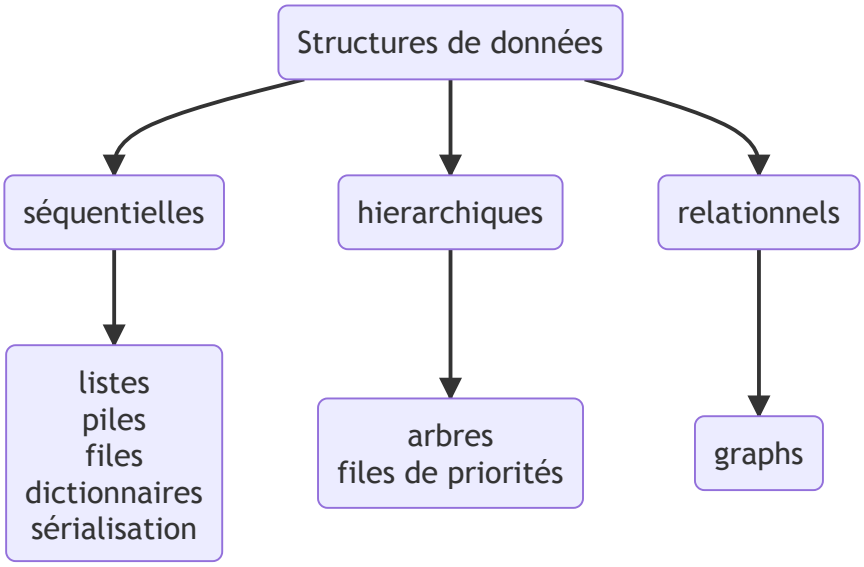
I. Introduction - définition

Utilisation des arbres

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

Classification



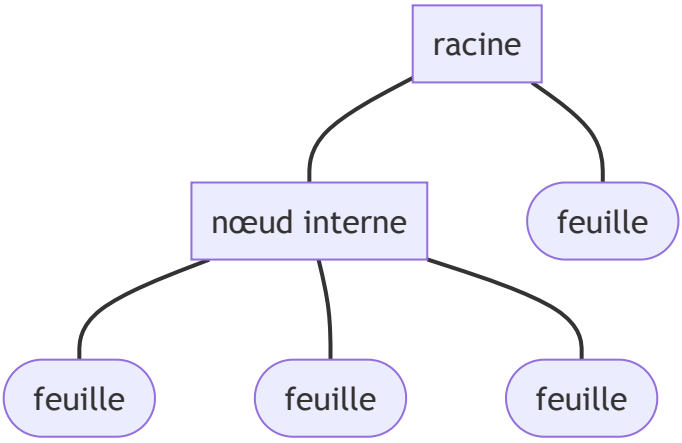
Définitions

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

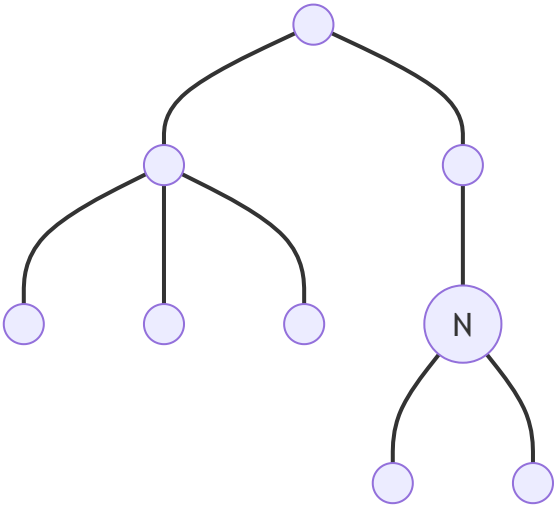


Un *arbre n-aire* est un arbre dont les nœuds sont d'arité maximale n Dans l'exemple, chaque nœud a au maximum une arité de 3 donc l'arbre est un arbre ternaire.

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



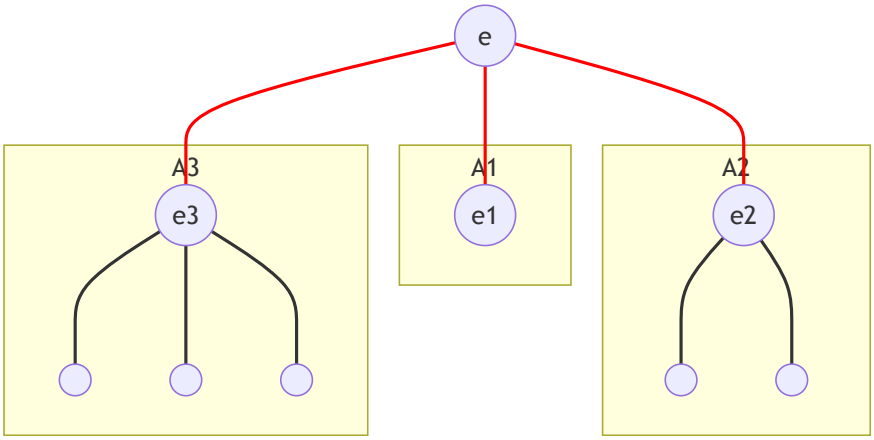
Par exemple, le nœud N a une profondeur de 2.

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.

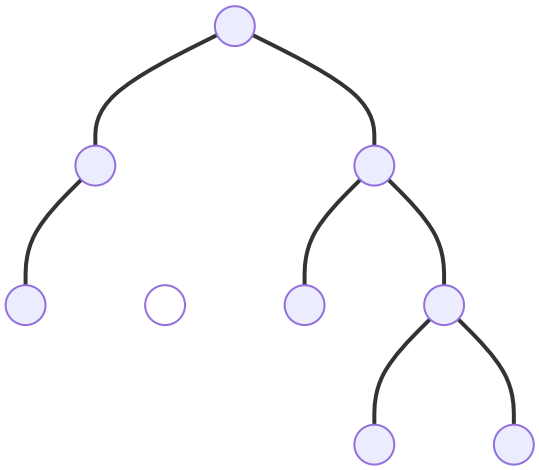
$$\sum_{i=0}^{h} 1 \geqslant n \geqslant \sum_{i=0}^{h} a^i$$
$$\iff h + 1 \geqslant n \geqslant \frac{a^{h+1} - 1}{a - 1}$$

II. Arbres particuliers

Définitions

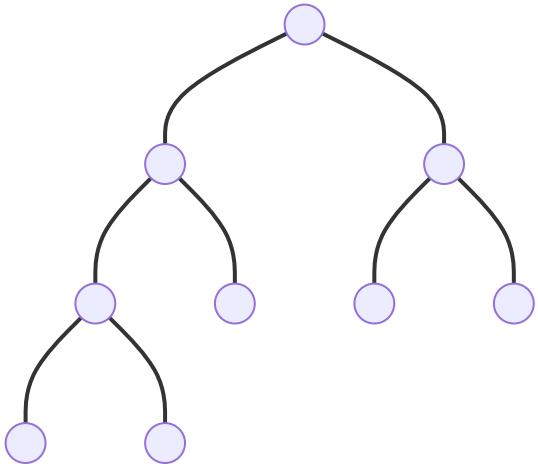
Arbres binaires:

Tous les nœuds sont d'arité au maximum 2.



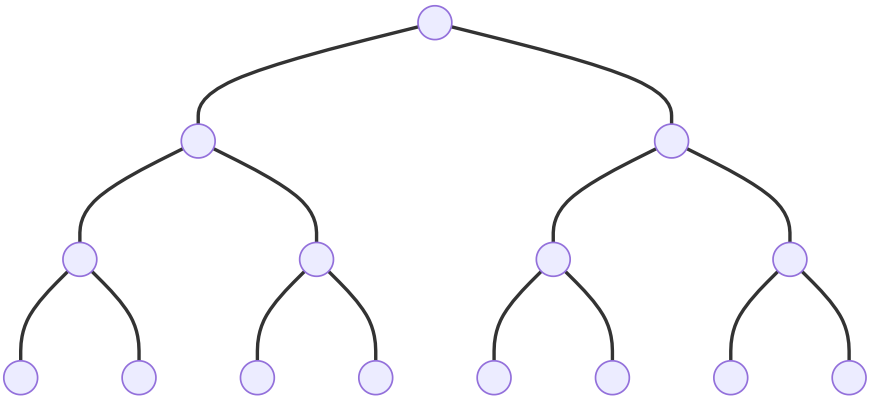
Arbre binaire entier:

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:

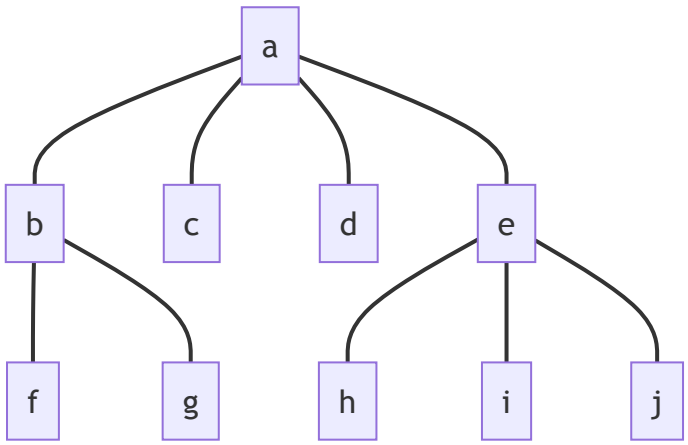
$$\begin{aligned} & \log_a((a-1)\times n+1)-1 \leq h \\ \Longrightarrow\ & \log_a((a-1)\times n) < \log_a((a-1)\times n+1) \leq h+1 \\ \Longrightarrow\ & \lfloor \log_a((a-1)\times n)\rfloor < h+1 \\ \Longrightarrow\ & \lfloor \log_a((a-1)\times n)\rfloor \leq h \end{aligned}$$

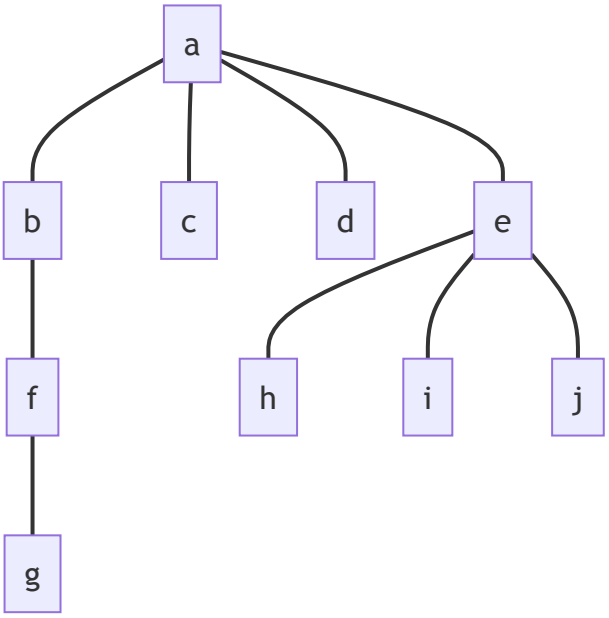
soit $\lfloor \log_a((a-1)\times n)\rfloor \leq h \leq n-1$

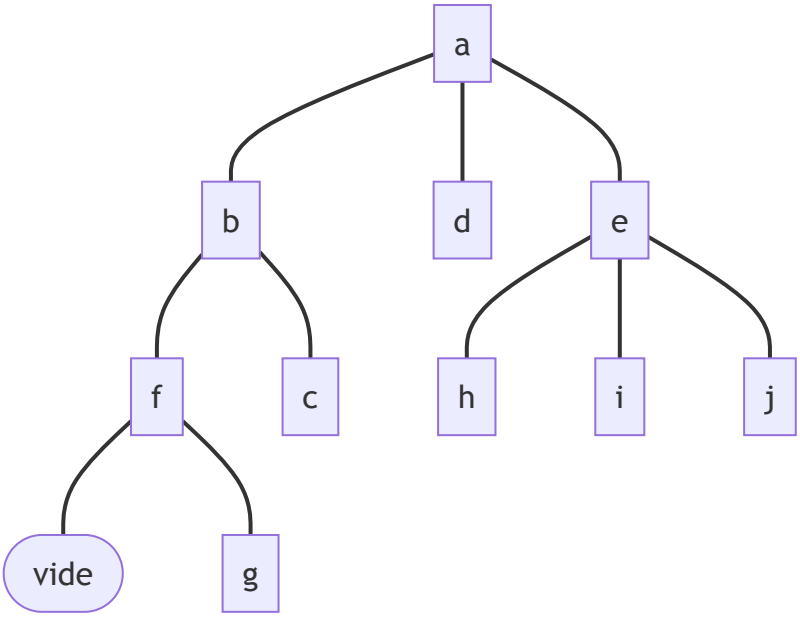
Transformation d'un arbre en arbre binaire

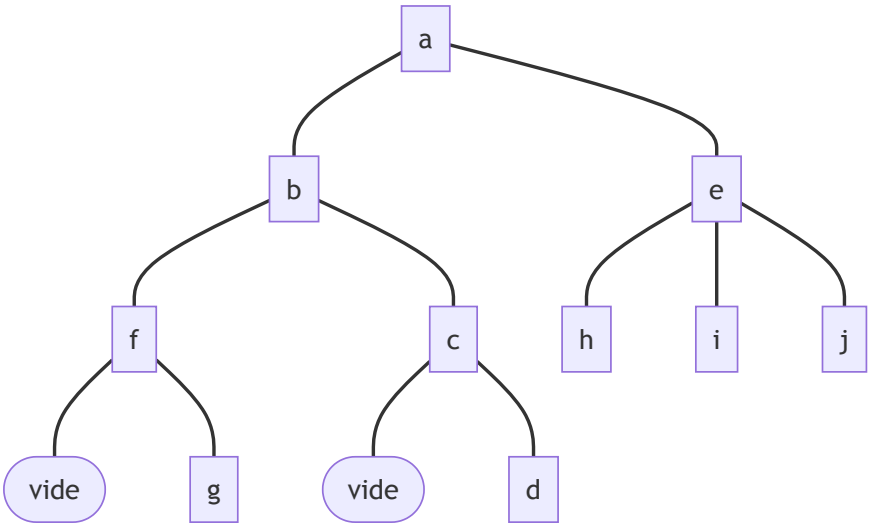
Algorithme de transformation d'un arbre n-aire A en arbre binaire:

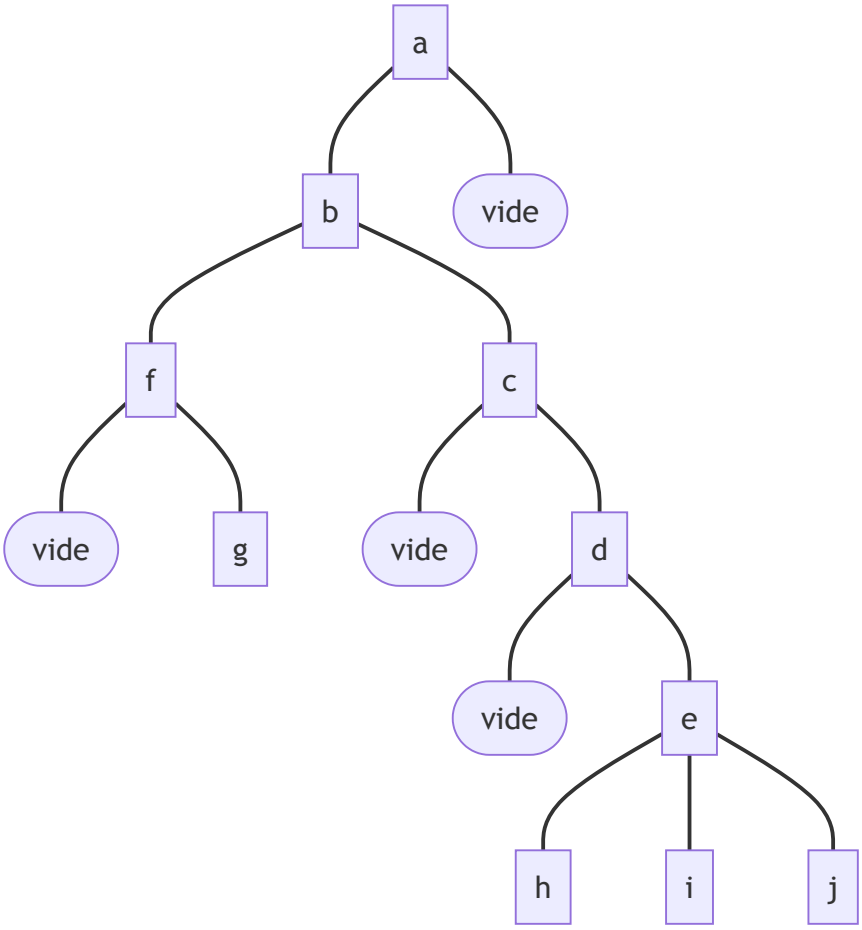
```
Pour chaque e ∈ A
Laisser e_g (fils le plus à gauche) en place
Supprimer les autres fils de e et les chainer à e_g
Le nœud e_d est le prochain frère de e
Fin pour
```

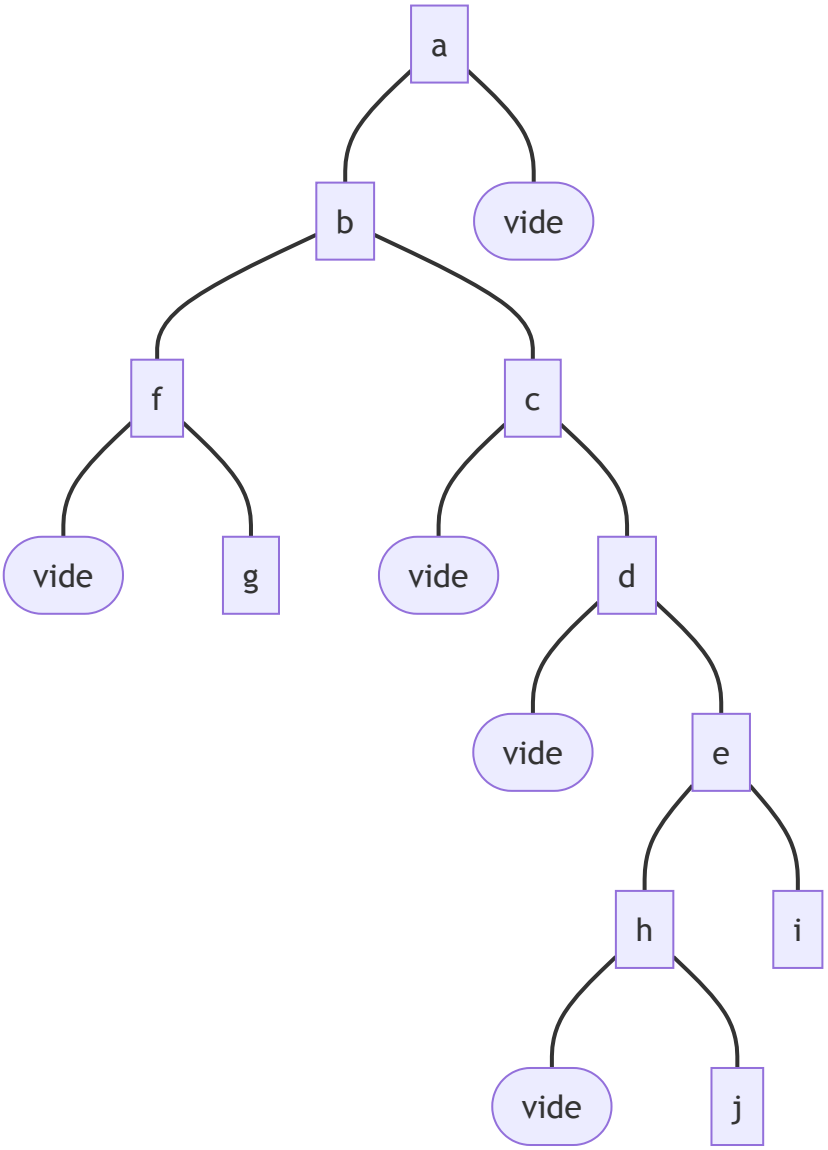


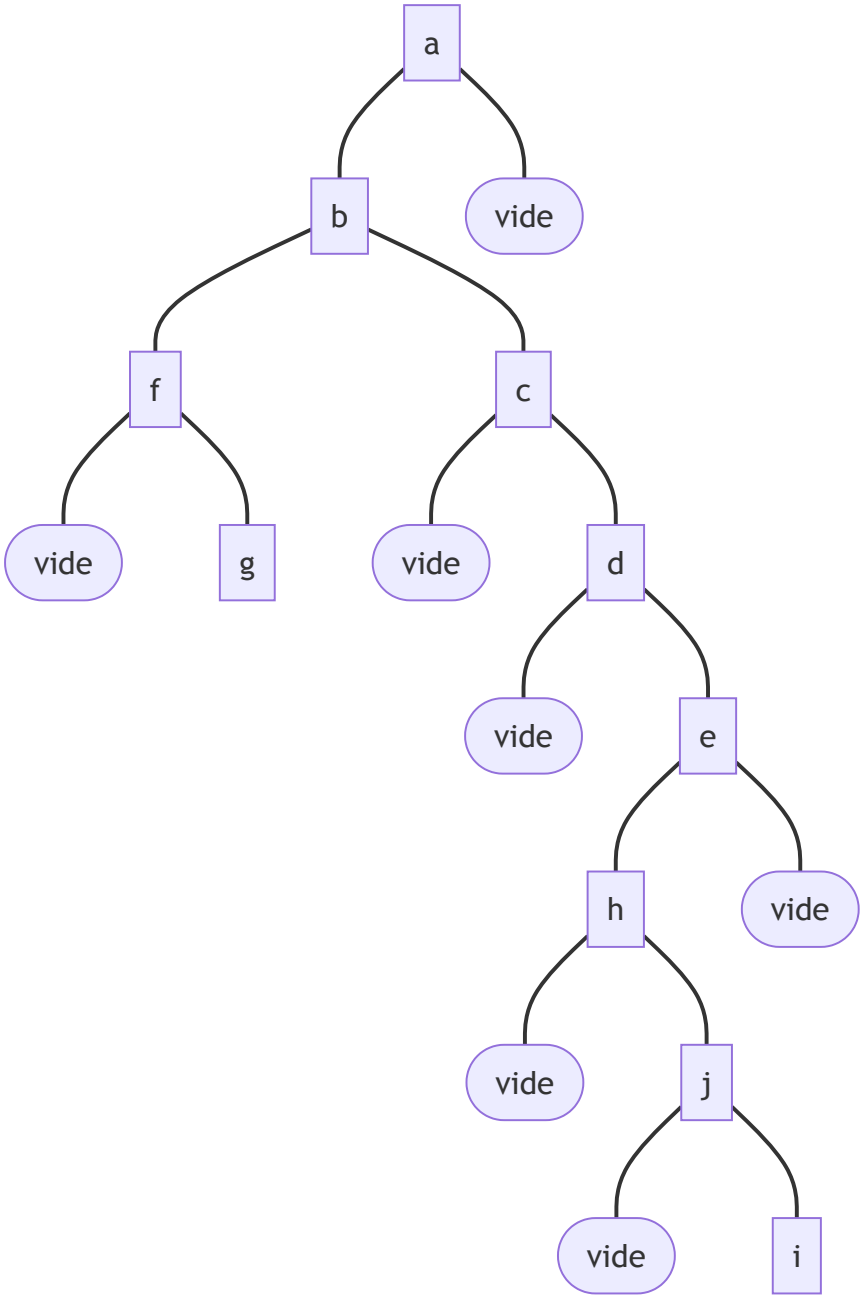

a
b
c
d
e
f
h
i
j
g


a
b
d
e
f
c
h
i
j
vide
g

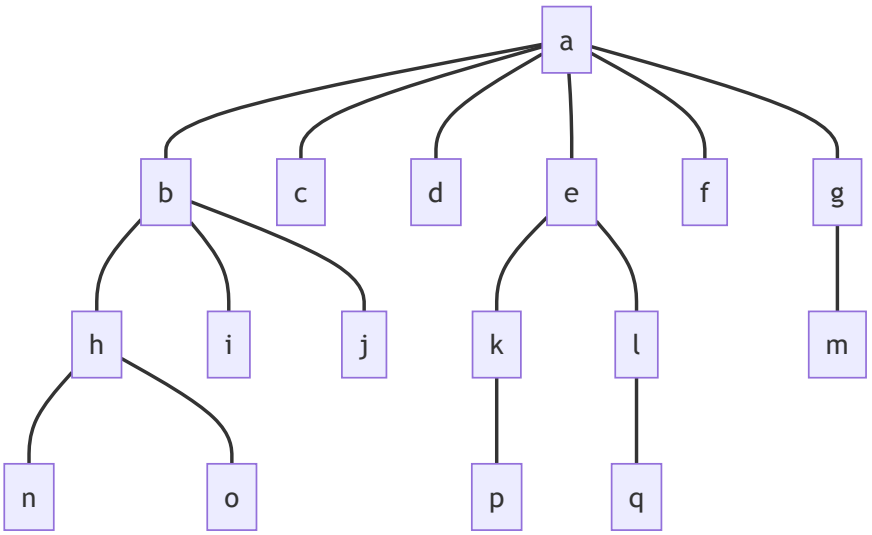

a
b
e
f
c
h
i
j
vide
g
vide
d

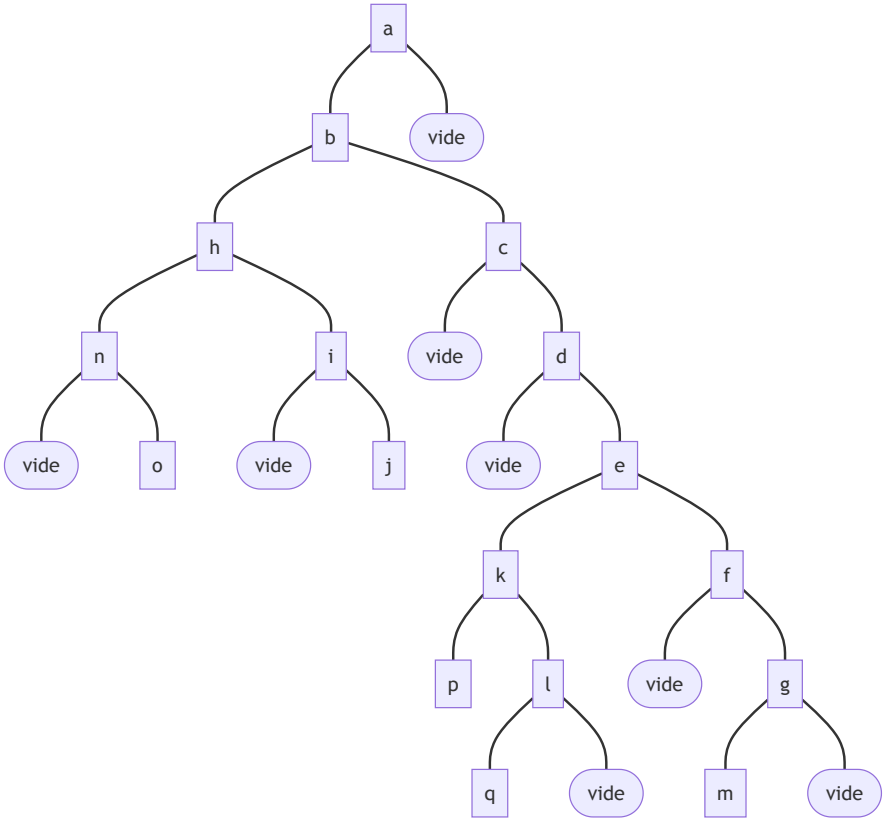

a
b
vide
f
c
vide
g
vide
d
vide
e
h
i
j


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)

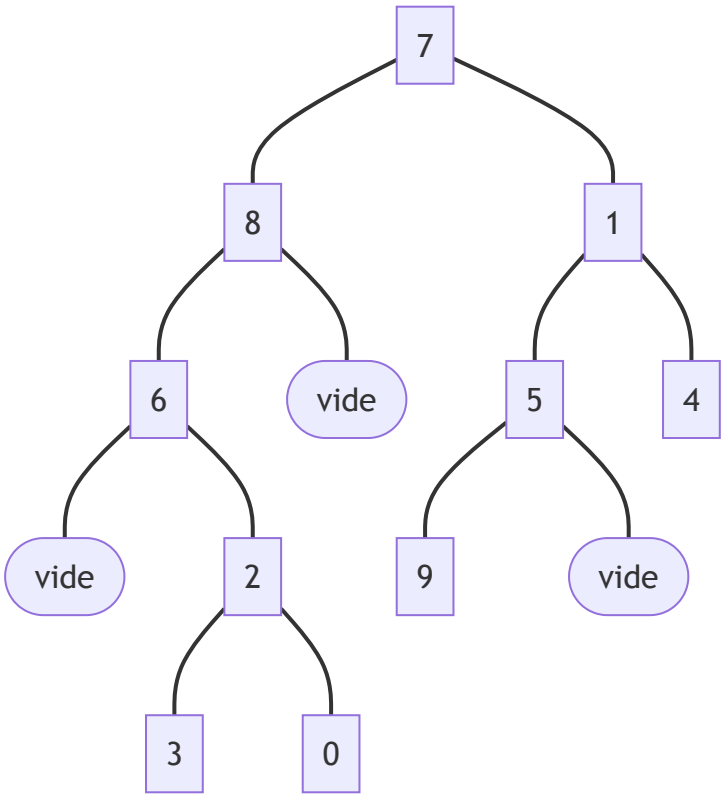
On stocke un arbre binaire dans un tableau d'enregistrements Chaque enregistrement E est défini par:

- 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

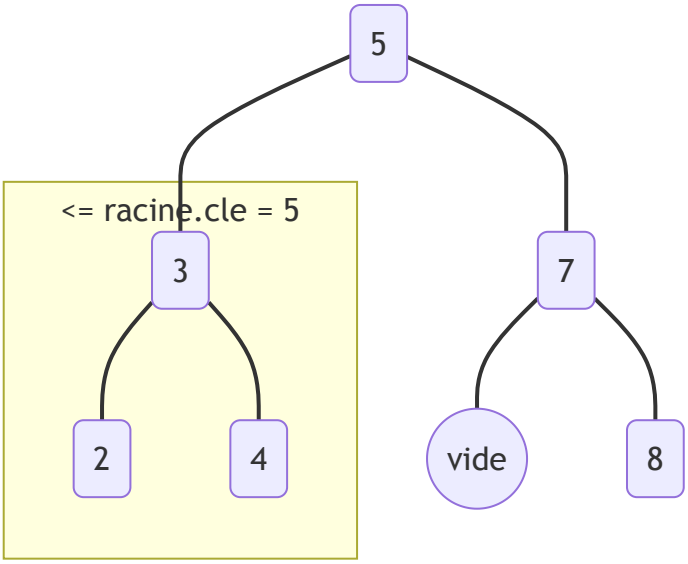
Arbre binaire marqué:

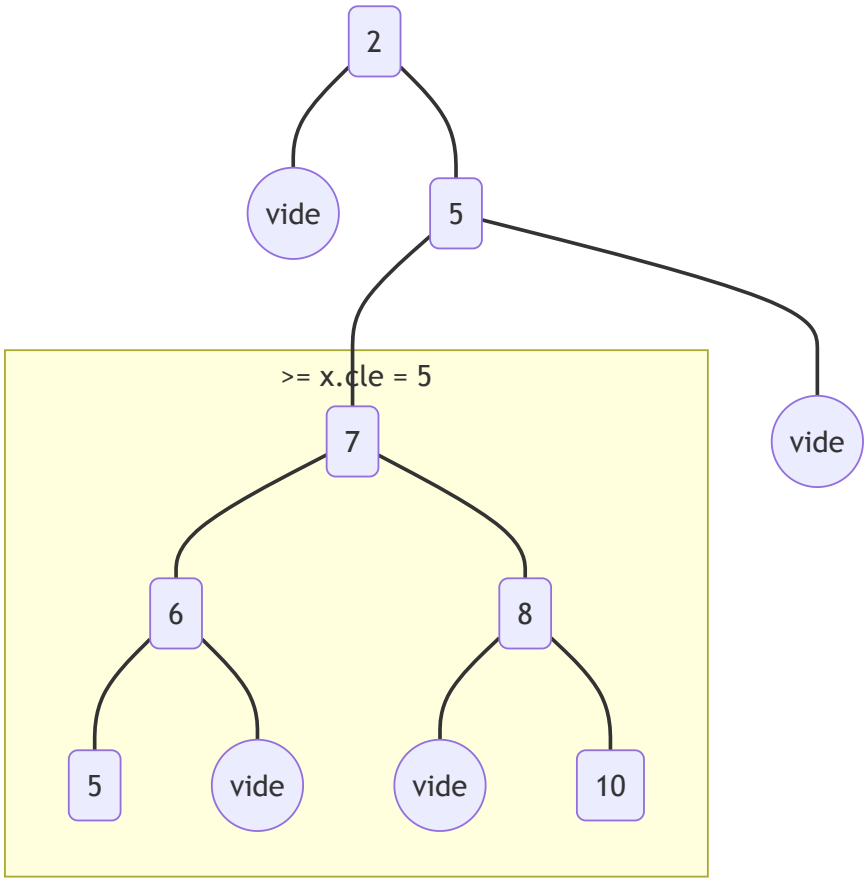
- Présence d'une étiquette pour chaque nœud x de A
- Étiquettes
  - Même type
  - Présence d'une clé appartenant à un ensemble E
  - Relation d'ordre total sur E
  - Notation x.clé

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

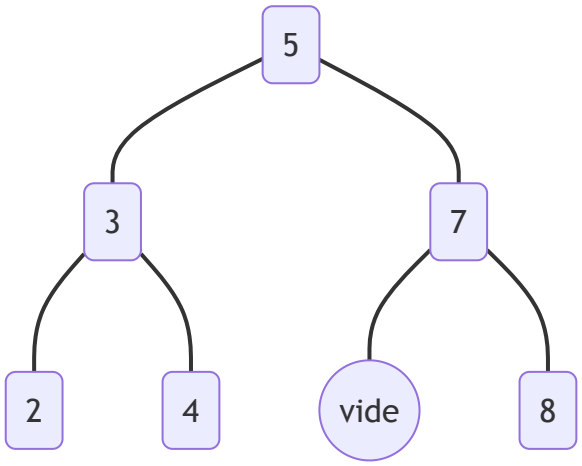
Opérations fondamentales des ABR:

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

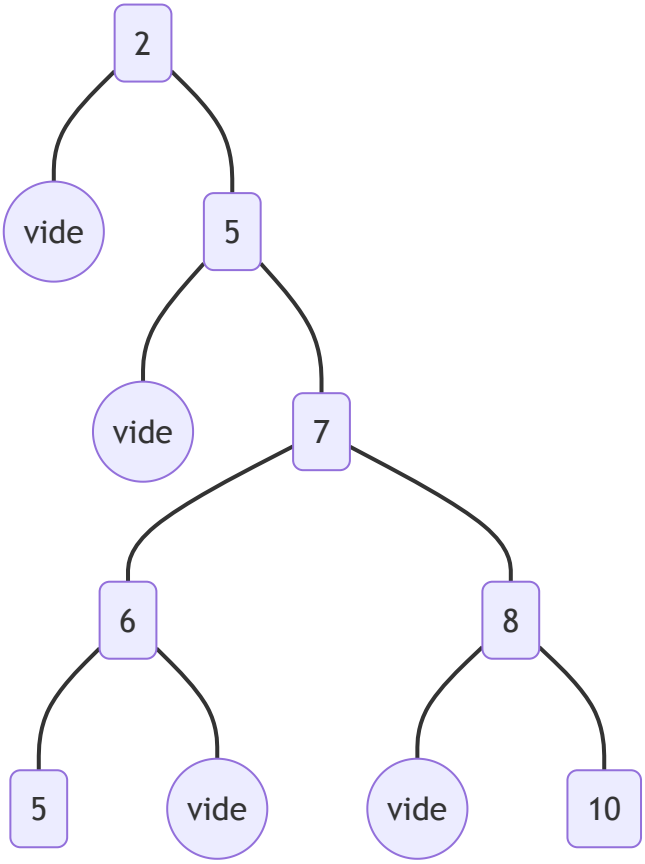
- Transformateurs
  - Insérer un nœud
  - Supprimer un nœud

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

- temps de recherche proportionnel à la hauteur h de l'arbre
  - $T(n) = \mathcal{O}(n)$ recherche linéaire dans le pire des cas (liste chainée)
  - $T(n) = \mathcal{O}\big(\log_2(n)\big)$ 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 $\mathcal{O}(n)$ 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:

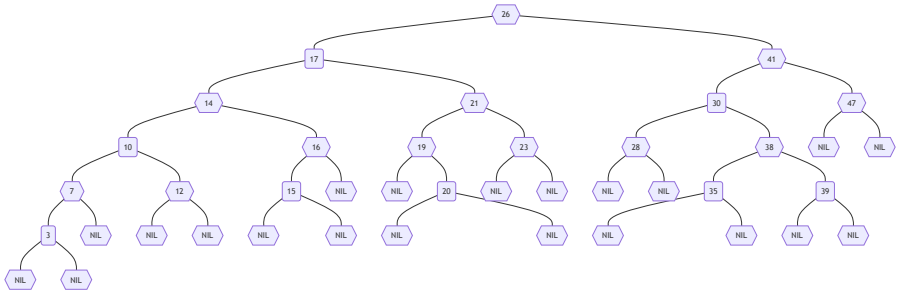
- chaque nœud est soit rouge, soit noir
- chaque feuille est noire
- la racine est noire
- si un nœud est rouge, ses deux enfants sont noirs
- pour chaque nœud, tous les chemins simples reliant le nœud à des feuilles situées plus bas dans l'arbre contiennent le même nombre de nœuds noirs

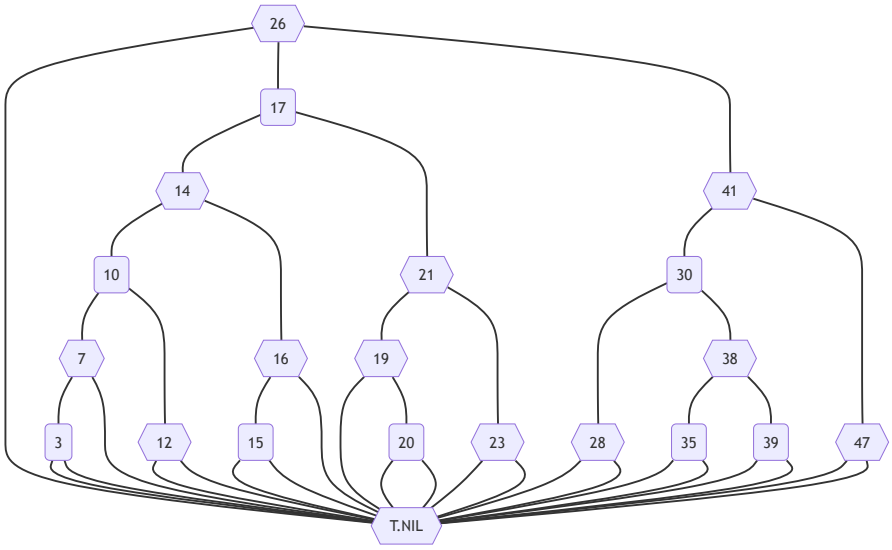
Remarque:

- 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 ⟨⟩

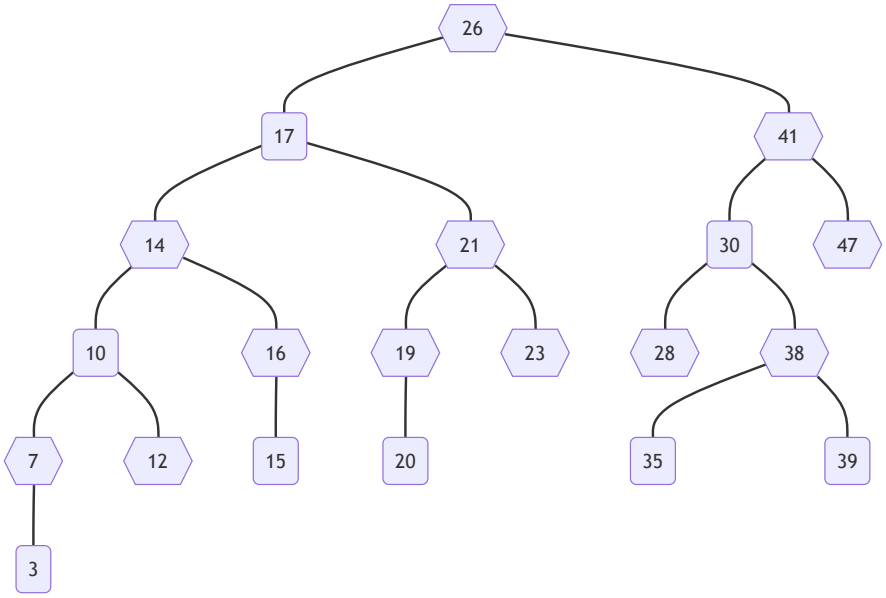




T.NIL : sentinelle

- 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:

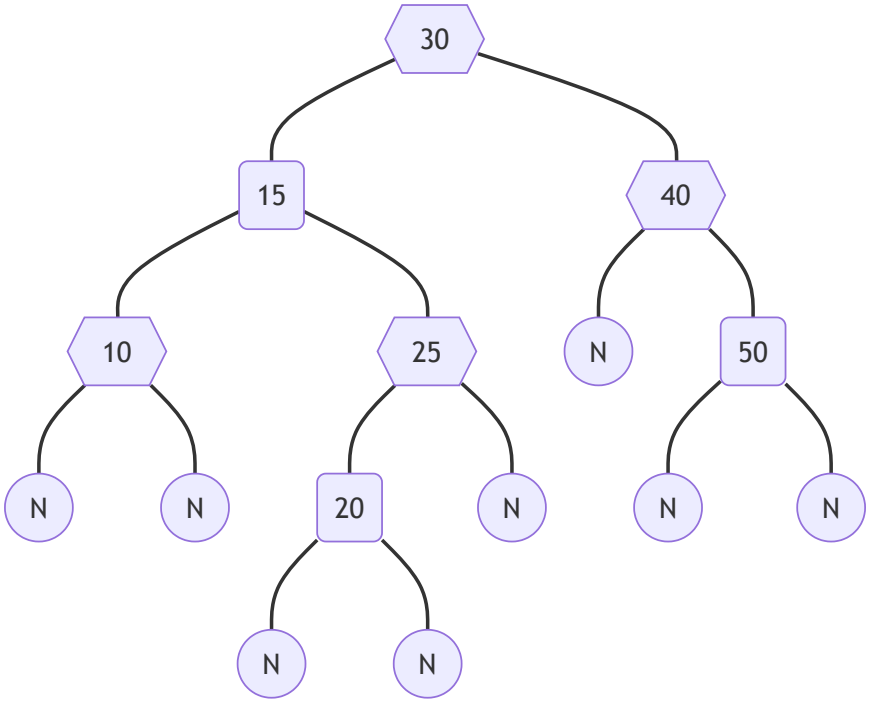
Si tous les nœuds sont noirs, alors h_n est égale à la profondeur de l'arbre.

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

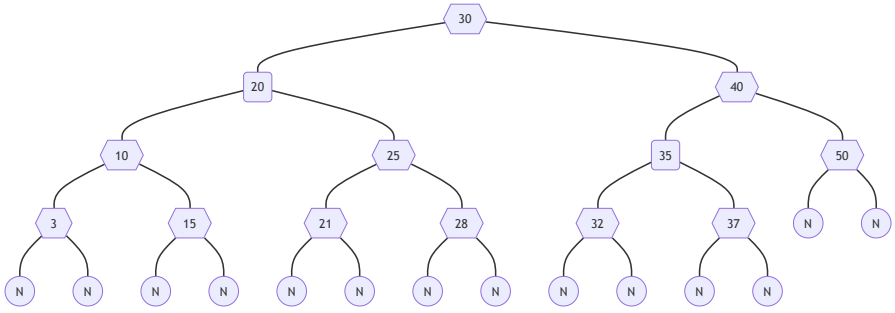
Exercice 1:

- 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.

