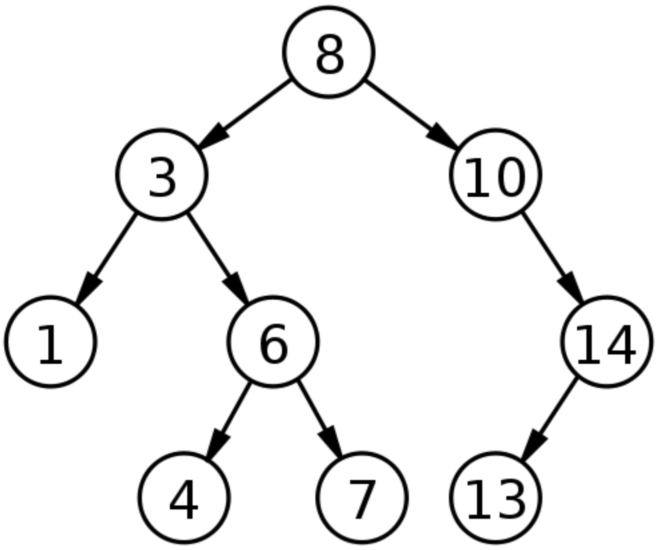
BST, Memoization, Serialization

```
-- rendu-tpcs14-login_x.zip
    |-- login_x
        |-- AUTHORS
        |-- README
        |-- TPCS14
            |-- ...
```

1 BSTs

A Binary Search Tree is defined as a group of a value and two sub-trees, potentially null. Furthermore, all the values contained in all the nodes of its left son are inferior or equal to its own value ; the values contained in all the nodes of its right son are strictly superior to its own value. It is a recursive structure.
Here is an example :



1.1 Définitions

Here are some notions that you will need.
— We call *size* of a tree the number of nodes that compose it (9 in the example).
— We call *depth* of a tree the number of "floors" of a tree. An empty tree has a depth of -1, a simple node a depth of 0 (3 in the example).

1.2 Warm up

Start your work by creating the class `BinaryTreeNode` following the description above. We want the class to be as functional as possible, so the class must be generic. Be careful, BST must have their elements comparable, so we must use conditions on the generic class.
Moreover, this class will only represent the nodes of the final structure, so it must implement `IComparable<T>` for more versatility. Enrich this class with 2 more methods.

```
/* Compute the size of the tree. We want you to write the
 * algorithm, so do not cache the size within the structure */
public int Size();

/* Compute the depth of a tree */
public int Depth();
```

2 Tree Browsing

Globally, there are 2 ways to browse a tree : the depth course and the width course. The second one will take care of all the brothers of a node before taking care of nodes' son, but the first one will take care of all the sons of a node before its brothers. The depth course can be displayed in three ways : prefix, infix and postfix, depending on if we take care of a node before, between, or after its sons. Moreover, browsing a tree is great, but doing something during the browse is better. For that we will use a delegate that we will apply on the value of each node found.
This delegate can represent a displaying function.
For now, we will only care about recursive browses.

```
public delegate void BinaryTreeDelegate(T arg);

public void PrefixOrder(BinaryTreeDelegate callback);
public void InfixOrder(BinaryTreeDelegate callback);
public void PostfixOrder(BinaryTreeDelegate callback);
```

3 Insertion & Search

Now that we can browse our tree, it would be wise to add new nodes. Don't forget that BSTs have an order relation !
Little tip : we add nodes only on a leaf, and never in the middle of a tree.
You also have to create a method that finds the first node containing a given value.

```
public void Insert(T value);
public BinaryTreeNode<T> Find(T value);
```

4 An Additionnal Level of Encapsulation

We created a class symbolizing the *nodes* of a tree, so we will now add a class that will act like an interface (not the Object Programming interface) between these nodes and what the user wants to do with a tree (for example that the tree can be empty).
This overlay also eases the removal of nodes.
We will also make its size, in order to avoid calculating it every time we need it.
All these methods are just calling the methods of `BinaryTreeNode` from the root of the tree.

```
public class BinaryTree<T>
    /* FIXME: condition on generic */
{
    /* Moved from from wherever it was */
    public delegate void BinaryTreeDelegate(T arg);

    public BinaryTree();
    public BinaryTree(T value);

    public int Size();
    public int Depth();
    public void Insert(T value);
    public BinaryNodeTree<T> Find(T value);

    public void PrefixOrder(BinaryTreeDelegate callback);
    public void InfixOrder(BinaryTreeDelegate callback);
    public void PostfixOrder(BinaryTreeDelegate callback);

    /* FIXME: attributs */
}
```

5 Width course

Now that recursivity doesn't have anymore secrets for you, let's take care of the width course that is, by nature, iterative. Because it is not recursive, it's appropriate to implement it directly in the class `BinaryTree`.
For this algorithm, use a `System.Collections.Generic.Queue`, which is an ideal data structure for this problem.

```
public void BreathFirst(BinaryTreeDelegate callback);
```

6 Bonus 1 - Delete

Try deleting the first node found corresponding to a given value.
We must differentiate three cases :

