COURS DE STRUCTURES DE DONNÉES
LICENCE 2 - UNIVERSITÉ CLERMONT 2

MAMADOU MOUSTAPHA KANTÉ

Table des matières

L'algorithmique remonte à l'antiquité. On peut citer le calcul des impôts à Babylone ou le calcul des surfaces cultivables après les crues du Nil dans l'Egypte Ancienne, le crible d'Erathostène qui permet de trouver tous les nombres premiers inférieurs à un certain entier, etc. De nos jours, avec l'avènement des ordinateurs

Date: 31 mars 2014.

l'algorithmique fait partie intégrante de notre vie de tous les jours. On peut citer : acheminement (poste, GPS, téléphone, Internet), ordonnancement (usines), multimédia (compression), flots (emploi du temps, embarquement), etc.

Dans ce cours on va étudier certaines méthodes pour manipuler les données des algorithmes. Ce cours est basé sur les livre [1] et [4] et le cours de J.G. Penaud auquel j'ai assisté quand j'étais étudiant à l'Université Bordeaux 1 [5]. Des idées sont empruntées au polycopié du cours dispensé par O. Raynaud les années précédentes [6]. Je remercie également Wikipedia. Les Chapitres 3, 5 et 6 ont chacun une section dédiée à une ou plusieurs applications.

1. Niveau de Description

Une partie de ce chapitre est basé sur le livre [7].

1.1. **Structure Générale d'un Ordinateur.**

Le modèle de nos ordinateurs est celui de Von Neumann (communément appelé *machines à registres*). Ils sont équivalents aux *machines de Turing* et au *lambda-calcul* de Church, et sont principalement composés de :

(1) d'une *mémoire adressable* en lecture et écriture,
(2) d'une *unité arithmético-logique* (communément appelé processeur) qui s'occupe des calculs,
(3) d'un *compteur d'instructions* pour exécuter les instructions une à une.

En Fig. **??** vous avez une version simplifiée de l'ordinateur. On peut trouver :

(1) un bus de données où circulent les valeurs représentant les données,
(2) un bus d'adresses où circulent les valeurs représentant les valeurs,
(3) dans la partie 1, on a :
    (a) une micro-mémoire : stockage des instructions de l'ordinateur,
    (b) un micro-pc : compteur pour les instructions de la micro-mémoire,
    (c) un décodeur d'instructions : les instructions en mémoire sont traduites par ce dernier en des instructions de la micro-mémoire,
(4) dans la partie 2, on a l'unité arithmético-logique avec deux registres $R0$ et $R1$ qui servent comme support de stockage et qui constitue l'unité de calcul de la machine [1],
(5) dans la partie 3, on a le compteur ordinal qui sert à incrémenter les instructions dans la mémoire (les programmes sont séquentiels),
(6) dans la partie 4, on a :
    (a) une mémoire centrale pour le stockage des programmes en exécution et les données de ces derniers,
    (b) un registre d'adresses : certaines données en mémoire représentent des adresses et il faut pouvoir les communiquer au bus d'adresses pour lire/écrire dessus.

A ces quatre parties, il faut ajouter les périphériques d'entrées/sorties, de stockage, etc. Il faut remarquer que ces différentes parties peuvent communiquer entre elles à travers d'autres circuits. Chaque partie d'un ordinateur est réalisée à partir de circuits dont les entrées et les sorties ont deux états possibles : un état positif (interprété comme 1) et un état négatif (interprété comme 0). Un état est appelé *bit*.

1. Un registre est un circuit séquentiel servant à stocker une séquence de bits. Il dispose d'une entrée spéciale qui commande le chargement des données.

Pour que les constructeurs puissent modifier leurs ordinateurs (avec de nouvelles instructions, amélioration des performances, ...) sans pour autant obliger à réécrire les programmes, seules quelques parties sont visibles aux programmeurs. En particulier, nous n'avons accès en particulier qu'à la mémoire centrale, le compteur ordinal et quelques registres dont $R0$ et $R1$.

1.2. **Mémoire Centrale.** La mémoire centrale est divisée en blocs (appelés *mots*) de même taille. La taille des mots représente la taille des bus et donc des adresses et permet de calculer ainsi la taille maximale d'une mémoire dans un ordinateur. Un mot se mesure en bits (ex : 32bits, 64bits, etc.). Si la taille d'un mot mémoire c'est n, alors le nombre de mots possibles c'est 2^n et ceci est la taille maximale d'une telle mémoire [2]. Ex : Les machines avec des mots de 32bits ne peuvent accepter que des mémoires avec moins de $4Go$.

Chaque mot de la mémoire contient soit une donnée ou une instruction ou une adresse. Il peut arriver qu'un mot contienne une instruction et les arguments de cette dernière.

1.3. **Langages.** Le `langage machine` c'est l'ensemble des instructions supportées par une machine. Les instructions d'une machine sont codées en binaire et malheureusement nous ne sommes pas faits pour réfléchir en binaire (d'ailleurs cela fait des programmes presque illisibles pour le commun des mortels, même si au début de l'informatique c'était la manière que l'on avait de programmer avec les cartes perforées). Pour remédier à ce problème, on crée des *langages de programmation* qui sont un ensemble de *mots clés* (chacun avec un sens) et de règles de constructions pour écrire des programmes. Ces programmes sont ensuite traduits en `langage machine`à l'aide de traducteurs.

Assembleur. La première tentative a été d'utiliser des instructions en base 16 (*hexa-décimal*) et le langage obtenu est appelé `assembleur`. Chaque instruction de la machine est codée en hexa-décimal et comme on dispose de plus de mots d'une même taille, on peut également coder des combinaisons d'instructions du `langage machine` en une seule instruction hexa-décimale. Lorsque l'on connaît les correspondances entre instructions machines et codes hexa-décimaux on peut traduire (à la main) son programme `assembleur` en `langage machine`.

Exemple 1. *Dans les machines x86,* 10110000 01100001 *qui signifie "écrire* 97 *dans le registre al" se traduit en* `assembleur` *par movb* $0x61, \%al$

Même si l'`assembleur` est moins verbeux que le `langage machine`, il reste un peu trop complexe à manipuler et est très dépendant de la machine. Néanmoins il reste le meilleur moyen raisonnable pour écrire des codes optimisés et reste encore utilisé dans beaucoup de programmes (comme les systèmes d'exploitation ou les pilotes de périphériques).

Les langages de haut niveau. On utilise les mécanismes des langages humains en utilisant les travaux déjà effectués dessus par les linguistes tels que Chomsky et al. Cependant, les langages humains sont trop complexes et nous ne pourrons jamais avoir un programme qui puissent les traduire en langages machines. Néanmoins on peut s'en inspirer et produire des modèles mathématiques qui conviennent à nos besoins. Les langages de haut niveau sont constitués :

2. Il faut se rappeler qu'une mémoire centrale est un circuit et donc chaque entrée a valeur soit 0 soit 1.

(1) d'un certain nombre d'expressions (chacun ayant un sens) et un ensemble de règles sophistiquées qui permet d'écrire des programmes compréhensibles par les humains entraînés (ceux qui prennent le temps de l'apprendre),
(2) d'un ensemble d'outils pour traduire les programmes en langages machines.

L'avantage de cette approche : un seul programme pour des types de machines différents, il suffit juste de disposer d'un traducteur pour chaque type.

On distingue 2 grandes classes de langages de haut niveau.

(1) les langages compilés : on écrit un code source puis on le compile. A la compilation un fichier en langage `assembleur` est créé qui est encore traduit en `langage machine` par un autre outil (le même outil peut faire les deux étapes). Ensuite, on appelle un éditeur de liens pour "relier" les différents bouts compilés séparément. Des exemples de langages compilés sont le `C`, `Ocaml`, `Lisp`, ...
(2) les langages interprétés : un outil lit le code source et "simule" l'exécution. Il traduit chaque instruction du langage et l' exécute. Des exemples sont les langages de script (`Javascript`, `HTML`, ...), `Java`, `Lisp`, `Ocaml`, ...

On peut également les classifier suivant trois principaux paradigmes.

(1) paradigme impératif : on a des effets de bord sur les objets manipulés (on les manipule à travers leurs adresses en mémoire). Un programme est structuré en instructions données à exécuter par la machine. On peut citer le `C`, `C++`, `Pascal`, ...
(2) paradigme objet : les valeurs manipulées sont regroupées en ensembles et chaque ensemble accepte un certain nombre d'opérations. On peut citer le `C++`, `Java`, `Scala`, ...
(3) paradigme fonctionnel : les fonctions sont les objets de base du langage (toute valeur est vue comme une fonction). On peut citer `Lisp`, `Ocaml`, `Scala`, `Haskell`, ...

Un langage peut combiner les différents paradigmes et les concepteurs peuvent proposer des outils pour l'interpréter et/ou le compiler. On peut mesurer la qualité d'un langage suivant plusieurs critères :

(1) la verbosité, *i.e.* la capacité de faire des gros programmes avec peu de lignes de code,
(2) les méthodes proposées pour structurer un programme,
(3) la complétude du langage, *i.e.*, est-ce que l'on peut programmer tout ce que peut permettre les `langage machine`.
(4) ...

2. Algorithmes, Valeurs, Types et Éléments du Langage

Le mot *algorithme* vient d'un mathématicien arabe du 9ème siècle (Al Khouwarizmi). Un *algorithme* c'est une série d'opérations à effectuer dans le but de résoudre un problème. Il prend en entrée des données et fournit le résultat. La mise en oeuvre de l'algorithme (appelée aussi implémentation), *i.e.*, écriture des différentes opérations dans un langage de programmation donne un *programme* dans le langage choisi. Avant d'écrire des algorithmes, il faut d'abord se rappeler de ces deux théorèmes prouvés respectivement en 1932 par Gödel, et en 1935-36 par Turing et Church indépendamment.

Theorem 1 (Indécidabilité)**.** *On ne peut pas résoudre tous les problèmes avec des algorithmes. Le problème de l'arrêt et la vérification de programmes sont des exemples.*

Theorem 2 (Thèse de Church-Turing)**.** *Les problèmes ayant une solution algorithmique sont exactement ceux que l'on peut résoudre par une machine de Turing (théorie de la calculabilité).*

2.1. **Données.** Un *identifiant* c'est une suite de caractères alphanumériques (on commence toujours par un caractère alphabétique).

Un *type* c'est un ensemble de valeurs muni d'un ensemble d'opérations sur les valeurs et un certain nombre d'axiomes/propriétés vérifiées par les opérations. Un type est représenté/identifié par un identifiant.

Exemple 2. *Si on prend le type* int *en* C *: les valeurs sont les entiers. Les opérations sont l'addition (+), la multiplication (*), la soustraction (-), la division (/)et le modulo (%). La division par* 0 *est interdite.*

Une *constante* c'est une valeur d'un certain type. Par exemple 10 est une constante (c'est une valeur du type int en C). Une *donnée* c'est soit une constante, soit une entité qui a un type, une valeur et un identifiant.

Une *variable* c'est un identifiant qui désigne un espace de stockage en mémoire et qui a un type T, *i.e.*, que dans cet espace on ne peut stocker que des valeurs de T. Une *affectation* d'une valeur v à une variable x consiste à stocker v dans l'espace de stockage désigné par x. Avant toute utilisation une variable doit être *initialisée* (il faut y affecter une valeur). Après initialisation, une variable peut être manipulée comme n'importe quelle valeur de son type. Il faut faire la différence entre une variable et le contenu qui y ait stocké.

On distingue deux sortes de types.

(1) Les *types primitifs*. Les valeurs sont non décomposables (*i.e.*, atomiques) et fournies par défaut. Dans la plupart des langages les types primitifs sont les entiers, les réels, les caractères. Certains proposent en plus les chaînes de caractères et les booléens.

