

# Arbres

Algorithmique et structures de données, 2023-2024

---

P. Kunzli (Cloud) et O. Malaspinas (A401), ISC, HEPIA

2024-03-05

En partie inspirés des supports de cours de P. Albuquerque

{Les arbres}

## Les arbres: définition

“Un arbre est un graphe acyclique orienté possédant une unique racine, et tel que tous les nœuds sauf la racine ont un unique parent.”

# Les arbres: définition

“Un arbre est un graphe acyclique orienté possédant une unique racine, et tel que tous les nœuds sauf la racine ont un unique parent.”

## Santé!

### Plus sérieusement

- Ensemble de **nœuds** et d'**arêtes** (graphe),
- Les arêtes relient les nœuds entre eux, mais pas n'importe comment: chaque nœud a au plus un **parent**,
- Le seul nœud sans parent est la **racine**,
- Chaque nœud a un nombre fini d'**enfants**,
- La hiérarchie des nœuds rend les arêtes **orientées** (parent -> enfants), et empêche les **cycles** (acyclique, orienté).
- La **feuille** ou **nœud terminal** est un nœud sans enfants,
- Le **niveau** est 1 à la racine et **niveau+1** pour les enfants,
- Le **degré** d'un nœud est le nombre de enfants du nœud.

# Les arbres: définition

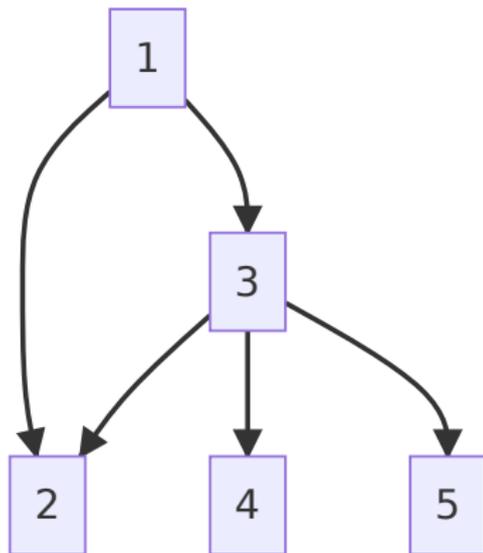
“Un arbre est un graphe acyclique orienté possédant une unique racine, et tel que tous les nœuds sauf la racine ont un unique parent.”

## Santé!

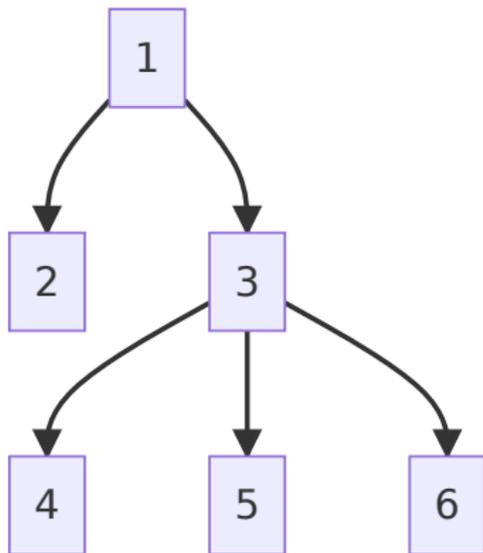
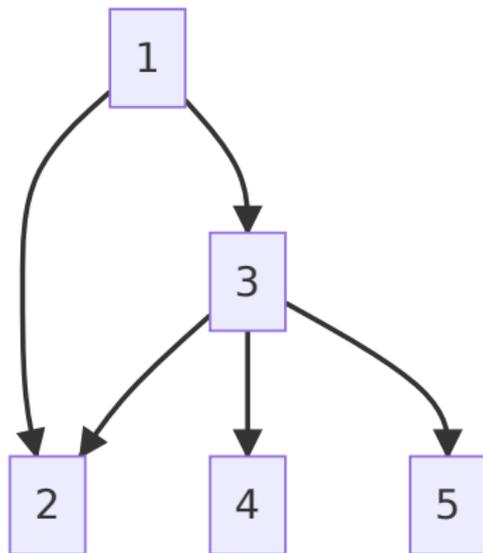
### Plus sérieusement

- Ensemble de **nœuds** et d'**arêtes** (graphe),
- Les arêtes relient les nœuds entre eux, mais pas n'importe comment: chaque nœud a au plus un **parent**,
- Le seul nœud sans parent est la **racine**,
- Chaque nœud a un nombre fini d'**enfants**,
- La hiérarchie des nœuds rend les arêtes **orientées** (parent -> enfants), et empêche les **cycles** (acyclique, orienté).
- La **feuille** ou **nœud terminal** est un nœud sans enfants,
- Le **niveau** est 1 à la racine et **niveau+1** pour les enfants,
- Le **degré** d'un nœud est le nombre de enfants du nœud.
- Chaque nœud est un arbre en lui même.
- La **récurtivité** sera très utile!

