | { |
|  | List<T> result = new List<T>(); |
|  | PreOrder(Root); |
|  |  |
|  | return result; |
|  |  |
|  | void PreOrder(Node<T> node) |
|  | { |
|  | result.Add(node.Key); |
|  | if (node.Left != null) |
|  | { |
|  | PreOrder(node.Left); |
|  | } |
|  | if (node.Right != null) |
|  | { |
|  | PreOrder(node.Right); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | public List<T> BreadthFirst() |
|  | { |
|  | if (Root is null) return null; |
|  | List<T> result = new List<T>(); |
|  | Queue<Node<T>> nodes = new Queue<Node<T>>(); |
|  | BreadthFirst(Root); |
|  | return result; |
|  |  |
|  | void BreadthFirst(Node<T> node) |
|  | { |
|  | result.Add(node.Key); |
|  | if (node.Left != null) |
|  | { |
|  | nodes.Enqueue(node.Left); |
|  | } |
|  | if (node.Right != null) |
|  | { |
|  | nodes.Enqueue(node.Right); |
|  | } |
|  | if (nodes.Count > 0) |
|  | { |
|  | BreadthFirst(nodes.Dequeue()); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | public void Display(DisplayMethod method) |
|  | { |
|  | List<T> result; |
|  | switch (method) |
|  | { |
|  | case DisplayMethod.InOrder: |
|  | result = InOrder(); |
|  | break; |
|  | case DisplayMethod.PreOrder: |
|  | result = PreOrder(); |
|  | break; |
|  | case DisplayMethod.PostOrder: |
|  | result = PostOrder(); |
|  | break; |
|  | case DisplayMethod.BreadthFirst: |
|  | default: |
|  | result = BreadthFirst(); |
|  | break; |
|  |  |
|  | } |
|  | if (result is null) |
|  | { |
|  | Console.WriteLine("[ ]"); |
|  | return; |
|  | } |
|  | Console.Write("[ "); |
|  | foreach (var value in result) |
|  | { |
|  | Node<T> node = Find(value); |
|  | int balanceFactor = BalanceFactor(node); |
|  | Console.Write(value + "(" + balanceFactor + ") "); |
|  | } |
|  | Console.WriteLine("]\n"); |
|  | } |
|  | } |
|  | } |

**Attachment 2**

Tester.cs

<https://github.com/pscompsci/SIT221_Library/blob/master/Task_7_1/Tester.cs>

|  |
| --- |
|  |
|  | Using System;  using System.Collections.Generic; |
|  | using System.Globalization; |
|  |  |
|  | namespace Task\_7\_1 |
|  | { |
|  | class Tester |
|  | { |
|  |  |
|  | static void Display(List<int> list) |
|  | { |
|  | Console.Write("["); |
|  | foreach(int i in list) |
|  | { |
|  | Console.Write(i + " "); |
|  | } |
|  | Console.WriteLine("]"); |
|  | } |
|  |  |
|  | static void Main(string[] args) |
|  | { |
|  | AVLTree<int> tree = new AVLTree<int>(); |
|  |  |
|  | tree.Insert(20); |
|  | tree.Insert(9); |
|  | tree.Insert(3); |
|  | tree.Insert(7); |
|  | tree.Insert(5); |
|  | tree.Insert(8); |
|  | tree.Insert(25); |
|  | tree.Insert(30); |
|  | tree.Insert(15); |
|  | tree.Insert(6); |
|  | tree.Insert(17); |
|  |  |
|  | List<int> breadth = tree.BreadthFirst(); |
|  | Console.Write("{0,-20}", "Breadth First:"); |
|  | Display(breadth); |
|  |  |
|  | List<int> preorder = tree.PreOrder(); |
|  | Console.Write("{0,-20}", "Pre-Order First:"); |
|  | Display(preorder); |
|  |  |
|  | List<int> postorder = tree.PostOrder(); |
|  | Console.Write("{0,-20}", "Post-Order First:"); |
|  | Display(postorder); |
|  |  |
|  | List<int> inorder = tree.InOrder(); |
|  | Console.Write("{0,-20}", "In-Order First:"); |
|  | Display(inorder); |
|  |  |
|  | Console.WriteLine("\n\n"); |
|  |  |
|  | tree.Remove(20); |
|  | tree.Remove(15); |
|  | tree.Remove(8); |
|  | tree.Remove(25); |
|  | tree.Remove(30); |
|  | tree.Remove(9); |
|  | tree.Remove(17); |
|  | tree.Remove(5); |
|  | tree.Remove(6); |
|  | tree.Remove(3); |
|  | tree.Remove(7); |
|  |  |
|  | Console.ReadKey(); |
|  | } |
|  | } |
|  | } |