(2) Les *types composés*. Ils sont construits à partir d'autres types. Dans la plupart des langages de programmation on fournit par défaut le type composé tableau, qui est un ensemble de cases mémoires où on stocke des valeurs du même type avec la propriété que l'on puisse accéder à chaque case mémoire à l'aide d'un indice (historiquement l'indice c'est entre 0 et $n-1$ avec n le nombre d'éléments). Suivant les langages de programmation voici les différentes techniques pour construire d'autres types composés.

   produit: Si $\mathbf{T_1}, \ldots, \mathbf{T_n}$ sont des types, on peut construire le type $\mathbf{T_1} \times \ldots \times \mathbf{T_n}$ composé des valeurs $\{(x_1, \ldots, x_n) \mid x_i \in \mathbf{T_i}\}$.

   somme: Si $\mathbf{T_1}, \ldots, \mathbf{T_n}$ sont des types, on peut construire le type $\mathbf{T_1} \cup \ldots \cup \mathbf{T_n}$ dont l'ensemble des valeurs c'est l'union des valeurs des $\mathbf{T_i}$.

   enregistrement: C'est un type composé de plusieurs entités appelé *champs*, chacun ayant un identifiant et un type. C'est un type produit mais où il n'y a pas d'ordre entre les champs. Comme pour les tableaux, chaque champ est un espace de stockage en mémoire.

2.2. **Tableaux statiques.** La propriété fondamentale des tableaux statiques c'est que l'on puisse accéder à chaque élément en spécifiant seulement son indice et le temps d'y accéder doit être proportionnel à la taille de l'élément. Le moyen le plus simple de représenter un tableau en mémoire est de stocker ses éléments consécutivement (et c'est ce qui est fait le plus souvent). Ainsi si *tab* est un tableau et chaque élément du tableau nécessite c mots pour son stockage, alors le $j$éme élément est stocké à l'adresse $tab_0 + cj$ où tab_0 est l'adresse du premier élément. Avec cette représentation, on n'a besoin de connaître que l'adresse du premier élément

appelé *adresse de base*. L'inconvénient majeur de cette représentation c'est qu'il faut connaître à l'avance le nombre d'éléments du tableau. En plus il faut faire attention car si le tableau contient n éléments, demander à accéder au $(n+i)$ème élément peut produire un comportement bizarre (on accède à une zone mémoire dans laquelle on n'a peut-être pas accès ou la valeur lue n'est pas cohérente), ce genre de comportement peut produire des erreurs dites *overflow*.

On peut étendre cette représentation des tableaux statiques à une dimension aux tableaux statiques à deux, trois, quatre, ... dimensions de telle sorte que pour accéder à un élément, on donne juste ses *coordonnées*.

Il faut noter que dans notre représentation des tableaux, il n'y a aucun moyen de connaître la taille d'un tableau. Ainsi, il faudra toujours, lorsque l'on donne un tableau en paramètre à une fonction, spécifier dans la liste des paramètres la taille du tableau si on en a besoin dans la fonction.

On verra plus tard comment faire des tableaux dynamiques (pas besoin de connaître à l'avance la taille).

2.3. **La Syntaxe du Langage.** On va utiliser une syntaxe proche de celle du C. Chaque instruction se termine par un point-virgule. On va utiliser la même syntaxe que le C pour les structures conditionnelles et itérative (voir n'importe quel livre sur le C, par exemple [3]). Une variable de type T est déclarée à l'aide de l'instruction T *id_var* ;. Pour affecter une valeur *val* à une variable *var* on écrit *var* := *val* ;.

Types primitifs. Nous allons disposer de int, float, bool, char, strings représentant respectivement les entiers, les réels, les booléens, les caractères et les chaînes de caractères. Pour afficher une valeur ou le contenu d'une variable d'un type primitif, on écrit print *val* ;. Pour effectuer des tests de comparaison sur les types primitifs on va utiliser les symboles $=, >, <, \leq, \geq, \neq$ avec la signification standard. Pour concaténer deux chaînes de caractères on va utiliser le caractère ^. Pour identifier les caractères des chiffres et des chaînes de caractères composées d'un seul caractère, on écrit les caractères entre apostrophes et les chaînes de caractères entre guillemets. Par exemple les valeurs '0', 0 et "0" sont respectivement de type char, int et string.

Types composés. Pour déclarer un tableau de n éléments de type T on écrit T *id_tab*[n] ;. Les indices d'un tableau de taille n vont de 0 à $n-1$. Pour accéder au $i$ème élément d'un tableau *tab* on écrit *tab*[i]. Comme chaque case d'un tableau représente un espace de stockage, alors *tab*[i] peut-être utilisée comme n'importe quelle variable.

Pour déclarer un tableau à k dimensions avec $n_1 \times \ldots \times n_k$ éléments de type T on écrit T *id_tab*[n_1][n_2]...[n_k] et pour accéder à la valeur stockée à la case $(i_1, \ldots, i_k)$, on écrit *id_tab*[i_1][i_2]...[i_k]. Ce qui est valable pour les tableaux à une dimension reste aussi valable.

Pour créer des types composés, on va utiliser le constructeur de type struct qui construit des types enregistrement. Si on veut créer un type id_type on écrit struct id_type liste des champs ; où chaque champ est donné sous la forme type *id_champ* ;. Après une déclaration du type id_type, on peut les utiliser pour déclarer des variables comme pour n'importe quel autre type. Lorsque *var* est une variable de type id_type, pour accéder à un champ *id_champs*, on écrit *var.id_champs*. Comme chaque champ est un espace de stockage, on peut le manipuler comme une variable.

Fonctions. La syntaxe pour déclarer et appeler des fonctions est la même qu'en C. On va juste rappeler que pour lister les paramètres d'une fonction, on a deux

possibilités pour expliciter les *modes d'échanges*, *i.e.*, la façon dont les arguments seront transmis à la fonction :

(1) le passage par *valeur* : l'argument est copié avant de le transmettre à la fonction et donc la fonction en a une copie locale et toute modification faite par la fonction est perdue au retour de la fonction,

(2) le passage par *référence* : on donne l'adresse en mémoire de l'argument et donc la fonction appelée partage avec la fonction appelante la variable représentant l'argument.

On va considérer le passage par valeur comme le mode par défaut. Si on veut spécifier un paramètre par référence, on le fait comme en `C`, ie on utilise le caractère & devant l'identifiant du paramètre. Avec notre représentation des tableaux, les tableaux sont toujours des paramètres par référence et donc on aura pas besoin d'ajouter le caractère &. Le tableau suivant compare les deux modes de passage des paramètres.

	Valeur	Référence
Rapidité	non	oui
Coût en mémoire	oui	non
Effet de bord	non	oui
sécurité	oui	non

3. Types de Données Abstraits

Dans ce chapitre on va définir ce que l'on appelle *types de données abstraits* (TDA) et ensuite introduire les TDA Pile, File et Liste qui sont les plus couramment utilisées. Pour chacun d'eux on va donner une ou deux applications.

3.1. **TDA.** Lorsque l'on écrit un algorithme, on manipule des ensembles de données. Ces ensembles, contrairement aux ensembles mathématiques, peuvent subir des modifications au cours de l'algorithme. Ensuite, chaque ensemble regroupe des valeurs et est défini par un certain nombre d'opérations à effectuer sur les valeurs. Ces ensembles et les opérations associées sont en général définis par les *spécifications*, *i.e.*, le cahier des charges, de l'algorithme.

Les ensembles et opérations spécifiés par le cahier des charges sont communément appelés *types de données à modéliser* (TDM). La formalisation mathématique des TDM est ce que l'on appelle *types abstraits de données* (TDA) et qui spécifie comment chacune des opérations devrait se comporter suivant le TDM. Le programme lui manipule ce que l'on appelle des *types de données concrets* (TDC) qui sont des implémentations des TDA (Pour un TDA donné, il peut exister plusieurs implémentations différentes comme on va le voir plus loin).

Definition 1. *Un TDA est une entité constituée d'une* signature *et d'un ensemble d'*axiomes*.*

(i) La signature est composée d'un identifiant (le nom par lequel désigner le TDA), les identifiants et signature de chaque opération et les types prédéfinis utilisés.

(ii) Les axiomes définissent le comportement des opérations sur les valeurs du TDA.

Lorsqu'un TDA est défini, la question fondamentale que l'on se pose c'est la complétude (tous les comportements sont modélisés) et la consistance des axiomes (pas de contradiction entre les axiomes).

Exemple 3. *On veut manipuler des ensembles de réels. On peut définir le TDA VecteurReel qui utilise les réels et les entiers avec les opérations suivantes :*

$$\begin{aligned}
&\mathit{ObtenirI} : \mathit{VecteurReel}*\mathit{Entier} \to \mathit{Reel}\\
&\mathit{ModifierI} : \mathit{VecteurReel}*\mathit{Entier}*\mathit{Reel} \to \mathit{Reel}\\
&\mathit{Sup} : \mathit{VecteurReel} \to \mathit{Entier}\\
&\mathit{Inf} : \mathit{VecteurReel} \to \mathit{Entier}
\end{aligned}$$

On peut utiliser des axiomes du genre :

$$\mathit{Inf}(V) \leq\ i \leq \mathit{Sup}(s) \Longrightarrow \mathit{ObtenirI}(\mathit{ModifierI}(s,i,e),i)=e$$

Maintenant, on va voir quelques exemples de TDA de structures linéaires. Il faut noter que dans la plupart des TDA les noms des opérations ne sont pas standard, mais par contre leurs comportements sont bien définis. Dans la suite, on va toujours suffixer les opérations avec le nom du TDA (lorsqu'il y a ambiguïté).

3.2. **TDA Dictionnaire.** Un *dictionnaire* est un ensemble qui supporte les opérations suivantes : insertion, suppression et test d'appartenance. Le TDA Dico_T utilise les types bool et T, et a comme opérations

$$\begin{aligned}
&\mathsf{creerDico} : () \to \mathsf{Dico_T}\\
&\mathsf{insererDico} : \mathsf{Dico_T}*\mathsf{T} \to \mathsf{Dico_T}\\
&\mathsf{supprimerDico} : \mathsf{Dico_T}*\mathsf{T} \to \mathsf{Dico_T}\\
&\mathsf{appartenanceDico} : \mathsf{Dico_T}*\mathsf{T} \to \mathsf{bool}
\end{aligned}$$

Les axiomes sont les suivants qui expriment en particulier que creerDico() crée un dictionnaire vide.

$$\begin{aligned}
&\mathsf{appartenanceDico}(\mathsf{creerDico}(), \mathsf{x}) = \mathsf{False}\\
&\mathsf{appartenanceDico}(\mathsf{supprimerDico}(\mathsf{D},\mathsf{x}), \mathsf{x}) = \mathsf{False}\\
&\mathsf{appartenanceDico}(\mathsf{insererDico}(\mathsf{D},\mathsf{x}), \mathsf{x}) = \mathsf{True}
\end{aligned}$$

3.3. **TDA Ensemble Dynamique.** Un ensemble dynamique est un dictionnaire mais avec la propriété que les objets stockés sont linéairement ordonnés. Le TDA Ensemble_Dynamique_T utilise les types bool et T, et a comme opérations

$$\begin{aligned}
&\mathsf{creerEnsDyn} : () \to \mathsf{Ensemble_Dynamique_T}\\
&\mathsf{insererEnsDyn} : \mathsf{Ensemble_Dynamique_T}*\mathsf{T} \to \mathsf{Ensemble_Dynamique_T}\\
&\mathsf{supprimerEnsDyn} : \mathsf{Ensemble_Dynamique_T}*\mathsf{T} \to \mathsf{Ensemble_Dynamique_T}\\
&\mathsf{appartenanceEnsDyn} : \mathsf{Ensemble_Dynamique_T}*\mathsf{T} \to \mathsf{bool}\\
&\mathsf{minEnsDyn} : \mathsf{Ensemble_Dynamique_T} \to \mathsf{T}\\
&\mathsf{maxEnsDyn} : \mathsf{Ensemble_Dynamique_T} \to \mathsf{T}\\
&\mathsf{sucEnsDyn} : \mathsf{Ensemble_Dynamique_T}*\mathsf{T} \to \mathsf{T}\\
&\mathsf{predEnsDyn} : \mathsf{Ensemble_Dynamique_T}*\mathsf{T} \to \mathsf{T}
\end{aligned}$$

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

Beaucoup de TDA sont des ensembles dynamiques, mais avec des restrictions. Nous allons en voir deux dans ce chapitre : les piles et les files.

3.4. **TDA Pile.** Comme on veut subdiviser les programmes en sous-programmes et que l'on veuille écrire des fonctions récursives, il faut avoir un mécanisme qui permet de gérer les appels de fonctions. Une méthode consiste à mettre dans le code machine de la fonction appelée une zone mémoire où on stocke l'adresse de la fonction appelante. Ainsi, dès qu'une fonction doit retourner à la fonction appelante, elle pointe vers cette zone mémoire. C'est par exemple ce qui a été implémenté dans les premières machines et le langage de programmation `fortran`. Cependant, cette méthode a un gros inconvénient : on ne peut pas faire d'appel récursif sinon au deuxième appel on écrase (et pour toujours) la valeur stockée dans cette zone mémoire. Pour corriger cela, on utilise la notion de *Pile d'exécution* où on enregistre les adresses des fonctions appelantes avec la propriété du type LIFO (Last In First Out) et ceci correspond bien au comportement souhaité. Le TDA Pile est une formalisation de la propriété LIFO.

Définition du TDA Pile. Le TDA Pile_T utilise les types bool et T et a comme opérations.

$$\begin{aligned}
&\mathsf{creerPile} : () \to \mathsf{Pile_T}\\
&\mathsf{estVidePile} : \mathsf{Pile_T} \to \mathsf{bool}\\
&\mathsf{empiler} : \mathsf{Pile_T}*\mathsf{T} \to \mathsf{Pile_T}\\
&\mathsf{depiler} : \mathsf{Pile_T} \to \mathsf{Pile_T}\\
&\mathsf{sommetPile} : \mathsf{Pile_T} \to \mathsf{T}
\end{aligned}$$

Les axiomes qui définissent le comportement d'une pile sont les suivants (il est prouvé que c'est complet et consistant). Pour toute pile p et tout élément x de type T, on a

$$\begin{aligned}
&\mathsf{estVidePile(creerPile())=True}\\
&\mathsf{estVidePile(empiler(p,x))=False}\\
&\mathsf{sommetPile(empiler(p,x))=x}\\
&\mathsf{depiler(empiler(p,x))=p}
\end{aligned}$$

On peut à l'aide d'un tableau et d'un entier (qui pointe toujours sur le sommet de la pile) proposé une implémentation facile (quelques lignes de code par opérations) du TDA Pile_T. Si le tableau est statique, alors il faut connaître à l'avance la taille maximale de la pile. En TD vous proposerez une implémentation.

Applications. Dans l'introduction de la section, nous avons parlé d'une première utilisation des piles en programmation pour gérer les appels de fonctions. Les piles sont également utilisés dans les compilateurs/interpréteurs pour analyser syntaxiquement un programme. Nous allons voir ici une utilisation des piles pour évaluer des expressions arithmétiques (la même méthode peut être utilisée pour évaluer d'autres expressions).

Pour simplifier, nous allons supposer que notre expression arithmétique ne contient que des chiffres et les 4 opérations $+, *, /$ et $-$ et qu'elles sont toutes binaires (on peut avoir aussi des parenthèses) et qu'elle est en notation infixe. On va gérer la priorité des opérations. On va supposer que l'expression arithmétique est donnée sous forme d'un tableau de caractères, terminés par le caractère $\#$. On va faire l'évaluation en deux étapes.

(1) Dans une première étape, on écrit une fonction qui prend l'expression arithmétique et la transforme en notation polonaise.

(2) Dans la deuxième étape, on écrira une fonction qui prend une expression en notation polonaise et donne le résultat de l'évaluation.

Ici, on va écrire la première étape et vous ferez la deuxième étape en TD.

Exemple 4. *Si on* $(3 * (5 + 2 * 3) + 4) * 2\#$*, le résultat devrait être* 3 5 2 3 $*$ + $*$ 4 + 2 $*$ #*.*

```
void expInfVersPol (char eau[], char eap[]) {
   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)) >= priorite(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 File.** Lorsque l'on gère les accès à une ressource limitée à plusieurs personnes, on gère les priorités d'accès à cette dernière. Par exemple, une imprimante avec mémoire reçoit un ensemble de documents à imprimer et doit les imprimer dans l'ordre d'arrivée, ou encore un système d'exploitation doit accorder à chaque programme du temps machine sans privilégier aucun, etc. Pour implémenter ce genre de comportement, on utilise des *files de priorité* qui se comportent comme

des files d'attente et qui ont la propriété du type FIFO (First In First Out). le TDA File est une formalisation de la propriété FIFO.

Définition du TDA File. Le TDA File_T utilise les types bool et T et a comme opérations.

$$\begin{aligned}
&\mathsf{creerFile} : () \to \mathsf{File_T}\\
&\mathsf{estVideFile} : \mathsf{File_T} \to \mathsf{bool}\\
&\mathsf{enfiler} : \mathsf{File_T}*\mathsf{T} \to \mathsf{File_T}\\
&\mathsf{defiler} : \mathsf{File_T} \to \mathsf{File_T}\\
&\mathsf{teteFile} : \mathsf{File_T} \to \mathsf{T}
\end{aligned}$$

Les axiomes qui définissent le comportement d'une file sont les suivants (il est prouvé que c'est complet et consistant). Pour toute file f et tout élément x de type T, on a

$$\begin{aligned}
&\mathsf{estVideFile(creerFile())=True}\\
&\mathsf{estVideFile(enfiler(f,x))=False}\\
&\mathsf{teteFile(enfiler(f,x))=x}\\
&\mathsf{estVideFile(f)} = \mathsf{False} \Longrightarrow \quad \mathsf{teteFile(enfiler(f,x))} = \mathsf{teteFile(f)}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{defiler(enfiler(creerFile(),x))} = \mathsf{creerFile}\ ()\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{defiler(enfiler(f,x))} = \mathsf{enfiler}\ (\mathsf{defiler(f)},\mathsf{x})
\end{aligned}$$

On peut proposer une implémentation du TDA File_T en utilisant des tableaux et deux entiers, l'un pointant sur la tête et l'autre sur la queue de la file. Comme le tableau est linéaire et peut poser des problèmes de perte de places, il faudra le voir comme un tableau circulaire (*i.e.*, n est considérée égale à 0 si n est la taille du tableau utilisé pour stocker les éléments de la file). Et pour distinguer un tableau plein d'un tableau vide, il faudra laisser une case vide entre la tête et la queue (pourquoi ?).

Applications. On va voir dans le parcours en largeur des graphes une utilisation des files. L'utilisation standard d'une file c'est la file d'attente.

3.6. **TDA Liste.** Les listes ont été inventées par les concepteurs du langage `IPL` (un langage conçu pour des programmes d'IA) et les ont définies comme le type de base du langage. Elles sont aujourd'hui utilisées dans beaucoup de programmes, en particulier pour la gestion de l'espace libre d'une mémoire ou pour implémenter des polynômes.

Définitions du TDA Liste. L'idée des listes c'est au lieu de stocker les éléments d'un ensemble de façon contiguë, on les enchaîne, *i.e.*, chaque élément contient un lien vers le suivant. En effet, lorsque des éléments sont stockés dans un tableau, si on veut supprimer l'élément à la case i, on est obligé de déplacer les éléments suivants vers la gauche et ceci a un coût. Les listes permettent de remédier à ce problème puisqu'avec une liste il suffira de ne plus pointer sur l'élément à supprimer pour le perdre. On peut également aisément insérer un élément à la $j$ème position. Seulement, contrairement aux tableaux, l'accès au $j$ème élément nécessite j itérations (alors que pour un tableau c'est constant). Un autre avantage des listes c'est qu'en mémoire on n'est plus obligé de ranger les éléments de l'ensemble dans un espace contiguë (l'allocation des ressources est facilitée). On peut faire l'union de deux listes en temps constant (pour les tableaux il faut un temps linéaire).

Une cellule_T est un enregistrement constitué de l'information à stocker (de type T) et d'une référence vers un objet de type cellule_T. Le TDA Liste_T utilise les types bool, cellule_T, T et int et on peut citer comme opérations.

valeurCellule : cellule_T → T
suivantCellule : cellule_T → cellule_T
insererSuivantCellule : cellule_T*cellule_T → cellule_T
dernierCellule : cellule_T → bool
creerListe : () → Liste_T
estVideListe : Liste_T → bool
teteListe : Liste_T → cellule_T
insererTeteListe : Liste_T*T → Liste_T
supprimerTeteListe : Liste_T → Liste_T
tailleListe : Liste_T → int
queue : Liste_T → Liste_T
obtenirElement : Liste_T*int → cellule_T
insererElement : Liste_T*T*int → Liste_T
supprimerElement : Liste_T*int → cellule_T

Les axiomes peuvent être trouvés dans par exemple [2]. Il faut noter que comme pour les tableaux, dans une liste il est suffisant de stocker que l'adresse du début de liste.

Implémentations des Listes. On va expliquer certaines techniques pour implémenter les listes (vous écrirez les algorithmes en TD). Lorsque l'on implémente des listes avec les tableaux, on a 2 choix : soit le suivant dans la liste est le suivant dans le tableau (suivant logique = suivant spatial) ou soit ce n'est pas le cas.

Dans le premier cas, on va disposer de deux entiers, LD et LF (pour représenter le début et la fin). A la création, les deux sont initialisés à 0 et on aura une liste vide si $LS \geq LF$. On incrémentera ou décrémentera LD ou LF suivant les opérations. Il faudra faire attention lorsque les bords du tableau sont atteints. Dans ce cas, dans une cellule il n'est pas nécessaire de stocker la référence au suivant puisque l'on sait que c'est celui qui suit dans le tableau.

Dans le deuxième cas, il faudra gérer la liste des cases vides, et contrairement au cas précédent on a besoin de garder dans une cellule le suivant dans la liste. La difficulté ici est la gestion de la mémoire et l'insertion/modification au milieu. On verra plus loin comment gérer l'espace libre.

Une autre manière d'implémenter les listes c'est de le faire avec des *pointeurs*[3]. Une cellule deviendra un pointeur vers un espace de stockage et la référence vers une cellule deviendra un pointeur vers cette cellule. Dans cette configuration, vous n'aurez pas besoin de gérer par vous même la liste des cellules libres (c'est déjà fait par le système). Cependant, il faut faire attention aux libérations des cellules supprimées dans la liste. Lorsque vous supprimez une cellule d'une liste et que vous ne le faîtes pas savoir au système, la cellule est définitivement perdue. Certains systèmes (ou langages comme `Java`) disposent de ce que l'on appelle un *récupérateur de place mémoire* (*garbage collector* en anglais) et dans ces cas vous n'avez pas besoin d'informer le système de la libération d'une cellule. En `C` pour allouer de

3. Un pointeur c'est une variable qui contient l'adresse d'une autre variable.

la mémoire à un pointeur on utilise la fonction malloc et pour libérer la mémoire free.

Extensions du TDA Liste. Une *liste bidirectionnelle* est une liste où chaque cellule est composée de trois champs : l'information à stocker, une référence vers le précédent et une référence vers le suivant. Un des intérêts d'une liste bidirectionnelle c'est que lors de certaines opérations (comme la suppressions) on n'a pas besoin de garder (par une variable temporaire) un lien vers la cellule précédente.

Une *liste circulaire* est une liste où la dernière cellule pointe vers la première cellule. On peut donner n'importe laquelle des cellules comme celle de début de liste. Cependant, comment faire pour déterminer si on a fini de parcourir une liste par exemple ? On peut au début du parcours stocker la cellule par laquelle on a commencé et considérer avoir fini lorsque l'on y revient. Mais que faire si on l'a supprimée ? Une solution serait de créer une cellule (que l'on ne supprimera jamais et donc la liste ne sera jamais vide à proprement parler) et la dernière cellule pointera sur cette cellule particulière.

Applications. Nous allons voir ici deux applications des listes : une concernant la gestion de l'espace mémoire libre et une autre pour la gestion des polynômes à une variable. On verra une autre application dans les graphes.

Lorsque l'on utilise une liste, il doit (généralement) exister un mécanisme qui permet de savoir s'il existe de l'espace libre pour créer une nouvelle cellule. Cette tâche est réalisée à l'aide d'une liste composée des espaces libres. On va appeler cette liste *Libre* (qui pointe sur la première cellule libre). Lorsque l'on veut une cellule libre (que l'on va stocker sur la variable *cell*), il suffit juste d'écrire