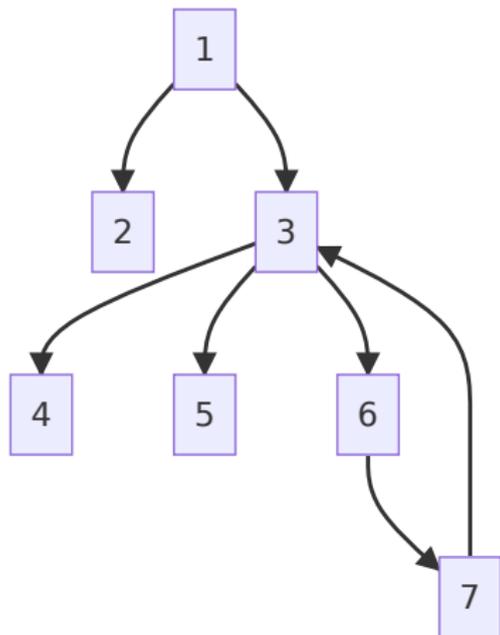
## Arbre ou pas arbre?



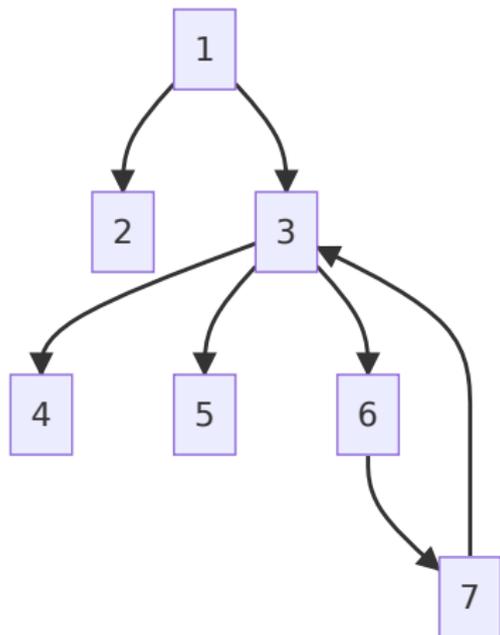
## Arbre ou pas arbre?



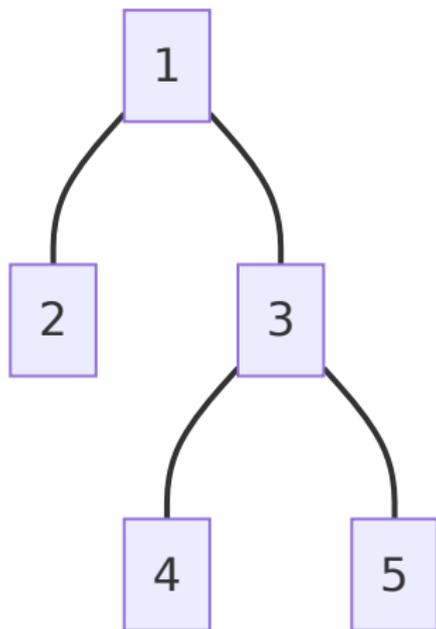
## Arbre ou pas arbre?



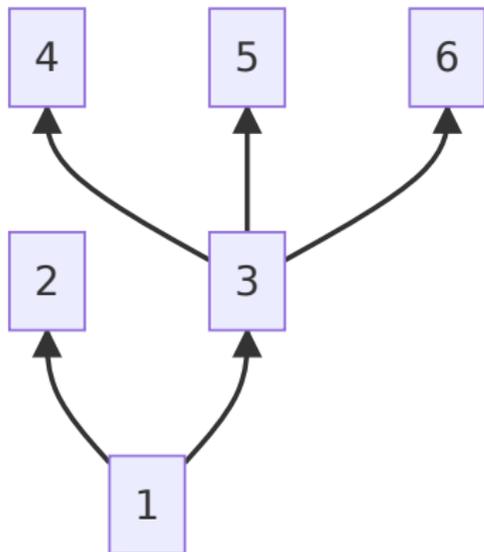
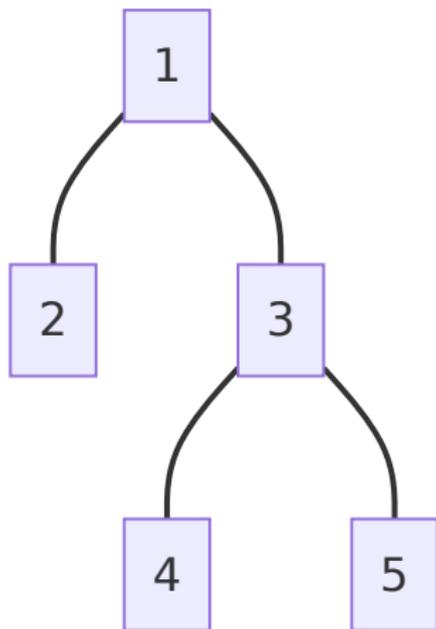
# Arbre ou pas arbre?



