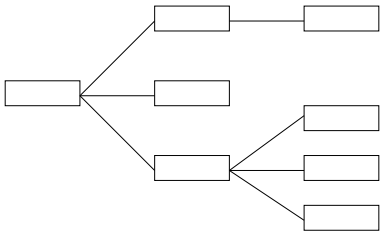
COURS : Arbres (binaires)

Distribuer la fiche IV d'ASD sur les primitives de manipulation des arbres binaires.

1 Description intuitive

La structure d'Arbre est une généralisation de la structure Liste : dans une liste, chaque élément possède un successeur (éventuellement vide), dans un arbre un élément peut avoir un nombre quelconque de successeurs :



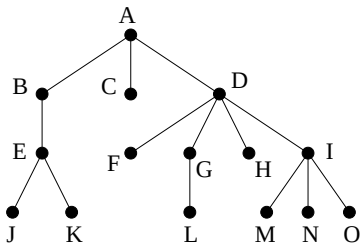
On obtient ainsi des structures qui se *ramifient* de plus en plus. Ces structures sont pour cette raison appelées *structures arborescentes*, ou plus simplement *arbres*.

De nombreuses informations peuvent être ainsi représentées : arbre généalogique, résultats de tournois sportifs, organigramme d'une société, organisation des fichiers en répertoires, schémas représentatifs d'objets de type entité, expressions arithmétiques, etc...

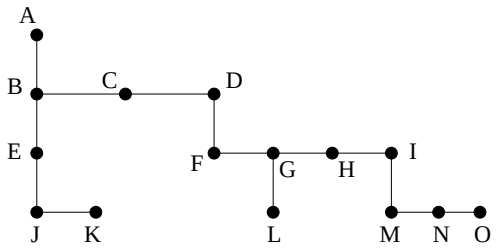
Un peu de vocabulaire (en s'appuyant sur un schéma). Dans un Arbre de TInfo :

- *sommet* : élément (de type TInfo) de la structure,
- *racine* : sommet particulier, "premier" élément, le seul qui n'a pas de prédécesseur,
- *feuille* : sommet n'ayant aucun successeur,
- *fils* : successeur d'un sommet,
- *père* : tout sommet (sauf la racine) a un unique prédécesseur, le père,
- *frère* : tous les sommets fils d'un même père sont frères,
- *sous-arbre* : les sous-arbres d'un sommet sont les arbres ayant pour racine les fils de ce sommet.

L'étude de la structure générale d'arbre est assez complexe. Nous allons ici étudier une classe particulière d'arbres, *les arbres binaires* : chaque sommet d'un tel arbre a au plus deux fils, que l'on nomme *fils gauche* et *fils droit*.
Justification : tout arbre général peut se représenter par un arbre binaire en appliquant la transformation suivante : en considérant les frères "ordonnés" de gauche à droite, à chaque sommet, on associe deux "successeurs" : son premier fils, et son frère suivant :



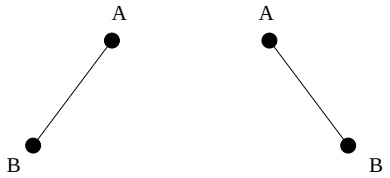
Arbre général



Arbre binaire associé

2 Les arbres binaires

Les arbres binaires vérifient la condition : tout sommet possède 0 fils, 1 fils (gauche ou droit) ou 2 fils (gauche et droit). Les deux arbres suivants sont considérés comme différents :



Nous utiliserons la définition de type suivante :

```
type TArbBin = arbre binaire de TInfo
```

Nous retrouvons également les mêmes notions de "flèches" et "adresses" que dans les listes, avec le type `TAdresse` et sa valeur particulière `NULL`.

Voici les primitives que nous utiliserons pour manipuler le type abstrait Arbre binaire :

```
Création :
  Action créerArbre (S T: TArbBin, E val: TInfo)
       // Crée un arbre dont la racine est val (pas d'arbre vide)
Consultation :
  Fonction adresseRacine (E T: TArbBin): TAdresse
  Fonction adresseFilsGauche (E T: TArbBin, adr: TAdresse): TAdresse
       // Retourne l'adresse du fils gauche de l'élément adr, NULL sinon
  Fonction adresseFilsDroit (E T: TArbBin, adr: TAdresse): TAdresse
  Fonction valeurSommet (E T: TArbBin, adr: TAdresse): TInfo
Edition (minimum...) :
  Action modifierValeurSommet (ES T: TArbBin, E adr: TAdresse, val: TInfo)
  Action insérerFilsGauche (ES T: TArbBin, E adr: TAdresse, val: TInfo )
       // Ajoute un fils gauche, s'il n'existe pas déjà, à l'élément adr
  Action insérerFilsDroit (ES T: TArbBin, E adr: TAdresse, val: TInfo)
```