```
if estVideListe(Libre) then Overflow
else x :=teteListe(Libre) ; Libre :=supprimerTeteListe(Libre) ;
```

Et lorsque l'on libère une cellule x, il suffit juste d'écrire

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

Comment faire pour construire la liste *Libre* au début ? On peut chaîner toutes les cellules libres dès le début. Cependant, ceci prend du temps et en général nous n'avons pas besoin de tout l'espace libre disponible. En plus, il se peut par exemple qu'il existe des tableaux dynamiques qui coexistent avec les listes et nous ne voulons pas réserver plus de mémoire que nous avons réellement besoin. Une technique consistera à regarder la mémoire comme un tableau et à définir une adresse de base pour la liste *Libre*, une adresse $MaxLibre$ qui délimite les cellules libres et une adresse $tabMin$ qui délimite le début des tableaux dynamiques. Vous proposerez une implémentation de la gestion des cellules libres avec ces données en plus (on n'aura plus besoin de construire la liste des cellules libres au début).

On va maintenant s'intéresser à la gestion des polynômes. Dans le TD 2 vous avez implémenter une méthode de gestion des polynômes à l'aide de tableaux. Cependant vous avez pu remarquer qu'il y avait beaucoup de cases libres (initialisées à 0) pour les polynômes épars (eg $x^{1000}-2$). Une technique assez simple consiste à représenter ces polynômes par des listes où dans chaque cellule l'information à stocker représente un monôme (ici un monôme c'est un réel - le coefficient - et un entier - le degré). On va supposer ici que la liste représentant le polynôme, les monômes sont ordonnés suivant l'ordre croissant des degrés. Alors que les opérations sur les polynômes deviennent difficiles avec les tableaux, avec les listes elles deviennent naturelles. Par exemple, voici un algorithme pour calculer la dérivée d'un polynôme. On va

définir le type Monome.

```
struct Monome = {float coef; int exp;};
```

```
Liste_Monome derivation(Liste_Monome p){
    Liste_Monome pd := creerListe();
    if (estVideListe(p)) then pd;
    else {
        do {
            Monome pm = valeurCellule(p);
            if (pm.exp ≠ 0) then {
                Monome m = {pm.coef, pm.exp-1};
                insererElement (pd,m,tailleListe(pd)+1);
            }
        }while(dernierCellule(p)=false and p :=suivantCellule(p));
    }
    return pd;
}
```

4. Éléments de Complexité

4.1. **Critères d'Évalutation.** Lorsque l'on propose un problème, un algorithme *valide* pour ce problème est un algorithme qui le résout. Formellement, une spécification pour un algorithme ce sont les types des entrées et le traitement qui doit être effectué. Un algorithme sera dit *valide* si pour toute instance valide, l'algorithme fournit le résultat escompté. Comme les ressources sont limitées, la validité d'un algorithme n'est pas toujours le seul critère d'évaluation, on voudrait aussi mesurer sa *qualité*. La *qualité* d'un algorithme valide se mesure suivant plusieurs critères. On peut citer :

(i) la *lisibilité* qui se mesure par la facilité de compréhension et d'analyse. On peut citer comme critères les noms des variables, les commentaires, etc.

(ii) la *granularité* qui consiste en la subdivision en algorithmes réutilisables.

(iii) L'*efficacité* qui est mesurée par rapport aux ressources à notre disposition. Dans un ordinateur nous disposons d'un temps de calcul (puissance du processeur) et de capacités de stockage. Pour mesurer l'efficacité d'un algorithme on mesure : le temps de calcul appelé la *complexité en temps* et l'espace utilisé appelé *complexité en espace*.

La *complexité en espace* mesure le nombre de bits utilisé par l'algorithme et les données. Mais comme les capacités de stockage sont beaucoup plus abondantes et presque "illimitées" on ne s'intéresse souvent qu'à la complexité en temps que l'on va définir ci-dessous.

4.2. **Évaluation du Temps d'Exécution.** Pour mesurer la complexité en temps d'un algorithme on peut juste mesurer le temps d'exécution (après implémentation) dans une machine et si pas assez rapide on augmente les capacités de calcul. D'ailleurs, d'après la loi de Moore tous les 18-24 mois la puissance des processeurs double. Cependant augmenter la puissance de calcul n'est pas toujours suffisant et voici un premier exemple.

Exemple 5. *Si je dispose de ce programme suivant.*

```
int cnp(int n, int p){
    if (p=0 or n = 0) then return 1 ;
    else return cnp(n-1,p-1)+cnp(n-1,p) ;
}
```

Ce programme pour des petites valeurs (inférieurs à 20*) produira un résultat rapidement alors que pour des valeurs de l'ordre de* 100 *vous pourrez attendre toute une vie sans voir le résultat et multiplier par des milliers la puissance de calcul ne permettrait pas d'accélérer le calcul significativement.*

Cet exemple montre que pour mesurer la complexité en temps d'un algorithme, l'exécuter dans une machine n'est pas suffisante puisque l'on ne sera pas capable de décider si c'est l'algorithme qui prend du temps ou c'est juste la machine qui est lente. Il faut alors trouver un autre moyen de mesurer le temps d'exécution. Le moyen que l'on a trouvé c'est de compter le nombre d'instructions. Et ceci est en fait le moyen le plus fiable pour comparer les temps d'exécution des algorithmes. L'exemple suivant en donne une petite idée.

Exemple 6. *J'ai un programme dans une machine qui lorsqu'on lui donne un entier n nécessite n^2 instructions. Lorsque n vaut* 10000*, cet algorithme s'exécute relativement rapidement. Quel est le temps d'exécution pour $n = 100000$? On a $100000^2 = 100 * 10000^2 \approx 2^7 * 10000^2$. Et comme la puissance augmente tous les 2ans environ, il faudrait environ 14ans alors pour que la machine finisse les calculs.*

On remarquera que si la machine donnait le résultat en $n \cdot \log(n)$ instructions, pour $n = 100000$ on aurait besoin de $17 * 100000 = 1700000$ instructions, i.e.*, $100/1.7$ moins de temps.*

Pourquoi mesure-t-on le nombre d'instructions ? En fait les ordinateurs ont un nombre constant d'instructions et chacune de ces instructions nécessite un nombre relativement petit de tours d'horloge (en général 2 à 4 cycles). Avant d'expliquer comment faire le calcul de la complexité, on va introduire certaines notations.

4.3. **Notations O, Ω et Θ.** Si $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$ sont deux fonctions.

(1) On écrira $g(n) = O(f(n))$ si il existe c et n_0 tels que pour tous $n \geq n_0$ on ait $g(n) \leq c \cdot f(n)$. (La fonction g est bornée supérieurement par f à une constante près.)

(2) On écrira $g(n) = \Omega(f(n))$ si il existe c et n_0 tels que pour tous $n \geq n_0$ on ait $g(n) \geq c \cdot f(n)$. (La fonction g est bornée inférieurement par f à une constante près.)

(3) On écrira $g(n) = \Theta(f(n))$ si il existe c_1, c_2 et n_0 tels que pour tous $n \geq n_0$ on ait $c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$. En d'autres termes $g(n) = \Theta(f(n))$ ssi $g(n) = O(f(n))$ et $g(n) = \Omega(f(n))$. (La fonction g est égale à f à des constantes près.)

Exemple 7.

$$3n^2 + 10n = O(n^2)$$
$$5n + 100000 = O(n)$$
$$\frac{1}{2}n^2 - 3n = \Theta(n^2).$$

Voici quelques fonctions standard en notation O.

(1) Une fonction constante est égale à $O(1)$.

(2) Une fonction linéaire est égale à $O(n)$.
(3) Une fonction quadratique est égale à $O(n^2)$.
(4) Une fonction polynomiale de degré k est égale à $O(n^k)$.
(5) $O(a^n)$, $O(n!)$ et $O(n^n)$ sont des fonctions exponentielles (tout algorithme de génération des sous-ensembles d'un ensemble nécessitera au moins $O(2^n)$ instructions, pourquoi ?).
(6) On notera aussi les fonctions étalon $O(n\log(n))$ (tri dans un tableau) et $O(\log(n))$ (recherche dans un tableau trié).

L'arithmétique avec la notation O est assez simple. On prendra toujours comme résultat le terme dominant (*i.e.*, celui qui domine lorsque n tend vers l'infini). Ainsi par exemple $O(1)+O(1)=O(1)$ et $O(n)+O(n^2)=O(n^2)$.

Les notations O, Ω et Θ peuvent être naturellement étendues aux fonctions à plusieurs variables. On laissera au lecteur le soin de donner les extensions.

4.4. **Règles de Calcul.** Il reste assez fastidieux de compter toutes les instructions machines d'un programme. On remarque alors que chaque instruction élémentaire de notre langage utilise un nombre constant d'instructions et par conséquent les compter reviendrait à compter (à des constantes près) les instructions machines. Deuxièmement comme étudier la complexité n'est intéressante que pour des entrées de grande taille, on étudie les performances asymptotiques, *i.e.*, la limite du temps d'exécution lorsque les *tailles* des entrées tendent vers l'infini. Il devient alors naturel d'utiliser la notation O pour mesurer le nombre d'instructions en comptant les instructions élémentaires du langage. On va noter $C(A)$ le nombre (asymptotique) d'instructions lorsque l'on exécute l'algorithme A. On a

$$C(A) = \sum_{\text{I instructions de P}} C(I).$$

On va montrer comment calculer $C(I)$, pour chaque instruction I, inductivement sur la forme des instructions.

(1) Les déclarations de types et de variables (sans affectations) on dira qu'elles nécessitent $O(1)$ instructions. De même les lectures de variables et de valeurs nécessitent également $O(1)$ instructions.
(2) Les expressions arithmétiques nécessitent également $O(1)$ instructions.
(3) Si l'expression I est une affectation $x := exp$, alors la $C(I) = C(exp)$.
(4) Si l'expression I est de la forme $f(exp_1, \ldots, exp_n)$ alors $C(I) = \sum_{1\leq i\leq n} C(exp_i) + C(f)$. (Notons que f peut être un opérateur arithmétique par exemple ou l'appel d'un algorithme).
(5) Si l'expression I est une boucle qui a n itérations et si à chaque itération i on exécute l'expression exp_i, alors $C(I) = \sum_{1\leq i\leq n} C(exp_i)$.
(6) Si l'expression est une structure conditionnelle utilisant les expressions Booléennes $C_1, \ldots, C_n$ et les expressions $exp_1, \ldots, exp_{n+1}$, alors $C(I) = \max_{1\leq i\leq n}\{\sum_{1\leq j\leq i} C(C_j) + C(exp_i)\}$.

Exemple 8.

```
int sommeN(int n){
    int s=0;
    for (int i=1;i≤ n; i++)
        s+=i;
    return s;
}
```

Les lignes 1,2 et 5 comptent chacune $O(1)$. La bouche for *tourne n fois et chaque itération nécessite $O(1)$. Donc $C(sommeN) = O(n)$ lorsqu'on lui donne un entier n. Si on prend l'algorithme suivant.*

```
int sommeNB(int n){
    return n*(n+1)/2;
}
```

On vérifiera que $C(sommeNB) = O(1)$ pour toute entrée n (supposant que la lecture de n est en $O(1)$).

On voit que $C(sommeN)$ ne donne pas la même valeur pour des n différents. Pour exprimer que le nombre d'opérations dépend des entrées, on exprime la complexité comme une fonction (en utilisant toujours la notation O) des entrées. Si un algorithme A prend n paramètres, on écrira $C_A(t_1, t_2, \ldots, t_n)$ avec t_i qui représente la taille du paramètre i (en général c'est un entier). Par exemple pour les fonctions sommeN et sommeNB, on aura $t_1 = n$, l'entier pris en entrée. Pour des paramètres de tableaux on prend souvent le nombre d'éléments du tableau, pour une liste la taille de la liste, ...

4.5. **P vs NP.** On va définir les ensembles P et NP que l'on appelle *classes de complexité*.

Classification. Un problème est dit *polynomial* (ou dans P) s'il existe un algorithme qui le résout et qui s'exécute en temps polynomial (par rapport aux entrées). On peut citer les problèmes de tri d'un ensemble, recherche dans un ensemble, l'évaluation d'une expression arithmétique, ...

Un problème est dit dans NP s'il existe un algorithme A qui s'exécute en temps polynomial et qui prend en entrée les solutions possibles du problème et décide si l'entrée donnée est une solution au problème ou pas. On appelle les problèmes dans NP les problèmes à *vérification polynomiale*.

Proposition 1. *Tout problème dans* P *est dans* NP.

Démonstration. Soit P un problème et A un algorithme le résolvant en temps polynomial. On exécute A et on stocke la solution. L'algorithme de vérification consistera à comparer toute entrée à la solution calculée. □

Il existe d'autres problèmes dans NP. On peut citer la satisfaisabilité d'un circuit, l'existence d'un chemin passant une et une seule fois par chaque ville, le problème du sac à dos, ...

Parmi les problèmes dans NP, il existe ceux que l'on appelle *NP-complet*. On dira qu'un problème P dans NP qu'il est *NP-complet* si pour tout problème P' dans NP, il existe un algorithme A qui s'exécute en temps polynomial et pour toute entrée I' de P' construit une entrée I pour P telle que I' est solution de P' ssi I est solution de P.

Conjecture à 1million$. Pour tout problème *NP-complet*, il n'existe pas d'algorithme polynomial pour le résoudre.

Conséquence. Il existe des problèmes dans NP qui ne sont pas *NP-complet* et qui ne peuvent pas être résolus par un algorithme polynomial.

Robustesse des classes. Un algorithme résout un problème et les instances du problème doivent être représentées sous une forme compréhensible par l'algorithme. Par exemple un entier n on peut le représenter par une suite de n bâtonnets (notation unaire) ou par sa représentation binaire ; un ensemble on peut le représenter par un tableau ou une liste.

Comme on exprime les complexités en fonction des entrées (leurs tailles pour être exact), on peut se demander si la représentation peut faire passer un algorithme de polynomial à exponentiel par exemple. Comme les machines sont binaires, on décrira chaque représentation d'un ensemble d'instances S par une fonction $e : S \to \{0,1\}^*$ (qui est en fait la suite de bits pour représenter chaque élément de S) que l'on appelle *encodage*. Si on prend la fonction sommeN, on remarque :

— Si l'entier n est représenté en unaire, alors on aura bien $C_{\mathsf{sommeN}}(n) = n$, qui est linéaire en la taille de l'entrée.
— Par contre, si n est représenté en binaire, alors la taille de l'entrée c'est $k := \log(n)$ et donc on aura $C_{\mathsf{sommeN}}(n) = O(2^k)$ qui est exponentiel en la taille de l'entrée.

L'encodage est donc très important. On va supposer que nos encodages vérifient certaines propriétés, en particulier :

— les entiers ne sont pas représentés en unaire,
— les instances sont codées de façon raisonnable et concise (par exemple un ensemble avec n éléments nécessitera $n *$ (taille d'un élément)).

Avec ces contraintes sur les encodages, nous avons la proposition suivante qui affirme que les classes de complexité P et NP sont robustes.

Proposition 2. *L'encodage ne change pas la classe de complexité d'un problème.*

De nos jours nous disposons de machines à plusieurs cœurs, des clusters à haute capacité de calcul. Cependant, les classes de complexité P et NP restent également les mêmes. En effet un problème *NP-complet* le reste même si on multiplie par des milliers le nombre de cœurs.

Que faire face aux problèmes *NP-complet* ? Il existe plusieurs techniques pour contourner la difficulté de trouver un algorithme efficace pour les problèmes *NP-complet*.

(1) On restreint les instances à des familles particulières et en général on arrive à montrer qu'avec ces restrictions on peut avoir un algorithme polynomial.

(2) On propose des *algorithmes d'approximation* qui donnent une solution approchée aux problèmes.

(3) On propose des *algorithmes randomisés* et on montre que dans la plupart des cas l'algorithme tourne en temps polynomial. Pour cela, on mesure la moyenne du nombre d'instructions sur toutes les instances possibles.

(4) On propose des *algorithmes heuristiques* qui permettent dans la plupart des cas de donner des solutions dites acceptables et qui tournent en temps polynomial.

(5) On donne des *algorithmes exponentiels* avec le temps d'exécution le plus petit possible. Par exemple un algorithme naïf donnerait une complexité de $O(3^n)$, on propose un algorithme plus abouti qui tournerai en $O(2^{\sqrt{n}})$.

(6) ...

5. Types Inductifs

Une définition inductive d'un ensemble X consiste à définir une manière de construire les éléments de X à partir d'autres éléments. Pour cela on donne explicitement un certain nombre d'éléments de X, appelés les *éléments de base*, et des opérations qui permettent de construire les autres éléments.

Definition 2. *Une* définition inductive *d'un ensemble E consiste en la donnée*

(1) d'un sous-ensemble fini B de X,

(2) d'un ensemble K d'opérations $\phi : X^{ar(\phi)} \to X$ où $ar(\phi)$ est l'arité de ϕ.

En fait X est le plus petit ensemble tel que

(B) $B \subseteq X$,

(I) Pour tout $\phi \in K$ et tous $x_1, \ldots, x_{ar(\phi)}$, on a $\phi(x_1, \ldots, x_{ar(\phi)}) \in X$.

Tout ensemble inductif sera défini par la donnée de B et des opérations de K.

Nous allons maintenant donner quelques exemples d'ensembles inductifs.

Exemple 9. *Si on pose $B := \{0\}$ et on pose l'opération suc : $n \mapsto n + 1$ pour tout $n \in X$, alors on peut prouver que X est exactement l'ensemble des entiers naturels. Tout entier correspond au nombre de fois que l'on a appliqué suc sur 0.*

Exemple 10. *Soit A un alphabet et A^* l'ensemble des mots finis sur l'alphabet A. Pour tout $a \in A$, on pose $\phi_a : u \mapsto ua$ pour tout $u \in A^*$. En posant $K := \{\phi_a \mid a \in A\}$ et $B := \{\varepsilon\}$, on peut vérifier que l'on obtient une définition inductive de A^*.*

Exemple 11. *Soit $A := \{(,)\}$. Soit $D \subseteq A^*$ l'ensemble des mots définis inductivement par $\varepsilon \in D$ et les deux opérations $\phi_1 : x \mapsto (x)$ et $\phi_2 : (x, y) \mapsto xy$. L'ensemble D est appelé l'ensemble des mots de Dyck, qui correspond à l'ensemble des parenthèsages bien formés.*

Exemple 12. *Soit A un alphabet. On note par $AB \subseteq \{A, (,)\}^*$ l'ensemble défini inductivement par $\varepsilon \in AB$ et pour tout $a \in A$, on définit $\phi_a : (g, d) \mapsto (a, g, d)$. L'ensemble AB est appelé l'ensemble des arbres binaires (avec nøeuds étiquetés sur A).*

Si on pose par exemple $A = \{a, b, c, d\}$, alors $(a, \varepsilon, \varepsilon)$, $(a, (b, \varepsilon, \varepsilon), (a, \varepsilon, \varepsilon))$, et $(d, (a, (b, \varepsilon, \varepsilon), (a, \varepsilon, \varepsilon)), (c, \varepsilon, \varepsilon))$ sont des arbres binaires sur A.

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

(i) tout élément de B vérifie la propriété P (en général c'est facile),

(ii) Pour toute fonction ϕ dans K d'arité n et tous éléments $x_1, \ldots, x_r$ vérifiant la propriété P, alors $\phi(x_1, \ldots, x_r)$ vérifie également P.

Nous allons donner dans ce chapitre des TDA définis inductivement. Tout d'abord rappelons que l'on en a déjà rencontré un : les listes.

Exemple 13 (TDA Liste)**.** *Nous rappelons qu'une liste chaînée c'est un ensemble d'éléments chaînés les uns à la suite des autres. Une façon usuelle de définir une liste chaînée (c'est d'ailleurs comme cela que c'est défini en `Ocaml` ou `Lisp` et même certaines implémentations en `C`) est de manière inductive. Un objet de type `Liste_T` c'est soit une liste vide (en général noté $[]$), soit une paire $a :: xs$ où xs est un objet de type `Liste_T` et a un objet de type `T`.*

5.1. **Arborescences.** Une *arborescence* c'est un ensemble V muni d'un sommet distingué $r \in V$ appelé la *racine*, et d'une relation binaire E telle que pour tout x, excepté la racine, il existe exactement un unique y, noté $pere(x)$, tel que $(y, x) \in E$. On le notera (V, E, r). On définit les arborescences de manière inductive de la sorte :

(A.B) tout singleton r est une arborescence,

(A.I) Si $T_1, \ldots, T_p$ sont des arborescences de racines $r_1, \ldots, r_p$, alors (V, E, r) est une arborescence où $V := r \cup \bigcup_{1 \leq i \leq p} V(T_i)$ et $E := \bigcup_{1 \leq i \leq p} \{(r, r_i)\} \cup \bigcup_{1 \leq i \leq p} E(T_i)$.

Les éléments de V sont appelés *noeuds* et ceux de E *arcs*. Pour un noeud x les noeuds $\{y \mid (x, y) \in E\}$ sont appelés les fils de x et un noeud sans fils est appelé *feuille*. Un chemin de taille k c'est une suite finie $(x_1, \ldots, x_k, x_{k+1})$ telle que $(x_i, x_{i+1}) \in E$ pour tout $1 \leq i \leq k$. Si x est un noeud, on note T_x la *sous-arborescence* issue de x constituée de tous les noeuds accessibles depuis x par un chemin. La *hauteur* d'une arborescence T, notée $h(T)$, c'est la longueur maximale d'un chemin depuis la racine. On note souvent n la taille de V. Voici quelques propriétés des arborescences.

Proposition 3. *Soit (V, E, r) une arborescence.*

(1) Pour tout noeud x, il existe un unique chemin de r à x.

(2) Une arborescence contient au moins une feuille.

(3) $h(T) = 1 + \max\{h(T_{r_i}) \mid r_i$ fils de $r\}$.

(4) $|E| = |V| - 1$.

Implémentation. Si on a une arborescence où les noeuds sont numérotés 0 à $n-1$, on peut le représenter par un tableau PERE de taille $n-1$, dit *tableau de Prüfer*, où PERE[i] contient $pere(i)$ (pour la racine on met -1). Cette représentation permet de montrer qu'il y a exactment n^{n-1} arborescences avec n noeuds (en utilisant la formule de Cayley). Cette représentation est intéressante lorsque l'on connaît à l'avance la taille de l'arborescence.

Lorsque l'on ne connaît pas la taille de l'arborescence ou lorsqu'elle évolue dans le temps, on peut préférer une représentation en utilisant des listes. On peut par exemple définir un type cellule qui représentera un noeud de la manière suivante.