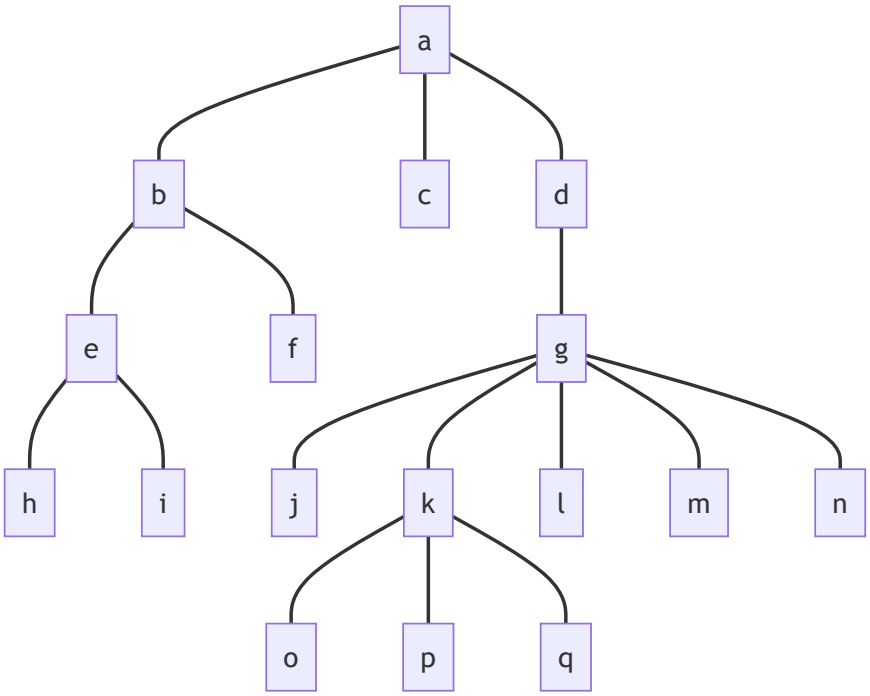
Ordre préfixé:

- 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 :

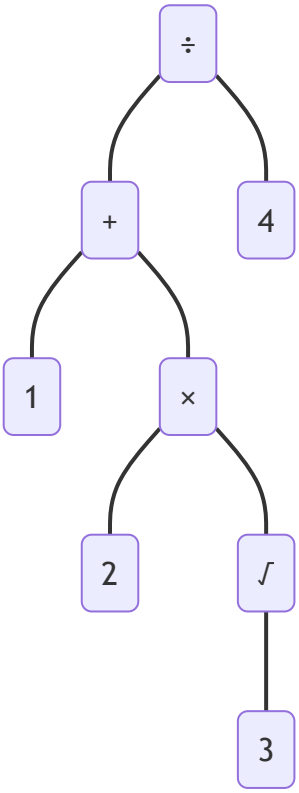
$$h \to i \to e \to f \to b \to c \to j \to o \to p \to q \to k \to l \to m \to n \to g \to d \to a$$

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 :



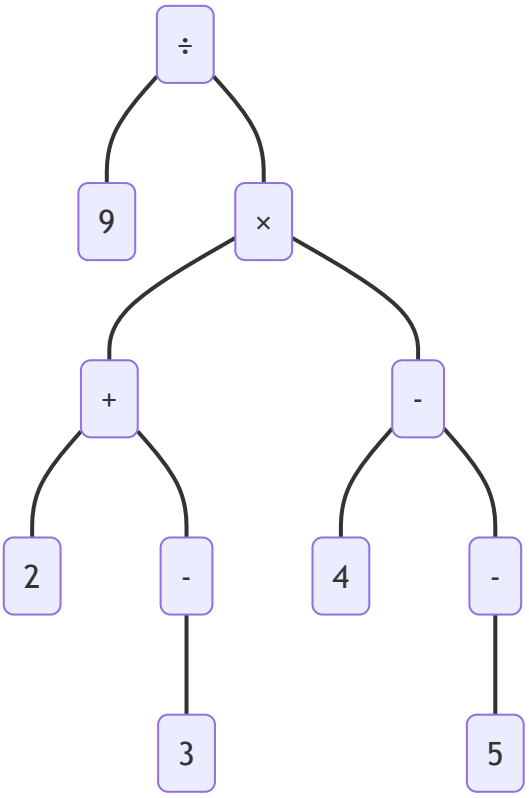
Parcours de l'arbre :

- 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				

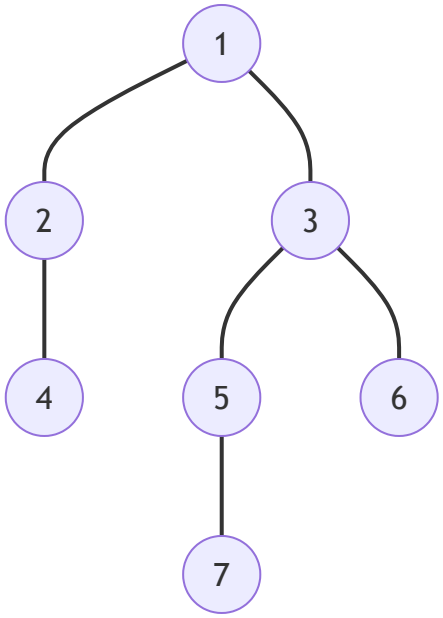
Parcours de l'arbre → utilisation d'une pile

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 :

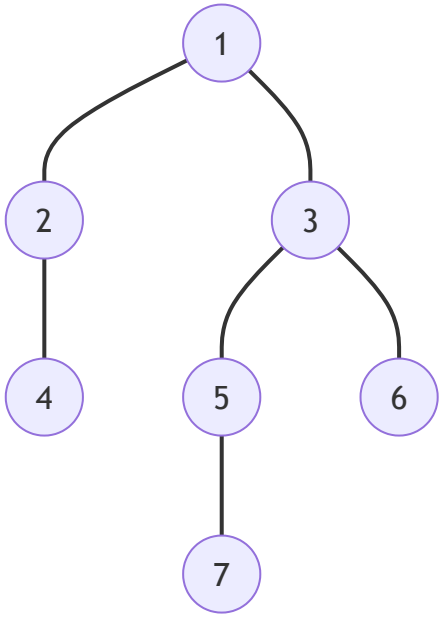
- Départ de la racine
- Exploration des nœuds par niveaux de l'arbre
- Pour un même niveau, parcours de gauche à droite

$\Longrightarrow$ 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 un 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`

Transformateur : insérer un nouveau nœud → nouvelle feuille. Plusieurs cas à envisager :

- Arbre vide → insérer au niveau de la racine
- Arbre non vide
  - Début de la racine
  - Parcours de l'arbre par chemin descendant à la recherche d'un `NULL/NIL`
  - Pointeur sur nœud, descendant l'arbre jusqu'à `NIL` (pointeur de traine)
  - Mise à jour du parent de nouveau
  - Mise à jour du fils gauche / droit du futur parent
  - Nécessité d'un pointeur temporaire conservant l'adresse du futur parent
  - Mise à jour dans la boucle du pointeur sur parent

  $\implies$ Deux phases :

  1. Recherche de la feuille
  2. Modification de l'arbre / mise à jours des nœuds

  Cette fonction est linéaire en temps : $\mathcal{O}(h)$ où h est la hauteur de l'arbre.

Transformateur : Supprimer un nœud

- Argument : pointeur sur l'arbre, valeur de la clé
- Retourne : arbre modifié
- Sémantique : ?

  Quel cas envisager ?

  Il faut vérifier que la clé est présent dans l'arbre

  Suppression d'une feuille

  Suppression d'un nœud interne / racine ayant un seul enfant

  Suppression d'un nœud interne / racine ayant deux enfants

- Suppression d'une feuille : modification du parent et suppression de la feuille
- Suppression d'un nœud interne / racine ayant un seul enfant

  - modification du parent (Parent.gauche ou Parent.doit pointe sur l'enfant non nul du nœud supprimé)
  - modification de l'enfant (Enfant.parent pointe sur le parent du nœud supprimé)

- Suppression d'un nœud interne / racine ayant deux enfants
  - Enfant droit est le *successeur* (la définition est après) du nœud à supprimer
  - Enfant droit devient le fils de Parent à la place du nœud à supprimer: mise à jour de Parent.gauche et de Enfant droit.parent
  - Enfant gauche devient le fils gauche de Enfant droit : mise à jours de Enfant droit.gauche et Enfant gauche.parent

Prédécesseurs et successeurs : Si toutes les clés sont distinctes, le successeur d'un nœud x est le nœud y possédant la plus petite clé supérieure à x.clé

Les algorithmes sont sur la feuille "Structures hiérarchiques — Arbres" mais uniquement pour information : on n'a pas à savoir les écrire.

Arbre bicolore

Modification du type : ajout d'un champ pour désigner la couleur

Opérations d'insertion et de suppression :

→ visite des nœuds pour préserver les propriétés des arbres bicolores

NON TRAITÉ (aucune indication dans le programme)

Équilibrage des arbres (arbres binaires de recherche)

Préservation des propriétés de l'arbre binaire de recherche

Utile pour équilibrage des arbres

→ se rapprocher d'un arbre entier

→ éviter les arbres filiformes (car la recherche est en $\mathcal{O}(n)$)

Arbre H-équilibré:

Déséquilibre d'un arbre en a :

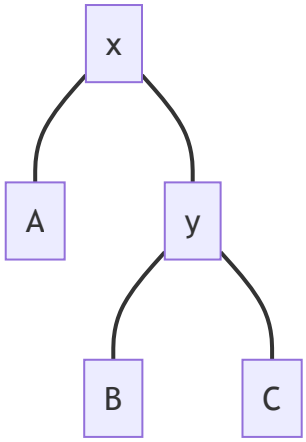
$$\text{déséquilibre}(a) = h\big(\text{gauche}(a)\big) - h\big(\text{droit}(a)\big)$$

Un arbre a est H-équilibré si, pour tous ses sous–arbres b on a :

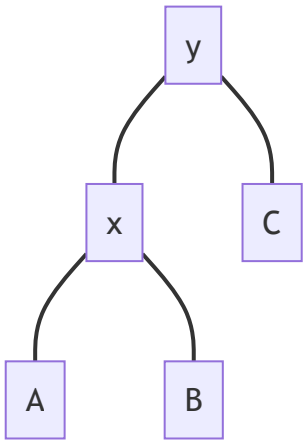
$$\text{déséquilibre}(b) \in \{-1, 0, 1\}$$

Un arbre AVL (Adelson – Velskii – Landis en 1962) est un arbre H-équilibré.

Arbre initial :



Après une rotation gauche, on a



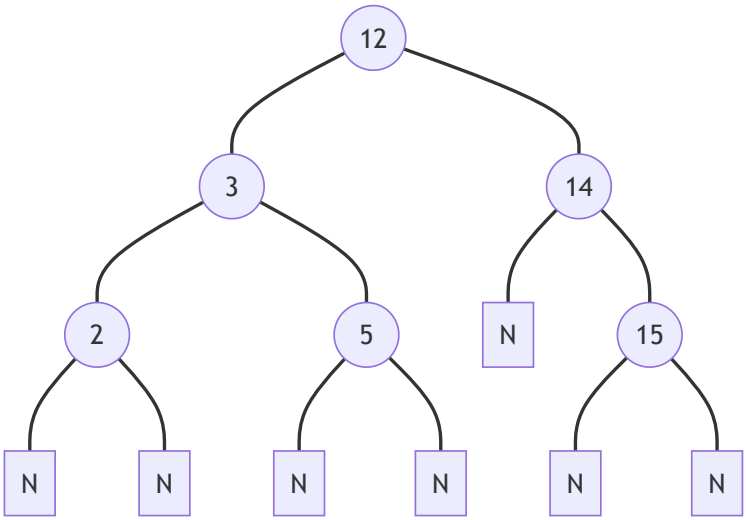
Après une rotation droite, on retrouve l'arbre initial.

Rotation gauche : x va à droite

Rotation droite : y va à gauche

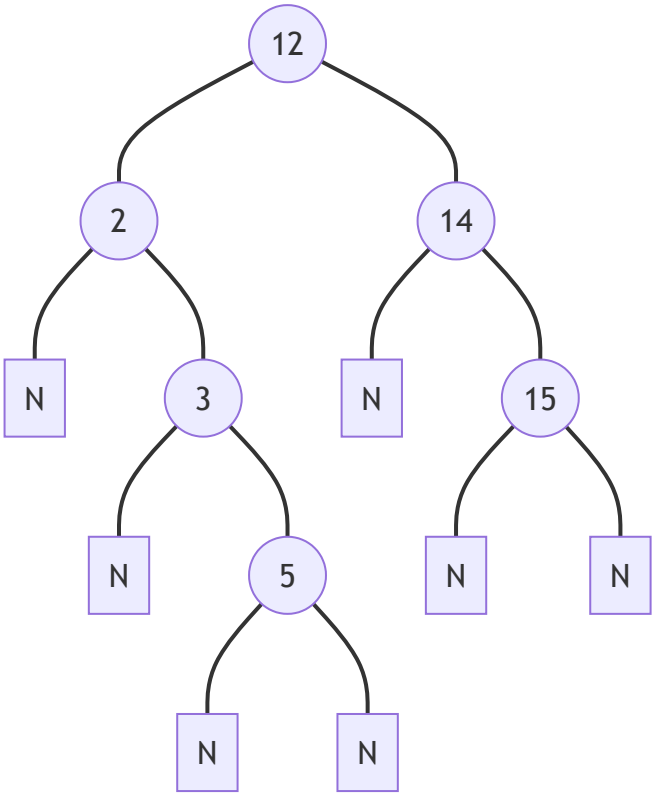
Exercice 1 :

1. Arbre A_1
    a. déséquilibre$(A_1) = 3 - 3 = 0$, il est donc H-équilibré

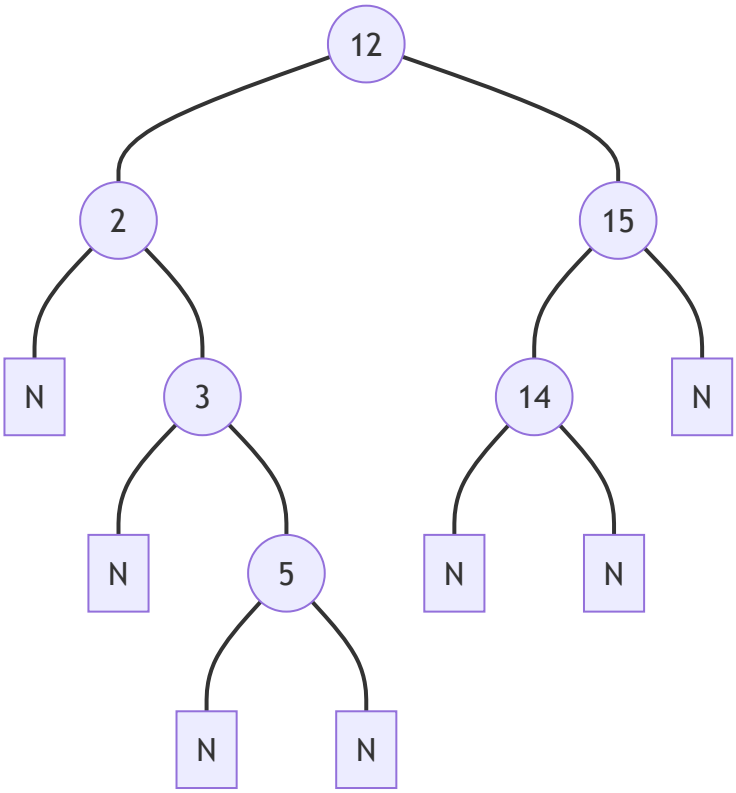


b.

Après une rotation à droite de centre le nœud 3 :



Après la rotation à gauche de centre le nœud 14 :



2. Arbre A_2
   a. déséquilibre$(A_2) = 0$
   b. Rotation à gauche de centre 5 puis à droite de centre 14

V. Tas et files de priorité

Tas : définition

Un tas est un arbre binaire :

- tel que (condition sur le squelette) : toutes les feuilles sont de profondeur h ou $h-1$. Pour toute profondeur $p < h$, il y a exactement 2^p nœuds. Toutes les feuilles (de profondeur h) sont le plus à gauche de l'arbre
- tournoi (condition sur les étiquettes) : l'étiquette d'un nœud est supérieure (inférieure) ou égale à celles de ses fils

Tas max : l'étiquette d'un nœud est supérieure ou égale à celles de ses fils.

