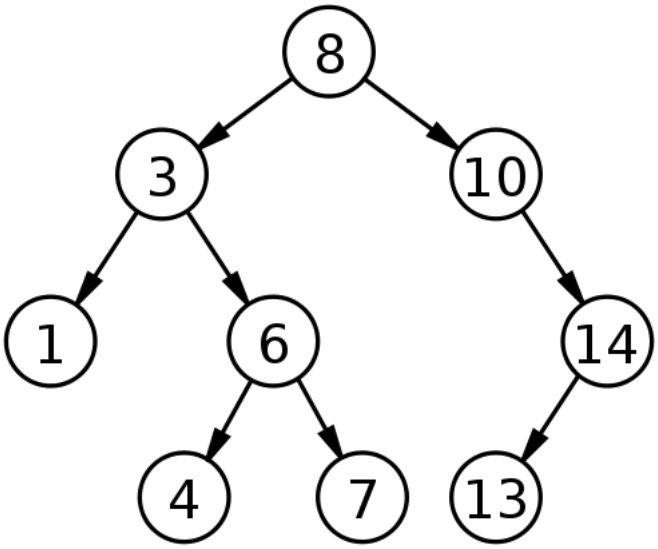
ABRs, Mémoïsation, Sérialisation

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

1 Les ABRs

Un arbre binaire de recherche est défini comme l'ensemble d'une valeur et de deux sous-arbres, potentiellement nuls. De plus, toutes les valeurs des noeuds de son sous-arbre gauche sont inférieures ou égales à sa valeur ; les valeurs des noeuds de son sous-arbre droit sont strictement supérieures à sa valeur.
On a une structure intrinsèquement récursive.
Voici un exemple :



1.1 Définitions

Voici quelques notions dont vous aurez besoin.
— On appelle la *taille* d'un arbre le nombre de noeuds qui le composent. (9 dans l'exemple)
— On appelle *profondeur* d'un arbre le nombre d'"étages" qui le composent. L'arbre vide a une profondeur de -1, un noeud seul une profondeur de 0. (3 dans l'exemple)

1.2 Échauffement

Commencez par réaliser la classe `BinaryTreeNode` suivant la description ci-dessus. On veut que cette classe soit la plus fonctionnelle possible, il faudra donc utiliser la généricité. Attention, les ABRs demandent que leurs éléments soient comparables, il faudra donc mettre des conditions à cette généricité.
De plus, cette classe ne représentera que les noeuds de notre structure finale, il faudra donc qu'elle implémente `IComparable<T>` pour plus de versatilité. Étoffez aussi cette classe de 2 méthodes :

```
/* 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 Parcours d'arbre

Il existe globalement 2 méthodes pour parcourir un arbre : le parcours profondeur et le parcours largeur. Le deuxième va traiter tous les frères avant de s'occuper des fils de chacun, contrairement à l'autre méthode qui va s'occuper de parcourir les fils de chaque noeud avant ses frères. Le parcours profondeur se décline en 3 saveurs : préfixe, infixe et postfixe, selon si l'on traite le noeud courant : avant, entre, ou après ses fils. De plus, parcourir un arbre c'est bien, mais faire quelque chose pendant le parcours, c'est mieux. Pour cela nous allons donc utiliser une délégué que nous allons appliquer sur la valeur de chacun des noeuds rencontrés.
Ce délégué peut très bien représenter une fonction d'affichage.
Pour l'instant nous ne nous occuperons que des parcours récursifs.

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

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

3 Insertion & Recherche

Maintenant que nous pouvons parcourir notre arbre, il serait judicieux d'y ajouter des noeuds. N'oubliez pas que les ABRs ont une relation d'ordre !
Petite indication : on ne rajoute des noeuds qu'en feuille d'un arbre, sûrement pas au milieu.
Réalisez aussi une méthode qui trouve le premier noeud qui contient une valeur donnée.

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

4 Un niveau d'encapsulation supplémentaire

Nous avons réalisé une classe symbolisant les *noeuds* d'un arbre, nous allons désormais ajouté une classe qui va s'occuper de faire l'interface entre ces noeuds et ce qu'un utilisateur attend vraiment d'un arbre. (par exemple qu'il puisse être vide).
Cette sur-couche facilite également fortement la suppression de noeuds.
On va aussi en profiter pour lui faire mémoriser sa taille, et ainsi ne pas avoir à la recalculer dès qu'on en a besoin.
Toutes ces méthodes ne font qu'appeler les méthodes des `BinaryTreeNode` depuis la racine de l'arbre.

```
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 Parcours largeur

Maintenant que la récursivité n'a plus de secret pour vous, passons au parcours largeur qui lui est par nature itératif. Du fait de sa nature non récursive, il peut être cohérent de l'implémenter directement dans la classe `BinaryTree`.
Pour cet algorithme, utilisez une `System.Collections.Generic.Queue`, qui est la structure de données idéale.

