Structures Hiérarchiques : Arbres, Induction, Arbres binaires de recherche, Tas

2nd Semestre

Plan

Conseil de Classe

Conseile de classe le **Jeudi 8 Février**.
Vous devriez pouvoir avoir accès à vos bulletin peu avant le conseil de classe. Je peux peut-être corriger d'éventuelles erreurs.
Il y a des petites différences entre les classements que j'avais donnés et les classement en pratique en raison des erreurs dans le décompte des points.

Bilan du semestre précédent

Ce que nous avons vu :

- Programmation en OCaml, récursivité, fonctions d'ordre supérieur ;
- Programmation en C, gestion de la mémoire, représentation mémoire, gestion des ressources ;
- Types et structures de données, structures linéaires ;
- Analyse de programme, correction, terminaison, complexité.

Bilan du semestre précédent

Ce que nous avons vu :

- Programmation en OCaml, récursivité, fonctions d'ordre supérieur ;
- Programmation en C, gestion de la mémoire, représentation mémoire, gestion des ressources ;
- Types et structures de données, structures linéaires ;
- Analyse de programme, correction, terminaison, complexité.

Il nous manque principalement des choses sur les tests que nous verrons à l'occasion du cours sur les graphes. Par ailleurs, nous n'avons pas pu approfondir les détails du système, car nous n'avions pas les PC sous linux pour le faire en TP ensemble.

Bilan du semestre précédent

Quelques soucis généraux :

- Des soucis dans le code, dans la gestion mémoire ;
- Des difficultés à s'approprier de nouvelles structures ou de nouveaux objets ;
- La mise en page, la rigueur, et la méthode pas parfaits.

Organisation du second semestre

Les heures de cours :

- 4 heures en classe entières : **Lundi 13h35-15h25**, **Lundi 13h35-15h25** ;
- 2 heures en groupes : **Mardi 7h45-9h50** pour le groupe 2, **Mardi 15h40-17h30** pour le groupe 1.

Support de Cours :

- Le polycopié ;
- Les diapos ;
- Les TP.

Évaluations :

- 1h de colle par trinôme toutes les 4 semaines ;
- 3 DS en classe entière les vendredi **15 Mars**, **3 Mai** et **7 Juin** ;
- 3 DM, le prochain pour le **4 Mars**, il contiendra une sélection d'exercices du TP.

Des colles en trinômes

- Colle en groupe de 3 à 4 ;
- Pas de machine, juste face au tableau ;
- Questions de cours ;
- Programmation potentiellement dans les deux langages toutes les semaines ;
- Et des exercices théoriques ou de programmation.

Programme du semestre suivant

- Arbres et structures hiérarchiques ;
- Décomposition en sous-problèmes : algo gloutons, diviser-pour-régner, programmation dynamique ;
- Graphes et structures relationnelles ;
- Base de données, SQL ;
- Logique ;
- Algorithme des textes.

Arbres et vocabulaire sur les arbres

Définition 1 : Arbre

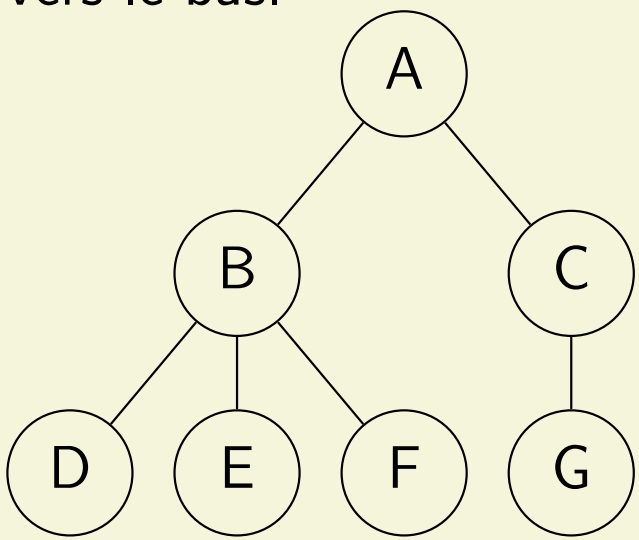
Un **arbre** est une structure de donnée hiérarchique finie qui se ramifie depuis une origine.

Définition 1 : Arbre

Un **arbre** est une structure de donnée hiérarchique finie qui se ramifie depuis une origine.

Exemple 1 : Exemple d'arbre

En informatique, on représente les arbres généralement dirigés vers le bas.



Exemple 2 : Utilisation des arbres

Les arbres ont de nombreuses applications directes :

- Appels récursifs d'une fonction avec plusieurs appels récursifs ;
- Représentation d'arbres de décision, de possibilités ;

Définition 2 : Nœud

Un **nœud** est un élément d'un arbre.

Définition 3 : Étiquette

Une **étiquette** est une information contenue dans un nœud, ou un identifiant pour ce nœud.
On limite parfois les étiquettes à un sous-ensemble des nœuds d'un arbre.

Définition 4 : Hiérarchie dans un arbre

Le **parent** ou **père** d'un nœud est le nœud qui est juste avant dans l'arbre.
Un **enfant** ou **fils** d'un nœud est un nœud qui est juste après dans l'arbre.

Définition 5 : Racine

La **racine** d'un arbre est un nœud qui n'a pas de parent.

Définition 6 : Feuille

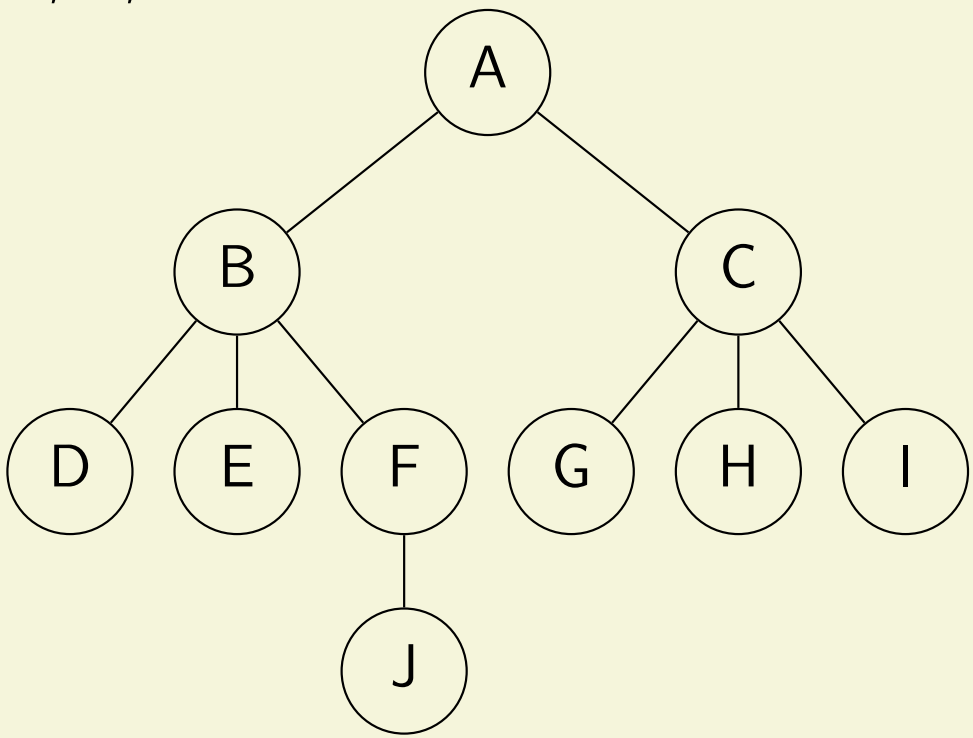
Une **feuille** (ou parfois **nœud externe**) est un nœud qui n'a pas d'enfants.