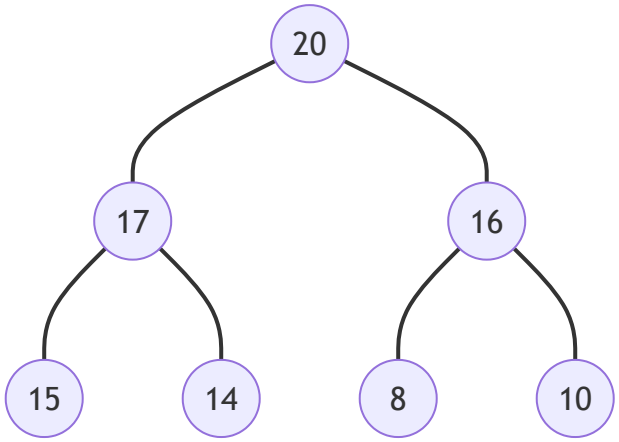
Tas min : l'étiquette d'un nœud est inférieure ou égale à celles de ses fils.

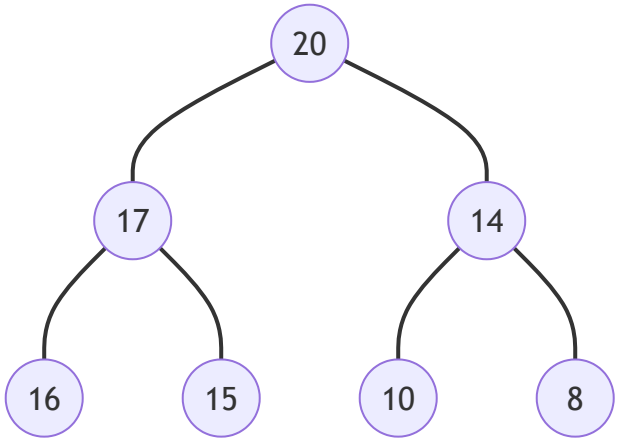
Structure bin adaptée pour gérer les files de priorité :

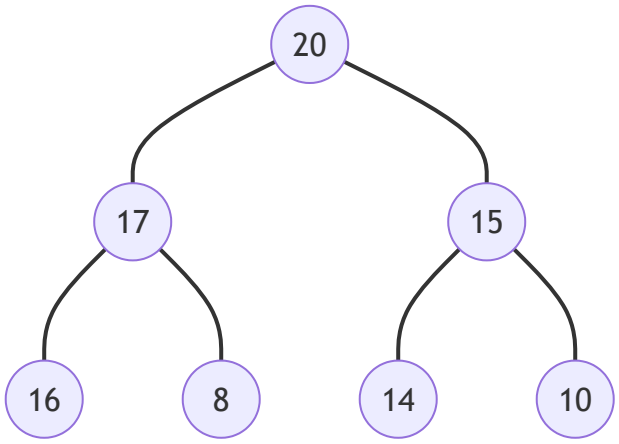
implémentation : (priorité, élément)

la priorité sert de clé

Exercice : construire 3 tas max contenant les nœuds de clés $\{8, 10, 14, 15, 16, 17, 20\}$







Implémentation des tas

Implémentation dans un tableau :

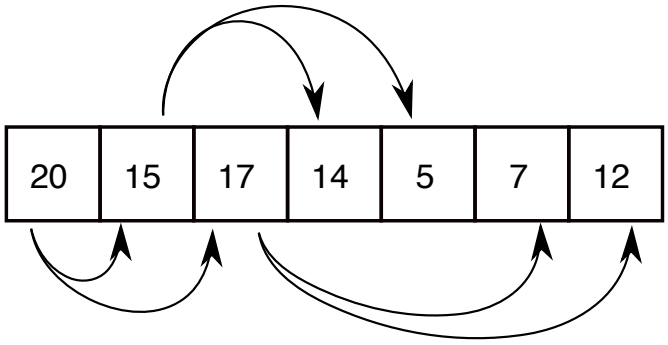
- Chaque nœud correspond à un élément du tableau
- Fils gauche d'un nœud i stocké à l'indice $2i$
- Fils droit d'un nœud i stocké à l'indice $2i + 1$
- Parent d'un nœud i stocké à l'indice $\left\lfloor \frac{i}{2} \right\rfloor$

Tableau A représentant un tas : deux attributs (algorithmique) :

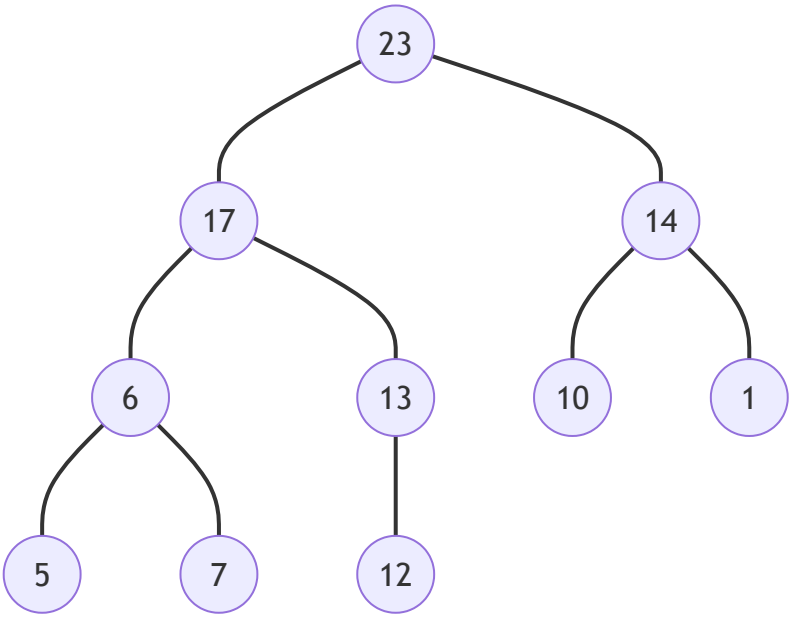
- A.longueur : nombre d'éléments dans un tableau,
- A.taille : nombre d'éléments dans le tas,
- Le premier élément du tableau A est la racine du tas,
- On a A.longueur $\leqslant$ A.taille.

Possibilités :

- Stocker la taille du tas à l'indice 0 du tableau lors de la programmation,
- Définir un enregistrement contenant la taille du tas et le tableau.



Exercice 2:

1. Au maximum, on a un arbre complet de hauteur h donc $2^{h+1} - 1$ nœuds. Au minimum, on a un arbre complet de hauteur $h - 1$ et le nœud le plus à gauche à un fils gauche. On a donc au minimum 2^h nœuds.
2. 

   Cet arbre n'est pas un tas max car $7 > 6$. Il faut donc enlever 7 et 12 :

$$[\ 23, 17, 14, 6, 13, 10, 1, 5\ ]$$

La taille de ce tas est 8.

3. On suppose que le nœud stocké en $\left\lfloor \frac{n}{2} \right\rfloor + 1$ est un parent. Alors, son fils gauche est stocké en $2\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)$. Or

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 > \frac{n}{2} \iff 2\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) > n.$$

Donc, l'indice étant en dehors du tableau, il n'a pas de fils. C'est donc une feuille.

Soit $k \in [\![1, \left\lfloor \frac{n}{2} \right\rfloor]\!]$. $k \times 2 \leqslant 2$ donc k a un fils donc ce n'est pas une feuille.

Opérations sur les tas

Trois opérations

1. Préservation de la propriété des tas max

   Reçoit un tableau et un indice du tableau

   Modifie le tableau pour qu'il contienne un tas max

2. Construction d'un tas max

   Reçoit un tableau

   Retourne un tas max

3. Tri par tas

   Tri d'un tableau par ordre croissant (décroissant) en utilisant les propriétés d'un tas max (min).

Conservation de la propriété des tas max

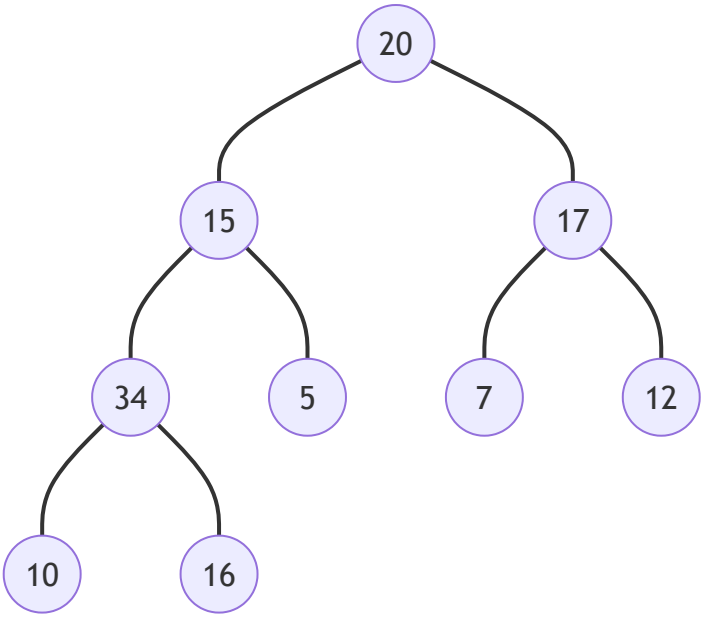
(code sur la feuille)

État de l'arbre et du tableau pour l'appel de `entasserMax(A, 2)` *avec*

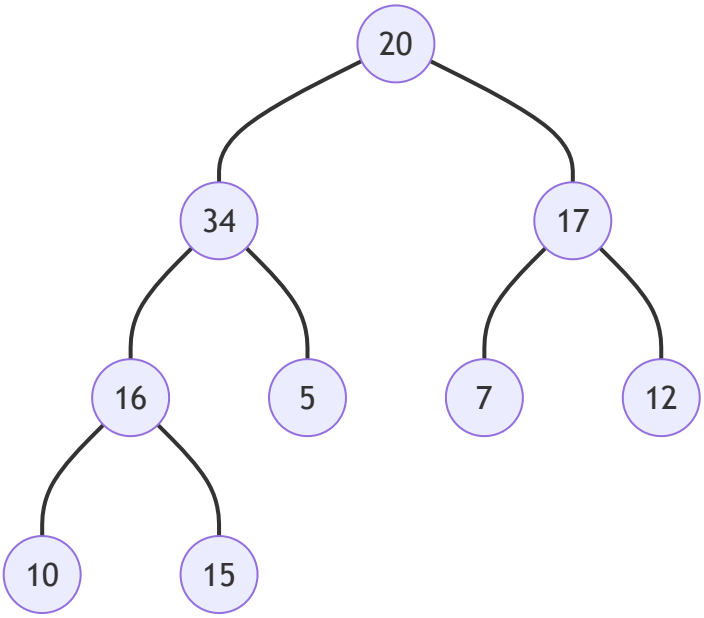
$$A = [\ 20, 15, 17, 34, 5, 7, 12, 10, 16\ ] \quad ?$$

Après une itération, on a $A = [\ 20, 34, 17, 15, 5, 7, 12, 10, 16\ ]$.

Après deux itération, on a $A = [\ 20, 34, 17, 16, 5, 7, 12, 10, 15\ ]$.



devient



Que représente i pour l'arbre résultant ?

Le i représente l'indice dans le tableau de l'arbre de la racine du tas max que l'on veut former.

Fonction `entasserMax` :

- Reçoit un tableau A représentant un tas,
- Reçoit un i,
- Retourne A modifié.

- `entasserMax` fait l'hypothèse que les sous arbres gauche et droit du nœud i sont des tais mais que le nœud i est peut être plus petit que ses enfant.

- `entasserMax` fait alors descendre le nœud i jusqu'à ce que la propriété de tas max soit rétablie localement.

- Complexité en $\mathcal{O}(\ln n) = \mathcal{O}(h)$

Fonction `construireTasMax`:

- Construction d'un tas,
- Conversion en tas max,
- Invariant : au début de chaque itération de la boucle pour, chaque nœud $i+1, i+2, \ldots, n$ est la racine d'un tas max,
- Complexité en $T(n) = \mathcal{O}(n \ln n)$.

Tri par tas (*heap sort*)

On fait remonter la dernière valeur du tableau en l'interchangeant avec la racine du tas (la plus grande valeur). Pour éviter de faire remonter la plus grande valeur à la racine, on diminue la taille du tas. On recrée un tas local en replaçant la valeur maximale à la racine.

Première étape :

> Construction du tas max
> La première valeur est la racine donc l'élément de clé la plus grande.
> On place cet élément en dernier.

Seconde étape :

> On rétablit la propriété de tas max sur le tableau privé du dernier élément.

Répétition du processus jusqu'à arriver à un tas de taille 2.

Complexité en $\mathcal{O}(n \ln n)$.

Files de priorité

Exemple d'utilisation très répandue des tas $\rightarrow$ gestion de files de priorités

Objectifs : classer des événements par ordre de priorité

Nœuds → association (priorité, élément)

Existence de files de priorités min et de files de priorités max

Exemple d'utilisation de files de priorités

- Ordonnancement des tâches
- Ordinateurs multi threads (programme de spé.)
- Parcours en largeur de graphs (Dijkstra, A^*, Prim, ...)

Opérations sur les files de priorités

Opérations sur les tas max → restent accessibles :

- `entasserMax(A, i)`
- `construireTasMax(A)`

Opérations propres aux files max :

- `maximumTas(A)`
- `extraireMaxTas(A)`
- `augmenterCle(A, i, cle)`
- `insererTasMax(A, cle)`

Opération propres aux files min :

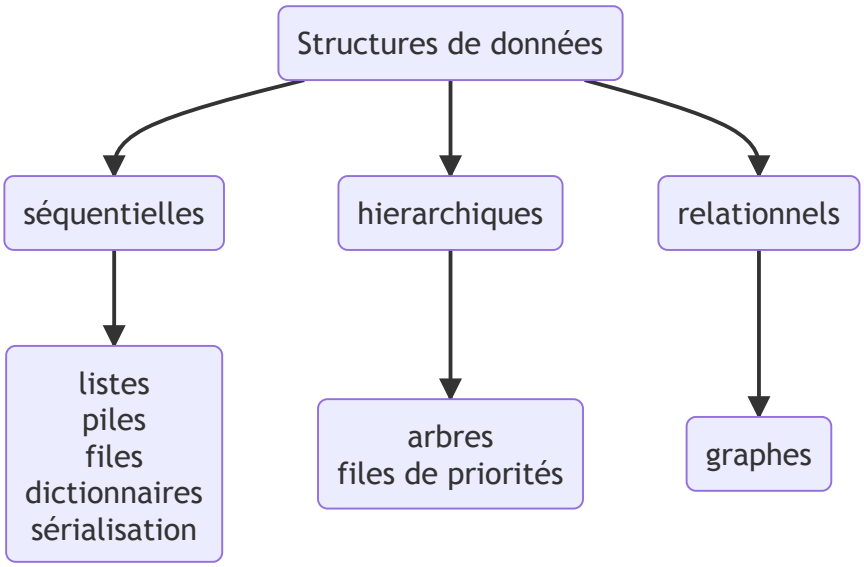
- `minimumTas(A)`
- `extraireMinTas(A)`
- `diminuerCle(A, i, cle)`
- `insererTasMin(cle)`

Opérations propres aux files de priorité max

- Opération `maximumTax(A)` :
  Entrée : tableau `A`
  Sortie : élément de priorité maximale
  Sémantique :
  Retourne -1 si le tas est vide
  Retourne la racine sinon
  Ne supprime pas les éléments du tas
- Opération `extraireMaxTas(A)` :
  Entrée : tableau `A`
  Sortie : élément de priorité maximale, tableau modifié
  Sémantique :
  Retourne -1 si le tas est vide
  Retourne la racine et le tableau modifié sinon
  Supprime l'élément du tas
  Reconstitue la propriété de tas max
- Opération `augmenterCle(A, i, cle)` (c.f. feuille)
- Opération `insérerTasMax(A, cle)` (c.f. feuille)

I. Introduction

Rappel: classification des structures de données



Exemples de domaines d'utilisation des graphes:

- Réseaux sociaux (liens "d'amitiés" entre deux membres)
- Réseau routier, internet, lignes aériennes, routes maritimes
- Interaction entre deux espèces (réseau trophique)
- Réseaux neuronaux
- Réseaux abstraits (modélisation de prédicats)

II. Définitions

Graphes non orientés

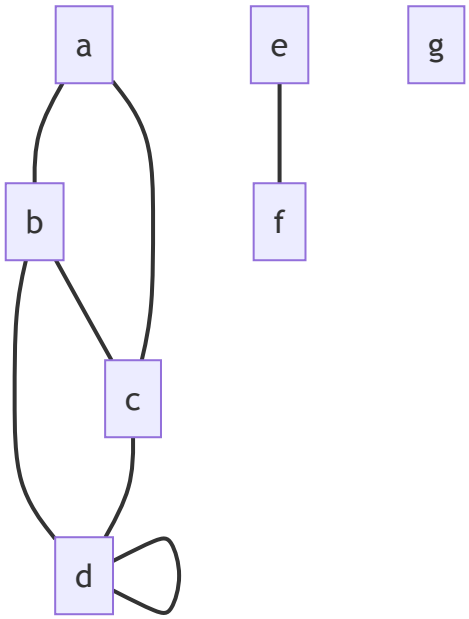
$G = (S, A)$ est un graphe non orienté si

- S est un ensemble fini
- A est une partie de S ayant 1 ou 2 éléments.

