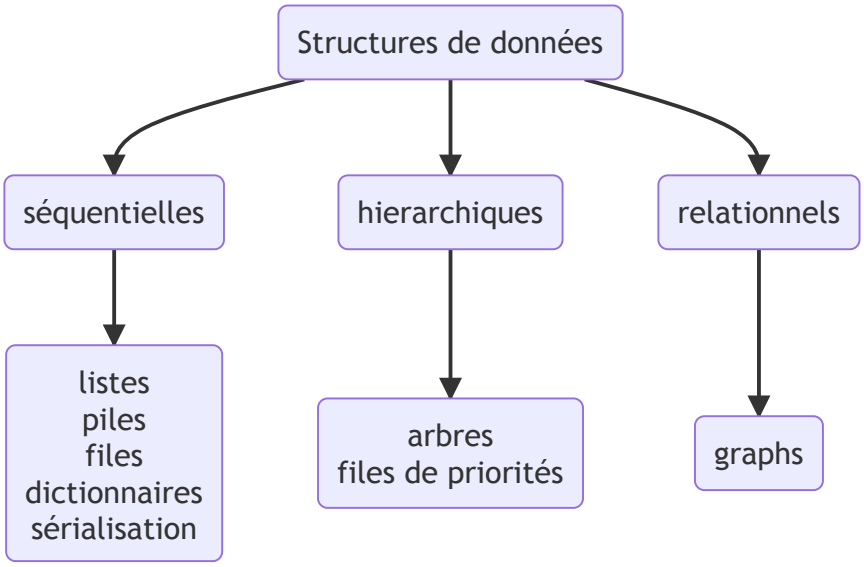
I. Introduction - définition

Utilisation des arbres

- Arbres syntaxiques
- Arbre lexicographique
- Arbre de décision / classification (ML)
- Compression de données
- Expressions mathématiques

Classification



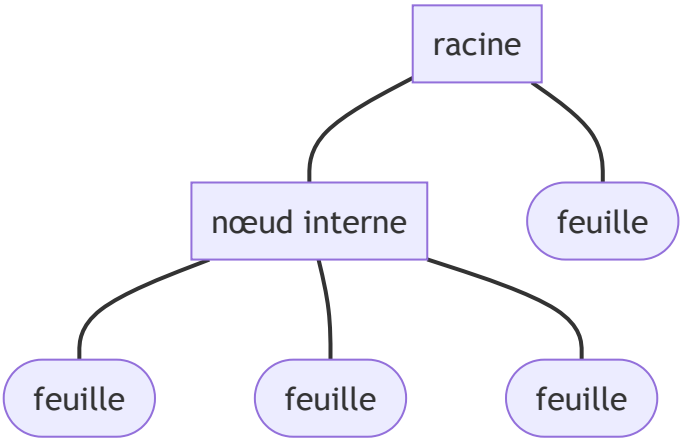
Définitions

Un arbre (enraciné) est

- soit un ensemble vide
- soit un ensemble fini non vide A muni d'une relation binaire $<$ (est le fils de $\dots$) telle que
- il existe $r \in A$ tel que $\forall x \in A, r < x$ (il existe un ancêtre)
- $\forall x \in A \setminus \{r\}, \exists! y \in A, x < y$ (il y a un père unique)
- $\forall x \in A \setminus \{r\}, \exists n \in \mathbb{N}$ et $(x_1, x_2, \dots, x_n) \in A^n, x < x_1 < x_2 < \cdots < x_n < r$ (chaque élément descend de l'ancêtre)

On parle aussi de frères, de descendance et d'ancêtres.

L'*arité* d'un père correspond au nombre de ses fils

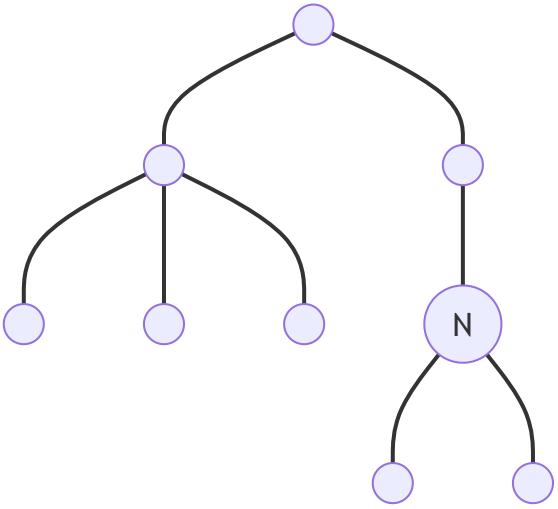


Un *arbre n-aire* est un arbre dont les nœuds sont d'arité maximale n Dans l'exemple, chaque nœud a au maximum une arité de 3 donc l'arbre est un arbre ternaire.

La *taille* de l'arbre est le nombre de nœuds qui le compose. Dans l'exemple, la taille est 6.

La *profondeur* d'un nœud est:

- -1 si l'arbre est vide
- 0 pour la racine
- le nombre de nœuds depuis la racine avant d'atteindre le nœud



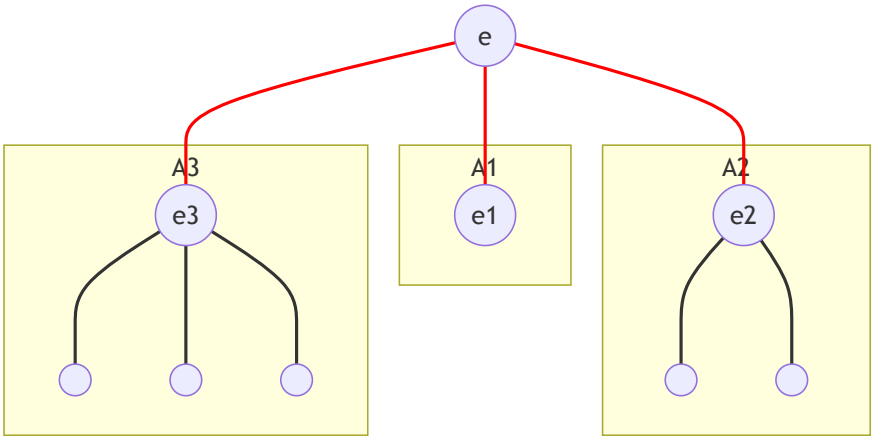
Par exemple, le nœud N a une profondeur de 2.

La *hauteur* de l'arbre est la profondeur maximale de ses feuilles. Dans l'exemple, la hauteur est de 3.

Définition inductive

Soit A un ensemble fini appelé nœuds.

- les arbres de hauteur 0 sont les éléments de A noté $(e, 0)$ où e est la racine de l'arbre
- Si $e \in A$ et Si $A_1, A_2, \ldots, A_n$ sont des arbres de hauteur respectives $h_1, h_2, \ldots, h_n$ et dont les racines respectives sont $e_1, e_2, \ldots, e_n$, alors en connectant e à $e_1, e_2, \ldots e_n$, on définit $\big(e, (A_1, A_2, \ldots, A_n)\big)$ l'arbre de racine e, de hauteur $1 + \max\limits_{k \in [\![1,n]\!]}(h_k)$ de sous arbres $A_1, A_2, \ldots, A_n$



Remarques:

Arbre marqué: Chaque nœud associé à une étiquette / valeur

Arbre non marqué: Les nœuds n'ont pas de valeurs associés

Un arbre est un graph:

- Simple: pas de boucles ou d'arêtes multiples
- Non orienté: les parcours $a \to b$ et $b \to a$ sont toujours possible
- Acyclique: pas de "boucles"
- Connexe: tous les nœuds sont accessibles

Propriété:

Pour un arbre de hauteur h d'arité $a > 1$, le nombre n de nœuds vérifie

$$h + 1 \geqslant n \geqslant \frac{a^{h+1} - 1}{a - 1}$$

Preuve:

Le nombre minimal de nœuds pour une hauteur donnée est un arbre contenant 1 nœud par hauteur. Le nombre maximal de nœuds pour une hauteur donnée est un arbre contenant i nœud par niveau i.

