# L2 Informatique - Notes du cours de Structure de données

Raphaël Cazenave

11 mars 2013

# Table des matières


0.0.0.0.1 Livres . . . . . 4

**1 Niveau de description** **5**
1.1 Structure générale d'un ordinateur . . . . . 5
1.1.1 Partie 1 . . . . . 6
1.1.2 Partie 2 . . . . . 6
1.1.3 Partie 3 . . . . . 6
1.1.4 Partie 4 . . . . . 6
1.1.5 Remarques . . . . . 6
1.2 Mémoire centrale . . . . . 7
1.3 Langages . . . . . 7
1.3.1 Assembleur . . . . . 7
1.3.2 Langages de haut niveau . . . . . 7
1.3.3 Les plus grands paradigmes de programmation . . . . . 8
1.3.4 Qualités d'un langage . . . . . 8

**2 Algorithmes, valeurs, types et élément du langage** **9**
2.1 Algorithmes . . . . . 9
2.1.0.0.2 Théorème 1 : Indécidabilité . . . . . 9
2.1.0.0.3 Théorème 2 : Thèse de Church-Turing . 9
2.1.0.0.4 Théorème 3 : Machine de Turing Universelle . . . . . 9
2.2 Données . . . . . 9
2.3 Types . . . . . 10
2.3.1 Types composés . . . . . 10
2.3.1.1 Type produit . . . . . 10
2.3.1.2 Type Somme . . . . . 10
2.3.1.3 Type Enregistrement . . . . . 11
2.3.2 Tableaux statiques . . . . . 11
2.4 Syntaxe du langage . . . . . 11

**3 Type de données abstraits (Pile, file, listes)** **13**
3.1 Définition du TDA . . . . . 13
3.1.1 Définition . . . . . 13
3.1.2 Exemple . . . . . 14

3.2 TDA Dictionnaire . . . 14
3.2.1 Exemple d'implémentation . . . 14
3.3 TDA Ensemble Dynamique . . . 15
3.4 TDA Pile . . . 16
3.4.1 Définition et opérations . . . 16
3.4.2 Axiomes . . . 16
3.4.3 Exemple 1 . . . 17
3.4.4 Exemple d'implémentation . . . 17
3.4.5 Applications . . . 17
3.4.6 Evaluation d'expressions arithmétiques . . . 17
3.4.6.1 Exemple . . . 17
3.5 TDA Pile . . . 19
3.5.1 Definition et Opérations . . . 19
3.5.2 Axiomes . . . 19
3.5.3 Implémentations . . . 19
3.5.4 Applications des files . . . 20
3.6 TDA Liste . . . 20
3.6.1 Definition . . . 20
3.6.2 Opérations . . . 20
3.6.3 Implémentation . . . 21
3.6.4 Implémentation 1 . . . 21
3.6.5 Implémentation 2 . . . 22
3.6.6 Implémentation 3 . . . 23
3.6.6.1 Extensions des listes . . . 24
3.6.6.1.1 Comment savoir que l'on a tout parcouru ? . . . 24
3.6.7 Gestion de l'espace libre . . . 24
3.6.8 Applications . . . 25

**4 Eléments de complexité** **26**
4.1 Critères d'évaluation . . . 26
4.2 Evaluation du temps d'exécution . . . 26
4.3 Notation O, omega, theta . . . 27
4.3.1 Exemples . . . 27
4.3.1.0.2 exemple . . . 28
4.4 Règles de calcul . . . 28
4.4.1 Exemple . . . 29
4.5 P vs NP . . . 29
4.5.0.0.3 classification : . . . 29
4.5.0.0.4 exemple . . . 29
4.5.1 NP-Complet . . . 30
4.6 Remarque . . . 30
4.6.1 Proposition . . . 30
4.6.1.0.5 exemple . . . 31

**5 Types inductifs** **32**
5.1 Exemple 5.1 . . . 32
5.2 Exemple 5.2 . . . 32
5.3 Exemple 5.3 . . . 33
5.4 Exemple 5.4 . . . 33
5.4.1 ex . . . 33
5.5 oO . . . 33
5.5.1 Ex . . . 33
5.6 Exemple 5.5, TDA liste . . . 34
5.7 Arborescences . . . 34
5.7.1 Proposition 5.1 . . . 35
5.7.2 Implémentation . . . 35
5.7.3 Arbre binaire . . . 36
5.7.4 Parcours d'un arbre . . . 36
5.7.4.1 Parcours en profondeur . . . 36
5.7.4.2 Parcours en largeur . . . 36
5.7.4.3 Exemple . . . 36
5.7.4.4 Préfixe . . . 37
5.7.4.5 Suffixe . . . 37
5.7.4.6 Infixe . . . 37
5.7.5 Proposition 5.2 . . . 37
5.7.6 Applications . . . 37
5.8 Tas binaires . . . 37
5.8.0.1 Application - Files de priorités . . . 38
5.9 Arbres de recherche . . . 39
5.9.1 Arbre binaire de recherche (ABR) . . . 39
5.9.1.1 Rechercher . . . 39
5.9.1.2 Inserer . . . 40
5.9.1.3 Supprimer . . . 40
5.9.1.4 Remarque . . . 40
5.9.1.5 Théorème 5.2 . . . 40
5.9.2 Arbre Rouge Noir (ARN) . . . 40
5.9.2.1 Proposition 5.3 . . . 40
5.9.2.2 primitives rotation gauche et droite . . . 41
5.9.2.2.1 Cas 1.1 . . . 41
5.9.2.2.2 Cas 1.2 . . . 41
5.9.2.2.3 Cas 1.3 . . . 41
5.9.2.2.4 . . . 41
5.9.2.2.5 Cas 1 . . . 42
5.9.2.2.6 Cas 2 . . . 42
5.9.2.2.7 Cas 3 . . . 42
5.9.2.2.8 Cas 4 . . . 42
5.9.2.2.9 Application . . . 42
5.9.2.2.10 Gestion de partition . . . 42

**0.0.0.0.1 Livres** Introduction à l'algorithmique (Cormen, Leiserson, Rivest, ...)
The art of computer programming (Knuth)
Le langage C (Ritchie et Kennighan)

# Chapitre 1

# Niveau de description

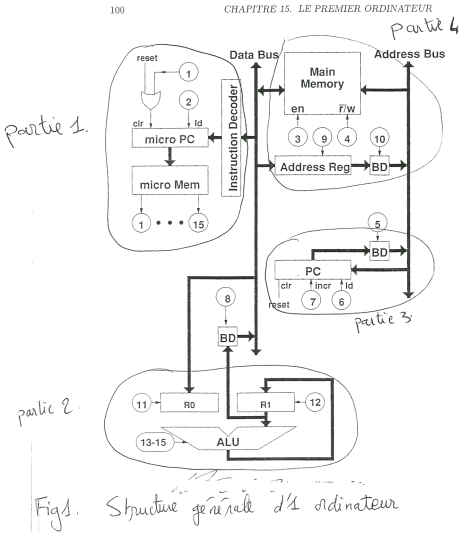
Exemples d'algorithmes :
- Calcul des impots à Babylone
- Algorithmes d'Euclide (PGCD)
- Crible d'Ératosthène (recherche des nombres premiers)

L'objectif de ce cours est l'étude des données manipulées par les algorithmes.

## 1.1 Structure générale d'un ordinateur

Le modèle actuel des ordinateurs est la machine à registre de Von Neuman. C'est un modèle équivalent aux machines de Turing, et au $\lambda$-calcul.

Les machines sont généralement composées de :
- Mémoire adressable (Lecture / écriture)
- Unité arithmétique-logique (processeur)
- Compteur d'instruction




### 1.1.1 Partie 1

**Micro-mémoire :** stockage des instructions de l'ordinateur

**Compteur micro-mémoire**

**Décodeur d'instructions**

### 1.1.2 Partie 2

**Unité arithmétique logique**

2 [registres] de stockage

### 1.1.3 Partie 3

Compteur ordinale, pour incrémenter et décrémenter les instructions dans la mémoire(pointeur d'instruction).

### 1.1.4 Partie 4

**Mémoire centrale :** stockage des données (programmes)

**registres d'adresses :** certaines données en mémoire représentent les adresses et il faut les passer au bus d'adresse.

### 1.1.5 Remarques

Chaque partie est un circuit séquentiel utilisant des portes logiques (à 2 états).

Du coté du programmeur, seules certaines parties sont visibles :
- Mémoire centrale
- Registres
- Compteur ordinal (pointeur d'instruction)

## 1.2 Mémoire centrale

Elle est divisée en blocs, appellés mots, de même taille. La taille des mots représente la taille des addresses et des registres.

Si n est la taille des mots, la capacité maximale de la mémoire est $2^n$.

Exemple : sur un machine 32bits (mots de 4 octets) : $2^{32} \approx 4292967266$mots adressables $\approx 4GB$

## 1.3 Langages

Le langage machine est l'ensemble des instructions supportées par la machine (celles stockées en micro-mémoire).

C'est un langage difficile à manipuler, d'où la création des langages de programmation qui s'inspirent de nos langues.

### 1.3.1 Assembleur

Les instructions sont en héxadécimal (par regroupement de bits).

Exemple : x86 : signifie écrire 97 sur le registre AL.
On écrirait cela en ASM : MOVB 0x61 %AL

L'assembleur est moins verbeux que le langage machine, et dépendant de l'architecture de l'ordinateur (non-portable).

### 1.3.2 Langages de haut niveau

Un pc se base sur les travaux des linguistes (Chomsky...) pour construire des modèles mathématiques qui conviennent à a nos besoin.

Un langage c'est :
- Un ensemble d'expressions (chacune à un sens).
- Un ensemble de règles qui permettent d'écrire des programmes compréhensibles.
- Un certain nombre d'outils pour traduire les programmes en langage machine.

Avantage : 1 seul programme pour toutes les machines (seuls les compilateurs changent).

On peut classer les langages en 2 groupes : les langages compilés et les langages interprétés.

Un langage compilé :
- Compilation des fichiers sources en langage machine
- Edition des liens pour la production d'un executable

### 1.3.3 Les plus grands paradigmes de programmation

**Impératif :** Utilisation d'effets de bords sur les objets manipulés, on les manipule à l'aide d'adresses mémoire.

**Objet :** Les valeurs manipulées sont regroupées en des ensembles et chaque ensemble autorise des opérations.

**Fonctionnel :** La fonction est l'objet de base, ce paradigme est basé sur le $\lambda$-calcul.

**Concurentiel :**

**Logique :** Consiste à exprimer les problèmes et les algorithmes sous forme de prédicats (comme en Prolog)

### 1.3.4 Qualités d'un langage

**Verbosité :** Faire peu avec beaucoup de mots.

Les méthodes pour structurer les programes.

**Complétude :** Peut-on programmer tout ce que permet les langages machines.

Manipulation des données de bas-niveau.

# Chapitre 2

# Algorithmes, valeurs, types et élément du langage

## 2.1 Algorithmes

Origine du mot algorithme : mathématicien arabe Al-Khawarizmi.

Algorithme : série d'opérations ayant pour but de résoudre un problème. Un algorithme prend des données en entrée, et produit un résultat en sortie.

La mise en oeuvre d'un algorithme (implémentation) consiste à l'écriture des différentes étapes dans un langage.

**2.1.0.0.2 Théorème 1 : Indécidabilité** On ne peut pas résoudre tous les problèmes avec des algorithmes, ex : (problème de l'arret, vérification d'un programme)

**2.1.0.0.3 Théorème 2 : Thèse de Church-Turing** Les problèmes ayant une solution algorithmique sont exactement ceux que l'on peut résoudre avec une machine de Turing ou avec un modèle équivalent (Von Neuman, $\lambda$-calcul), grâce à la théorie de la calculabilité.

**2.1.0.0.4 Théorème 3 : Machine de Turing Universelle** $\exists$ une machine de Turing Universelle (qui peut simuler toute machine de Turing).

## 2.2 Données

Une données est caractérisée par un identifiant (une suite de caractères alphanumériques commençant par une lettre).

Un type est un ensemble de valeurs muni d'un ensemble d'opérations, et des axiomes (propriétés).

Un type est représenté par un identifiant.

Exemple :
"int" prenant une valeur entière, avec des opérations (+, - , *, /, %), avec des axiomes (division par 0 interdite...).

Une constante est uen valeur d'un certain type.
Une donnée c'est soit une constante, soit une entité qui a un type, une valeur, et un identifiant.
Une variable est un identifiant qui désigne un espace de stockage (dans les langages impératifs, un id correspond à une adresse en mémoire).

## 2.3 Types

- Les types primitifs : valeurs atomiques (non-décomposable)
- Les types composés : construit à partir d'autres types.

Ex : tableau : ensembe d'espaces de stockage stockant des valeurs de même type, avec un accès par indice.

### 2.3.1 Types composés

#### 2.3.1.1 Type produit

Soit $T_1, T_2, ..., T_n$ des types, Alors $T_1 \times T_2 \times ... \times T_n$, constitué des valeurs $\{(x_1, x_2, ..., x_n) | x_i \in T_i\}$ est un type produit.

#### 2.3.1.2 Type Somme

Soit $T_1, T_2, ..., T_n$ des types, Alors $T_1 \bigcup T_2 \bigcup ... \bigcup T_n$, est un type somme.
Ocaml :

```
type couleur = Bleu | Blanc | Rouge;;
let b = Bleu;;
```

C :

```
union nom_type = {
        T1 chp1,
        T2 chp2,
        T3 chp3
};
```

#### 2.3.1.3 Type Enregistrement

C'est un type produit composé de plusieurs entités, appellées champs. Chacun avec un identifiant et un type (c'est un type produit sans ordre entre les champs).

Ocaml :

```
type complexe = {re : float; im : float};;
let a = {re = 4.00; im = 2.00};;
```

C :

```
struct nom_type {
        float re;
        float im;
};
```

Java : on peut considérés les classes comme des types enregistrements.

### 2.3.2 Tableaux statiques

- Taille fixée lors de la création
- Accès avec un indice en temps constant
- Eléments consécutifs en mémoire

Si tab est un tableau, tab[0] est la première case, on accéde à la j-ième case par tab[0] + (c)j, où c est la taille en mémoire d'un élément du tableau. Avantage : on a juste à stocker l'adresse du premier élément. Inconvénient : il faut de l'espace contigue en mémoire. Il faut stocker la taille si le langage ne le fait pas.

## 2.4 Syntaxe du langage

proche du c

**fin d'instruction :** ;

**structures itératives et conditionnelles :** idem au c

**déclaration :** comme en c

**affectation :** var := valeur ;

Types primitifs :

- int
- bool
- char (')
- string ("")
- float

Affichage : print val ;

Comparaison : comme en C

Concaténation : ^

opérateurs entiers : +, -, *, /, %

opérateurs flottant : +, -, *, /
opérateurs bool : and, or, not

types composés : Type nomTab[n] ;
indices de 0 à n-1.
accès : nomTab[i] ;

enregistrements : comme en C
fonctions : comme en C
par valeur ou par référence (en spécifiant le &)
implicitement un tableau est passé par référence.

Passage par copie :
- Avantages :
- - Sécurité
  - Pas d'effet de bords
- Inconvénients :
- - Rapidité
  - Cout mémoire

Passage par référence :
- Avantages :
- - Rapidité
  - Cout mémoire
- Inconvénients :
  - Sécurité
  - Effets de bords

# Chapitre 3

# Type de données abstraits (Pile, file, listes)

## 3.1 Définition du TDA

Lorsque l'on écrit un algorithme, on manipule des données, regroupées en ensembles. Les données subissent des modifications (insertion, suppression). Chaque ensemble regroupe des valeurs et est défini par un ensemble d'opérations à effectuer sur les valeurs. Les valeurs et les opérations sont définies par les spécifications, le cahier des charges de l'algorithme. Les ensembles et les opérations spécifiées par le cahier des charges sont appellés : Type de données à modéliser. La formalisation mathématique des TDM est appellée TDA (et qui spécifie comment chaque opération doit se comporter). Les types de données manipulés par le programme sont appellés Type de données concrets (TDC) et sont une implémentation des TDA.

### 3.1.1 Définition

Un TDA est une entité constituée d'uns signature et d'un ensemble d'axiomes.

- La signature est composée d'un identifieur (par lequel on désigne le TDA), les identifiants et signatures des opérations, et les types prédéfinis utilisés.
- Les axiomes définissent les comportements des opérations sur les valeurs des TDA.

A la définition d'un TDA se pose la question de la complétude (est ce que tous les comportements sont modélisés) et de la consistance (pas de contradiction entre les axiomes) des axiomes.

### 3.1.2 Exemple

Supposons que l'on veuille manipuler un ensemble de réels, on peut définir le TDA VecteurRéel qui utilise les réels et les entiers comme type prédéfinis et avec les opérations suivantes :

- ObtenirIème : VecteurReels * Entier → Reel
- ModifierIème : VecteurReels * Entier * Reel → VecteurReels
- Sup : VecteurReels → Entier
- Inf : VecteurReels → Entier

On peut utiliser un axiome du genre :

$$\text{Inf}(V) \leq i \leq \text{Sup}(V) \Longrightarrow \text{Obtenir}(\text{Modifier}(V, i, e), i) = e$$

## 3.2 TDA Dictionnaire

Un dictionnaire est un ensemble qui supporte les opérations suivantes :

- insertion
- suppression
- test d'appartenance

Le TDA *Dico_t* utilise les types bool et T et les opérations suivantes :

- créerDico : () → Dico_T
- insérerDico : Dico_T * T → Dico_T
- supprimerDico : Dico_T * T → Dico_T
- appartenanceDico : Dico_T → bool

Les axiomes que l'on peut définir qui expriment en particulier que créerDico cré un dictionnaire vide :

$\forall x \in T$ :

- appartenanceDico(créerDico(), x) = false
- appartenanceDico(supprimerDico(D, x), x) = false
- appartenanceDico(insérerDico(D,x), x) = true

### 3.2.1 Exemple d'implémentation

On va implémenter un dictionnaire d'entiers (T= int) :

```
struct Dico_int {
  int tab[MAX];
  int i;
}
```

```
Dico_int creerDico() {
  Dico_int d;
  creeCaseVide();
  d.i = 0;
  return d;
}
```

```
bool appartenanceDico(Dico_int d, int x) {
  int i;
  for(i = 0; i < MAX; ++i) {
    if (d.tab[i] == x) {
      return true;
    }
  }
  return false;
}
```

```
Dico_int insererDico (Dico_int d, int x) {
  if (d.i < MAX) {
    d.tab[i] = x;
    d.i = prochainLibre(d);
    return d;
  }
  exit (ERROR);
}
```

```
Dico_int supprimerDico (Dico_int d, int x) {
  int i;
  for(i = 0; i < MAX; ++i) {
    if (d.tab[i] == x) {
      marquerLibre(d, i);
      return d;
    }
  }
  return (Error);
}
```

## 3.3 TDA Ensemble Dynamique

Un ensemble dynamique est un dictionnaire avec la propriété que les objets stockés sont linéairement ordonnés(ordre total, c'est à dire que chaque élément peut être comparé).

Le TDA Ensemble_Dynamique_T utilise les types bool et T et à comme opérations :

- creerEnsembleDynamique : () → Ens_Dyn_T
- insererEnsembleDynamique : Ens_Dyn_T * T → Ens_Dyn_T
- supprimerEnsembleDynamique : Ens_Dyn_T * T → Ens_Dyn_T
- appartenanceEnsembleDynamique : Ens_Dyn_T * T → bool
- minEnsembleDynamique : Ens_Dyn_T → T

– maxEnsembleDynamique : Ens_Dyn_T → T
– sucEnsembleDynamique : Ens_Dyn_T → T
– precEnsembleDynamique : Ens_Dyn_T → T
Exemple d'utilisation : notes d'élèves, graphe pour la fabrication d'une pièce (éléments ordonnée).

Les axiomes sont ceux du TDA dictionnaire, plus ceux qui expriment que le minimum (respectivement le maximum) n'a pas de prédécesseur (respectivement de successeur).
Il faut également ceux qui expriment que tout élément (sauf le minimum) ai un prédécesseur et un successeur (sauf le maximum).

## 3.4 TDA Pile

C'est un TDA qui est un cas particulier (avec des restrictions sur les opérations).

Lorsque l'on subdivise un programme en sous-programmes, il faut avoir un moyen à la fin de la fonction appellée, de retourner à l'instruction suivant de la fonction appellante (dans un ordinateur, on ne dispose que d'un compteur ordinal(pointeur d'instruction)).

Un moyen basique c'est de stocker dans le code machine de la fonction appellée l'instruction suivante de la la fonction appellante et d'y faire un *Jump*. Mais cela pose un problème d'écrasements lors de la récursivité.

Une pile est LIFO.

### 3.4.1 Définition et opérations

Le TDA Pile_T utilise les types bool et T et a comme opérations :
– creerPile : () → Pile_T
– estVidePile : Pile_T → bool
– empiler : Pile_T * T → Pile_T
– depiler : Pile_T → Pile_T
– sommetPile : Pile_T → T

### 3.4.2 Axiomes

soit x de type T, soit P une Pile_T.

– estVidePile(creerPile()) = true
– estVidePile(empiler(P, x)) = false
– sommetPile(empiler(P, x)) = x

– depiler(empiler(P, x)) = P

### 3.4.3 Exemple 1

empile(depile(empile(empile(creerPile(), 5), 6)), 7)
trace :

```
_
5
6 ,  5
5
7 ,  5
```

### 3.4.4 Exemple d'implémentation

- on peut utiliser un tableau statique (avec une taille MAX) et une variable de type int qui pointe toujours sur l'indice du sommet de la pile. Pour créer une pile, on met à -1 dans la variable indexant le sommet de la pile, nommé sommet (sP).
- Pour empiler, on stocke dans tab[sP+1] et on incrémente sP (on gère les cas où le tableau peut être plein)
- Pour dépiler, on décrémente sP (on gère le cas où sP=-1)

Les piles sont très utilisées dans les langages *contexte free*, ce sont des langage utilisant des piles.

### 3.4.5 Applications

- gestion es appels de programmes (ou des contextes des processus)
- vérification syntaxique lors de la compilation dans les langages de programmation (langage contexte-free = automates à piles)
- l'évaluation d'expressions dans les calculatrices

### 3.4.6 Evaluation d'expressions arithmétiques

On supposera que les nombres sont des chiffres. Toutes les opérations sont binaires (+, -, *, /) On supposera que l'expression est donnée sous forme d'un tableau de caractères, terminé par le caractère #. On fait l'évalutation en 2 étapes.

- on transforme l'expression en notation polonaise (on enleve les parenthèses
- on évalue l'expression en notation polonaise

#### 3.4.6.1 Exemple

expression infixe : 3*((5+(2*3))+4)# expression préfixe (polonaise inversée) : 3 5 2 3 * + 4 + * #


```
void (char eau[], char[]) {
  int i=0, j=0;
  char x = eau[i];
  Pile_Char P = creerPile();
while(x!= '#') {
  switch(x) {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      eap[j++]=x;
    case '(': empiler(P,x);
    case ')':
      while(sommetPile(P) != '(') {
        eap[j++] = sommetPile(P);
        depiler(P);
      }
      depiler(P);
    default:
      while(priorite(sommetPile(P)) >= priorit (x)) {
        eap[j++] = sommetPile(P);
        depiler(P);
      }
      empiler(P, x);
  }
  }
  x = eau[++i];
  while (estVidePile(P) = false) {
  eap[j++] = sommetPile(P);
  depiler(P);
  }
```

```
  eap[j] = '#';
}
```

## 3.5 TDA Pile

Lorsque l'on gère des accès à des ressources limités à plusieurs personnes , on doit gérer les priorits d'accès.

exemple : pool d'impression d'une imprimante à mémoire. la gestion des temps de calculs du processeur accordée aux processeurs par l'os.

On utilise le TDA File qui est une structuer de donnée de type FIFO.

### 3.5.1 Definition et Opérations

Le TDA File_T utilise les types bool et T et a comme opérations :
- creerFile : () → File_T
- estVideFile : File_T → bool
- enfiler : File_T * T → File_T
- defiler : File_T → File_T
- teteFile : File_T → T

### 3.5.2 Axiomes

qqsoit F appartenant à File_T et x appartenant à T :

- estVideFile(creerFile())= true
- estVideFile(enfiler(F, x)) = true
-
```
teteFile(enfiler(F, x)) = {
  teteFile(F) SI F != 0
  x SINON
}
```
- estVideFile(F) = false => teteFile(enfiler(F, x)) = teteFile(F)
- defiler(enfiler(creerFile(), x) = creerFile()
- defiler(enfiler(F,x)) = enfiler(defiler(F), x)

### 3.5.3 Implémentations

On peut utiliser un tableau et un entier (tete file)
- defiler consistera à décrémenter teteFile
- creerFile on met teteFile à -1
- on enfile toujours à le case 0 (oblige de decaler tout le monde vers la droite, implémentation couteuse pour l'enfilement

On peut utiliser un tableau et voir comme un cercle. Pour cela, on utilise 2 entiers (représentant les indices) : queueFile et teteFile
il faudra considérer les indices Aucune opérations n'est couteuse (juste quelques incrémentation et decrementation)
On peut avoir du mal à distinguer une file vide d'une file pleine (on pourrait tester queueFile == teteFile, mais il faut ajouter une distinction). Pour cela, on va toujours laisser une case vide entre teteQueue et teteFile.

On pourrait utiliser 2 piles pour implementer une file...

### 3.5.4 Applications des files

- Parcours en largeur d'un arbre(en ordre)
- Tri topologique d'un ordre

## 3.6 TDA Liste

Créateur des listes : par le créateur du langage IPL (langange concu pour des programmes en IA) et est définie comme le type de base.
Elles sont utilisées pour la gestion de l'espace libre en mémoire, l'implémentation des polynômes...

### 3.6.1 Definition

l'idée générale c'est au lieu de stocker les objets de manière contigue (comme avec les tableaux), on les enchaine, chaque élément stocké contient un lien vers l'élément suivant.

Avantage :
- suppresion des éléments
- gestion de l'espace facilitée (pas besoi n de stocker dans un esapce contigue)
- l'union de 2 listes est constante

Inconvénient :
- pas d'acces direct aux éléments

### 3.6.2 Opérations

Le tda Liste_T utilise les types bool, T, int, et Cellule_T.
le type Cellule_T est un enregistrement à 2 champs : l'information à stocker et une référence vers un objet de type cellule_T
Les opérations sont les suivantes :
- valeurCellule : Cellule_T $\rightarrow$ T

- suivantCellule : Cellule_T → Cellule_T
- insererSuivantCellule : Cellule_T * Cellule_T → Cellule_T
- dernierCellule : Cellule_T → bool
- teteListe : Liste_T → Cellule_T
- creerListe : () → Liste_T
- estVideListe : Liste_T → bool
- insererTeteListe : Liste_T * T → Liste_T
- supprimmerTeteListe : Liste_T → LIste_T
- tailleListe : Liste_T → int
- obtenirIemeElement : Liste_T * int → cellule_T
- insererIemePosition : Liste_T * int * T → Liste_T
- supprimmerIemeElement : Liste_T * int → Liste_T
- queue : Liste_T → Liste_T

Une queue est une sous-liste privée de sa tête.

On peut trouver les axiomes dans ce livre "Christine Froideveaux, Marie claude Gaudel, Michele Soria - Types de données et algorithmes - INRIA, 3e édition, 1993"

### 3.6.3 Implémentation

### 3.6.4 Implémentation 1

tableaux statiques suivant logique = suivant physique, on a un tableau et le suivant d'une cellule c'est l'indice suivant.

```
struct Liste {
  T tab[MAX];
  int LD, LF;
}
```

```
creerListe() {
  LD = LF = 0;
}
```

```
insererTeteListe (liste l, T e) {
  if (l.LF < MAX-1) {
    int i = l-LF+1;
    while (i>=1) {
      l.tab[i] = l.tab[i-1];
    }
    l.tab[++l.LF] = e;
  }
  else {
    ERROR(liste pleine)
  }
}
```

```
supprimerTeteListe(liste l) {
  int i;
  for (i=0;i<l.LF; i++) {
    l.tab[i] = l.tab[i+r]
  }
}
```

### 3.6.5 Implémentation 2

Tableaux statiques suivant une logique différente de celle du suivant physique.

```
struct Cellule_T {
  T info;
  int suiv;
};
```

```
struct liste {
  cellule_T tab[MAX];
  int debut;
};
```

Le probleme dans cette implémentation c'est qu'il faut à chaque moment savoir où se situe l'ensemble des cases libres. Pour cela, on modifie l'enregistrement liste en ajoutant une structure de données qui va stocker l'ensemble des cases libres.

```
struct liste {
  cellule_T tab[MAX];
  int debut;
  Pile_int P;
}
```

A la création d'une cellule, on met toutes les cases dans la Pile. Lorsque l'on insère 1 élément, on regarde dans la Pile s'il existe une case libre et on la récupère, on dépile et on éxécute l'insertion

exemple : pour insererDebut :

- récupèrer case vide
- stocker l'information dans la cellule
- poser cellule.suivant = liste.debut
- poser liste.debut = indice cellule

Lorsque l'on supprime 1 élément, on prend l'indice et on le met dans la liste des cases libres (après avoir bien sur refait le chainage).

### 3.6.6 Implémentation 3

Nous utiliserons des pointeurs(variable ou l'on stocke des adresses).

Ici, pour déclarer une variable pointeur on fait comme en c : *type * identifiant ;*
Pour demander au systeme de l'espace libre on écrit : *pointeur = new type ;*
pour rendre l'espace alloué : *free(pointeur) ;*
La fonction free est indispensable pour ne pas se trouver sans espace libre (sauf si le systeme à un garbage collector).

Lorsque l'on implémente une liste en utilisant les pointeurs, comme pour l'implémentation avec les tableaux, demander au système l'allocation de mémoire et surtout penser tout le temps à libérer l'espace non utilisé.

```
struct cellule {
  T info;
  cellule * suiv;
};
```

```
//la liste sera un pointeur sur une cellule:
creerListe() {
  cellule * l;
  l = NULL; // pointeur qui ne pointe sur rien
  return l;
}
```

```
insererDebut(cellule * l, T e) {
  cellule * p = new cellule;
  p.info = e;
  p.suiv = l;
  l = p;
  return l;
}
```

```
supprimerDebut(cellule * l) {
  if (l != NULL) {
    cellule * p = l;
    l = l.suiv;
    free(p);
  }
}
```

```
estVideListe (cellule * l) {
  return (l==NULL);
}
```

#### 3.6.6.1 Extensions des listes

Liste bidirectionnelle où chaque cellule de la liste pointe sur la suivante et sur la précédente cellule (double chainage). Un intéret : lors de certaines opérations, comme la suppression, on n'a pas besoin de stocker temporairement un lien vers la cellule précédente.

Liste circulaire où la dernière cellule pointe sur la premiere (elle est vue comme un cercle). On peut commencer depuis n'importe quelle cellule et parcourir toute la liste.

##### 3.6.6.1.1 Comment savoir que l'on a tout parcouru ?

(et que l'on n'est pas revenu au début)

Une manière naive : stocker la première cellule de début de parcours et la prochaine fois qu'on le rencontre (dans le parcours) on saura que l'on a tout parcouru. Le probleme c'est lorsque l'on fait des suppressions en parcourant on peut supprimer la cellule stockée et donc ne plus revenir dessus. Le seul moyen : c'est de créér une cellule qui ne stocke jamais d'information et qui sera considéré comme le début et qui ne sera jamais supprimmées (physiquement la liste ne sera jamais vide)

### 3.6.7 Gestion de l'espace libre

gestion de l'espace mémoire libre, on utilise une liste, nommées LIbre et qui contient toutes les cellules libres de la mémoire. Lorsque un utilisateur appelle l'instruction new, vous executez un code de ce type :

```
if (estVideListe(Libre)) {
  then Overflow
}
else {
  x = teteListe(Libre);
  libre = supprimerTeteListe(Libre);
}
```

Lorsque l'on apelle free :

```
libre = insererSuivantCellule(x, Libre);
```

(ici on gère notre liste comme une pile)

Comment construire la liste libre ?

on peut chainer toute les cases libres au début, mais ça peut prendre du temps... et en général nous n'avons pas besoin de tout l'espace libre. Une technique consiste à regarder la mémoire comme un tableau. On a une adresse MaxLibre qui délimite les cellules libres et une adresse tabMin qui délimite le début des tableaux dynamiques.

En exercice vous donnerez le code pour gérer l'espace libre (sans construire le liste libre au début).

### 3.6.8 Applications

Gestion des polynomes, dans la feuille 2 vous utiliserez des tableaux pour gérer des polynomes. Mais ceci n'est pas efficace lorsque le polynôme est peu dense, (ie) peu de coefficients non nuls.

avec des tableaux, on stocke tous les coefficients nuls ou pas (ceci n'est pas efficace).

on peut utiliser des listes : on stocke les monômes non nuls. ex : $x^{1000} - 2$ : 2 monôme non-nuls 1 monôme = coefficient, degré

```
struct monome {
  float coeff;
  int degre;
};
```

Un polynome sera une liste de monôme.

# Chapitre 4

# Eléments de complexité

## 4.1 Critères d'évaluation

Pour un problème donné, un algorithme valide est un algorithme qui le résout. Formellement, une spécification pour un algorithme, ce sont les types d'entrée et le traitement à effectuer.

Un algorithme valide est un algorithme qui pour toute instance (vérifiant les conditions d'entrées), fournit le résultat escompté.

Comme les ressources sont limités, la validité n'est pas le seul critère d'évaluation. On voudrait aussi mesurer la qualité. La qualité d'un algorithme se mesure suivant plusieurs critères. On peut citer :
- La lisibilité
- La granularité
- L'efficacité mesurée par rapport aux ressources à notre disposition.
  Les ressources disponibles sont la mémoire et les capacités de calcul.

On mesure l'efficacité en calculant l'espace utilisé (appelé complexité en espace) et le temps de calcul(appelé complexité en temps). Complexité en espace : nombre de bits utilisés.

## 4.2 Evaluation du temps d'exécution

Pour mesurer la complexité en temps, on peut calculer le temps d'execution (après implémentation). L'inconvénient est que la rapidité va dépendre du processeur. Si le processeur n'est pas assez rapide, on le change et d'après la loi de Moore les capacités de calculs doublent tous les 18-24 mois, malheureusement ceci n'est pas suffisant.

Il existe des algorithmes rapides sur des petites instances, et qui croissent très rapidement.

Mesurer le temps de calcul n'est alors pas suffisant. La seule constante dont on dispose c'est que toutes les machines ont un nombre constant d'instructions, et la différence c'est le temps d'execution d'une instruction. On peut alors mesurer le nombre d'instructions pour calculer la complexité en temps. Pour avoir le temps d'execution dans uen machine x, il suffit de connaitre le temps pour une instruction.

## 4.3 Notation O, omega, theta

si f :$\mathbb{R} \to \mathbb{R}$ et g :$\mathbb{R} \to \mathbb{R}$ sont deux fonctions.

$g(n) = O(f(n))$ si $\exists c, n_0 | \forall n \geq n_0, g(n) \leq c.f(n)$

$g(n) = \Omega(f(n))$ si $\exists c, n_0 | \forall n \geq n_0, g(n) \geq c.f(n)$

$g(n) = \Theta(f(n))$ si $\exists c_1, c_2, n_0 | \forall n \geq n_0, c_1.f(n) \leq g(n) \leq c_2.f(n)$

### 4.3.1 Exemples

$3n^2 + 10n = O(n^2)$
$5n + 100000000000000 = O(n)$
$(1/2)n^2 - 3n = O(n^2)$

**constante** : $O(1)$

**logarithmique** : $O(log(n)$

**Linéaire** : $O(n)$

**Quadratique** : $O(n^2)$

**Polynomiale de degré k** : $O(n^k)$

$O(a^n), O(n!), O(n^n)$ sont des fonctions exponentielles. (ex : tout algorithme générant tous les sous-ensemble d'un ensemble de tout n nécéssitera au moins $2^n$ instructions).

On notera également les fonctions étalon O(n.log(n)) (tri d'un tableau) et O(log(n)) (recherche dans un tableau trié).

Remarque : l'arithmétique avec la notation O est simple. on prendra tuojours comme résultat le terme dominant.

#### 4.3.1.0.2 exemple

$O(n) + O(1) = O(n)$
$O(n^2) + O(log(n)) = O(n^2)$

## 4.4 Règles de calcul

On va compter le nombre d'instructions basiques de notre langage car chacune d'elle peut être implémentée avec un nombre constant d'instructions machines.

Etudier la complexité n'est intéressant que pour les entrés de grande taille, on ne s'interessera qu'aux comportements asymptotiques (la limite lorsque la taille des instance tend vers l'infini).

Du coup la notation O s'impose et devient naturelle.

On va noter C(A) la complexité en temps d'un algorithme A. On a :

$C(A) = \sum C(I)$
I : instruction de A
C(I) = nombre d'instructions basiques de l'instruction I.
On montre comment calculer C(I) inductivement :

Les déclencheurs de types, de variables (sans affectation), lectures de variables et de valeurs on dira qu'elle necessitent O(1) instructions.

Les expressions arithmétiques c'est borné par le nombre d'instructions pour lire les valeurs, du coup une expression arithmétique où tous les opérandes sont soit des constantes, soit des variables nécessite O(1).

Si l'instruction I est une affectation x :=exp, alors C(I)=C(exp).

Si l'instruction I est de la forme $f(\exp_1, \exp_2, ..., \exp_n)$ l'appel de la fonction f alors :

$C(I) = \sum_{i=1 \to n} C(\exp_i + C(f)$

Si l'instruction I est une boucle avec n itérations et à chaque itération on exécute l'expression $\exp_i$, alors

$C(I) = \sum_{i=1 \to n} C(\exp_i$

Si l'instruction I est une structure conditionnelle utilisant les expressions booléennes $C_1, C_2, ..., C_n$ et les expressions $\exp_1, \exp_2, ..., \exp_n$ .

### 4.4.1 Exemple

```
1 int sommeN(int n) {
2   int s := 0;
3   for (i = 1; i <= n; i++) {
4     s += i;
5   }
6   return
7 }
```

ligne 2 : déclaration de s et affection de 0 à s : O(1)
ligne 3 : n*O(1) = O(n)
ligne 4 : O(1)
algorithme : O(1)+O(1)+O(n)= O(n)

La complexité de sommeN dépend de l'entrée alors que ce n'est pas le cas pour sommeNB. Pour spécifier que la complexité peut dépendre des entrées, on va utiliser une notation fonctionnelle. Si un algorithme A prend n entrées, on écrit $C_A(t_1, t_2, ..., t_n)$ avec $t_i$ la taille de l'entrée i.

Pour les fonctions sommeN on écrira : $C_{\text{sommeN}}(n) = O(n)$

## 4.5 P vs NP

On définit les ensemble P et NP que l'on appelles "classes de complexité".

**4.5.0.0.3 classification :** un problème est dit polynomial (ou dans P) s'il existe un algorithme qui le résout et qui s'execute temps polynomial (par rapport aux entrée). On peut citer le tri d'un tableau, recherche de plus court chemin...

Un problème est dans NP s'il existe un algorithme polynomial qui prend en entrée des solutions possibles et décide se l'entrée est une solution au problème ou pas. On les appelle des problèmes à vérification polynomiale. (NP -> machine de Turing non-déterministe)

**4.5.0.0.4 exemple**
Recherche d'un chemin qui parcourt chaque ville exactement une fois.
Remplir un sac à dos de manière optimale.

### 4.5.1 NP-Complet

Parmis les problèmes dans NP, il existe ceux que l'on apelle NP-COMPLET. Un problème P est NP-Complet si pour tout problème P, il existe un algorithme polynomial A et qui pour toute entrée I‘ de P construit une entrée I de P telle que I a une solution pour P ssi I‘ a une solution pour P‘.
NP-Complet : sorte de méta-problème qui permet d'affirmer que tout probleme P inclu dans NP inclu dans NP-Complet.

## 4.6 Remarque

Prenons sommeN et mesurer la complexité en la taille de n.

Si on prend la notation unaire : (avec les batonets) :
$C_{sommeN}(n) = O(n)$
Si on prend la notaiton binaire :
$C_{sommeN}(k) = O(2^k)$

Certaines règles à respecter pour l'encodage sont les suivantes :

Les entiers ne sont pas représentés en unaires(1 = 1, 2 = 11, 3 = 111, 4 = 1111), mais en base b$\geq$2

Les instances sont codées de façon raisonnable et concise (par exemple un ensemble avec n-élément necessitera n*(taille d'un élément).

Avec ces contraintes, on à la proposition suivante :

### 4.6.1 Proposition

L'encodage ne change pas la classe de complexité d'un problème.

Avec les machines à coeurs multiples, aux clusters à haute capacité de calcul, la classification entre P et NP ne change pas, par contre se pose le problème de la parrallélisation des calculs.

Face aux problèmes NP-Complets (rencontrés tous les jours), plusieurs techniques ont été élaborées :
- (1) restreindre les instances à des instances particulières
- (2) au lieu de donner une solution exacte, on donne une solution approchée (algo d'approximation)
- (3) on introduit de l'aléatoire et on contruit des algorithmes randomisés (simplex)

- (4) on propose des heuristiques (qui marchent souvent) ex : collecte des déchets ménagers
- (5) résoudre de manière exacte (et donc on obtient un algorithme exponentiel pour les problèmes NP-Complets), mais on essaie de diminuer drastiquement la base de l'exponentiel et/ou l'exposant.

**4.6.1.0.5 exemple** : Avoir $O((\sqrt{2})^n)$ au lieu de $O(2^n)$
ou, $O(2^{\sqrt{n}})$ au lieu de $O(2^n)$

# Chapitre 5

# Types inductifs

def : Une définition inductive d'un ensemble E consiste en la donnée
– d'un ensemble fini B $\subseteq$ E
– d'un ensemble K d'opérations $\emptyset : E^{ar(\Theta)->E}$ où $ar(\Theta)$ est l'arité de $\Theta$
En fait E est le plus petit ensemble tel que :
– $B \subseteq E$
– $\forall \Theta \subseteq K, \forall (x_1, x_2, ..., x_n) \subseteq E, \Theta(x_1, x_2, ..., x_n) \subseteq E$
Tout ensemble inductif sera défini par la donnée de B et de K

## 5.1 Exemple 5.1

si on prend $B = 0$ et on pose l'opération $suc \rightarrow n+1, \forall n \subseteq |N$ Alors on peut prouver que c'est une définition inductive des entiers. Tout entier n correspond au nombre de fois que l'on a appliqué suc sur 0.
1 = suc(0)
2 = suc(suc(0))

## 5.2 Exemple 5.2

Soit $A = a, b$ et $A^x$l'ensemble des mots finis sur A. ex : ab, aaa, abaa,...
On pose $\Theta_a : u \rightarrow ua$ et $\Theta_b : u \rightarrow ub$
$\forall u \subseteq A^x$
ex : $\Theta_a(ab) = aba$
$\Theta_b(ab) = abb$
en posant $B = \epsilon$ et $K = \Theta_a, \Theta_b$
où $\epsilon$ est le mot vide.
On peut vérifier que $A^x$ est défini inductivement par B et K.

## 5.3 Exemple 5.3

Soit $A = (,)$.
Soit $D \subseteq A^x$ défini inductivement par $\epsilon \subseteq D$, et les deux opérations
$\Theta_1 : u \to (x)$ et
$\Theta_2 : (x, y) \to xy$
Ex :
$\Theta_1(\epsilon) = () \subseteq D$
$\Theta_1(\Theta_1(\epsilon)) = (()) \subseteq D$
$\Theta_2(\Theta_1(\epsilon), \Theta_1(\epsilon)) = ()() \subseteq D$
L'ensemble des mots de Dyck = D (l'ensemble des mots bien parenthésés)

## 5.4 Exemple 5.4

Soit $A = 0, 1, (,)$
On note $AB \subseteq A^x$ l'ensemble défini inductivement
$\epsilon \subseteq AB$ $\Theta_0 : (g, d) -> (0, g, d)$ $\Theta_1 : (g, d) -> (1, g, d)$
est appelé l'ensemble des arbres binaires avec les noeuds étiquetés par 0 ou 1.

### 5.4.1 ex

$(0, \epsilon, \epsilon), (0, (1, \epsilon, \epsilon), (0, \epsilon, \epsilon)))$

## 5.5 oO

Une définition inductive est utile pour prouver des propriétés. Supposons que l'on dispose d'une définition inductive (B,K) d'un ensemble E et que l'on veuille montrer que les éléments de E vérifient une propriété P. Il suffit de montrer :

- Tout élément de B vérifie la propriété (en général c'est facile)
- $\forall \Theta \subseteq K$ et tout $(x_1, x_2, ..., x_{ar(\Theta)})$ Vérifiant la propriété P, alors $\Theta(x_1, x_2, ..., x_{ar(\Theta)})$ vérifie P.

Avir une définition inductive est aussi très pratique pour calculer des fonctions sur des éléments -> des fonctions récursives

### 5.5.1 Ex

la taille d'un mot c'est le nombre de lettres. On peut en donner une définition récursive :
$taille(\epsilon) = 0$
$taille(\Theta_a(u)) = 1 + taille(u)$
$taille(\Theta_b(u)) = 1 + taille(u)$

## 5.6 Exemple 5.5, TDA liste

On rappelle qu'une liste chaines c'est un ensemble d'élément chaines les uns à la suite des autres. Une façon indique de définir une liste chainée (C'est d'ailleur la définition donnée en Ocaml ou en Lisp ou les implémentation avec les cellules) est de dire qu'elle est soit vide (notée [] en Ocaml ou NULL en c) soit une paire (a, xs) où xs est une liste et a un élément.

## 5.7 Arborescences

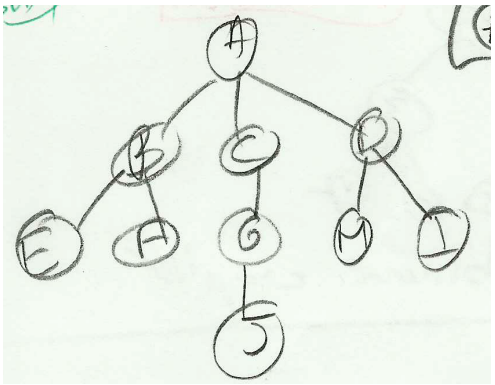
Une arborescence c'est un ensemble V muni d'un sommet distingué r, appelé racine, et d'une relation binaire E telle que $\forall x \in V$, excepté la racine, $\exists$ un unique $y \in V$ et noté $p(x)$, tel que $(y, x) \in E$. La définition inductive est la suivante (on notera une arborescence $(r, V, E)$)

- tout singleton de V est une arborescence
- Si $T_1, T_2, ..., T_p$ sont des arborescences, ave $T_i = (r_i, V_i, E_i)$, alors $(r, V, E)$ est une arborescence où $V = \{r\} \cup \bigcup_{1 \leq i \leq p} V_i$ et $E = \{(r, r_i) | 1 \leq i \leq pV_i$ A COMPLETER

Soit $T = (r, V, E)$ une arborescence. Pour tout noeud x, les noeuds dans l'ensemble $\{y \in V | (y, x) \in E\}$ sont appelés les fils de x et x est appellé leur père. Les noeuds sans fils sont appelés feuilles. Un chemin de taille k est une suite finie $(x_1, x_2, ..., x_n)$ telle que pour tout $1 \leq i \leq k$ , $(x_i, x_{i+1}) \in E$

Pour x un noeud, on note $T_x$ la sous-arborescence de T enracinée en c et constituée des noeuds accessibles depuis x par 1 chemin.

