Types Abstraits & Bases de la POO
III - Types de Données Abstraits

- - - - - - - - - -

- - - - - - - - - -

ENSMA A3-S5 - période A

2023-2024

M. Richard
richardm@ensma.fr

Sommaire

I - Définition

1 – Définition formelle. . .
2 – Principe d'implémentation

Un type de données abstrait (TDA) ou Abstract Data Type (ADT), est défini par un ensemble d'éléments et d'opérations manipulant ces éléments.

- la définition d'un TDA comporte trois grandes parties :
  - la définition des *signatures* des opérations
  - les axiomes définissant le comportement des méthodes
  - les pré-conditions dans le cas d'opérations partiellement définies

- la définition d'un TDA est très formelle et :
  - ne dit généralement rien concernant l'ensemble d'élément si ce n'est son nom,
  - fait totalement abstraction du type des éléments contenus dans cet ensemble (principe de généricité),
  - ne dit généralement rien sur la taille de l'ensemble (structure dynamique),
  - à contrario, les opérations sont elles définies précisément à l'aide d'une *spécification algébrique*.

- On distingue 2 grandes familles de TDA :
  - les *TDA linéaires :*
    - Tableau, Pile, File Liste, Deque, Set, . . .
  - les *TDA hiérarchiques :*
    - Arbre et graphe principalement

↪ Intérêt d'un TDA

- Vous l'aurez compris l'intérêt d'un TDA en général est de formaliser fortement l'écriture lors de la phase de conception afin de pouvoir écrire et vérifier un certain nombre de propriétés

↪ Implémentation d'un TDA

- bien évidemment, il existe *différentes implémentations pour un même TDA* dépendant le plus souvent du langage d'implémentation.
- l'implémentation d'un TDA est le plus souvent dynamique
- de plus, la construction de la structure de données est au maximum cachée à l'utilisateur du TDA.
- l'implémentation des opérations définies par le TDA doit transformer la notation fonctionnelle en méthodes, en respectant les grandes règles ci-dessous. Soit un TDA T :
  - opérations de construction : T est résultat sans être paramètre.
  - opérations d'accès : T est paramètre sans être résultat.
  - opérations de transformation : T est paramètre et résultat, i.e. mode modifiable.
- La majorité des langages proposent l'implémentation de différents TDA.
  - L'objectif n'est pas ici de les utiliser, mais de les réaliser ! !

II - Types abstraits de données linéaires

1 – Tableau
2 – Pile
3 – File
4 – Deque
5 – Vecteur
6 – Liste

↪ Définition du TDA tableau $Tab(T, n)$

Voyons par exemple comment se défini le TDA *Tableau* :

Opérations :

- Create : $(t_1, ..., t_n) \rightarrow Tableau(T, n)$
- Get : $Tableau(T, n) \times Entier \rightarrow T$
- Set : $Tableau(T, n) \times Entier \times T \rightarrow Tableau(T, n)$

Préconditions :

- Get(t, i) défini seulement si $0 \leq i < n$ avec $t \in Tableau(T, n)$.

Axiomes : $\forall x, y \in T$, $i, j \in Entier$, et $t \in Tab(n, T)$

- si $i \neq j$ alors $\text{Set}(\text{Set}(t, i, x), j, y) = \text{Set}(\text{Set}(t, j, y), i, x)$;
- si $i = j$ alors $\text{Set}(\text{Set}(t, i, x), i, y) = \text{Set}(t, i, y)$;
- $\text{Get}(\text{Create}(t_1, ..., t_n), i) = t_i$;
- $\text{Get}(\text{Set}(t, i, x), i) = x$.

↪ Définition du TDA Pile

Une pile définit une collection d'éléments liés et utilise la politique LIFO (Last In First Out) pour la manipulation des éléments. Plus précisément, le dernier élément ajouté à la collection sera le premier à être supprimé.

