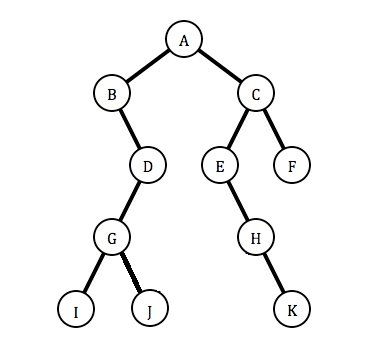
# Instructions

* Work alone on this assignment.
* Complete Part A in a Word document (can do on paper and scan images if you want), then export it in pdf format for submission.
* Complete Part B in a copy of your code from class.
* Put your pdf document in the root of your code project. Submit to the folder on Dropbox.
* Due date: Oct. 10, 2022

# Part A Written [20 marks]

1. Consider the following binary tree. [3 marks]
   1. Write the preorder, inorder and postorder traversals of the binary tree shown below.  
      

* INORDER**: B, I, G, J, D, A, E, H, K, C, F**
* PREORDER**: A, B, D, G, I, J, C, E, H, K, F**
* POSTORDER**: I, J, G, D, B, K, H, E, F, C, A**

1. A binary tree has a preorder traversal of DIJNEBGOKMLHAFO and an inorder traversal of IENJGBODMKHFAOL. What is its postorder traversal? [3 marks]

* **PREORDER:** DIJNEBGOKMLHAFO
* **INORDER:** IENJGBODMKHFAOL

According to preorder we can identify that root is D.

According to inorder we can identify that left subtree is “IENJGBO” and right subtree is “MKHFAOL”.

Here, root of left subtree is “I”, according to preorder.

Similarly, root of right subtree is “K”.

And by doing this recursively…

* **POSTORDER:** ENGOBJIMFOAHLKD

1. If a BST stores 1 billion nodes (show any formulas used), [4 marks]
   1. **What are the minimum and maximum numbers of levels**?

* Minimum number of levels: -
  + **floor (log2n)**
  + **floor (29.9)**
  + **29**
* Maximum number of levels: -
  + If there are n nodes in a binary search tree, maximum height of the binary search tree is n-1.
  + So, answer is billion – 1.
  + **999,999,999**
  1. **If the tree were an AVL tree, what would be the approximate maximum number of levels**?
* the minimum height is Log2(n) and the maximum height is 1.44\* Log2(n)
  + minimum height.
  + Log2(n)
  + Log2(1000000000)
  + **29.9**
* The maximum height is 1.44\* Log2(n)
  + 1.44 \* 29.9
  + **43.056**

1. For the following tree, show the resultant tree assuming the Remove method from class was used. For each question below, assume you are removing from the original tree. Show all intermediate steps and explain what is going on in each step. [6 marks]
   1. Remove node 76 from the original tree.

* Step1: Here, 76 > 50, so we will check right subtree of the tree.
* Step2: Now, 76 do not have any right subtree.
* Step3: Join parent node of deleted node to the left node of deleted node.

Diagram

Description automatically generated

* 1. Remove node 23 from the original tree.
* 23 < 50, so checking at left subtree.
* 23 > 17, so checking at right subtree.
* Here, 23 has no right child.
* So, join parent of deleted node with left child of deleted node.

Diagram

Description automatically generated

* 1. Remove node 50 from the original tree.
* Node has two child. So, we will find the largest element in left subtree.
* Copy that value into node which we have to delete.
* Remove the node whose value we copied.

Diagram

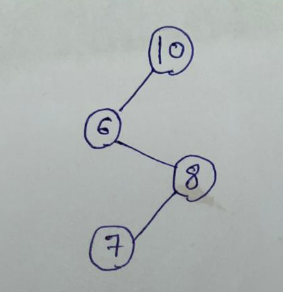
Description automatically generated

A picture containing clipart

Description automatically generated

1. Given the following AVLT tree that balances when adding, but NOT on removing, show the rotations that would take place after removing the 20 and then adding 7 to the tree. Show the intermediate steps as well as the final balanced tree. [4 marks]

|  |  |
| --- | --- |
| Diagram  Description automatically generated |  |

Step 1 is removing 20 without balancing and then adding 7.  


