UNIVERSITÉ Grenoble Alpes

INF201
Algorithmique et Programmation Fonctionnelle
Cours 11 : Structures arborescentes

Année 2019 - 2020

Plan

A propos d’arbres (0)

Motivation

Représenter une “collection” d’éléments de même type ?

A propos d'arbres (0)

Motivation

Représenter une “collection” d'éléments de même type ?

La notion de liste (ou séquence)

- permet d'implémenter des **séquences**, des **ensembles**, des **multi-ensembles**
- chaque élément a (au plus) un précédent et un suivant
  $\rightarrow$ notion d'ordre total

A propos d'arbres (0)

Motivation

Représenter une “collection” d'éléments de même type ?

La notion de liste (ou séquence)

- permet d'implémenter des **séquences**, des **ensembles**, des **multi-ensembles**
- chaque élément a (au plus) un précédent et un suivant
  $\rightarrow$ notion d'ordre total

Représenter une classification d'espèces (ex : des “êtres vivants”) ?

- “être vivant” = différentes espèces
  mammifère, insecte, oiseau, etc
- chaque espèce est-elle même divisée en “sous-espèces”
  - oiseau = rapace, passereaux, etc.
  - mammifères = rongeurs, canidés, etc.
  - insectes = coléoptères, diptères, etc.

A propos d'arbres (0)

Motivation

Représenter une "collection" d'éléments de même type ?

La notion de liste (ou séquence)

- permet d'implémenter des **séquences**, des **ensembles**, des **multi-ensembles**
- chaque élément a (au plus) un précédent et un suivant
  $\rightarrow$ notion d'ordre total

Représenter une classification d'espèces (ex : des "êtres vivants") ?

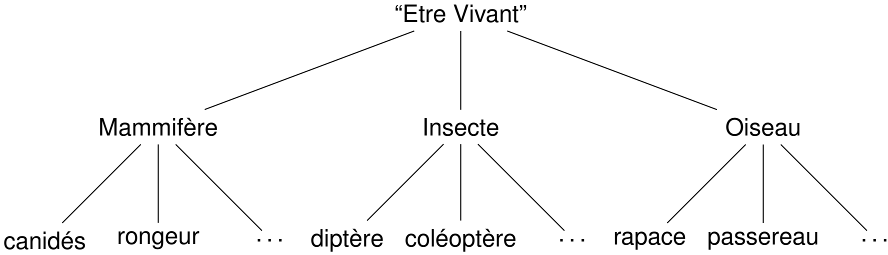
- "être vivant" = différentes espèces
  mammifère, insecte, oiseau, etc
- chaque espèce est-elle même divisée en "sous-espèces"
  - oiseau = rapace, passereaux, etc.
  - mammifères = rongeurs, canidés, etc.
  - insectes = coléoptères, diptères, etc.

$\boxed{\rightarrow \text{notion d'ordre partiel } (\neq \text{ structure de liste})}$

A propos d'arbres (1)

intuition

Classification d'espèces :



Remarque

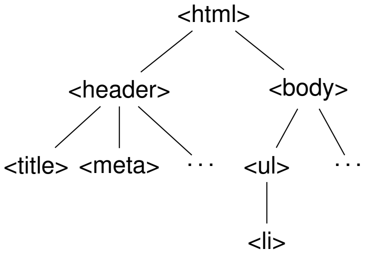
- noeuds avec étiquette, répétition possible d'étiquette
- noeud “racine”, noeuds sans/avec “sous-arbres”, noeud “père”
- structure hiérarchique
  - notion de niveau dans l'arbre
  - partition des noeuds en sous-arbres disjoints

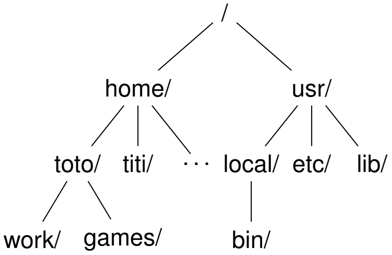
A propos d’arbres (2)

intuition

Intérêt : fournir une notion de hiérarchie (ordre partiel)
(contrairement aux listes = structure séquentielle, ordre total)

- facilite l’accès aux données
  (ex : système de fichiers, répertoires et sous-répertoires)
- permet de structurer l’information
  (ex : document HTML, organigramme, table des matières, etc.)
- permet de représenter des niveaux d’imbrications (parenthésage), ou des priorités (expressions arithmétiques)
- etc.