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

6 Bonus 1 - Suppression

Tentez, une valeur donnée, de supprimer le premier noeud rencontré qui possède la même. Il faut différencier 3 cas :

- 0 fils : tout va bien c'est une feuille.
- 1 fils : Pas mal, on peut remplacer le noeud par son fils.
- 2 fils : Ouch, il faut remplacer le noeud par son immédiat successeur qui se trouve le plus à gauche de son sous-arbre droit. (le plus petit des plus grandes valeurs)

```
/* 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 - Construction d'arbre avancée

La problématique : construire un arbre à partit de son parcours préfixe et infixe.
Une feuille et un stylo devrait être vos meilleurs amis pour trouver la solution.

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

8 Sérialisation

La sérialisation permet de transformer un objet en "texte", et ainsi de le sauvegarder sous forme de fichier, ou même de l'envoyer, via un réseau par exemple. C'est un procédé qui peut vous être extrêmement utile pour vos projets libres.
Le texte peut être formaté sois sous forme binaire, sois sous forme XML. Nous allons ici utiliser la première méthode, mais sachez qu'il n'y a presque rien à modifier pour obtenir un XML.

8.1 En C#

```
[Serializable] /* Obligatoire si l'on veut que la classe soit sérialisable */
class A
{
  /* int est serializable, tout va bien */
  public int a = 10;
}

class B
{
  public int a = 10;
}

[Serializable]
class C
{
  /* B n'est pas une classe sérialisable, on doit donc ajouter
   * ce paramètre */
  [NonSerialized]
  public B b = new B();

  /* c est toujours sérialisé avec la classe */
  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);
    /* Écrit l'objet dans le fichier "a.bin" */
    formatter.Serialize(stream, a);
    stream.Close();

    stream = new FileStream("a.bin", FileMode.Open, FileAccess.Read);
    /* Recréé l'objet à partir du fichier */
    A b = (A)formatter.Deserialize(stream);
    Console.WriteLine(b.a); /* Affiche 15 */
  }
}
```

Notes :

— Toutes les classes sont dans `System.Runtime.Serialization` et `System.Runtime.Serialization.Formatters.Binary`.

— Tous les attributs de la classe doivent être sérialisables. On doit sinon rajouter `[NonSerialized]`.

— Plutôt que de rajouter `[Serializable]` il est possible d'implémenter l'interface `ISerializable`. Il faudra alors sur-définir la méthode `GetObjectData` qui définit quels attributs de la classe seront sérialisés.

8.2 Un peu de pratique

Nous allons faire en sorte de sauvegarder nos précieux ABRs, grâce à la sérialisation. Modifier `BinaryTreeNode` pour qu'elle soit sérialisable. Ajouter également les méthodes suivantes à `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 Mémoïsation

La mémoïsation est un concept de programmation permettant de diminuer très fortement la complexité d'un algorithme. Elle correspond à la conservation des résultats des calculs effectués, afin de ne pas avoir à les recalculer. Elle permet à des algorithmes comme Fibonacci d'atteindre une complexité linaire, et bien évidement constante sur des cas déjà calculés.
Cette technique est majoritairement utilisée au sein de langages fonctionnels, qui manquent de structures de contrôle itératives comme les boucles. Néanmoins à titre pédagogique, nous allons la mettre en place aujourd'hui.

9.1 La classe `Algorithm`

La classe `Algorithm` représentera tout algorithme mémoïsable. Afin que cette classe puisse décrire *n'importe quel algorithme* nous allons réutiliser la *généricité*.
Il est fortement conseillé de lire tout le TP avant de commencer, il n'y a que comme ça que vous verrez où vous allez.
La classe `Algorithm` doit :

- Être abstraite.
- Être générique.
- Posséder un attribut `Dictionnary`, permettant de sauvegarder les resultats.
- Posséder la méthode abstraite `Compute` calculant le résultat de l'algorithme concerné. Le type de l'argument n'est pas nécessairement le même que le type de retour.

9.2 La classe `Benchmark`

Étant donné que nous travaillons sur des soucis d'optimisation, nous allons créer un utilitaire permettant de mesurer la différence de temps d'exécution entre la version mémoïsée et non mémoïsée. Vous devez donc créer une nouvelle classe `Benchmark` et un `delegate`. Le `delegate` doit être générique pour représenter tout algorithme prenant un argument et retournant un résultat d'un type qui peut être différent de celui de l'argument. La classe `Benchmark`, elle, ne comporte qu'une méthode statique :

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

Cette méthode exécute les délégués avec l'argument donné, affiche leur temps d'éxecutions respectifs en vert pour la plus rapide, en rouge pour l'autre.
Jetez un oeil du côté des `Stopwatch`.

9.3 La classe `Fibonacci`

Comme mise en jambe, on va commencer par adapter l'algorithme de Fibonacci avec cette technique.
Faites donc hériter une classe `Fibonacci` de votre classe `Algorithm`, et implémentez la méthode `Compute`.
Réalisez ensuite une fonction de calcul de Fibonacci récursive classique, et testez leurs temps d'exécutions grâce à votre outil de mesure du temps d'exécution.

9.4 La classe `Catalan`

Même chose que précédemment mais avec les nombres de Catalan. Cette suite est très utilisée dans le cadre du dénombrement et $C(n)$ correspond par exemple au nombre d'arbres binaires à $n+1$ feuilles. Cette suite est définie récursivement par :

$$C(0) = 1$$

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