Définition 7 : Nœud interne

Un **nœud interne** est un nœud qui a au moins un enfant.

Exemple 3

La racine de cet arbre est le nœud A. B a trois enfants qui sont D, E, et F. Le parent de G est C.
Les feuilles sont D, E, G, H, I et J. Les nœuds internes sont A, B, F et C.



Définition 8 : Profondeur d'un nœud

La **profondeur** d'un nœud est la distance à la racine du nœud, c'est-à-dire le nombre de saut qu'il faut faire depuis la racine pour atteindre le nœud.

Définition 9 : Hauteur d'un arbre

La **hauteur** d'un arbre (ou parfois **profondeur**) est le maximum de profondeur de ses nœud.

Exercice 1. Quelles sont les profondeurs des nœuds de l'arbre de l'exemple précédent ? Quelle est la hauteur de l'arbre ?

Définition 10 : Niveau d'un arbre

Un **niveau** d'un arbre est l'ensemble des nœuds à une profondeur donnée.

Définition 11 : Arité d'un arbre

Un **arbre d'arité** n est un arbre dont tous les nœuds sont d'arité au plus n.

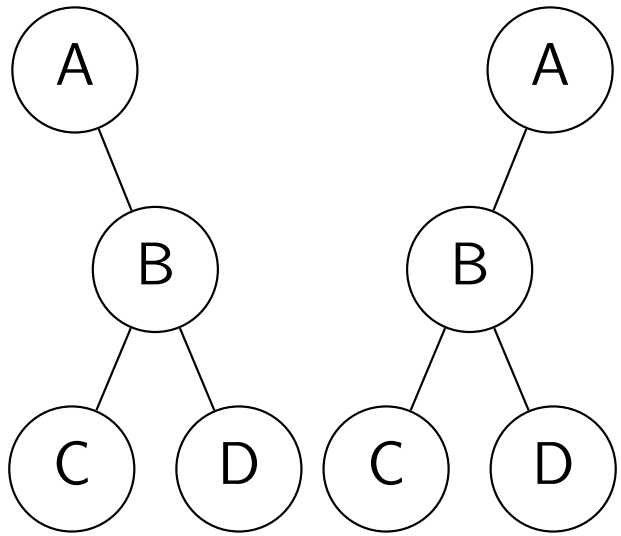
Définition 12 : Arbre binaire

Un arbre binaire est un arbre d'arité 2.

Définition 13 : Fils gauche, fils droit

Chaque enfant d'un nœud d'un arbre binaire est un **fils gauche** ou un **fils droit**. Il y a au plus un de chaque pour chaque nœud.

Cela sous-entends que les deux arbres suivants sont différents :



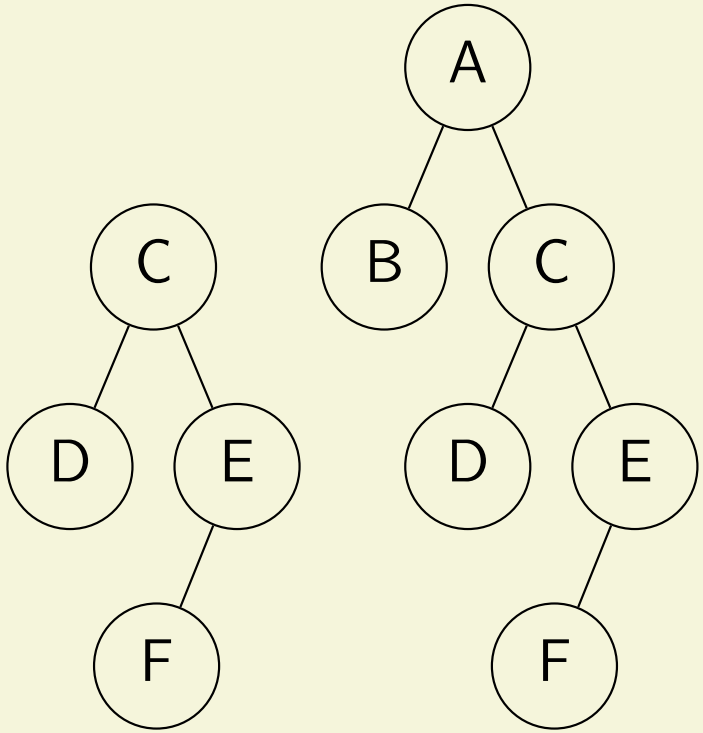
En effet, dans le premier cas, B est le fils droit de A, tandis que dans le second cas, B en est le fils gauche.

Définition 14 : Sous-arbre

Un **sous-arbre** a' d'un arbre a est un arbre dont la racine est un nœud n de a, et qui contient tous les descendant de n dans a.

Exemple 4 : Exemple de sous-arbre

L'arbre de gauche est un sous-arbre de l'arbre de droite :



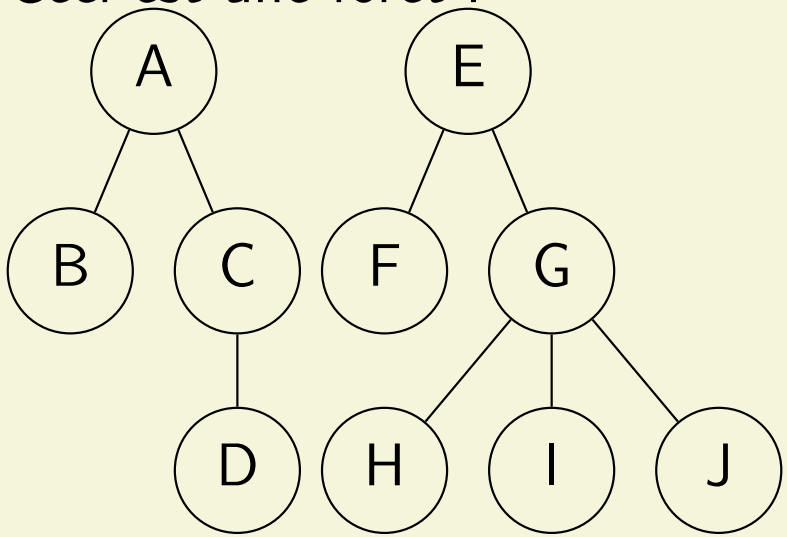
Exercice 2. Quels sont les sous arbres de l'arbre de racine A ?

Définition 15 : Forêt

Une **forêt** est une juxtaposition finie d'arbres.

Exemple 5 : Exemple de forêts

- Ceci est une forêt :



- La forêt vide est une forêt ;
- Un arbre est une forêt ;
- La juxtaposition de deux forêts est une forêt.

Définition 16 : Taille d'une forêt

La **taille d'une forêt** est la somme des tailles des arbres qui la compose.

Définition 17 : Profondeur d'une forêt

La **profondeur d'une forêt** est le maximum des profondeurs des arbres qui la compose.

Principe d'induction structurelle

Définition 18 : Définition inductive

On se donne des éléments de bases un ensemble d'éléments, et un ensemble de règles de construction qui partir d'un certain nombre d'éléments construisent un autre élément. Un ensemble **défini inductivement** (ou parfois **défini récursivement**) est un ensemble E construit comme le plus petit ensemble (au sens de l'inclusion) qui vérifie :

- E contient les éléments de bases ;
- E est stable pour toutes les règles de construction.