Remarques et justifications : dans la suite, nous nous intéresserons plus à des algorithmes de parcours et consultation des arbres binaires que de leur édition. Nous les éditerons essentiellement dans le but de les initialiser avant de lancer un traitement. Ainsi, pour simplifier les éditions (et les primitives associées), nous considérons qu'un arbre binaire n'est jamais vide, il contient au moins un sommet (racine), et les insertions se font uniquement aux feuilles de l'arbre, pas "au milieu". De même les suppressions ne sont pas présentées. Nous pourrions éventuellement rajouter :

```
  Action supprimerFilsGauche (ES T: TArbBin, E adr: TAdresse)
       // Supprime le fils gauche de l'élément adr, si c'est une feuille
  Action supprimerFilsDroit (ES T: TArbBin, E adr: TAdresse)
```

2.1 Approche plus formelle

On peut aussi étendre aux arbres la représentation d'une liste comme un couple (élément, sous-liste). Il suffit par exemple de dire qu'un arbre binaire est un triplet (élément, sous-arbre, sous-arbre). Une autre façon aurait été de dire qu'une sous-liste peut aussi être un élément. L'arbre de l'exercice 2 peut ainsi s'écrire :

(racine, (homme, (male, (), ()), (femelle, (), ())), (machine, (), (ordinateur, (), ())))

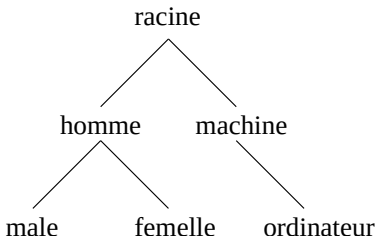
Exercice 1 : Voilà une description formelle d'un arbre. Dessinez-le.

(1, (2, (4, (7, (), ()), (8, (), ())), ()), (3, (5, (), ()), (6, (), ())))

3 Construction “à la main” d’arbres binaires

Le but ici est uniquement d’apprendre à utiliser les primitives d’insertion.

Exercice 2 : Donnez la suite d’instructions permettant de créer l’arbre binaire ci-dessous.



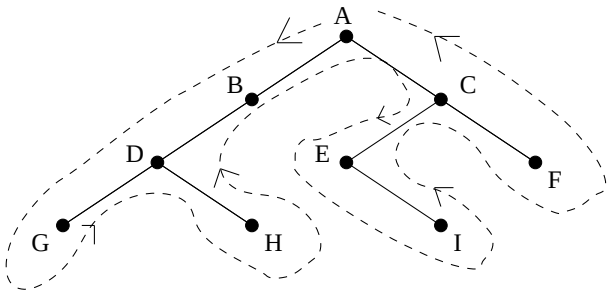
Réponse :

```
type TArbBin = arbre binaire de chaine
Action construction (S A : TArbBin)
Var ag, ad : TAdresse
Début
    créerArbre( A, "racine" );
    insérerFilsGauche( A, adresseRacine(A), "homme" );
    insérerFilsDroit( A, adresseRacine(A), "machine" );
    ag <-- adresseFilsGauche( A, adresseRacine(A) );
    insérerFilsGauche( A, ag, "male" );
    insérerFilsDroit( A, ag, "femelle" );
    ad <-- adresseFilsDroit( A, adresseRacine(A) );
    insérerFilsDroit( A, ad, "ordinateur" );
Fin
```

4 Parcours en profondeur (version récursive)

Généralement, l’utilisation d’un AB nécessite d’examiner dans un certain ordre chacun des sommets pour effectuer un même traitement (par exemple afficher le contenu). Cette opération est appelée parcours de l’AB, et peut être réalisée de plusieurs façons. Le choix dépend de ce que l’on cherche à faire. Voyons tout d’abord le parcours en profondeur.