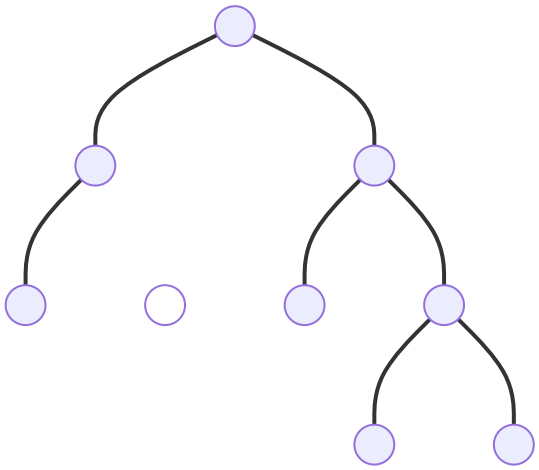
$$\sum_{i=0}^{h} 1 \geqslant n \geqslant \sum_{i=0}^{h} a^i$$
$$\iff h + 1 \geqslant n \geqslant \frac{a^{h+1} - 1}{a - 1}$$

II. Arbres particuliers

Définitions

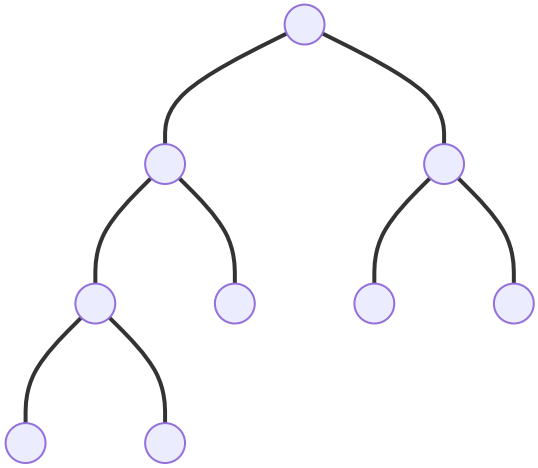
Arbres binaires:

Tous les nœuds sont d'arité au maximum 2.



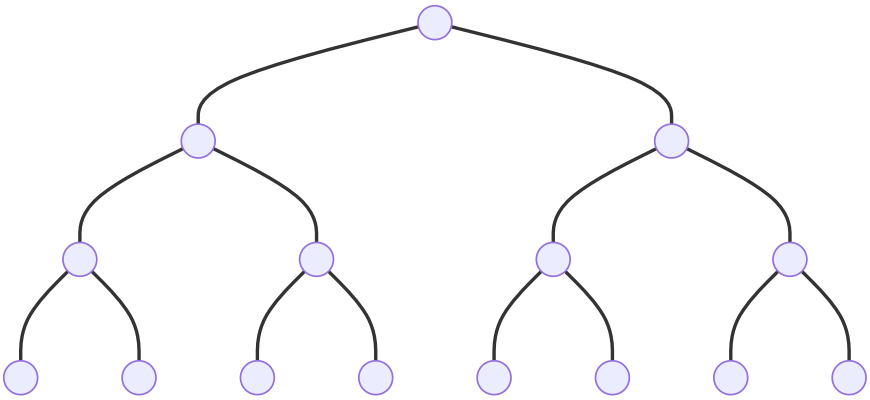
Arbre binaire entier:

Tous les nœuds sont d'arité 2



Arbre binaire complet

Arbre binaire dont les feuilles ont toutes la même profondeur. Il possède le nombre maximal de feuilles pour une hauteur donnée. On parle donc de fils gauche et de fils droit.



Propriétés

Nombre de nœuds $n \leqslant 2^p$ (pour un niveau p)

Hauteur h: $\lfloor \log_2(n) \rfloor < h \leqslant n - 1$

Arbre binaire à i nœuds interne (et la racine) d'arité 2 et f feuilles : $f = i + 1$

Nombre de feuilles $f \leqslant 2^h$

Profondeur maximale d'un arbre d'arité a composée de n nœuds:

$$\log_a\big((a-1)\times n+1\big)-1 \leqslant h \leqslant n-1$$

Remarque:

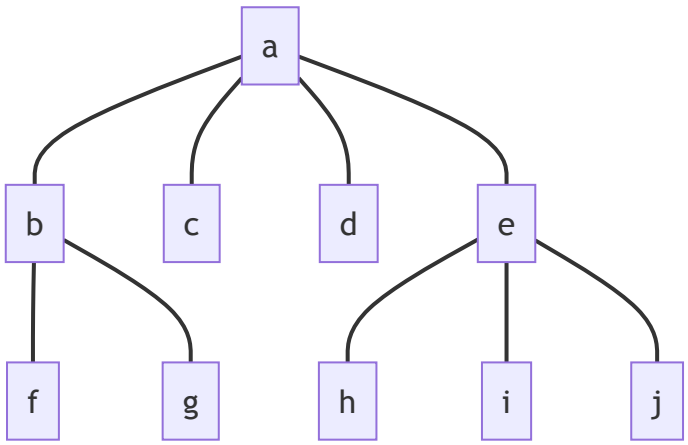
$$\begin{aligned} & \log_a((a-1)\times n+1)-1 \leq h \\ \Longrightarrow\ & \log_a((a-1)\times n) < \log_a((a-1)\times n+1) \leq h+1 \\ \Longrightarrow\ & \lfloor \log_a((a-1)\times n)\rfloor < h+1 \\ \Longrightarrow\ & \lfloor \log_a((a-1)\times n)\rfloor \leq h \end{aligned}$$

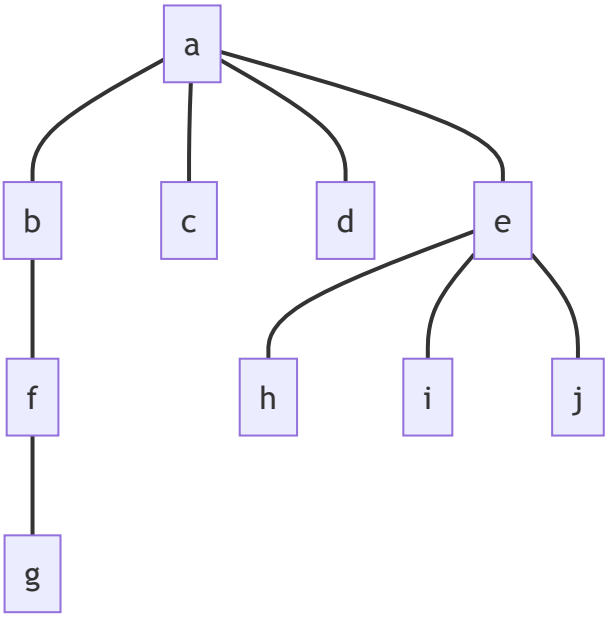
soit $\lfloor \log_a((a-1)\times n)\rfloor \leq h \leq n-1$

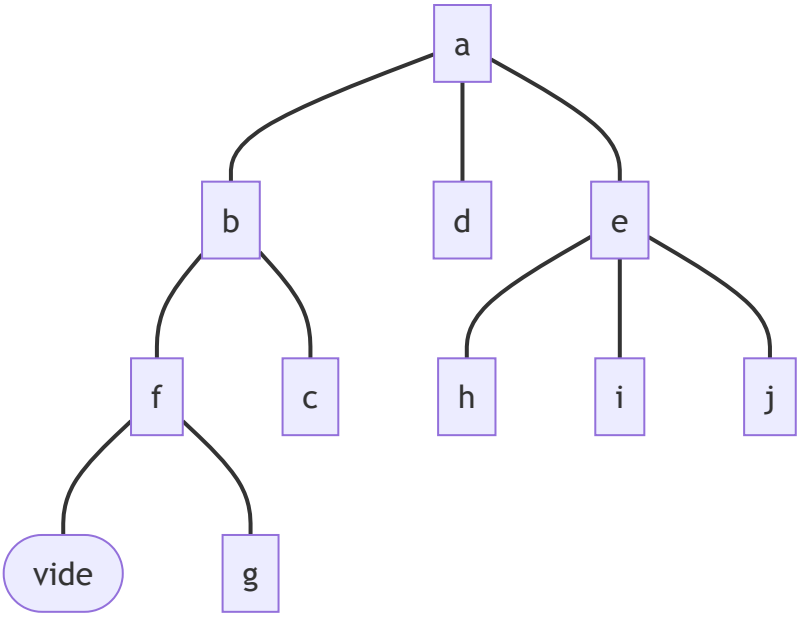
Transformation d'un arbre en arbre binaire