Les éléments de S sont appelés nœuds ou sommets de G (*vertices*). Les éléments de A sont appelés arêtes (*edges*).

Exemple :

$$G_2 = \Big(\{a, b, c, d, e, f, g\}, \big\{\{a, b\}, \{b, c\}, \{b, d\}, \{c, d\}, \{d\}, \{e, f\}\big\}\Big)$$



L'arête $\{s\}$ est une boucle.

Un graphe est dit simple s'il ne contient pas de boucles.

L'ordre du graphe est $|S|$ et la taille du graphe est $|S| + |A|$.

Un graphe discret est un graphe sans arêtes. Un graphe vide est un graphe ne contenant aucun sommet et aucune arête.

Le degré d'un sommet $d(s)$ est le nombre d'arêtes ayant s pour extrémité.

Exemple : $d(a) = 2$; $d(g) = 0$; $d(d) = 4$ (on compte les boucles deux fois).

Lemme: *c.f. poly*

Soit $G = (S, A)$ un graphe orienté ou non. Le chemin d'un sommet s à un sommet t de G est la suite finie $c = (s_0, s_1, \ldots, s_n)$ non vide de sommets tels que

- $s_0 = s$
- $s_n = t$
- $\forall i \in [\![0, n-1]\!], \exists a \in A, a = \{s_i, s_{i+1}\}$

- t est accessible depuis s s'il existe un chemin joignant s à t.
- L'entier n est la longueur du chemin.

- La distance de s à t, noté $d(s,t)$ est la longueur minimale d'un chemin joignant s à t.
- Si t est inaccessible depuis s, alors $d(s,t) = \infty$.

Un cycle, ou chemin fermé est un chemin joignant un sommet s à lui-même. Dans un graphe non orienté, la longueur d'un chemin fermée est au moins égale à 1. Toutes les arêtes du chemin sont distinctes. Un chemin de longueur 1, $\{s\} \in A$ est une boucle. Un graphe acyclique est un graphe n'ayant pas de cycles.

Un graphe non orienté est dit connexe lorsque tous les sommets sont à une distance finie les uns des autres. Un graphe non connexe peut être décomposé en plusieurs composantes connexes, les sous graphes connexes maximaux.

Théorème : Un graphe connexe d'ordre n possède $n-1$ arêtes.

Démonstration :

- Avec $n = 1$, on a un graphe contenant 1 nœud, il a donc aucune ou une arête, donc au moins 0.
- On a un graphe connexe ayant n nœuds et au moins $n-1$ arêtes. On ajoute un nœud. Pour avoir un graphe connexe d'ordre n, il faut relier ce nouveau nœud à au moins 1 sommet préexistant. On a donc au moins n arêtes.

Théorème : Si, dans un graphe G simple, tout sommet est de degré supérieur ou égal à 2, alors G possède au moins un cycle.

Démonstration : Partons d'un sommet arbitraire noté V_1. On peut construire une suite finie de sommets $v_1, v_2, \ldots, v_k$ telle que $v_i \notin \{v_1, v_2, \ldots, v_{i-1}\}$ et v_i est un voisin de v_{i-1}. Puisque les sommets de G sont un nombre fini, cette construction se termine. Or, v_k est au moins de degré 2. Donc, il possède, outre v_{k-1}, un voisin v_j dans la séquence. Alors $(v_j, v_{j+1}, \ldots, v_k, v_j)$ est un cycle de G.

Graphes orientés

Un graphe $G = (S, A)$ est orienté si

- S est un ensemble fini
- A est une partie de $S \times S$

Les éléments de S sont appelés nœuds ou sommets de G, et un élément $(s, s') \in A$ est un arc, représenté par une flèche de s à s'.

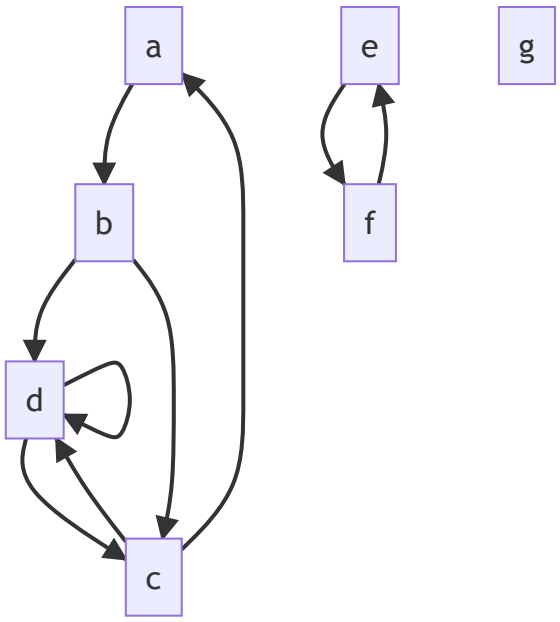
Un arc (s, s) est une boucle. Un graphe est dit simple s'il ne contient pas de boucles.

Avec $z = (s, s')$, on dit que s et s' sont les extrémités de z, ou qu'il sont reliés par z. On dit que s est l'extrémité initiale (de départ) de z, ou un prédécesseur s (voisin entrant). On dit que s' est l'extrémité finale (d'arrivée) de z. L'ordre du graphe est le nombre de sommets, $|S|$. La taille du graphe est $|S| + |A|$.

Un graphe discret est un graphe sans arc (ou arête). Un graphe vide $G = (\varnothing, \varnothing)$ est un graphe sans sommet ni arc.

Le degré sortant d'un sommet s, $d_+(s)$, est le nombre d'arcs partant de s. Le degré entrant d'un sommet s, $d_-(s)$, est le nombre d'arcs arrivant en s. Le degré d'un sommet s, $d(s)$ est la somme du degré sortant et du degré entrant.

$$\begin{cases} d_+(s) = \#\{s' \in E \mid (s, s') \in A\} \\ d_-(s) = \#\{s' \in E \mid (s', s) \in A\} \\ d(s) = d_+(s) + d_-(s). \end{cases}$$



Par exemple, le degré de a est 2, le degré de g est 0 et le degré de d est 5.

Un cycle est un chemin joignant un sommet s à lui-même.

Graphes pondérés

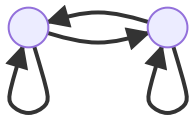
On dit que $G = (S, A, f)$ est un graphe pondéré ou étiqueté si (S, A) est un graphe orienté ou non, et f est une application de A dans un ensemble E.

Si $z \in A$, alors $f(z)$ est l'étiquette de z. On indique $f(z)$ à proximité de l'arête ou de l'arc sur le diagramme sagittal (cf ci-dessous).

Si $E = \mathbb{R}$, $f(z)$ est le coût ou poids de l'arête ou de l'arc.

Le poids ou le coût d'un chemin dans un graphe pondéré est la somme des poids des arêtes parcourus. On l'appelle parfois longueur du chemin (mais c'est en contradiction avec la définition de longueur dans un graphe non pondéré).

E peut être autre chose qu'un nombre. Dans ce cas, on parle de graphe étiqueté (par exemple, les automates).



Graphes bipartis

Soit $G = (S, A)$ un graphe orienté ou non. On dit que G est biparti si l'on peut partitionner S en deux parties S_1 et S_2 telles que deux sommets appartenant à la même partie ne sont pas adjacents (voisins).

Un problème complexe (par exemple un problème NP difficile) sur les graphes peut être simplifié si on a un graphe biparti (cf programme de spé).

Théorème : Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration :

- Soit G un graphe biparti non pondéré de classe S_1 et S_2.

  Tout cycle de ce graphe peut se parcourir en partant d'un sommet v de S_1 et doit revenir au sommet de départ en alternant des sommets de S_2 et de S_1.

  Pour revenir à un sommet de S_1, on est obligé de faire des allers retours, qui, par définition, sont de longueur 2.

  Chacun des cycles est de longueur paire.

- On suppose, pour simplifier, que G est connexe (sinon, appliquer le raisonnement à chacune des composantes connexes), non biparti et simple.

  Soit I l'ensemble des sommets à une distance impaire de v. Soit P l'ensemble des sommets à une distance paire de v. On a $S = P \cup I$ et $P \cap I = \varnothing$. Soit $v \in P$.

  Comme G n'est pas biparti, il existe deux sommets adjacents x_1 et x_2 appartenant à P (où à I).

  Comme G est connexe, il existe un chemin c_1 allant de x_1 à v et un chemin c_2 allant de v à x_2 de même parité que c_1.

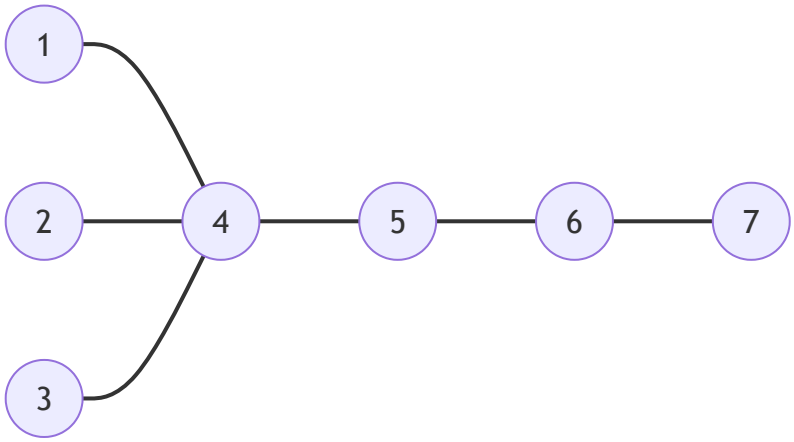
La somme des longueurs de c_1 et c_2 est donc paire. Pour former un cycle, il faut alors constituer le chemin $c_2 \cup \{x_1, x_2\} \cup c_1$ de longueur impaire.

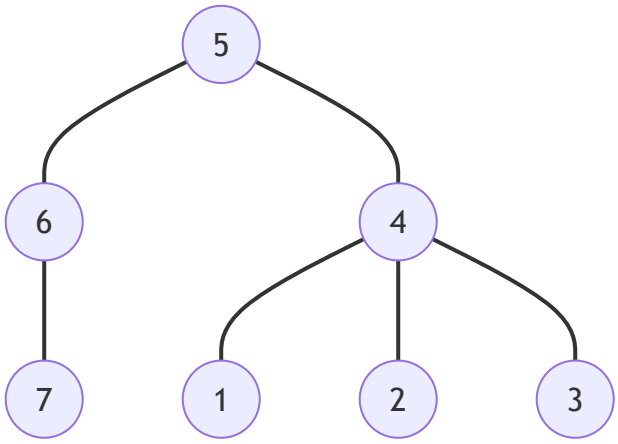
Rappel : Un arbre est un graphe connexe acyclique non orienté.

Théorème : Si a et b sont deux sommets distincts d'un arbre G, il existe un unique chemin reliant a et b.

Démonstration : G est connexe, ce qui assure l'existence d'un chemin reliant a et b. S'il existe un autre chemin, un cycle pourrait être construit en utilisant les deux chemins reliant a et b. Or, G est acyclique. On en déduit que le chemin reliant a et b est unique.

Conséquence : Soit un arbre représenté par un graphe G connexe acyclique non orienté. On a la possibilité de choisir arbitrairement un sommet r de G et d'orienter les arêtes de G pour créer les chemins reliant r à chacun des sommets. On obtient un arbre enraciné en r.





III. Implémentation

Listes d'adjacences

Une liste d'adjacence est la liste des sommets adjacents à un sommet donné. Dans le cas d'un graphe orienté, c'est la liste des successeurs (ou des prédécesseurs).

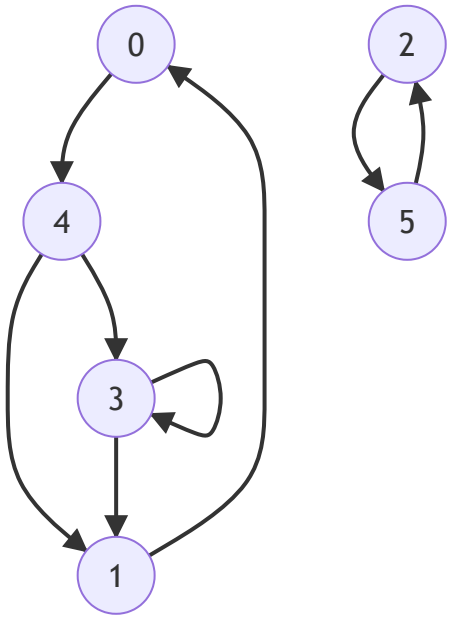
Attention : coïncidence entre le numéro de sommet et l'indice dans la liste.

Par exemple, la matrice d'adjacence du graphe suivant est

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

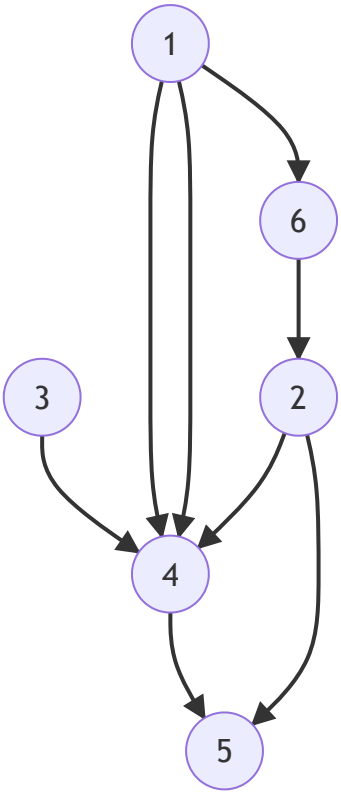
et la liste d'adjacence est

$$\big[[4],\ [0,\ 3],\ [5],\ [1,\ 3],\ [1,\ 3],\ [2]\big].$$



Enregistrements utilisés :

- `Sommet`: liste des sommets adjacents.
  1. identifiant nommé `id` de type `'a`
  2. liste des sommets adjacents, nommé `voisin` de type `'a list`
- `Graphe` : liste des sommets adjacents.