Here, we can see RL imbalance. So we will perform RL single right, single left rotation.

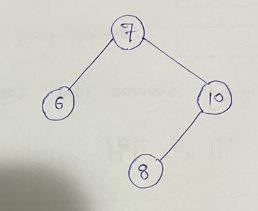
A drawing of a question mark

Description automatically generated with low confidence

Diagram

Description automatically generated

Now, we can see our root is left heavy and left subtree is balanced. So, we will perform single right rotation.



# Part B Code [20 marks]

1. Implement the following two methods of the LinkedList done in the class: [ 4 marks = 2 + 2 marks]

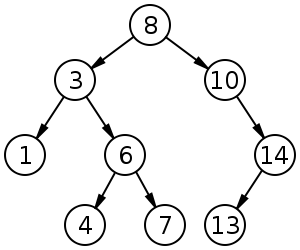
* code is also submitted in separate file. However, here is the main methods.
  1. public override T RemoveAt(int index)

|  |
| --- |
| int ind = 1;  Node prev = null;  public T RecRemoveAt(int index)  {  T olddata ;  prev = this.head;  if (index < 0 || index >= this.Count)  {  throw new IndexOutOfRangeException("Invalid index " + index);  }  else if (this.head == null)  {  throw new Exception("linked list is empty");  }  else if (index == 0)  {  olddata = this.head.data;  this.head = this.head.next;  return olddata;    }  else  {  olddata = RecRemoveHelp(this.head.next , index);  }  return olddata;      }    private T RecRemoveHelp(Node newNode, int index)  {  Console.WriteLine(ind +"==>> "+newNode.data);  if(ind == index)  {    prev.next = newNode.next;  Console.WriteLine(ind + "===>1> " + newNode.data);    return newNode.data;  }  else  {  ind++;  prev = newNode;  Console.WriteLine(ind + "===>1> " + newNode);  return RecRemoveHelp(newNode.next, index);  }  } |

* 1. public override T ReplaceAt(int index, T data)

|  |
| --- |
| int myIndex = 1;  public T RecReplaceAt(int index, T data)  {  Console.WriteLine(this.Count);  Console.WriteLine(this.Count);  T oldData;  if (index < 0 || index >= this.Count)  {  throw new IndexOutOfRangeException("Invalid index ===>" + index);  }  else if (index == 0)  {  oldData = this.head.data;  this.head.data = data;  return oldData;  }  else  {  oldData = RecReplaceHelp(index,data, this.head.next);  }  return oldData;  }  T RecReplaceHelp(int index, T data, Node nNode)  {  T oldData;  if (index == myIndex)  {  oldData = nNode.data;  nNode.data = data;  }  else  {  myIndex++;  oldData=RecReplaceHelp(index,data, nNode.next);  }  return oldData;  } |

1. Write code for a method of your BST class that will create a sorted (descending order) linked list from the provided tree. The linked list from the project built in class will be used. If the root is null, an empty list will be returned.  
   ie. Given the BST: [4 marks]

[](http://upload.wikimedia.org/wikipedia/commons/d/da/Binary_search_tree.svg)

The Linked List would beDiagram

Description automatically generated

Things to do:

* Create the method as  
   public LinkedList.LinkedList<T> ToLinkedList()
* code is also submitted in separate file. However, here is the main methods.

|  |
| --- |
| public LinkedList.LinkedList<T> ToLinkedList()  {  LinkedList.LinkedList<T> list = new LinkedList.LinkedList<T>();  void dosomething(T data)  {  list.Add(data);  }  this.Iterate(dosomething, TRAVERSALORDER.IN\_ORDER);  int start = 0;  int end = list.Count - 1;  while (start < end)  {  T temp = list.ElementAt(start);  list.ReplaceAt(start, list.ElementAt(end));  list.ReplaceAt(end, temp);  start++;  end--;  }  return list;  } |

* Using the linked list
  + You will need to convert your linked list project from a console application to a class library. This can be done in the project properties.
  + Remove the program.cs file from LinkedList (can’t have a main method in a class library project).
  + Build the linked list project.
  + Add a reference in the Binary Tree project to the Linked list dll.
  + You will need to fully qualify the name of your linked list when you use it. For example, look at the return type of the method mentioned above. This will distinguish our linked list from the built in one.