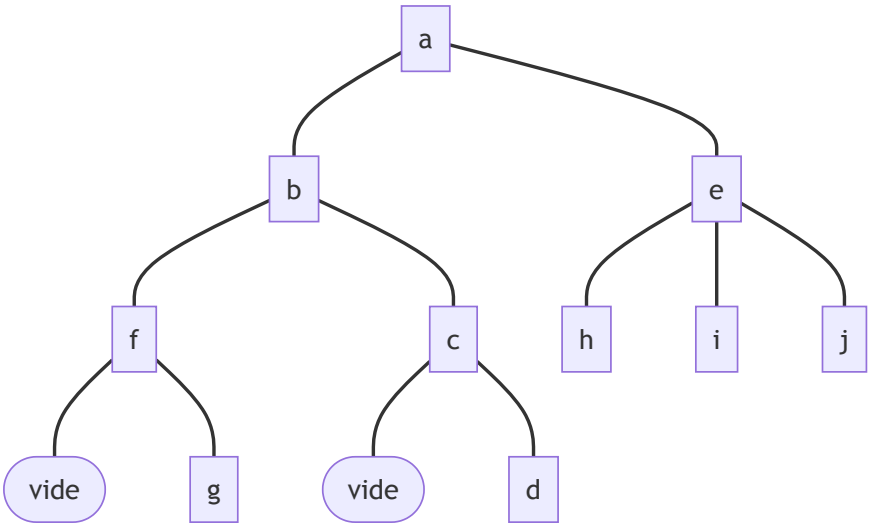
Algorithme de transformation d'un arbre n-aire A en arbre binaire:

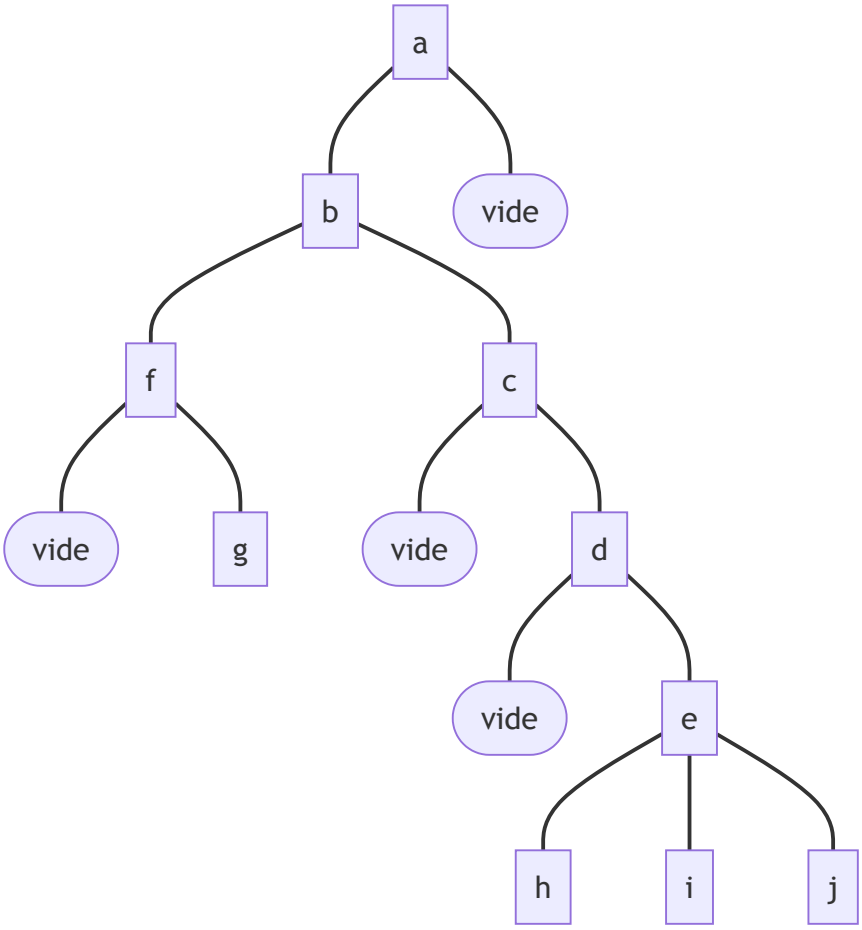
```
Pour chaque e ∈ A
Laisser e_g (fils le plus à gauche) en place
Supprimer les autres fils de e et les chainer à e_g
Le nœud e_d est le prochain frère de e
Fin pour
```