## Arbre ou pas arbre?

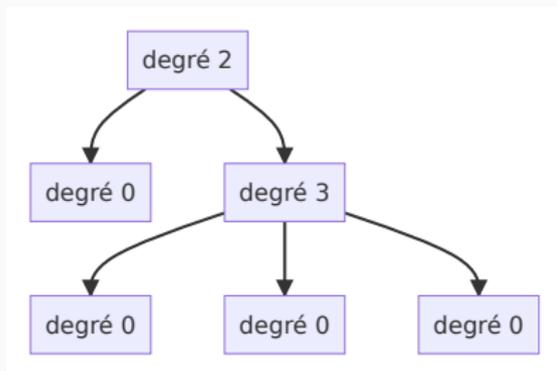


## Arbre ou pas arbre?



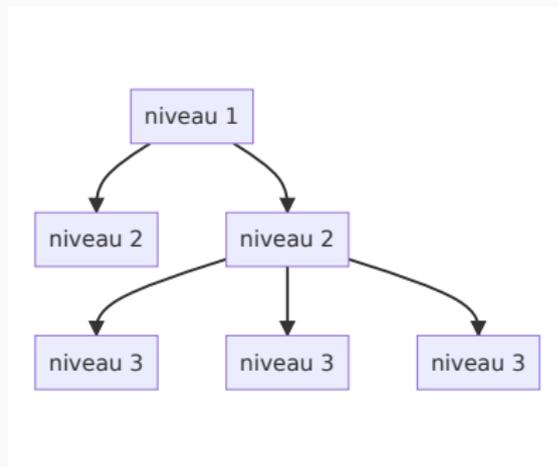
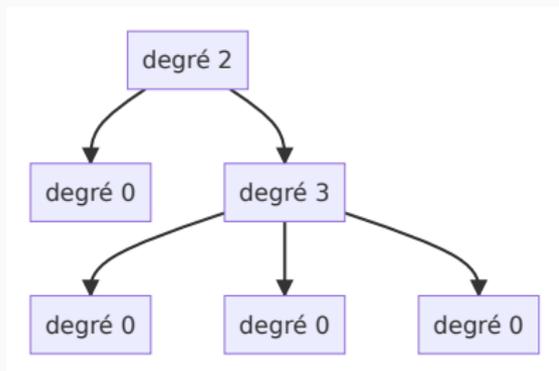
# Degré et niveau

- Illustration du degré (nombre d'enfants) et du niveau (profondeur)



# Degré et niveau

- Illustration du degré (nombre d'enfants) et du niveau (profondeur)



- Les nœuds de degré 0, sont des feuilles.

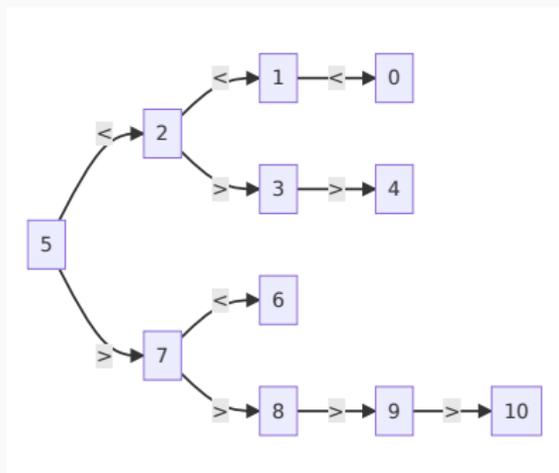
## Application: recherche rapide

Pouvez vous construire un arbre pour résoudre le nombre secret?

# Application: recherche rapide

Pouvez vous construire un arbre pour résoudre le nombre secret?

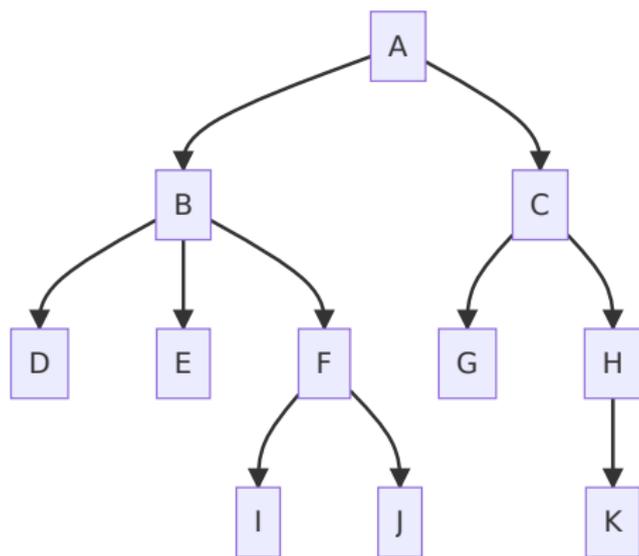
- Le nombre secret ou la recherche dichotomique (nombre entre 0 et 10).



