Cours algorithmique

VAN DE MERGHEL Robin

2023

Contents

Cours 1.

Les '==' dans le PDF sont à ignorer pour l'instant, ils permettent du surlignage en markdown.

Introduction

Objectif principal : Comment définir des structures de données et des algorithmes pour les manipuler efficacement ?

→ Une structure de données est un objet qui permet de stocker des informations et de faire des requêtes sur ces informations.

Exemple

Un éditeur de texte : manipuler une chaine de caractères.

Note : on ne parle pas du rendu visuel

Définition abstraite :

- Insérer une chaîne de caractères à un endroit
- Supprimer une chaîne de caractères à un endroit
- Rechercher une chaîne de caractères dans un texte
- Rechercher une chaîne de caractères / un sous mot dans un texte
- Mise en page visuelle
  - Chaque caractère a un certain nombre d'attributs (couleur, police, taille, etc.)

Lors de l'étape de la programmation il faut que cette définition abstraite devienne concrète dans la machine en utilisant les constructions du langage de programmation. Tout cela va constituer la structure de données de l'éditeur de texte : c'est **l'implémentation**.

Détails Une manière de faire :

- structure en `C` pour représenter un caractère et ses attributs
- l'objet pour stocker les caractères est un tableau de caractères

Structure de données concrète pour éditeur de texte ici sera : ce tableau ainsi que les fonctions qui permettent de manipuler ce tableau (`lire`, `ecrire`, `inserer`, `supprimer`, `rechercher`, etc.).

En java, le constructeur classe encapsule les différentes fonctions manipulant le tableau de caractères.

Remarque

- Une modification de caractère revient à supprimer un caractère et à insérer un autre caractère. Donc la question de la complétude de la structure de données des opérations proposées se pose pour chaque chaque structure de données.

Complétude : Est-ce que tout les comportements voulus sont définis par les opérations proposées ?

- Pour être sûr que la fonction implémentée est correcte, on décrit un "bon comportement" en langage mathématique. On peut ensuite vérifier que la fonction implémentée correspond bien à ce comportement à l'aide d'une preuve (exemple : récurrence). Ensuite, pour chaque implémentation, on montre que l'implémentation respecte la spécification.

Retour sur les détails Est-ce que c'est une bonne implémentation ?

- Bonne par rapport à quoi ? On définit des objetifs (par exemple une fonction associant à chaque implémentation une valeuret notre objectif sera une certaine valeur).
  - Exemple complexité en temps (grosso modo temps d'execution)
  - Complexité en mémoire (espace en mémoire auxiliaire utilisée)

Temps d'execution	Mémoire utilisée
`ajouter` / `supprimer` (on va devoir décaler les caractères suivants) La `recherche` est lente : on ne peut pas utiliser d'algorithme complexe (le texte n'est pas forcément trié)	Mémoire ? Comment on gère ? Si on utilise la structure de données avec tableau dynamique, c'est un peux mieux

Langage de description

- Langage usuel pour décrire des structure de données avec de temps en temps des notations mathématiques ou constructions mathématiques (que vous avez déjà vues)
- Pour expliquer une implémentation d'une structure de données :
  - La description d'étapes élémentaires (basée sur le cours d'Alhorithmique)
  - De temps en temps du pseudo-code est équivalent au mélange de `C` et de pseudo-code vu en L1.

Chapitre 1 : Algorithmique, Types, Valeurs

- **Algorithme** : Suite d'opérations en vue de résoudre un prolbème.
  - Un algorithme a des entrées, et produit une sortie (réponse à la question)
  - Un algorithme manipule des données
  - Chaque opération est définie par un nom et une description
    - Ces **données** on les classifie par rapport aux opérations possibles : c'est ce que l'on appelle le typage
    - L'encodage des données produit un ensemble de **valeurs**

Cours 2.

On va classifier les types : chaque type est identifié par un ensemble de valeurs, et l'ensemble des opérations possibles sur les valeurs. Une classe sera un type.

- Une constante est une valeur particulière pour ce type.
- Une variable va représenter des valeurs possibles pour ce type.
  - Faire la distinction entre la variable et le contenu (la variable est l'ensemble des valeurs possibles pour ce type, le contenu est une valeur de ce type)

Les types

Il y a deux types de types :

- Les ==types primitifs==
- Les ==types composés==

Types primitifs

Ils sont non décomposables et fournis par défaut. Dans ce cas les type primitifs sont entiers, réels, carctères, booléens, chaînes de caractères.

Types composés

Ils sont construits à partir d'autres types : 3 constructeurs de types :

- *Produit* : $T_1, T_2, ..., T_m$ sont des types, alors $T_1 \times T_2 \times ... \times T_m$ est un type avec comme valeurs les couples $(v_1, v_2, ..., v_m)$ où v_i est une valeur de T_i.
- *Somme* : $T_1, T_2, ..., T_m$ sont des types, alors $T_1 \oplus T_2 \oplus ... \oplus T_m$ est un type avec comme valeurs les couples $(v_1, v_2, ..., v_m)$ où v_i est une valeur de T_i.
  - **Exemple** : les constructeurs `union` en C
- *Enregistrements* : chaque valeur d'un enregistrement est composé de plusieurs entités, appelées **champs**, chacun ayant un identifiant, un type et une valeur.
  - **Exemple** : les constructeurs `struct` en `C`, ou encore `class` en `Java`
- *Constructeur de tableau* : Un tableau est un ensemble contigu de valeurs où on a accès en temps constant à chaque valeur. Dans la plupart des implémentations de tableau toute les valeurs ont le même type.
  - Pour réserver de la place mémoire pour un tableau de taille n, et la taille d'une valeur c'est p, on réserve $n \times p$ octets bits. Si adresse de début c'est adr_0, alors pour accéder à l'indice $0 \leq i \leq n-1$:

$$i \times p + adr_0 \text{ (opération constante)}$$

Chapitre 2 : Types de données abstrait

Définition 2.1 : Types de données abstrait

Un ==type de données abstrait (TDA)== est un type de données composé, dont on définit :

- Une **signature** : un identifiant, et type de retour / paramètres de chaque opération et d'un ensemble de types prédéfinis à utiliser.
- Une **liste d'axiomes** qui vont définir le comportement des opérations sur des valeurs du TDA.

Exemple

On choisit les entiers. - Opérations, addition, soustraction prend deux entiers en paramètres et renvoie un entier.

Pour avoir des ==types de données concrètes (TDC)==, on propose une façon de représenter dans la machine une TDA, en proposant la façon de manipuler cette représentation à travers les opérations du TDA. C'est ce qu'on appelle une ==implémentation d'un TDA==.

Exemple

- Un TDC : on choisit de représenter les entiers par des nombres binaires en représentation complément à deux. On va donc implémenter les opérations de base sur les entiers.
- La définition d'un `String` en `C` : on choisit de représenter les chaînes de caractères par des tableaux de caractères. On va donc implémenter les opérations de base sur les chaînes de caractères. C'est un TDA.