Exemple 6 : Construction des entiers

On peut construire les entiers de la manière suivante :

- 0 est un entier ;
- Si n est un entier, alors $n+1$ est un entier.

Exemple 7 : Construction des listes

On peut construire récursivement l'ensemble des listes d'entiers de la manière suivante :

- La liste vide `[]` est dans l'ensemble ;
- Si q est dans l'ensemble, alors pour tout $n \in \mathbb{Z}$, alors n :: q est dans l'ensemble.

Proposition 1 : Principe d'induction structurelle

Soit E un ensemble défini inductivement. Soit P une propriété. Si on a les hypothèse suivantes :

1. P est vraie pour tous les éléments de base de E ;
2. Pour chaque régle de construction, si P est vraie pour des éléments, alors, pour l'élément construit avec ces éléments et cette règle, P est vraie.

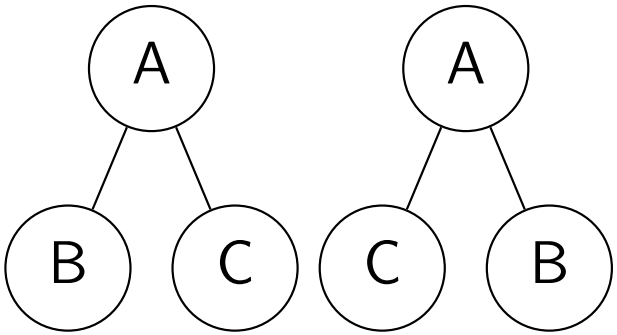
Alors, pour tout $x \in E$, $P(x)$ est vrai.

Définition 19 : Définition inductive des arbres quelconques

On définit les arbres étiquettés par l'ensemble E inductivement de la manière suivante :

- Les nœuds dont l'étiquette est dans E sont des arbres ;
- Pour chaque élément $e \in E$, et pour chaque ensemble fini et non vide d'arbres, on peut construire l'arbre dont la racine est un nœud étiquetté e et dont les enfants sont ces arbres.

Dans cette définition, les enfants sont considérés comme un ensemble d'arbres, et les deux arbres suivants sont donc les mêmes :



Définition 20 : Définition inductive des arbres quelconques avec des enfants ordonnés

On définit les arbres étiquettés par l'ensemble E inductivement de la manière suivante :

- Les nœuds dont l'étiquette est un élément de E ;
- Pour chaque élément $e \in E$, et pour chaque suite finie d'arbres a_0, a_1, ..., a_k (avec $k \geqslant 0$), on peut construire l'arbre dont la racine est un nœud étiquetté e et dont les enfants sont ces arbres dans l'ordre.

Définition 21 : Définition inductive des arbres binaires

On définit les arbres binaires étiquettés par l'ensemble E inductivement de la manière suivante :

- Les nœuds dont l'étiquette est un élément de E ;
- Pour chaque élément $e \in E$, pour chaque arbre a, on peut construire l'arbre dont la racine est étiquettée par e et dont l'enfant droit est a.
- Pour chaque élément $e \in E$, pour chaque arbre a, on peut construire l'arbre dont la racine est étiquettée par e et dont l'enfant gauche est a.
- Pour chaque élément $e \in E$, pour chaque pair d'arbres a et a', on peut construire l'arbre dont la racine est étiquettée par e et dont les enfants gauche et droit sont respectivement a et a'.

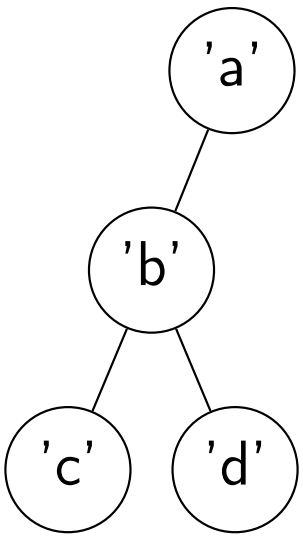
Propriétés sur les arbres binaires, implémentation d'arbres

```
1 type 'a arbre1 = Feuille of 'a | Noeudg of 'a arbre1
      | Noeudd of 'a arbre1 | Noeud2 of 'a arbre1 * 'a
      arbre1
```

```
1 type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a
    arbre
```

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a
    arbre
```

Exercice 3. Comment représenter avec ce type l'arbre suivant :



Exemple 8 : Construire un arbre dont les étiquettes ont été mutlipliées par deux

```
let rec multiplier_etiquette a = match a with
| Vide -> Vide
| Noeud(k, fg, fd) ->
    let nouveau_fg = multiplier_etiquette fg in
    let nouveau_fd = multiplier_etiquette fd in
    Noeud(k*2, nouveau_fg, nouveau_fd)
```

Exercice 4. On considère le type arbre suivant :

```
type arbre = Vide | Noeud of bool * arbre * arbre
```

Proposer une fonction de signature `arbre -> arbre` qui retire tous les nœuds (ainsi que les descendants) dont l'étiquette est `false`.

Proposition 2 : Hauteur d'un arbre

On peut définir la hauteur d'un arbre à l'aide de la quantité suivante :

- La hauteur d'une feuille est 0 ;
- La hauteur d'un élément construit à partir d'enfants est égale au maximum des hauteurs de ses enfants incrémentés de 1.

Exemple 9 : Calcul de la hauteur d'un arbre

```
1 let rec hauteur a = match a with
2 | Vide -> -1
3 | Noeud (_, gauche, droit) -> 1 + max (hauteur
    gauche) (hauteur droit)
```

Définition 22 : Taille d'un arbre

La **taille d'un arbre** est le nombre de nœud de cet arbre.

Proposition 3 : Taille d'un arbre

Une définition équivalente de la taille d'un arbre est la définition inductive suivante :

- La taille d'une feuille est 1 ;
- La taille d'un élément construit à partir d'enfants est égale à la somme des tailles de ses enfants incrémentées de 1.

Exemple 10 : Calcul de la taille d'un arbre en OCaml

```
let rec taille a = match a with
| Vide -> 0
| Noeud (_, gauche, droit) -> 1 + (taille
    gauche) + (taille droit)
```

Exercice 5. Proposer une fonction qui calcule la somme des étiquettes d'un arbre dont les étiquettes sont des entiers.

DM pour le 4 Mars

À rendre sur papier, le code écrit à la main, pour le **4 Mars** :

- Exercice 5 ;
- Exercice 36 ;
- Exercice 37.

Correction de définition

Définition 23 : Arité dans un noeud

L' **arité** d'un nœud est le nombre de ses enfants.

Correction de définition

Définition 23 : Arité dans un noeud

L' **arité** d'un nœud est le nombre de ses enfants.

Définition 24 : Arité d'un arbre

Un **arbre d'arité** n est un arbre dont tous les nœuds sont d'arité au plus n.

Correction de définition

Définition 23 : Arité dans un noeud

L' **arité** d'un nœud est le nombre de ses enfants.

Définition 24 : Arité d'un arbre

Un **arbre d'arité** n est un arbre dont tous les nœuds sont d'arité au plus n.

Définition 25 : Arbre binaire

Un arbre binaire est un arbre d'arité 2.

Définition 26 : Arbre complet

Un arbre binaire est dit **complet** si tous ses nœuds internes ont deux enfants, et que toutes ses feuilles ont la même profondeur.

Proposition 4 : Nombre maximum de nœuds d'une profondeur donnée

Un arbre binaire a au plus 2^l nœuds de profondeur l.
Ce maximum est atteint si tous les nœuds de niveau $k < l$ ont deux enfants.

Proposition 5 : Taille maximum d'un arbre binaire de profondeur donnée

Un arbre binaire de profondeur h a au plus $2^{h+1} - 1$ nœuds.
Ce maximum est atteint pour l'arbre complet de hauteur h.

Proposition 6 : Hauteur minimale d'un arbre binaire de taille donnée

Un arbre binaire de taille n a au moins une profondeur de $\lceil \log_2(n+1) \rceil - 1$.

Définition 27 : Arbre localement complet

Un arbre binaire est dit **localement complet** (ou parfois **binaire stricte**) si tous ses nœuds internes ont deux enfants.

Proposition 7 : Relation entre le nombre de nœuds internes et de feuilles dans un arbre localement complet

Pour tout arbre binaire localement complet avec n_i nœuds internes et n_f feuille, on a :

$$n_i + 1 = n_f$$

Exemple 11 : Implémentation des arbres binaire en C