**Question:** Quelle est la complexité pour trouver un nombre?

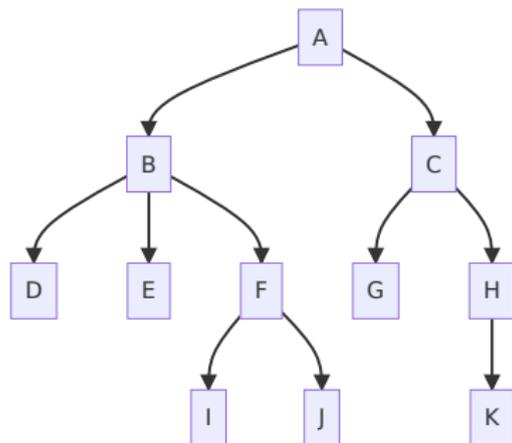
## Autres représentation

- Botanique
- **Exercice:** Ajouter les degrés/niveaux et feuilles



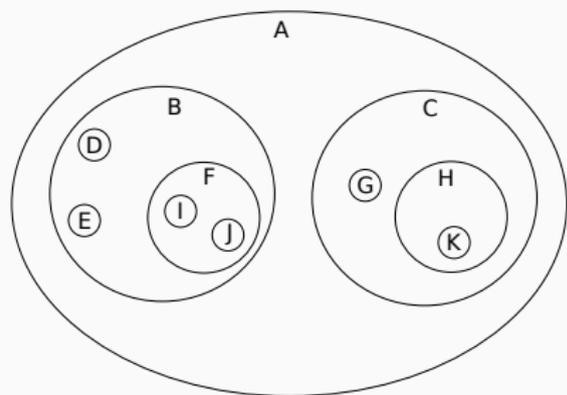
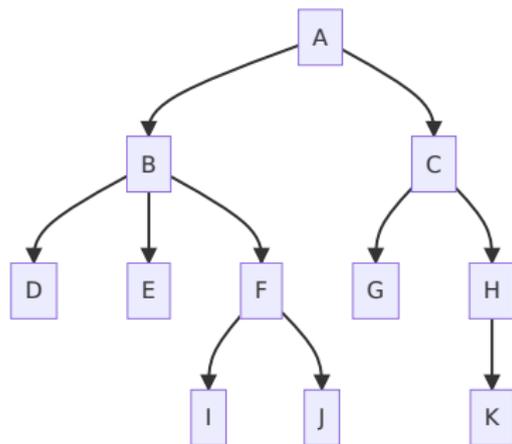
# Autres représentation

- Ensembliste



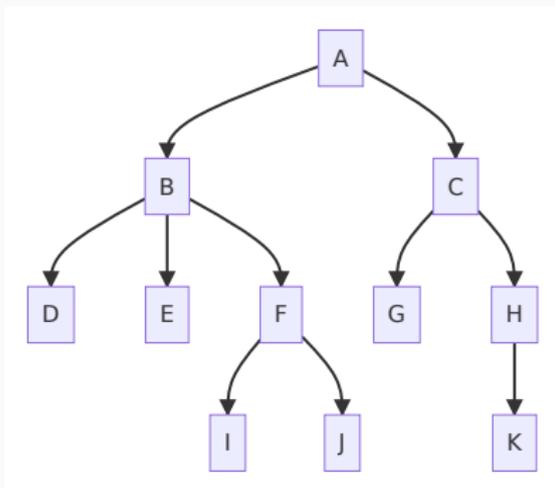
# Autres représentation

- Ensembliste



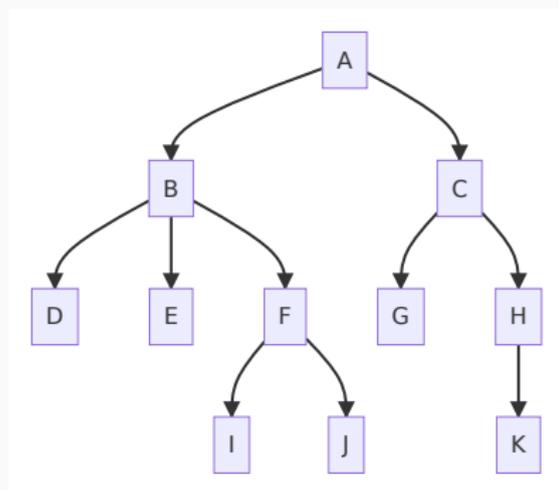
# Autres représentation

- Liste



# Autres représentation

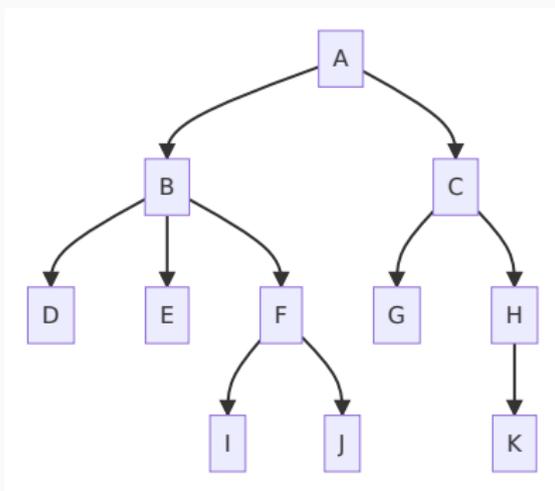
- Liste



(A  
  (B  
    (D)  
    (E)  
    (F  
      (I)  
      (J)  
    )  
  )  
  (C  
    (G)  
    (H  
      (K)  
    )  
  )  
)

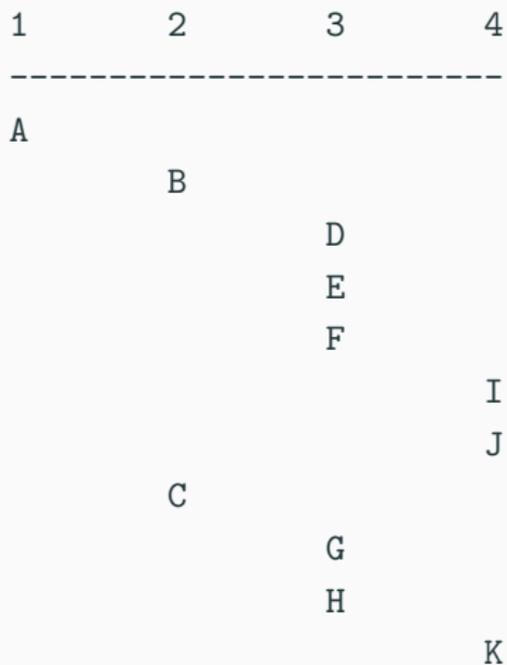
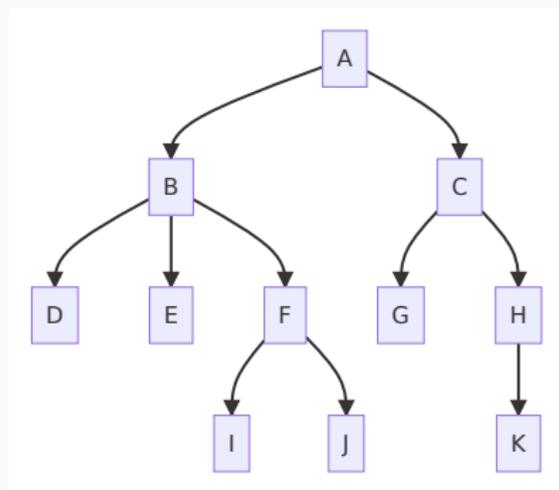
# Autres représentation

- Par niveau



# Autres représentation

- Par niveau



# L'arbre binaire

- Structure de données abstraite,
- Chaque nœud a au plus deux enfants: gauche et droite,
- Chaque enfants est un arbre.

**Comment représenteriez vous une telle structure?**

# L'arbre binaire

- Structure de données abstraite,
- Chaque nœud a au plus deux enfants: gauche et droite,
- Chaque enfants est un arbre.

**Comment représenteriez vous une telle structure?**

`<R, G, D>`

R: racine

G: sous-arbre gauche

D: sous-arbre droite

**Comment cela s'écrirait en C?**

# L'arbre binaire

- Structure de données abstraite,
- Chaque nœud a au plus deux enfants: gauche et droite,
- Chaque enfants est un arbre.

## Comment représenteriez vous une telle structure?

<R, G, D>

R: racine

G: sous-arbre gauche

D: sous-arbre droite

## Comment cela s'écrirait en C?

```
typedef struct _node {  
    contenu info;  
    struct _node *left, *right;  
} node;  
typedef node *tree;
```

# L'arbre binaire

## Que se passerait-il avec

```
typedef struct _node {  
    int info;  
    struct _node left, right;  
} node;
```

- On ne sait pas quelle est la taille de node, on ne peut pas l'allouer!

## Interface minimale

- Qu'y mettriez vous?

# L'arbre binaire

## Que se passerait-il avec

```
typedef struct _node {
    int info;
    struct _node left, right;
} node;
```

- On ne sait pas quelle est la taille de node, on ne peut pas l'allouer!

## Interface minimale

- Qu'y mettriez vous?

```
NULL          -> arbre (vide)
<n, arbre, arbre> -> arbre
visiter(arbre) -> nœud (la racine de l'arbre)
gauche(arbre)  -> arbre (sous-arbre de gauche)
droite(arbre)  -> arbre (sous-arbre de droite)
```

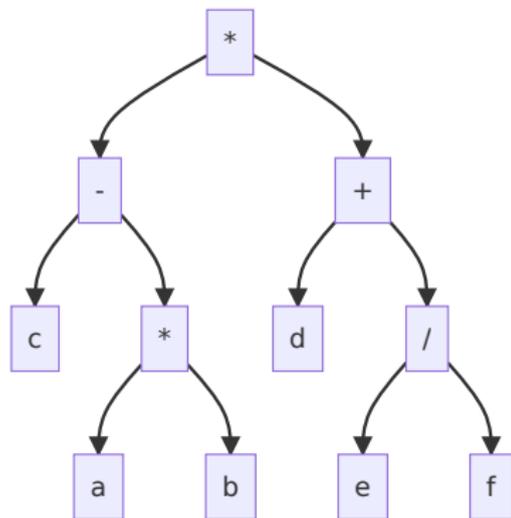
- Les autres opérations (insertion, parcours, etc) dépendent de ce qu'on stocke dans l'arbre.

## Exemple d'arbre binaire

- Représentez  $(c - a * b) * (d + e / f)$  à l'aide d'un arbre binaire (matrix)

# Exemple d'arbre binaire

- Représentez  $(c - a * b) * (d + e / f)$  à l'aide d'un arbre binaire (matrix)



## Remarques

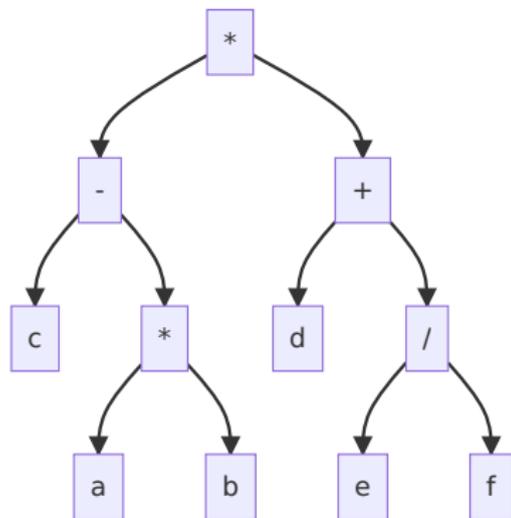
- L'arbre est **hétérogène**: le genre d'info est pas le même sur chaque nœud (opérateur, opérande).
- Les feuilles contiennent les opérandes.
- Les nœuds internes contiennent les opérateurs.

# Parcours d'arbres binaires

- Appliquer une opération à tous les nœuds de l'arbre,
- Nécessité de **parcourir** l'arbre,
- Utiliser uniquement l'interface: visiter, gauche, droite.

## Une idée de comment parcourir cet arbre?

- 3 parcours (R: Racine, G: sous-arbre gauche, D: sous-arbre droit):



1. Parcours **préfixe** (R, G, D),
2. Parcours **infixe** (G, R, D),
3. Parcours **postfixe** (G, D, R).

## Le parcours infixe (G, R, D)

- Gauche, Racine, Droite:
  1. On descend dans l'arbre de gauche tant qu'il est pas vide,
  2. On visite la racine du sous arbre,
  3. On descend dans le sous-arbre de droite (s'il est pas vide),
  4. On recommence.

# Le parcours infixe (G, R, D)

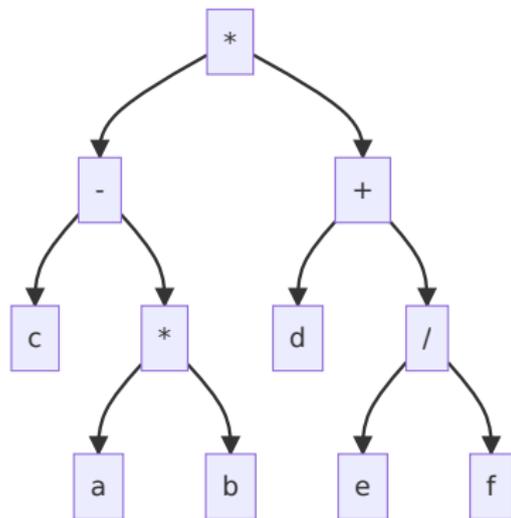
- Gauche, Racine, Droite:
  1. On descend dans l'arbre de gauche tant qu'il est pas vide,
  2. On visite la racine du sous arbre,
  3. On descend dans le sous-arbre de droite (s'il est pas vide),
  4. On recommence.

## Incompréhensible?

- La récursivité c'est la vie.

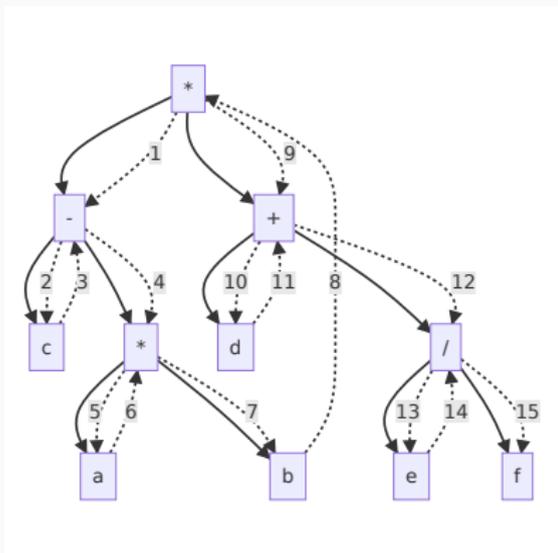
```
parcours_infixe(arbre a)
  si est_pas_vide(gauche(a))
    parcours_infixe(gauche(a))
  visiter(A)
  si est_pas_vide(droite(A))
    parcours_infixe(droite(A))
```

# Graphiquement (dessins)



```
parcours_infixe(arbre a)
  si est_pas_vider(gauche(a))
    parcours_infixe(gauche(a))
  visiter(A)
  si est_pas_vider(droite(A))
    parcours_infixe(droite(A))
```

# Graphiquement (mermaid c'est super)



```
parcours_infixe(arbre a)
  si est_pas_vide(gauche(a))
    parcours_infixe(gauche(a))
  visiter(A)
  si est_pas_vide(droite(A))
    parcours_infixe(droite(A))
```

## Remarque

Le nœud est visité à la **remontée**.