Principe : à partir de la racine, on **descend** dans l’arbre tant que c’est possible, en privilégiant la direction **gauche** ; d’ailleurs on dit aussi parfois en profondeur à main gauche ; mais par défaut, c’est à main gauche. Lorsqu’on est bloqué, on remonte à la recherche d’une autre issue. Le parcours se termine lorsqu’on est revenu à la racine et que l’on a exploré complètement ses deux sous-arbres.



Ordre de parcours : A B D G D H D B A C E I E C F C A.
Chaque sommet ayant **deux fils** est visité **trois fois**. Lorsqu’un sommet a un seul fils (ou zéro), la

primitive adresseFilsXXX retourne NULL pour ce fils "fictif", et le parcours de ce sous-arbre s'arrête. Ainsi, chaque sommet étant visité trois fois, on peut agir sur chacun d'eux lors de chaque visite par Traitement1, Traitement2, Traitement3. D'où l'algorithme de parcours suivant :

```
Action Parcours( E A : TArbBin, Adr : TAdresse)
// parcours en profondeur du sous-arbre de A ayant pour racine le sommet
// d'adresse Adr
Début
    Si Adr <> NULL
    Alors Début
            T1( A, Adr )
            Parcours( A, adresseFilsGauche(A, Adr) )
            T2( A, Adr )
            Parcours( A, adresseFilsDroit(A, Adr) )
            T3( A, Adr )
          Fin
Fin
```

Le parcours de l'arbre s'obtient par `Parcours( A, adresseRacine(A) )`.

On remarque que cet algorithme, naturellement récursif, réalise le parcours de l'arbre binaire complété : les sommets fictifs rajoutés correspondent à des sous-arbres vides (Adr=NULL) et aucun traitement n'est effectué.
L'arbre complété représente *l'arbre des appels récursifs* de cette action `Parcours`.

Trois cas particuliers de ce parcours en profondeur sont des ordres classiques d'exploration des sommets d'un AB :

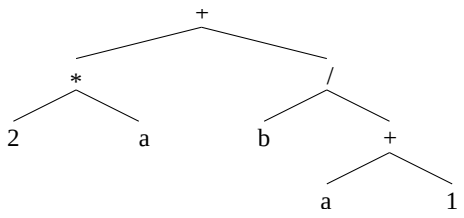
1. Ordre préfixe : T2 et T3 absents
2. Ordre infixe : T1 et T3 absents
3. Ordre postfixe : T1 et T2 absents

Exercice 3 : Affichage d'un AB
Ecrire l'action `ParcoursAffiche` qui affiche le contenu de l'arbre en ordre préfixe. Indiquer pour infixe et postfixe.
Réponse :

```
Action ParcoursAffiche( E A : TArbBin, Adr : TAdresse)
Début
    Si Adr <> NULL
    Alors Début
            écrire( valeurSommet( A, Adr ) )
            ParcoursAffiche( A, adresseFilsGauche(A, Adr) )
            ParcoursAffiche( A, adresseFilsDroit(A, Adr) )
          Fin
Fin

Utilisation : ParcoursAffiche( A, adresseRacine(A) )
```

Remarque : lorsque l'arbre code une expression arithmétique, ces ordres de parcours correspondent aux notations classiques préfixe, infixe et postfixe.



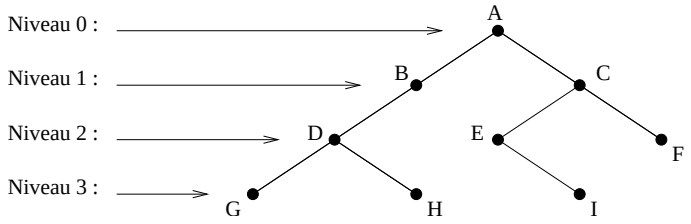
L'expression : 2*a + b / (a+1)
- préfixe : + * 2 a / b + a 1
- infixe : 2 * a + b / a + 1
- postfixe : 2 a * b a 1 + / +

On remarque que seul l'ordre infixe, le plus "sympathique", est FAUX (il faudrait rajouter des parenthèses).

Complexité : si les traitements sont réalisés en temps constant, le parcours en profondeur est linéaire, $O(n)$ où n est le nombre de sommets.

5 Parcours en largeur

L'ordre de parcours est lié à la notion de niveau dans l'arbre : les sommets de même niveau sont les sommets dont la distance à la racine est la même. Chaque niveau est alors parcouru de gauche à droite :



Ordre de parcours : A B C D E F G H I.