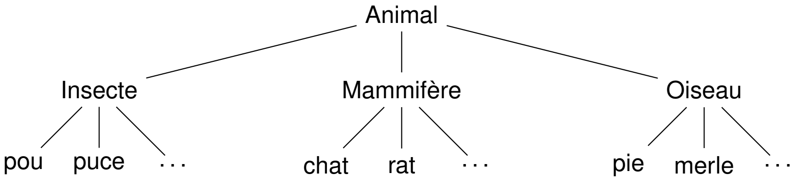


Arbre (étiqueté)

Un **arbre** est une structure récursive qui est :

- soit vide
- soit un noeud auquel est associé :
  - une étiquette
  - des fils : une séquence d'**arbres** (évent. vide)

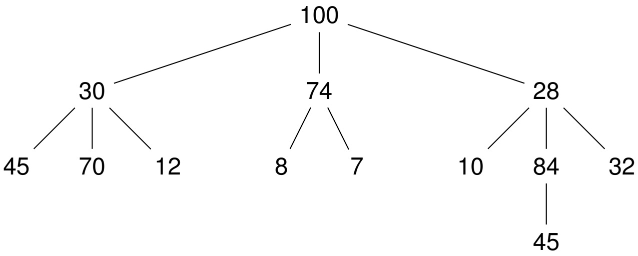
$\rightarrow$ permet de stocker des éléments (les étiquettes) **de même type**



Vocabulaire

- Le noeud “le plus haut” est la racine
- La donnée associée à un noeud est son étiquette (ou label, ou élément)
- Les (sous-)arbres associés à un noeud sont ses fils, noeud père
- Un noeud sans fils est une feuille
- le chemin au noeud $n1$ est une séquence de noeuds père $\rightarrow$ fils allant de la racine à $n1$
- niveau d’un noeud :
  longueur (en nombre de noeud) du chemin à ce noeud
- hauteur (ou profondeur) d’un arbre :
  le niveau d’un noeud de niveau maximal
- taille d’un arbre : le nombre de noeuds qu’il contient

Exemple



- racine : 100
- étiquettes : 100, 30, 74, 28, 45, 70, 12, 8, 7, 10, 84, 32, 45
- feuilles : 45, 70, 12, 8, 7, 10, 45, 32
- fils du noeud 30 : 45, 70, 12
- 100 est le père de 30
- 100 est au niveau 1, 7 est au niveau 3
- la hauteur de l'arbre est 4
- [100;30;12] est le chemin au noeud d'étiquette 12

Plan

Un arbre est un arbre binaire si chaque noeud a *au plus* deux fils
Formellement :

$$Abin(Elt) = \{Vide\} \cup \{Noeud(Ag, e, Ad) \mid e \in Elt \wedge Ag, Ad \in Abin(Elt)\}$$

Un arbre est un arbre binaire si chaque noeud a *au plus* deux fils
Formellement :

$$Abin(Elt) = \{Vide\} \cup \{Noeud(Ag, e, Ad) \mid e \in Elt \wedge Ag, Ad \in Abin(Elt)\}$$

Exemple : Arbre binaire sur des entiers

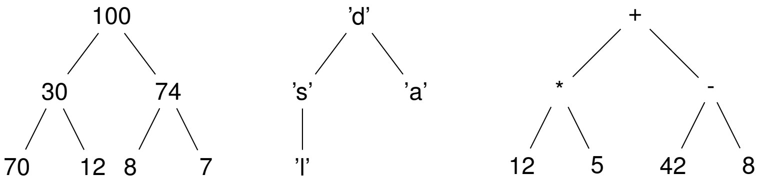
$$Abin(\mathbb{N}) = \{Vide\} \cup \{Noeud(Ag, e, Ar) \mid e \in \mathbb{N} \wedge Ag, Ad \in Abin(\mathbb{N})\}$$

Un arbre est un arbre binaire si chaque noeud a *au plus* deux fils
Formellement :

$$Abin(Elt) = \{Vide\} \cup \{Noeud(Ag, e, Ad) \mid e \in Elt \wedge Ag, Ad \in Abin(Elt)\}$$