1. The approach used to code the BST was to use recursion to implement Add. Override the Add method in the AVLT class by using an iterative approach. Note that you need to balance the tree in the method [6 marks, 3 marks for iterative Add and 3 marks for balancing the tree]

* code is also submitted in separate file. However, here is the main methods.

|  |
| --- |
| public void newAdd(T data)  {  if (nRoot == null)  {  nRoot = new Node<T>(data);  }  else  {  IterativeAdd2(data);    //nRoot = Balance(nRoot);  }  iCount++;  }    private void IterativeAdd2( T data)  {  Stack<Node<T>> stack1 = new Stack<Node<T>>();  Stack<Node<T>> stack2 = new Stack<Node<T>>();  Node<T> root = nRoot;  Node<T> newnode = new Node<T>(data);  while (root!= null)  {  stack1.Push(root);  if (data.CompareTo(root.Data) < 0)  {  if(root.Left == null)  {  root.Left = newnode;  stack1.Push(newnode);  break;  }  else  {  root = root.Left;  }  }  else  {  if(root.Right == null)  {  root.Right = newnode;  stack1.Push(newnode);  break;  }  else  {  root = root.Right;  }  }    }  while (stack1.Count > 0)  {  Node<T> node2 = stack1.Pop();    //nRoot = Balance(node2);  if (stack1.Count == 1)  {  if (nRoot.Left != null)  {  Console.WriteLine("-=before==peekleft=->>>" + stack1.Peek().Right.Data);  //nRoot.Right = node2;    }  }    }  } |

1. The approach used to code the BST done in class was to use recursion to implement Iterate. Override the Iterate method in the AVLT class so that it uses an iterative approach. [6 marks]

* code is also submitted in separate file. However, here is the main methods.

|  |
| --- |
| * public override void Iterate(ProcessData<T> pd, TRAVERSALORDER to) * { * if (to == TRAVERSALORDER.PRE\_ORDER) * { * preorderIterative(pd, nRoot); * } * else if (to == TRAVERSALORDER.IN\_ORDER) * { * inorderIterative(pd, nRoot); * } * else if (to == TRAVERSALORDER.POST\_ORDER) * { * PostorderIterative(pd, nRoot); * } * } * public void PostorderIterative(ProcessData<T> pd, Node<T> root) * { * Stack<Node<T>> FirstStack = new Stack<Node<T>>(); * Stack<Node<T>> secondStack = new Stack<Node<T>>(); * if (root == null) * return; * FirstStack.Push(root); * while (FirstStack.Count > 0) * { * Node<T> myNode = FirstStack.Pop(); * secondStack.Push(myNode); * if (myNode.Left != null) * FirstStack.Push(myNode.Left); * if (myNode.Right != null) * FirstStack.Push(myNode.Right); * } * while (secondStack.Count > 0) * { * Node<T> temp = secondStack.Pop(); * pd(temp.Data); * } * } * public void preorderIterative(ProcessData<T> pd, Node<T> node) * { * if (node == null) * { * return; * } * Stack<Node<T>> tempstack = new Stack<Node<T>>(); * tempstack.Push(node); * while (tempstack.Count > 0) * { * Node<T> mynode = tempstack.Peek(); * pd(mynode.Data); * tempstack.Pop(); * if (mynode.Right != null) * { * tempstack.Push(mynode.Right); * } * if (mynode.Left != null) * { * tempstack.Push(mynode.Left); * } * } * } * public void inorderIterative(ProcessData<T> pd, Node<T> root) * { * Stack<Node<T>> JayStack = new Stack<Node<T>>(); * Node<T> MyNode = root; * while (JayStack.Count > 0 || MyNode != null) * { * if (MyNode != null) * { * JayStack.Push(MyNode); * MyNode = MyNode.Left; * } * else * { * MyNode = JayStack.Pop(); * pd(MyNode.Data); * MyNode = MyNode.Right; * } * } * } |