2.1 : TDA Pile

Le ==TDA Pile== est apparu lorsque l'on a eu besoin de structures, les programmes en blocs que l'on peut référencer (bloc pouvant se référencer).

Principale question : comment fournir des données aux blocs référencés et ensuite retourner au bloc appelant ? C'est représenter les références aux blocs dans un SDD (Structure de Données Dynamique) ayant le *même comportement qu'une pile d'assiette*.

Définition 2.2 : Pile

Les ==différentes opérations== sur une pile sont :

- `creerPile`: () → `Pile_T` : Créer une pile vide
- `estVidePile`: `Pile_T` → `Booléen` : Vérifie si la pile est vide
- `empiler`: `Pile_T` × `T` → `Pile_T` : Empile un élément sur la pile
- `depiler`: `Pile_T` → `Pile_T` : Dépile un élément de la pile
- `sommetPile`: `Pile_T` → `T` : Renvoie le sommet de la pile

==Axiomes== :

- `estVidePile(creerPile())= True`
- `estVidePile(empiler(p, x))= False`
- `sommetPile(empiler(p, x))= x`
- `depiler(empiler(p, x))= p`

Le TDA `T` est utilisé pour dire que la pile on l'utilise pour stocker que des objets du même type. On ne fait aucune supposition sur `T`, à part les valeurs de T existent.

Implémentation possible On utilise un tableau pour stocker les valeurs et un entier qui va pointer sur l'indice du sommet de pile.

En `C` :

```c
struct pile {
    int T[MAX];
    int sommetPile;
};
```

Ici, `sommetPile` permet de savoir où l'on doit poser le sommet de la pile. On peut donc empiler et dépiler en temps constant (au lieu de chercher l'élément le plus en bas de la pile).

Le TDA File

Le besoin : gestion des accès à des ressources limitées ou pour éviter les accès concurrents.

==Paradigme== : Premier arrivé, premier servi.

Description du TDA Le TDA `File_T` est un TDA composé de :

- `creerFile`: () → `File_T` : Créer une file vide
- `estVideFile`: `File_T` → `Booléen` : Vérifie si la file est vide
- `enfiler`: `File_T` × `T` → `File_T` : Enfile un élément dans la file
- `defiler`: `File_T` → `File_T` : Défile un élément de la file
- `teteFile`: `File_T` → `T` : Renvoie le premier élément de la file

==Axiomes== :

- `estVideFile(creerFile())= True`
- `estVideFile(enfiler(f, x))= False`
- `teteFile(enfiler(f, x))= x`
- `defiler(enfiler(f, x))= f`
- `defiler(enfiler(enfiler(f, x), y))= enfiler(f, y)`

Implémentation possible On peut représenter une file par un tableau et deux entiers :

- Tête qui point sur l'indice de la tête de file
- Queue qui pointe sur l'indice de la queue de la file

En `C` :

```c
struct file {
    int T[MAX];
    int tete;
    int queue;
};
```

Pour un fil, comme c'est un ensemble ordonné : on doit toujours connaître l'ordre de la file, ce qui va permettre de défiler et d'enfiler en temps constant.

```
digraph G {
    rankdir=LR;
    node [shape=record];
    a [label="{ <data> 1 : Tête | <ref>  }", width=1.2]
    b [label="{ <data> 2 | <ref>  }", width=1.2]
    c [label="{ <data> 3 | <ref>  }", width=1.2]
    d [label="{ <data> 4 : Queue | <ref>  }", width=1.2]

    a:ref -> b
    b:ref -> c
    c:ref -> d

}
```

On défile la tête de la file, on décale la tête de la file vers la droite.

```
digraph G {
    rankdir=LR;
    node [shape=record];
    a [label="{ <data> 1 : Tête | <ref>  }", width=1.2, style=filled, fillcolor=red]
    b [label="{ <data> 2 | <ref>  }", width=1.2]
    c [label="{ <data> 3 | <ref>  }", width=1.2]
    d [label="{ <data> 4 : Queue | <ref>  }", width=1.2]

    a:ref -> b
```

```
    b:ref -> c
    c:ref -> d

}
```

On obtient donc :

```
digraph G {
    rankdir=LR;
    node [shape=record];
    b [label="{ <data> 2 : Nouvelle tête | <ref>  }", width=1.2, style=filled,
        fillcolor=green]
    c [label="{ <data> 3 | <ref>  }", width=1.2]
    d [label="{ <data> 4 : Queue | <ref>  }", width=1.2]

    b:ref -> c
    c:ref -> d

}
```

La file est stockée dans un intervalle de tableau, on peut donc décaler la tête de la file en temps constant.

Par contre on n'arrive pas à distinguer File vide de tableau Plein.

→ Ce n'est pas insurmontable :

- Compter le nombre d'éléments
- Garde une case vide entre queue et tête
- …

Cours du 20 février

Chapitre 4 : TDA Listes

Idée : On ne veut pas stocker les éléments dans un espace contigu (on n'a ==pas souvent assez de place==).

Définition

Une *liste* est un ==ensemble ordonné==, et on a accès qu'au premier élément. À partir du premier élément, on peut accéder au deuxième, puis depuis le deuxième, on peut accéder au troisième, etc.

C'est un ensemble ordonné avec la ==fonction `successeur`==.

Implémentation classique

L'implémentation classique se fait avec des listes chaînées :

- Chaque élément de la liste est un maillon

- Chaque maillon contient une valeur et un pointeur vers le maillon suivant

En C :

```
struct maillon {
    T data;
    struct maillon *suivant;
};
```

Description du TDA `Cellule_T`

Le TDA `Liste_T` va utiliser le TDA de `cellule_T` :

- `creerCellule`: () → `cellule_T` : Créer une cellule vide
- `valeurCellule`: `cellule_T` → `T` : Renvoie la valeur de la cellule
- `suivantCellule`: `cellule_T` → `cellule_T` : Renvoie le maillon suivant (une référence)

Les cellules sont utilisées pour représenter les éléments d'une liste.

Dans une cellule on a deux informations :

- Une valeur à stocker
- Une référence à une autre cellule (qui peut être vide indiquant la fin de la liste)

```
digraph G {
  rankdir=LR;
  node [shape=record];
  a [label="{ <data> 1 | *2 | <ref>  }", width=1.2]
  b [label="{ <data> 2 | *3 | <ref>  }", width=1.2]
  c [label="{ <data> 3 | *4  |<ref>  }", width=1.2]
  d [label="{ <data> 4 | NULL | <ref>  }", width=1.2]

  a:ref -> b
  b:ref -> c
  c:ref -> d

}
```

Ici, `*2` est une référence à la cellule `2`.