```
struct cellule_T = { T info; cellule_T pere; cellule_T filsG; cellule_T frereD;};
```

Cette représentation utilise $O(n)$ espaces. Une arborescence sera tout simplement une référence vers le noeud représentant la racine.

Il existe plusieurs autres implémentations. On peut par exemple pour chaque noeud ne garder que son père et ne pas garder de liens vers les fils (on gardera la liste des noeuds et la racine sera le noeud sans père). La représentation dépendra toujours de l'application.

Arbre. Un *arbre* est une arborescence où les fils sont ordonnés. Ainsi, on distinguera le premier fils du deuxième fils, etc. Dans une implémentation des arbres, on doit pouvoir identifier les fils (par exemple dans notre représentation ci-dessus filsG devrait pointer sur le premier fils et si x est le i^{eme} fils de y, alors x.frereD devrait pointer sur le $(i+1)^{\text{eme}}$.

Definition 3 (Arbres Binaires)**.** *Un* arbre binaire *est un arbre où chaque noeud a au plus* 2 *fils. Il est appelé* arbre binaire complet *si chaque noeud a exactement* 0 *ou* 2 *fils et si en plus toutes les feuilles sont exactement à la même hauteur (*i.e.*, même distance depuis la racine) on l'appelera alors* arbre binaire parfait*. Un* arbre binaire quasi-parfait *est un arbre binaire où tous les niveaux, excepté le dernier, sont totalement remplis, et le dernier niveau est rempli en partant de la gauche et jusqu'à un certain point.*

Dans un arbre binaire, on a la hauteur h qui vérifie les inégalités suivantes $\lfloor \log(n) \rfloor \leq h \leq n-1$. On a aussi $n \leq 2^{h+1}-1$ et $f \leq 2^h$ où f est le nombre de feuilles. Dans un arbre binaire complet, on a $n = 2f-1$ et dans un arbre binaire parfait on a $n = 2^{h+1}-1$. Dans un arbre binaire quasi-parfait on a $h = O(\log(n))$.

Dans un arbre binaire, on dispose des primitives suivantes.

$$\begin{aligned} &\mathsf{racine} : \mathsf{Arbre\_T} \to \mathsf{cellule\_T} \\ &\mathsf{arbreG} : \mathsf{Arbre\_T} \to \mathsf{Arbre\_T} \\ &\mathsf{arbreD} : \mathsf{Arbre\_T} \to \mathsf{Arbre\_T}. \end{aligned}$$

On notera $\mathsf{filsG}(T)$ et $\mathsf{filsD}(T)$ les racines de $\mathsf{arbreG}(T)$ et $\mathsf{arbreD}(T)$ respectively.

Definition 4 (Parcours)**.** *Le parcours d'un arbre consiste à visiter les noeuds de l'arbre (exactement une fois chaque noeud) dans un certain ordre.*

Le parcours en profondeur *consiste à visiter la racine et ensuite visiter chaque sous-arbre (issu de la racine) dans l'ordre.*

Le parcours en largeur *consiste à visiter les noeuds de l'arbre niveau par niveau (*i.e.*, on visite les noeuds à distance i de la racine avant de visiter ceux qui se trouvent à distance $j > i$).*

Le mot préfixe *d'un arbre est une permutation des noeuds telle que le noeud x précède le noeud y dans la permutation si x est ancêtre de y ou s'il existe trois noeuds z, x' et y' tels que x' est ancêtre de x, y' celui de y, z est père de x' et y' et $x' \preceq y'$. (On lit l'information pour la première fois.)*

Le mot suffixe *d'un arbre est une permutation des noeuds telle que le noeud x précède le noeud y dans la permutation si y est ancêtre de x ou s'il existe trois noeuds z, x' et y' tels que x' est ancêtre de x, y' celui de y, z est père de x' et y' et $x' \preceq y'$. (On lit l'information pour la dernière fois.)*

Le mot infixe *d'un arbre binaire est une permutation des noeuds telle que tout noeud x est entre ces fils gauche et droit. (On lit l'information pour la deuxième fois.)*

Proposition 4. *Le mot préfixe tout seul ne permet pas de reconstruire un arbre. De même pour le mot suffixe. Cependant le couple (mot préfixe, mot suffixe) permet de reconstruire un arbre.*

Application. Les arbres sont utilisés par exemple pour représenter le système de fichiers d'un disque.

On peut utiliser les arbres pour analyser la complexité en temps d'un algorithme récursif. Pour cela, les noeuds représenteront les appels de fonction et la relation de parenté sera appelé-appelant. Ceci permet d'obtenir une équation récursive pour calculer la complexité d'une fonction récursive. Ceci est également utile pour dérécursiver un algorithme récursif (comment ?).

On peut également utiliser les arbres pour décider de la valeur de vérité d'une expression booléenne par exemple. Pour cela, on utilise l'arbre des possibilités. Le même principe peut être utilisé en théorie des jeux (jeu d'échec, sudoku, ...). Les arbres sont également utilisés pour représenter l'arbre généalogique d'une famille ou tout simplement pour représenter l'évolution d'une espèce en biologie. Nous allons voir ci-dessous d'autres utilisations des arbres.

5.2. **Tas.** Un *tas binaire* est un arbre binaire quasi-parfait. Les tas binaires sont utilisés pour stocker de l'information et pouvoir y faire des requêtes du genre "obtenir le min" ou "obtenir le maximum". Un *tas-min* est un tas binaire où pour chaque noeud x on a $info(pere(x)) \leq info(x)$ et un *tas-max* est un tas binaire où pour chaque noeud x on a $info(pere(x)) \geq info(x)$. Dans un tas-min (ou tas-max) le minimum (ou maximum) se trouve toujours à la racine. Ainsi, on peut utiliser un

tas binaire pour trier un ensemble et en effet cela permet de trier n éléments en $O(n \cdot \log(n))$. Voici les primitives d'un tas-max : construire-tas, inserer, entasser-max, tri-tas, maximum, retirer-maximum, augmenter-cle.

La construction d'un tas consiste à inserer un à un les éléments de manière à garder la structure de tas. Remplacer la clé d'un élément consiste à lui donner une nouvelle valeur plus grande et à réorganiser le tas. Insérer un élément consiste à l'insérer à la bonne place de manière à garder la structure de tas. La fonction entasser-max consiste à réorganiser de manière à avoir une structure de tas. On va écrire la fonction entasser-max et vous vous en inspirerez pour écrire les autres primitives. On va utiliser un arbre binaire pour représenter les tas.

```
tas entasser-max (tas t, cellule r) {
    fg = filsG(r);
    fd = filsD(r);
    if (fg != NULL and valeur(fg) >= valeur (r))
       max = fg;
    else
       max = r;
    if (fd != NULL and valeur(fd) >= valeur(max))
       max = fd;
    if (max != r){
      echanger-donnees(max,r);
      entasser-max(t,max);
   }
   return t;
}
```

Pour augmenter la clé d'un noeud il faut remplacer la clé par la nouvelle, et pour respecter la propriété de tas-max tant que la valeur du noeud courant est plus grande que celle de son père, permuter les deux valeurs et considérer le noeud père comme le noeud courant.

Pour insérer dans un tax-max il faut d'abord insérer comme dans un arbre binaire quasi-parfait, *i.e.*, chercher le dernier niveau et insérer dans ce dernier (en remplissant à droite que si le niveau gauche est totalement rempli) et y mettre disons la valeur $-\infty$. Ensuite, augmenter la valeur avec la méthode augmenter-cle.

Pour retirer le maximum, il faut permuter les valeurs du dernier noeud à être insérer et de la racine, supprimer le dernier noeud et ensuite appeler entasser-max sur la racine pour obtenir un tas. Il faut remarquer que les sous-arbres gauche et droit de la racine après échange des données sont toujours des tas et du coup on peut utiliser entasser-max.

Voici l'algorithme du tri par tas.