```c
typedef struct _arbre {
    int valeur;
    struct _arbre * fg;
    struct _arbre * fd;
} arbre_t;
```

Où les pointeurs pour les fils gauches et droits `fg` et `fd` sont égaux à NULL quand le nœud n'a pas le fils correspondant.

Utilisation des arbres par un pointeur

```
1  int taille(arbre_t t){
2      int total = 1;
3      if (t.fg!=NULL){
4          total += taille(*t.fg);
5      }
6      if (t.fd!=NULL){
7          total += taille(*t.fd);
8      }
9      return total;
10 }
```

Utilisation des arbres par un pointeur

```
1  int taille(arbre_t t){
2      int total = 1;
3      if (t.fg!=NULL){
4          total += taille(*t.fg);
5      }
6      if (t.fd!=NULL){
7          total += taille(*t.fd);
8      }
9      return total;
10 }
```

```
1  int taille(arbre_t * t){
2      if (t==NULL){
3          return 0;
4      }
5      return 1 + taille(t->fg) + taille(t->fd);
6  }
```

Parcours d'arbre

Définition 28 : Parcours d'arbre

Un **parcours d'arbre** est un algorithme de visite et de traitement des nœuds d'un arbre.

Définition 28 : Parcours d'arbre

Un **parcours d'arbre** est un algorithme de visite et de traitement des nœuds d'un arbre.

L' **ordre** des nœuds obtenu à partir d'un parcours est l'ordre des nœud

Définition 29 : Parcours préfixe

Un **parcours d'arbre préfixe** est un parcours que l'on obtient en traitant le nœud lors de l'arrivée dans le nœud, et avant la visite des enfants.

Définition 29 : Parcours préfixe

Un **parcours d'arbre préfixe** est un parcours que l'on obtient en traitant le nœud lors de l'arrivée dans le nœud, et avant la visite des enfants.

Exemple 12 : Parcours préfixe en OCaml

```
let rec parcours_prefixe a = match a with
| Vide -> ()
| Noeud(e, fg, fd) ->
        print_string e;
        parcours_prefixe fg;
        parcours_prefixe fd
```

Définition 30 : Parcours postfixe

Un **parcours d'arbre postfixe** est un parcours que l'on obtient en traitant le nœud après la visite des enfants.

Définition 30 : Parcours postfixe

Un **parcours d'arbre postfixe** est un parcours que l'on obtient en traitant le nœud après la visite des enfants.

Exemple 13 : Parcours postfixe en OCaml

```
let rec parcours_postfixe a = match a with
| Vide -> ()
| Noeud(e, fg, fd) ->
        parcours_postfixe fg;
        parcours_postfixe fd
        print_string e;
```

Définition 31 : Parcours infixe

Un **parcours d'arbre infixe** est un parcours que l'on obtient en traitant le nœud entre la visite du fils gauche et le fils droit.

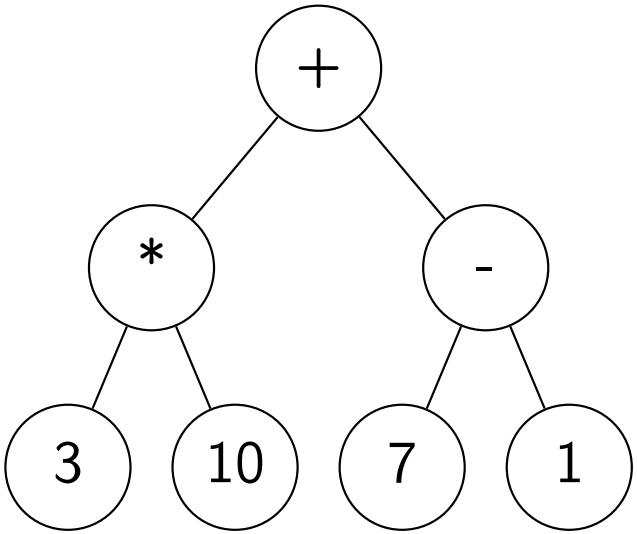
Définition 31 : Parcours infixe

Un **parcours d'arbre infixe** est un parcours que l'on obtient en traitant le nœud entre la visite du fils gauche et le fils droit.

Exemple 14 : Parcours infixe en Ocaml