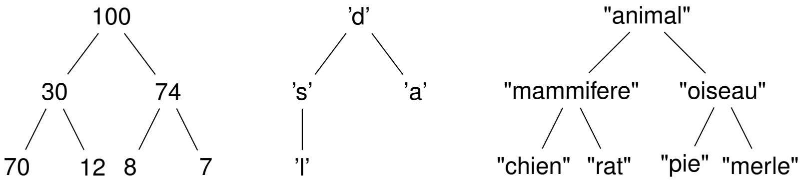
Exemple : Arbre binaire sur des entiers

$$Abin(\mathbb{N}) = \{Vide\} \cup \{Noeud(Ag, e, Ar) \mid e \in \mathbb{N} \wedge Ag, Ad \in Abin(\mathbb{N})\}$$



Vocabulaire

- Le premier (resp. second) fils est appelé fils gauche (resp. fils droit)
- Un arbre binaire *a* est complet ssi $\mathrm{taille}(a) = 2^{\mathrm{hauteur}(a)} - 1$



Définir le type `arbre_binaire` ?

Définir le type `arbre_binaire` ?
c'est un type somme, récursif, avec deux constructeurs :

- le constructeur `Vide` : l'arbre vide
  `Vide` $\in$ `arbre_binaire`
- le constructeur `Noeud` :
  ajout d'un noeud racine à partir d'une étiquette, d'un fils gauche et d'un fils droit
  `Noeud` $\in$ `etiq` $\times$ `arbre_binaire` $\times$ `arbre_binaire`

Arbres binaires d'entiers
En OCaml

Définir le type `arbre_binaire` ?
c'est un type somme, récursif, avec deux constructeurs :

- le constructeur `Vide` : l'arbre vide
  `Vide` ∈ `arbre_binaire`
- le constructeur `Noeud` :
  ajout d'un noeud racine à partir d'une étiquette, d'un fils gauche et d'un fils droit
  `Noeud` ∈ `etiq` × `arbre_binaire` × `arbre_binaire`

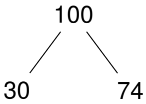
En OCaml :

```
type etiq = ... (* un type quelconque *)
type arbre_binaire =
  | Vide
  | Noeud of etiq * arbre_binaire * arbre_binaire
```

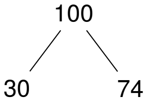
ou

```
type arbre_binaire =
  | Vide
  | Noeud of arbre_binaire * etiq * arbre_binaire
```

Exemple d’éléments du type `arbre_binaire`

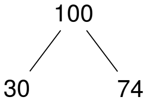


Exemple d'éléments du type `arbre_binaire`

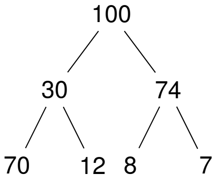



```
let ab1 =
   Noeud (100,
       Noeud (30,Vide,Vide),
       Noeud (74,Vide,Vide)
    )
```

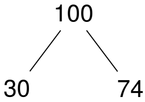
Exemple d'éléments du type `arbre_binaire`



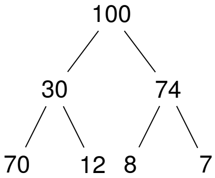

```
let ab1 =
   Noeud (100,
       Noeud (30,Vide,Vide),
       Noeud (74,Vide,Vide)
    )
```



Exemple d'éléments du type `arbre_binaire`




```
let ab1 =
   Noeud (100,
       Noeud (30,Vide,Vide),
       Noeud (74,Vide,Vide)
    )
```




```
let ab2 =
   Noeud(
    100,
    Noeud(30,
       Noeud(70,Vide,Vide),
       Noeud(12,Vide,Vide)
      ),
    Noeud(74,
       Noeud(8,Vide,Vide),
       Noeud(7,Vide,Vide)
      )
   )
```

Fonctions sur les arbres binaires ?

Ecrire une fonction qui prend un (des) arbre(s) en paramètres ?

$$\texttt{f} : \texttt{arbre\_bin} \rightarrow \ldots \rightarrow \ldots$$

avec

```
type arbre_binaire =
  | Vide
  | Noeud of etiq * arbre_binaire * arbre_binaire
```

Fonctions sur les arbres binaires ?

Ecrire une fonction qui prend un (des) arbre(s) en paramètres ?

$$\texttt{f} : \texttt{arbre\_bin} \to ... \to ...$$

avec

```
type arbre_binaire =
  | Vide
  | Noeud of etiq * arbre_binaire * arbre_binaire
```

Le type `arbre_binaire` est un type récursif . . .
→ les fonctions sur les arbres sont des fonctions récursives

- Cas de base : l'arbre est vide

  f (Vide) = ...