Pour faciliter la manipulation des cellules, on peut ajouter des fonctions de modifications :

- ==Modifier la valeur== de la cellule
- ==Modifier la référence== de la cellule

Exemple d'implémentation

```
Enregistrement cellule1 {
    T valeur;
    int pC;
}
```

```
Enregistrement cellule2 {
    T valeur;
    cellule2 pC;
}
```

Note : `Enregistrement` est l'équivalent de `struct` en `C`

Opérations sur les listes

- `creerListe`: () → `Liste_T` : Créer une liste vide
- `estVide`: `Liste_T` → `bool` : Vérifie si la liste est vide
- `teteListe`: `Liste_T` → `T` : Renvoie la tête de la liste
- `taille`: `Liste_T` → `int` : Renvoie la taille de la liste

Les fonctions d'insertion à des rangs fixes :

- `insererTete`: `Liste_T`, `T` → `Liste_T` : Insère un élément en tête de liste
- `insererQueue`: `Liste_T`, `T` → `Liste_T` : Insère un élément en queue de liste

Fonction d'insertion à un rang donné :

- `insererPosition`: `Liste_T`, `T`, `int` → `Liste_T` : Insère un élément à une position donnée

Pour (l, x, i) on veut que l'élément x soit inséré à la position i.

```
// On a x1, x2, ..., xn, et on veut insérer x à la position i
digraph G {
  rankdir=LR;
  node [shape=record];
  a [label="{ <data> x1 | <ref>  }", width=1.2]
  b [label="{ <data> x2 | <ref>  }", width=1.2]
  c [label="{ <data> ... | <ref>  }", width=1.2]
  d [label="{ <data> xN | <ref>  }", width=1.2]

  a:ref -> b
  b:ref -> c
  c:ref -> d

}
```

Avec $i = 2$:

```
digraph G {
  rankdir=LR;
  node [shape=record];
  a [label="{ <data> x1 | <ref>  }", width=1.2]
  b [label="{ <data> x | <ref>  }", width=1.2, style=filled, fillcolor=green]
  c [label="{ <data> x2 | <ref>  }", width=1.2]
  d [label="{ <data> ... | <ref>  }", width=1.2]
  e [label="{ <data> xN | <ref>  }", width=1.2]
```

```
  a:ref -> b
  b:ref -> c
  c:ref -> d
  d:ref -> e

}
```

Les fonctions pour supprimer des éléments :

- `supprimerTete`: `Liste_T` → `Liste_T` : Supprime la tête de la liste
- `supprimerQueue`: `Liste_T` → `Liste_T` : Supprime la queue de la liste

Fonction pour supprimer un élément à une position donnée :

- `supprimerPosition`: `Liste_T`, `int` → `Liste_T` : Supprime un élément à une position donnée

```
// On a x1, x2, ..., xn, et on veut supprimer l'élément à la position i
digraph G {
  rankdir=LR;
  node [shape=record];
  a [label="{ <data> x1 | <ref>  }", width=1.2]
  b [label="{ <data> x2 | <ref>  }", width=1.2, style=filled, fillcolor=red]
  c [label="{ <data> x3 | <ref>  }", width=1.2]
  d [label="{ <data> ... | <ref>  }", width=1.2]
  e [label="{ <data> xN | <ref>  }", width=1.2]

  a:ref -> b
  b:ref -> c
  c:ref -> d
  d:ref -> e

}
```

Avec $i = 2$:

```
digraph G {
  rankdir=LR;
  node [shape=record];
  a [label="{ <data> x1 | <ref>  }", width=1.2]
  b [label="{ <data> x3 | <ref>  }", width=1.2]
  c [label="{ <data> ... | <ref>  }", width=1.2]
  d [label="{ <data> xN | <ref>  }", width=1.2]

  a:ref -> b
  b:ref -> c
  c:ref -> d

}
```

- `queue`: `Liste_T` → `Liste_T` : Supprime la tête de la liste (renvoie la liste sans la tête)

Implémentation TDA Liste

On va prendre l'implémentation par listes chaînées : on va représenter une liste par une référence vers la tête de la liste.

```
Enregistrement liste {
    cellule l;
}
```

- Une première implémentation, une cellule est représentée par un index dans un tableau
  - Les éléments de la liste sont stockés à ce qui ressemble à un tableau

Question : Où se trouve la cellule référencée ?

- De façon implicite : on considère que c'est l'indice suivant
  - Dans ce cas, les indices sont stockés dans un intervalle fermé $[0, n-1]$
- De façon explicite : chaque cellule va contenir un pointeur vers la cellule suivante
  - Une cellule : 2 entiers (un index et un pointeur vers la cellule suivante) et la valeur de l'élément

On stock des éléments de la liste dans un tableau de cellules. Pour cette implémentation, il faut la notion de cellule `NULL`.

Insérer revient à demander l'index d'une cellule libre pour l'utiliser pour stocker le nouvel élément (et en refaisant le chaînage si besoin).

On va stocker un tableau de cellules qui va avoir :

- la valeur de l'élément
- un pointeur vers la cellule suivante

```
Enregistrement cellule {
  T val;
  cellule suiv;
}
```

Et on a un tableau de cellules :

```
Enregistrement liste {
  cellule tab[100];
  int tete;
  int taille;
}
```

Exemple

```
digraph G {
  rankdir=LR;
  node [shape=record];


  a [label="{ <data> 0 | <ref>  }", width=1.2];
  b [label="{ <data> 1 | <ref>  }", width=1.2];
  c [label="{ <data> 2 | <ref>  }", width=1.2, style=filled, fillcolor=red];
  d [label="{ <data> 3 | <ref>  }", width=1.2, style=filled, fillcolor=red];
  e [label="{ <data> 4 | <ref>  }", width=1.2];
  f [label="{ <data> 5 | <ref>  }", width=1.2];
  g [label="{ <data> 6 | <ref>  }", width=1.2];

  // On relie tout ce qui n'est pas vide
  a:ref -> b
  b:ref -> e
  e:ref -> f
  f:ref -> g


}
```

Les cases vides sont en rouge.

On essaye de distinguer les cases vides avec des cases pleines avec des valeurs spéciales (ex : -1).

Il a le ==suivant logique== et le ==suivant spacial== (le suivant logique est le suivant dans la liste, le suivant spacial est le suivant dans le tableau).

Ajout possible : On peut rajouter une pile qui contient les cases libres du tableau.

Quand utiliser les listes

On utilise les listes pour représenter les ensembles car on ne connaît pas d'avance la taille de ses ensembles.

