Structure de données et algorithmique

Exercices et Résumés de Cours

Sandrine Vial modifié par Yann Strozecki

2015-2016

Avant-Propos

Dans ce document vous trouverez 3 parties :

— une partie qui contient les exercices que vous ferez en TD ;

— une partie qui contient quelques conseils sur la rédaction des preuves et des algorithmes ainsi que quelques rappels mathématiques utiles pour le cours.

— une partie qui contient les résumés des cours ;

Remerciements

Je remercie Lélia Blin [1], Christian Destré [2], Christian Laforest [3], Cécile Mailler [4], Pascal Petit [5] et Nicolas Thibault [6] pour leurs commentaires avisés et l'amélioration de ce document.

1. Maitre de Conférences à l'université d'Evry Val d'Essonne
2. Ingénieur Orange Labs
3. Professeur à l'Université de Clermont-Ferrand
4. Doctorante à l'Université de Versailles-St Quentin
5. enseignant à l'Université d'Evry Val d'Essonne
6. Maître de Conférences à l'Université de Paris 2

Table des matières

Table des matières **v**

1 Exercices **1**
1.1 Tableaux et complexité 1
1.2 Listes 4
1.3 Les tris 5
1.4 Le hachage ou l'adressage dispersé 6
1.5 Les structures de données abstraites 8
1.6 Les arbres 9
1.7 Introduction à la Programmation Dynamique 13

2 Conseils **15**
2.1 Pseudo-Code 15
2.2 Mathématiques élémentaires 18
2.3 Techniques de preuves 19

3 Résumés de cours **21**
3.1 Introduction à la complexité des algorithmes 21
3.2 Etudes de complexité 23
3.3 Les tris 28
3.4 Les structures de données abstraites 30
3.5 Les arbres 35
3.6 Les Arbres Binaires de Recherche 43

Chapitre 1

Exercices

1.1 Tableaux et complexité

Exercice 1 – Recherche simultanée du maximum et du minimum.
On cherche simplement à donner un algorithme permettant de trouver à la fois le maximum et le minimum d'un tableau nommé *Tab* de n éléments distincts.

1. Écrire tout d'abord un algorithme qui donne la place de l'élément minimum dans le tableau ainsi que sa valeur.
2. Combien de comparaisons effectue cet algorithme ? Quelle est sa complexité ?
3. Écrire ensuite un algorithme pour trouver les places et valeurs à la fois du minimum et du maximum dans ce tableau.
4. Pouvez-vous proposer un algorithme qui n'effectue que $\frac{3n}{2}$ comparaisons ?

Exercice 2 – Recherche Séquentielle
On considère un tableau A de n valeurs. On cherche un algorithme permettant de savoir à quel endroit du tableau se trouve une valeur *clé*.

1. On supppose tout d'abord que *clé* se trouve bien dans le tableau au moins une fois et on cherche la première occurrence. Donner un algorithme simple qui résout ce problème.
2. A quoi correspond le pire des cas ? En déduire la complexité de l'algorithme.
3. A quoi correspond le meilleur des cas ?
4. Comment faut-il modifier cet algorithme si l'on est pas sûr que *clé* appartienne au tableau ? Cela change-t-il quelque chose au niveau de la complexité ?
5. Intéressons-nous maintenant à la complexité moyenne.
   a) En supposant que l'élément *clé* apparaisse une fois et une seule dans le tableau et que toutes les positions de *clé* dans le tableau sont équiprobables, quelle est la complexité moyenne ?
   b) En supposant que l'élément *clé* apparaisse au plus une fois, qu'il puisse donc ne pas figurer mais qu'on connaisse la probabilité qu'il y figure (égale à p), que devient cette complexité moyenne ?
6. Si le tableau est trié par ordre croissant, est-il possible d'améliorer l'algorithme ?

Exercice 3 – Complexité en fonction de deux paramètres
Déterminer la complexité des algorithmes suivants (par rapport au nombre d'itérations effectuées), où m et n sont deux entiers positifs.

Algorithme 1.1 Algorithme A

```
AlgoA(m : entier, n : entier)
▷ Entrées : m et n
 Début
▷ Variables locales
   i,j : entier ;
   i ← 1 ; j ← 1 ;
    tant que ((i ≤ m) et (j ≤ n))  faire
       i ← i + 1 ;
       j ← j + 1 ;
    fin tant que

 Fin
```

Algorithme 1.2 Algorithme B

```
AlgoB(m : entier, n : entier)
▷ Entrées : m et n
 Début
▷ Variables locales
   i,j : entier ;
   i ← 1 ; j ← 1 ;
    tant que ((i ≤ m) ou (j ≤ n))  faire
       i ← i + 1 ;
       j ← j + 1 ;
    fin tant que

 Fin
```

Algorithme 1.3 Algorithme C

```
AlgoC(m : entier, n : entier)
▷ Entrées : m et n
 Début
▷ Variables locales
   i,j : entier ;
   i ← 1 ; j ← 1 ;
    tant que (j ≤ n)  faire
        si (i ≤ m)
           i ← i + 1 ;
        sinon
           j ← j + 1 ;
        fin si
    fin tant que

 Fin
```

Algorithme 1.4 Algorithme D

```
AlgoD(m : entier, n : entier)
▷ Entrées : m et n
 Début
▷ Variables locales
   i,j : entier ;
   i ← 1 ; j ← 1 ;
    tant que (j ≤ n)  faire
        si (i ≤ m)
           i ← i + 1 ;
        sinon
           j ← j + 1 ;
           i ← 1 ;
```

fin si
fin tant que

Fin

Exercice 4 – Âge du capitaine

Tournoi d'âge du capitaine. Vous pouvez utiliser l'opération %. Essayez de faire des programmes obscurs qui font les choses suivantes :

— ne s'arrêtent jamais
— prennent un temps très long
— calculent un très gros entier

Exercice 5 – Taille d'un entier

Donner un algorithme un algorithme récursif qui prend en entrée un entier n et renvoie sa taille en base 10. Quelle est sa complexité ?

Exercice 6 – Fusion

1. Proposer un algorithme qui prend deux tableaux triés en entrée et qui renvoie le tableau trié constitué de l'union des éléments des deux tableaux.
2. Quel est la complexité de cet algorithme ?
3. Supposons que l'entrée est un tableau T de taille n et un entier m tel que $T[1 \dots m]$ et $T[m+1 \dots n]$ sont triés. Comment trier T sans utiliser de tableau auxiliaire ? Il faut le faire en temps linéaire (question difficile pour ceux qui sont en avance).

Exercice 7 – Doublons

1. Soit T un tableau de taille n, proposer un algorithme simple qui supprime les doublons dans ce tableau. Montrer que sa complexité est en $O(n^2)$.
2. Proposer un algorithme plus efficace

Exercice 8 – Exponentiation rapide

1. On se donne un entier r et on veut calculer sa représentation en base 2. On stocke celle-ci dans un tableau d'entier T de taille $\lceil \log(n) \rceil$ avec le bit de poids faible au début, c'est à dire

$$r = \sum_{i=0}^{\lceil \log(n) \rceil} T[i] * 2^i$$

   — Proposer un algorithme qui donne la décomposition en base 2 de r.
   — Quelle est la complexité de votre algorithme ?

2. On veut calculer les puissances entières d'un entier : n^r. On suppose que la machine idéale sur laquelle on travaille peut manipuler des entiers de taille arbitraire.
   — Proposer un algorithme qui prend en entrée n et r et calcule n^r en effectuant $\max(0, r-1)$ multiplications.
   — Proposer un algorithme qui calcule n^{2^l} en effectuant au plus l multiplications.

   — Proposer un algorithme qui prend en entrée n et r et calcule n^r en effectuant moins de $2k$ multiplications si $2^k \leq r < 2^{k+1}$. Cet algorithme est appelé exponentiation rapide.
   Idée : À l'aide de la première question, décomposer r en base 2 et remarquer que

$$n^r = n^{\left(\sum_{i=0}^{\lceil \log(n) \rceil} T[i] * 2^i \right)} = \prod_{i=0}^{\lceil \log(n) \rceil} n^{T[i]*2^i}$$

Exercice 9 – Tableau circulairement trié

Soit T un tableau trié d'entiers (positifs et négatifs) tous distincts. On sait que ce tableau est circulairement trié dans l'ordre croissant, c'est-à-dire qu'il existe un indice i tel que en commençant

la lecture à $T[i]$ jusqu'a $T[n]$ et recommençant à $T[1]$ jusqu'à $T[i-1]$ on obtient une suite croissante. Par exemple, la suite $[15, 17, 20, 2, 3, 5, 8, 10]$ est circulairement triée, et l'indice de départ à considerer est $i = 4$.

1. Donner un algorithme en temps linéaire qui permette de trouver l'indice i du plus petit élément du tableau.
2. Expliquer pourquoi la monotonie d'un sous tableau peut se tester en comparant ses bornes. De plus, quand on coupe en deux le tableau T, si on obtient une seule moitié monotone alors la solution recherchée se trouve dans l'autre moitié. Verifiez cette propriété sur le tableau donné en exemple.
3. Donner un algorithme inspiré de la recherche dichotomique qui trouve l'indice i du minimum en temps logarithmique.

Exercice 10 – Sous-somme

On considère un tableau A contenant des entiers signés (positifs et négatifs). On se propose de rechercher un sous-tableau de A (i.e. un ensemble d'éléments consécutifs de A) ayant la somme maximale.

1. Ecrire une procédure qui si on lui donne i et j renvoie la somme des éléments du sous-tableau de borne i et j.
2. Ecrire un algorithme qui éxecute la procédure précédente pour tous les couples (i, j) afin de trouver la somme maximale.
3. Donner la complexité de votre algorithme
4. Améliorer cet algorithme naïf en essayant de ne pas refaire tous les calculs. Indice : utiliser le résultat de (i, j) pour calculer $(i, j + 1)$
5. Donner la complexité de cette amélioration
6. Proposer un algorithme *Somme(A,j)* pour calculer la plus grande somme d'éléments d'un sous-tableau de A terminant par la case j.
7. Comment calculer simplement *Somme(A,j)* à partir de *Somme(A,j-1)* ? En déduire un algorithme linéaire pour calculer la somme maximale.

1.2 Listes

Exercice 11 – Opérations de base sur une liste

Proposer des algorithmes pour les opérations de liste suivantes et donner leur complexité :

1. Écrire un algorithme $sum(l)$ qui renvoie la somme des éléments de la liste l.
2. Écrire un algorithme $map(f, l)$ qui prend en argument la fonction f et la liste l et qui applique f à chacun des éléments de l et renvoie la liste obtenue.
3. Écrire un algorithme $concat(l_1, l_2)$ qui renvoie la concaténation de l_1 et de l_2.
4. Écrire un alorithme $min(l)$ qui renvoie le minimum de la liste.
5. Écrire un algorithme $inser(l, e)$ qui insère l'élément e dans la liste l. Le faire aussi dans le cas d'une liste triée.
6. Écrire une algorithme $miroir(l)$ qui renverse l. La complexité est-elle bien linéaire ?
7. Écrire un algorithme $palindrome(l)$ qui teste si l est un palindrome.
8. Écrire un algorithme $simplifie(l)$ qui remplace les éléments consécutifs identiques par un seul d'entre eux.

Exercice 12 – Liste doublement chaînée

1. Proposer une structure de donnée pour les listes doublement chaînées.
2. Écrire un algorithme $inser(l, e)$ qui insère l'élément e dans la liste l.
3. Écrire un algorithme $concat(l_1, l_2)$ qui renvoie la concaténation de l_1 et de l_2.

4. Écrire une algorithme $miroir(l)$ qui renverse l.

Exercice 13 – Liste dans un tableau

On se propose d'implémenter une liste dans un tableau. On sait que la liste ne sera jamais de taille supérieure à N la taille du tableau. Chaque élément du tableau a deux champs : *val* la valeur de l'élément et *suivant* l'indice de l'élément suivant.

1. Proposer un algorithme pour insérer au début de la liste.
2. Proposer un algorithme pour insérer à la fin de la liste.
3. Donner deux algorithmes de recherche de l'élément x dans une liste L.

Exercice 14 – Générateur de nombre premier

1. Donner un algorithme qui teste si un nombre est premier.
2. Donner un algorithme qui génère tous les nombres premiers de 1 à n (donné en entrée de l'algorithme).
3. Quelle est sa complexité ? Peut-on l'améliorer ?
4. Le crible d'Ératosthène fonctionne de la manière suivante : on écrit tous les nombres de 1 à n puis pour chaque nombre de 1 à n si il n'a pas été rayé c'est un nombre premier et on raye tous ses multiples sinon on passe au nombre suivant. Implémenter cet algorithme avec un tableau puis avec une liste (pour représenter les nombres qui sont rayés au fur et à mesure). Quelle est la complexité dans les deux cas ?

1.3 Les tris

Exercice 15 – Tri par selection.

Une méthode pour trier un ensemble de manière croissante consiste à chercher le minimum, puis à le mettre en première position, et à recommencer ainsi sur le reste du tableau.

1. Écrire un algorithme récursif pour trier un tableau par cette méthode.
2. Écrire un algorithme itératif pour faire la même chose.
3. Calculer la complexité de cet algorithme.
4. On peut se servir du premier exercice pour calculer conjointement le minimum et le maximum. Quelle forme a alors l'algorithme ? Cela change-t-il quelque chose à sa complexité ?

Exercice 16 – Amélioration du tri à bulles

Nous allons étudier les améliorations de l'algorithme du tri à bulles vu en cours.

```
bulle (T : tableau d'entiers)
début
  var locales : i,j, tmp : entiers

  pour i de 1 à NB faire
    pour j de NB-1 en décroissant jusqu'à i faire
        si (T[j-1] > T[j])
        faire
              tmp = T[j-1];
              T[j-1] = T[j];
              T[j] = tmp;
        finsi
    finpour
  finpour
fin
```

1. On constate parfois qu'à une étape donnée aucune permutation n'a été faite. Proposez une amélioration de l'algorithme qui utilise la constatation que l'on vient de faire.

2. Une autre possibilité pour améliorer le tri à bulles est de mémoriser le plus grand indice à partir duquel les échanges ne se font plus, c'est-à-dire l'indice en dessous duquel la liste est triée. Proposez un algorithme basée sur cette nouvelle amélioration.

Exercice 17 – Calcul rapide de médiane

On veut calculer raidement le $k$ème élément d'un tableau sans le trier !

1. Modifier l'algorithme de tri rapide pour trouver le $k$ème élément d'un tableau.
2. Évaluer sa complexité dans le pire des cas et en moyenne.
3. On peut utiliser une idée subtile pour améliorer la complexité dans le pire des cas. Il faut grouper les éléments par 5, trouver les médians de ces bloc et trouver le médian de ces médians. Le médian des médians est un bon élément pour partitionner le tableau ! Donner une version modifiée de l'algorithme avec une complexité linéaire.

Exercice 18 – Tri expérimental

Pour cet exercice, se munir d'un paquet de carte. On considère deux ordres : l'ordre des valeurs (as, deux, trois, ..., roi) et l'ordre des familles (carreau, trèfle, coeur, pique).

1. Choisir un algorithme de tri "lent" comme le tri par insetion ou sélection. Mélanger le paquet de carte. Appliquer l'algorithme pour trier le paquet dans l'ordre des familles. L'appliquer une seconde fois pour le trier dans l'ordre des valeurs.
2. Le tri utilisé est-il stable ? Comment obtenir un paquet de carte trié selon l'ordre généralement utilisé ?
3. Répéter l'expérience des deux tris successifs selon les familles puis les valeurs en utilisant le tri fusion. Que constatez-vous ?

Exercice 19 – Tri par dénombrement.

1. Est-ce-que le tri par dénombrement vu en cours est stable ?
2. