Arbres

Image : Field background photo created by wirestock - www.freepik.com

I Arbres binaires

I.1 Définition inductive

Définition I.1 Un arbre binaire étiqueté par $\mathcal{E}$ est :
- soit vide, et on le note alors nil ou $\perp$
- soit un triplet (g, x, d) où $x \in \mathcal{E}$ et g et d sont des arbres binaires.

Remarque Cette définition inductive ressemble à la définition des listes.

On peut la formaliser en introduisant $T_b(\mathcal{E})$ l'ensemble des arbres étiquetés par $\mathcal{E}$. C'est le **plus petit ensemble** tel que :
- $\text{nil} \in T_b(\mathcal{E})$
- $\forall x \in \mathcal{E}, \forall g, d \in T_b(\mathcal{E}), (g, x, d) \in T_b(\mathcal{E})$

Les conséquences de la précision **plus petit ensemble** sont importantes :

- les arbres sont nécessairement des expressions finis, c'est-à-dire qu'il ne comportent qu'un nombre finis de constructions. C'est automatique car l'ensemble des arbres finis vérifie les conditions précédentes ;
- pour tout triplet (g, x, d) il ne peut exister qu'un arbre, sinon, en enlevant un arbre en double on vérifierait encore les deux conditions précédentes.

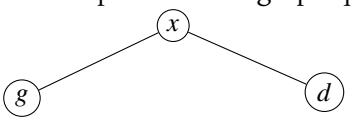
On en déduit directement la notion de preuve par **induction structurelle** sur les arbres :

$$\forall a \in T_b(\mathcal{E}), P(a) \iff \begin{cases} P(\text{nil}) \\ \forall x \in \mathcal{E}, \forall g, d \in T_b(\mathcal{E}), P(g) \wedge P(d) \Rightarrow P(g, x, d) \end{cases}$$

Cette induction correspond à un pseudo principe de récurrence qui nous permettra de faire les preuves.

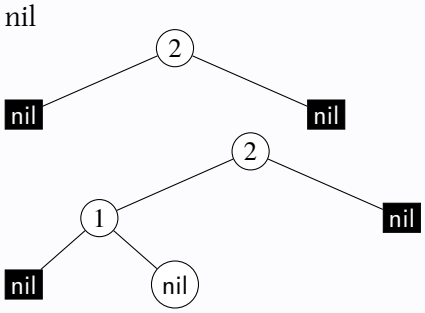
Pour démontrer cette inégalité, il suffit de vérifier que $T_P = \{t \in T_b(\mathcal{E}), P(t)\}$ vérifie les relations de la définition inductive et $T_P \subset T_b(\mathcal{E})$ et par minimalité, on a bien $T_P = T_b(\mathcal{E})$.

Exemple Si $\mathcal{E} = \mathbb{N}$, les éléments suivants sont des arbres :
- nil
- $(\text{nil}, 2, \text{nil})$
- $((\text{nil}, 1, \text{nil}), 2, \text{nil})$

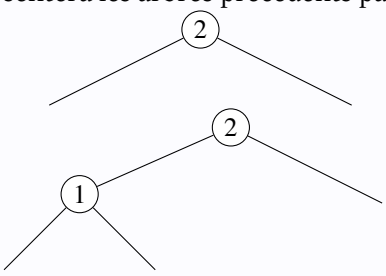
On adoptera une représentation graphique très naturelle pour les arbres binaires où un nœud (g, x, d) sera représenté par 

Exemple Les trois arbres précédentes sont donc représentés par :

nil



Afin d'alléger la notation, on omettra nil, sauf pour l'arbre vide. On fera cependant attention à conserver les arêtes donnant sur nil pour ne pas confondre $(\text{nil}, 1, \text{nil}(\text{nil}, 2, \text{nil}))$ et $((\text{nil}, 2, \text{nil}), 1, \text{nil})$. Ainsi on représentera les arbres précédents par :



I.2 Vocabulaire

Définition I.2 Soit a un arbre binaire. Les arbres non vides présents dans a sont appelés les **nœuds** de a. Parmi ceux-ci on distingue ceux qui sont de la forme (nil, x, nil). appelés des **feuilles.** Les nœuds qui ne sont pas des feuilles sont appelés des **nœuds internes**. Le nœud a lui-même est appelé la racine de l'arbre. On note $N(a)$ les nœuds de a.

Si $n \in N(a)$ n'est pas pas la racine, il est le fils gauche ou le fils droit d'un unique nœud appelé le **père** de x.

Si $n \in N(a)$, on appelle **sous-arbre** de a l'arbre dont n est la racine.

Remarque En partant de la définition inductive, il y a une identification entre un sous-arbre et sa racine. Mais afin de raisonner, on distinguera un nœud en tant qu'emplacement au sein d'un arbre et le sous-arbre

lui-même.

Définition I.3 Soit a un arbre binaire.
- On appelle **taille** de a, et on note $|a|$ le nombre de nœuds de a.
- On appelle **hauteur** de a l'entier

$$h(a) = \begin{cases} -1 & \text{si } a = \text{nil} \\ 1 + \max(h(g), h(d)) & \text{si } a = (g, x, d) \end{cases}$$

Définition I.4 Soit a un arbre binaire et $n \in N(a)$. On appelle **profondeur** de n l'unique entier $p(a)$ tel qu'il existe une suite finie $(n_0, n_1, \ldots, n_{p(a)})$ de n vérifiant :
- n_0 est la racine de a
- $n_{p(a)} = n$
- pour tout i, n_i est le père de n_{i+1}

Cette suite finie est le **chemin** de la racine à n. Il est nécessairement unique car chaque nœud autre que la racine a un unique père.

Théorème I.1 Soit a un arbre, si a est non vide, alors $h(a) = \max_{n \in N(a)} p(n)$.

■ **Preuve**

Par induction structurelle sur a.
- **Initialisation** Si $a = \text{nil}$ la prémisse est fausse, donc l'implication est trivialement vérifiée.
- **Hérédité** Supposons la propriété vérifiée pour deux arbres g et d, soit $x \in \mathcal{E}$, on va montrer qu'elle est vérifiée pour $a = (g, x, d)$. On a quatre cas pour le couple (g, d) :
  - ⋆ Soit $g \neq \text{nil}$ et $d \neq \text{nil}$. Dans ce cas, par hypothèse $h(g) = \max_{n \in N(g)} p_g(n)$ où p_g est la profondeur de n en tant que nœud de l'arbre g. Or, mis à part la racine de a, le chemin menant dans a au nœud n est dans g. On a donc directement $p_g(n) = p(n) - 1$. Ainsi $h(g) = \max_{n \in N(g)} p(n) - 1$. De même, $h(d) = max_{n \in N(d)} p(n) - 1$. On a

$$h(a) = 1 + \max(h(g), h(d)) = \max(\max_{n \in N(g)} p(n), \max_{n \in \mathbb{N}(d)} p(n))$$

    Or, le seul nœud de a qui n'est ni dans g ni dans d est sa racine, qui est de profondeur nulle donc $h(a) = \max_{n \in N(a)} p(n)$.
  - ⋆ Soit $g = \text{nil}$ et $d \neq \text{nil}$. Ainsi $h(g) = -1$ et donc $h(a) = 1 + h(d) = \max_{n \in N(d)} p(n)$ par l'analyse précédente. On conclut donc avec la propriété voulue.
  - ⋆ Soit $g \neq \text{nil}$ et $d = \text{nil}$. Cas symétrique du précédent.
  - ⋆ Soit $g = d = \text{nil}$. Auquel cas, $h(a) = 0$ qui est bien la profondeur de son unique nœud.

On a bien montré la propriété voulue par induction structurelle.

■

I.3 Implémentations

I.3.i En OCaml

En OCaml, on traduit directement la définition inductive par un type récursif :

OCaml
```
type 'a arbre = Noeud of 'a arbre * 'a * 'a arbre | Nil
```

On pourra alors définir des fonctions récursives sur les arbres par induction structurelle à l'aide d'un filtrage.

OCaml
```
let rec taille a = match a with
    | Nil -> 0
    | Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur a = match a with
    | Nil -> -1
    | Noeud(g, x, d) -> 1 + max (hauteur g) (hauteur d)
```

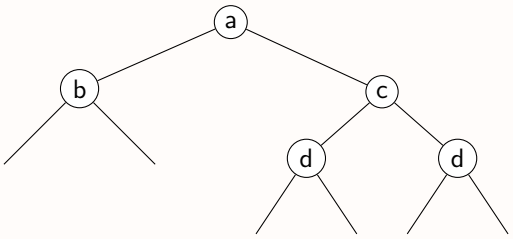
I.3.ii En C

I.4 Arbres binaires stricts

Définition I.5 Un arbre binaire est dit **strict** lorsqu'aucun de ses nœuds n'a qu'un fils vide.

Cela revient à dire que les seules les feuilles ont des fils vides. On peut donc omettre l'arbre vide et ne garder que les feuilles à la place.

Remarque L'arbre de gauche ici est un arbre binaire strict alors que ce n'est pas le cas de l'arbre de droite :



Théorème I.2 Soit a un arbre binaire strict non vide. Si a a n nœuds internes, alors il a $n+1$ feuilles.

■ **Preuve**

Par induction structurelle :

- **Initialisation** Si a a 0 nœud interne, c'est une feuille et on a directement la relation.
- **Hérédité** Supposons que la propriété soit vraie pour des arbres g et d, et soit $a = (g, x, d)$ un arbre dont ce sont les fils. On note $n_i(t)$ le nombre de nœuds internes et $n_f(t)$ le nombre de feuilles de l'arbre t. On a $n_i(a) = 1 + n_i(g) + n_i(d)$ et $n_f(a) = n_f(g) + n_f(d) = 2 + n_i(g) + n_i(d) = 1 + n_i(a)$. La propriété est démontrée pour a.

■

Cela permet naturellement de considérer des arbres où les feuilles et les nœuds internes ont deux types différents d'étiquettes.

L'exemple classique d'un tel arbre est celui des expressions arithmétiques :

- les nœuds internes sont étiquetés par des opérateurs binaires
- les feuilles sont étiquetées par des nombres.

En `OCaml`, on peut ainsi représenter un tel arbre par le type récursif :

OCaml
```
type ('a, 'b) arbre_bin = Feuille of 'a
    | Noeud of ('a, 'b) arbre_bin * 'b * ('a, 'b) arbre_bin
```

Remarque On ne peut plus représenter l'arbre vide avec ce type.

Et pour les expressions, on pourra ainsi définir un type pour les opérateurs et écrire :

OCaml
```
type operator = Plus | Times | Minus | Divides

let e = Noeud(Feuille 2, Plus, Noeud(Feuille 3, Mult, Feuille 4))
```

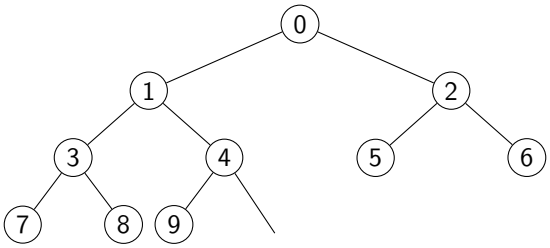
L'évaluation de telles expressions est étudiée dans le TP si dessous.

I.5 Arbres binaires complets

Définition I.6 Un arbre binaire dont tous les niveaux sont plein sauf éventuellement le dernier est dit **complet**. Si son dernier niveau est également plein, on dit qu'il est **parfait**.

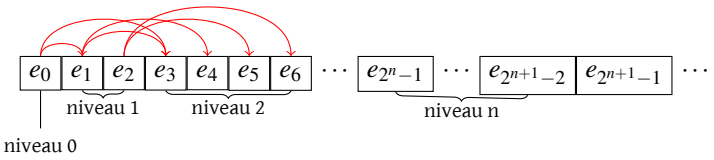
On peut représenter un arbre complet dans un tableau niveau par niveau en partant de la racine.

Si on place les indices comme étiquettes on aura, par exemple :



Et on pourra donc le représenter par le tableau des étiquettes dans cet ordre d'énumération.
On remarque que :
- le nœud 0 est toujours la racine
- si on considère un nœud i, son fils gauche est le nœud $2i+1$ et son fils droit est le nœud $2i+2$.
- le fils gauche de $i+1$ est $2(i+1)+1 = 2i+3 > 2i+2$. On est bien directement après.
- le $k$ème nœud du niveau l, en commençant à numéroter à 0, a pour indices $\sum_{j=0}^{l-1} 2^j + k = 2^l - 1 + k$.
- si $i > 0$, son père est le nœud $\lfloor \frac{i-1}{2} \rfloor$.

On peut donc, avec cette représentation, manipuler assez simplement un arbre binaire **à plat** .
Si le nœud de numéro i a l'étiquette e_i, le tableau a la structure suivante :



II Arbres

II.1 Définition

On étend directement la définition inductive des arbres binaires au cas où les nœuds plus d'éléments.

Définition II.1 Un arbre étiqueté par $\mathcal{E}$ est un couple (x, f) où $x \in \mathcal{E}$ et f une suite **finie** d'arbres.

On étend naturellement le vocabulaire des arbres binaires :
- le nœud de tête d'un arbre, en fin de compte sa *valeur*, est appelé sa racine
- les éléments de f sont appelés les fils du nœuds
- si f est vide, on dit que le nœud est une **feuille**, sinon, on dit que c'est un nœud interne.
- $|f|$, le nombre d'éléments de f, est appelé l'**arité** du nœud. Les feuilles sont donc les nœuds zéroaire.
- une suite finie d'arbres, ou plus généralement un ensemble d'arbres, est appelé une **fôret**.

Si on note $T(\mathcal{E})$ l'ensemble des arbres étiquetés par $\mathcal{E}$, on a

$$\forall x \in \mathcal{E}, \forall n \in \mathbb{N}, \forall a_1, \dots, a_n \in T(\mathcal{E}), (x, (a_1, \dots, a_n)) \in T(\mathcal{E})$$

Attention contrairement à ce que peut laisser entendre la formule précédente, il est tout à fait possible que $n = 0$ ce qui correspond à la suite finie vide et donc à une feuille. On perd ici la possibilité de représenter un arbre vide.

Remarque Un arbre binaire est un arbre dont les nœuds sont soit unaires soit binaires. On remarque en disant cela qu'on perd une information : la position gauche ou droite de l'unique fils d'un nœud unaire.

Comme pour les arbres binaires stricts, on pourra être amenés à considérer des arbres dont les nœuds internes et les feuilles sont étiquettés par des arbres

II.2 Implémentation

L'implémentation d'un arbre repose sur l'implémentation d'une suite finie. Celle-ci peut être faite par un tableau ou par une liste chaînée.

On retrouve alors des implementations différentes des arbres mais qui sont, au fond, très proches.

En `OCaml`, on pourra alors avoir les deux implémentations suivantes :

OCaml
```
type 'a arbre_l = { etiquette : 'a; enfants : 'a arbre_l list }
type 'a arbre_a = { etiquette : 'a; enfants : 'a arbre_a array }
```

On va alors avoir des programmes à la présentation assez différente suivant le type choisi. Voici, par exemple, les deux implémentations du calcul de la taille d'un arbre.

Avec des listes et aucune fonction du module `List`, on écrit souvent une fonction récursive sur les arbres et une fonction récursive sur les forêts.

OCaml
```
let rec taille_arbre a =
    1 + taille_foret a.enfants
    and taille_foret l = match l with
        | [] -> 0
        | t::q -> taille_arbre t + taille_foret q
```

On peut aussi utiliser directement `fold_left` et `map` :

OCaml
```
let rec taille_arbre a =
    1 + List.fold_left (+) 0 (List.map taille_arbre a.enfants)
```

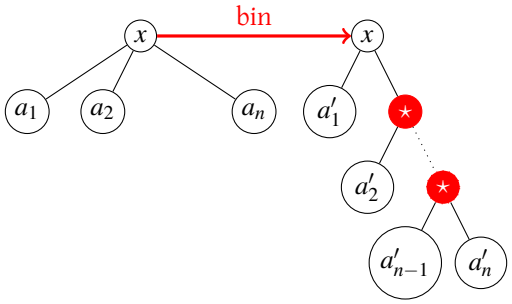
Avec l'autre représentation, on procédera avec un mélange de fonction récursive et de boucles. Cela n'est pas sans rappeler ce qui a été fait pour le backtracking.

OCaml
```
let rec taille_arbre a =
    let t = ref 1 in
    for i = 0 to Array.length a.enfants - 1 do
        t := !t + taille_arbre a.enfants.(i)
    done;
    !t
```

II.3 Représentation par un arbre binaire

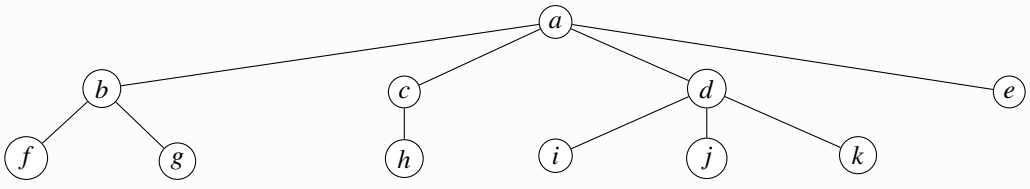
Pour des raisons d'efficacité ou de réutilisation de programmes existants, on peut vouloir représenter un arbre comme un arbre binaire.

On va définir une application : bin : $T(\mathcal{E}) \to T_b(\mathcal{E} \cup \{\star\})$ ainsi : $\text{bin}(x, ()) = (\text{nil}, x, \text{nil})$, $\text{bin}(x, (a)) = (\text{bin}(a), x, \text{nil})$ et pour un nœud d'arité strictement plus grande que 1 :

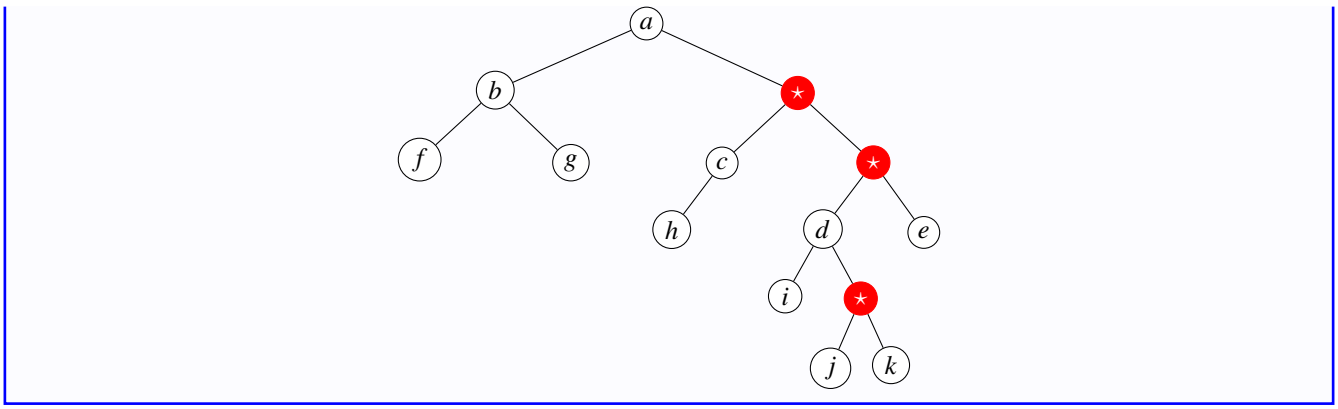


où $a'_i = \text{bin}(a_i)$. Les nœuds rouges sont des nœuds ayant une étiquette spéciale $\star$ telle que $\star \notin \mathcal{E}$.

Exemple L'arbre suivant :



sera alors représenté par l'arbre binaire :



II.4 Applications

II.4.i Arbres d'expressions

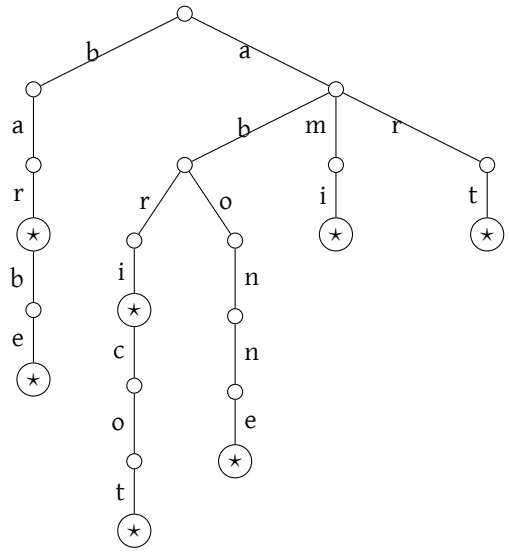
On a déjà vu l'utilisation des arbres binaires pour représenter des expressions arithmétiques avec des opérateurs unaires ou binaires. On généralise naturellement cela à des expressions dont les opérateurs sont d'arité quelconque.

C'est ainsi qu'on représente un programme en mémoire, on parle d'arbre syntaxique.

II.4.ii Arbres préfixes ou tries

Un *arbre préfixe* ou *trie* est un arbre permettant de représenter un ensemble de mots. On étiquette les arêtes par des caractères et les nœuds par un booléen indiquant si la suite des étiquettes qui mène de la racine à ce nœud est un mot.

Par exemple, le trie :



permet de représenter l'ensemble de mots : bar, barbe, art, ami, abri, abricot, abonne.

Afin de représenter un tel arbre, il est nécessaire d'avoir une étiquette sur les arêtes, le plus simple pour cela est de remplacer la liste des enfants par une liste de couples (étiquette, enfant) ainsi :

OCaml
```
type trie = {
    mot : bool;
    enfants : (char * trie) list
    }
```

Les fonctions de manipulations de ce type de donnée sont étudiées dans le TP **TODO**.

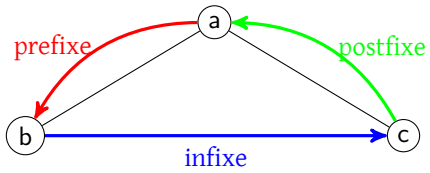
III Parcours

III.1 Parcours récursif d'un arbre binaire

On a déjà vu dans les paragraphes précédent ce qui constitue le cœur du parcours d'un arbre :

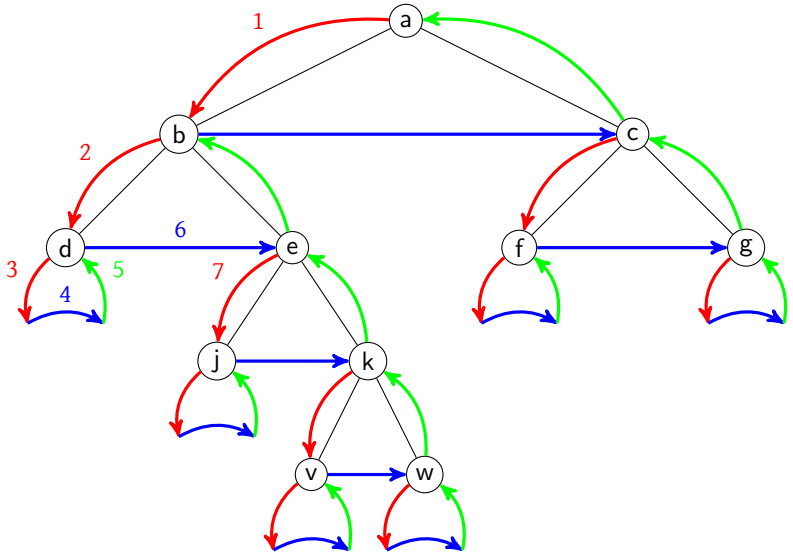
OCaml
```
let rec parcours a =
    match a with
    | Nil -> ()
    | Noeud(g, x, d) ->
        parcours g;
        parcours d
```

Cette fonction ne fait rien, mais elle va parcourir chacun des nœuds de l'arbre. Si on veut effectuer un traitement sur le nœud, on peut le faire à trois moments :



- Avant l'appel à `parcours g`, on parle de *traitement préfixe*
- Entre les deux appels, on parle de *traitement infixe*
- Après l'appel à `parcours g`, on parle de *traitement postfixe*

On représente ici l'ordre dans lequel on va effectuer chacun de ces trois traitements selon le code couleur : rouge pour préfixe, bleu pour infixe et vert pour postfixe. On a également indiqué, pour les premiers traitements, l'ordre dans lequel ils sont effectués par des numéros.



Pour illustrer ces traitements, on peut rajouter trois arguments au parcours :

OCaml
```
let rec parcours prefixe infixe postfixe a =
    match a with
    | Nil -> ()
    | Noeud(g, x, d) ->
        prefixe a;
        parcours g;
        infixe a;
        parcours d;
        postfixe a
```

On peut, par exemple, définir les deux fonctions suivante :

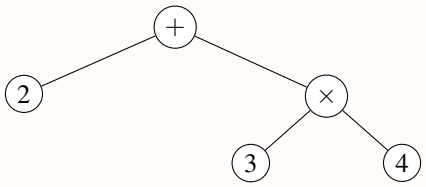
OCaml
```
let idle a = () (* ne fait rien *)

let print a = match a with
    | Nil -> failwith "Vide"
    | Noeud(_,x,_) -> print_char a
```

On peut alors observer les nœuds affichés sur l'arbre précédent selon les traitements effectués :

- `parcours print idle idle a` : **abdejkvwcfg** on suit uniquement les arêtes rouges

- `parcours idle print idle a` : *dbejvkwafcg* on suit uniquement les arêtes bleues
- `parcours idle idle print a` : djvwkebfgca on suit uniquement les arêtes vertes
- `parcours print print print a` : **abd***ddb***ejjje*k**v***vvk***w***wwkeb*...* on suit toutes les arêtes

Remarque Si on considère un arbre d'expression comme :



On va alors obtenir en affichant à chaque traitement la suite : $+222+\times333\times444\times+$. Chaque nombre n étant une feuille, on va observer l'affichage de nnn qu'on peut réduire à n. Pour les opérateurs, si on remplace le premier affichage par (et le dernier par). En procédant ainsi, on affiche $(2+(3\times4))$ et on ainsi retrouvé l'écriture bien parenthésée de l'expression.

Définition III.1 L'ordre dans lequel on effectue un traitement *préfixe* dans un parcours sur les nœuds d'un arbre est appelé l'ordre *préfixe* sur les nœuds.

On définit, de même, l'ordre *infixe* et l'ordre *postfixe*.

III.2 Parcours impératifs

III.2.i Cadre général

On va condisérer une structure de donnée abstraite qui généralise les piles et les files et permettre de représenter un ensemble de tâches à traiter. Pour cela, on dispose d'un type paramétrique `'a t` et de l'interface :

- `cree : unit -> 'a t` crée un ensemble de tâches
- `ajoute : 'a t -> 'a -> unit` ajoute une nouvelle tâche à traiter
- `retire : 'a t -> 'a` retire une tâche de l'ensemble des tâches à traiter
- `est_vide : 'a t -> bool` renvoie un booléen indiquant si l'ensemble de tâches est vide

On a ainsi vu deux possibles implémentations, qui sont elles-mêmes des structures abstraites mais un peu moins abstraites que celle-ci :

- les piles pour lesquelles on retire le dernier élément ajouté
- les files pour lesquelles on retire l'élément le plus anciennement ajouté

On écrire un parcours impératif générique :

OCaml
```
let parcours traitement a =
    let avisiter = cree () in
    ajoute avisiter a;
    while not (est_vide avisiter) do
        let a = retire avisiter in
        match a with
        | Nil -> ()
        | Noeud(g, _, d) -> traitement a;
                            ajoute avisiter g;
                            ajoute avisiter d
    done
```

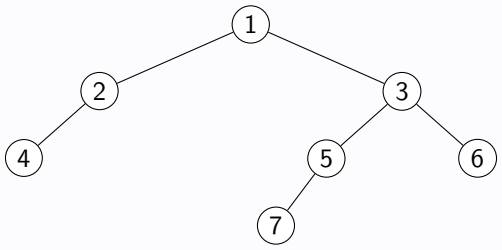
III.2.ii Cas du parcours en profondeur

Dans le cas où on utilise une pile, on retrouve essentiellement le parcours récursif précédent avec quelques changements :

- on empile ici d'abord `g` puis `d`, donc on va à droite avant d'aller à gauche. Il suffit de permuter les deux ajouts pour retrouver l'ordre précédent
- on effectue un unique traitement préfixe
- si on considère la pile d'appels récursifs du parcours récursif, on constate que le nombre de structure pile est majoré par la hauteur de l'arbre. Ici, on va potentiellement ajouter tous les nœuds sur la pile, donc, on a une complexité en espace en $O(|a|)$ plutôt qu'en $O(h(a))$.

On parle de **parcours en profondeur** ou depth-first search (DFS) en anglais.

Exemple Si on considère l'arbre suivant :



En ignorant les arbres vides, par exemple en ne les ajoutant pas, on va avoir le déroulement suivant du parcours :

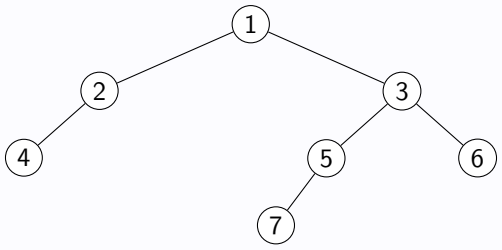
avisiter	etiquette traitée
$1 \rightarrow$	
$2, 3 \rightarrow$	1
$2, 5, 6 \rightarrow$	3
$2, 5 \rightarrow$	6
$2, 7 \rightarrow$	5
$2 \rightarrow$	7
$4 \rightarrow$	2
	4

III.2.iii Cas du parcours en largeur

Dans le cas où on utilise une file, on obtient un parcours appelé le **parcours en largeur** en breadth-first search (BFS) en anglais.

Dans ce parcours, on va visiter les nœuds *niveau par niveau* en partant de la racine jusqu'aux feuilles de plus grande profondeur.

Exemple Si on considère l'arbre suivant :



En ignorant les arbres vides, par exemple en ne les ajoutant pas, on va avoir le déroulement suivant du parcours :

avisiter	etiquette traitée
$\rightarrow 1 \rightarrow$	
$\rightarrow 3, 2 \rightarrow$	1
$\rightarrow 4, 3 \rightarrow$	2
$\rightarrow 6, 5, 4 \rightarrow$	3
$\rightarrow 6, 5 \rightarrow$	4
$\rightarrow 7, 6 \rightarrow$	5
$\rightarrow 7 \rightarrow$	6
	7

Ce parcours est particulièrement intéressant quand on cherche une information avec la plus petite profondeur possible. Par exemple, dans le cas de la recherche d'une solution à un problème, on peut vouloir tester les petites solutions avant les plus grandes.

Pour des problèmes d'énumération, c'est également intéressant, car on va obtenir des éléments par ordre

croissant de la longueur du chemin depuis la racine. Par exemple, pour les tries, on en déduit les mots par ordre croissant de longueur.

III.3 Parcours d'arbres

Pour des arbres, on va suivre les mêmes principes. La différence va se situer au niveau de l'implémentation car il faudra alors manipuler des listes ou des tableaux d'enfants.

Notons qu'on a déjà vu un tel problème quand on a résolu des problèmes par backtracking. En effet, avec le backtracking, on a un arbre implicite des positions partielles dont les enfants sont les mouvements possibles vers de nouvelles positions.

IV Arbres binaires de recherche

IV.1 Objectif

On souhaite ici réaliser une structure efficace d'ensembles finis. Pour cela, on cherche à définir une structure immuable `'a ensemble` muni de quatre opérations

- `ensemble_vide : 'a ensemble` l'ensemble vide
- `ajoute : 'a ensemble -> 'a -> 'a ensemble` rajoute un élément à un ensemble et renvoie le nouvel ensemble
- `supprime : 'a ensemble -> 'a -> 'a ensemble` retire un élément à un ensemble et renvoie le nouvel ensemble
- `contient : 'a ensemble -> 'a -> bool` teste si l'ensemble contient un élément
- `cardinal : 'a ensemble -> int` renvoie le nombre d'éléments de l'ensemble

Notons qu'on pourra souvent relacher la contrainte naturelle des ensembles en permettant d'ajouter plusieurs fois un même élément.

Une implémentation possible de cette structure serait d'utiliser des listes ou des tableaux triés, on aurait alors des opérations en $O(n)$ pour manipuler n éléments ($O(log_2 n)$ en optimisant la recherche avec une recherche dichotomique).

Ici, on présente une implémentation rendant possible une complexité en $O(log_2 n)$ pour chaque opération sous certaines hypothèses dont on montrera qu'elles peuvent être satisfaites dans un second temps.

IV.2 Définition

Définition IV.1 On dit qu'un arbre a étiquetté par X, muni d'une relation d'ordre total, est un **arbre binaire de recherche** (abr) lorsque :

- soit $a = \text{nil}$
- soit $a = (g, x, d)$ où g et d sont des arbres binaires de recherches et de plus

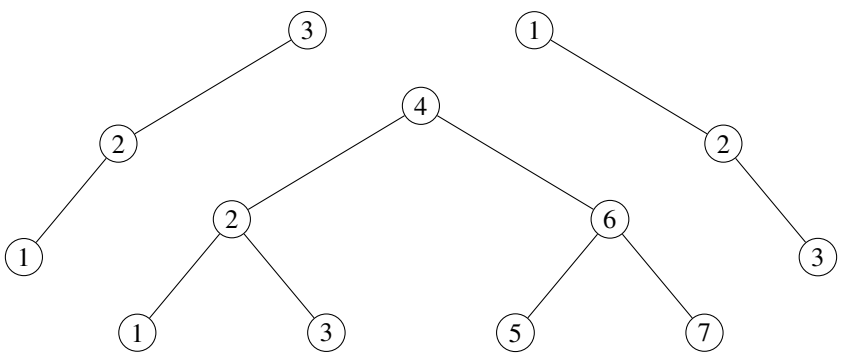
$$\max_{y \in g} e(y) \leq x \leq \min_{y \in d} e(d)$$

où $e(y)$ est l'étiquette du nœud y.

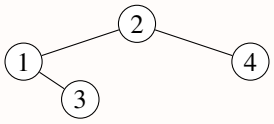
Remarque

- Tout sous-arbre d'un arbre binaire de recherche est un arbre binaire de recherche.
- Pour un ensemble donné d'étiquettes il n'y a pas unicité de l'arbre binaire de recherche : voir ci-après.

Exemples d'abr :



Remarque La propriété caractéristique des ABR n'est pas locale c'est-à-dire tester si chaque nœud a une étiquette supérieure à son fils gauche et inférieure à son fils droit ne suffit pas.



IV.3 Opérations

IV.3.i Minimum et maximum

On peut obtenir la valeur minimale et maximale d'un abr en effectuant un parcours le long de la branche la plus à gauche ou de la branche la plus à droite.

En effet, si l'arbre est (g, x, d) : soit g est vide et $x \leq \min d$ est le minimum, soit g est non vide et on a alors $\min g \leq \max g \leq x$ donc le minimum de g est celui de l'arbre.

On en déduit le programme suivant :

OCaml
```
let rec minimum a =
    match a with
    | Nil -> failwith "vide"
    | Noeud(Nil, x, _) -> x
    | Noeud(g, _, _) -> minimum g
```

On remarque que pour parcourir cette branche, on est en $O(h(a))$ où a est l'arbre considéré.

!susubsection(Test si un arbre est un abr) Pour tester si un arbre est bien un arbre binaire de recherche on doit connaitre la valeur maximum de l'arbre gauche et minimum de l'arbre droit et tester si l'étiquette de chaque nœud n vérifie

$$\max_{\eta \in A_g} e(\eta) \leq e(n) \leq \min_{\eta \in A_d} e(\eta)$$

On peut écrire une procédure qui indique si l'arbre est un arbre binaire de recherche et les valeurs maximum et minimum de l'arbre (puisqu'on doit les calculer...)

OCaml
```
let rec abr_test a = match a with
| Nil -> failwith "Arbre vide"
| Noeud(Nil,n,Nil) -> (true, n, n)
| Noeud(Nil,n,d) -> let bd, md, Md = abr_test d in
    (bd && n < md, min n md, max n Md)
| Noeud(g,n,Nil) -> let bg, mg, Mg = abr_test g in
    (bg && n >= Mg, min n mg, max n Mg)
| Noeud(g,n,d) ->
    let bg, mg, Mg = abr_test g in
    let bd, md, Md = abr_test d in
    (bg && n >= Mg && n < md, min n (min mg md), max n (max Mg Md))
```

Ici on a fait le choix de ne rien associer à l'arbre vide et de le traiter en amont au niveau des nœuds. Cela permet de ne pas imposer de contraintes sur le type des étiquettes comme cela aurait été le cas en mettant des valeurs ad hoc dans le cas de l'arbre vide (typiquement $\pm\infty$).

Remarque On rajoute des fonctions permettant de manipuler facilement ces données et on obtient alors un algorithme de test plus concis :

```OCaml
let valeur a = match a with
    | None -> failwith "None n'a pas de valeur"
    | Some x -> x

let opt_bin f a b = match a, b with
    | None, _ -> b
    | _, None -> a
    | Some a, Some b -> Some (f a b)

let omax = opt_bin max
let omin = opt_bin min

let abr_test a =
    let rec aux a = match a with
        | Nil -> (true, None, None)
        | Noeud(g,n,d) ->
            let bg, mg, Mg = aux g in
            let bd, md, Md = aux d in
            (bg && (Mg = None || n >= valeur Mg)
                && (md = None || n < valeur md),
            omin (Some n) (omin mg md),
            omax (Some n) (omax Mg Md))
    in let est_abr, min_abr, max_abr = aux a in
    (est_abr, valeur min_abr, valeur max_abr)
```

On verra en exercice qu'une lecture infixe permet de conclure également.

IV.3.ii Recherche d'un élément

Comme on leur nom l'indique, la recherche est adaptée à ce type d'arbres. Le modèle par adjonction s'applique et donc la hauteur moyenne d'un arbre binaire de recherche est un $\bigcirc(\log_2(n))$.On peut construire la fonction de recherche :

```OCaml
let rec recherche x a = match a with
  |Nil -> false
  |Noeud(gauche,n,droite) when x=n -> true
  |Noeud(gauche,n,droite) when x>n -> recherche x droite
  |Noeud(gauche,n,droite)  -> recherche x gauche
```

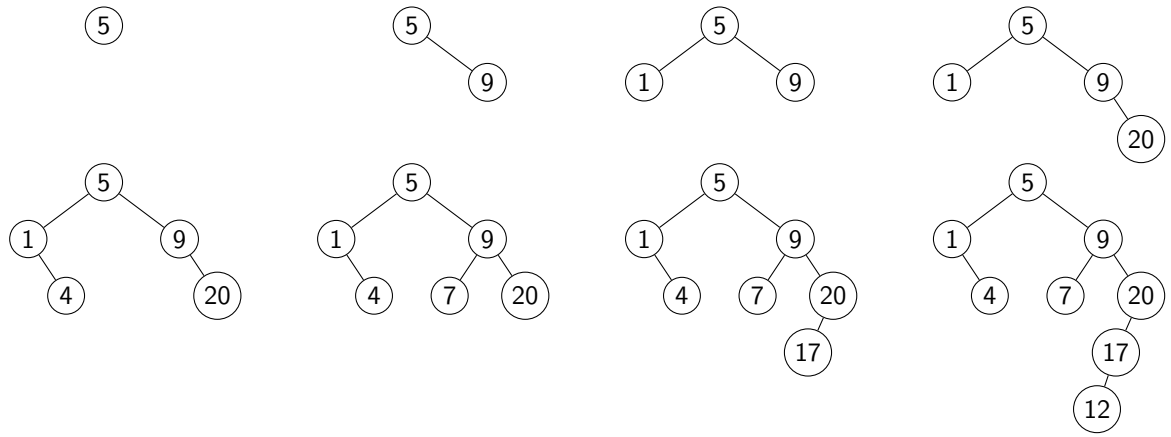
IV.3.iii Insertion d'un élément

Étant donné un arbre binaire de recherche a et une étiquette x, on veut construire un nouvel arbre binaire de recherche contenant les étiquettes des nœuds de a et x. Il n'y a bien sur pas unicité de la solution... Dans un arbre binaire de recherche, l'opération se décompose en une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément, suivie de l'adjonction proprement dite.

```OCaml
let rec insere x a = match a with
  | Nil -> Noeud(Nil,x,Nil)
  | Noeud(gauche,n,droite) when x>n ->  N(gauche,n,insere x droite)
  | Noeud(gauche,n,droite) -> N(insere x gauche,n,droite)
```

On compare x à la racine pour déterminer s'il faut l'ajouter dans le sous-arbre gauche ou le sous-arbre droit, et l'on rappelle la procédure récursivement. Le dernier appel récursif se fait sur un arbre vide et l'on crée alors à cette place le nœud d'étiquette x. Le nouvel arbre est reconstruit de proche en proche depuis cette branche vers la racine, sans modifier les branches non explorées. Les anciens nœuds ne sont pas modifiés et l'ancien arbre reste donc disponible.

Construction par insertion successive de l'ABR pour les éléments 5, 9, 1, 20 , 4, 7 , 17 , 12 :



IV.3.iv Suppression d'un élément

Le problème de la suppression d'un élément (ou plutôt de sa première occurrence) d'un arbre binaire de recherche est plus compliqué. On peut le décomposer en :

- Recherche de l'élément
- Suppression
- Recomposition de l'arbre

Le nœud à supprimer est la racine d'un sous-arbre de l'arbre binaire de recherche. Commençons par considérer le problème de la suppression de la racine z d'un arbre binaire de recherche.

- Si les deux branches issues de z sont vides, on supprime et c'est fini (cas (a) de la figure $z = 13$).
- Si l'une des deux branches issues z est vide, on se contente de garder l'autre branche. On conserve ainsi un arbre binaire de recherche (cas (b) de la figure $z = 16$).
- Si les deux branches sont non vides, soit y l'élément le plus à gauche du sous-arbre droit A_d de z. On a $y > z$ par la propriété des ABR.

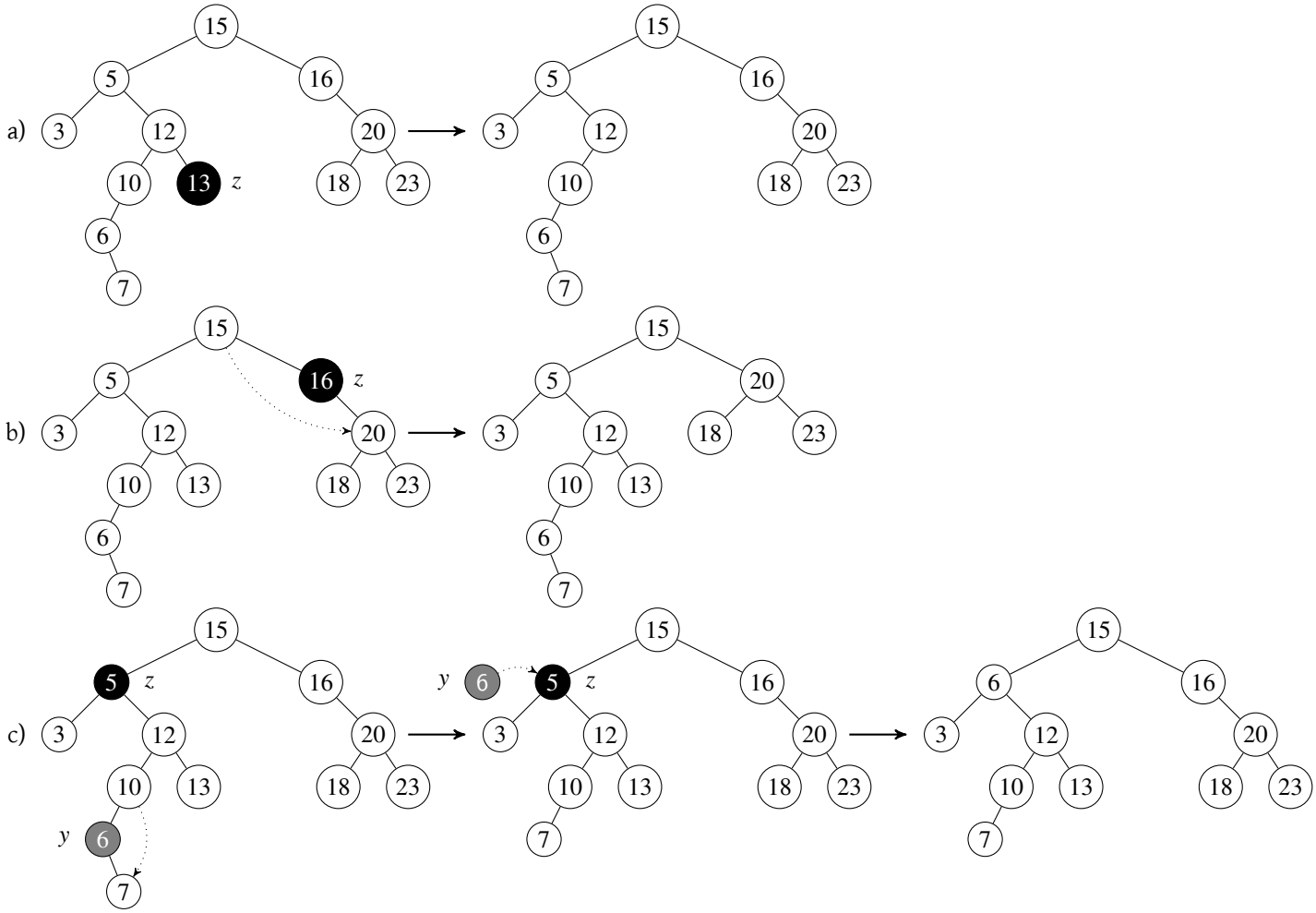
Remarquons que y ne peut avoir de fils gauche car sinon il ne serait pas le plus à gauche de A_d.

On peut donc le supprimer et le reporter en lieu et place de z : on obtient alors un nouvel arbre qui est un arbre binaire de recherche. En effet :

- La branche droite est un arbre binaire de recherche (on a supprimé un nœud sans fils gauche) et l'étiquette de la racine est, par construction, inférieure aux étiquettes des nœuds de cette branche.
- La branche gauche est un arbre binaire de recherche (on n'y a pas touché) et la nouvelle étiquette de la racine y est supérieure ou égale à z qui majorait les étiquettes des nœuds de la branche gauche.

(cas (c) de la figure $z = 5$ et $y = 6$). La localisation de y est facile : c'est nécessairement l'élément le plus à gauche de A_d.

Les trois cas de suppression :



On en déduit le programme suivant :

OCaml
```
let rec supprime a x =
    match a with
    | Nil -> Nil
    | Noeud(g, y, d) when y < x ->
        Noeud(g, y, supprime d x)
    | Noeud(g, y, d) when y > x ->
        Noeud(supprime g x, y, d)
    (* à partir d'ici l'etiquette est forcement x *)
    | Noeud(Nil, _, Nil) -> Nil
    | Noeud(Nil, _, d) -> d
    | Noeud(g, _, Nil) -> g
    | Noeud(g, x, d) ->
        let y = minimum d in
        Noeud(g, y, supprime d y)
```

IV.4 Équilibrage

IV.4.i Principe et approche naïve

Les opérations précédentes sont toutes en $O(h(a))$, or on sait que $h(a) \leq |a| \leq 2^{h(a)} - 1$ donc $\log_2(|a|+1) \leq h(a) \leq |a|$. On aimerait se rapprocher au plus de $\log_2 |a|$ et pour cela, il faut que l'arbre soit le plus proche possible d'un arbre parfait. On parle alors d'équilibrage.

Une approche naïve consiste à extraire les valeurs dans l'ordre croissant à l'aide d'un parcours infixe puis à reconstruire un arbre équilibré par dichotomie : on place la valeur médiane à la racine et on construit récursivement les sous-arbres gauche et droite.

IV.4.ii Arbres 2-3

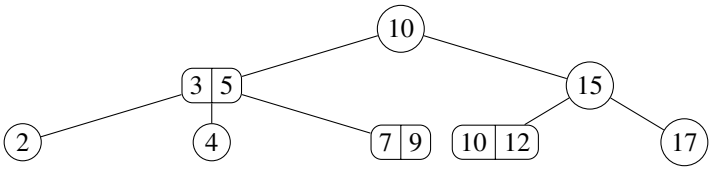
Définition IV.2 Un arbre 2-3 est un arbre étiqueté contenant des nœuds de deux types :

- des nœuds binaires $N(g, x, d)$ où g, d sont des arbres et x une étiquette tel que $\max g \leq x \leq \min d$ (c'est la même condition que les ABR);

- des nœuds ternaires $M(g, x, m, y, d)$ où g, m, d sont des arbres 2-3 et x, y sont des étiquettes telles que : $\max g \leq x \leq \min m \leq \max m \leq y \leq \min d$.

et parfaitement équilibré : tous les chemins de la racine à une feuille ont même longueur.

Un nœud ternaire contient donc un fils central dont les valeurs sont comprises entre ses deux étiquettes.
Voici un exemple d'arbre 2-3 :



Théorème IV.1 Un arbre 2-3 à n nœuds est de hauteur au plus $\lfloor \log_2 n \rfloor$.

■ **Preuve**

Les nœuds ternaires rendent l'arbre compact. Le maximum de hauteur correspond donc au cas où il ne contient que des nœuds binaires sauf sur son dernier niveau, un tel arbre est bien de hauteur $\lfloor \log_2 n \rfloor$.

■

Recherche

Pour rechercher un élément x on adapte l'algorithme des ABRs :

- pour un noeud binaire $N(g, y, d)$
  - ⋆ si $x < y$ on cherche dans g ;
  - ⋆ si $x = y$ on a trouvé le nœud voulu ;
  - ⋆ si $x > y$ on cherche dans d.
- pour un noeud ternaire $M(g, y, m, z, d)$
  - ⋆ si $x < y$ on cherche dans g ;
  - ⋆ si $x = y$ ou $x = z$ on a trouvé le nœud voulu ;
  - ⋆ si $z > x > y$ on cherche dans m ;
  - ⋆ si $z < x$ on cherche dans d.

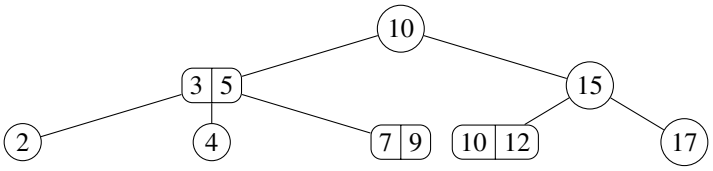
La complexité de cet algorithme est donc encore en $O(h)$ où h est la hauteur de l'arbre. En tenant compte de la remarque précédente, elle est en $O(\log n)$.

Insertion

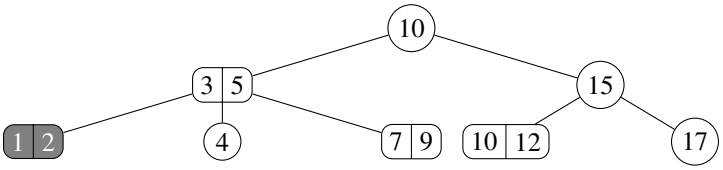
Pour insérer un élément x dans un arbre 2-3, on va temporairement enrichir sa structure par des nœuds quaternaires satisfaisant les même requêtes sur les clés que les nœuds binaires et ternaires.

Pour insérer on effectue alors une recherche qui nous amène au dernier nœud avant les feuilles. On transforme alors ce nœud en augmentant son arité et en plaçant x à la bonne position par rapport aux étiquettes qu'il contient.

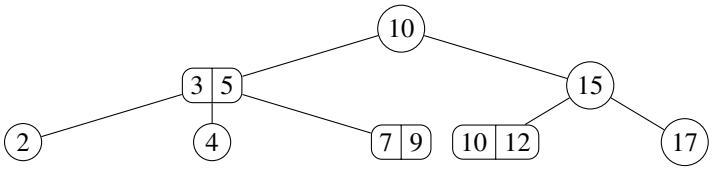
Exemples : * 1 dans



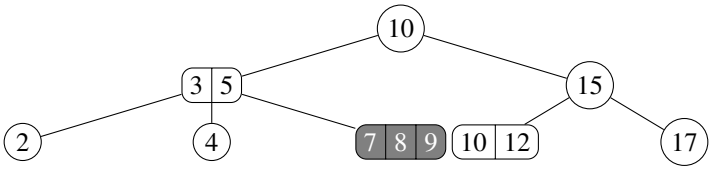
donne



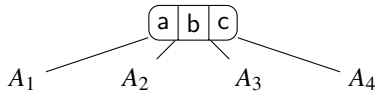
- 8 dans



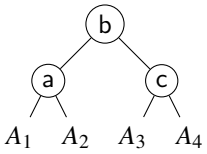
donne



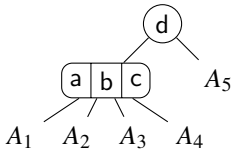
Maintenant on va supprimer les nœuds quaternaires en les faisant remonter, au pire, jusqu'à la racine pour laquelle on peut faire l'opération suivante :



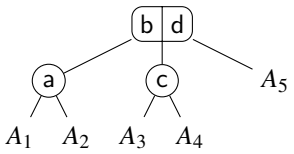
→



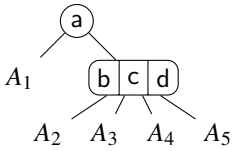
Voici tous les cas de transformations selon le type du père et la position du nœud quaternaire : * fils gauche d'un binaire :



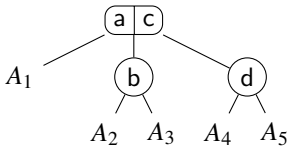
→



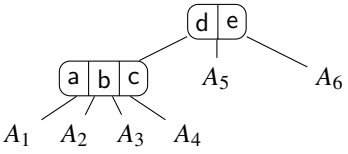
- fils droit d'un binaire :



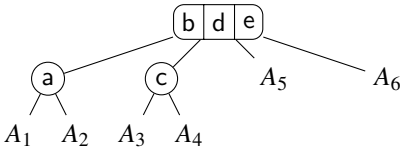
→



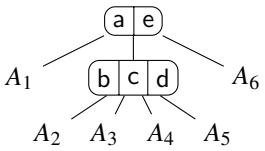
- fils gauche d'un ternaire :



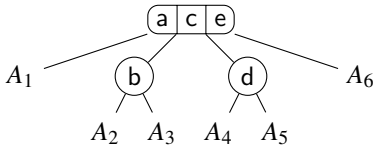
→



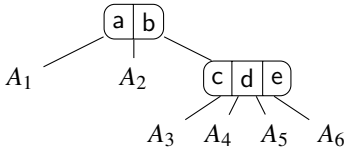
- fils central d'un ternaire :



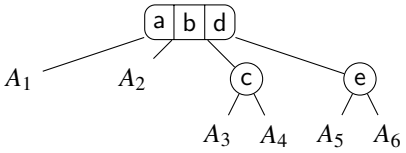
$\rightarrow$



- fils droit d'un ternaire :



$\rightarrow$



On observe rapidement que toutes ces transformations préservent la position des sous-arbres vis-à-vis des étiquettes des nœuds. Ainsi, au final on aura obtenu un arbre 2-3.

Remarque sur l'implémentation

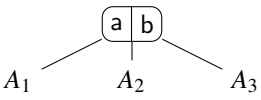
Les arbres 2-3 sont simples à comprendre mais complexes à programmer :

- beaucoup de cas à traiter, notamment en fonction des parents ;
- des nœuds intermédiaires 2-3-4 qui demandent d'avoir un type de travail et un type final.

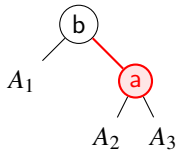
On va voir dans un dernier temps des arbres quasiment équivalent à ceux-ci mais bien plus simples à programmer.

IV.4.iii Arbres rouges et noirs

On va transformer un arbre 2-3 en ABR en transformant les nœuds ternaires ainsi :



$\rightarrow$



Afin de se souvenir que le nœud a provient de cette séparation, on le marque en rouge. Ici on a également marqué l'arête gauche en rouge mais on l'omettra dans la suite.

Quelles remarques :

- la racine est noire et les feuilles sont noires ;
- le fils gauche d'un nœud est toujours noir ;
- il ne peut exister d'arêtes liant deux nœuds rouges car par construction les nœuds rouges sont toujours entourés de nœuds noirs ;
- les chemins d'une feuille à la racine dans l'arbre 2-3 sont tous de même taille, à chacun des nœuds de l'arbre 2-3 correspond un nœud noir, on en déduit donc que tous les chemins de la racine aux feuilles contiennent le même nombre de nœud noirs.

Réciproquement un arbre ABR dont les nœuds peuvent avoir ces deux couleurs et qui vérifie ces propriétés est appelé un arbre rouge et noir.

En fait, on a une bijection entre les arbres 2-3 et les arbres rouges et noirs ainsi définis.

La proposition suivante permet de voir que les arbres rouges et noirs sont de bons ABRs :

Théorème IV.2 La hauteur d'un arbre rouge et noir est au maximum égale au double de la hauteur de l'arbre 2-3 qui lui est associé.

Ainsi tout arbre rouge et noir de taille n est de hauteur $O(\log n)$.

■ **Preuve**

La hauteur n'augmente que par séparation des nœuds ternaires. Or, il ne peut il y en avoir plus que la hauteur de l'arbre 2-3.

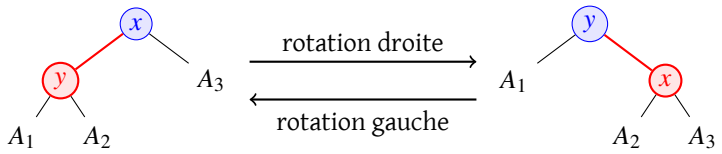
■

Pour l'insertion d'un nœud, on va déstabiliser la coloration en mettant trop de nœuds rouges, puis on va rétablir les contraintes en remontant vers la racine. Cela rappelle l'élimination des nœuds quartenaires des arbres 2-3. Mettre des nœuds rouges à l'avantage de préserver l'equilibrage noir.

Pour rétablir les couleurs, on va alors utiliser une succession de deux opérations élémentaires : les rotations et la bascule.

Rotations

On définit les deux rotations gauche et droite, duales l'une de l'autre, ainsi :

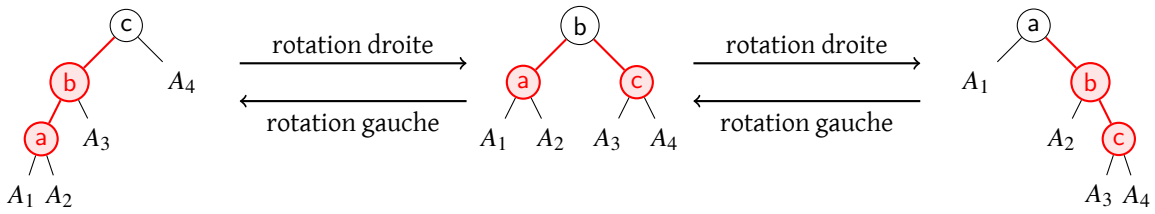


Ici, on a utilisé la couleur bleue pour représenter la couleur du nœud de départ qui peut être soit rouge soit noire mais va être préservée.

Les rotations permettent donc de faire pencher des nœuds rouges à droite ou à gauche.

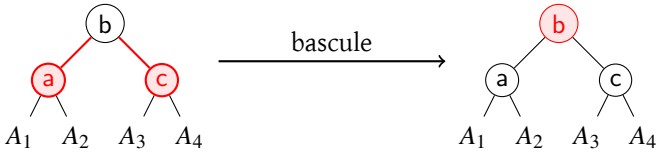
On remarque que la hauteur en nœuds noirs de l'arbre n'a pas changé.

On peut passer d'une des trois représentations convenables des nœuds quaternaires à une autre à l'aide de rotations :



Bascule

Une autre opération importante dans les arbres rouges et noirs est le bascule qui permet de faire remonter les défauts de coloration vers la racine tout en préservant l'équilibre en nœuds noirs :



Si on effectue une bascule directement sur la racine de l'arbre consideré, on finit par la noircir ce qui augmente de un la profondeur noire de chaque feuille mais ne change pas l'équilibrage global.

Insertion

Selon les remarque précédente, on ne va pas hésiter à perdre la propriété des arbres rouges et noirs temporairement. De toutes les propriétés de ces arbres, c'est surement le fait qu'ils soient parfaitement équilibrés en nœuds noirs qui est le plus dur à assurer. On ne va donc pas toucher à cette propriété : on va insérer des nœuds rouges sous une feuille.

En faisant cela on risque d'avoir deux types de problèmes :

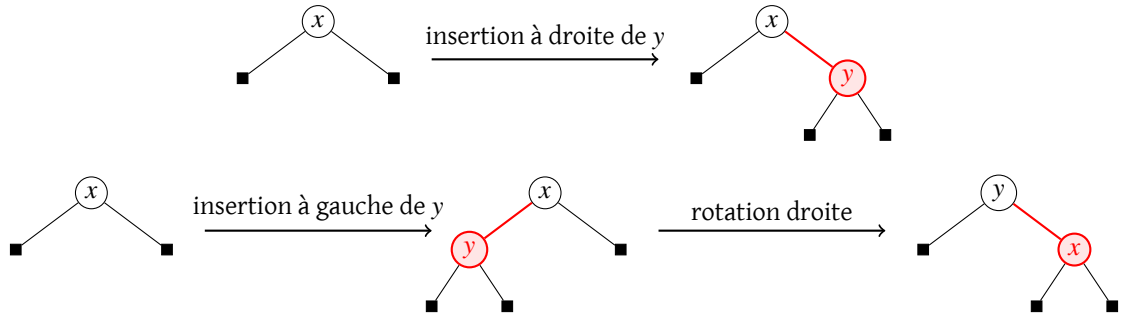
- des arêtes rouge-rouge ;
- des fils gauches rouges.

On va alors traiter les différents cas d'insertion et voir comment on peut les ajuster pour obtenir des arbres rouges et noirs.

Insertion dans un nœud issu d'un nœud binaire

Un nœud noir issu d'un nœud binaire a nécessairement ses deux fils qui sont vides ou non vides simultanément, car ils proviennent directement de la structure parfaitement équilibré des arbres 2-3.

On a alors les deux cas suivants d'insertion :

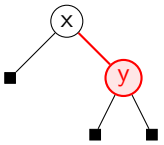


Dans les deux cas on se ramène à un nœud ternaire directement.

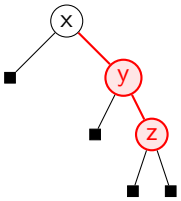
Insertion dans un nœud issu d'un nœud ternaire

On a trois cas d'insertions selon la position de la nouvelle valeur par rapport aux nœuds deux valeurs du nœud ternaire.

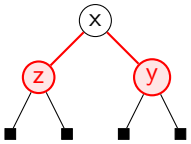
Supposons que l'on insère z dans l'arbre :



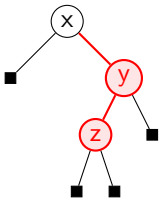
- Si $x \leq y \leq z$:



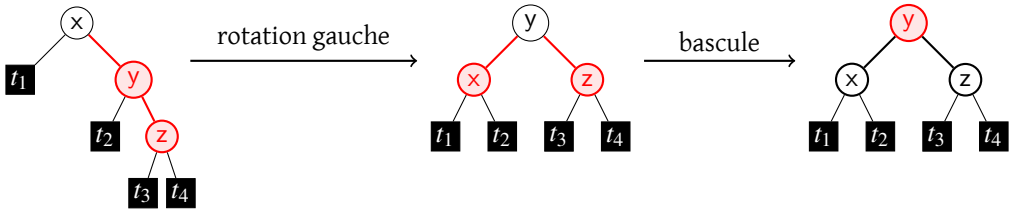
- Si $z \leq x \leq y$:

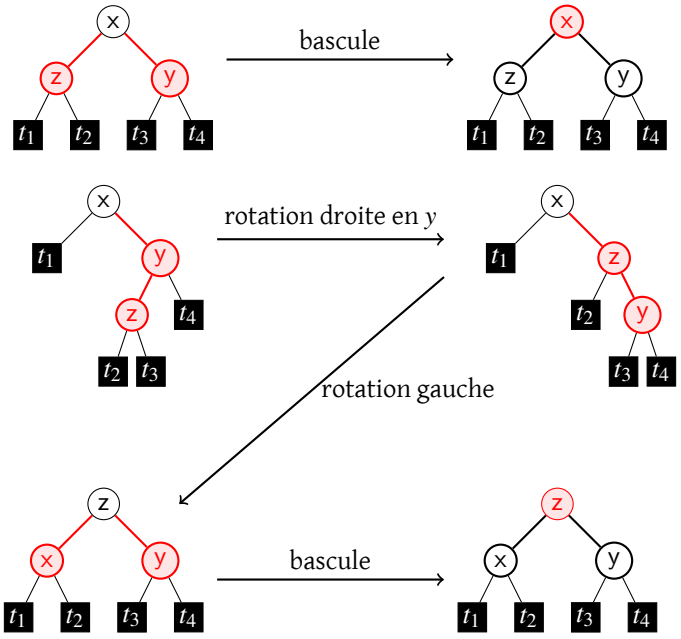


- Si $x \leq z \leq y$:



Ici, on ne peut pas rétablir tout de suite la structure mais on peut faire remonter le problème jusqu'à la racine, quitte à faire une bascule finale qui va la noircir. On va donc partir d'un des trois problèmes précédents mais supposer qu'il se situe sur un nœud quelconque avec des sous-arbres rouges et noirs t_1, t_2, t_3 et t_4.





On remarque qu'on effectue essentiellement les mêmes opérations, et quitte à les inhiber quand on ne peut pas les appliquer, on peut se dire que pour insérer z dans l'arbre a, on le rajoute récursivement à gauche ou à droite, on obtient ainsi a' et on effectue alors

$$(\text{bascule}^? \circ \text{rot\_gauche}^? \circ \text{rot\_droite}^?)(a')$$

où ici, $\text{op}^?$ signifie qu'on effectue l'opération uniquement si elle s'applique :

- pour une rotation droite, il faut que le fils gauche soit rouge et le fils droit noir, pour ne pas tomber dans le cas d'une bascule
- pour une rotation gauche, il faut que le fils droit et le fils droit du fils droit soient rouges
- pour une bascule, que les deux fils soient rouges

Implémentation

On commence par définir le type des arbres rouges et noirs :

OCaml
```
type couleur = R | N
type 'a arn = Nil | Noeud of couleur * 'a arn * 'a * 'a arn
```

C

Python
```
ERROR: src/structuresdonnees/../../snippets/structures/arn.py does not exist
```

On définit ensuite des fonctions qui nous seront utiles vue la remarque précédente pour tester les couleurs à gauche et à droite :

OCaml
```
let gauche a = match a with
    | Nil -> failwith "Invalide"
    | Noeud(_,g,_,_) -> g
let droite a = match a with
    | Nil -> failwith "Invalide"
    | Noeud(_,_,_,d) -> d
let couleur a = match a with
    | Nil -> N
    | Noeud(c,_,_,_) -> c
let rouge a = couleur a = R
let noir a = not (rouge a)
```

C

Python

```
ERROR: src/structuresdonnees/../../snippets/structures/arn.py does not exist
```

On implémente alors les rotations et la bascule avec le filtrage.

OCaml

```
let rotation_gauche a = match a with
    | Noeud(c,t1,x,Noeud(R,t2,y,t3)) ->
            Noeud(c,Noeud(R,t1,x,t2),y,t3)
    | _ -> failwith "Invalide"
let rotation_droite a = match a with
    | Noeud(c,Noeud(R,t1,x,t2),y,t3)
        -> Noeud(c,t1,x,Noeud(R,t2,y,t3))
    | _ -> failwith "Invalide"
let bascule a = match a with
    | Noeud(N,Noeud(R,t1,x,t2),y,Noeud(R,t3,z,t4))
        -> Noeud(R,Noeud(N,t1,x,t2),y,Noeud(N,t3,z,t4))
    | _ -> failwith "Invalide"
let noircit a = match a with
    | Noeud(_,g,x,d) -> Noeud(N,g,x,d)
    | Nil -> Nil
```

C

Python

```
ERROR: src/structuresdonnees/../../snippets/structures/arn.py does not exist
```

Puis, les versions optionnelles qui ne s'effectuent que si c'est nécessaire :

OCaml

```
let essai_rotgauche a =
    if rouge (droite a) && rouge (droite (droite a))
    then rotation_gauche a else a
let essai_rotdroite a =
    if rouge (gauche a) && noir (droite a)
    then rotation_droite a else a
let essai_bascule a =
    if rouge (gauche a) && rouge (droite a)
    then bascule a else a
```

C

Python

```
ERROR: src/structuresdonnees/../../snippets/structures/arn.py does not exist
```

On peut alors réaliser l'insertion récursivement :

OCaml

```
let insere a x =
    let rec aux a =
        match a with
        | Nil -> Noeud(R,Nil,x,Nil)
        | Noeud(c,g,y,d) ->
            let a' =
                if x < y
                then Noeud(c,aux g,y,d)
                else Noeud(c,g,y,aux d)
            in
            essai_bascule
                (essai_rotgauche
```

```
                    (essai_rotdroite a'))
    in noircit (aux a)
```

C

Python

```
ERROR: src/structuresdonnees/../../snippets/structures/arn.py does not exist
```

V Tas

V.1 Présentation

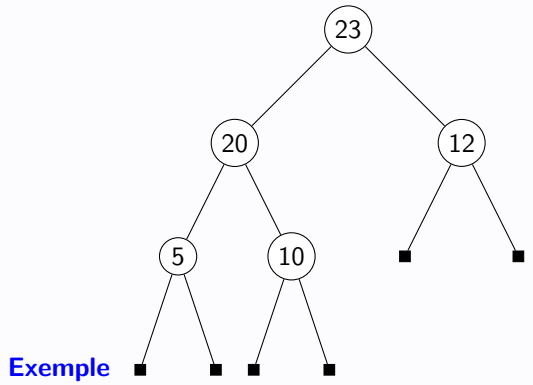
On va considérer une structure de donnée permettant de réaliser efficacement deux opérations :

- récupérer la plus grande des valeurs ajoutées et la supprimer
- ajouter une nouvelle valeur

Avec des arbres, on cherche à réaliser ces opérations en $O(\log n)$ où n est le nombre de valeurs présentes.

Définition V.1 On appelle *tas sur l'ensemble* X un arbre binaire étiquetté par X et complet à gauche tel que pour tout nœud de l'arbre, son étiquette est supérieure ou égale à celle de ses fils.

Exemple



Remarque On parle de *heap* en anglais. Ici, nous présentons des tas-max permettant de déduire le maximum, on pourrait définir des tas-min, mais quitte à renverser la relation d'ordre, il s'agit de la même structure.

V.2 Opérations

Dans les deux opérations qu'on souhaite implémenter, on va avoir besoin d'un accès rapide à la dernière feuille insérée et la place, à sa droite, de la prochaine feuille qu'on pourrait insérer.

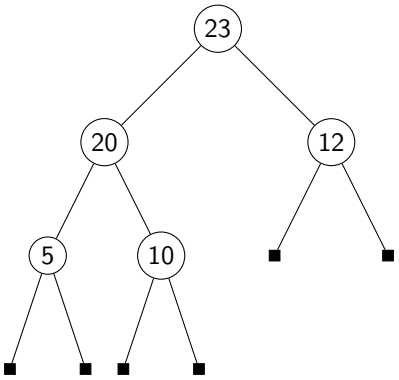
V.2.i Insertion

Pour insérer un élément dans un tas on adopte la stratégie suivante :

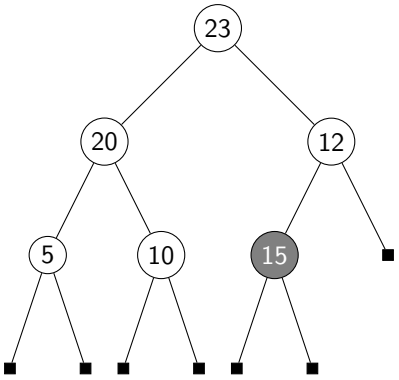
- on le place dans la seule place libre permettant de garantir que l'arbre soit complet à gauche ;
- on l'insère à la bonne place sur la branche qui le relie à la racine en effectuant des échanges successifs jusqu'à ce que la condition de tas soit vérifiée (étape appelée *remontée du tas*).

On va illustrer cet algorithme sur un exemple.

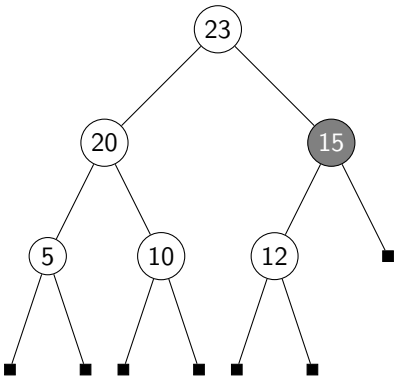
On part du tas



dans lequel on insère la valeur 15.
On commence par ajouter ce nœud dans le premier emplacement libre à droite de 10 :



Comme le fils droit de 12 a une valeur plus grande que celui-ci on échange les étiquettes pour obtenir :

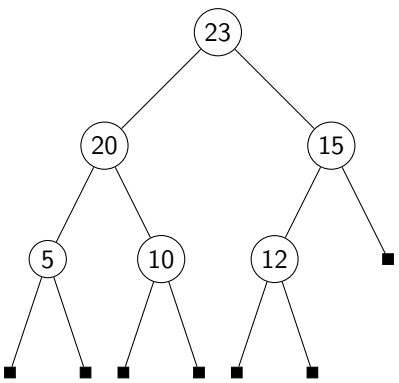


Maintenant le père de 15 est bien plus grand que celui-ci, on s'arrête car on a obtenu un tas.

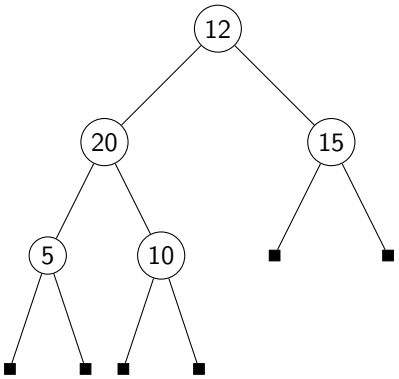
V.2.ii Extraction du maximum

Pour supprimer la racine :
- soit x l'étiquette du nœud situé juste avant le prochain emplacement libre (Il s'agit de la dernière position remplie par une insertion), on place la valeur x à la racine et le nœud où était situé x devient une feuille ;
- on l'insère ensuite à la bonne place à chaque étape avec le plus grand de ses fils, ce qui garantit de prendre un nouveau père respecte bien la condition de tas (étape appelée *descente du tas*).

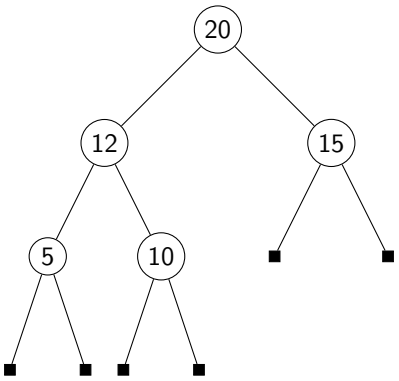
Sur le tas suivant on a $x = 12$



On obtient donc après la première étape l'arbre



On échange alors 12 avec 20 qui est le plus grand des fils :



Maintenant 12 est à la bonne place : on a obtenu un tas.

V.2.iii Complexité

Dans l'hypothèse de pouvoir accéder rapidement (au moins en $O(\log n)$) à la fin du dernier niveau, on peut réaliser ces deux opérations en remontant ou en descendant le long d'une branche, donc en $O(\log n)$.

V.3 Implémentation

On va reposer sur l'implémentation à plat d'un arbre comme décrit précédemment. On va supposer qu'on dispose d'une constante N et que l'on sait que le tas contiendra au maximum N valeurs. Cette implémentation est alors à rapprocher de celle faite pour les files ou les piles dans des tableaux.

OCaml
```
ERROR: src/structuresdonnees/../../snippets/structures/tas.ml does not exist
```

C
```c
struct tas {
    int *elements;
    int taille;
    int taille_max;
};
typedef struct tas tas;

tas *tas_cree(int N)
{
    tas *t = malloc(sizeof(tas));
    t->elements = malloc(N * sizeof(int));
    t->taille = 0;
    t->taille_max = N;
}

void tas_libere(tas *t)
{
    free(t->elements);
    free(t);
```

```c
}

void swap(int *t, int i, int j)
{
    int z = t[i];
    t[i] = t[j];
    t[j] = z;
}

int pere(int i)
{
    return (i-1)/2;
}

int gauche(int i)
{
    return 2*i+1;
}

int droite(int i)
{
    return 2*i+2;
}

void tas_insere(tas *t, int x)
{
    assert(t->taille < t->taille_max);
    int n = t->taille;
    t->taille++;
    t->elements[n] = x;
    while(n > 0 && t->elements[pere(n)] < t->elements[n])
    {
        swap(t->elements, n, pere(n));
        n = pere(n);
    }
}

bool feuille(tas *t, int i)
{
    return gauche(i) >= t->taille;
}

int max_gauche_droite(tas *t, int i)
{
    int vg = t->elements[gauche(i)];
    int vd = t->elements[droite(i)];
    if (vg > vd)
        return vg;
    else
        return vd;
}

int tas_extrait_max(tas *t)
{
    int root = t->elements[0];
    int x = t->elements[t->taille - 1];
    t->elements[0] = x;
    t->taille--;

    int n = 0;
    while(!feuille(t, n) && t->elements[n] < max_gauche_droite(t, n))
    {
        int vg = t->elements[gauche(n)];
        int vd = t->elements[droite(n)];
```

```
        if (vg > vd)
        {
            swap(t->elements, n, gauche(n));
            n = gauche(n);
        }
        else
        {
            swap(t->elements, n, droite(n));
            n = droite(n);
        }
    }

    return root;
}
```

Python
```
ERROR: src/structuresdonnees/../../snippets/structures/tas.py does not exist
```

V.4 Application au tri

Pour trier, il suffit de créer un tas, d'insérer tous les éléments puis de les extraire un à un. On appelle cela le *tri par tas*. C'est d'ailleurs l'origine historique de la structure de donnée.

Une manière intéressante de faire cet algorithme consiste à le faire en place, c'est-à-dire sans utiliser un autre tableau pour le tri. Pour cela, on va maintenir dans le même tableau le tas en construction en préfixe et les éléments à insérer en suffixe.

Pour reconstruire le tableau trié, il suffit alors d'extraire les maximums et des les placer à la fin du tableau en faisant ici grossir le suffixe trié et réduire le tas préfixe.

OCaml
```
ERROR: src/structuresdonnees/../../snippets/structures/tripartas.ml does not exist
```

C
```
void heapify(int *t, int sz)
{
    for(int i = 1; i < sz; i++)
    {
        while(i > 0 && t[(i-1)/2] < t[i])
        {
            int p = (i-1)/2;
            swap(t, i, p);
            i = p;
        }
    }
}

void sort(int *t, int sz)
{
    heapify(t, sz);
    for(int i = sz-1; i > 0; i--)
    {
        swap(t, i, 0);
        swap(t, i-1, 0);
        int j = 0;
        while(2*j+1 < i-1 && t[j] < max(t[2*j+1],t[2*j+2]))
        {
            int c = 2*j+2;
            if(t[2*j+1] >= t[2*j+2])
                c = 2*j+1;
            swap(t, j, c);
            j = c;
```

```
        }
    }
}
```

Python
```
ERROR: src/structuresdonnees/../../snippets/structures/tripartas.py does not exist
```

V.5 Application aux files de priorité

A la manière des dictionnaires, on peut considérer des couples $(prio, val)$ pour les étiquettes en utilisant uniquement la première composante dans le tas. Ainsi, on réalise la structure de données files de priorité introduite dans un chapitre précédent.

VI TP

VI.1 Arbres en OCaml

VI.1.i Premières fonctions

On va considérer le type des arbres binaires `'a arbre` défini par :

OCaml
```
type 'a arbre = Nil | Noeud of 'a arbre * 'a * 'a arbre
```

Question VI.1 Écrire des fonctions :

OCaml
```
(* Calcule le nombre de noeuds de a *)
let taille (a : 'a arbre) : int
(* Calcule la hauteur de a *)
let hauteur (a : 'a arbre) : int
(* Indique si a est réduit à une feuille *)
let feuille (a : 'a arbre) : bool
```

■ **Preuve**

OCaml
```
let rec taille a = match a with
    | Nil -> 0
    | Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur a = match a with
    | Nil -> -1
    | Noeud(g,x,d) ->
            1 + max (hauteur g) (hauteur d)

let feuille a = match a with
    | Nil -> false
    | Noeud(g, x, d) -> g = Nil && d = Nil
```

■

Question VI.2 Écrire une fonction `sous_arbres` qui renvoie la liste des sous-arbres non vides d'un arbre.
En déduire des fonctions `noeuds`, `feuilles` et `noeuds_internes` qui renvoient la liste des étiquettes des `noeuds` correspondants.

■ **Preuve**

OCaml
```
let rec sous_arbres a = match a with
    | Nil -> []
    | Noeud(g, x, d) ->
```

```
      a :: (sous_arbres g @ sous_arbres d)

let rec noeuds a = match a with
    | Nil -> []
    | Noeud(g, x, d) ->
        x :: (noeuds g @ noeuds d)

let rec feuilles a = match a with
    | Nil -> []
    | Noeud(g, x, d) ->
        if feuille a then [x]
        else feuilles g @ feuilles d

let rec noeuds_internes a = match a with
    | Nil -> []
    | Noeud(Nil, _, Nil) -> []
    | Noeud(g, x, d) ->
        x :: (noeuds_internes g @ noeuds_internes d)
```

■

Pour accéder à un nœud de l'arbre, on descend en partant de la racine et en allant à gauche ou à droite. On peut donc représenter un tel chemin par une liste de déplacements :

OCaml
```
type deplacement = Gauche | Droite
type chemin = deplacement list
```

Question VI.3 Écrire une fonction `chemin_noeud : 'a arbre -> chemin -> 'a option` qui renvoie l'étiquette d'un nœud donné par son chemin. On utilise un type `option` en cas de chemin invalide.

■ **Preuve**

OCaml
```
type deplacement = Gauche | Droite
type chemin = deplacement list

let rec chemin_noeud a l =
    match a, l with
    | Nil, _ -> None
    | Noeud(g,x,d), Gauche::q -> chemin_noeud g q
    | Noeud(g,x,d), Droite::q -> chemin_noeud d q
    | Noeud(g,x,d), [] -> Some x
```

■

VI.1.ii Parcours

Sur le même modèle que l'exploration par *backtracking*, on va réaliser un parcours en profondeur d'un arbre en explorant à gauche puis à droite ses sous-arbres de manière récursives. Chaque nœud est donc vu trois fois :

- une première fois quand on le découvre avant d'explorer son sous-arbre gauche
- entre les deux explorations
- enfin quand on a fini d'explorer son sous-arbre droit.

Question VI.4 Écrire des fonctions `affiche_avant`, `affiche_milieu` et `affiche_apres` qui parcourt et affiche les étiquettes d'un `string arbre` selon les trois cas précédents. *Note : il s'agit essentiellement du même code à une ligne près.*

■ **Preuve**

OCaml
```
let rec affiche_avant a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
            print_string x;
            affiche_avant g;
```

```
            affiche_avant d


let rec affiche_milieu a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
            affiche_milieu g;
            print_string x;
            affiche_milieu d


let rec affiche_apres a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
            affiche_apres g;
            affiche_apres d;
            print_string x
```

■

Question VI.5 On peut représenter une expression arithmétique comme $(2+3)\times 4$ en tant qu'arbre avec des opérations pour les nœuds internes et des valeurs pour les feuilles. Quitte à utiliser des `string` dans les deux cas, qu'obtient-on sur une telle expression avec les trois opérations précédentes ?

■ **Preuve**

OCaml
```
let rec evalue a = match a with
    | Nil -> failwith "Vide"
    | Noeud(Nil,x,Nil) -> int_of_string x
    | Noeud(g, x, d) ->
            let eg = evalue g in
            let ed = evalue d in
            match x with
            | "+" -> eg + ed
            | "*" -> eg * ed
            | _ -> failwith "Ope inconnue"
```

L'appel à `affiche_avant` sur l'expression $(2+3)\times 4$ va afficher `*+234`.
L'appel à `affiche_milieu` sur l'expression $(2+3)\times 4$ va afficher `2+3*4`.
L'appel à `affiche_apres` sur l'expression $(2+3)\times 4$ va afficher `23+4*`.
On retrouve ainsi les notions préfixes, infixes et postfixes d'une expression arithmétique.

■

On redonne ici l'implémentation d'une pile et d'une file en OCaml :

OCaml
```
let pile_creer () = ref []
let pile_empile p x = p := x :: !p
let pile_depile p =
    match !p with
    | [] -> failwith "Pile vide"
    | x::q -> p := q; x
let pile_est_vide p = !p = []

let file_creer () = (pile_creer (), pile_creer ())
let file_bascule (pin, pout) = pout := List.rev !pin; pin := []
let file_enfile (pin, pout) x = pile_empile pin x
let file_defile (pin, pout) =
    if pile_est_vide pout
    then file_bascule (pin, pout);
    pile_depile pout
let file_est_vide (pin, pout) = pile_est_vide pin && pile_est_vide pout
```

Ces deux structures de données ont la même interface mais deux comportements différents. On va utiliser un unique type pour les représenter afin de permettre à une fonction d'utiliser soit une pile soit une file.

```OCaml
type ('a, 'b) taches = {
    creation : unit -> 'a;
    ajouter : 'a -> 'b -> unit;
    retirer : 'a -> 'b;
    est_vide : 'a -> bool
}

let taches_pile = {
    creation = pile_creer;
    ajouter = pile_empile;
    retirer = pile_depile;
    est_vide = pile_est_vide
    }

let taches_file = {
    creation = file_creer;
    ajouter = file_enfile;
    retirer = file_defile;
    est_vide = file_est_vide
    }
```

Remarque Ici, on a une limitation du système de types. On aimerait paramètrer `taches` par `('a 'b, 'a)` où `'b` est un type paramètré, ainsi on aurait soit `('a pile, 'a)` où `('a file, 'a)` comme pour les deux case. Cependant, ce n'est pas possible, les types génériques ne sont pas paramètriques. C'est pour cela qu'on a un `('a, 'b)` sans avoir de lien apparent entre `'a` et `'b`.

On pourra alors écrire une fonction prenant un gestionnaire de tâches en paramètre :

```OCaml
let f t a =
    let a_traiter = t.creation () in
    for i = 0 to Array.length a - 1 do
        t.ajouter a_traiter a.(i)
    done;
    while not (t.est_vide a_traiter) do
        print_int (t.retirer a_traiter)
    done
```

Ainsi `f taches_pile [|1;2;3|]` va afficher `321` et `f taches_file [|1;2;3|]` va afficher `123`.

Question VI.6 En déduire une fonction

```OCaml
parcours : ('a, 'b arbre) taches -> 'b arbre -> unit
```

qui effectue un parcours de l'arbre en affichant les étiquettes des nœuds visités dans l'ordre induit par le gestionnaire de tâches passé en paramètre.

■ **Preuve**

```OCaml
let parcours t a =
    let avisiter = t.creation () in
    t.ajouter avisiter a;
    while not (t.est_vide avisiter) do
        match t.retirer avisiter with
        | Nil -> ()
        | Noeud(g,x,d) ->
                print_int x;
                t.ajouter avisiter g;
                t.ajouter visiter d
    done
```

■

VI.1.iii Arbres binaires stricts

Pour représenter des arbres dont les nœuds ont tous deux fils non vides, on va utiliser le type

```OCaml
type ('a, 'b) arbre_bin = Feuille of 'a
    | NoeudI of ('a, 'b) arbre_bin * 'b * ('a, 'b) arbre_bin
```

Ce type permet de représenter naturellement des expressions arithmétiques comme :

```OCaml
NoeudI(NoeudI( Feuille 2, '+', Feuille 3), '*', Feuille 5)
```

Question VI.7 Écrire une fonction `evalue : (int, char) arbre_bin -> int` qui évalue une telle expression.

■ **Preuve**

```OCaml
let rec evalue a = match a with
    | Feuille n -> n
    | NoeudI(g, x, d) ->
            let eg = evalue g in
            let ed = evalue d in
            match x with
            | '+' -> eg + ed
            | '*' -> eg * ed
            | _ -> failwith "Ope inconnue"
```

■

Si on considère le type

```OCaml
type ('a, 'b) etiquette = F of 'a | N of 'b
```

On peut passer d'un `('a, 'b) arbre_bin` à un `('a, 'b) etiquette arbre` et, dans certains cas, d'un `('a, 'b) etiquette arbre` à un `('a, 'b) arbre_bin`.

Question VI.8 Écrire ainsi deux fonctions de conversion :

```OCaml
arbre_bin_vers_arbre : ('a, 'b) arbre_bin -> ('a, 'b) etiquette arbre
arbre_vers_arbre_bin : ('a, 'b) etiquette arbre -> ('a, 'b) arbre_bin option
```

■ **Preuve**

```OCaml
let rec arbre_bin_vers_arbre a =
    match a with
    | Feuille n -> Noeud(Nil, F n, Nil)
    | NoeudI(g, x, d) -> Noeud(arbre_bin_vers_arbre g,
                               N x,
                               arbre_bin_vers_arbre d)

let rec arbre_vers_arbre_bin a =
    match a with
    | Noeud(Nil, F f, Nil) -> Feuille f
    | Noeud(g, N x, d) -> begin
            match arbre_vers_arbre_bin g, arbre_vers_arbre_bin d with
            | Some g', Some d' -> NoeudI(g', x, d')
            | _ -> None
        end
    | _ -> None
```

■

VI.1.iv Dessin d'arbres

On va réaliser dans cette partie une fonction de dessin d'arbres avec `graphics`. L'idée est de placer la racine et de dessiner les sous-arbres gauche et droit en dessous. Pour cela, il va falloir connaitre la taille de ces sous-arbres en pixels.

On pourra consulter la documentation de `graphics` ici : Graphics

Question VI.9 Écrire une fonction `pixels : int arbre -> int * int` qui renvoie un couple `(largeur, racine)` où `largeur` est la largeur en pixels d'un arbre dans son affichage et `racine` l'abscisse de la racine dans cet affichage.

■ **Preuve**

OCaml
```
let rnoeud = 5 (* Le rayon du noeud *)
let marge = 2 (* la marge autour du noeud *)

let rec pixels a = match a with
    | Nil -> 0, 0
    | Noeud(g, _, d) ->
            let lg, _ = pixels g in
            let ld, _ = pixels d in
            lg + ld + 2*(marge + rnoeud), lg + marge + rnoeud
```

■

Question VI.10 En déduire une fonction `dessine : int arbre -> int -> int -> unit` tel que `dessine a x y` dessine l'arbre a en plaçant la racine en (x, y).

■ **Preuve**

OCaml
```
let rec dessine a x y =
    match a with
    | Nil -> ()
    | Noeud(g, e, d) ->
            (* On précalcule les tailles des sous-arbres *)
            let lg, rg = pixels g in
            let _, rd = pixels d in
            (* position de la racine gauche relativement à (x,y) *)
            let v_rg = x - lg - marge - rnoeud + rg in
            (* position de la racine droite relative (x, y) *)
            let v_rd = x + marge + rnoeud + rd in
            let dec_y = y - 2 * rnoeud - marge in

            moveto x y; lineto v_rg dec_y;
            moveto x y; lineto v_rd dec_y;
            fill_circle x y rnoeud;

            moveto (x-tw/2) (y-th/2);
            set_color white;
            draw_string e;
            set_color black;

            dessine g v_rg dec_y;
            dessine d v_rd dec_y
```

■

VI.1.v Génération aléatoire d'arbres

Dans cette partie, ce qui nous intéresse est de générer aléatoirement des arbres pour en observer la structure. Les étiquettes ne sont pas pertinentes, on pourra donc au choix, soit considérer des `unit arbre`, soit redéfinir un type d'arbre non étiquettés.

On va réaliser ici deux modèles de génération aléatoire d'arbres. Le premier modèle consiste, pour générer un arbre de n nœuds, à choisir aléatoirement k dans $[\![0, n]\!]$, à générer aléatoirement un arbre g à k nœuds et un

arbre d à $n-k$ nœuds puis à renvoyer l'arbre `Noeud(g, (), d)`.

Question VI.11 Implémenter ce modèle avec une fonction `genere_arbre_1 : int -> unit arbre`. Tester votre fonction, notamment avec l'affichage de la partie précédente.

■ **Preuve**

```OCaml
let rec genere_arbre_1 n =
    if n = 0
    then Nil
    else if n = 1 then Noeud(Nil, (), Nil)
    else let k = Random.int (n-1) in
        Noeud(genere_arbre_1  k, (),
            genere_arbre_1  (n-1-k))
```

■

Pour le deuxième modèle, on va choisir un sous-arbre vide uniformément et le remplacer par un nœud.

Question VI.12 Écrire une fonction `chemins_vides : 'a arbre -> chemin list` qui étant donné un arbre renvoie la liste des chemins permettant d'aboutir à un sous-arbre vide.

■ **Preuve**

```OCaml
let chemins_vide a =
    let rec aux a l =
        match a with
        | Nil -> [ List.rev l ]
        | Noeud(g, _, d) ->
              aux g (Gauche::l)
            @ aux d (Droite::l)
    in aux a []
```

■

Question VI.13 Écrire une fonction `remplit : 'a arbre -> chemin -> 'a arbre` qui remplace un sous-arbre vide donnée par son chemin par un nœud.

■ **Preuve**

```OCaml
let rec remplit a c =
    match a, c with
    | Nil, [] -> Noeud(Nil, (), Nil)
    | Noeud(g,x,d), Gauche::q ->
            Noeud(remplit g q, x, d)
    | Noeud(g,x,d), Droite::q ->
            Noeud(g, x, remplit d q)
    | _ -> failwith "Error"
```

■

Question VI.14 En déduire une fonction `genere_arbres_2 : int -> unit arbre` qui prend un entier `n` et construit un arbre à n nœuds en partant de l'arbre vide et en réalisant n remplacement d'un arbre vide par un nœuds, le choix de l'arbre vide se faisant uniformément.

Note : on pourra utilser `List.nth` et `Random.int` Voir par exemple Random

■ **Preuve**

```OCaml
let rec genere_arbre_2 n =
    if n = 0 then Nil
    else
        let a = genere_arbre_2 (n-1) in
```

```
        let l = chemins_vide a in
        let k = Random.int (List.length l) in
        let ch = List.nth l k in
        remplit a ch
```

■

VI.2 Arbres non binaires, tries

VI.2.i Arbres non binaires

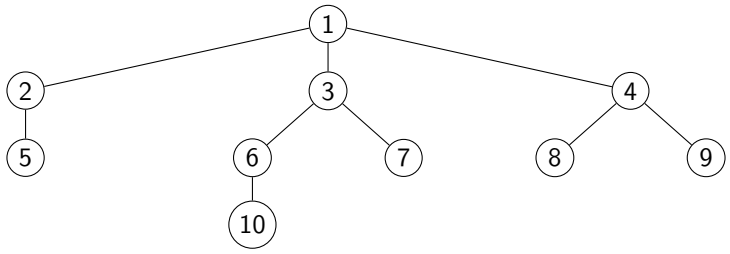
On va considérer ici les arbres d'arité quelconque avec des listes d'enfants pour les nœuds.

OCaml
```
type 'a arbre = { etiquette : 'a; enfants : 'a arbre list }
```

Afin d'effectuer des tests, on pourra considérer l'arbre suivant :

OCaml
```
let t = { etiquette = 1;
    enfants = [
        { etiquette = 2; enfants = [
            { etiquette = 5; enfants = [] }
            ] };
        { etiquette = 3; enfants = [
            { etiquette = 6; enfants = [
                { etiquette = 10 ; enfants = []}
            ]};
            { etiquette = 7; enfants = []}
        ] };
        { etiquette = 4; enfants = [
                { etiquette = 8; enfants = []};
                { etiquette = 9; enfants = []}
        ] }
    ] }
```

qui correspond à la représentation :



Comme présenté dans ce chapitre, on peut réaliser des fonctions mutuellement récursives pour travailler avec ces arbres et sur la forêt induite par les enfants d'un nœud :

OCaml
```
let rec taille_arbre a =
    1 + taille_foret a.enfants
    and taille_foret l = match l with
        | [] -> 0
        | t::q -> taille_arbre t + taille_foret q
```

Question VI.15 Écrire une fonction `hauteur_arbre : 'a arbre -> int` qui calcule la hauteur d'un arbre sur le modèle de la fonction précédente. On fera attention au fait qu'il n'est pas possible de représenter l'arbre vide mais la forêt vide jouera le même rôle vis-à-vis de -1.

■ **Preuve**

OCaml
```
let rec hauteur_arbre a =
    1 + hauteur_foret a.enfants
    and hauteur_foret l = match l with
```

```
    | [] -> -1
    | t::q -> max (hauteur_arbre t) (hauteur_foret q)
```

■

Question VI.16 A l'aide d'un parcours récursif, écrire une fonction `affiche_etiquettes : int tree -> unit` qui affiche toutes les étiquettes d'un `int tree`.

■ **Preuve**

OCaml
```
let rec affiche_etiquettes a =
    Printf.printf "%d\n" a.etiquette;
    affiche_foret a.enfants
    and affiche_foret l = match l with
        | [] -> ()
        | t::q -> affiche_etiquettes t; affiche_foret q
```

■

VI.2.ii Représentation par des arbres binaires

On va maintenant revenir ici sur la représentation d'un `'a arbre` en tant qu'arbre binaire. Pour cela, on va avoir besoin d'étiquettes factices pour les nouveaux nœuds qui ne servent qu'à supporter la structure des nœuds d'arité > 2. On va ainsi les représenter par des `a option arbre_bin` où l'étiquette sera celle des nœuds factices.

OCaml
```
type 'a arbre_bin = Noeud of 'a arbre_bin * 'a * 'a arbre_bin | Nil
```

Question VI.17 On va commencer par écrire une fonction `peigne : 'a option arbre_bin list -> 'a option arbre_bin` qui va transformer une forêt d'arbres binaires en sa représentation en peigne à l'aide de nœuds d'étiquettes `None`.

Note Dans le cas où la liste est vide, on renverra l'arbre vide et si c'est un singleton, on renverra directement l'arbre qu'il contient.

■ **Preuve**

OCaml
```
let rec peigne l =
    match l with
    | [] -> Nil
    | [x] -> x
    | a::q -> Noeud(a, None, peigne q)
```

■

Question VI.18 Écrire une fonction `convert : 'a arbre -> 'a option arbre_bin` qui va réaliser la conversion vue en classe.

■ **Preuve**

OCaml
```
let rec convert a =
    match a.enfants with
    | [] -> Noeud (Nil, Some a.etiquette, Nil)
    | a1::q -> Noeud(convert a1, Some a.etiquette,
                        peigne (List.map convert q))
```

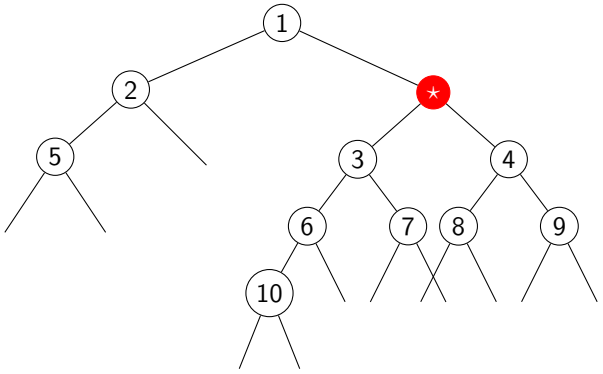
■

Afin de tester votre fonction, on vous donne une fonction d'affichage textuel d'un tel arbre :

OCaml
```
let rec print_arbre_bin pref t =
    match t with
```

```
    | Nil -> Printf.printf "%snil\n%!" pref
    | Noeud(g,x,d) -> Printf.printf "%s%s\n%!" pref
                (match x with None -> "*" | Some n -> string_of_int n);
        print_arbre_bin (String.make (String.length pref) ' '
                         ^ "|" ^ String.make 5 '-') g;
        print_arbre_bin (String.make (String.length pref) ' '
                         ^ "|" ^ String.make 5 '-') d
```

L'affichage doit correspondre à la représentation suivante :



VI.3 Tries

On va reprendre les tries vus plus haut pour représenter un ensemble de mots. Pour cela, on réutilise le type :

OCaml
```
type trie = {
    mot : bool;
    enfants : (char * trie) list
    }
```

VI.3.i Premières manipulations

Question VI.19 Écrire une fonction `taille : trie -> int` qui renvoie le nombre de nœuds d'un trie et une fonction `nb_mots : trie -> int` qui renvoie le nombre de mots d'un trie. On s'inspirera directement des fonctions de la première partie du TP.

■ **Preuve**

OCaml
```
let rec taille a =
    1 + taille_foret a.enfants
    and taille_foret l = match l with
        | [] -> 0
        | (_, t) :: q -> taille t + taille_foret q

let rec nb_mots a =
    (if a.mot then 1 else 0) + nb_mots_foret a.enfants
    and nb_mots_foret l = match l with
        | [] -> 0
        | (_, t) :: q -> nb_mots t + nb_mots_foret q
```

■

Question VI.20 Écrire une fonction `trie_apres_char` tel que l'appel à `trie_apres_char l c` va renvoyer `Some t'` si il y a un couple `(c, t')` dans la liste `l` et `None` sinon. On pourra s'en servir avec `trie_apres_char t.enfants c` pour trouver le trie potentiel dont l'arête est étiquetée par `c`.

■ **Preuve**

OCaml
```
let rec trie_apres_char (l:(char * trie) list) (c:char) =
    match l with
    | [] -> None
    | (c',t)::q -> if c = c' then Some t else trie_apres_char q c
```

■

Question VI.21 En déduire une fonction `contient : trie -> string -> bool` qui teste si un mot est présent dans le trie.

■ **Preuve**

```OCaml
let contient (t:trie) (s:string) : bool =
    let rec aux t i =
        if i = String.length s
        then t.mot
        else match trie_apres_char t.enfants s.[i] with
            | None -> false
            | Some t' -> aux t' (i+1)
    in aux t 0
```

■

Question VI.22 Écrire une fonction `enumere : trie -> string list` qui renvoie tous les mots contenus dans un trie par un parcours récursif.

Indication : on pourra utiliser :

- `List.concat` pour passer d'une liste de listes à la concaténée de ses éléments. Ainsi `List.concat [ [1;2]; [3]; [4;5] ]` va renvoyer `[1;2;3;4;5]`.
- `String.make 1 c` transforme le caractère `c` en chaîne de caractère
- On rappelle que `^` permet de concaténer deux chaînes.

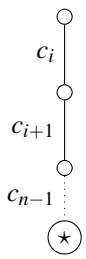
■ **Preuve**

```OCaml
let enumere (t:trie) =
    let rec aux t acc =
        let l = List.concat (List.map (fun (c,t') ->
                aux t' (acc ^ String.make 1 c)) t.enfants) in
        if t.mot then acc :: l else l
    in aux t ""
```

■

VI.3.ii Ajout d'un mot à un trie

Pour effectuer l'ajout d'un mot à un trie, on va commencer par écrire une fonction qui étant donné une chaîne de caractère `s` = "$c_0 \dots c_{n-1}$" et un entier $i \leq n$ va renvoyer le trie `t` correspondant à la branche



Question VI.23 écrire la fonction `suffixe : int -> string -> trie`

■ **Preuve**

```OCaml
let rec suffixe (i:int) (s:string) : trie =
    let n = String.length s in
    if i = n
    then { mot = true; enfants = [] }
    else { mot = false; enfants = [ (s.[i], suffixe (i+1) s) ] }
```

■

Question VI.24 Écrire une fonction `avec_trie_pour_char : (char * trie) list -> char -> trie -> (char * trie) list` telle que `avec_trie_pour_char l c t` va renvoyer la liste obtenue en remplaçant `(c,t')` par `(c,t)` dans `l` si un tel couple est présent, ou en ajoutant le couple `(c,t)` sinon à la fin.

■ **Preuve**

OCaml
```
let rec avec_trie_pour_char l c t =
    match l with
    | [] -> [ (c, t) ]
    | (c', t') :: q -> if c = c' then (c, t) :: q else (c', t') :: avec_trie_pour_char q c t
```

■

Question VI.25 Écrire une fonction `ajoute : trie -> string -> trie` qui renvoie le trie obtenu en ajoutant un nouveau mot.

■ **Preuve**

OCaml
```
let ajoute t s =
    let n = String.length s in
    let rec aux t i =
        if i = n
        then { t with mot = true }
        else let t' = match trie_apres_char t.enfants s.[i] with
                    | None -> suffixe (i+1) s
                    | Some t' -> aux t' (i+1) in
            { t with enfants = avec_trie_pour_char t.enfants s.[i] t' }
    in aux t 0
```

■

On va utiliser la fonction précédente pour retrouver le trie donné en début de partie.

OCaml
```
let rec trie_of_list l = match l with
    | [] -> { mot = false; enfants = [] }
    | s::q -> ajoute (trie_of_list q) s
```

On va maintenant récupérer une grande liste de mots dans le fichier `wordlist` récupérable à l'adresse https://raw.githubusercontent.com/dwyl/english-words/master/words_alpha.txt à raison d'un mot par ligne :

OCaml
```
let wordlist = let rec lit_fichier f =
        try
            let s = input_line f in
                s :: lit_fichier f
            with End_of_file -> []
        in
        lit_fichier (open_in "wordlist")
```

Question VI.26 En déduire un trie permettant de représenter cette wordlist et comparez :
- Le nombre de caractère total
- Le nombre de nœuds du trie

Que peut-on en déduire ?

■ **Preuve**

OCaml
```
let taille_totale = List.fold_left (+) 0 (List.map String.length wordlist) in
taille_totale, taille (trie_of_list wordlist)
```

On voit qu'on 595775 caractères au total pour 229239 nœuds. Attention cependant à ne pas sous-estimer la taille d'un nœud qui dépasse très largement celle d'un caractère. Cependant, le trie apporte la possibilité de chercher un mot m en $O(|m|)$ et donc indépendamment du nombre de mots.

■

VI.4 Arbres binaires en C, ABR, arbres rouges et noirs

VI.4.i Arbres binaires en C

En C, un nœud d'un arbre binaire va avoir exactement la même structure qu'un maillon d'une liste doublement chaînée : une valeur et deux pointeurs vers d'autres nœuds, ici ce ne sont pas `suivant` et `precedent` mais `gauche` et `droite` :

```c
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    int valeur;
};

typedef struct noeud *arbre;
```

Comme pour les listes, l'arbre vide sera représenté par le pointeur `NULL`.

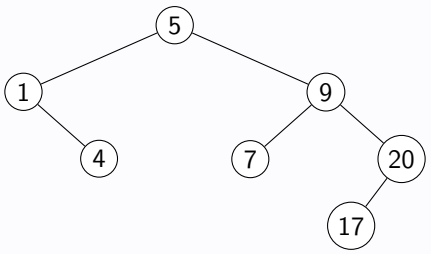
Question VI.27 Écrire une fonction `arbre cree_feuille(int x)` qui alloue et crée une feuille de valeur `x`.

■ **Preuve**

```c
arbre cree_feuille(int x)
{
    arbre n = malloc(sizeof(struct noeud));
    n->valeur = x;
    n->gauche = n->droite = NULL;
    return n;
}
```

■

Question VI.28 En déduire, une représentation en C de l'arbre :



Pour cela, on vous suggère de créer autant de feuilles que de nœuds et d'effectuer les liens ensuite avec les pointeurs.

Cet arbre nous servira d'exemple pour vérifier les fonctions suivantes.

■ **Preuve**

```c
    arbre n5 = cree_feuille(5);
    arbre n1 = cree_feuille(1);
    arbre n4 = cree_feuille(4);
    arbre n9 = cree_feuille(9);
    arbre n7 = cree_feuille(7);
    arbre n20 = cree_feuille(20);
    arbre n17 = cree_feuille(17);
    n5->gauche = n1;
    n1->droite = n4;
    n5->droite = n9;
    n9->gauche = n7;
    n9->droite = n20;
    n20->gauche = n17;
```

```
    arbre a = n5;
```

■

Question VI.29 Écrire une fonction `int taille(arbre a)` qui renvoie le nombre de nœuds d'un arbre binaire.

■ **Preuve**

```c
int taille(arbre a)
{
    if (a == NULL)
        return 0;
    return 1 + taille(a->gauche) + taille(a->droite);
}
```

■

Question VI.30 Écrire une fonction `int hauteur(arbre a)` qui renvoie la hauteur d'un arbre binaire.

■ **Preuve**

```c
int max(int a, int b)
{
    if (a > b) return a;
    else return b;
}

int hauteur(arbre a)
{
    if (a == NULL)
        return -1;
    return 1 + max(hauteur(a->gauche), hauteur(a->droite));
}
```

■

Question VI.31 Écrire des fonctions `void affiche_prefixe(arbre a)`, `affiche_infixe` et `affiche_postfixe` qui affiche les valeurs des nœuds de l'arbre `a` selon le traitement correspondant dans un parcours.

Écrire une fonction `void affiche(arbre a)` qui, à l'aide des trois traitements, permet d'afficher un arbre avec des triplets comme nœuds. Par exemple, pour l'arbre précédent, on pourra afficher :

`((nil,1,(nil,4,nil)),5,((nil,7,nil),9,((nil,17,nil),20,nil)))`

■ **Preuve**

```c
void affiche_prefixe(arbre a)
{
    if (a != NULL)
    {
        printf("%d", a->valeur);
        affiche_prefixe(a->gauche);
        affiche_prefixe(a->droite);
    }
}

void affiche_infixe(arbre a)
{
    if (a != NULL)
    {
        affiche_infixe(a->gauche);
        printf("%d", a->valeur);
        affiche_infixe(a->droite);
```

```c
    }
}

void affiche_postfixe(arbre a)
{
    if (a != NULL)
    {
        affiche_postfixe(a->gauche);
        affiche_postfixe(a->droite);
        printf("%d", a->valeur);
    }
}

void affiche(arbre a)
{
    if (a != NULL)
    {
        printf("(");
        affiche(a->gauche);
        printf(",%d,", a->valeur);
        affiche(a->droite);
        printf(")");
    } else printf("nil");
}
```

■

Question VI.32 *(Optionnel en première lecture)* À l'aide d'une file dans un tableau de longueur $|a|$, réalisez une fonction `void affiche_largeur(arbre a)` qui affiche les valeurs des nœuds de l'arbre `a` lors d'un parcours en largeur.

■ **Preuve**

```c
void affiche_largeur(arbre a)
{
    arbre *file = malloc(sizeof(arbre) * taille(a));
    file[0] = a;
    int debut = 0;
    int fin = 1;
    while(fin > debut)
    {
        arbre r = file[debut];
        debut++;
        printf("%d", r->valeur);
        if(r->gauche != NULL)
        {
            file[fin] = r->gauche;
            fin++;
        }
        if(r->droite != NULL)
        {
            file[fin] = r->droite;
            fin++;
        }
    }
}
```

■

Question VI.33 Écrire une fonction `void libere(arbre a)` qui libère tous les nœuds de l'arbre `a`. On fera attention à l'ordre des libérations avec `free` par rapport aux déréférencements de structure (les `->`).

■ **Preuve**

```c
void libere(arbre a)
{
    if (a != NULL)
    {
        libere(a->gauche);
        libere(a->droite);
        free(a);
    }
}
```

■

VI.4.ii Arbres binaires de recherche

On conserve le type précédent, mais on considère maintenant qu'on a des arbres binaires de recherche.

Question VI.34 Écrire une fonction `arbre recherche(arbre a, int x)` qui renvoie le premier nœud rencontré d'étiquette `x` dans un ABR ou `NULL` s'il n'est pas présent.

■ **Preuve**

```c
arbre recherche(arbre a, int x)
{
    if (a == NULL)
        return NULL;
    if (a->valeur == x)
        return a;
    if (a->valeur < x)
        return recherche(a->droite, x);
    return recherche(a->gauche, x);
}
```

■

Question VI.35 Écrire une fonction `arbre insere(arbre a, int x)` qui renvoie la racine de l'ABR obtenu en insérant une nouvelle feuille d'étiquette `x`.

■ **Preuve**

```c
arbre insere(arbre a, int x)
{
    if (a == NULL)
        return cree_feuille(x);

    if (a->valeur >= x)
    {
        a->gauche = insere(a->gauche, x);
    }
    else
    {
        a->droite = insere(a->droite, x);
    }

    return a;
}
```

■

Question VI.36 Écrire une fonction `arbre minimum(arbre a)` qui renvoie le nœud de valeur minimale de `a` en allant le plus à gauche dans l'arbre `a`. On supposera `a` non vide.

■ **Preuve**

```c
arbre minimum(arbre a)
{
    if (a->gauche == NULL)
        return a;
    return minimum(a->gauche);
}
```

■

Question VI.37 *(Optionnel en première lecture)* Écrire une fonction `arbre supprime(arbre a, int x)` qui renvoie la racine de l'ABR obtenu en supprimant le premier nœud rencontré d'étiquette `x`.

■ **Preuve**

```c
arbre supprime(arbre a, int x)
{
    if (a == NULL)
        return a;
    if (a->valeur == x)
    {
        if (a->gauche == NULL && a->droite == NULL)
        {
            free(a);
            return NULL;
        }
        arbre min_a = minimum(a);
        if (min_a == a)
        {
            arbre a_p = a->droite;
            free(a);
            return a_p;
        }
        int min_a_valeur = min_a->valeur;
        supprime(a, min_a_valeur);
        a->valeur = min_a_valeur;
    } else {
        if (a->valeur > x)
            a->gauche = supprime(a->gauche, x);
        else
            a->droite = supprime(a->droite, x);
    }

    return a;
}
```

■

VI.4.iii Arbres rouges et noirs

On va rajouter un champ au type précédent pour indiquer la couleur d'un nœud :

```c
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    bool rouge;
    int valeur;
};
typedef struct noeud *arbre;
```

On vous conseille ici de copier le fichier précédent et de faire un nouveau programme.

Question VI.38 Reprendre `cree_feuille` pour que les feuilles soient rouges par défaut.

■ **Preuve**

```c
arbre cree_noeud(int x)
{
    arbre n = malloc(sizeof(struct noeud));
    n->valeur = x;
    n->rouge = true;
    n->gauche = n->droite = NULL;
    return n;
}
```

■

Question VI.39 Écrire des fonctions `arbre rotation_droite(arbre a)`, `arbre rotation_gauche(arbre a)` et `arbre bascule(arbre a)` qui effectuent une des trois opérations en renvoyant la nouvelle racine.

■ **Preuve**

```c
arbre rotation_droite(arbre x)
{
    arbre y = x->gauche;
    x->gauche = y->droite;
    y->droite = x;
    y->rouge = x->rouge;
    x->rouge = true;
    return y;
}

arbre rotation_gauche(arbre y)
{
    arbre x = y->droite;
    y->droite = x->gauche;
    x->gauche = y;
    x->rouge = y->rouge;
    y->rouge = true;
    return x;
}

arbre bascule(arbre a)
{
    a->rouge = true;
    a->gauche->rouge = false;
    a->droite->rouge = false;
    return a;
}
```

■

Question VI.40 Écrire des fonctions utilitaires `bool rouge(arbre a)` et `bool noir(arbre a)` qui teste la couleur de la racine. On fera attention au fait que l'arbre vide est noir.

■ **Preuve**

```c
bool rouge(arbre a)
{
    if (a == NULL) return false;
    return a->rouge;
}

bool noir(arbre a)
{
    return !rouge(a);
}
```

■

Question VI.41 Écrire des fonctions `arbre essai_rotgauche(arbre a)`, `essai_rotdroite` et `essai_bascule` qui n'effectuent ces opérations que dans les cas nécessaires pour l'insertion.

■ **Preuve**

```c
arbre essai_bascule(arbre a)
{
    if(rouge(a->gauche) && rouge(a->droite))
        return bascule(a);
    return a;
}

arbre essai_rotgauche(arbre a)
{
    if(rouge(a->droite) && rouge(a->droite->droite))
        return rotation_gauche(a);
    return a;
}

arbre essai_rotdroite(arbre a)
{
    if(rouge(a->gauche) && noir(a->droite))
        return rotation_droite(a);
    return a;
}
```

■

Question VI.42 En déduire une fonction `arbre insere(arbre a, int x)` qui effectue l'insertion dans un arbre rouge et noir. On pourra définir une fonction auxiliaire `arbre insere_aux(arbre a, int x)` qui effectue l'insertion et qu'on appellera dans `insere` avant de faire le noircissement final de la racine.

■ **Preuve**

```c
arbre insere_aux(arbre a, int x)
{
    if (a == NULL)
        return cree_noeud(x);
    if (a->valeur > x)
        a->gauche = insere_aux(a->gauche, x);
    else
        a->droite = insere_aux(a->droite, x);
    return essai_bascule(essai_rotgauche(essai_rotdroite(a)));
}

arbre insere(arbre a, int x)
{
    arbre a2 = insere_aux(a, x);
    a2->rouge = false;
    return a2;
}
```

■