- Insertion début : `a` :
  - On demande à `p` 1 case : `p` renvoie 0
  - On stock `a` dans `tab[0].v`
  - On stock LD dans `tab[0].s`
  - On stock dans LD : 0
- Insertion en fin `c` :
  - On demande à `p` 1 case : `p` renvoie 1
  - On stock `c` dans `tab[1].v`
  - On stock -1 dans `tab[1].s`
- Supprimer en début :
  - On stock LD dans `p`
  - Dans LD on stock `tab[LD].s`
- Insérer `b` en fin

– `p` renvoie 0

Exemple d'analyse de complexité

Graphe : $G : (V, E), V :$ ensemble de sommets. $E \subset V \times V$ relation binaire.

```
PP(G: graphe, r: sommet) {
    colorie r en gris
    Pour chaque voisin blanc : v de r Faire {

        PP(G, v)

    }
    colorie r en noir
}
```

==Definition== :

- w est un voicin de v si $(v, w) \in E$
- Chaque sommet a une couleur : blanc, gris ou noir

Avec l'algo PP on a au moins 4 opérations

- Connaître la couleur d'un sommet
- Modifier la couleur d'un sommet
- Créer un graphe
- Connaître l'ensemble des voisins d'un sommet
- Accéder à un sommet

On modifie le programme comme suit :

```
PP(G: graphe, r: sommet, T: graphe) {
    colorie r en gris
    Pour chaque voisin blanc : v de r Faire {

        ajouter à E(T) la pair (r,v)
        PP(G, v)

    }
    colorie r en noir
}
```

On doit rajouter les opérations suivantes :

- Ajouter une arête

Opération	Complexité
1. Tableau de taille n : les indices représentant les sommets et les valeurs les couleurs	Un tableau de taille n (indices représentant les sommets) et chaque case contient 2 champs : La couleur, une liste contenant la liste des voisins
Matrice booléenne de taille $n×n $: case $(i, j) = 1 \leftrightarrow (i, j)$ est une arrête	
Matrice d'adjacence taille : $n^2 + n$	taille : $O(n + n)$

- Connaître la liste des voisins de i :
  - Parcourir la ligne i de la matrice d'adjacence
  - Si la case est à 1, alors i est un voisin de j (ajouter j dans un ensemble)

```
Pour j de 0 à n-1 Faire {
  Si mat[i][j] = 1 Alors {
    Ajouter j à la liste des voisins de i
  }
}
```

==Complexité== : O(nombre de voisins de $i \times$ complexité d'ajout dans L) + $O(n -$ nombre de voisins)

- Suivant le type de L,
  - L est une pile avec empiler en $O(1)$ et dépiler en $O(1)$
    - O(nombre de voisins de $i \times 1$) + $O(n -$ nombre de voisins) = $O(n)$
  - L est un file avec enfiler en $O(1)$ et défiler en $O(1)$
    - O(nombre de voisins de $i \times 1$) + $O(n -$ nombre de voisins) = $O(n)$
  - Si M est une liste avec insertion début en $O(1)$, pareil que pile / file si on insère au début
  - Si L est liste et on fait une inseertion en fin, qui a une compleité en temps $O(m)$.
  - SI L est une liste et on fait une insertion en fin qui a une complexité en temps $O(m)$:

$$\sum_{j \text{ pas de voisin de } i} O(1) + \sum_{j \leq j \leq \text{nombre de voisins}} O(j) = O(n - \text{nombre de voisins} + \text{nombre de voisins}^2)$$

Matrice d'incidence

- Ajouter une arête : (i, j)
  - Ajouter à la ligne $T[i]$ la valeur j :
    - $O(1)$ si au début
    - $O(n)$ si à la fin
    - La complexité dépend de la fonction d'ajout
- Connaître l'ensemble des voisins de i :
  - retourner la ligne $T[i]$
- On estime la complexité de PP

Complexité de `PP(G, r, T)`

```
digraph {
  rankdir=LR;
  a -> b;
  b -> c;
  c -> d;
  c -> e;
  d -> e;
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
}
```

Supposons `r = c`.

- $T = (\{c, d\}, \{c \to d\}), c$ colorié gris

```
digraph {
  rankdir=LR;
  // On colorie l'arrête qu'on explore en rouge, les autres en noir
  c -> d [color=red];
  a -> b;
  b -> c;
  c -> e;
  d -> e;
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
}
```

- $PP(G, d, T)$: voisins blancs de d : $\{e, f\}$
  - T vaudra : $(\{c, d, e\}, \{c \to d, d \to e\})$

```
digraph {
  rankdir=LR;
  c -> d;
  a -> b;
  b -> c;
  c -> e;
  d -> e [color=red];
  e -> f;
```

```
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=gray, style=filled];
}
```

- $PP(G, e, T)$: e colorié gris, voisins blancs de e : $\{f\}$
  - T vaudra : $(\{c, d, e, f\}, \{c \to d, d \to e, e \to f\})$

```
digraph {
  rankdir=LR;
  c -> d;
  a -> b;
  b -> c;
  c -> e;
  d -> e;
  e -> f [color=red];
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=gray, style=filled];
  e [fillcolor=gray, style=filled];
}
```

- $PP(G, f, T)$: f colorié gris, voisins blancs de f : $\{g\}$
  - T vaudra : $(\{c, d, e, f, g\}, \{c \to d, d \to e, e \to f, f \to g\})$

```
digraph {
  rankdir=LR;
  c -> d;
  a -> b;
  b -> c;
  c -> e;
  d -> e;
  e -> f;
  f -> g [color=red];
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=gray, style=filled];
  e [fillcolor=gray, style=filled];
  f [fillcolor=gray, style=filled];
```

```
}
```

- $PP(G, g, T)$: g colorié gris, voisins blancs de g : rien. Donc on colorie g en noir.
  - On sort de la boucle `Pour`, finit et retourne à l'appelant : `PP(G, f, T)`

```
digraph {
  // Comme on n'est pas sur une arrête, on entoure le sommet en vert foncé
  rankdir=LR;
  c -> d;
  a -> b;
  b -> c;
  c -> e;
  d -> e;
  e -> f  [color=red];
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=gray, style=filled];
  e [fillcolor=gray, style=filled];
  f [fillcolor=gray, style=filled, color=darkgreen];
  g [fillcolor=black, style=filled, fontcolor=white];
}
```

- $PP(G, f, T)$: f colorié gris, il n'a plus de voisins blancs. On colorie f en noir.
  - On sort de la boucle `Pour`, finit et retourne à l'appelant : `PP(G, e, T)`

```
digraph {
  rankdir=LR;
  c -> d;
  a -> b;
  b -> c;
  c -> e;
  d -> e [color=red];
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=gray, style=filled];
  e [fillcolor=gray, style=filled];
  f [fillcolor=black, style=filled, fontcolor=white];
  g [fillcolor=black, style=filled, fontcolor=white];
}
```

- $PP(G, e, T)$: e colorié gris, il n'a plus de voisins blancs. On colorie e en noir.
  - On sort de la boucle `Pour`, finit et retourne à l'appelant : `PP(G, d, T)`

```
digraph {
  rankdir=LR;
  a -> b;
  b -> c;
  c -> d [color=red];
  c -> e;
  d -> e;
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=gray, style=filled];
  e [fillcolor=black, style=filled, fontcolor=white];
  f [fillcolor=black, style=filled, fontcolor=white];
  g [fillcolor=black, style=filled, fontcolor=white];
}
```

- $PP(G, d, T)$: d colorié gris, il n'a plus de voisins blancs. On colorie d en noir.
  - On sort de la boucle `Pour`, finit et retourne à l'appelant : `PP(G, c, T)`

```
digraph {
  rankdir=LR;
  a -> b;
  b -> c;
  c -> d;
  c -> e;
  d -> e;
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=black, style=filled, fontcolor=white];
  e [fillcolor=black, style=filled, fontcolor=white];
  f [fillcolor=black, style=filled, fontcolor=white];
  g [fillcolor=black, style=filled, fontcolor=white];
}
```