Opérations :

- `PileVide` : $\{\} \rightarrow Pile(T)$
- `EstVide` : $Pile(T) \rightarrow Boolean$
- `Empiler` : $T \times Pile(T) \rightarrow Pile(T)$
- `Depiler` : $Pile(T) \rightarrow T \times Pile(T)$

Préconditions :

- `Depiler(p)` défini si et seulement si `!EstVide(p)`

Axiomes :

- `EstVide(PileVide(p)) = VRAI`
- `EstVide(Empiler(e,p)) = FAUX`
- `Depiler(Empiler(,e,p)) = (e,p)`

↪ Implémentation du TDA Pile 1/2

- Structure de données généralement dynamique
- on peut distinguer deux grandes constructions :
- *suite d'éléments liés :* dans ce cas, une pile est définie par une référence (i.e. pointeur) sur une cellule contenant :
  - un élément, i.e. la donnée
  - une référence, i.e. pointeur, sur elle même permettant la liaison avec la cellule suivante.
- *sentinelle et suite d'éléments liés :* contenant les mêmes champs que décrits précédemment. La sentinelle permet de stocker à la fois la référence du sommet de la pile, mais également d'autres informations permettant de faciliter la mise en œuvre des opérations (par exemple la taille, ...).
  - exemple :

```
1           typedef struct t_pile *t_pile;
2           typedef struct cell{
3           int data;
4           struct cell *suiv;
5           } t_cell;
6           typedef struct t_pile{
7           int nbelem;
8           t_cell *sommet;
9           } * t_pile;
```

↪ Implémentation du TDA Pile 2/2

- Opérations sur une pile (stack) :
  - en français : `empiler, depiler, sommet, pilevide`
  - en anglais : `push, pop, top, empty`

```
1       ...
2       t_pile init();
3       void push(t_pile p, int val);
4       int pop(t_pile p);
5       int size(const t_pile p);
6       int isEmpty(const t_pile p);
7       int top(const t_pile p);
8       ...
```

- Utilisation : ce TDA est particulièrement utilisé pour les compilateurs, la gestion des appels de fonctions, l'évaluation d'expressions ou encore les algorithmes de parcours (en profondeur) de structures hiérarchiques.

↪ Définition du TDA File

la file, ou queue, définit une collection d'éléments liés utilisant la politique FIFO (First In First Out) pour la manipulation des éléments. Plus précisément, le premier élément ajouté à la collection sera le premier à être supprimé.

Opérations :

- `FileVide` : $\{\} \rightarrow File(T)$
- `EstVide` : $File(T) \rightarrow Boolean$
- `Enfiler` : $T \times File(T) \rightarrow File(T)$
- `Defiler` : $File(T) \rightarrow T \times File(T)$

Préconditions :

- `Defiler(f)` défini si et seulement si `!EstVide(f)`

Axiomes :

- `EstVide(FileVide(f)) = VRAI` ; `EstVide(Enfiler(e,f)) = FAUX`
- `Defiler(Enfiler(e,FileVide())) = (e,FileVide())`
- si `!EstVide(f)` alors : `Defiler(Enfiler(e,f)) = (d,Enfiler(e,g))` avec `(d,g)=Defiler(f)`

↪ Implémentation du TDA File

- Opération sur une file (queue)
  - en français : `enfiler, défiler, tête, queue, filevide`
  - en anglais : `enque, dequeue, haid, tail, empty`
- Implémentation :
  - très proche de celle de la pile
  - on ajoutera une référence (i.e. pointeur) sur le début (i.e. la queue) de la file
- Utilisation : ce TDA est particulièrement utilisé comme tampons, ou pour le parcours (en profondeur) de structure hiérarchique.

↪ Définition du TDA Deque

La deque est une généralisation du TDA pile et du TDA file. Précisément, il autorise les opérations d'ajout et de retrait aux deux extrémités de la structure.

Opérations :

- `Deque init()` : création d'une deque vide
- `insertFirst(e,Deque)` : ajout d'un élément en tête
- `insertLast(e,Deque)` : ajout d'un élément en queue
- `Elem removeFirst(Deque)` : retrait d'un élément en tête ; nécessite Deque non vide
- `Elem removeLast(Deque)` : retrait d'un élément en queue ; nécessite Deque non vide
- `Boolean isEmpty(Deque)` : Vrai si vide, faux sinon
- `Elem first(Deque)` : valeur de l'élément de tête ; nécessite Deque non vide
- `Elem last(Deque)` : valeur de l'élément de queue ; nécessite Deque non vide

↪ Définition du TDA Vecteur

le vecteur est une abstraction d'un tableau de taille dynamique n. L'accès aux élément s'effectue par le rang r (indice), avec $0 \leq r < n$ *au moment de l'accès.*

Opérations :

- `Vecteur init(n)` : création d'un vecteur vide
- `Elem elemAtRank(r)` : valeur de l'élément au rang r, avec $0 \leq r < n$
- `Elem replaceAtRank(r,e);` : remplace l'élément au rang r, avec $0 \leq r < n$, par e et retourne l'élément remplacé
- `void insertAtRank(r, e);` : insère l'élément e au rang r, avec $0 \leq r < n$
- `Elem removeAtRank(r);` : supprime l'élément e au rang r, avec $0 \leq r < n$
- `Integer size();` : taille du vecteur
- `Boolean isEmpty();` : vecteur est vide ?