```
let rec parcours_infixe a = match a with
| Vide -> ()
| Noeud(e, fg, fd) ->
        parcours_infixe fg;
        print_string e;
        parcours_infixe fd
```

Exercice 6. Quels sont les ordres infixes, préfixes et suffixes pour l'arbre suivant :



Définition 32 : Parcours d'arbre par niveau

Un **parcours par niveau** est un parcours d'arbre qui parcourt les arbres par profondeur croissante.

Définition 32 : Parcours d'arbre par niveau

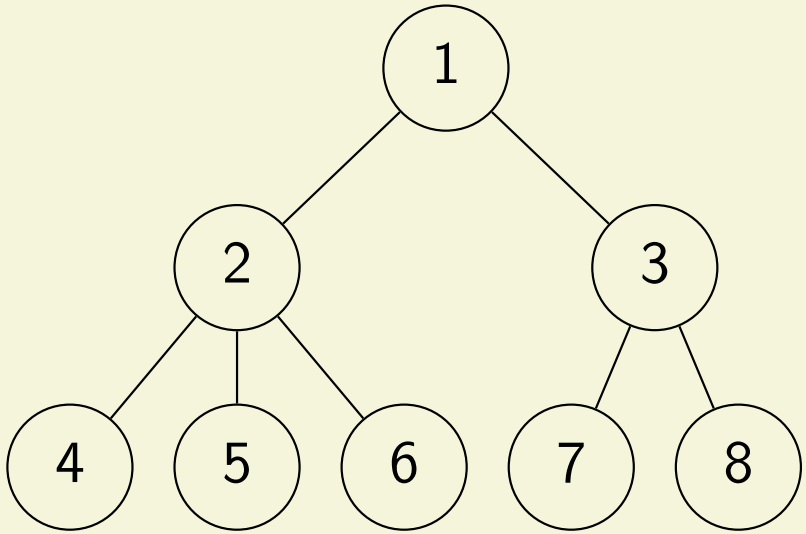
Un **parcours par niveau** est un parcours d'arbre qui parcourt les arbres par profondeur croissante.

Définition 33 : Ordre par niveau

L' **ordre par niveau** est l'ordre des nœuds obtenu par un parcours par niveau.

Exemple 15 : Ordre des nœuds par niveau

Dans l'arbre suivant, les nœuds sont ordonnés par ordre croissant dans l'ordre par niveau.



Algorithme 1 : Parcours par niveau

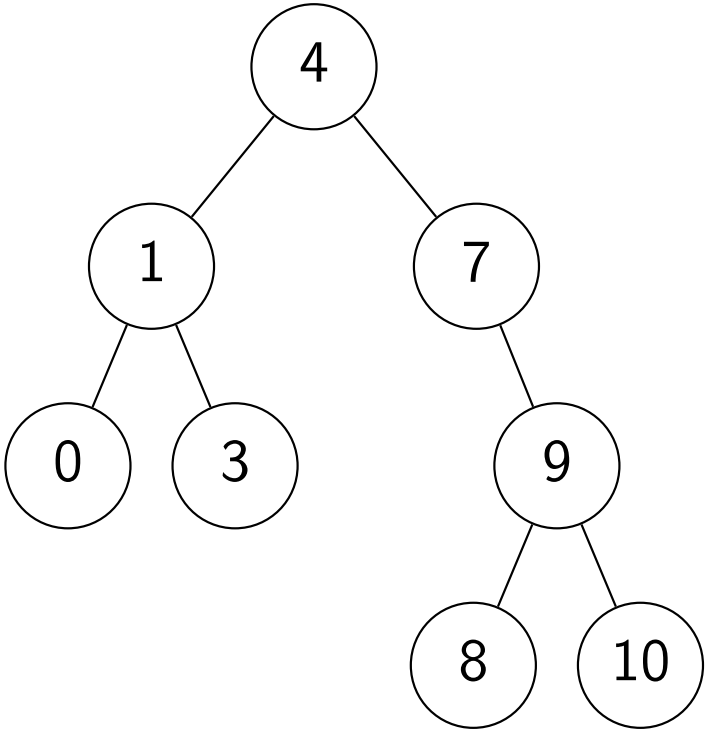
file ← une file qui contient la racine
Tant Que La file n'est pas vide **Faire**
    e ← le premier élément que l'on prend de la file.
    Afficher *e*
    Ajouter tous les enfants de *e* à *file*.

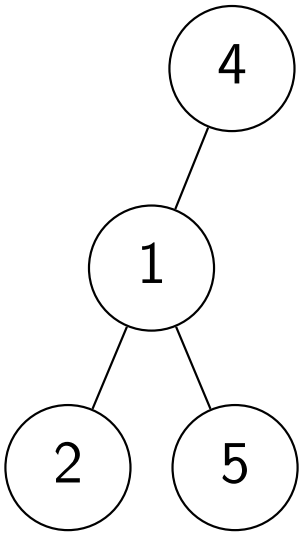
Algorithme 1 : Parcours par niveau

file ← une file qui contient la racine
Tant Que La file n'est pas vide **Faire**
    e ← le premier élément que l'on prend de la file.
    Afficher *e*
    Ajouter tous les enfants de *e* à *file*.

Exercice 7. Implémenter un parcours d'arbre binaire par niveau en utilisant une file qui affiche les étiquettes des nœuds dans leur ordre par niveau.
On pourra utiliser le module Queue.

Arbres binaires de recherche


4
1
7
0
3
9
8
10


4
1
2
5

Définition 34 : Relation d'ordre

Une relation d'ordre sur un ensemble E est une relation binaire $\leqslant$ qui vérifie les propriétés suivantes :

$$\forall x \in E, x \leqslant x \qquad \text{(Reflexivité)}$$

$$\forall x, y \in E^2, (x \leqslant y \wedge y \leqslant x) \Rightarrow x = y \qquad \text{(Antisymmétrie)}$$

$$\forall x, y, z \in E^3, (x \leqslant y \wedge y \leqslant z) \Rightarrow x \leqslant z \qquad \text{(Transitivité)}$$

Définition 35 : Ensemble totalement ordonné

Un ensemble muni d'une relation d'ordre $(E, \leqslant)$ est dit totalement ordonné si pour tout $x, y \in E$, on a $x \leqslant y$ ou $y \leqslant x$ (ou les deux).

Définition 36 : Arbre binaire de recherche

Soit $(E, \leqslant)$ un ensemble totalement ordonné. Un arbre binaire est un **binaire de recherche** lorsque, pour tous nœud n d'étiquette e :

- Soit n n'as pas de fils gauche, soit pour tout nœud du sous-arbre gauche d'étiquette e', $e' \leqslant e$.
- Soit n n'as pas de fils droit, soit pour tout nœud du sous-arbre droit d'étiquette e', $e \leqslant e'$.

Définition 36 : Arbre binaire de recherche

Soit $(E, \leqslant)$ un ensemble totalement ordonné. Un arbre binaire est un **binaire de recherche** lorsque, pour tous nœud n d'étiquette e :

- Soit n n'as pas de fils gauche, soit pour tout nœud du sous-arbre gauche d'étiquette e', $e' \leqslant e$.
- Soit n n'as pas de fils droit, soit pour tout nœud du sous-arbre droit d'étiquette e', $e \leqslant e'$.

Exercice 8. Construire un arbre binaire de recherche avec les nombres de 1 à 12. On cherchera à minimiser la hauteur de l'arbre.

Algorithme 2 : Recherche dans un arbre binaire de recherche

Entrée : A un arbre, x une étiquette

```
r ← la racine de a.
Tant Que vrai Faire
    Si l'étiquette de r est égale à x Alors
        Renvoyer Vrai
    Si l'étiquette de r est supérieure strictement à x Alors
        Si r a un fils droit Alors
            r ← le fils droit de r.
        Sinon
            Renvoyer Faux
    Sinon
        Si r a un fils gauche Alors
            r ← le fils gauche de r.
        Sinon
            Renvoyer Faux
```

Exemple 16 : Recherche dans un arbre binaire de recherche en OCaml

```
let rec chercher a x = match a with
| Vide -> False
| Noeud(e, fg, fd) ->
        if e = x then true
        else if e>x then chercher fg x
        else chercher fd x
```

Exemple 16 : Recherche dans un arbre binaire de recherche en OCaml