- Cas récursif : l'arbre est non vide

```
f (Noeud (e, fg, fd)) = ...
    (* appels recursifs sur fg et/ou fd *)
```

Terminaison ?

Fonctions sur les arbres binaires ?

Ecrire une fonction qui prend un (des) arbre(s) en paramètres ?

f : arbre_bin → ... → ...

avec

```
type arbre_binaire =
  | Vide
  | Noeud of etiq * arbre_binaire * arbre_binaire
```

Le type `arbre_binaire` est un type récursif ...
→ les fonctions sur les arbres sont des fonctions récursives

- Cas de base : l'arbre est vide

  f (Vide) = ...

- Cas récursif : l'arbre est non vide

```
f (Noeud (e, fg, fd)) = ...
    (* appels recursifs sur fg et/ou fd *)
```

Terminaison ?

appels récursifs sur des sous-arbres “plus petits”
(mesure = taille de l'arbre)

Quelques fonctions (classiques) sur les arbres

```
type arbre_binaire =
  | Vide
  | Noeud of etiq * arbre_binaire * arbre_binaire
```

Taille : fonction qui calcule le nombre de noeuds d'un arbre binaire.

Quelques fonctions (classiques) sur les arbres

```
type arbre_binaire =
  | Vide
  | Noeud of etiq * arbre_binaire * arbre_binaire
```

Taille : fonction qui calcule le nombre de noeuds d'un arbre binaire.

```
let rec taille (a:arbre_binaire):int=
 match a with
  | Vide → 0
  | Noeud (_, a1, a2) → 1 + (taille a1) + (taille a2)
```

Quelques fonctions (classiques) sur les arbres

```
type arbre_binaire =
   | Vide
   | Noeud of etiq * arbre_binaire * arbre_binaire
```

Taille : fonction qui calcule le nombre de noeuds d'un arbre binaire.

```
let rec taille (a:arbre_binaire):int=
 match a with
  | Vide → 0
  | Noeud (_, a1, a2) → 1 + (taille a1) + (taille a2)
```

Exercices

Définir les fonctions suivantes :

- `somme` : renvoie la somme des éléments d'un arbre (d'entiers)
- `hauteur` : renvoie la hauteur d'un arbre (d'entiers)
- `maximum` : renvoie l'élément maximal d'un arbre (d'entiers)

$\rightarrow$ On peut paramétrer un arbre binaire par le type de ses éléments

```
type α arbre_binaire =
  | Vide
  | Noeud of α * α arbre_binaire * α arbre_binaire
```

Permet de définir plusieurs types "arbres binaires" :
`int arbre_binaire`, `char arbre_binaire`,
`string arbre_binaire`,...

DEMO: Définition d'arbres binaires

Appartient :
existence d'un élément de type α dans un α `arbre_binaire` ?

Appartient :
existence d'un élément de type α dans un α arbre_binaire ?

```
let rec appartient (elt:α) (a:α arbre_binaire):bool =
 match a with
  | Vide → false
  | Noeud (e,ag,ad) →
     (e=elt) || appartient elt ag || appartient elt ad
```

Liste des éléments d'un arbre :
Etant donné un α arbre_binaire, renvoie la α liste de ses éléments

Appartient :
existence d'un élément de type α dans un α arbre_binaire ?

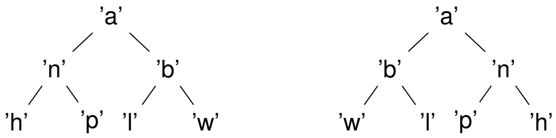
```
let rec appartient (elt:α) (a:α arbre_binaire):bool =
 match a with
  | Vide → false
  | Noeud (e,ag,ad) →
     (e=elt) || appartient elt ag || appartient elt ad
```

Liste des éléments d'un arbre :
Etant donné un α arbre_binaire, renvoie la α liste de ses éléments

```
let rec liste_elem (a:α arbre_binaire):α list=
 match a with
  | Vide → []
  | Noeud (elt,ag,ad) → (liste_elem ag)@(elt::(liste_elem ad))
```

Exercices : Définir les fonctions suivantes

- `taille`: nombre de noeuds d'un arbre binaire
- `feuilles`: liste des feuilles d'un arbre binaire
- `est_complet`: indique si un arbre binaire est un arbre "complet"
- `miroir`: image miroir d'un arbre binaire



Arbres binaires et ordre supérieur

On peut identifier plusieurs “schémas de fonction” sur les arbres :

Arbres binaires et ordre supérieur

On peut identifier plusieurs “schémas de fonction” sur les arbres :

- produire un nouvel arbre en appliquant une fonction à chaque noeud ($\sim$ opérateur **map**)
  - incrémenter toutes les étiquettes
  - remplacer chaque étiquette par la somme cumulée des étiquettes de ses fils

map (f : $\alpha \rightarrow \beta$) (a : α arbre_binaire) : β arbre_binaire =

...