- l'implémentation peut se faire par un tableau ou par une structure dynamique

↪ Définition du TDA Liste

la liste définit une collection d'éléments liés les uns aux autres. L'accès aux éléments de la liste s'effectue par `position` (référence) et par déplacement successif aux éléments contiguës (prédécesseur et successeur).

Opérations :

- `Liste init()` : création d'une liste vide
- `Position first()` : Position du premier élément de la liste
- `Position last()` : Position du dernier élément de la liste
- `Position prev(p)` : Position de l'élément précédent de p
- `Position next(p)` : Position de l'élément suivant de p
- `Elem replace(p,e)` : remplace la valeur de l'élément correspondant à `Position` par e et retourne l'ancienne valeur
- `Position insertFirst(e)` : insère l'élément e en première position et retourne `Position`
- `Position insertLast(e)` : insère l'élément e en dernière position et retourne `Position`

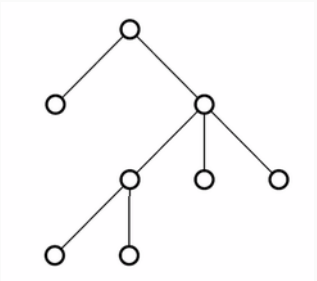
Opérations (suite) :

- `Position insertBefore(pe)` : insère l'élément `e` avant `p` et retourne `Position`
- `Position insertAfter(p, e)` : insère l'élément `e` après `p` et retourne `Position`
- `Elem remove(p)` : supprime l'élément en `p` et retourne la valeur contenue dans l'élément
- `Integer size()` : taille de la liste
- `Boolean isEmpty()` : liste est vide ?

- Cas d'erreur : la manipulation du type `Position` en paramètre de ces différentes opérations peut entrainer des erreurs qui devront être détectées :
  - `p = NULL`
  - `p` a été supprimée de la liste
  - `p` appartient à une autre liste
  - `p` est la première position et appel à l'opération `prev(p)`
  - `p` est la dernière position et appel à l'opération `next(p)`

III - Types abstraits de données hiérarchiques : les arbres

↪ Généralités 1/3



Un arbre est un modèle abstrait d'une structure hiérarchique. Il est constitué de *nœuds* reliés par des *arrêtes* (relation parent-enfant). Il n'y a pas de boucles.

On utilise classiquement la terminologie suivante pour décrire les arbres :

- La *racine* de l'arbre est l'unique nœud qui n'a pas de *père*.
- Un *nœud interne* possède au moins un *fils*.
- Une *feuille* de l'arbre est un nœud qui n'a pas de fils.
- Un *sous-arbre* d'un nœud A est un arbre dont la racine est un fils de A.

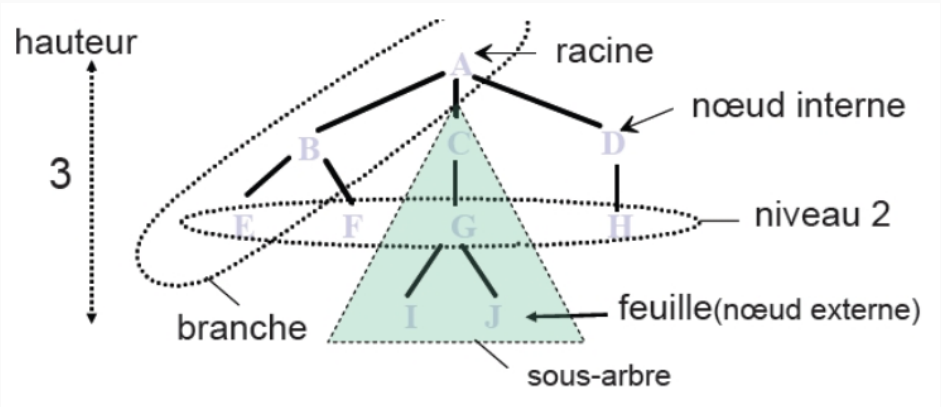
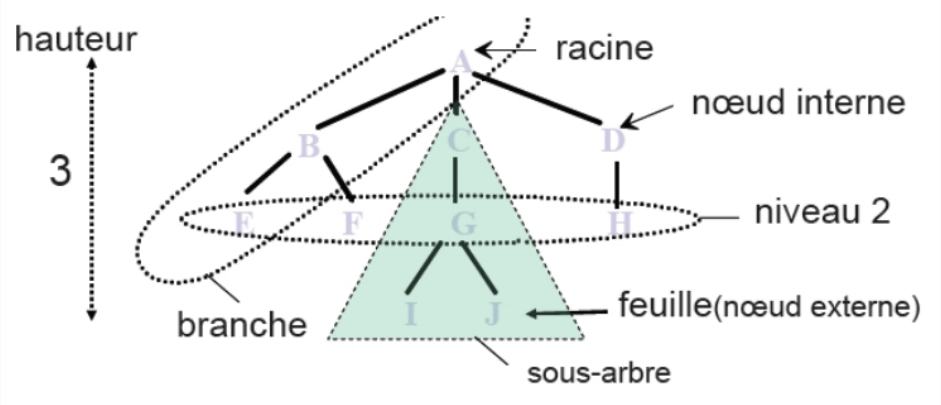


Figure 1 – Terminologie dans un arbre

↪ Généralités 2/3



Les termes père, fils sont repris du langage "généalogique". De même, les nœuds (C,D) sont *frères* de B, (A,C) *ancêtres* de G, et (G,I) *descendants* de C. Les termes suivants sont plus spécifiques :

- *Un chemin* qui relie une feuille à la racine est une branche.
- *La profondeur* d'un nœud est son nombre *d'ancêtres*.
- Le *niveau* situe les nœuds de même profondeur.
- La *hauteur* d'un arbre est sa profondeur maximale.

Les arbres sont des structures utilisés dans de nombreuses modélisation de problèmes comme par exemple la représentation d'une expression arithmétique, l'organisation d'une hiérarchie de dossiers/fichiers, les arbres de jeu, ...

↪ Généralités 3/3

Il existe différents types d'arbres possédant différentes propriétés. On peut distinguer deux grandes familles :

arbres n-aire :

- les arbres dont les nœuds ont au plus *n* fils sont des *arbres n-aire*. cette famille d'arbres comprend par exemple les *B-arbres* très utilisés dans les mécanismes de gestion de bases de données ou de systèmes de fichiers.

arbres binaire :

- lorsque *n* vaut 2, l'arbre est dit *binaire*. Dans ce cas, on utilise les termes de *fils gauche* et *fils droits* d'un nœud, ainsi que les notions de *sous-arbre gauche(SAG)* et de *sous-arbre droit (SAD)*. On trouve dans cette famille les ABR (arbres binaires de recherche) et les AVL [1] qui sont des ABR équilibrés (i.e. $|hauteur(SAG) - hauteur(SAD)| \leq 1$).

1. Nom provenant de ses deux inventeurs Georgii Adelson-Velsky et Evguenii Landis.

La forme d'arbre la plus simple à manipuler est l'arbre binaire. De plus les formes plus générales d'arbres (avec $n > 2$) peuvent souvent être représentées par des arbres binaires.

↪ Définition d'un arbre binaire

> *En utilisant pleinement le coté récursif de sa structure on peut définir un arbre comme suit : un arbre est soit vide, soit composé d'une valeur (i.e. d'un élément), d'un sous-arbre gauche et d'un sous-arbre droit.*

↪ TDA arbre binaire

Opérations :

- `arbre arbreVide()`
  - constructeur d'un arbre vide
- `arbre arbreCons(Elem e, arbre fg, arbre fd)`
  - constructeur d'un sous arbre constitué de sa racine, de son sous-arbre gauche et de son sous-arbre droit
- `Boolean vide(arbre a)`
  - l'arbre est vide ?
- `Elem valeur(arbre a)`
  - valeur de la racine de a ; nécessite un arbre non vide
- `arbre sag(arbre a)`
  - retourne le sous-arbre gauche ; nécessite a non vide
- `arbre sad(arbre a)`
  - retourne le sous-arbre droit ; nécessite a non vide
- `Integer taille(arbre a)`
  - retourne le nombre de nœuds de l'arbre a
- `Integer hauteur(arbre a)`
  - retourne la hauteur de l'arbre a

↪ Implémentation d'un arbre binaire

Deux manières d'implémenter l'arbre peuvent être envisagées :

- la plus courante qui consiste à ne modéliser que les liens entre un noeuds et ses fils gauche et droite
- enrichir la précédente en ajoutant la modélisation du lien du nœud vers son père. Cette modélisation peut simplifier l'écriture de certains algorithmes mais nécessite un peu plus de mémoire pour le stockage de l'arbre.