1. Écrire les instructions définissant les deux types et créant le graphe ci dessus :

```
(* Sommet : identifiant et liste de voisins *)
type 'a sommet = { id : 'a; voisins : 'a list};;

(* Graphe : liste de sommets *)
type 'a graphe = 'a sommet list;;

let g = [
    { id = 1; voisins = [2; 4; 6] };
    { id = 2; voisins = [4; 5] };
    { id = 3; voisins = [4] };
    { id = 4; voisins = [5] };
    { id = 5; voisins = [] };
    { id = 6; voisins = [2] }
];;
```

2. Écrire un fonction récursive `ajoute_arete g a b` de type: `ajoute_arete`

`: 'a sommet list -> 'a -> 'a -> 'a sommet list = <fun>`. Aide : `List.mem` indique si le premier argument est un élément de la liste passée (2nd argument).

Pré-condition : les sommets passés en arguments sont des sommets du graphe.

```
(* pré-condition : a et b sont des sommets du graphe *)
let rec ajoute_arete g a b = match g with
  | [] -> failwith "Graphe vide"
    (* impossible d'ajouter une arête *)
  | s :: q when s.id = a && List.mem b s.voisins -> g
    (* teste si a est en tête de liste : si oui (s.id = a) et que b est déjà
       présent parmi les voisins de a (List.mem b s.voisins).
       Dans ce cas, on retourne le graphe inchangé *)
  | s :: q when s.id = a -> { id = a; voisins = b :: (s.voisins) } :: q
  | s :: q -> s :: (ajoute_arete q a b);;
```

3. Écrire une fonction recursive `supprime_arete g a b` de signature `supprime_arete : 'a sommet list -> 'a -> 'a -> 'a sommet list = <fun>` supprimant l'arête entre le sommet a et le sommet b contenu dans le graphe g.

```
let rec supprime_arete g a b = match g with
  | [] -> []
  | s :: q when s.id = a ->
    let rec aux l = match l with
      | [] -> []
      | u :: q when u = b -> q
      | u :: q -> u :: (aux q)
    in { id = a; voisins = (aux s.voisins) } :: q
  | s :: q -> s :: (supprime_arete q a b);;
```

ou

```
let rec supprime_arete g a b =
  let rec aux = function
    (* parcours de la liste des voisins de a *)
    | [] -> failwith "sommet b absent des voisins de a"
      (* pas de voisins *)
    | t :: q when t = b -> q
      (* b : premier voisin dans la liste *)
    | t :: q -> t :: (aux q)
      (* recherche de b parmi les voisins suivants *)
  in match g with
    | [] -> failwith "sommet a absent"
    | s :: q when s.id = a -> { id = a; voisins = aux s.voisins } :: q
      (* sommet a trouvé : appel de aux pour rechercher b parmi les voisins *)
```

```
  | s :: q -> s :: (supprime_arete q a b);;
    (* recherche de a parmi les sommets suivants dans le graphe *)
```

4. Écrire une fonction récursive `ajoute_sommet` de signature `ajoute_sommet : 'a sommet list -> 'a -> 'a sommet list = <fun>` ajoutant un sommet.

```
let rec ajoute_sommet g a =
  match g with
    | [] -> [{ id = a; voisins = {} }]
      (* graphe vide -> ajout du sommet *)
    | x :: q when x.id = a -> g
      (* a dans le graphe -> pas de modification de g *)
    | x :: q -> x :: (ajoute_sommet q a);;
      (* poursuite de la recherche de a dans le reste du graphe *)
```

5. Écrire un fonction récursive `supprime_sommet g a` permettant de supprimer un sommet a du graphe g avec la signature `supprime_sommet : 'a sommet list -> 'a -> 'a sommet list = <fun>`.

```
let rec supprime_sommet g a =
  (* exploration de la liste des voisins pour éliminer a *)
  let rec aux = function
    | [] -> []
    | t :: q when t = a -> q
    | t :: q -> t :: (aux q)
  in match g with
    | [] -> []
    | s :: q when s.id = a -> supprime_sommet q a
    | s :: q ->
      let ns = { id = s.id; voisins = aux s.voisins }
        in ns :: (supprime_sommet q a);;
```

Bilan :

En C : tableau de pointeurs sur liste chaînée d'enregistrements (voir chaînage externe)

Ordre des sommets arbitraire dans la liste des voisins

Remplacement du numéro du sommet par un doublet (sommet, poids) pour un graphe pondéré.

Avantages :

- représentation assez souple : facilité pour ajouter ou supprimer des sommets ou des arêtes,
- occupation mémoire en $n + p$ (n sommets et p arcs)

Inconvénient : pas d'accès rapide à un sommet ou à une arête particulière

Matrices d'adjacence

```
let liste_to_matrice liste =
  let n = List.length liste in
  let matrice = Array.make_matrix n n 0 in

  for a = 0 to (n - 1) do
    let voisins = (List.nth liste a).voisins in

    for b = 0 to (List.length voisins - 1) do
      matrice.(a).(List.nth voisins b) <- 1;
    done;
  done;

  matrice;;

let matrice_to_liste matrice =
  let liste = ref [] in
  let n = Array.length matrice in
  for a = 0 to (n - 1) do
    let voisins = ref [] in
    for b = (n-1) downto 0 do
      if matice.(a).(b) <> 0 then
        voisins := b :: !voisins;
    done;

    let enr = { id = a; voisins = !voisins } in
    liste := enr :: !liste;
  done;
  liste;;
```

Bilan

Remplacement du 1 par le poids de l'arc pour un graphe pondéré

Avantages :

- Facilité en coût constant aux informations (coût d'un accès tableau)
- Facilité pour ajouter ou supprimer des sommets ou des arêtes

Inconvénients :

- Ajout / Suppression de sommets
- Coût en mémoire : n^2 variables où n est le nombre de sommets

III. Parcours de graphe

Définitions

Parcours de graphe : Énumération de l'ensemble des sommets accessibles par un chemin pour le appliquer un traitement

Objectif : Étude d'une propriété du graphe (biparti, connexité, ...)

Différentes possibilités mais nécessité de maintenir à jour deux listes :

- Liste des sommets déjà traités → déjà vus
- Liste des sommets en cours de traitement → à traiter

Différence entre parcours :

- Manière dont sont ajoutés et extraits les sommets
- Dépend de la structure de données utilisée

Principe :

```
à_traiter <-- [ s0 ]
déjà_vus <-- [ s0 ]

tant que à_traiter != [] faire
  extraire s de à_traiter
  appliquer le traitement à s
  ajouter s à déjà_vus

  pour tout les voisins t de s faire
    si t n'est pas déjà dans déjà_vus alors
      ajouter t à à_traiter
    fin si
  fin pour
fin tant que
```

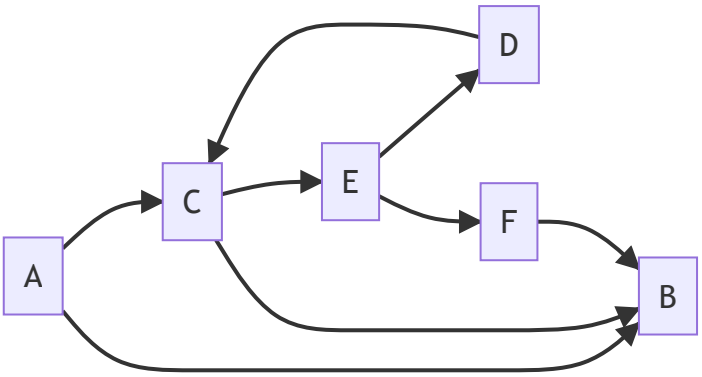
Parcours en largeur

Breadth First Search (BFS)

Structure utilisée : file d'attente (FIFO) pour stocker les sommets

Utilisé notamment pour les recherches de plus court chemin

Exemple : pour le graphe



→ à traiter	déjà vus
A	∅
BC	A
C	AB
E	ABC
DF	ABCE
F	ABCED
∅	ABCEDF

Pour gérer une file en OCamL, on utilise le module `Queue` et éventuellement la fonction `List.iter`. (les fonctions ne sont pas à connaître; si on en a besoin, elle seront rappelées)

```
let bfs_queue traitement g initial =
  (* Création d'un array de false : mémoire des sommets traités *)
  let deja_vus = Array.make (Array.length g) false in
  (* Creation d'une file pour ordre de traitement *)
  let a_traiter = Queue.create() in

  (* Fonction traitement d'un sommet et mise à jour des structures *)
  let ajoute x =
    if not deja_vus.(x) then
      begin
        (* Consultation de deja_traite : si le sommet n'a pas été traité *)
        traitement x;
        deja_vus.(x) <- true (* Mise à jours de la liste des sommets traités : true *)
        Queue.push x a_traiter (* Ajout du sommet de la file *)
      end
  in
    ajoute initial; (* Appel de ajout avec le sommet initial *)
```

```
  (* Tant que la file n'est pas vide *)
  while not (Queue.is_empty a_traiter) do
    let x = Queue.pop a_traiter in (* Extraction du sommet de la file *)

    (* Pour appeler ajoute à chacun des éléments de g.(x), donc les successeurs de x *)
    List.iter ajoute g.(x);
  done;;

let bfs_artisanal traitement g initial =
  let deja_vus = Array.make (Array.length g) false in
  deja_vus.(initial) <- true;

  let rec traite_sommet liste = match liste with
    | []     -> ()
    | x :: q -> traitement x;
                traite_sommet (q @ voisins g.(x))
  and voisins liste = match liste with
    | []                          -> []
    | y :: q when deja_vus.(y)  -> voisins q
    | y :: q                      -> deja_vus.(y) <- true;
                                     y :: voisins q
  in traite_sommet [ initial ];;
```

Complexité majorée par la taille de graphe donc $\mathcal{O}\big(|S| + |A|\big)$.

Départ du sommet O.

Quand la file `a_traiter` est vide, on identifie les composantes connexes.

IV. Algorithmes des graphes

Algorithmes du plus court chemin

A. Dijkstra

1. Définition et théorème Plus court chemin dans un graphe pondéré : longueur / coût d'un chemin : somme des poids des arêtes

Objectif : déterminer le chemin de longueur minimale entre deux sommets s et s'

Théorème : Soit $G = (S, A)$ un graphe orienté pondérés à poids positifs ou nuls. Soient s et s' deux sommets de G. Si s et s' sont reliés par un chemin, alors il existe un plus court chemin de s à s'.

Démonstration : Soit c un chemin joignant s à s'. Si c passe à deux reprises par le même sommet, un cycle est présent sur le chemin c. Ce cycle peut être supprimé du chemin c. Le processus peut être itéré jusqu'à la disparition des cycles du chemin. Le plus court chemin de s à s' est donc un chemin pour lequel

tous les sommets $(s, s_1, \ldots, s')$ sont distincts. Le nombre de sommets étant fini, le nombre de chemins passant par des sommets distincts l'est aussi. Il existe donc un chemin de longueur minimale parmi ces chemins.

Algorithme de Dijkstra : plus court chemin depuis un sommet vers tous les autres sommets du graphe.

Soit $G = (S, A)$ un graphe pondéré et soit M un ensemble de sommets. Un M-chemin est un chemin dont tous les sommets jusqu'à l'avant denier sont dans M et le dernier sommet n'est pas dans M.

Théorème : Soit c un M-chemin, de longueur minimale et x sont dernier sommet (on fixe c PUIS on fixe x). Alors c est le plus court chemin de s à x.

Démonstration (par l'absurde):

> Supposons qu'il existe c' un chemin de s à x, de longueur strictement inférieure à la longueur de c :
>
> $$\ell(c') < \ell(c)$$
>
> où $\ell(c)$ est la longueur du chemin c.
>
> Soit x', le premier sommet de c' n'étant pas dans M.
>
> Le sous chemin c'' de s à x' est un M-chemin. Donc $\ell(c'') \leqslant \ell(c')$ car les poids sont positifs ou nuls.
>
> Et, $\ell(c'') < \ell(c)$, ce qui contredit l'hypothèse selon laquelle c est un chemin de longueur minimale allant de s à x.

2. Principe de l'algorithme de Dijkstra On définit:

- distance : ensemble des distances depuis s; distance$[i]$ est la longueur du plus court chemin de s à i. À l'initialisation, toutes les distances sont infinies. L'algorithme affine progressivement l'estimation du plus court chemin.
- prédécesseur : ensemble des prédécesseurs pour un chemin depuis s où prédécesseur$[i]$ est le prédécesseur du sommet i sur le le plus court chemin provisoire depuis s.
- M est l'ensemble des sommets déjà traités; on utilise le théorème des M-chemins pour bâtir progressivement les plus courts chemins depuis s.
- E est l'ensemble des sommets en attente d'être traités. On a constamment $S = M \cup E$: invariant de l'algorithme de Dijkstra.