## Résultat

$c - a * b * d + e / f$

Live code

# Et en C?

## Live code

```
typedef int data;
typedef struct _node {
    data info;
    struct _node* left;
    struct _node* right;
} node;
typedef node* tree_t;
void tree_print(tree_t tree, int n) {
    if (NULL != tree) {
        tree_print(tree->left, n+1);
        for (int i = 0; i < n; i++) {
            printf(" ");
        }
        printf("%d\n", tree->info);
        tree_print(tree->right, n+1);
    }
}
```

## Question

**Avez-vous compris le fonctionnement?**

## Question

**Avez-vous compris le fonctionnement?**

**Vous en êtes sûr · e · s?**

# Question

**Avez-vous compris le fonctionnement?**

**Vous en êtes sûr · e · s?**

**OK, alors deux exercices:**

1. Écrire le pseudo-code pour le parcours R, G, D (matrix).
2. Écrire le pseudo-code pour la parcours G, D, R (matrix),

**Rappel**

```
parcours_infixe(arbre a)
    si est_pas_vide(gauche(a))
        parcours_infixe(gauche(a))
    visiter(a)
    si est_pas_vide(droite(a))
        parcours_infixe(droite(a))
```

# Correction

- Les deux parcours sont des modifications **triviales**<sup>1</sup> de l'algorithme infixe.

## Le parcours postfixe

```
parcours_postfixe(arbre a)
  si est_pas_vider(gauche(a))
    parcours_postfixe(gauche(a))
  si est_pas_vider(droite(a))
    parcours_postfixe(droite(a))
  visiter(a)
```

## Le parcours préfixe

```
parcours_prefixe(arbre a)
  visiter(a)
  si est_pas_vider(gauche(a))
    parcours_prefixe(gauche(a))
  si est_pas_vider(droite(a))
    parcours_prefixe(droite(a))
```

---

<sup>1</sup>Copyright cours de mathématiques pendant trop d'années.

# Correction

- Les deux parcours sont des modifications **triviales**<sup>1</sup> de l'algorithme infixe.

## Le parcours postfixe

```
parcours_postfixe(arbre a)
  si est_pas_vider(gauche(a))
    parcours_postfixe(gauche(a))
  si est_pas_vider(droite(a))
    parcours_postfixe(droite(a))
  visiter(a)
```

## Le parcours préfixe

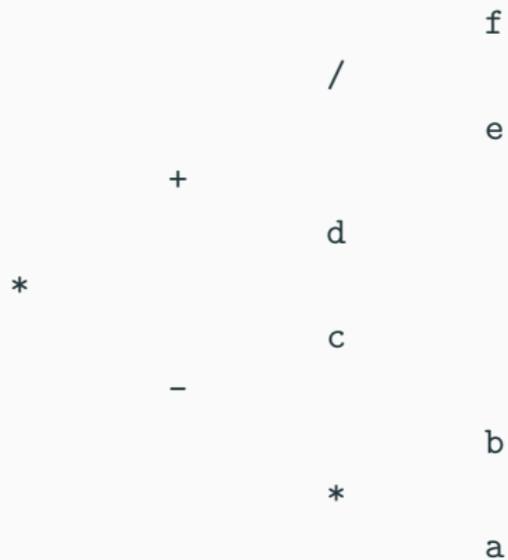
```
parcours_prefixe(arbre a)
  visiter(a)
  si est_pas_vider(gauche(a))
    parcours_prefixe(gauche(a))
  si est_pas_vider(droite(a))
    parcours_prefixe(droite(a))
```

**Attention:** L'implémentation de ces fonctions en C sont à **faire** en exercice (inspirez vous de ce qu'on a fait avant)!\_\_\_\_\_

<sup>1</sup>Copyright cours de mathématiques pendant trop d'années.

## Exercice: parcours

Comment imprimer l'arbre ci-dessous?



## Exercice: parcours

Comment imprimer l'arbre ci-dessous?

```
          f
        /
       e
      +
     d
    *
   c
  -
 b
 *
 a
```

**Bravo vous avez trouvé!**

- Il s'agissait du parcours D, R, G.

# Implémentation

**Vous avez 5 min pour implémenter cette fonction et la poster sur matrix!**

# Implémentation

**Vous avez 5 min pour implémenter cette fonction et la poster sur matrix!**

```
void pretty_print(tree_t tree, int n) {
    if (NULL != tree) {
        pretty_print(tree->right, n+1);
        for (int i = 0; i < n; ++i) {
            printf(" ");
        }
        printf("%d\n", tree->info);
        pretty_print(tree->left, n+1);
    }
}
```

## Exercice supplémentaire (sans corrigé)

Écrire le code de la fonction

```
int depth(tree_t t);
```

qui retourne la profondeur maximale d'un arbre.

Indice: la profondeur à chaque niveau peut-être calculée à partir du niveau des sous-arbres de gauche et de droite.

## La recherche dans un arbre binaire

- Les arbres binaires peuvent retrouver une information très rapidement.
- À quelle complexité? À quelle condition?

# La recherche dans un arbre binaire

- Les arbres binaires peuvent retrouver une information très rapidement.
- À quelle complexité? À quelle condition?

