2025-09-11

# The trees collections

## Introduction

A [binary tree](https://en.wikipedia.org/wiki/Binary_tree) is a precise mathematical concept that can be defined as a restriction on graphs. As an abstract data type, it generally uses the following definitions:

* A binary tree is made of *nodes*, each of which contain *one value*, can have up to 2 *children*.
* A (rooted) binary tree has exactly one node with 0 parent (that is not the child of any other node), called *the root*. Except for the root, all the nodes have exactly one parent.
* A *leaf* is a node *without children*.
* The *depth of a node* is the distance (i.e., the number of times we must go to its parent) from it to the root.
* The *depth of a tree* is the greatest distance of its nodes.
* A *subtree* is the tree obtain by considering a particular node in a tree as the root of the tree made of its children.

From there, operations generally include, as usual

* Creating an empty tree,
* Adding a node to a tree,
* Finding the smalles value in the tree,
* Removing a node from the tree,

## Possible Implementation

### Binary Tree

We implement a binary tree class abstractly, because two methods will be missing: how to insert a value, and how to delete a value. We also mark the Find method as virtual because we will override it for something more efficient. Note, finally, that we use the protected keyword, so that classes inheriting the BTree class will be able to manipulate Nodes.

using System;
using System.Collections.Generic;

public abstract class BTree<T>
 where T : IComparable<T>
{
 protected class Node
 {
 public T Data { get; set; }
 public Node left;
 public Node right;

 public Node(
 T dataP = default(T),
 Node leftP = null,
 Node rightP = null
 )
 {
 Data = dataP;
 left = leftP;
 right = rightP;
 }

 public override string ToString()
 {
 return "| " + Data.ToString() + " |";
 }
 }

 protected Node root;

 public BTree()
 {
 root = null;
 }

 public void Clear()
 {
 root = null;
 }

 public bool IsEmpty()
 {
 return root == null;
 }

 public override string ToString()
 {
 string returned = "Depth: " + Depth() + "\n";
 if (root != null)
 {
 returned += Stringify(root, 0);
 }
 return returned;
 }

 private string Stringify(Node nodeP, int depth)
 {
 string returned = "";
 if (nodeP != null)
 {
 for (int i = 0; i < depth; i++)
 {
 returned += " ";
 }
 returned += nodeP + "\n"; // Calls Node's ToString method.
 if (nodeP.left != null)
 {
 returned += "L" + Stringify(nodeP.left, depth + 1);
 }
 if (nodeP.right != null)
 {
 returned += "R" + Stringify(nodeP.right, depth + 1);
 }
 }
 return returned;
 }

 public int Depth()
 {
 int depth = 0;
 if (root != null)
 {
 depth = Depth(root, 0);
 }
 return depth;
 }

 private int Depth(Node nodeP, int depth)
 {
 int result = depth;
 int depthL = 0;
 if (nodeP.left != null)
 {
 depthL = Depth(nodeP.left, result + 1);
 }
 int depthR = 0;
 if (nodeP.right != null)
 {
 depthR = Depth(nodeP.right, result + 1);
 }
 if (nodeP.left != null || nodeP.right != null)
 {
 result = Math.Max(depthL, depthR);
 }
 return result;
 }

 public virtual bool Find(T dataP)
 {
 bool found = false;
 if (root != null)
 {
 found = Find(root, dataP);
 }
 return found;
 }

 private bool Find(Node nodeP, T dataP)
 {
 bool found = false;
 if (nodeP != null)
 {
 if (nodeP.Data.Equals(dataP))
 {
 found = true;
 }
 else
 {
 found =
 Find(nodeP.left, dataP)
 || Find(nodeP.right, dataP);
 }
 }
 return found;
 }

 public abstract void Insert(T dataP);
 public abstract bool Delete(T dataP);
}

[*(Download this code)*](https:///princomp.github.io/code/projects/Tree.zip)

### Binary Search Tree

A binary *search* tree (BST) is a specific type of binary tree that ensures that

* each node’s value is greater than all the values stored in its left subtree,
* each node’s value is less than all the values stored in its right subtree,
* a value cannot occur twice.

﻿using System;
using System.Collections.Generic;

public class BSTree<T> : BTree<T>
 where T : IComparable<T>
{
 public override void Insert(T dataP)
 {
 root = Insert(dataP, root);
 }

 private Node Insert(T dataP, Node nodeP)
 {
 if (nodeP == null)
 {
 return new Node(dataP, null, null);
 }
 else if (dataP.CompareTo(nodeP.Data) < 0) // dataP < nodeP.Data
 {
 nodeP.left = Insert(dataP, nodeP.left);
 }
 else if (dataP.CompareTo(nodeP.Data) > 0) // dataP > nodeP.Data
 {
 nodeP.right = Insert(dataP, nodeP.right);
 }
 else
 {
 throw new ApplicationException(
 "Value " + dataP + " already in tree."
 );
 }
 return nodeP;
 }

 public override bool Delete(T dataP)
 {
 return Delete(dataP, ref root);
 }

 private bool Delete(T dataP, ref Node nodeP)
 {
 bool found = false;
 if (nodeP != null)
 {
 if (dataP.CompareTo(nodeP.Data) < 0) // dataP < nodeP.Data
 {
 found = Delete(dataP, ref nodeP.left);
 }
 else if (dataP.CompareTo(nodeP.Data) > 0) // dataP > nodeP.Data
 {
 found = Delete(dataP, ref nodeP.right);
 }
 else // We found the value!
 {
 found = true;
 if (nodeP.left != null && nodeP.right != null)
 {
 nodeP.Data = FindMin(nodeP.right);
 Delete(nodeP.Data, ref nodeP.right);
 }
 else
 {
 if (nodeP.left != null)
 {
 nodeP = nodeP.left;
 }
 else
 {
 nodeP = nodeP.right;
 }
 }
 }
 }
 return found;
 }

 public override bool Find(T dataP)
 {
 bool found = false;
 if (root != null)
 {
 found = Find(root, dataP);
 }
 return found;
 }

 private bool Find(Node nodeP, T dataP)
 {
 bool found = false;
 if (nodeP != null)
 {
 if (nodeP.Data.Equals(dataP))
 {
 found = true;
 }
 else
 {
 if (dataP.CompareTo(nodeP.Data) < 0) // dataP < nodeP.Data
 {
 found = Find(nodeP.left, dataP);
 }
 else if (dataP.CompareTo(nodeP.Data) > 0) // dataP > nodeP.Data
 {
 found = Find(nodeP.right, dataP);
 }
 }
 }
 return found;
 }

 public T FindMin()
 {
 if (root == null)
 {
 throw new ApplicationException(
 "Cannot find a value in an empty tree!"
 );
 }
 else
 {
 return FindMin(root);
 }
 }

 private T FindMin(Node nodeP)
 {
 T minValue;
 if (nodeP.left == null)
 {
 minValue = nodeP.Data;
 }
 else
 {
 minValue = FindMin(nodeP.left);
 }
 return minValue;
 }
}

[*(Download this code)*](https:///princomp.github.io/code/projects/Tree.zip)