Arbres binaires et ordre supérieur

On peut identifier plusieurs “schémas de fonction” sur les arbres :

- produire un nouvel arbre en appliquant une fonction à chaque noeud ($\sim$ opérateur **map**)
  - incrémenter toutes les étiquettes
  - remplacer chaque étiquette par la somme cumulée des étiquettes de ses fils

  map (f : $\alpha \rightarrow \beta$) (a : α arbre_binaire) : β arbre_binaire =

  ...

- produire un résultat en “accumulant” une valeur lors d’un parcours complet de tous les noeuds d’un arbre ($\sim$ opérateur **fold**)
  - nombre de noeuds, nombre de feuilles
  - liste des étiquettes

  fold (f:$\alpha \rightarrow \beta \rightarrow \beta \rightarrow \beta$) (acc:$\beta$) (a:$\alpha$ arbre_binaire) : β =

Arbres binaires et ordre supérieur

On peut identifier plusieurs “schémas de fonction” sur les arbres :

- produire un nouvel arbre en appliquant une fonction à chaque noeud ($\sim$ opérateur **map**)
  - incrémenter toutes les étiquettes
  - remplacer chaque étiquette par la somme cumulée des étiquettes de ses fils

  map (f : $\alpha \to \beta$) (a : α arbre_binaire) : β arbre_binaire =
  ...

- produire un résultat en “accumulant” une valeur lors d’un parcours complet de tous les noeuds d’un arbre ($\sim$ opérateur **fold**)
  - nombre de noeuds, nombre de feuilles
  - liste des étiquettes

  fold (f:$\alpha \to \beta \to \beta \to \beta$) (acc:$\beta$) (a:$\alpha$ arbre_binaire) : β =

Différents ordres de parcours possibles d’un noeud `Noeud (elt, ag, ad)`

- traiter `elt`, puis parcourir `ag`, puis parcourir `ad` $\mapsto$ parcours prefixé
- parcourir `ag`, puis traiter `elt`, puis parcourir `ad` $\mapsto$ parcours infixé
- parcourir `ag`, puis parcourir `ad`, puis traiter `elt` $\mapsto$ parcours postfixé

Exemple d'opérateur “fold”

`fold_gauche_droite_racine`:
applique une fonction `f`

- à la racine
- et aux résultats obtenus (récursivement) sur les fils droit et gauche

Exemple d'opérateur "fold"

`fold_gauche_droite_racine`:
applique une fonction `f`

- à la racine
- et aux résultats obtenus (récursivement) sur les fils droit et gauche

```
let rec fold_gdr (f:α → β → β → β) (acc:β) (a:α arbre_binaire):β=
 match a with
  | Vide → acc
  | Noeud (elt, ag, ad) →
    let rg = fold_gdr f acc ag
     and rd = fold_gdr f acc ad
       in f elt rg rd
```

Exemple d'opérateur "fold"

`fold_gauche_droite_racine`:
applique une fonction `f`

- à la racine
- et aux résultats obtenus (récursivement) sur les fils droit et gauche

```
let rec fold_gdr (f:α → β → β → β) (acc:β) (a:α arbre_binaire):β=
 match a with
   | Vide → acc
   | Noeud (elt, ag, ad) →
     let rg = fold_gdr f acc ag
       and rd = fold_gdr f acc ad
           in f elt rg rd
```

En utilisant la fonction `fold_gdr`, redéfinir les fonctions suivantes :

- `taille`
- `hauteur`
- `miroir`

Chemins dans un arbre

Exercice : fonction *chemins*

→ Déterminer l'ensemble des **plus longs chemins** dans un arbre ?
Pour s'aider :

- Comment représenter un ensemble de chemins ?
- Définir une fonction `ajouter_a_tous` qui ajoute un élément en tête de chaque chemin de cet ensemble

Chemins dans un arbre

chemin = Seq(Elt)