Exercice 4 : Écrire une action réalisant le parcours en largeur d'un AB.
Réponse : Le problème ici est que lorsqu'on visite un sommet, on a un accès immédiat à ses deux fils, mais ceux-ci ne doivent être visités que lorsque le niveau courant sera terminé ; on utilise donc une file de TAdresse permettant de mémoriser les adresses de ces deux fils qui ne seront parcourus qu'ultérieurement.

```
Action ParcoursLargeur(E A : TArbBin)
var Adr : TAdresse
    F : File de TAdresse
Début
    Adr <-- adresseRacine( A )
    Si Adr <> NULL    // pas utile ici car arbre non vide
    Alors Début
            créerFile( F )
            enfiler( F, Adr )
            Tant Que Non fileVide( F )
            Faire Début
                    Adr <-- valeurPremier( F)
                    défiler( F )
                    Traitement( A, Adr )
                    Si adresseFilsGauche( A, Adr ) <> NULL
                    Alors enfiler( F, adresseFilsGauche( A, Adr ) )
                    Si adresseFilsDroit( A, Adr ) <> NULL
                    Alors enfiler( F, adresseFilsDroit( A, Adr ) )
                  Fin
          Fin
Fin
```

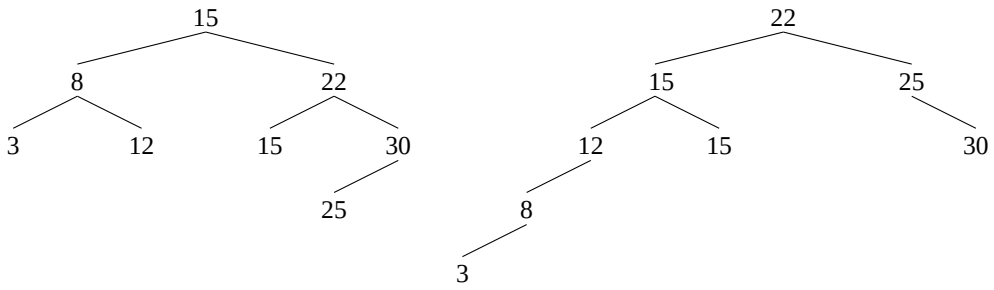
Complexité : si les traitements sont réalisés en temps constant, le parcours en largeur est linéaire, $O(n)$ où n est le nombre de sommets.

6 Arbres Binaires de Recherche

Un *Arbre Binaire de Recherche* (ABR) est un arbre binaire dont les sommets sont organisés (ou ordonnés) de la façon suivante :

chaque sommet a une valeur supérieure aux valeurs des sommets de son sous-arbre gauche et inférieure ou égale aux valeurs des sommets de son sous-arbre droit.

Soit par exemple



Remarquons que ces deux ABR contiennent exactement les mêmes informations. Le premier est *mieux équilibré* car il est de profondeur moindre. (nous y reviendrons par la suite).

6.1 Parcours infixe d'un ABR - Tri

Exercice 5 : Parcours d'un ABR
Sur les exemples d'ABR précédents, faites tourner les algorithmes de parcours préfixe, infixe, postfixe. Que remarquez-vous ?

Un parcours infixe d'un ABR produit la liste *triée* des éléments contenus dans celui-ci. On peut donc utiliser une telle structure pour écrire un algorithme de tri :

1. on insère un à un les éléments à trier dans un ABR,
2. on effectue un parcours infixe de l'ABR obtenu.

6.2 Recherche dans un ABR

La structure d'ABR présente entre autre l'avantage de permettre une recherche très efficace : on peut descendre directement de la racine vers le sommet que l'on recherche en choisissant de descendre par la gauche ou par la droite selon le résultat de la comparaison entre la valeur cherchée et la valeur du sommet visité. Ainsi, il n'est pas nécessaire de parcourir la totalité des sommets, et le nombre de comparaisons est au plus égal à la profondeur de l'ABR + 1.