```
let rec chercher a x = match a with
| Vide -> False
| Noeud(e, fg, fd) ->
        if e = x then true
        else if e>x then chercher fg x
        else chercher fd x
```

Exercice 9. Proposer une fonction pour trouver le minimum dans un arbre binaire de recherche.

Proposition 8 : Complexité temporelle de la recherche

La recherche dans un arbre binaire de recherche a une complexité temporelle en $O(h)$ où h est la hauteur de l'arbre dans le pire des cas.
La recherche dans un arbre binaire de recherche a une complexité temporelle en $O(n)$ où n est la taille de l'arbre dans le pire des cas.

Définition 37 : Arbre presque complet côté gauche

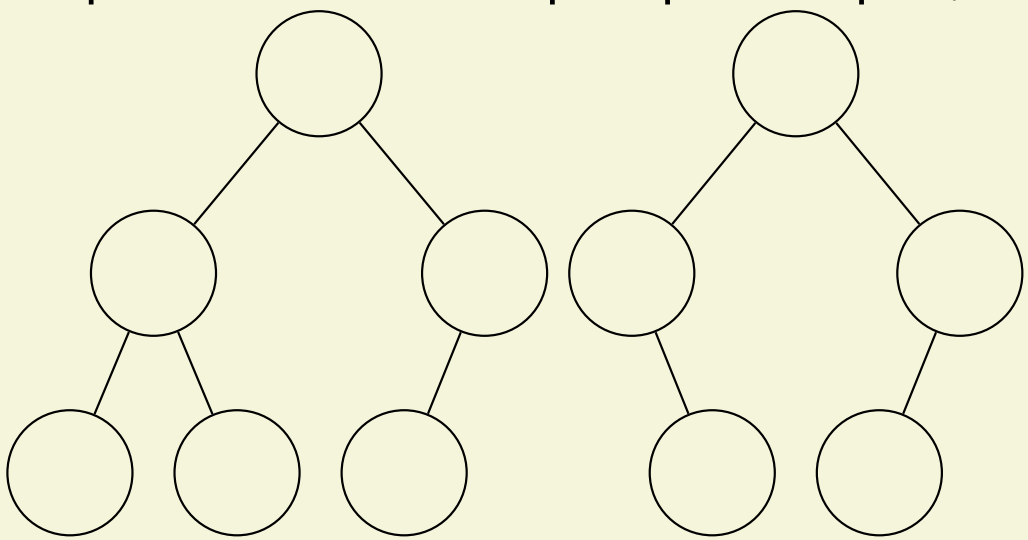
On dit qu'un arbre binaire est **presque complet côté gauche** (ou parfois juste **presque complet**) lorsque tout ses niveaux sont complets, sauf potentiellement le dernier dont tous les nœuds sont alignés à gauche.

Définition 37 : Arbre presque complet côté gauche

On dit qu'un arbre binaire est **presque complet côté gauche** (ou parfois juste **presque complet**) lorsque tout ses niveaux sont complets, sauf potentiellement le dernier dont tous les nœuds sont alignés à gauche.

Exemple 17 : Arbre presque complet

Le première arbre est presque complet, mais pas le deuxième.



Proposition 9 : Hauteur d'un arbre presque complet

Un arbre presque complet de taille n est de profondeur $\lceil \log_2(n+1) \rceil - 1$.

Proposition 9 : Hauteur d'un arbre presque complet

Un arbre presque complet de taille n est de profondeur $\lceil \log_2(n+1) \rceil - 1$.

Proposition 10 : Recherche dans un arbre binaire presque complet

La complexité temporelle de la recherche dans un arbre presque complet de taille n est en $O(\log(n))$.

Algorithme 3 : Ajout dans un arbre binaire de recherche

Entrée : A un arbre, x une étiquette

```
r ← la racine de a.
Tant Que vrai Faire
    Si l'étiquette de r est supérieure strictement à x Alors
        Si r a un fils droit Alors
            r ← le fils droit de r.
        Sinon
            Le fils droit de r devient la feuille d'étiquette x.
            Renvoyer a
    Sinon
        Si r a un fils gauche Alors
            r ← le fils gauche de r.
        Sinon
            Le fils gauche de r devient la feuille d'étiquette
            x.
            Renvoyer a
```

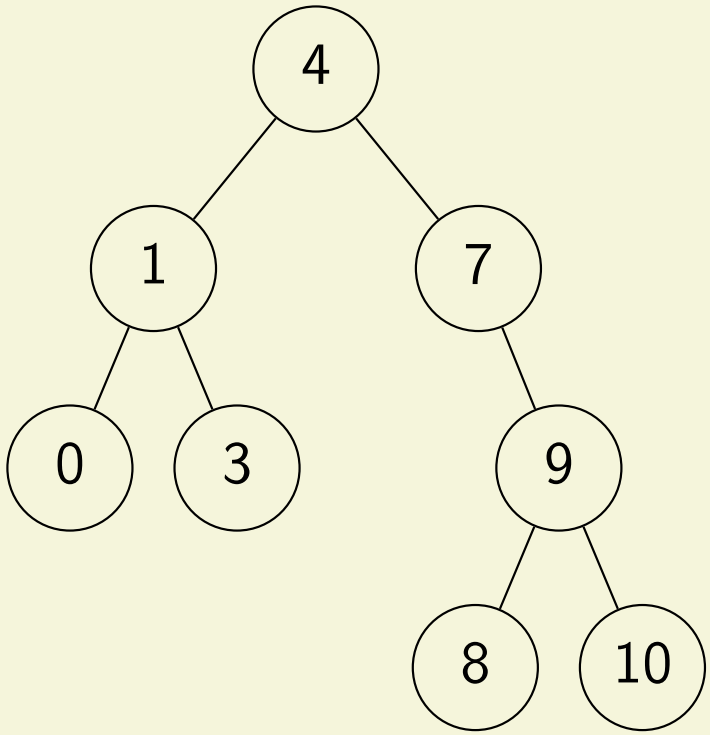
Exercice 10. Proposer une fonction OCaml de signature `'a arbre -> 'a -> 'a arbre` qui calcule l'arbre binaire de recherche après l'ajout d'un élément en entrée.

Définition 38 : Prédécesseur, successeur dans un arbre binaire de recherche

Le *prédécesseur* d'un nœud n dans un arbre binaire de recherche est le nœud dont l'étiquette est la plus grande parmi les nœuds dont l'étiquette est plus petite que celle de n.
Le *successeur* d'un nœud n dans un arbre binaire de recherche est le nœud dont l'étiquette est la plus petite parmi les nœuds dont l'étiquette est plus grande que celle de n.

Exemple 18 : Prédecesseur, successeur dans un ABR

Dans l'arbre suivant, le prédecesseur du nœud d'étiquette 7 est la racine d'étiquette 4. Le sucesseur du nœud d'étiquette 1 est le nœud d'étiquette 3. Le successeur du nœud d'étiquette 3 est le nœud d'étiquette 4.



Algorithme 4 : Recherche d'un successeur dans un arbre binaire de recherche

Entrée : Arbre a, noeud n

  Si n a un enfant droit **Alors**

    Renvoyer Minimum du sous-arbre droit de n

  Tant Que n est un enfant droit **Faire**

    $n \leftarrow$ parent de n

  Renvoyer n

Attention à ne pas faire des calculs dans le vent

```
let rec ordre_postfixe acc a = match a with
| Vide -> ...
| Noeud(x, fg, fd) ->
    x::acc ;
    ordre_postfixe acc fg ;
    ordre_postfixe acc fd
```

Attention à ne pas faire des calculs dans le vent

```
let rec ordre_postfixe acc a = match a with
| Vide -> ...
| Noeud(x, fg, fd) ->
    x::acc ;
    ordre_postfixe acc fg ;
    ordre_postfixe acc fd
```

```
        let rec ordre_postfixe acc a = match a with
| Vide -> ...
| Noeud(x, fg, fd) ->
    let acc2 = x::acc in
    let acc3 = ordre_postfixe acc2 fg in
    ordre_postfixe acc3 fd
```

Exercice 11. Implémenter la fonction de retrait en OCaml.

Implémentation du retrait (1)

On commence par se donner une fonction qui cherche l'élément à retirer :

```
let rec retirer a x = match a with
| Vide -> failwith "Element absent"
| Noeud(y, _, _) when x = y -> retirer_racine a
| Noeud(y, fg, fd) ->
    if x > y then
        Noeud(y, fg, retirer fd x)
    else
        Noeud(y, retirer fg x, fd)
```

Implémentation du retrait (2)

Il faut ensuite une fonction qui retire la racine :

```
let retirer_racine a = match a with
| Vide -> failwith "Element absent"
| Noeud(_, Vide, fd) -> fd
| Noeud(_, fg, Vide) -> fg
| Noeud(_, fg, fd) ->
    let successeur, reste = successeur_et_reste fg
    in
    Noeud(successeur, reste, fd)
```

Implémentation du retrait (3)

Enfin, il nous faut une fonction qui à partir de l'arbre droit calcule l'étiquette du successeur dans l'arbre droit, ainsi que l'arbre une fois qu'on a retiré le successeur. Cette fonction est plus facile qu'un retrait normal car le successeur n'a nécessairement pas d'enfant gauche.