- $PP(G, c, T)$: c colorié gris, on a h en blanc. On colorie h en gris.
  - T vaudra : $(\{c, d, e, f, g, h\}, \{c \to d, d \to e, e \to f, f \to g, c \to h\})$

```
digraph {
```

```
  rankdir=LR;
  a -> b;
  b -> c;
  c -> d;
  c -> e;
  d -> e;
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h [color=red];
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=black, style=filled, fontcolor=white];
  e [fillcolor=black, style=filled, fontcolor=white];
  f [fillcolor=black, style=filled, fontcolor=white];
  g [fillcolor=black, style=filled, fontcolor=white];
  h [fillcolor=gray, style=filled];
}
```

- $PP(G, h, T)$: h colorié gris, il n'a plus de voisins blancs. On colorie h en noir.
  - On sort de la boucle `Pour`, finit et retourne à l'appelant : `PP(G, c, T)`

```
digraph {
  rankdir=LR;
  a -> b;
  b -> c;
  c -> d;
  c -> e;
  d -> e;
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=gray, style=filled];
  d [fillcolor=black, style=filled, fontcolor=white];
  e [fillcolor=black, style=filled, fontcolor=white];
  f [fillcolor=black, style=filled, fontcolor=white];
  g [fillcolor=black, style=filled, fontcolor=white];
  h [fillcolor=black, style=filled, fontcolor=white];
}
```

- $PP(G, c, T)$: c colorié gris, il n'a plus de voisins blancs. On colorie c en noir.
  - On sort de la boucle `Pour`, finit et retourne à l'appelant : `PP(G, b, T)`

```
digraph {
  rankdir=LR;
  a -> b;
```

```
  b -> c;
  c -> d;
  c -> e;
  d -> e;
  e -> f;
  f -> g;
  g -> d;
  f -> c;
  c -> h;
  d -> f;
  c [fillcolor=black, style=filled, fontcolor=white];
  d [fillcolor=black, style=filled, fontcolor=white];
  e [fillcolor=black, style=filled, fontcolor=white];
  f [fillcolor=black, style=filled, fontcolor=white];
  g [fillcolor=black, style=filled, fontcolor=white];
  h [fillcolor=black, style=filled, fontcolor=white];
}
```

On est revenu à l'appelant de `PP(G, a, T)`, on a donc fini. On a fini d'explorer le graphe, on a donc fini l'algorithme.

==Borner la complexité de cet algo== : c'est borner le nombre le nombre d'appels récursifs et ensuite sommer la complexité de chaque appel récursif.

La complexité d'un appel $PP(G, x, T)$: si on oublie les appels récursifs, on a :

- Calculer au les voisins de x
- Parcourir les voisins de 1 à 1 et si le voisin y est blanc, ajouter 1 arrête à T
- Chaque sommet x, on fait au maximum un appel récursif $PP(G, x, T)$ car la première chose que fait $PP(G, x, T)$ est de colorier x en gris
- Il suffit donc de compter les sommets coloriés gris, qui sont exactement les sommets dans T : c'est exactement les sommets accessibles depuis x

x est accessible depuis r si $\exists x_0, x_1, \dots, x_n$ avec $x_0 = r$ et $\forall 0 \leq i \leq n-1$, il existe une arrête (x_i, x_{i+1}).

==Complexité de $PP(G, \wedge, \phi)$:

$$\sum_{x \text{ accessible depuis } r} \text{Complexité boucle pour parcourir les voisins de } x$$

→ Si matrice d'adjacence et pile pour stocker les voisins :

$$\sum_{x \text{ accessible depuis } \wedge} O(n) = O(n \times Z),\ Z \text{ le nombre de sommets accessibles}$$

$$\sum_{x \text{ accessible depuis } r} O(nbVoisins(x)) \leq \sum_{x \text{ sommet}} O(nbVoisins(x))$$

$$\leq O(m+n),\ m \text{ le nombre d'arrêtes},\ n \text{ le nombre de sommets}$$

Prise de note du 17 mars 2023

Les types de données Inductifs

Définition inductive informelle : ensemble d'objets donnés et des opérérateurs qui permettent de construire des objets de cet ensemble.

Definition 5.1

Une définition inductive d'un ensemble E consiste en :

- Un sous ensemble fini B de E
- Un ensemble K d'opérations ϕ : $E^{ar(\phi)} \to E$ où $ar(\phi)$ est l'arité de ϕ : le nombre de paramètres de ϕ

Exemple 5.1

Posons $B = \{\epsilon\}$: $suc : X \to X$, $x \to (x)$

L'ensemble X défini inductivement par $(B, \{suc\})$ est exactement l'ensemble des entiers naturels.

Exemple 5.2

Posons $A = \{0, 1\}$ appelé alphabet. A^* est l'ensemble des mots sur A est construit inductivement par :

- $B = \{\epsilon\}$: ϵ est appelé mot vide
- $K = \{\phi_0, \phi_1\}$, $ar(\phi_0) = 1$, $ar(\phi_1) = 1$, $\phi_0 : u \to u_0$, $\phi_1 : u \to u_1$ où $u = u_0u_1$ et $u_0, u_1 \in A^*$

Ex : $0, 1, 00 = \phi_0(\phi_0(\epsilon))$, $01 = \phi_1(\phi_0(\epsilon))$

Exemple 5.3

Une définition inductive des listes :

- $B = \{[]\}$, [] est une liste vide
- $K = \{::\}$, $ar(::) = 2$

$x :: xs$, x est l'objet à stocker, xs la liste, le résultat est donc une nouvelle liste.

Exemple 5.4

A un ensemble fini appelé alphabet. $AB \subset (\{(,)\} \cup A)^*$ est défini inductivement par :

- $B = \{\epsilon\}$: ϵ étiqueté vide
- $\forall a \in A$, on a une opération ϕ_a d'arité 2, $\phi_a : g, c \to (a, g, d)$

Exemple 5.5

Posons \$A = {0,1}, quelques exemples d'objets dans AB.

- $\phi_0(\epsilon, \epsilon) = (0, \epsilon, \epsilon)$

```
digraph {
  rankdir=LR;
  // Point
  0 [label="0"];
}
```

- $\phi_0(\phi_0(\epsilon, \epsilon), \epsilon) = (0, (1, \epsilon, \epsilon), \epsilon)$

```
digraph {
  rankdir=LR;
  // Point
  0 [label="0"];
  1 [label="1"];
  0 -> 1;
}
```

5.1 Arborences

Une arborescence c'est un ensemble V muni d'un sommet distingué $r \in V$, appelé racine, et d'une relation binaire $E \subset V \times V$ telle que pour tout $x \in V/\{r\}$, il existe un unique $y \neq x$ tel que $(y, x) \in E$. L'unique y tel que $(y, x) \in E$ est appelé le père de x et est noté $pere(x)$.

On écrira (V, E, r) pour une arborescence.