Ajout à tous

- Spécification:
  - Profil: $\mathit{ajout\_a\_tous} : \mathit{Elt} * \mathit{Seq}(\mathit{Seq}(\mathit{Elt})) \rightarrow \mathit{Seq}(\mathit{Seq}(\mathit{Elt}))$
  - Sémantique :
    `ajout_a_tous` (`n`, [`ch1`;...;`chn`]) = [`n`::`ch1` ; ...; `n`::`chn`]
- Implémentation:

Chemins dans un arbre

chemin = Seq(Elt)

Ajout à tous

- Spécification:
  - Profil: $ajout\_a\_tous : Elt * Seq(Seq(Elt)) \rightarrow Seq(Seq(Elt))$
  - Sémantique :
    `ajout_a_tous` (n, [ch1;...;chn]) = [n::ch1 ; ...; n::chn]
- Implémentation:
  1. `ajout_a_tous` (n,[]) = []
  2. `ajout_a_tous` (n,c::cs) = (n::c) :: (`ajout_a_tous` (n,cs))

Chemins dans un arbre

chemin = Seq(Elt)

Ajout à tous

- Spécification:
  - Profil: $ajout\_a\_tous : Elt * Seq(Seq(Elt)) \rightarrow Seq(Seq(Elt))$
  - Sémantique :
    `ajout_a_tous` (n, [ch1;...;chn]) = [n::ch1 ; ...; n::chn]
- Implémentation:
  1. `ajout_a_tous` (n,[]) = []
  2. `ajout_a_tous` (n,c::cs) = (n::c) :: (`ajout_a_tous` (n,cs))

Chemins Maximaux

- Spécification
  - Profil: $Chemins : Abin(Elt) \rightarrow Seq(Chemins)$
  - Sémantique : `chemins`(a) est l'ensemble des chemins maximaux de `a`.
- Implémentation :

Chemins dans un arbre

chemin = Seq(Elt)

Ajout à tous

- Spécification:
  - Profil: $ajout\_a\_tous : Elt * Seq(Seq(Elt)) \to Seq(Seq(Elt))$
  - Sémantique :
    `ajout_a_tous` (`n`, [`ch1`;...;`chn`]) = [`n`::`ch1` ; ...; `n`::`chn`]
- Implémentation:
  1. `ajout_a_tous` (`n`,[]) = []
  2. `ajout_a_tous` (`n`,`c`::`cs`) = (`n`::`c`) :: (`ajout_a_tous` (`n`,`cs`))

Chemins Maximaux

- Spécification
  - Profil: $Chemins : Abin(Elt) \to Seq(Chemins)$
  - Sémantique : `chemins`(`a`) est l'ensemble des chemins maximaux de `a`.
- Implémentation :
  1. `chemins` (`Vide`) = [[]] : `Seq`(`Chemins`) = `Seq`(`Seq`(`Elt`))
  2. `chemins` (`Noeud` (`Ag`,`e`,`Ad`)) = `ajouter_a_tous` (`e` , `chemins`(`g`) `@` `chemins`(`d`))

Plan

Généralités sur les arbres

Arbres Binaires

Arbres Binaires de Recherche

Motivation : Recherche d'un élément dans un ensemble *E*

Solution 1 : *E* est représenté par une liste

```
let rec appartient (elt :'a) (l : α list) : bool =
  match l with
   |[] → false
   |e::lprime → (e=elt) || (appartient elt lprime)
```

Si `elt` $\notin E$: exécution de `appartient` = parcours de **toute** la liste (|`l`| comparaisons).

Motivation : Recherche d'un élément dans un ensemble E

Solution 1 : E est représenté par une liste

```
let rec appartient (elt :'a) (l : α list) : bool =
 match l with
  |[] → false
  |e::lprime → (e=elt) || (appartient elt lprime)
```

Si `elt` $\notin E$: exécution de `appartient` = parcours de **toute** la liste (|`l`| comparaisons).

Solution 2 : E (**ordonné**) est représenté par une liste **croissante**

```
let rec appartient (elt :'a) (l : α list) : bool =
 match l with
  |[] → false
  |e::lprime → (e=elt) || (e<elt) && (appartient elt lprime)
```

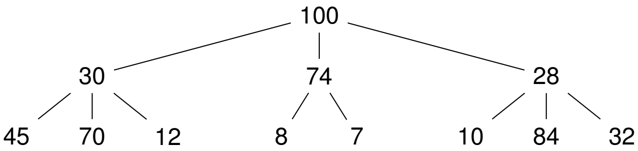
Si `elt` $\notin E$: exécution de `appartient` = parcours de **toute** la liste (|`l`| comparaisons).

→ Peut-on réduire ce nombre de comparaisons ???

Représenter l'ensemble *E* par un arbre ?

Retour sur la fonction `appartient` ...

```
let rec appartient (elt:'a) (a:'a abin):bool =
 match a with
  | Vide → false
  | Noeud (e,ag,ad) →
     (e=elt) || (appartient elt ag) || (appartient elt ad)
```



- Comment être certain qu'un élément n'appartient pas à un arbre ?
  ↪ en parcourant l'arbre complet (similaire à la recherche dans une liste)
- Le nombre de comparaison depend encore de la taille de l'arbre (nombre total d'éléments)

→ optimisation possible :
"ranger" ces éléments dans l'arbre en fonction de leur valeurs relatives ?

Arbre Binaire de Recherche : définition

Définition: Arbre Binaire de Recherche (ABR)

Soit $(\mathit{Elt}, <)$ un ensemble totalement ordonné et soit $\mathcal{A}$ un arbre binaire dont les éléments/étiquettes sont de type *Elt* ($\mathcal{A} \in \mathit{Abin}(\mathit{Elt})$).

$\mathcal{A}$ est un ABR ssi, pour tout noeud `n=Noeud(elt,ag,ad)`, avec `e` l'étiquette/élément associé à `n`, et `ag` (resp. `ad`) le fils gauche (resp. droit) de `n`, nous avons :

1. `e` est supérieur ou égal à tous les éléments de `ag` ;
2. `e` est strictement inférieur à tous les éléments de `ad` ;
3. `ag` et `ad` sont des ABR

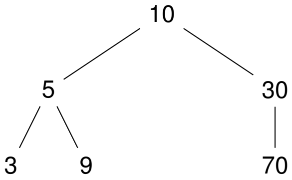
Exercice : un arbre binaire est-il un ABR ?

Définir la fonction `est_abr` qui vérifie si un arbre binaire est bien un ABR.

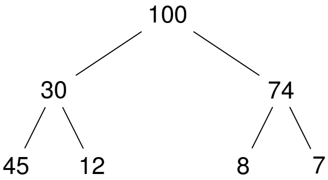
Arbre Binaire de Recherche

exemple et contre-exemple

Un arbre qui est un ABR :



Un arbre qui n'est PAS un ABR :



Retour sur la fonction `appartient`

On peut maintenant exploiter les propriétés des ABR ...

Recherche d'un élément dans un ABR :

```
let rec appartient (elt:' a) (a:' a abr):bool=
  match a with
    | Vide → false
    | Noeud (e, ag, ad) →
      (e=elt)
      || (e>elt) && (appartient elt ag)
      || (e<elt) && (appartient elt ad)
```

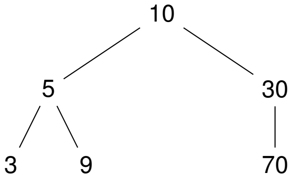
Un seul des deux sous-arbres est parcouru à chaque appel récursif

→ l'exécution de la fonction `appartient` ne nécessite plus de parcourir l'ensemble des noeuds de l'arbre

→ on peut montrer que si l'ABR `a` est équilibré :
nbre de comparaisons effectuées par `appartient` = $\log_2 |a|$

Une exécution de `appartient`

Cherchons l'élément 9 dans l'arbre suivant :



Parcours d’un ABR
Encore un algorithme de tri . . .

Etant donné un ABR, comment produire la **liste ordonné** de ses éléments ?
$\hookrightarrow$ parcours de l’arbre

Etant donné un ABR, comment produire la **liste ordonné** de ses éléments ?
↪ parcours de l'arbre

Lorsque l'on atteint le noeud `Noeud (elt, ag, ad)`, il y a plusieurs choix possibles pour poursuivre le parcours :

- placer `elt` dans la liste, puis parcourir `ag`, puis `ad`: parcours préfixé
- parcourir `ag`, placer `elt` dans la liste, parcourir `ad`: parcours infixé
- parcourir `ag`, puis `ad`, puis placer `elt` dans la liste: parcours postfixé

→ pour un ABR, le parcours infixé va produire une liste ordonnée :

```
let rec tri (a:'a abin):'a list=
 match a with
  | Vide → []
  | Node (elt, ag, ad) → (tri ag) @ (elt::(tri ad))
```