La hauteur de T, notée h(T) c'est la longueur maximale d'un chemin depuis la racine.




les fils de A = $\{B, C, D\}$
(A,C,G) est un chemin
J,E,F,H,I sont les feuilles.

$h(T) = 4$ (le chemin de longueur maximal c'est (A,C,G,J))

### 5.7.1 Proposition 5.1

Soit $T = (r, V, E)$
- Pour tout noeud x, il existe un unique chemin de r à x
- Une arborescence contient au moins une feuille
- $h(T) = 1 + max\{h(\text{sousArbreG}(T)), h(\text{sousArbreD}(T))\}$
- $|E| = |V| - 1$

### 5.7.2 Implémentation

Pour une arborescence où les noeuds sont numérotés de 0 à n-1, on peut le représenter par un tableau PERE de taille n, dit tableau de Prifer, où PERE[i] contient p(i).

(Pour la racine on met -1)

Cette représentation permet de montrer qu'il existe exactement $n^{n-1}$ arborescences avec n noeuds

Cette représentation est interessante lorsque l'on connait à l'avance le nombre de noeuds. Lorsque l'on ne connait pas la taille où lorsqu'elle évolue dans le temps, on peut préférer une représentation avec des listes. On peut par exmeple définir un type $cellule_T$, qui représentera un noeud, :

```
1 struct cellule_T = {
2   T info;
3   cellule_T pere;
4   cellule_T filsG;
5   cellule_T filsD;
6   cellule_T frereD;
7 };
```

Une arborescence sera jsute un lien vers la racine. L'espace utilisé est $O(n *$ (taille info))

Dans la représentation avec des listes, il existe plusieurs autres implémentations. On peut par exemple ne garder que la référence vers le père (la racine sera le noeud sans père) et garder en plus la liste des noeuds.

L'implémentation dépendra de l'application.

Arbre est une arborescence où les fils sont ordonnés (on a un premiere fils, deuxieme fils,...). Dans l'implémentation d'un arbre, on doit pouvoir identifier les fils. Par exemple dans l'implémentation ci dessus, filsG devrait pointer sur le premier fils et si x est le $i^eme$ fils, alors x.frereD devrait pointer sur le (i+1)ème fils.

### 5.7.3 Arbre binaire

Un arbre binaire est un arbre où chaque noeud a au plus 2 fils.

Il est appelé arbre binaire *complet* si chaque noeud à exactement 0 ou 2 fils.

Il est *parfait* si toutes les feuilles sont ou même niveau.

Ou *quasi-parfait* si tous les niveaux, excepté le dernier, sont remplis, et le dernier est rempli en partant de la gauche.

Dans un arbre binaire on a les primitives suivantes :

- racine : Arbre_T -> cellule_T
- arbreG : Arbre_T -> Arbre_T
- arbreD : Arbre_T -> Arbre_T

### 5.7.4 Parcours d'un arbre

Le parcours d'un arbre consiste à visiter tous les noeuds exactement une fois (dans un certain ordre).

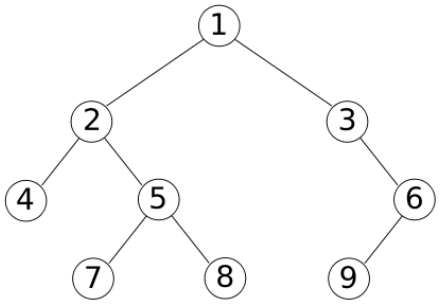
#### 5.7.4.1 Parcours en profondeur

visiter la racine, ensuite chaque sous-arbre issu de ses fils (dans l'ordre).

#### 5.7.4.2 Parcours en largeur

visiter la racine, ensuite tous ses fils, et ainsi de suite (on visite niveau par niveau).

#### 5.7.4.3 Exemple




préfixe : 1, 2, 4, 5, 7, 8, 3, 6, 9
suffixe : 4, 7, 8, 5, 2, 9, 6, 3, 1
infixe : 4, 2, 7, 5, 8, 1, 3, 9, 6

#### 5.7.4.4 Préfixe

Le mot préfixe d'un arbre est une permutation des noeuds telle que le noeud x précéde le noeud y si x ancêtre de y ou il existe z, x‘,y‘ tels que
- x‘ ancêtre de x
- y‘ ancêtre de y
- z père de x‘, y‘ et x‘ $\leq$ y‘

Le parcours en profondeur donne un mot préfixe (en lisant l'information la première fois que l'on rencontre le noeud)

#### 5.7.4.5 Suffixe

Le mot suffixe est une permutation des noeuds telle que le noeud x précéde le noeud y si y ancêtre de x ou il existe z, x‘, y‘, tels que
- x‘ ancêtre de x
- y‘ ancêtre de y
- z père de x‘, y‘ et x‘ $\leq$ y‘

Le parcours en profondeur donne le mot suffixe (on lit l'information après avoir parcours les sous-arbres issus de x)

#### 5.7.4.6 Infixe

Le mot infixe d'un arbre binaire est une permutation où tout noeud x est entre ses arbres gauche et droit

### 5.7.5 Proposition 5.2

Le mot préfixe tout seul ne suffit pas pour reconstruire l'arbre. De même que le mot suffixe. Cependant le coupls (mot préfixe, mot suffixe) est suffisant.

### 5.7.6 Applications

- Système de fichier
- Analyse de complexité d'un algorithme
- Décider de la valeur de vérité d'une expression booléenne (on utilise pour cela l'arbre des possibilités) (TODO FIG 10)

## 5.8 Tas binaires

Un tas binaire est un arbre binaire *quasi-parfait*. Ils sont utilisés pour répondre à des requêtes du genre "obtenir le minimum", "obtenir le maximum".

Un tas-min est un tas binaire où info(père(x)) $\leq$ info(x)

Un tas-max est un tas binaire où info(x) $\leq$ info(pere(x))

Le minimum (maximum) se trouve à la racine dans un tas min (tas-max), les primitives d'un tas-max :

- Construire-tas
- Insérer
- entasser-maximal
- tri_tas
- maximum
- retirer-element
- remplacer-clé

La structure de tas suppose l'existence d'un ordre linéaire dans les objets stockés.
TODO algo1, algo2

```
1  entasser-max (tas t, cellule r)
2    fg = filsG(r);
3    fd = filsD(r);
4    max = r;
5    if ((fg != NULL) && (valeur(racine(fg)) >=  valeur (r))
         ) {
6      max = racine(fg);$
7    }
8    if ((fg != NULL) && (valeur(racine(fd)) >= valeur (max)
         )) {
9      max = racine(fd);
10   }
11   if (max != r) {
12     echanger(max, r);
13     entasser(t, max);
14   }
15   return t;
16 }
```

```
1  T[] tri-tas (T tab[]) {
2    tas t = constuire-tas-tab);
3    for(i = n-1; i>=0; --i) {
4      echanger(tab[i], maximum(t));
5      retirer-element(t, maximum(t));
6    }
7    return tab;
8  }
```

A faire : montrer que la complexité c'est O(n * log(n))

#### 5.8.0.1 Application - Files de priorités

Une file de priorité est une structure de données qui permet de gérer un ensemble S où chacun à une valeur appellée (niveau de priorité ou clé) qui

permet de gérer les priorités. Les primitives sont les suivantes :
- inserer
- maximum
- extraire-max
- remplacer-cle

Un tableau n'est pas adapté (toutes les opérations excepté obtenir le maximum coùtent O(n).