```
T[] tri-tas (T tab[n]){
   t = construire-tas(tab);
   for (i=n-1; i>=0; i--) {
      echanger (tab[i], maximum(t));
      retier-maximum (t);
   }
   return tab;
}
```

Application : Files de priorité. Une *file de priorité* est une structure de données qui permet de gérer un ensemble S d'éléments où chacun a une valeur appelée *clé* qui permet de gérer les priorités. Dans une file de priorité max par exemple, on a

besoin des primitives suivantes (on supposera que le TDA est nommé FileP_T :

inserer : FileP_T*T → FileP_T
maximum : FileP_T → T
extraire-max : FileP_T → FileP_T
remplacer-cle : FileP_T*T*T' → FileP_T.

Les files de priorité sont utilisées pour gérer par exemple les pools d'impression ou en général pour planifier des tâches (gestion des processus, des accès aux ressources, ...). La structure de tas-max (plutôt qu'une file) est une structure de données efficace pour gérer une file de priorité. En effet, en utilisant un tas on peut

— insérer en temps $O(\log(n))$ un élément dans la file de priorité (il faut garder la structure de tas),
— obtenir le maximum en temps $O(1)$ en lisant la racine,
— extraire le maximum en temps $O(\log(n))$ (supprimer la racine et réorganiser la structure de tas),
— pour remplacer une clé, on met à jour et et ensuite on réorganise si nécessaire (en $O(\log(n))$).

5.3. **Arbres de Recherche.** Un *arbre de recherche* est une structure de données pouvant supporter un ensemble d'opérations dynamique. Les plus importantes sont les suivantes : rechercher, minimum, maximum, predecesseur, successeur, inserer et supprimer. L'objectif c'est que toutes les opérations se fassent relativement rapidement et en particulier la recherche qui est une opération souvent effectuée (par exemple dans une bibliothèque ou dans un dictionnaire on construit la base au début et après la plupart des requêtes sont recherches).

Nous avons vu que la structure de tas-max (ou min) est une bonne structure de données pour récupérer le maximum (ou le minimum) en temps $O(1)$, mais la recherche est relativement lente (elle peut être linéaire) et toutes les autres opérations peuvent être égalent coûteuses. On peut utiliser des structures de données plus adaptées.

Arbre binaire de recherche. Un *arbre binaire de recherche* (ABR) est un arbre binaire où pour chaque noeud x on a la propriété suivante : si x_1 et x_2 sont ses fils, alors pour tout $y \in T_{x_1}$ et $z \in T_{x_2}$,

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

Ainsi, lorsque l'on a un ABR un parcours infixe permet d'avoir une permutation triée. On va maintenant expliquer les implémentations des opérations (h est la hauteur de l'arbre).

minimum: C'est la valeur stockée dans le noeud le plus à gauche ($O(h)$).

maximum: C'est la valeur stockée dans le noeud le plus à droite ($O(h)$).

predecesseur: C'est la valeur maximum dans le sous-arbre gauche ($O(h)$).

successeur: C'est la valeur minimum dans le sous-arbre droit ($O(h)$).

rechercher: Il faut comparer avec la racine, si c'est plus petit faire la recherche dans le sous-arbre gauche, sinon faire la recherche dans le sous-arbre droit ($O(h)$).

inserer: Faire une recherche du noeud pere et ensuite faire l'insertion ($O(h)$). (On s'arrête lorsque l'on a trouvé NULL.)

supprimer: Faire une recherche du noeud à supprimer et ensuite réorganiser (trouver un pere à ses sous-arbres) ($O(h)$).

Toutes les opérations s'exécutent en temps linéaire sur la hauteur. Cependant, si la hauteur est linéaire en n toutes les opérations sont relativement coûteuses. Donc un ABR n'est efficace que si on peut s'arranger à la construction et à chaque modification de garder un ABR avec une hauteur logarithmique. Cependant, ceci n'est pas toujours possible. Par contre, on a ce théorème qui dit qu'en moyenne un ABR a une hauteur logarithmique.

Theorem 3. *La hauteur moyenne d'un ABR construit aléatoirement avec n clés a une hauteur de* $O(\log(n))$.

Dans un ABR seule la suppression est compliquée. En effet, lorsque l'on veuille supprimer un noeud z,

(i) Si z est une feuille, alors le supprimer et ne rien faire d'autres.

(ii) Si z a un seul fils, alors mettre ce fils comme celui remplaçant z dans $pere(z)$.

(iii) Si z a deux fils, alors echanger-donnees z et $suc(z)$ et supprimer $suc(z)$.

Arbre rouge noir. Un *arbre rouge noir* est un ABR où on garantit qu'il y a toujours une hauteur logarithmique. Pour cela, on stocke dans chaque noeud un bit de couleur : *rouge* ou *noir*. Un ABR est un *arbre rouge noir* si

(1) chaque noeud est soit rouge, soit noir,

(2) la racine est noire,

(3) si un noeud est rouge, alors ses deux enfants sont noirs,

(4) tous les chemins depuis n'importe quel noeud vers n'importe quelle feuille contient le même nombre de noeuds noirs.

Nous noterons $bh(x)$ le nombre de noeuds noirs depuis un noeud x vers une feuille (par définition c'est le même quelque soit le choix de la feuille).

Lemma 1. *Un arbre rouge noir ayant n noeuds internes a une hauteur au plus égale à* $2 \cdot \log(n+1)$.

Démonstration. Soit T un arbre rouge noir et x un noeud. On montre par récurrence sur la hauteur de x que le nombre de noeuds de $T_x \geq$ à $2^{hb(x)} - 1$.

Notons h la hauteur de T. D'après la propriété (3) un chemin depuis la racine jusqu'à une feuille doit contenir au moins $h/2$ noeuds noirs (hormis la racine). Par ce qui précède on a $n \geq 2^{h/2} - 1$. Ainsi, on a $h \leq 2 \cdot \log(n+1)$. □

Par le lemme 1 les opérations minimum, maximum, rechercher, predecesseur et successeur s'exécutent en temps $O(\log(n))$. On va montrer que l'insertion et la suppression peuvent être implémentées en temps $O(\log(n))$. On va utiliser l'opération *rotation*.

La primitive rotation-gauche est une primitive qui prend un ABR T, un noeud x tel que le sous-arbre enraciné en x est de la forme $(x, \alpha, (y, \beta, \gamma))$ et qui construit l'ABR $(y, (x, \alpha, \beta), \gamma)$. La primitive rotation-doite est l'opération symmétrique, *i.e.*, on passe de $(y, (x, \alpha, \beta), \gamma)$ à $(x, \alpha, (y, \beta, \gamma))$. Ces deux primitives qui consistent à modifier des références peuvent être implémentées en $O(1)$.

L'insertion dans un arbre rouge noir peut se faire en deux étapes :

(i.1) insérer l'élément comme dans un ABR et le marquer rouge.

(i.2) réorganiser en arbre rouge noir (on peut violer la condition (3)).

On va expliquer comment réorganiser en arbre rouge noir. En fait ce qui se passe c'est que le nombre de noeuds noirs (et la propriété (3)) dans un chemin permet de dire qu'un arbre rouge noir est équilibré lorsque le père du noeud inséré est noir (pourquoi ?). Lorsque le père du noeud inséré est rouge, alors insérer l'élément

peut ne pas garantir la hauteur $O(\log(n))$. On va modifier pour que ça soit le cas. Notons z le noeud inséré et posons $gp = pere[pere[z]]$. On a deux cas symétriques : $pere[z] = filsG(gp)$ ou $pere[z] = filsD(gp)$. Supposons le premier cas, notons $y = filsD(gp)$. On a 3 sous cas en supposant que gp existe (s'il n'existe pas on colorie la racine en noir et on peut vérifier que toutes les conditions sont vérifiées).

Cas 1.1: y est rouge. Alors, comme gp est noir, on colorie y et $pere[z]$ en noir et on colorie gp en rouge. Comme on peut violer la condition (3) entre gp et $pere[gp]$, on recommence en posant $z := gp$.

Cas 1.2: y est noir. Alors si $z = filsD(pere[z])$, alors on fait une rotation-gauche de $pere[z]$ de telle sorte que $pere[z]$ devienne le fils gauche de z et on remplace z par $pere[z]$ pour se réduire au Cas 1.3.

Cas 1.3: y est noir et $z = filsG(pere[z])$. Alors on colorie $pere[z]$ en noir, $pere[pere[z]]$ en rouge et on fait une rotation-droite de $pere[pere[z]]$. L'algorithme va s'arrêter car vous remarquerez que toutes les conditions sont vérifiées.

Dans tous les cas, à la fin on colorie la racine en noir si ce n'est pas fait. Lorsque $pere[z] = filsD(gp)$, on a les mêmes sous cas en remplaçant gauche par droite et inversement. On peut montrer que ces opérations à la fin nous garantissent la propriété rouge noir et donc on a une hauteur logarithmique.

La suppression dans un arbre rouge noir se fait de la même manière que la suppression dans un ABR, sauf qu'à la fin on peut être amené à corriger l'arbre pour qu'il respecte les conditions d'un arbre rouge noir. On rappelle que si z est une feuille ou a un seul fils, alors le noeud supprimé c'est z, mais lorsque z a deux fils alors ce sera son successeur $suc(z)$ que l'on supprimera et on copiera les données satellites de $suc(z)$ dans le noeud représentant z. Notons donc y le noeud réellement supprimé. Si y est rouge, on peut vérifier que (1) aucune hauteur noire n'est modifiée (2) il n'y a pas eu d'apparitions de noeuds rouges (3) y ne peut pas être la racine qui est noire. Donc toutes les propriétés des arbres rouge noir sont vérifiées. Maintenant, on va supposer que y est noir et du coup on peut violer les conditions des arbres rouge noir et dans ce cas il faudra faire une correction. On va considérer que toutes les feuilles ont des fils, tous égaux à $NULL$ et avec la couleur noire.

(1) Si y est une feuille, alors on supprimera y et on notera x le noeud $NULL$ gauche.

(2) Si y a un seul fils, alors on supprimera y et on notera x le fils qui remplacera y dans $pere[y]$.

Notons maintenant w le frère de x. On va boucler tant que $x \neq racine$ et $couleur[x] = noire$. On a deux cas : soit $x = filsG(pere(x))$ ou $x = filsD(pere(x))$. Supposons $x = filsG(pere(x))$, alors on distingue quatre sous cas à chaque itération. Le but c'est de faire remonter la couleur noire vers le haut jusqu'à arriver à la racine en faisant soit des rotations ou en permutant des couleurs.

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 frère de x, qui était un enfant de w, est maintenant noir. On ramène ce cas à l'un des 3 cas suivants.

Cas 2: w est noir et ses deux enfants sont noirs. On recolorie w en rouge et comme on a peut-être modifié une propriété des arbres rouge noir, à la prochaine itération x devient $pere(x)$. On remonte vers la racine.

Cas 3: w est noir et l'enfant gauche est rouge, l'enfant droite est noire. On permute les couleurs de w et de son enfant gauche et ensuite on fait une rotation droite sur w. On se ramène au Cas 4.

Cas 4: w est noir et son enfant droite rouge. Le noeud w reçoit la couleur de $pere[x]$, et $pere[x]$ et $filsD(w)$ deviennent tous les deux noirs. Ensuite on fait une rotation gauche sur $pere(x)$ et on arrête la boucle.

A la fin on colorie x en noir. Si $x = filsD(pere(x))$, c'est la même chose en remplaçant gauche par droite et inversement. On peut montrer que ces opérations à la fin nous garantissent la propriété rouge noir et donc on a une hauteur logarithmique.

Autres arbres de recherche. Il existe d'autres structures de données représentant des arbres de recherche. On peut citer.

- Les B-arbres. Ce sont des arbres équilibrés conçus pour être efficaces sur des disques magnétiques ou autres périphériques secondaires de stockage à accès direct (pour minimiser le temps d'attente du déplacement de la tête de lecture). A la différence d'un arbre rouge noir, un noeud dans un B-arbre peut avoir plusieurs fils.
- Les tas fusionables. On peut citer les tas binomiaux et les tas de fibonacci. Ces tas ont été inventés lorsque l'on a voulu faire l'union de tas de manière efficace.
- Les AVL. Ce sont des ABR équilibrés, *i.e.*, les hauteurs des sous-arbres gauches et droits diffèrent de 1 au plus.

5.4. **Applications.** Ici on va montrer l'utilisation des arbres dans la gestion de sous-ensembles d'un ensemble.

Gestion partition. Supposons que l'on dispose d'un ensemble de villes et petit à petit un ensemble de routes commerciales sont mises en place. Ainsi à chaque moment les routes commerciales définissent une partition des villes : chaque partie contient un ensemble de villes reliées directement (par une route) ou par un chemin. Au début chaque ville formait une seule classe. On voudrait à tout moment savoir si entre deux villes il existe une route commerciale. On voudrait également lorsqu'une nouvelle route commerciale est mise en place, fusionner les classes contenant les deux villes en question. La fonctionnalité désirée ici est la gestion des ensembles disjoints ou d'une partition d'un ensemble. On rappelle qu'une partition d'un ensemble E est une collection finie $\{E_1, \ldots, E_p\}$ de sous-ensembles de E tels que pour tout $i \neq j$ on ait $E_i \cap E_j = \emptyset$.

Le TDA EnsembleDisjoint_T utilise les types T et a comme opérations.

creerEnsemble : T → EnsembleDisjoint_T

union : T*T → EnsembleDisjoint_T

trouver : EnsembleDisjoint_T*T → T

EN TD, faire la gestion des ensembles disjoints par une liste et par une matrice. Quels sont les inconvénients ? En fait on peut montrer ce théorème.

Theorem 4. *Avec la représentation par liste chaînée des ensembles disjoints et en supposant qu'à chaque union c'est le représentant de la liste la plus longue qui devient le représentant de la nouvelle liste, une séquence de m opérations parmi lesquelles on a n opérations creerEnsemble consomme un temps $O(m + n \cdot \log(n))$.*

En utilisant des arbres par contre on peut avoir un meilleur temps. Chaque ensemble est un arbre (la racine étant le représentant) et l'union consistera à fusionner deux arbres. On va également toujours considérer l'arbre avec plus de noeuds comme celui gardant son représentant. L'union consistera à toujours lier les racines. On peut l'implémenter de manière à ce que chaque arbre ait toujours une

hauteur logarithmique (il faudra compresser les chemins). On obtiendra ainsi un temps moyen de $O(m \cdot \alpha(n))$ où $\alpha(n) := \min\{k \mid A_k(1) \geq n\}$ et

$$A_k(j) := \begin{cases} j+1 & \text{si } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{sinon.} \end{cases}$$

(On $f^0(x) = x$ et $f^{(i)}(x) = f(f^{(i-1)}(x))$ si $i \geq 1$). On a $\alpha(n) \leq 4$ pour $n \leq A_4(1) = 16^{512} \gg 10^{80}$. Voici une implémentation possible.

```
creerEnsemble (x) {
   p(x) = x;
   rang(x) = 0;
}

union (x,y) {
  lier (trouver(x), trouver(y));
}

lier (x,y) {
  if (rang(x) > rang(y)) p(y) = x;
  else {
     p(x) = y;
     if (rang(x) = rang(y)) rang(y) = rang(y)+1;
  }
}

trouver (x) {
  if (x!= p(x))
    p(x) = trouver(p(x));
  return p(x);
}
```

Dictionnaire. Un dictionnaire est un ensemble de mots ordonnés avec peut-être des données satellites (par exemple les significations des mots dans un dictionnaire classique ou la translation dans un dictionnaire entre deux langues). Je rappelle que le TDA Dico_T utilise les types bool et T, et a comme opérations

creerDico : () → Dico_T
insererDico : Dico_T*T → Dico_T
supprimerDico : Dico_T*T → Dico_T
appartenanceDico : Dico_T*T → bool

Pour gérer un dictionnaire on peut utiliser un arbre. Lequel préconiseriez-vous ?

6. Tables de Hashage

Le problème principal est le suivant : Comment associer un indice à un objet informatique ? Par exemple, vous voulez manipuler un annuaire téléphonique A avec pour primitives : Recherche, Ajout et Suppression. Le défi c'est un accés constant à une personne de l'annuaire. Comment faire ? On peut associer à chaque entrée un entier et tel que deux entrées aient des indices différents. L'accés est constant, mais malheureusement la taille du tableau sera souvent beaucoup trop grand par rapport à la taille de l'annuaire réellement stocké. En effet, si chaque nom est limité par exemple à c caractères, alors la taille du tableau serait de 26^c (si $c = 10$, on aurait quelque chose comme 10^{13}). Une telle table est appelée *table à adressage direct*.

Une deuxième solution qui permettra de résoudre le problème de la taille du tableau consistera à prendre un tableau de taille $m \ll |A|$. Il faudra alors trouver une fonction $h : A \to \{0, \dots, m-1\}$, dite *fonction de hashage*, pour pouvoir trouver l'indice d'une entrée dans le tableau. Cependant, il faudra gérer plusieurs problèmes.

Collision: Comme $m < |A|$, par le principe des *cages à pigeon* il y aura forcément deux entrées différentes x et y avec $h(x) = h(y)$ et on parlera de *collision* dans ce cas. Comment faire pour les limiter ?

Choix de h: Comment choisir la fonction h et l'entier m de manière à limiter les collisions ?

Garantie Accés Constant: Pourra-t-on garantir un accés constant ? ou au moins un accés moyen constant ?

Une bonne fonction de hashage sera une fonction qui permettra de limiter les collisions afin de garantir un accés moyen constant. Comme on cherche également un accés rapide il faudra que la fonction de hashage puisse être calculée en temps constant.

Exemple 14. *Dans l'exemple de l'annuaire précédent, on peut par exemple prendre un entier premier m, disons* 367. *Ensuite, on associe à chaque lettre c un entier $v(c)$ entre* 1 *et* 26. *La fonction h est définie telle que pour tout mot $a := a_1 \cdots a_n$ on ait $h(a) = \left(\sum_{1 \leq i \leq n} v(a_i)\right) \mod m$.*

6.1. **Résolution des collisions par chaînage.** Supposons que l'on dispose d'un tableau T de taille m pour stocker les éléments de notre univers U et d'une fonction de hashage $h : U \to \{0, \dots m-1\}$. Chaque case du tableau $T[i]$ pointera sur une liste chaînée qui contient l'ensemble $\{x \in U \in h(x) = i\}$. On pourra insérer en temps constant en insérant tout le temps en début de liste, par contre il faudra faire une recherche dans la liste $T[i]$ pour supprimer ou rechercher un élément. Le comportement le pire c'est lorsque tout est hashé dans une seule liste. Calculant par contre les performances lorsque l'on dispose d'une *bonne fonction de hashage*. Supposons que la fonction de hashage h est *simplement uniforme*, *i.e.*, que chaque élément a la même probabilité d'être hashé vers l'une des cases $\{0, \dots, m-1\}$. On supposera également que $h(x)$ se calcule en temps constant. Si on note n_i la longueur de la liste $T[i]$, alors on aura $\sum_{0 \leq i \leq m-1} n_i = |U|$, et que la valeur moyenne de n_i $E[n_i] = \frac{n}{m}$ que l'on notera α comme c'est le même pour tout i.

Theorem 5. *Si h est une fonction de hashage simplement uniforme, alors une recherche infructueuse dans une représentation des collisions par chaînage nécessite un temps moyen $\theta(1 + \alpha)$.*

Démonstration. Comme la longueur moyenne d'une liste est α et que toutes les listes ont les mêmes probabilités d'être examinées, on peut en déduire qu'en moyenne on visitera une liste. En incluant le temps de calcul de h on obtient le temps moyen annoncé. □

Theorem 6. *Si h est une fonction de hashage simplement uniforme, alors une recherche réussie dans une représentation des collisions par chaînage nécessite un temps moyen $\theta(1 + \alpha)$.*

Démonstration. La probabilité que l'élément à chercher x est l'un des $|U|$ éléments de U est le même. Le temps de recherche est le temps nécessaire pour calculer $h(x)$ plus le temps de parcourir la liste $T[h(x)]$ jusqu'à trouver x. Pour trouver le nombre moyen d'éléments examinés on prend la moyenne sur les $|U|$ éléments x de U du nombre moyen d'éléments ajoutés après x dans $T[h(x)]$ (plus 1 car il faut aussi calculer h). En effet, si on note X_{ij} la variable indicatrice $h(x_i) = h(x_j)$, alors

comme h est simplement uniforme on a $Pr(X_{ij} = 1) = \frac{1}{m}$ et donc $E[X_{ij}] = \frac{1}{m}$. Du coup le nombre moyen d'éléments examinés dans une recherche réussie est

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right].$$

En utilisant la linéarité de l'espérance et quelques petits calculs on obtient le résultat escompté. □

6.2. **Fonctions de Hashage.** Une bonne fonction de hashage est une fonction simplement uniforme. Cependant, en pratique il est difficile d'avoir une fonction simplement uniforme car il est difficile de savoir comment les éléments sont distribués et la distribution n'est pas toujours uniforme (les éléments ne sont pas forcément indépendants). Par contre il existe des heuristiques pour avoir des fonctions de hashage plutôt efficaces. On considérera chaque élément comme un entier (on pourra toujours trouver un moyen d'associer à chaque élément un entier et deux éléments ayant des entiers différents de façon efficace).

Extraction. On choisit aléatoirement p bits, p un nombre premier. Ceci est trop arbitraire et est mauvais. Une solution est de choisir par exemple p bits au milieu de la représentation ou du carré. Ainsi on utilise le fait que l'on peut énumérer uniformément les mots de taille p pour avoir de bonnes propriétés.

Compression. On découpe l'entrée en des blocs de taille fixe, ensuite on utilise une fonction de compression pour compresser chaque bloc et on utilise un opérateur binaire (par exemple le xor) qui combine tous les compressés pour obtenir le résultat final. Fonction classique de compression : MD5, SHA-*, ...

Division. On définit $h(x) := x \mod m$. Il faudrait quand même que $m \neq 2^k$ car sinon on utiliserait que les k bits de poids faible. De même $m \neq 2^k - 1$ car une permutation des caractères de x ne modifie pas $h(x)$ lorsque x est interprétée en base 2^k. Knuth donne trois propriétés désirées pour m : (1) m premier, (2) m ne divise pas $r^k \pm a$ pour des petites valeurs de a et k, (3) m est sans diviseur premier.

Multiplication. On définit $h(x) := \lfloor m \cdot (\cdot x \cdot A \mod 1)\rfloor$ où $x \mod 1$ est la partie décimale de x avec $0 < A < 1$. On choisit en général une valeur de m égale à une puissance de 2 et pour A on prend $\frac{s}{2^w}$ où w est la taille des mots de l'ordinateur et $0 < s < 2^w$. La méthode fonctionne mieux pour certaines valeurs de A que pour d'autres, et par exemple Knuth suggère une valeur de A égale à $\pm\frac{(\sqrt{5}-1)}{2}$

Ensemble Universel. Quelque soit la fonction de hashage choisie, il ne faut pas que le choix dépende des données. En effet, si on fixe une fonction de hashage on pourra toujours s'arranger pour trouver des clés qui seront renvoyées toutes sur le même indice. La meilleure stratégie est de choisir une fonction de hashage indépendamment des données à hacher et de manière aléatoire. On parle alors de *hashage universel*. Cela suppose qu'il faut choisir les fonctions de hashage dans des ensembles $\mathcal{H}$ dits *universels* où pour tout x et y $|\{h \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{m}$. Ce qui implique qu'un choix aléatoire dans $\mathcal{H}$ donne de bonnes performances. Voici un exemple d'ensembles universels. Soit $p > m$ un nombre premier assez grand. Pour tout $a \in \mathbb{Z}_p^*$ et $b \in \mathbb{Z}_p$ on pose $h_{a,b}(x) := ((a \cdot x + b) \mod p) \mod m)$. L'ensemble $\mathcal{H}_{p,m} := \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ est universel.

6.3. **Adressage Ouvert.** L'adressage d'une table de hachage est dit *ouvert* si les éléments du tableau sont conservés dans le tableau. En d'autres termes un élément du tableau soit contient un élément soit vaut $NULL$. Dans ce cas une seule valeur par case du tableau. Pour insérer un élément, on recherche une case vide, on parle souvent de *sonde*, pour l'y insérer. Au lieu de faire une recherche linéaire ce qui prendrait $O(m)$, la fonction de hachage prendrait en compte le nombre de sondages et ainsi la fonction de hachage serait $h : U \times [m] \to \{0, \ldots, m-1\}$. Il faudrait bien entendu que $h(x, i) \neq h(x, j)$ pour $i \neq j$. Ainsi, chercher une place serait d'appeler la fonction $h(x, i)$ pour $i = 0 \to m - 1$ jusqu'à trouver une place. De la même manière pour la recherche. Pour la suppression, on cherche la case où se trouve l'élément et on le met à $NULL$.

Voici quelques exemples de fonction de sondage. Soit $h' : U \to \{0, \ldots, m-1\}$ une fonction de hachage.

sondage linéaire: $h(x, i) := (h'(x) + i) \mod m$.

sondage quadratique: $h(x, i) := (h'(x) + c_1 \cdot i + c_2 \cdot i^2) \mod m$ où c_1 et c_2 sont des constantes entières.

double hachage: $h(x, i) := (h_1(x) + i \cdot h_2(x)) \mod m$ où h_1 et h_2 sont des fonctions de hachage.

Références

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2002.

[2] Christine Froidevaux, Marie-Claude Gaudel, and Michèle Soria. *Types de données et Algorithmes*. INRIA, 3rd edition, 1993.

[3] Brian W. Kernighan and Dennis Ritchie. *Programmieren in C - mit dem C-Reference Manual in deutscher Sprache ; ANSI C (2. Ausgabe)*. Hanser, 1990.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume I : Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[5] J.G Penaud. Cours d'algorithmique et de structures de données, 2000. Université Bordeaux 1.

[6] O. Reynaud. Cours de structures de données, 2010. Université Clermont 2.

[7] R. Strandh and I. Durand. *Architecture de l'Ordinateur pour Informaticiens*. MétaModulaire, 2002.

KANTE@ISIMA.FR