↪ Définition

La structure d'arbre (binaire ou *n*-aire) peut se parcourir de différentes façons en fonction de la manière dont elle est utilisée. Plus précisément, on peut distinguer 2 familles de parcours :

- *parcours en profondeur* : cette famille de parcours consiste à parcourir l'arbre de la racine jusqu'au feuille puis de la gauche vers la droite. On distingue dans cette famille trois types de parcours se distinguant par le moment auquel l'on effectue le traitement sur le nœud intermédiaire (i.e. n'étant pas une feuille) visité :
  - *parcours en profondeur préfixe* : le traitement s'effectue lors de la première visite d'un nœud intermédiaire
  - *parcours en profondeur infixe* : le traitement s'effectue lorsque tout le sous-arbre gauche du nœud a été visité
  - *parcours en profondeur suffixe* : le traitement s'effectue lorsque tous les fils du nœud ont été visité

- *parcours en largeur* : ce parcours consiste à explorer l'arbre de la gauche vers la droite puis de la racine vers les feuilles. Le traitement effectué est fait sur le nœud le plus proche de la racine.

↪ Exemple

Voici le résultat
des différents parcours sur l'exemple de la figure 2 :

- parcours en profondeur préfixe :
  → + * a b / - b c d
- parcours en profondeur infixe :
  → a * b + b - c / d
- parcours en profondeur suffixe :
  → a b * b c - d / +
- parcours en largeur : → + * / a b - d b c

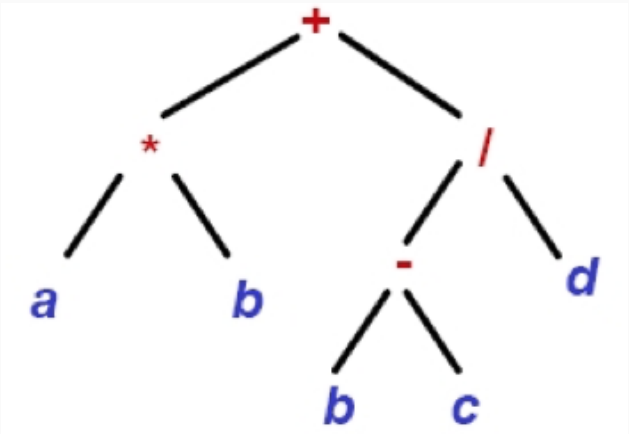


Figure 2 – Arbre d'expression

↪ Implémentation

Parcours en profondeur :

- les parcours en profondeur préfixe, infixe et suffixe s'écrivent assez naturellement et simplement de manière récursive.
  - La seule différence provenant du moment auquel on effectue le traitement sur le nœud.
- la version itérative d'un parcours en profondeur, nécessite de conserver une liste des nœuds en attente de traitement pour remplacer l'utilisation implicite de la pile mémoire de la version récursive.
  - Pour respecter le principe du parcours en profondeur (toujours visiter le nœud le plus éloigné de la racine), la liste utilisée aura une structure de *pile* de nœuds.

Parcours en largeur :

- La version itérative du parcours en largeur est très proche de celle du parcours en profondeur. En effet, seule la structure de liste de nœuds en attente change :
  - on utilisera une structure de *file* permettant de visiter le nœud le plus proche de la racine.

↪ Définition

Un arbre binaire de recherche (ABR), ou binary search tree (BST) en anglais, est une structure de données permettant de représenter un ensemble totalement ordonné de valeur. L'avantage de cette structure est de proposer des opérations (recherche, insertion, ...) plus rapide que d'autres structures de données.

Structurellement, un arbre binaire de recherche diffère d'un arbre binaire classique par les propriétés suivantes :

- chaque nœud possède une clé (qui peut être la valeur stockée)
- chaque nœud du sous arbre gauche doit posséder une clé inférieure ou égale à celle contenue dans le nœud considéré
- chaque nœud du sous arbre droit doit posséder une clé supérieure ou égale à celle contenue dans le nœud considéré
- l'insertion d'un nœud dans l'arbre est toujours une feuille
- le parcours infixe permet de récupérer l'ensemble des valeurs d'un arbre binaire de recherche dans l'odre croissant

↪ Définition (suite)

Dans la famille des arbres binaires de recherche, on parle :

- d'arbre binaire de recherche *complet* quand chaque nœud possède 0 ou 2 fils
- d'arbre binaire de recherche *parfait* quand celui-ci est complet et que toutes les feuilles sont à la même hauteur
- d'arbre binaire de recherche *dégénéré* quand tout nœud de l'arbre à au plus 1 fils
- d'arbre binaire de recherche *équilibré* si tout *chemins* de la racine aux feuilles ont même longueur

↪ Fonctionnalités

Les trois opérations classiques d'un arbre binaire de recherche sont : recherche, insertion et suppression.

Recherche :

- la fonction de recherche dans un arbre binaire de recherche s'écrit très simplement de manière récursive. En effet, la structure de l'arbre permettant de choisir à chaque nœud le prochain sous-arbre à visiter. Précisément :
  - la recherche commence par la racine
  - Si la valeur de la racine est celle recherchée l'algorithme se termine et retourne `Vrai`
  - Sinon
    - Si la valeur recherchée est strictement inférieure alors on relance la recherche sur le sous-arbre gauche
    - Sinon la recherche est relancée sur le sous-arbre droit
  - Si l'on atteint une feuille possédant une valeur différente de celle recherchée, alors l'algorithme se termine et retourne `Faux`

↪ Fonctionnalités (suite)

Insertion :

- cette fonction est également très simple puisque se faisant au niveau d'une feuille et après une recherche. Ainsi
  - on commence par recherchée la valeur à insérer
  - lorsque l'on arrive à une feuille :
    - si la valeur est strictement inférieure, la feuille devient un nœud et on insère un nouveau nœud comme fils gauche de celui-ci ;
    - si la valeur est strictement supérieure, la feuille devient un nœud et on insère un nouveau nœud comme fils droit de celui-ci ;
    - si la valeur est égale, on insère alternativement un nouveau nœud à droite ou à gauche. Cette méthode permet d'éviter un trop fort déséquilibre de l'arbre.

↪ Fonctionnalités (suite)

Suppression :

- l'algorithme de suppression est un peu plus complexe, en particulier lors de la suppression d'un nœud possédant 2 fils. Voyons en détails les différents cas pouvant se présenter après avoir recherché et trouvé le nœud à supprimer :
  - le nœud est une feuille : c'est le cas le plus simple, puisqu'il suffit simplement de supprimer le nœud de l'arbre ;
  - le nœud ne possède qu'un fils : dans ce cas, le nœud est supprimé et on le remplace par son père, i.e. le sous-arbre gauche ou droit est "remonté" de 1 ;
  - le nœud possède deux fils : dans ce cas, le principe est de se ramener, après un échange, à l'un des deux cas précédents. Pour effectuer l'échange deux solutions peuvent êtres utilisées, là encore alternativement pour éviter le déséquilibre de l'arbre :
    - soit on échange le nœud à supprimer avec son plus proche successeur, i.e. le nœud le plus à gauche de son sous-arbre droit, puis on applique de nouveau l'algorithme de suppression sur le nœud à son nouvel emplacement (soit une feuille, soit un nœud avec un fils unique (le frère du nœud échangé))
    - soit on échange le nœud à supprimer avec son plus proche prédécesseur, i.e. le nœud le plus à droite de son sous-arbre gauche, puis on applique de nouveau l'algorithme de suppression sur le nœud à son nouvel emplacement.

IV - Types abstraits de données hiérarchiques : les graphes

↪ Généralités

Les graphes sont des structures plus générales que les arbres. Comme les arbres, ils sont très utiles dans la modélisation de nombreux problèmes (réseaux (routiers, télécommunications, aériens, ...)), cartes, automates, ...).

↪ Définition

De manière générale, un graphe $G(V, E)$ est constitué :

- d'un ensemble V de *sommets* (Vertex),
- d'un ensemble E d'*arcs* ou d'*arêtes* (Edge) ; un arc E se définit comme un couple de sommets.

Dans la famille des graphes, il faut différencier les *graphes non orientés* des *graphes orientés*. La différence entre ces 2 types de graphe se fait dans la définition d'un arc.