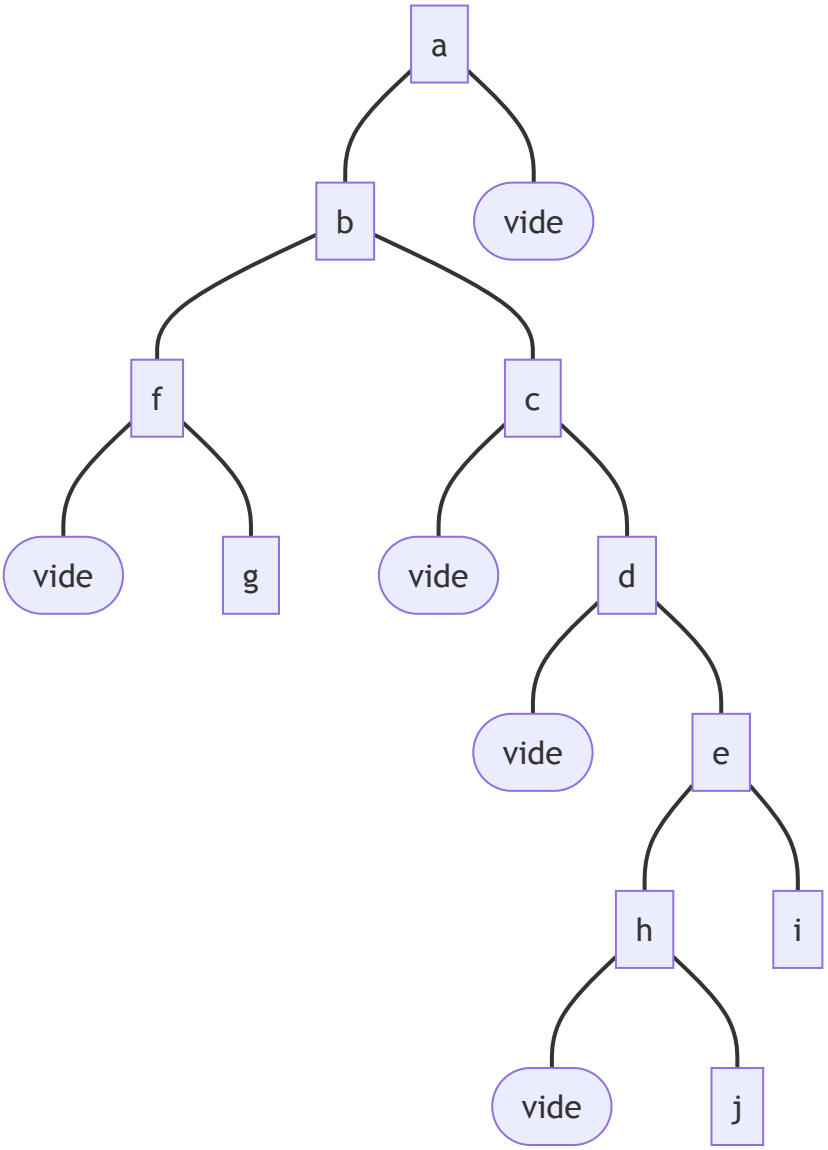


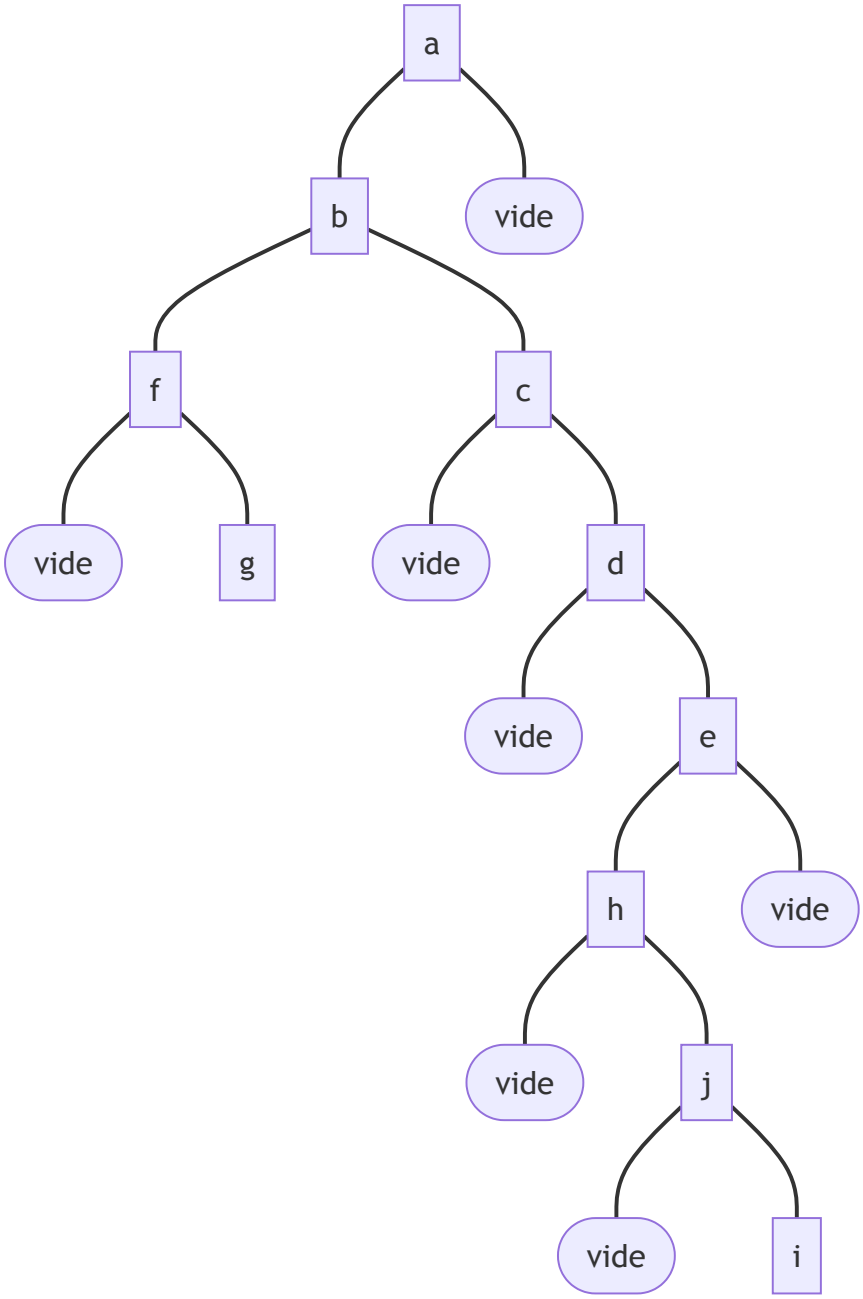

a
b
c
d
e
f
h
i
j
g


a
b
d
e
f
c
h
i
j
vide
g

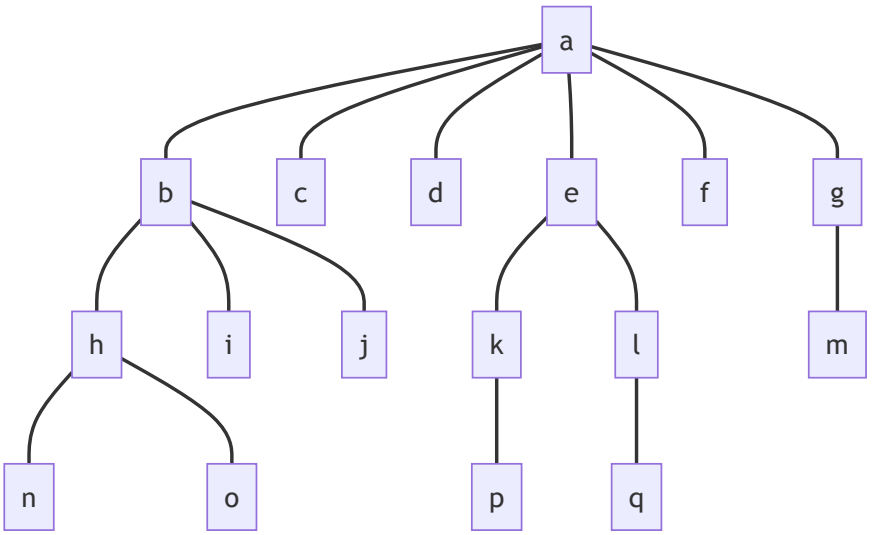

a
b
e
f
c
h
i
j
vide
g
vide
d

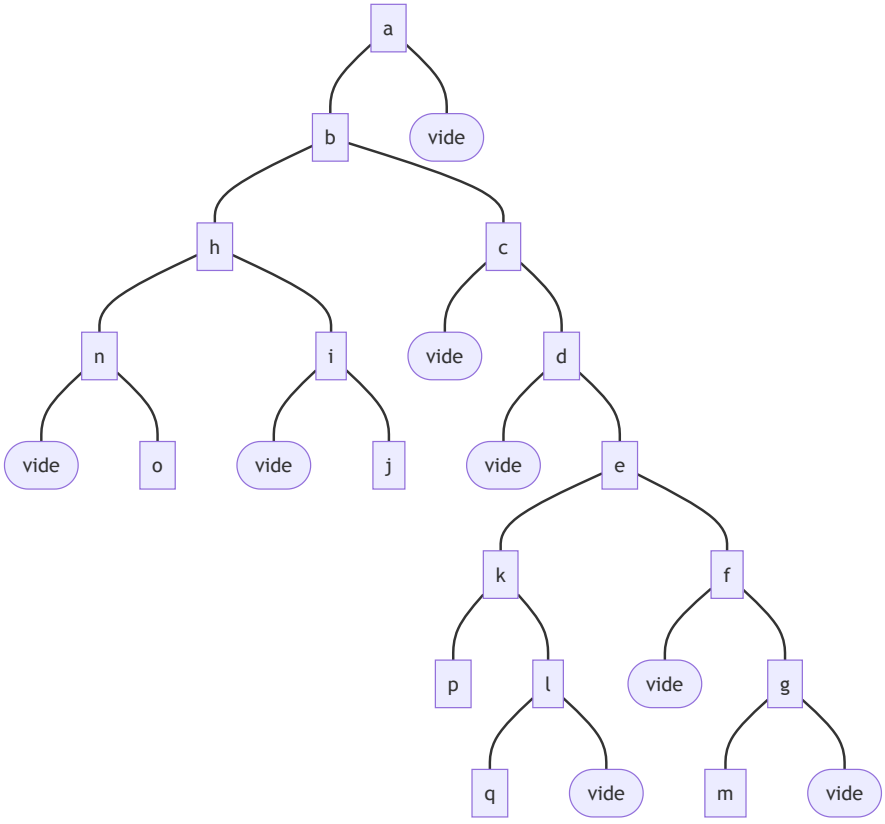

a
b
vide
f
c
vide
g
vide
d
vide
e
h
i
j


a
b
vide
f
c
vide
g
vide
d
vide
e
h
i
vide
j



Exercice:


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q


a
b
vide
h
c
n
i
vide
d
vide
o
vide
j
vide
e
k
f
p
l
vide
g
q
vide
m
vide

Arbres binaires : implémentation sous forme de tableau en C (sérialisation)

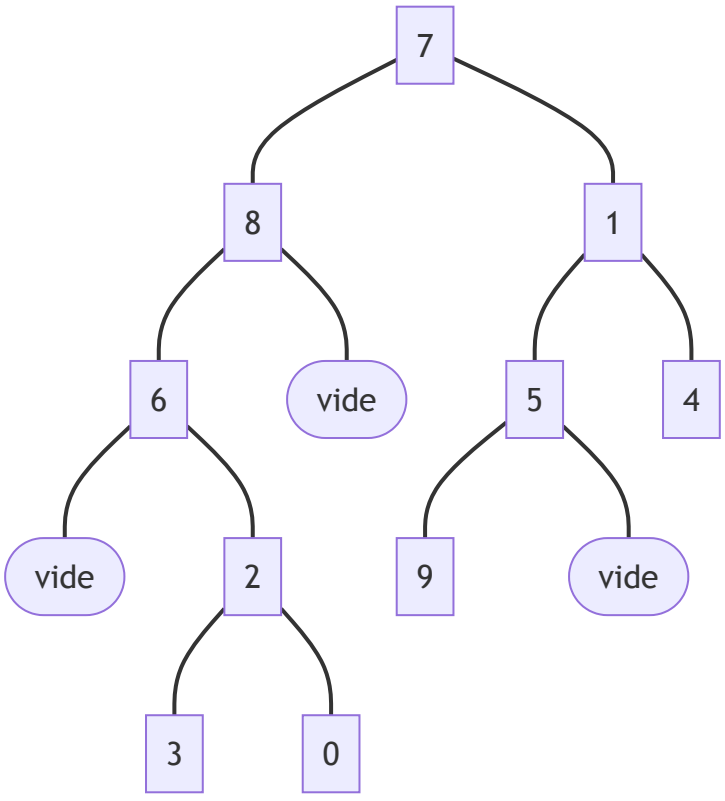
On stocke un arbre binaire dans un tableau d'enregistrements Chaque enregistrement E est défini par:

- E.clé : valeur du nœud
- E.gauche : indice du fils gauche dans le tableau
- E.droit : indice du fils droit dans le tableau

Une valeur de E.gauche ou E.droit égale à -1 indique qu'il n'existe pas de fils.