En utilisant 1 tas-max :
- insérer : O(log(n))
- maximum : O(1)
- extraire-max : O(log(n))
- remplacer-cle : O(log(n))

## 5.9 Arbres de recherche

Un arbre de recherche est une structure de données supportant un ensemble d'opérations dynamiques les plus importantes sont

- rechercher
- minimum
- maximum
- predecesseur
- successeur
- inserer
- supprimer

### 5.9.1 Arbre binaire de recherche (ABR)

C'est un arbre binaire où pour chaque noeud x on a la propriété suivante : si $x_1$ et $x_2$ sont les fils de x, alors $\forall y \in T_{x_1}$, $\forall z \in T_{x_2}$

$$info(y) \leq info(x) \leq info(x)$$

On va noter h la hauteur de l'ABR.
minimum : le minimum se trouve dans le noeud le plus à gauche (O(h))
maximum : le maximum se trouve dans le noeud le plus à droite (O(h))
predecesseur : c'est le maximum du sous-arbre (O(h))
successeur : minimum du sous-arbre droit (O(h))

#### 5.9.1.1 Rechercher

comparer avec l'information stockée à la racine.
si valeurs égales, n s'arrête

sinon si inférieur => rechercher dans le sous-arbre gauche
sinon recherche dans le sous-arbre droit

c'est du O(h) : on regarde niveau par niveau et chaque niveau on fait un nombre constant d'opérations.

#### 5.9.1.2 Inserer

Faire comme une recherche et l'insérer comme fils de la feille où on s'arrête (O(h))

#### 5.9.1.3 Supprimer

Faire une recherche du noeud à supprimer et réorganiser (trouver un père à ses sous-arbres) => O(h)

#### 5.9.1.4 Remarque

Toutes les opérations se font en temps linéaire sur la hauteur

Si on s'arrange pour qu'a chaque fois l'ABR ait une hauteur logarithmique, on a gagné (on ne peut pas espérer mieux).
Mais il existe des instances où la hauteur est linéaire en le nombre de noeuds.

#### 5.9.1.5 Théorème 5.2

La hauteur moyenne d'un arbre binaire de recherche construit aléatoirement avec n clés à une hauteur O(log(n)).

### 5.9.2 Arbre Rouge Noir (ARN)

C'est un arbre qui est toujours équilibré.

Un arbre rouge noir est une ABR si :
- (1)chaque noeud est soit rouge soir noir
- (2)la racine est noire
- (3)si le noeud est rouge, alors ses deux fils sont noirs
- (4)tous les chemins depuis n'importe quel noeud vers n'importe qu'elle feuille contient le même nombre de noeud noirs.

TODO F11

#### 5.9.2.1 Proposition 5.3

Un ARN ayant n noeud a une hauteur au plus 2 * (log(n) + 1)
Preuve : on note bh(x) le nombre de noeuds noirs depuis un noeud x vers une

feuille. On montre par récurrence que le nombre de noeuds $T_x \leq 2^{hb(x)} - 1$

Si h est la hauteur de l'ARN.

D'après la propriété (3), vous devez avoir au moins depuis la racine jusqu'a la 1ère feuille $\frac{h}{2}$ noeuds noirs (sur chaque tel chemin). Donc on a : $n \geq 2^{h/2} - 1$ => $n + 1 \geq 2^{h/2}$ => $log(n+1) \geq (h/2)$ => $h \leq 2 \times log(n+1)$

#### 5.9.2.2 primitives rotation gauche et droite

sont des fonctions qui prennent 1 ABR :
FIG100

La compléxité en temps des opérations c'est O(1) (que des copies de référence)

L'insertion dans un 1 ARN

Elle se fait en 2 étapes :

(i,1) insérer léléméent comme dans un ABR et le marquer rouge
(i,2) réorganiser en ARN (on peut violer la condition (3))

Notons 2 le noeud inséré et gp = pere[pere[z]]. On a 2 cas : symétriques : pere(z) = filsG(gp) ou pere(z) = filsD(gp)
supposons pere(z)=filsG(gp) et nnotons y = filsD(gp). on a alors 3 sous-cas :

**5.9.2.2.1 Cas 1.1** Y est rouge, alors gp est noir, on colorie y et pere(z) en noir et on colorie gp en rouge. ENSUITE ON RECOMMENCE EN POSANT Z = GP (C'EST pour vérifier que condition (4) reste vérifiée)

**5.9.2.2.2 Cas 1.2** y est noir, si z = filsD(pere(z)), alors on fait une 1ere rotation gauche de pere(z) et ensuite on pose z = pere(z)

**5.9.2.2.3 Cas 1.3** sinon z = filsG(pere(z)). alors on colorie pere(z) en noir, gp en rouge et on fait une rotation droite de gp.

Ala fin, on colorie la racine en noir. Si pere(z) = filsD(gp), alors on a les mêmes cas. 1.1 -> 1.3, mais on remplacer gauche par droite.

Il faudra montrer que ces opérations garantissent à la fin la propriété d'être un ARN.

FIG101

**5.9.2.2.4** Elle se fait en 2 étapes :

1. supprimer l'élément comme dans un ABR
2. réorganiser en ARN (sous certaines conditions)

Lorsque l'on supprime un noeud z, en fait "physiquement" le noeud z. N'est pas supprimé mais son successeur, si.

On notera y = suc(z)

Si y est rouge, alors il n'y a rien à réorganiser, toutes les propriétés d'un ARN sont vérifiées. Le problème c'est lorsque y est noir.

Notons x fils gauche de pere (y) et w le fils droit.

On a alors 2 cas : soit y = filsG(pere(y)), soit y = filsD(pere(y)).

Comme c'est 2 cas symétriques, on suppose x = filsG(pere(y))

**5.9.2.2.5 Cas 1** w est rouge. on permute les couleurs de w et pere(x), puis effectuer une rotation gauche sur pere(x). On peut remarquer que le nouveau frere de x, qui était un enfant de w est rouge.

**5.9.2.2.6 Cas 2** w est noir et ses 2 enfants sont noirs. On recolorie w en rouge et on pose x = pere(x)

**5.9.2.2.7 Cas 3** w est noir et l'enfant gauche est rouge, on permute les coueleurs de w et de son enfant gauche, ensuite on fait une rotation droite sur w.

**5.9.2.2.8 Cas 4** w est noir, enfant gauche noir, enfant droit rouge.

On permute les couleurs de w et de son enfant droit et on fait une rotation gauche sur pere(x).

On itère tant que x $\neq$ racine et sa couleur est noire.

Si y = filsD(pere(y)), même cas (remplacer droite par gauche).

FIG102

**5.9.2.2.9 Application** Gestion de partitions (sous ensembles qui ne s'intersectes pas)

**5.9.2.2.10 Gestion de partition** On rapelle que une partition de E c'est une collection de sous-ensembles $E_1, ..., E_n$ de E tq :
- $\bigcup_{1 \leq i \leq n} E_i = E$
- $\forall i \neq j$, $E_i \cap E_j = \emptyset$

Pour gérer une partition on peut utiliser le TDA ensembleDisjoint.

Le TDA ensembleDisjoint_T utilise le type T et a comme primitives :

creerEnsemble : T -> ensembleDisjoint_T
union : T x T x ensembleDisjoint_T -> T (l'union des 2 sous-ensembles contenant les 2 arguments)
trouver : ensembleDisjoint_T x T -> T (trouver le représentant du sous-ensemble contenant l'argument)
On peut implémenter ensembleDisjoint_T avec une matrice M (nxn), |T|=n

M[i,j] = 1 si i et j sont dans le même sous-ensemble.

On peut aussi implémenter avec des listes : Pour $E_1, ..., E_p$, on a p listes et chaque liste i contient les éléments de $E_i$ . (pas efficace)