Une définition inductive des arborescences :

- (A, B) tout singleton est une arborescence
- (A, I) Si $T_1, T_2, ..., T_n$ sont des arborescences, avec $T_i = (V_i, E_i, r_i)$, alors on peut construire une nouvelle arborescence (V, E, r) avec :

$$V = \bigcup_{1 \leq i \leq p}^{n} V_i \cup \{r\}, r \notin \bigcup_{1 \leq i \leq p} V_i$$

$$E = \bigcup_{1 \leq i \leq p}^{n} E_i \cup \{(r, r_i) : 1 \leq i \leq n\}$$

```
// r a des files T_1, T_2 ... T_p
// Afficher les "..."
digraph {
  rankdir=LR;
  // Point
  0 [label="r"];
  1 [label="T_1"];
  2 [label="T_2"];
```

```
  3 [label="..."];
  4 [label="T_p"];
  0 -> 1;
  0 -> 2;
  0 -> 3;
  0 -> 4;
}
```

Quelques terminologies

Soit $T = (V, E, r)$ une arborescence :

- les éléments de V sont appelés ==noeuds==
- les éléments de E sont appelés ==arcs==
- Tout noeud y tel que $(y, x) \in E$ est appelé ==fils de x==
- Tout noeud sans fils est appelé ==feuille==
- Un chemin de taille k est une séquence $(x_0, x_1, ..., x_k)$ de noeuds telle que $(x_i, x_{i+1}) \in E$ pour tout $1 \leq i \leq k$
- Si x est un noeud, on notera T_x l'arborecence (V_x, E_x, x) où $E_x = E \cap (V_x \times V_x)$, V_x l'ensemble des noeuds accessibles depuis x par un chemin
- La hauteur de T notée $h(T)$ est la longueur du plus long chemin de r à une feuille

Exemple 5.6

```
/*

1 -> {2, 3, 5}

3 -> 7

5 -> {8,4,6,9}

8 -> 10

*/
digraph {
  rankdir=TB;
  // Point
  1 [label="1"];
  2 [label="2"];
  3 [label="3"];
  4 [label="4"];
  5 [label="5"];
  6 [label="6"];
  7 [label="7"];
  8 [label="8"];
  9 [label="9"];
  10 [label="10"];
  1 -> 2;
  1 -> 3;
```

```
  1 -> 5;
  3 -> 7;
  5 -> 8;
  5 -> 4;
  5 -> 6;
  5 -> 9;
  8 -> 10;
}
```

- noeuds : $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- arcs : $E = \{(1, 2), (1, 3), (1, 5), (3, 7), (5, 8), (5, 4), (5, 6), (5, 9), (8, 10)\}$
- feuilles : $F = \{2, 4, 6, 7, 9, 10\}$
- hauteur : $h(T) = 4$
- fils de 5 : $F(5) = \{8, 4, 6, 9\}$
- sous-arborescence de 5 : $T_5 = (V_5, E_5, 5)$ où $V_5 = \{5, 8, 4, 6, 9, 10\}$ et $E_5 = \{(5, 8), (5, 4), (5, 6), (5, 9), (8, 10)\}$:

```
/*
5 -> {4,6,8,9}
8 -> 10
*/

digraph {
  rankdir=TB;
  // Point
  5 [label="5"];
  4 [label="4"];
  6 [label="6"];
  8 [label="8"];
  9 [label="9"];
  10 [label="10"];
  5 -> 4;
  5 -> 6;
  5 -> 8;
  5 -> 9;
  8 -> 10;
}
```

Proposition 5.1

Soit $T = (V, E, r)$ une arborescence. Alors :

- $\forall$ neoud x, il existe un unique chemin de r à x
- T contient au moins une feuille
- $h(T) = 1 + \max_{x \in V}\{h(T_x)\}$
- $|E| = |V| - 1$: nombre d'arc = nombre de noeud - 1

Exemple de démonstration par induction

$\forall T = (V, E, \wedge)$, $|E| = |V| - 1$, preuve par induction sur $|V|$.

- $|V| = 1$, $T = (V, \emptyset, r)$, et $V = \{r\}$ (par définition de l'arborescence). Donc $|E| = |V| - 1 = 0$.
- Si $|V| > 1$, alors $\exists\ T_1, T_2, ..., T_p$, pour un certain entier p, où $T_i = (V_i, E_i, r_i)$ est une arborescence et $r_1, r_2, ..., r_p$ sont les fils de r et T_i la sous arborescence de T issue de r_i.

Visuellement :

```
// r a des files T_1, T_2 ... T_p
// Afficher les "..."
digraph {
  rankdir=TB;
  // Point
  0 [label="r"];
  1 [label="T_1"];
  2 [label="T_2"];
  3 [label="..."];
  4 [label="T_p"];
  0 -> 1;
  0 -> 2;
  0 -> 3;
  0 -> 4;
}
```

En particulier, $|V_i| < |V|$. Si on suppose par hypothèse d'induction (P) vraie pour les arborescences avec n noeuds, si T est une arborescence avec $n+1$ noeuds, alors $\forall 1 \leq u \leq p$, T_i est une arborescence avec $n_i \leq n$ noeuds. Par hypothèse d'induction, $|E_i| = |V_i| - 1$.

$$V = \bigcup V_i \cup \{r\}, W = 1 + \sum_{1 \leq i \leq p} n_i$$

$$E = \bigcup_{1 \leq i \leq p} E_i \cup \{(r, r_i)\ :\ 1 \leq i \leq p\}, |E| = p + \sum_{1 \leq i \leq p} |E_i|$$

$$|E| = p + \sum_{1 \leq i \leq p} (n_i - 1)$$

$$|E| = p + \sum_{1 \leq i \leq p} n_i - p$$

$$|E| = \sum_{1 \leq i \leq p} n_i$$

$$|E| = |V| - 1$$

Prise de note de 7 avril 2023

Arbre Rouge Noir

Une ARN c'est un ABR où chaque noeud a une couleur (rouge) ou noir) avec les propriétés suivatnes :

- La racine est noire
- Une noeud R a ses fils N
- Pour tout noeud x de l'arbre, le nombre de noeud noirs sur le chemin de x à une feuille est le même

On notera $h_b(x)$ le nombre de noeud noirs sur le chemin de x à une feuille.

Ce qui nous intéresse c'est de montrer que l'on peut maintenir la propriété d'un ARN après insertion et suppression en temps $O(\log n)$.

```
digraph {
  rankdir=TB;
  // Point
  10
}
```

```
digraph {
  rankdir=TB;
  // Point
  10 [color=black];
  5 [color=red];
  10 -> 5;
}
```

```
digraph {
  rankdir=TB;
  // Point
  10 [color=black];
  5 [color=red];
  20 [color=red];
  10 -> 5;
  10 -> 20;
}
```

La figure suivante n'est pas un ARN car le chemin de 10 à 6 a plus de noeuds que le chemin de 10 à 20.

```
digraph {
  rankdir=TB;
  // Point
  10 [color=black];
  5 [color=red];
  20 [color=red];
  6 [color=black];
  10 -> 5;
  10 -> 20;
  5 -> 6;
}
```

Lemme 5.1

Un ARN T ayant n noeuds internes a une hauteur $O(\log n)$.