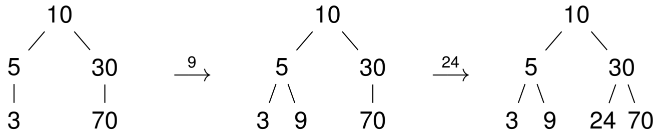
Insertion dans un ABR

Insertion en tant que feuille (le plus simple)

But: insérer un élément `elt` dans un ABR `a`

- préserver les propriétés de l'ABR
- insérer l'élément en tant que **nouvelle feuille**

Exemple : Insertion de deux éléments



Idée : distinguer deux cas

- `a` est vide, en insérant `elt` on obtient `Noeud` (`elt`, `Vide`, `Vide`)
- `a` est non vide, donc de la forme `Noeud` (`e`,`ag`,`ad`), alors
  - si `elt` <= `e`, alors `elt` doit être inséré dans `ag`
  - si `elt` > `e`, alors `elt` doit être inséré dans `ad`

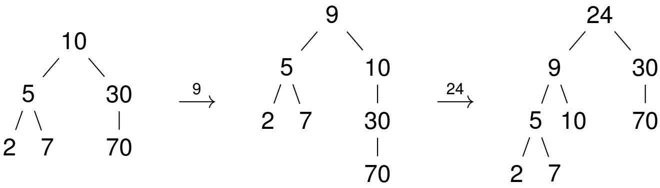
Insertion dans un ABR

Insertion en tant que racine

But: insérer un élément `elt` dans un ABR `a`

- préserver les propriétés de l'ABR
- inserérer l'élément comme **nouvelle racine** de `a`

Exemple : Insertion de deux éléments



Idée : procéder en 2 étapes

- "couper" l'arbre en deux ABR `ag` et `ad` tels que :
  - `g` contient tous les noeuds étiquetés par des éléments plus petits que `elt`
  - `d` contient tous les noeuds étiquetés par des éléments plus grands que `elt`
- construire l'ABR `Noeud` (`elt`,`ag`,`ad`)

ABR : Mise en oeuvre de l'insertion

Exercice : insertion en tant que feuille

Définir la fonction OCaml `insertion` qui insère un élément dans un ABR en tant que feuille

Exercice : insertion en tant que racine

Définir les fonctions :

- `partition` qui partitionne un ABR en 2 ABR par rapport à un élément
- `insertion` qui insère un élément dans un ABR en tant que racine, en utilisant `partition`

Exercice : création d'un ABR

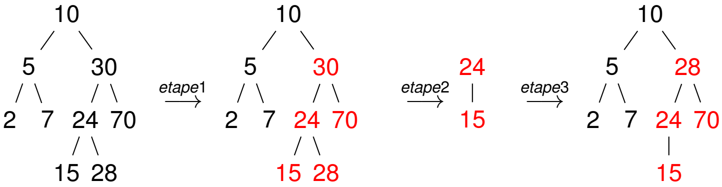
Définir deux fonctions `creation_abr` qui, étant donnée une liste d'éléments, crée un ABR contenant ces éléments en utilisant les deux méthodes d'insertion.

Supprimer un élément dans un ABR

Supprimer un élément `elt` d'un ABR consiste à :

1. Identifier le sous-arbre `Noeud` (`elt`, `ag`, `ad`) où la suppression doit avoir lieu
2. Supprimer le plus grand élément `max` de `ag` $\rightarrow$ On obtient un ABR `agprime`
3. Construire l'ABR `Noeud` (`max`,`agprime`, `ad`)

Exemple : Supprimer 30



ABR : Mise en oeuvre de la supression

Exercice : suppression dans un ABR

Définir les fonctions :

- `supp_max` qui supprime le plus grand élément d'un ABR et renvoie cet élément max et le nouvel ABR obtenu
- `suppression` qui supprime un élément dans un ABR

Conclusion (du chapitre sur les arbres)

- notion d'arbre :
  - représentation d'une relation de "hiérarchie" entre les éléments d'un type
  - nombreuses applications (recherche, tri, etc.)
- Type de données doublement récursif
- Deux classes importantes d'arbres :
  - les arbres binaires
  - les arbres binaires de recherche (ABR)
  - *il en existe beaucoup d'autres . . .*
- Exemples de fonctions sur les arbres :
  - recherche d'un élément
  - parcours (différents mode de parcours)
  - modification (insertion et suppression de noeuds)
  - ordre supérieur (équivalent de `map` et `fold` sur les listes)
  - etc.