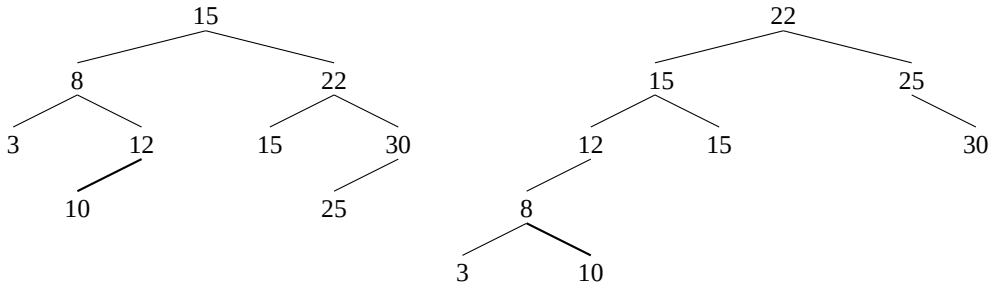
Remarque : l'élément minimum se trouve au bout de la branche gauche et l'élément maximum au bout de la branche droite.

Exercice 6 : Recherche dans un ABR
Montrer sur un exemple le principe de la recherche, faire remarqer qu'il n'y a pas besoin de récursivité ici.

Remarque : en terme de complexité, si n est le nombre de sommets de l'ABR, la recherche est en $O(logn)$ s'il est bien équilibré, et en $O(n)$ dans le cas le pire (un chemin "rectiligne" gauche ou droite).

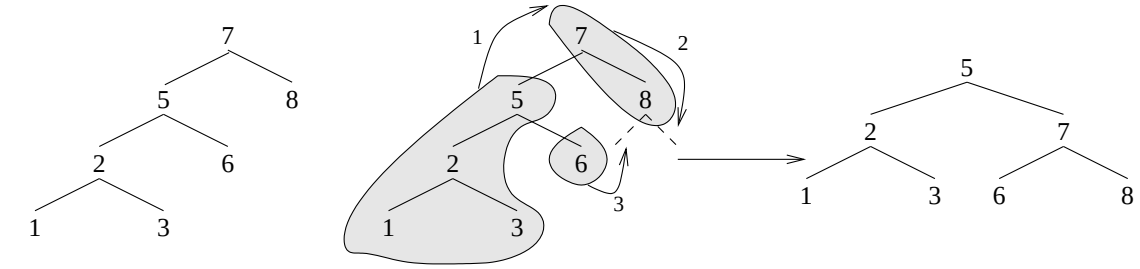
6.3 Insertion dans un ABR - Équilibrage

Pour créer un ABR, comme par exemple dans l'algorithme de tri vu plus haut, on a besoin d'insérer un à un les éléments dans l'ABR.
La façon la plus simple d'insérer un élément consiste à insérer une nouvelle feuille *bien placée* : on descend selon le principe ABR, quand on ne peut plus, on insère une nouvelle feuille. Il est facile de vérifier qu'il n'existe qu'une seule façon possible de rajouter une feuille bien placée dans un ABR. Par exemple, rajoutons 10 dans l'exemple précédent, on obtient :



Mais ce n'est pas la meilleure méthode : la complexité des algorithmes définis sur les ABR étant fortement liée à la profondeur de l'arbre, il est préférable d'insérer d'une façon qui tienne compte de l'équilibre de l'arbre. En effet, dans cet algorithme, l'équilibrage de l'ABR construit est totalement dépendant de l'ordre dans lequel les éléments sont insérés. Les cas les pires sont bien sûr d'avoir une liste d'éléments triés, en ordre croissant ou décroissant, car on obtient un arbre "rectiligne" droit ou gauche, qui ressemble plus à une liste chaînée qu'à un arbre. Il est donc important de pouvoir équilibrer des ABR ou de savoir les construire équilibrés.

Il existe des algorithmes qui insèrent toujours aux feuilles, mais qui vérifient l'équilibrage de l'ABR, et le rééquilibrent si besoin (cad dès que la différence entre les hauteurs de deux branches d'un sous-arbre est supérieure ou égale à 2). Il est possible d'effectuer une *rotation* :



ABR non équilibré — Rotation — ABR équilibré

Autre exemple trouvé dans les notes de Michel Marcus : Il existe d'autres algorithmes d'insertion qui permettent d'insérer "à l'intérieur" de l'arbre, par exemple à la racine. Le principe, pour ajouter X, consiste à couper l'ABR A en deux ABR, l'un G contenant toutes les valeurs inférieures à X, l'autre D toutes les valeurs supérieures ou égales à X ; on forme alors l'ABR A' de racine X et SAG G et SAD D. Le plus délicat est évidemment la coupure, et nous passons.
voir d'autres algos ? !...
COMPLETER cette partie avec des exemples.