Preuve

Il faut montrer par récurrence que $|T_x| \geq 2^{Ab(x)} - 1$ pour tout noeud x de T.

Si h est la hauteur de T, d'après le 3, un chemin depuis la racine vers une feuill doit contenir au moins $\frac{1}{2}h$ noeuds noirs. En combinant ceci avec A., on a $n \geq 2^{h-1} - 1$.

Dans les opérations d'insertion et suppression, on va utiliser les deux opérations suivantes qui permettent de réorganiser une ARN.

```
digraph {
  rankdir=TB;
  // Point
  x -> {alpha, y}
  y -> {beta, delta}
}
```

```
digraph {
  rankdir=TB;
  // Point
  y -> {delta, x}
  x -> {alpha, beta}
}
```

```
digraph {
  rankdir=TB;
  // Point
  x -> {alpha, y}
  y -> {beta, delta}
}
```

```
digraph {
  rankdir=TB;
  // Point
  y -> {delta, x}
  x -> {alpha, beta}

}
```

Propriété

Les deux opérations rotation gauche et rotation droite maintiennent les propriétés d'un ARN.

On va maintenant s'intéresser à l'insertion et la suppression. Avant, on ajoute ces deux hypothèses de simplification :

- Tout noeud a 2 fils : qui à ajouter un fils
- Tout noeud ϵ a un père : qui à supprimer un fils

Insertion

Elle se fait en 2 étapes

- Insérer comme dans un abr et marquer le nouveau noeud R : fonction insertion dans un abr
- Réorganiser en ARN : condition à respecter

En ajoutant ce noeud, on a violé peut-être la propriété 3, le père du nouveau noeud est aussi R. On va maintenant réorganiser pour satisfaire cette propriété (la seule non respectée).

Notations :

- \$z = \$ le nouveau noeud
- gp = père(z) = père(père(z))

```
digraph {
  rankdir=TB;
  // Point noirs
  11 [color=black];
  1 [color=black];
  7 [color=black];
  14 [color=black];
  13 [color=black];
  16 [color=black];

  // Point rouges
  2 [color=red];
  5 [color=red];
  8 [color=red];
  15 [color=red];

  // Lien
  11 -> {2, 14};
  2 -> {1, 7};
  7 -> {5, 8};
  14 -> 15;
  15 -> {13, 16};
}
```

```
insererARN(T, 4)
  - insererABR(T, 4)
  - colorier nouveau noeud en rouge
  - réorganiser si pas ARN
```

On a deux cas :

- $pere(z) = filsG(gp)$

```
digraph {
  rankdir=TB;
  // Point noirs
  gp [color=black];
  pere [color=red label="prere(z)"];
  y [color=black];

  gp -> {pere, y};
}
```

- $pere(z) = filsD(gp)$

```
digraph {
  rankdir=TB;
  // Point noirs
  gp [color=black];
  yz [color=red label="prere(z)"];
  y [color=black];

  gp -> {y, yz};
}
```

```
CorrectionInsertion(A,z) {

  while (pere(z) et z) {

    Si (y est R) {
      colorier pere(z) et y en N
      colorier gp en R
      z = gp
    }

    Si (y est N et z est filsD(pere(z))) {
      rotationGauche(A, pere(z))
      z = pere(z)
    }

    Si (y est N et z est filsG(pere(z))) {
      colorier pere(z) en N
      colorier gp en R
      rotationDroite(A, gp)
    }

  }

  colorier racine en N

}
```

Cas 1.3 On effectue `rotationGauche(pere(z))` et on passe au cas 1.3.

```
digraph {
  rankdir=TB;
  // Point noirs
  11 [color=black];
  14 [color=black];
  8 [color=black];
  5 [color=black];
  1 [color=black];

  // Point rouges
  7 [color=red];
  2 [color=red];
  4 [color=red];
  15 [color=red];

  11 -> {14,7};
  7 -> {8,2};
  2 -> {5,1};
  5 -> 4;
  14 -> 15;
}
```

```
digraph {
  rankdir=TB;
  // Point noirs
  7 [color=black];
  5 [color=black];
  1 [color=black];
  8 [color=black];
  14 [color=black];

  // Point rouges
  2 [color=red];
  11 [color=red];
  4 [color=red];
  15 [color=red];

  7 -> {2,11};
  2 -> {5,1};
  5 -> 4;
  11 -> {8,14};
  14 -> 15;
}
```

Complexité en temps de `CorrectionInsertion`

- Le nombre d'itérations de la boucle est le nombre de fois où le cas 1.1 est appliqué
-
  - (≤ 1 fois pour le cas 1.2)
-

- (≤ 1 fois pour le cas 1.3)

Comme à chaque fois que cas 1.1 est appliqué, à la prochaine itération, $z = pere(pere(z))$, le nombre d'itérations est barré par la hauteur de l'arbre $0 \leq h \log n$.

Suppression

Elle se fait également en 2 étapes :

- Supprimer comme dans un abr et marquer le noeud R : fonction suppression dans un abr
- Réorganiser en ARN : condition à respecter

Pour supprimer dans un ABR un noeud z :

- Si z est une feuille, on le supprime
- Si z a un seul fils, on le supprime et :
  - $z = filsD(pere(z)) \implies y = filsD(pere(z))$
  - $z = filsG(pere(z)) \implies y = filsG(pere(z))$
- Si z a deux fils, on le remplace par son successeur y et on supprime y (qui est une feuille ou a un seul fils)

```
digraph {
  rankdir=TB;
  pere [label="pere(z)"];
  z [label="z"];
  y [label="y"];

  pere -> z
}
```

Notation

Notons y le noeud réellement supprimé.

- Si y est R, il est facile de vérifier que l'arbre est toujours un ARN
- À partir de maintenant, y est N, x est un fils de y :
  - Si y $feuille, x = \epsilon$
  - y a un seul fils, x est ce fils

Notons w le frère de x après avoir supprimé y.

```
digraph {
  rankdir=TB;
  pere [label="pere(y)"];
  y [label="y"];
  x [label="x"];
  w;
}
```

Gagner des pts

EXEMPLE ARBRE ROUGE NOIR

-> Insertion, 2 fois cas 1.1 -> On arrive 1.2 -> On arrive 1.3

-> On explique, toutes les conditions sont respectées

Cours du 28 avril 2023

Correction suppression `(A,x)` :