Indice dans le tableau	0	1	2	3	4	5	6	7	8	9
Clé	23	2	3	5	7	11	13	37	41	19
Gauche	-1	5	3	-1	-1	9	-1	8	6	-1
Droit	-1	4	0	-1	-1	-1	2	1	-1	-1

1. Représenter cet arbre. Est-il binaire ? Entier ? Complet ?



C'est un arbre binaire mais il n'est pas complet ni entier.

2. Définir le code en C permettant de définir ce tableau (statique)

```c
typedef struct item {
    int val;
    int left;
    int right;
};

item table[10];
```

3. Écrire une fonction en C qui renvoie l'enregistrement de la racine

```c
int getRoot(item* table, int len, item* root) {
    for(int i = 0; i < len; i++) {
        for(int j = 0; j < len; j++) {
            // on vérifie si i est un fils de j
            if(table[j].left == i || table[j].right == i) {
                break; // nouvelle itération de la boucle exterieure
            }
        }

        return i;
    }

    // pas de racine, il y a une boucle
    return -1;
}
```

Solution:

```c
item getRoot(item* table) {
    int search[10] = {0};

    for(int i = 0; i < 10; i++) {
        if(table[i].left != -1) {
            search[table[i].left] = 1;
        }

        if(table[i].right != -1) {
            search[table[i].right] = 1;
        }
    }

    for(int i = 0; i < 10; i++) {
        if(search[i] == 0)
            return table[i];
```

```
        }
    }
```

Solution 2:

```
item getRoot(item* table, int len) {
    int root = 0;

    for(int i = 0; i < len; i++) {
        root += i;

        if(table[i].left != -1) {
            root -= table[i].left;
        }

        if(table[i].right != -1) {
            root -= table[i].right;
        }
    }

    return table[root];
}
```

4. Écrire une fonction en C qui affiche la valeur de toutes les feuilles de T

```
void showValues(item* table, int len) {
    item root;
    int rootIndex = getRoot(table, len, &root);

    showChildValues(rootIndex, table);
}

void showChildValues(int index, item* table) {
    item val = table[index];

    if(val.left == -1 && val.right == -1) {
        printf("%d\n", val.val);
    }

    if(val.left != -1)
        showChildValues(val.left, table);

    if(val.right != -1)
        showChildValues(val.right, table);
}
```

Solution:

```
void showValues(item* table, int len) {
```

```
        for(int i = 0; i < len; i++) {

            // uniquement vrai si gauche = droite = -1
            if(table[i].left == table[i].right) {
                printf("%d", table[i].val);
            }
        }
    }
```

Arbres binaires de recherche

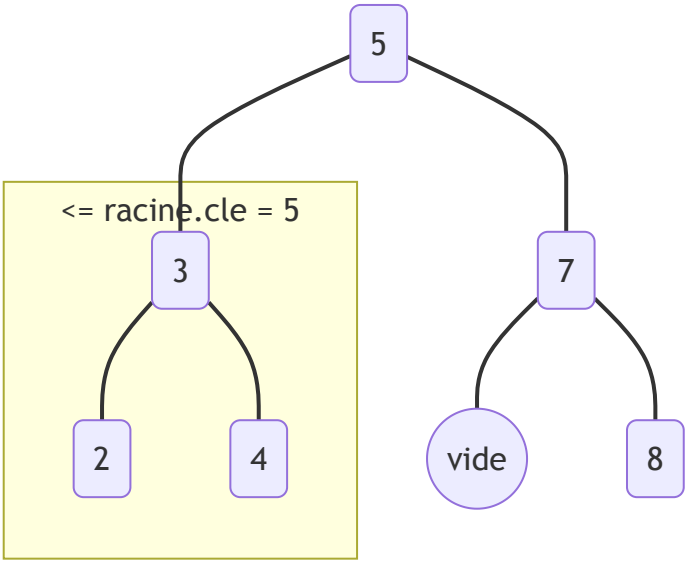
Arbre binaire marqué:

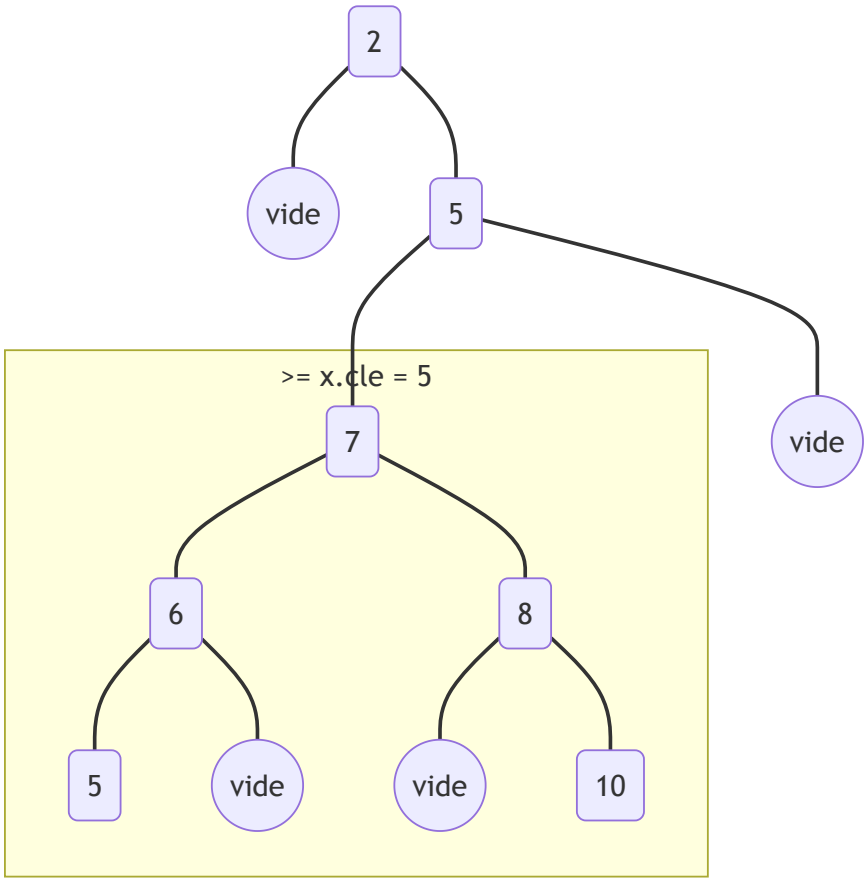
- Présence d'une étiquette pour chaque nœud x de A
- Étiquettes
  - Même type
  - Présence d'une clé appartenant à un ensemble E
  - Relation d'ordre total sur E
  - Notation x.clé

Organisation de l'arbre binaire de recherche: soit un nœud x de A.

- Alors toutes les étiquettes des nœuds y des sous arbres gauches de x sont telles que y.clé $\leqslant x$.clé
- Et toutes les étiquettes des nœuds y des sous arbres droits de x sont telles que y.clé $\geqslant x$.clé

Exemples:


5
<= racine.cle = 5
3
7
2
4
vide
8



Nœud:

- Étiquette
  - clé
  - information
- Fils gauche
- Fils droit

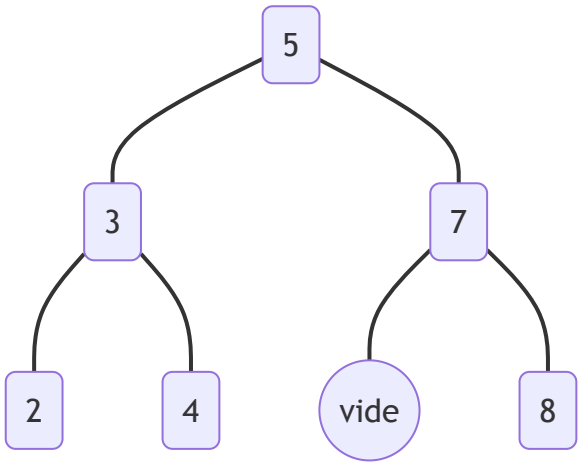
Opérations fondamentales des ABR:

- Constructeur
- Accesseurs
  - Recherche d'une clé
  - Rechercher le minimum
  - Rechercher le maximum
  - Rechercher un successeur
  - Rechercher un prédécesseur