## Condition

- Le contenu de l'arbre est **ordonné** (il y a une relation d'ordre (<, > entre les éléments).

## Complexité

- La profondeur de l'arbre (ou le  $\mathcal{O}(\log_2(N))$ )

# La recherche dans un arbre binaire

- Les arbres binaires peuvent retrouver une information très rapidement.
- À quelle complexité? À quelle condition?

## Condition

- Le contenu de l'arbre est **ordonné** (il y a une relation d'ordre (<, > entre les éléments)).

## Complexité

- La profondeur de l'arbre (ou le  $\mathcal{O}(\log_2(N))$ )

## Exemple: les arbres lexicographiques

- Chaque nœud contient une information de type ordonné, la **clé**,
- Par construction, pour chaque nœud  $N$ :
  - Toutes clé du sous-arbre à gauche de  $N$  sont inférieurs à la clé de  $N$ .
  - Toutes clé du sous-arbre à droite de  $N$  sont inférieurs à la clé de  $N$ .

# Algorithme de recherche

- Retourner le nœud si la clé est trouvée dans l'arbre.

```
arbre recherche(clé, arbre)
  tante_que est_non_vide(arbre)
  si clé < clé(arbre)
    arbre = gauche(arbre)
  sinon si clé > clé(arbre)
    arbre = droite(arbre)
  sinon
    retourne arbre
retourne NULL
```

# Algorithme de recherche, implémentation (live)

# Algorithme de recherche, implémentation (live)

```
typedef int key_t;
typedef struct _node {
    key_t key;
    struct _node* left;
    struct _node* right;
} node;
typedef node* tree_t;
tree_t search(key_t key, tree_t tree) {
    tree_t current = tree;
    while (NULL != current) {
        if (current->key > X) {
            current = current->gauche;
        } else if (current->key < X){
            current = current->droite;
        } else {
            return current;
        }
    }
    return NULL;
}
```

## Exercice (5-10min)

Écrire le code de la fonction

```
int tree_size(tree_t tree);
```

qui retourne le nombre total de nœuds d'un arbre et poster le résultat sur matrix.

Indication: la taille, est  $1 +$  le nombre de nœuds du sous-arbre de gauche additionné au nombre de nœuds dans le sous-arbre de droite.

## Exercice (5-10min)

Écrire le code de la fonction

```
int tree_size(tree_t tree);
```

qui retourne le nombre total de nœuds d'un arbre et poster le résultat sur matrix.

Indication: la taille, est  $1 +$  le nombre de nœuds du sous-arbre de gauche additionné au nombre de nœuds dans le sous-arbre de droite.

```
int arbre_size(tree_t tree) {
    if (NULL == tree) {
        return 0;
    } else {
        return 1 + tree_size(tree->left)
            + tree_size(tree->right);
    }
}
```

# L'insertion dans un arbre binaire

- C'est bien joli de pouvoir faire des parcours, recherches, mais si on peut pas construire l'arbre...

## **Pour un arbre lexicographique**

- Rechercher la position dans l'arbre où insérer.
- Créer un nœud avec la clé et le rattacher à l'arbre.

## Exemple d'insertions

- Clés uniques pour simplifier.
- Insertion de 5, 15, 10, 25, 2, -5, 12, 14, 11.
- Rappel:
  - Plus petit que la clé courante => gauche,
  - Plus grand que la clé courante => droite.
- Faisons le dessin ensemble

### Exercice (3min, puis matrix)

- Dessiner l'arbre en insérant 20, 30, 60, 40, 10, 15, 25, -5

## Pseudo-code d'insertion (1/2)

- Deux parties:
  - Recherche le parent où se passe l'insertion.
  - Ajout de l'enfant dans l'arbre.

### Recherche du parent

```
arbre position(arbre, clé)
  si est_non_vider(arbre)
    si clé < clé(arbre)
      suivant = gauche(arbre)
    sinon
      suivant = droite(arbre)
  tant que clé(arbre) != clé && est_non_vider(suivant)
    arbre = suivant
  si clé < clé(arbre)
    suivant = gauche(arbre)
  sinon
    suivant = droite(arbre)
```

## Pseudo-code d'insertion (2/2)

- Deux parties:
  - Recherche de la position.
  - Ajout dans l'arbre.

### Ajout de l'enfant

```
ajout(arbre, clé)
```

```
  si est_vide(arbre)
```

```
    arbre = nœud(clé)
```

```
  sinon
```

```
    si clé < clé(arbre)
```

```
      gauche(arbre) = nœud(clé)
```

```
    sinon si clé > clé(arbre)
```

```
      droite(arbre) = nœud(clé)
```

```
    sinon
```

```
      retourne
```

Recherche du parent (ensemble)

# Code d'insertion en C

## Recherche du parent (ensemble)

```
tree_t position(tree_t tree, key_t key) {
    tree_t current = tree;
    if (NULL != current) {
        tree_t subtree = key > current->key ? current->right :
            current->left;
        while (key != current->key && NULL != subtree) {
            current = subtree;
            subtree = key > current->key ? current->right :
                current->left;
        }
    }
    return current;
}
```