- 0 sons : Everything is all right, it's a leaf.
- 1 son : Not bad, we can replace the node by its son.
- 2 sons : Ouch, it is necessary to replace the node by its immediate successor that can be found the on the extreme left of its right sub-tree. (the smallest of the greatest values.)

```
/* Function in BinaryTree */
public void Remove(T value);

/* Functions you will probably need in BinaryTreeNode (not mandatory) */
public BinaryTreeNode<T> Remove();
private BinaryTreeNode<T> FindSuccessor(out BinaryTreeNode<T> ancestor);
```

7 Bonus 2 - Advanced Tree Construction

The problem : build a tree from its preorder and inorder traversal.
A sheet of paper and a pen will be your best friends in order to find the solution.

```
/* In BinaryTree */
public static BinaryTree<T> ReBuildTree(T[] prefix, T[] infix);
```

8 Serialization

Serialization allows the user to transform an object into "text", and to save it in a file, or even send it through a network for example. It is an extremely useful technique for your projects. The text can be either binary or as an XML. We will use the first one, but be aware that there is almost nothing to change in order to get an XML file.

8.1 In C#

```
[Serializable] /* A must-have if we want our class to be serialized. */
class A
{
  /* int is serializable, OK */
  public int a = 10;
}

class B
{
  public int a = 10;
}

[Serializable]
class C
{
  /* B is not serializable, so we must add
   * this parameter */
  [NonSerialized]
  public B b = new B();

  /* it is always serialized with the class */
  public int c = 10;
}

public class Program
{
  public static void Main(string[] args)
  {
    A a = new A();
    a.a = 15;
    IFormatter formatter = new BinaryFormatter();
    Stream stream = new FileStream("a.bin", FileMode.Create, FileAccess.Write);
    /* Write the object in the "a.bin"  file*/
    formatter.Serialize(stream, a);
    stream.Close();

    stream = new FileStream("a.bin", FileMode.Open, FileAccess.Read);
    /* Creation of the object from the file */
    A b = (A)formatter.Deserialize(stream);
    Console.WriteLine(b.a); /* Affiche 15 */
  }
}
```

Notes :

- All classes are in `System.Runtime.Serialization` and `System.Runtime.Serialization.Formatters.Binary`.
- All attributes of the class must be serializable. If not we must add `[NonSerialized]`.
- Instead of adding `[Serializable]` it is possible to implement the interface `ISerializable`. It will be necessary to implement the method `GetObjectData` that defines which one of the attributes of the class would be serialized.

8.2 A little practice

We will save our precious BSTs, thanks to serialization. Modify `BinaryTreeNode` in order to make it serializable. Add the following methods to `BinaryTree` :

```
/* Write the current tree to the file. */
public void SerializeTree(string fileName);
/* Replace the current tree by the one in the file */
public void LoadTree(string fileName);
```

9 Memoization

Memoization is a programming concept allowing the user to cut algorithms' complexities. It relies on caching intermediate results of calculations, in order to avoid re-calculating them. It allows certain algorithms like Fibonacci to reach a linear complexity, and of course constant complexity on already calculated cases.
This technique is mostly used in functional languages that occasionally lack control structures like loops. But we will still see its use today, in C#.

9.1 The `Algorithm` class

The `Algorithm` class will represent every memoizable algorithm. Because we want it to embody *every algorithm*, we will use a generic approach again.
It is really advised to read the whole subject before starting, as this is the only way for you to understand where we are leading you.
The `Algorithm` class must :

- Be abstract.
- Be generic.
- Have a `Dictionary` attribute, allowing it to save results.
- Have the abstract method `Compute`, calculating the result of the concerned algorithm. The type of the argument is not always the same as the return type.

9.2 The `Benchmark` class

Because we are working on optimisation, we will create a class that will highlight the differences between two functions (memoized or not, for example).
You must create the new class `Benchmark` and a `delegate`. The `delegate` must be generic in order to represent every algorithm taking an argument and returning a result (that can have another type). The`Benchmark` class has only one static method :

```
public static void CompareCompute<T, Y>(GenericDelegate<T, Y> fct1,
                                        GenericDelegate<T, Y> fct2,
                                        T arg);
```

This method executes the delegates with the given argument, and displays

- The execution time of the fastest in green.
- The execution time of the slowest in red.
- The difference between the two executions.

Have a look on `Stopwatch`.

9.3 The `Fibonacci` class

To get started, we will adapt the Fibonacci algorithm with this technique.
Inherit a `Fibonacci` class from your `Algorithm` class, and implement the `Compute` method.
Then create a classic Fibonacci function, and compare the two performances using the `CompareCompute` method.

9.4 The `Catalan` class

Same as previous, but with Catalan numbers. These numbers are used in counting and $C(n)$ corresponds for example to the number of binary trees with $n + 1$ leaves. This suite is recursive and defined by :

$$C(0) = 1$$

$$C(n) = \sum_{i=0}^{n-1} C_i C_{n-1-i}$$