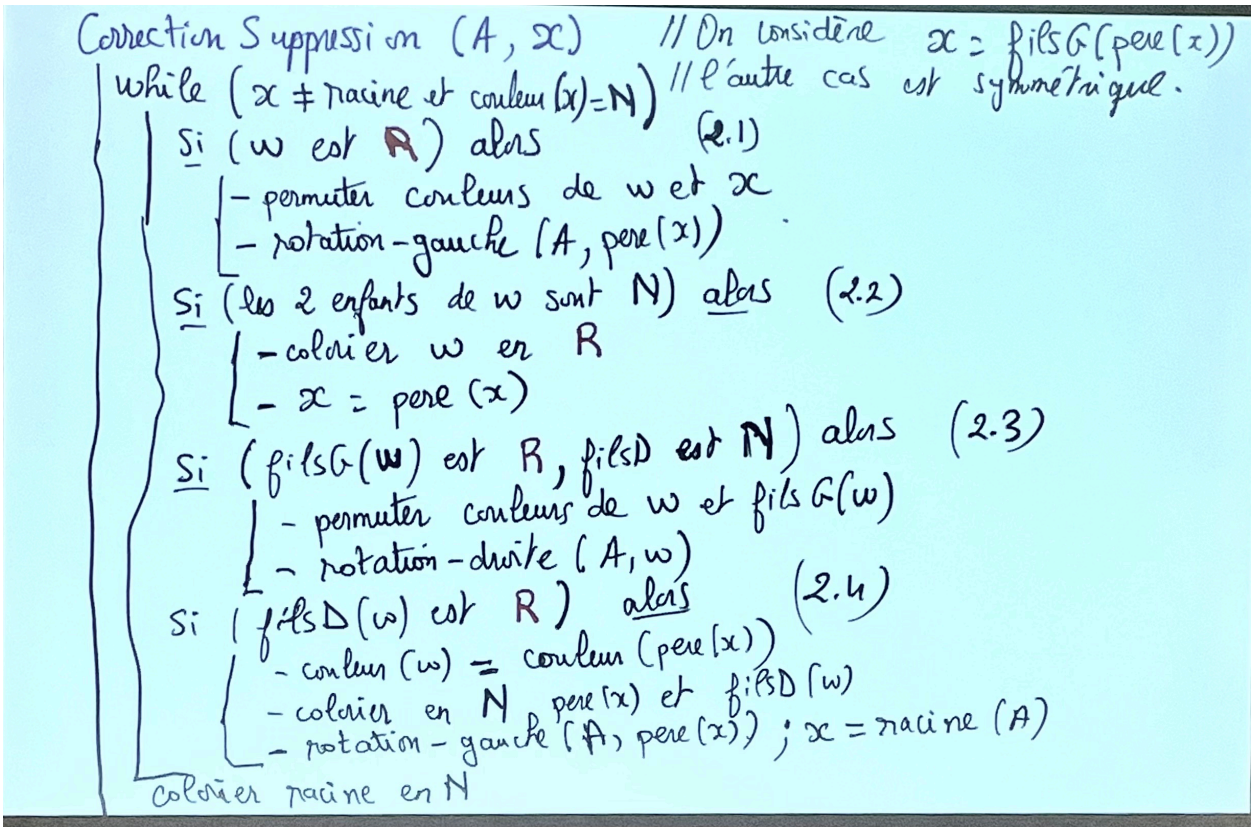

Figure 1: Correction suppression

Remarque

- Dans le cas 2.1, on se ramène aux [...]

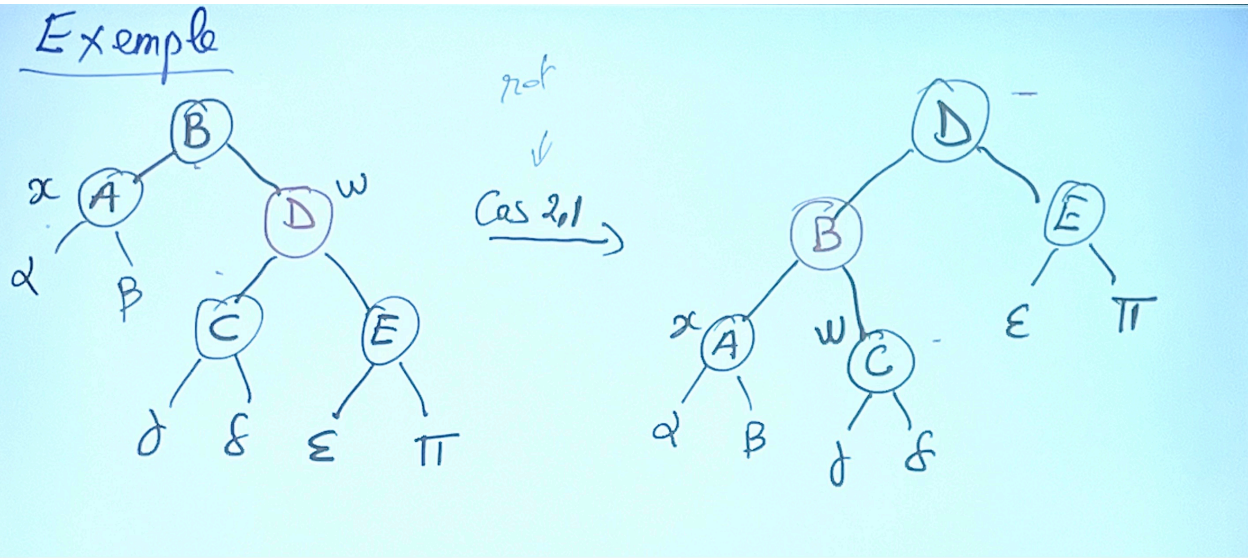

Figure 2: Exemple

Exemple

- Sur le schémas ci-dessus, on a une rotation gauche sur B

Exemple 2

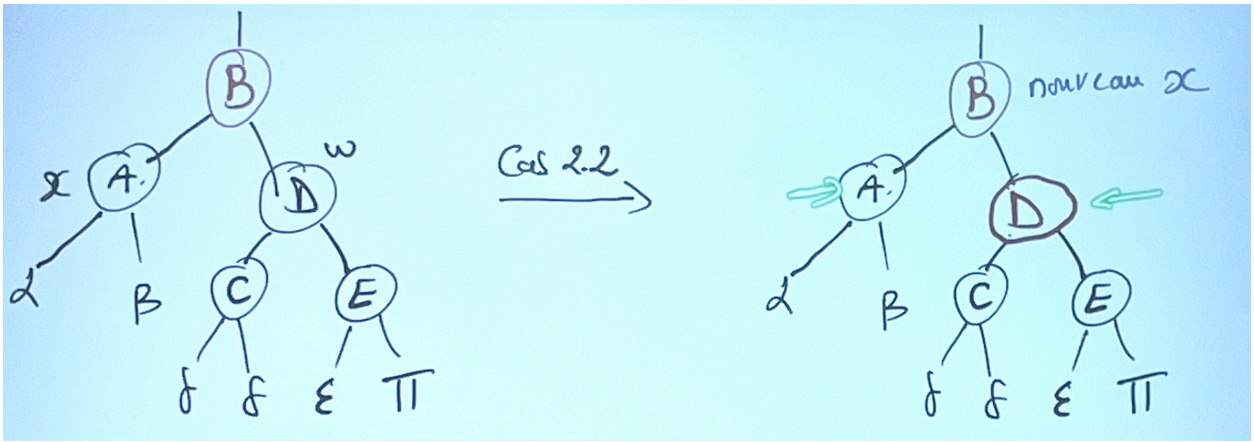

Figure 3: Exemple 2