↪ **Graphe non orienté :**

- dans le cas d'un graphe non orienté, E est un ensemble de couples *non ordonnés* de sommets

↪ **Graphe orienté :**

- pour un graphe orienté, E est un ensemble de couples *ordonnés* de sommets $\{v_i, v_j\}$
  v_i est le sommet *origine* ou *initial*, et v_j est le sommet *terminal* ou *extrémité*.
  L'arc $e = v_i, v_j$ est dit *sortant* en v_i et *incident* en v_j. v_j est un *successeur* de v_i et inversement, v_i est un *prédécesseur* de v_j. L'ensemble des successeurs de v_i est noté $Succ(v_i)$ et l'ensemble des prédécesseurs de v_i est noté $Pred(v_i)$.

↪ Propriétés 1/2

Différentes propriétés permettent de caractériser les graphes :

- *Adjacence :* soit un arc $e_1 = \{v_i, v_j\}$, alors ont dit que v_i et v_j sont des sommets *adjacents*. Ainsi $Adj(v_i)$ est l'ensemble des sommets adjacents au sommet v_i.
- *Boucle :* une boucle est un arc reliant un sommet à lui-même.
- *Graphe simple :*un *graphe simple* est un graphe non orienté ne comportant pas de boucle et si il ne possède jamais plus d'un seul arc entre deux sommets.
- *Multi-graphe :* un *multi-graphe* est un graphe non orienté non simple.
- *Graphe élémentaire :* un graphe orienté est *élémentaire* si il ne contient pas de boucle.
- *p-graphe :* un graphe orienté est un *p-graphe* si il contient au plus p arc(s) entre deux sommets.
- *Ordre :* l'ordre d'un graphe correspond au nombre de ses sommets, i.e. $card(V)$.

↪ Propriétés 2/2

- *Taille :* la taille d'un graphe est définie comme le nombre de ses arcs, i.e. $card(E)$.
- *Degré d'un sommet :* on distingue les cas d'un graphe non orienté et d'un graphe orienté :
  - dans un graphe non orienté, le *degré d'un sommet* v_i, noté $d(v_i)$, se définit comme le nombre d'arc(s) incident(s) à v_i. Attention, dans ce cas, une boucle compte pour 2.
  - dans un graphe orienté, on distingue :
    - *le demi degré extérieur ou sortant* d'un sommet v_i noté $d^+(v_i)$ se définit comme le nombre d'arc(s) partant du sommet v_i
    - *le demi degré intérieur ou entrant* d'un sommet v_i noté $d^-(v_i)$ se définit comme le nombre d'arc(s) arrivant en v_i.

    ainsi, le *degré d'un sommet* v_i dans un graphe orienté est défini par :
    $d(v_i) = d^+(v_i) + d^-(v_i)$.

↪ Différents choix :

Il est possible d'implémenter un graphe de différentes manières ; les 3 implémentations les plus courantes sont :

- les listes d'adjacence, dynamiques ou non,
- la matrice d'adjacence, dynamique ou non,
- la structure chaînée.

↪ listes d'adjacence :

Soit $G = (V, E)$ un graphe d'ordre n avec des sommets numérotés de 1 à n. Cette implémentation consiste en un tableau de n listes d'adjacence (une par sommet). Cette dernière se définit comme une liste chaînée de tous les sommets v_j tels qu'il existe un arc (v_i, v_j). Il n'y a pas de notion d'ordre dans les listes d'adjacence. Dans le cas d'un graphe non orienté, v_j appartient à la liste d'adjacence de v_i et inversement.

↪ **matrice d'adjacence :**

Soit $G = ()V, E)$ un graphe d'ordre n avec des sommets numérotés de 1 à n. L'implémentation par matrice d'adjacence consiste en une matrice M de taille $n * n$ telle que $M(i, j) = 1$ si il existe $e = \{v_i, v_j\}$ dans le graphe et $M(i, j) = 0$ sinon. Dans le cas d'un graphe non orienté, la matrice est symétrique selon sa diagonale descendante et l'on ne mémorise donc dans ce cas que la composante triangulaire supérieure de celle-ci.

↪ **Structure chaînée :**

Cette représentation consiste a réalisée une construction similaire à l'arbre généralisé. Néanmoins, elle est beaucoup plus complexe et n'est plus adaptée dès lors que l'on souhaite associer une valeur à un arc (qui serait modélisé ici par un pointeur).