```
1 let rec successeur_et_reste a = match a with
2 | Vide -> failwith "arbre vide"
3 | Noeud(y, Vide, fd) -> y, fd
4 | Noeud(y, fg, fd) ->
5     let successeur, reste = successeur_et_reste fd
      in
6     successeur, Noeud(y, reste, fd)
```

Proposition 11 : Complexité des opérations dans un arbre binaire de recherche

On note n la taille de notre arbre, et h sa hauteur. Les complexités temporelle des opération sur un arbre binaire de recherche sont données par le tableau suivant :

Recherche	Ajout	Retrait
$O(h)$, $O(n)$	$O(h)$, $O(n)$	$O(h)$, $O(n)$

Définition 39 : Table d'association

Une **table d'association** (ou un **dictionnaire**) sur un ensemble de clefs E et d'éléments F est une structure dans laquelle il est possible de stocker des pairs d'éléments de $E \times F$ et de trouver ou retirer rapidement un élément à partir d'une clef e.
Dans une **table d'association**, pour chaque $e \in E$, il existe au plus une paire (e, f).

Exemple 19 : Table d'association étiquettée par des entiers en OCaml

On peut utiliser le type suivant en OCaml pour stocker des éléments dans une table d'association dont les clefs sont des entiers :

```
1 type 'a table = Vide | Feuille of int * 'a * 'a
      table * 'a table
```

Exemple 19 : Table d'association étiquettée par des entiers en OCaml

On peut utiliser le type suivant en OCaml pour stocker des éléments dans une table d'association dont les clefs sont des entiers :

```
type 'a table = Vide | Feuille of int * 'a * 'a table * 'a table
```

Exercice 12. Proposer une fonction OCaml qui ajoute un élément dans un tableau d'association.

Définition 40 : Arbre bicolore

Un **arbre bicolore**

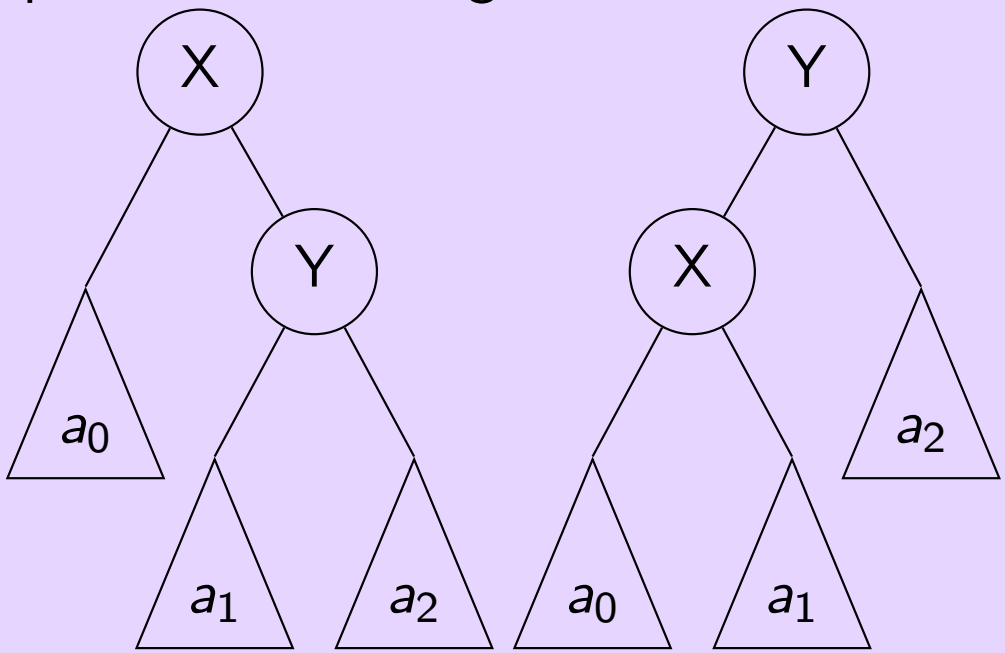
- Un nœud est soit noir soit rouge ;
- La racine est noire ;
- Tous les nœud ont 2 enfants qui peuvent être des nœud internes ou des feuilles ;
- Les étiquettes sont portées par les nœuds internes ;
- Les feuilles sont noires ;
- Les enfants d'un nœud rouge sont noirs ;
- Le chemin de la racine à toutes les feuilles contient le même nombre de nœuds noirs.

Proposition 12 : Recherche dans un arbre rouge-noir

La recherche dans un arbre de recherche rouge-noir de taille n s'exécute dans le pire des cas en $O(\log n)$.

Définition 41 : Rotations Gauche et Droite

La **rotation gauche** est la transformation de l'arbre de gauche en l'arbre de droite, c'est-à-dire la transformation d'un arbre d'étiquette X d'enfant droit d'étiquette Y en l'arbre d'étiquette Y d'enfant gauche X.



La **rotation droite** est la transformation inverse.

Proposition 13 : Respect des propriétés des arbres binaires de recherches

Les rotations sur un nœud d'un arbre binaire de recherche donnent des arbres qui respectent les propriétés des arbres binaires de recherche.

Algorithme 5 : Ajout dans un arbre rouge-noir

- **Le nœud ajouté est la racine :** il suffit de colorier ce nœud en noir.
- **Le parent du nœud inséré est noir :** il n'y a pas de modifications à faire, l'arbre vérifie déjà les propriétés attendues.
- **Le parent et l'oncle sont rouges :** l'oncle et le parent sont coloriés en noire et le grand-parent est colorié en rouge s'il était noir. Dans le cas où le grand parent était colorié en noir initialement, il faut appliquer récursivement la rectification de la structure à partir de cet élément.

Algorithme 5 : Ajout dans un arbre rouge-noir

- **Le parent est rouge, mais l'oncle est noir, et le nœud est un petit-enfant interne :** on fait une rotation pour se ramener au cas suivant.
- **Le parent est rouge, mais l'oncle est noir, et le nœud est un petit-enfant externe :** le parent est ramené à la position du grand parent par une rotation, et le parent change de couleur pour noir, tandis que le grand parent devient rouge.

Proposition 14 : Complexité des opérations dans un arbre bicolore

On note n la taille de notre arbre, et h sa hauteur. Les complexités temporelle des opération sur un arbre bicolore sont données par le tableau suivant :

Recherche	Ajout	Retrait
$O(h)$, $O(\log n)$	$O(h)$, $O(\log n)$	$O(h)$, $O(\log n)$

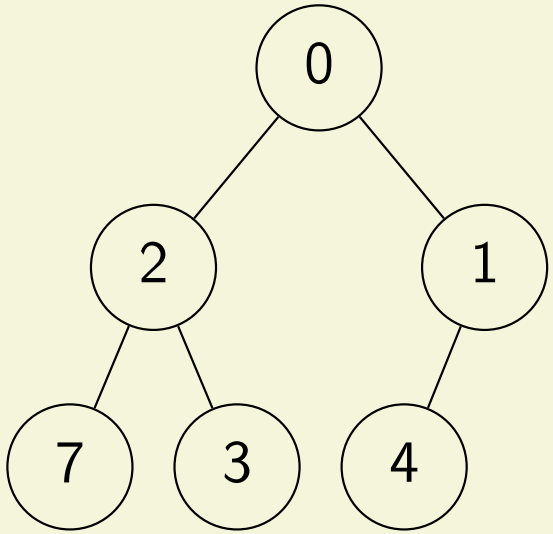
Tas et files de priorité

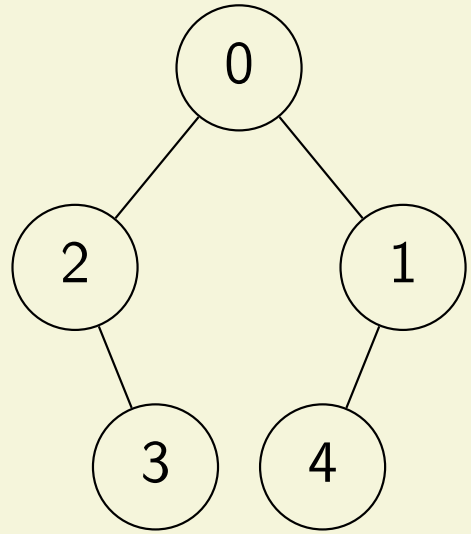
Définition 42 : Tas binaire

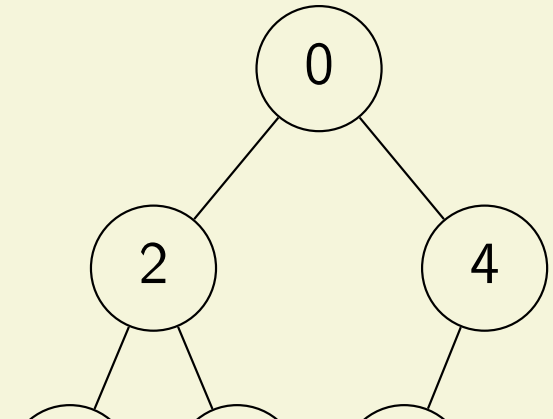
Un **tas min binaire** sur un ensemble totalement ordonné $(E, \leqslant)$ est une structure d'arbre binaire presque complet telle que pour tout nœud interne, l'étiquette du nœud est plus petite que toutes les étiquettes de ses enfants.
De même, on peut définir un **tas max binaire** comme une structure d'arbre binaire presque complet telle que pour tout nœud interne, l'étiquette du nœud est plus grande que toutes les étiquettes de ses enfants.

Exemple 20 : Tas

Parmi les arbres suivants, le premier est un tas min, le deuxième n'est pas un tas car il n'est pas presque complet, et enfin, le dernier n'est pas un tas min, car il ne respecte pas la propriété sur les étiquettes.







Retour de vacances

- Je ramasse les DM ;
- Programme de colle avec un peu de retard, je ferai mieux la semaine prochaine ;
- DS dans une semaine.

Programme du DS de la semaine prochaine

- Chaînes de caractère, flottants ;
- Arbres ;
- Induction, ordres bien fondés.

Des nouvelles de la classe mobile

Les PC linux ne devraient plus trop tarder.

Correction exercice 19

On se donne deux ensembles E et F d'intersection nulle, et on considère A l'ensemble des arbres localement complets dont les étiquettes des nœuds internes sont dans E et les étiquettes des feuilles sont dans F.

1. Montrer qu'il existe au plus un unique arbre $a \in A$ qui puisse avoir un ordre postfixe donné.
2. Montrer qu'il existe au plus un unique arbre $a \in A$ qui puisse avoir un ordre prefixe donné.
3. Montrer qu'avec un ordre infixe donné, il peut y avoir plusieurs arbres dans A qui ont cet ordre infixe.

Correction de l'exercice d'implémentation d'un tas à l'aide d'un tableau en C

Implémenter un tas min à l'aide d'un tableau dont le premier élément contient le nombre d'élément stocké.