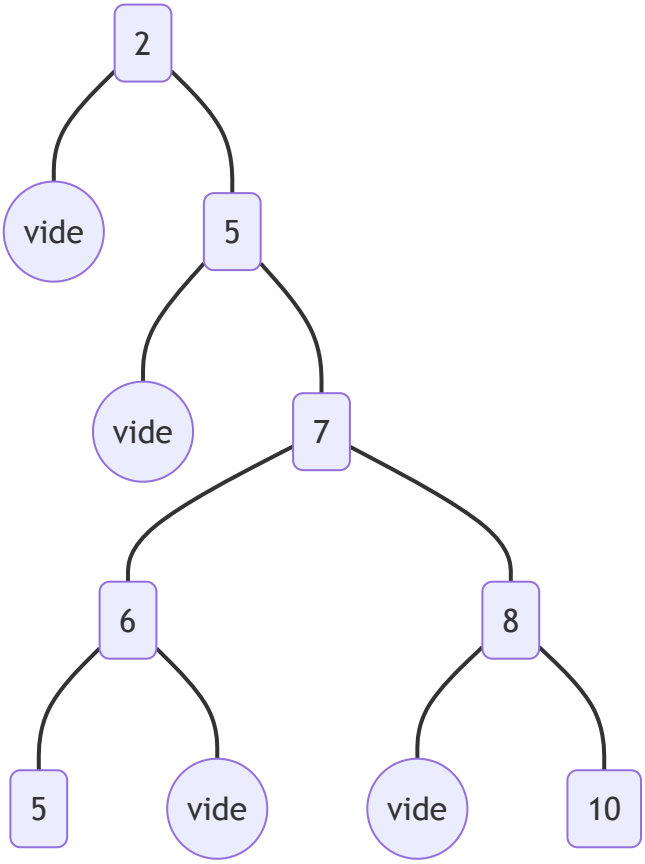
- Transformateurs
  - Insérer un nœud
  - Supprimer un nœud

Utilité: faciliter une recherche avec une recherche par clé dans l'ABR

- temps de recherche proportionnel à la hauteur h de l'arbre
  - $T(n) = O(n)$ recherche linéaire dans le pire des cas (liste chainée)
  - $T(n) = O\big(\log_2(n)\big)$ recherche en temps logarithmique en moyenne pour un ABR complet.
- Dépend de la construction de l'arbre:



6 nœuds, hauteur 2 : Efficace



6 nœuds, hauteur 4 : Peu efficace

L'efficacité augment quand la hauteur diminue.

Arbre dégénéré (filiforme) : chaque nœud n'a qu'un enfant.

Arbres binaires bicolores (rouge - noir)

Arbre de recherche: possibilité d'arbre déséquilibré (filiforme, dégénéré)

- Recherche en $O(n)$ où n est le nombre de nœuds
- aucun apport par rapport à une liste chainée

Arbre bicolore: arbre de recherche particulier, approximativement équilibré

- Recherche en $\Theta\big(\log_2(n)\big)$

Arbre binaire bicolore $\rightarrow$ marqué

- présence d'une étiquette pour chaque nœud x de A

- Fils gauche, fils droit
- Étiquettes
  - même type
  - clé
  - information
  - information supplémentaire : *couleur* (1 bit)

Propriétés rouge noir:

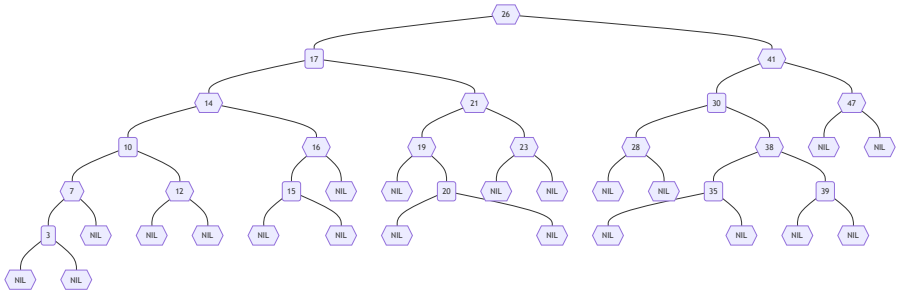
- chaque nœud est soit rouge, soit noir
- chaque feuille est noire
- la racine est noire
- si un nœud est rouge, ses deux enfants sont noirs
- pour chaque nœud, tous les chemins simples reliant le nœud à des feuilles situées plus bas dans l'arbre contiennent le même nombre de nœuds noirs

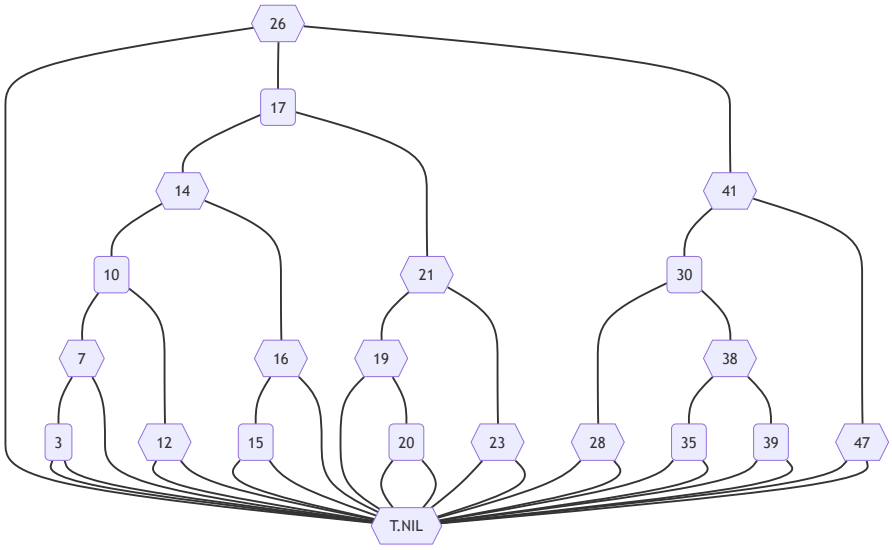
Remarque:

- Ajout de nœuds noirs vide (pointeur sur NIL) pour générer les feuilles
- Informations contenues dans les nœuds internes

Exemple:

Les nœuds rouges sont entre [] et les nœuds noirs entre ⟨⟩

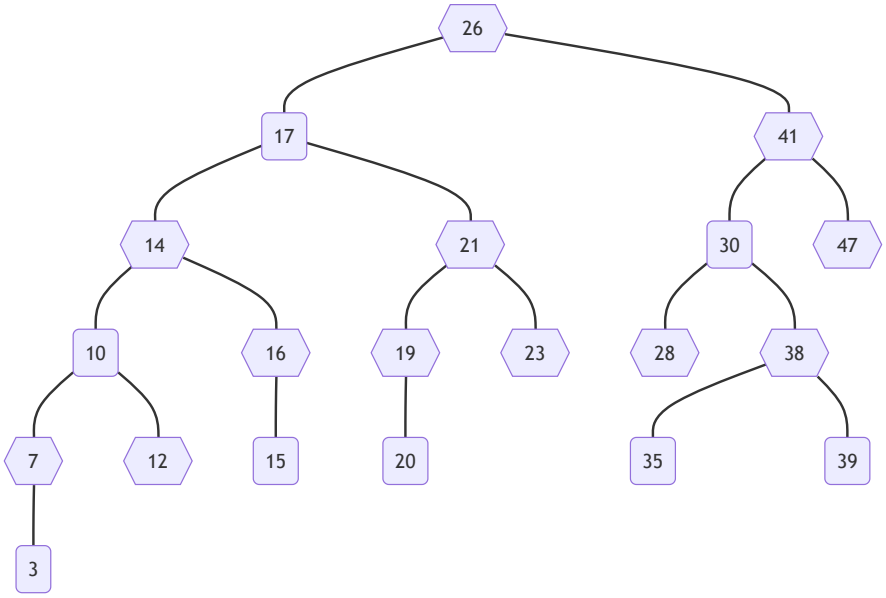




T.NIL : sentinelle

- Simplification des conditions aux limites
- Même attributs qu'un nœud ordinaire:
  - Valeur: noir
  - Autres attributs: valeurs quelconques
- Aussi parent de la racine

Hauteur noire omise



Propriété rouge - noire : la longueur L d'un chemin allant de la racine à une feuille est comprise entre h_n et $2h_n$.

Chemin: suite de nœuds dont chacun est le prédécesseur ou le successeur du suivant.

h_n étant la profondeur noire de l'arbre.