Propriétés : cf poly

3. Exemple

1. Indiquer l'état de `Distances`, `Prédécesseurs`, M et E après chaque itération.

Pour l'initialisation,

$$\begin{cases} M = \varnothing, \\ E = \{0, 1, 2, 3, 4\}, \\ \text{Distance} : [0, +\infty, +\infty, +\infty, +\infty], \\ \text{Prédécesseur} : [\text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}]. \end{cases}$$

Après une itération, on a

$$\begin{cases} M = \{0\}, \\ E = \{1, 2, 3, 4\}, \\ \text{Distance} : [0, 10, +\infty, 5, +\infty], \\ \text{Prédécesseur} : [\text{nil}, 0, \text{nil}, 0, \text{nil}]. \end{cases}$$

Après deux itérations, on a

$$\begin{cases} M = \{0, 3\}, \\ E = \{1, 2, 4\}, \\ \text{Distance} : [0, 8, 14, 5, 7], \\ \text{Prédécesseur} : [\text{nil}, 3, 3, 0, 3]. \end{cases}$$

Après trois itérations, on a

$$\begin{cases} M = \{0, 3, 4\}, \\ E = \{1, 2\}, \\ \text{Distance} : [0, 8, 13, 5, 7], \\ \text{Prédécesseur} : [\text{nil}, 3, 4, 0, 3]. \end{cases}$$

Après quatre itérations, on a

$$\begin{cases} M = \{0, 3, 4, 1\}, \\ E = \{2\}, \\ \text{Distance} : [0, 8, 9, 5, 7], \\ \text{Prédécesseur} : [\text{nil}, 3, 1, 0, 3]. \end{cases}$$

Après cinq itérations, on a

$$\begin{cases} M = \{0, 3, 4, 1, 2\}, \\ E = \varnothing, \\ \text{Distance} : [0, 8, 9, 5, 7], \\ \text{Prédécesseur} : [\text{nil}, 3, 1, 0, 3]. \end{cases}$$

Autre présentation :

0	1	2	3	4	sommet séléctionné
0	∞	∞	∞	∞	0
$\varnothing$	10_0	∞	5_0	∞	3
$\varnothing$	8_3	14_3	$\varnothing$	7_3	4
$\varnothing$	8_3	13_4	$\varnothing$	$\varnothing$	1
$\varnothing$	$\varnothing$	9_1	$\varnothing$	$\varnothing$	2

2. Pour `distance`, `prédécesseur`, M et E, on utilise des tableaux. Pour le graphe G, on utilise une matrice d'adjacence.
3. Pour le graphe, on utilise une liste d'adjacence car ça économise de la mémoire et l'accès aux voisins est plus simple. Pour `distance` et `prédécesseur`, on utilise deux tableaux d'entiers de dimension $\#S$. Pour M et E, on utilise un tableau de bouléens de taille $\#S$ où $x \to$ `true` représente $x \in E$ et $x \notin M$ et $x \to$ `false` représente $x \in M$ et $x \notin E$.
4. On utilise la stratégie gloutonne.

```
Alorgithme Dijkstra(G, s):
    distances = créer_tableau(nb_sommets, +infini)
    prédécesseurs = créer_tableau(nv_sommets, nil)
    distances[s] = 0
    dist = créer_file(nb_sommets)
    E = créer_tableau(nb_sommets, true)

    Tant que dist n'est pas vide faire
        u <- extraire_min(dist)
        rétablire_propriété_file(dist) # O(|S| ln(|S|))
        E[u] = false

        Pour chaque voisin v de u présent tel que E[v] = true faire # O(|A|)
            Si distances[v] >= distances[u] + pondération(u,v) alors
                distances[v] = distances[u] + pondération(u,v) alors
                diminuer_clé(dist, v) # O(ln(|S|))
                prédécesseurs[v] = u
            Fin Si
        Fin Pour
    Fin Tant Que
```

La complexité de cet algorithme est donc

$$\mathcal{O}\Big(\big(|A| + |S|\big) \times \ln\big(|S|\big)\Big).$$

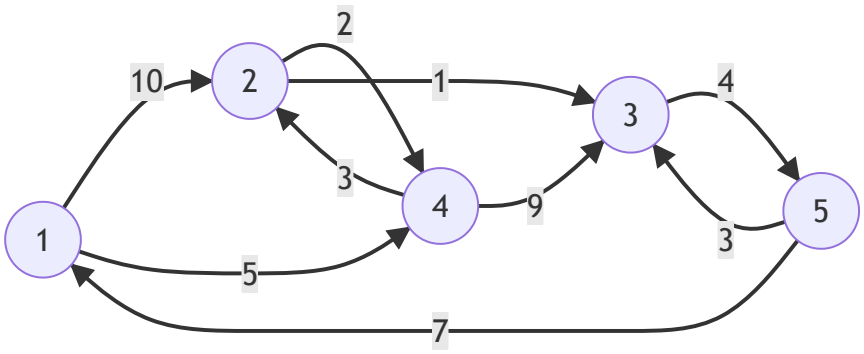
La terminaison et l'invariant sont sur le polycopié.

B. Floyd-Warshall

Plus court chemin dans un graphe pondéré avec des pondérations positives ou négatives mais pas de cycle de poids strictement négatifs.

Plus courts chemins entre toutes les paires de sommets.

Utilisation de la programmation dynamique : sous structure optimale, chevauchement de sous-problèmes et mémorisation des résultats.



Dans le graphe, quel est le plus court chemin de 5 à 2 ?

$$5 \to 1 \to 4 \to 2 \qquad \text{coût : } 15$$

Mise en place de deux matrices:

- une matrice contenant les chemins (prédécesseurs) $\to P$
- une matrice contenant les distances $\to D$

$$D = \begin{array}{c} \\ \text{sommet de départ} \end{array} \overset{\text{sommet d'arrivée}}{\begin{bmatrix} 0 & 8 & 9 & 5 & 7 \\ 11 & 0 & 1 & 2 & 4 \\ 11 & 19 & 0 & 16 & 4 \\ 9 & 3 & 4 & 0 & 2 \\ 7 & 15 & 6 & 12 & 0 \end{bmatrix}} \qquad P = \begin{bmatrix} 1 & 4 & 2 & 1 & 4 \\ 5 & 2 & 2 & 2 & 4 \\ 5 & 4 & 3 & 1 & 3 \\ 5 & 4 & 2 & 4 & 4 \\ 5 & 4 & 5 & 1 & 5 \end{bmatrix}$$

Structure d'un plus court chemin Sur un chemin $c = (v_1, v_2, \ldots, v_m)$, on définit les sommets intermédiaires de c comme étant l'ensemble des sommets $\{v2, \ldots, v_{m-1}\}$.

On considère:

- $S = \{1, 2, \ldots, n\}$, les sommets de $G = (S, A, f)$, graphe pondéré sans cycle de longueur strictement négative.
- Sous-ensemble de G : $\{1, 2, \ldots, k\}$ les sommets pour un k donné

- Un couple (i, j) de sommets quelconques de G
- Tous les chemins allant de i à j dont les sommets intermédiaires sont tous dans $\{1, 2, \dots k\}$.
- Parmi ces chemins, p est le chemin de longueur minimale

k est-il visité par p ?

> **Non** $\rightarrow$ un plus court chemin de i à j ayant tous ses sommets intermédiaires dans $\{1, 2, \dots, k-1\}$ est aussi un plus court chemin ayant tous ses sommets intermédiaires dans $\{1, 2, \dots, k\}$.
>
> **Oui** $\rightarrow$ le chemin p se divise en $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ avec p_1 le plus court chemin de i à k ayant tous ses sommets intermédiaires dans $\{1, 2, \dots, k-1\}$ et p_2, plus court chemin de k à j ayant tous ses sommets intermédiaires dans $\{1, 2, \dots, k-1\}$.

Exemple : pour aller de $i = 5$ à $j = 2$, le chemin le plus court $(5, 1, 4, 2)$. Les sommets intermédiaires sont 1 et 4; 4 dernier sommet intermédiaire de p. Les sommets intermédiaires sont dans le sous-ensemble de sommets $\{1, 3, 4\}$.

> Pour $k = 4$, 4 est sur le chemin p; il se décompose donc en un chemin p_1 de 5 à 4 dont les sommets intermédiaires sont dans $\{1, 3\}$, et en un chemin p_2 de 4 à 2 dont tous les chemins intermédiaires sont dans $\{1, 3\}$.

p_1 et p_2 sont les chemins de longueur minimale entre 5 et 4, d'une part, et entre 4 et 2 d'autre part.

Mise en place d'une relation de récurrence en considérant ensuite p_1 et p_2 avec $k = 3$.

On a donc $D = (d_{i,j})$ avec

$$d_{i,j}^{(k)} = \begin{cases} f_{i,j} & \text{si } k = 0 \\ \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right) & \text{si } k > 0. \end{cases}$$

Le cas où $k = 0$ correspond à l'initialisation (chemin sans sommet intermédiaire). La matrice D correspond alors à la matrice d'adjacence

$$d_{i,j} = \begin{cases} 0 & \text{si } i = j \\ f_{i,j} & \text{si } i \neq j \text{ et } (i,j) \in A \\ +\infty & \text{si } i \neq j \text{ et } (i,j) \notin A. \end{cases}$$

Idée de l'algorithme :

- On fixe un sommet k.
- Pour chaque couple de sommets (i, j), on regarde si inclure k dans le chemin permet d'obtenir un chemin plus court.
- On itère pour chaque sommet k de S.

Mise à jour des prédécesseurs : matrice P

En pratique, deux manière de procéder

1. Détermination de D puis recherche de P à partir de D
2. Détermination simultanée de D et de $P \rightarrow$ formulation d'une relation de récurrence entre $P^{(k)}$ et $P^{(k-1)}$.

Initialisation de $P \rightarrow P^{(0)}$:

$$p_{i,j} = \begin{cases} \text{nil ou } -1 & \text{si } i = j \text{ ou } f_{i,j} = \infty \\ i & \text{si } i \neq j \text{ et } f_{i,j} < \infty \end{cases}$$

Relation de récurrence pour la matrice P :

$$p_{i,j}^{(k)} = \begin{cases} p_{i,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} \leqslant d_{i,k}^{(k-1)} d_{k,j}^{(k-1)} \\ p_{k,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} d_{k,j}^{(k-1)} \end{cases}$$

Pour $k = 4$, on a

$$D = \begin{pmatrix} 0 & 8 & 9 & 5 & 7 \\ \infty & 0 & 1 & 2 & 4 \\ \infty & \infty & 0 & \infty & 4 \\ \infty & 3 & 4 & 0 & 2 \\ 7 & 15 & 6 & 12 & 0 \end{pmatrix} \qquad P = \begin{pmatrix} & 4 & 2 & 1 & 4 \\ ? & & 2 & 2 & 4 \\ ? & ? & & ? & 3 \\ ? & 4 & 2 & & 4 \\ 5 & 4 & 5 & 1 & \end{pmatrix}.$$

Avec $k = 5$, les matrices deviennent

$$D = \begin{pmatrix} 0 & 8 & 9 & 5 & 7 \\ 11 & 0 & 1 & 2 & 4 \\ 11 & 19 & 0 & \infty & 4 \\ \infty & 3 & 4 & 0 & 2 \\ 7 & 15 & 6 & 12 & 0 \end{pmatrix} \qquad P = \begin{pmatrix} & 4 & 2 & 1 & 4 \\ 5 & & 2 & 2 & 4 \\ 5 & 4 & & ? & 3 \\ ? & 4 & 2 & & 4 \\ 5 & 4 & 5 & 1 & \end{pmatrix}$$

En effet, $d_{3,2} = \infty$ et $d_{3,5} + d_{5,2} = 4 + 15 = 19 \rightarrow$ remplacement dans D.

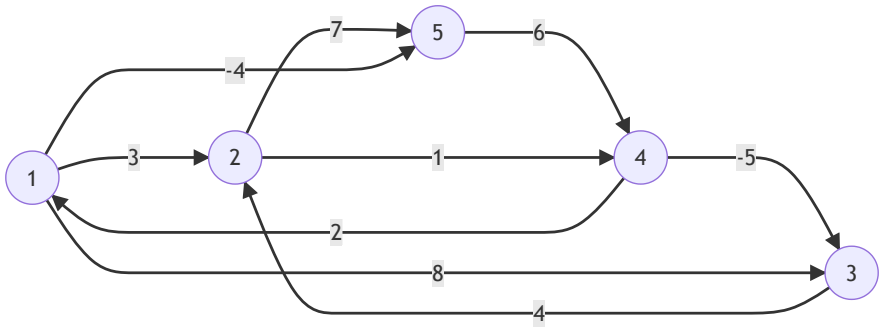
Pour le chemin : décomposition de $(3, \ldots, 2)$ en $(3, \ldots, 5)$ et $(5, \ldots, 2)$; et on exclut 5 du sous-ensemble des sommets intermédiaires, on ne considère plus que le chemin 5 à 2.

> $\rightarrow$ prédécesseur de 2 dans le chemin de 3 à 2 devient prédécesseur de 2 dans le chemin de 5 à 2 donc 4.

```
Algorithme FloydWarsallAscendant(G)
    Début
        n = |S|
        D = G
        P = créer_tableau(n * n, prédécesseur dans G)
```

```
        Pour k = 1 à n faire
            Pour i = 1 à n faire
                Pour j = 1 à n faire
                    Si i != j et D(i,j) > D(i,k) + D(k,j) alors
                        D(i,j) = D(i,k) + D(k,j)
                        P(i,j) = P(k,j)
                    Fin Si
                Fin Pour
            Fin Pour
        Fin Pour
    Fin
```

Évolution des matrices D et P pour le graphe suivant ?



Pour $k = 0$, on a

$$D = \begin{pmatrix} / & 3 & 8 & \infty & -4 \\ \infty & / & \infty & 1 & 7 \\ \infty & 4 & / & \infty & \infty \\ 2 & \infty & -5 & / & \infty \\ \infty & \infty & \infty & 6 & / \end{pmatrix} \quad \text{et} \quad P = \begin{pmatrix} / & 1 & 1 & ? & 1 \\ ? & / & ? & 2 & 2 \\ ? & 3 & / & ? & ? \\ 4 & ? & 4 & / & ? \\ ? & ? & ? & 5 & / \end{pmatrix}$$

Pour $k = 1$, on a

$$D = \begin{pmatrix} / & 3 & 8 & \infty & -4 \\ \infty & / & \infty & 1 & 7 \\ \infty & 4 & / & \infty & \infty \\ 2 & 5 & -5 & / & -2 \\ \infty & \infty & \infty & 6 & / \end{pmatrix} \quad \text{et} \quad P = \begin{pmatrix} / & 1 & 1 & ? & 1 \\ ? & / & ? & 2 & 2 \\ ? & 3 & / & ? & ? \\ 4 & 1 & 4 & / & 1 \\ ? & ? & ? & 5 & / \end{pmatrix}$$

Pour $k = 2$, on a

$$D = \begin{pmatrix} / & 3 & 8 & 4 & -4 \\ \infty & / & \infty & 1 & 7 \\ \infty & 4 & / & 5 & 11 \\ 2 & 5 & -5 & / & -2 \\ \infty & \infty & \infty & 6 & / \end{pmatrix} \quad \text{et} \quad P = \begin{pmatrix} / & 1 & 1 & 2 & 1 \\ ? & / & ? & 2 & 2 \\ ? & 3 & / & 2 & 2 \\ 4 & 1 & 4 & / & 1 \\ ? & ? & ? & 5 & / \end{pmatrix}$$

Pour $k = 3$, on a

$$D = \begin{pmatrix} / & 3 & 8 & 4 & -4 \\ \infty & / & \infty & 1 & 7 \\ \infty & 4 & / & 5 & 11 \\ 2 & -1 & -5 & / & -2 \\ \infty & \infty & \infty & 6 & / \end{pmatrix} \qquad \text{et} \qquad P = \begin{pmatrix} / & 1 & 1 & 2 & 1 \\ ? & / & ? & 2 & 2 \\ ? & 3 & / & 2 & 2 \\ 4 & 3 & 4 & / & 1 \\ ? & ? & ? & 5 & / \end{pmatrix}$$

Pour $k = 4$, on a

$$D = \begin{pmatrix} / & 3 & -1 & 4 & -4 \\ 3 & / & -4 & 1 & -1 \\ 7 & 4 & / & 5 & 3 \\ 2 & -1 & -5 & / & -2 \\ 8 & 5 & 1 & 6 & / \end{pmatrix} \qquad \text{et} \qquad P = \begin{pmatrix} / & 1 & 4 & 2 & 1 \\ 4 & / & 4 & 2 & 1 \\ 4 & 3 & / & 2 & 1 \\ 4 & 3 & 4 & / & 1 \\ 4 & 3 & 4 & 5 & / \end{pmatrix}$$

Pour $k = 5$, on a

$$D = \begin{pmatrix} / & 1 & -3 & 2 & -4 \\ 3 & / & -4 & 1 & -1 \\ 7 & 4 & / & 5 & 3 \\ 2 & -1 & -5 & / & -2 \\ 8 & 5 & 1 & 6 & / \end{pmatrix} \qquad \text{et} \qquad P = \begin{pmatrix} / & 3 & 4 & 5 & 1 \\ 4 & / & 4 & 2 & 1 \\ 4 & 3 & / & 2 & 1 \\ 4 & 3 & 4 & / & 1 \\ 4 & 3 & 4 & 5 & / \end{pmatrix}$$

On implémente G à l'aide d'une matrice d'adjacence. La complexité de cet algorithme est en $\mathcal{O}\big(|S|^3\big)$; c'est intéressant pour les graphes de petite taille ou de taille modérée.