```c
void ajouter(int * tas, int taille, int x)
```

```c
int retirer_racine(int * tas, int taille)
```

```c
void ajouter(int * tab, int taille, int x){
    int actuel;
    int temporaire;

    assert(tab[0]+1<taille);

    tab[0] = tab[0] + 1;
    tab[tab[0]] = x;
    actuel = tab[0];

    while (actuel>1){
        if (tab[actuel] < tab[actuel/2]){
            temporaire = tab[actuel];
            tab[actuel] = tab[actuel/2];
            tab[actuel/2] = temporaire;
            actuel = actuel / 2;
        }
        else{
            return;
        }
    }
}
```

Ordres bien fondés

Définition 43 : Prédécesseur, sucesseur

Un *prédécesseur* de x est un élément y tel que $y \leqslant x$.
Un *successeur* de x est un élément y tel que $x \leqslant y$.

Définition 44 : Prédécesseur et successeur immédiat

Un *prédécesseur immédiat* de x est un prédécesseur de x distinct de x, y, tel qu'il n'existe pas de z distinct de x et y tel que $y \leqslant z \leqslant x$.
Un *successeur immédiat* de x est un successeur de x distinct de x, y, tel qu'il nexiste pas de z distinct de x et y tel que $x \leqslant z \leqslant y$.

Définition 45 : Élément minimal

Un *élément minimal* est un élément tel qu'il n'existe pas de y distinct de x tel que $y \leqslant x$.

Définition 46 : Ordre bien fondé

Un *ordre bien fondé* sur un ensemble E est une relation d'ordre telle qu'il n'existe pas de suite décroissante strictement infinie dans E au sens de cet relation.

Définition 47 : Ensemble inductif

Un ensemble est dit *inductif* quand il dispose d'une relation d'ordre bien fondé.

Proposition 15 : Ensemble construit par induction

Soit un sous-ensemble $B \subset E$ qu'on nomme les éléments de base, et un ensemble de fonctions $R = (r_i)_i$ tels que r_i soit une fonction d'arité a_i qui prend ses éléments arguments dans E et qui est à valeur dans E.
Il existe un petit (au sens de l'inclusion) ensemble F qui vérifie les propriétés suivantes :

- $B \subset F$;
- F est stable par chacun des $r_i \in R$.

On dit que F est **construit par induction avec les éléments de B et des règles R.**

Définition 48 : Ensemble construit par induction sans ambiguïté

On dit qu'un ensemble F construit par induction est **construit sans ambiguïté** si, pour chaque élément $e \in F$:

- Si e est un élément de base, alors il ne peut pas être construit avec une règle de construction ;
- Si e n'est pas un élément de base, alors il peut être construit avec exactement une règle r_i et un unique a_i-uplets d'éléments de F.

Proposition 16 : Ordre induit

Soit F ensemble **construit sans ambiguïté**. On définit **l'ordre induit** comme étant la relation suivante $\leqslant$ sur F par, pour tout f et f', $f \leqslant f'$ si et seulement si f' s'écrit avec f dans son unique construction.
Cet ordre induit est une relation d'ordre bien fondée.

Exercice 13. Montrer qu'un élément de base est un élément minimal pour l'ordre induit.

Proposition 17 : Fonction définie sur un ensemble construit par induction sans ambiguïté

Soit E' un ensemble. Soit F un ensemble construit sans ambiguïté.
On se donne une fonction f_i de E^{a_i} dans E' pour chaque règle r_i, et une valeur $e'_j \in E'$ pour tout élément de base e_j. Il existe exactement une fonction f telle que :

- $f(e_j) = e'_j$ pour chaque $e_j \in E$;
- Pour tout r_i, et tout a_i-uplets $b_1, ..., b_{a_i}$ de E, on a :

$$f(r_i(b_1, ..., b_{a_i})) = f_i(f(b_1), ..., f(b_{a_i}))$$