Justification:

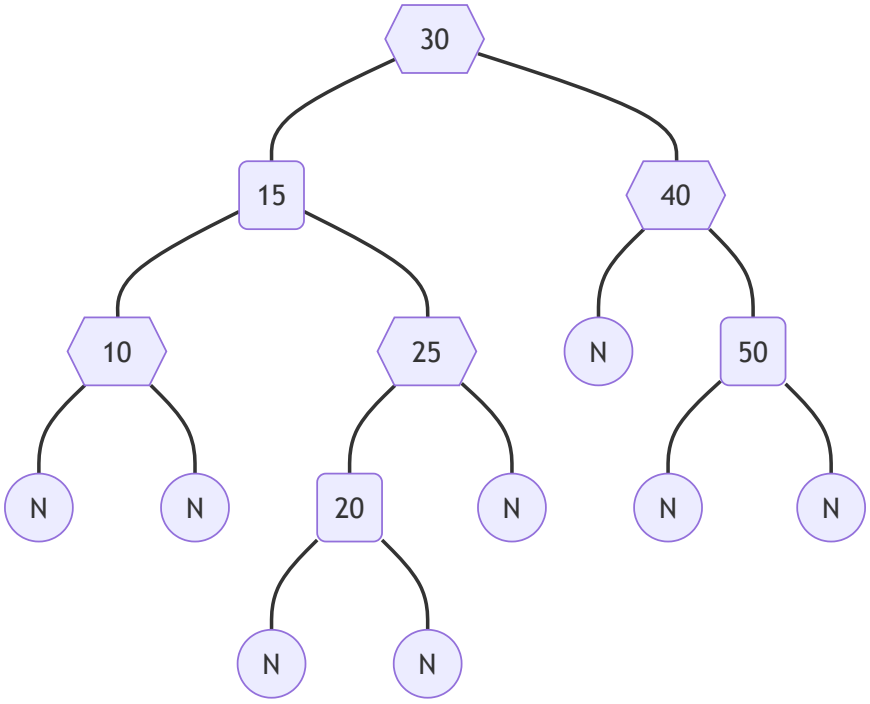
Si tous les nœuds sont noirs, alors h_n est égale à la profondeur de l'arbre.

S'il y a systématiquement une alternance entre nœuds rouges et noirs, alors la hauteur totale h de l'arbre vaut $2h_n$

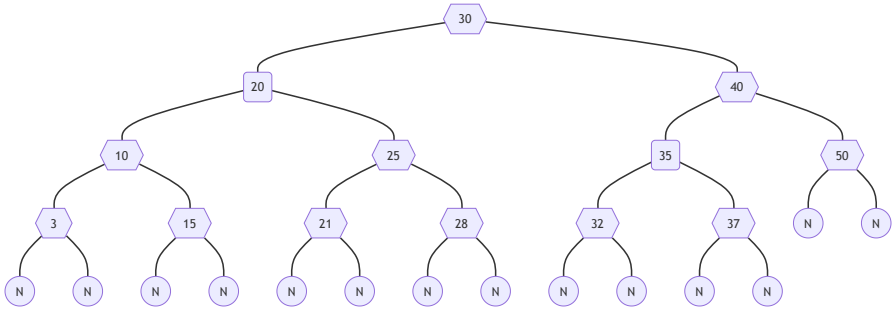
Exercice 1:

- Arbre 1 : non (rouge $\rightarrow$ rouge)
- Arbre 2 : non (racine rouge)
- Arbre 3 : non (nombre de nœuds noirs $\neq$)
- Arbre 4 : non (n'est pas un ABR car $10 < 22$)

Exercice 2:



Exercice 3:



1. hauteurs $h_n(30) = 3$, $h_n(20) = 3$, $h_n(35) = 2$ et $h_n(50) = 1$
2. Dans le meilleur des cas, il n'y a pas de nœuds rouges, donc $h = h_n$ et donc le nombre de feuille f vaut 2^h donc $2^{h_n(x)}$. Or, le nombre de nœuds internes i valide $f = i + 1$ donc $i = f - 1$ et donc $i \geqslant 2^{h_n(x)} - 1$

3. Dans le meilleur des cas, on a $h = h_n$.

$$\begin{aligned} i \geqslant 2^{h_n(x)} - 1 &\iff i + 1 \geqslant 2^{h_n(x)} \\ &\iff \log_2(i+1) \geqslant h_n(x) = h \end{aligned}$$

III. Parcours d'arbres

Définition

Soit un arbre A de taille n.

Objectif:

- visiter les nœuds pour traitement
- contrainte: chaque nœud doit être traité une seule fois

Définition:

Un parcours d'arbre est une succession de nœuds $(n_1, n_2, \ldots, n_n)$, indiquant l'ordre dans lequel ils ont été visités

Pricipe général des parcours d'arbre :

- marquage du nœud après sont traitement pour ne plus le traiter
- maintien d'un ensemble de nœuds en attente de traitement

Existence de plusieurs manières de parcourir un arbre

Utilisation des arbres

Arbres d'expression $\rightarrow$ représentation d'expressions

- Arbre syntaxique
- Expressions mathématiques

Arbres préfixes (ou trie) $\rightarrow$ représentation d'un ensemble de mots

Parcours en profondeur

Définitions

Principe de parcours en profondeur:

- Utilisation de la définition inductive des arbres
- Proposition d'algorithmes récursifs

$\Longrightarrow$ Exploration d'un des sous-arbres avant d'explorer le sous-arbre suivant

Plusieurs posibilités:

- Exploration en partant de la racine
- Exploration en partant des feuilles
- Exploration symétrique

Remarque: Le choix de traiter de gauche à doite est arbitraire.

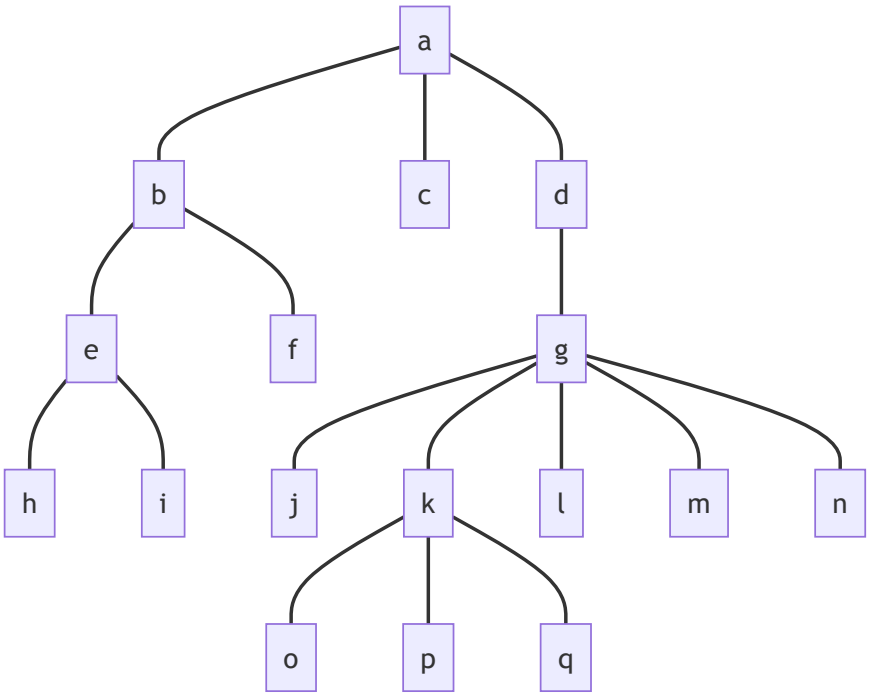
Ordre préfixé:

- Départ à la racine
- Exploration préfixe du sous-arbre gauche
- Exploration préfixe du sous-arbre droit
- $\vdots$
- Exploration du sous-arbre A_n

Pour un arbre binaire, ordre = père, fils gauche, fils droit.

Pour l'arbre ci-dessous, l'ordre préfixé est

$$a \to b \to e \to h \to i \to f \to c \to d \to g \to j \to k \to o \to p \to q \to l \to m \to n$$



Ordre infixé ou parcours symétrique:

- Exploration infixé du sous-arbre gauche
- Puis racine

- Exploration infixé du sous-arbre droit
- $\vdots$
- Exploration des nœuds de A_n en ordre infixé

Pour un arbre binaire, ordre : fils gauche, père, fils droit.

Pour un ABR : ordre infixé

- Obtention de la suite ordonnée des étiquettes

Pour l'arbre précédent, l'ordre infixé est

$$h \to e \to i \to b \to f \to a \to c \to j \to g \to o \to k \to p \to q \to l \to m \to n \to d$$

Ordre postfixé:

- Exploration postfixé du sous-arbre gauche
- Exploration postfixé du sous-arbre droit
- $\vdots$
- Exploration du sous-arbre A_n
- Traitement de la racine

Pour une arbre binaire, l'ordre est fils gauche, fils droit, père

Parcours en profondeur de l'arbre précédent :

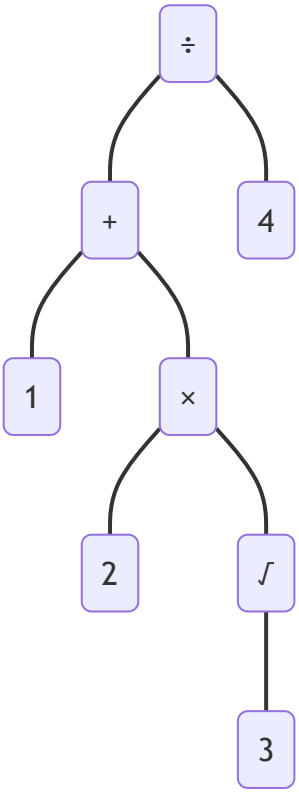
$$h \to i \to e \to f \to b \to c \to j \to o \to p \to q \to k \to l \to m \to n \to g \to d \to a$$

Parcours en profondeur : arbre d'expression

Problème lié aux expressions infixes : gestion des parenthèses

$\left(1 + 2 \times \sqrt{3}\right) / 4$ pour $\frac{1+2\sqrt{3}}{4}$

Représentation d'une expression sous forme d'arbre :



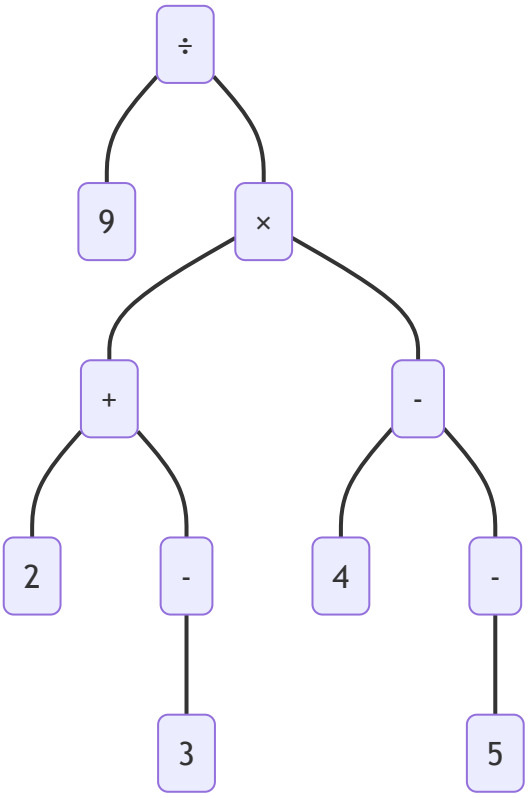
Parcours de l'arbre :

- Infixe : $(1 + (2 \times \sqrt{3})) / 4$
- Préfixe : $/ (+ 1 (\times 2 \sqrt{3})) 4$
- Postfixe : $(1 (2 (3 \sqrt{}) \times) +) 4 /$

Notation polonaise inversée (NPI):

- Notation postfixé
- Nécessité de deux piles :
  - pile d'expression
  - pile de calcul

$$\frac{9}{(2+(-3))\times(4-(-5))}$$



Infixe : $9 \div ((2 + (-3)) \times (4 - (-5)))$

Préfixe : $\div 9 (\times (+ 2 (- 3)) (- 4 (- 5)))$

Postfixe : $9 ((2 (3 -) +) (4 (5 -) -) \times) \div$

Etape	1	2	3	4	5	6	7	8	9	10	11	12
Mot lu	9	2	3	neg	+	4	5	neg	−	×	÷	
Pile	9	2	3	−3	−1	−1	5	−5	9	−9	−9	−1
		9	2	2	9	9	4	4	−1	9	9	
			9	9			−1	−1	9			
			9	9				9				

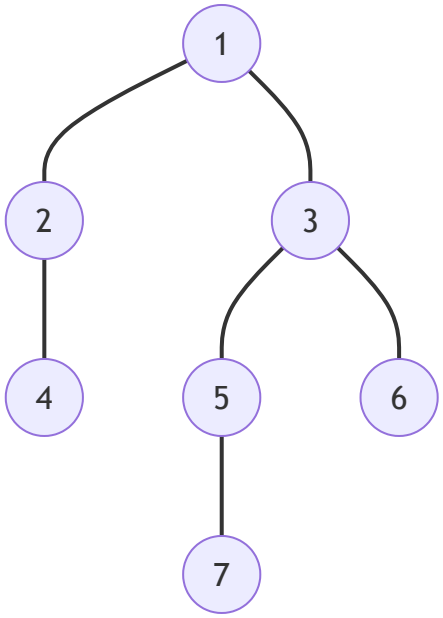
Parcours de l'arbre → utilisation d'une pile

Visite des fils droits avant les fils gauches pour retrouver l'ordre préfixe (fils gauche en somme de pile)

Pile :

1	
3	2
3	4
3	
6	5
6	7
6	

Clés traité: $1 \to 2 \to 4 \to 3 \to 5 \to 7 \to 6$



Bilan

Parcours d'un arbre → fonction récursive

Comportement d'une pile

Depth First Search (DFS)

```
Fonction ParcoursArbre(Élément)
  Entrée: pointeur sur un nœud de l'arbre
  Sortie: traitement sur chacun des nœuds du sous-arbre enraciné en un élément
  Début
    Si Élément != NIL Alors
```

```
    Traitement Élément     != parcours préfixé
    Parcours arbre.gauche
    Traitement Élément     != parcours infixé
    Parcours arbre.droit
    Traitement Élément     != parcours postfixé
  Fin Si
Fin
```

Parcours en largeur

Principe du parcours en largeur :

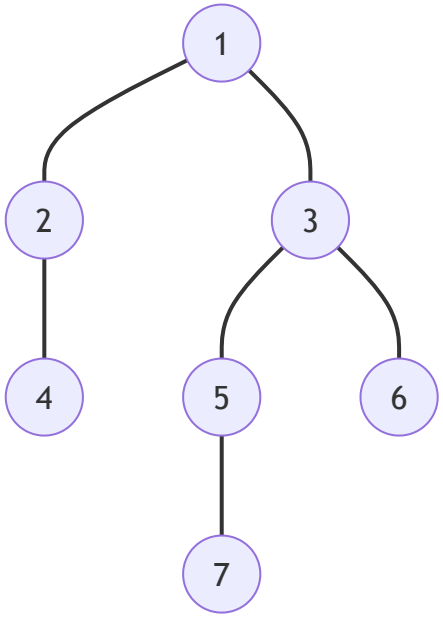
- Départ de la racine
- Exploration des nœuds par niveaux de l'arbre
- Pour un même niveau, parcours de gauche à droite

$\Longrightarrow$ Parcours réalisé à l'aide d'une file

Visite des fils gauches avant les fils droits

Parcours intéressant pour

- une recherche avec la plus petite profondeur possible
- pour une énumération



File:

1		
3	2	
4	3	
6	5	4
6	5	
7	6	
7		

Clés traitées: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7$

IV. Implémentation

Arbres binaires de recherche

Type arbre n'existe pas → sérialisation (stockage sous un autre format)

Deux implémentations possibles :

1. Par tableaux (voir exercice)
2. Par enregistrement et pointeurs

Solution par enregistrement:

- Création du type nœud
- Creation du type arbre pointant sur la racine de l'arbre (en C)
- Arbre complet contenu dans en enregistrement d'enregistrements (en OCAML)

$\Longrightarrow$ structure récursive

Quels champs pour un nœud ?

- entier `clé`
- pointeur `fils_gauche`
- pointeur `fils_droit`
- pointeur `parent`
- enregistrement `informations`

En C, nécessité d'écrire une fonction créant un nœud :

- Entrée:
  - valeur (`int`)
  - info (enregistrement contenu dans le nœud)
- Sortie:
  - pointeur sur le nœud créé (les champs `fils_gauche`, `fils_droit` et `parent` sont initialisé à `NULL`)

Constructeur : créer un arbre

- Entrée : aucun
- Sortie : pointeur sur la racine, qui peut être `NULL`/`NIL`
- Sémantique :
  - Fonction créant un pointeur sur la racine
  - Pointeur sur la racine initialisé à `NULL`/`NIL`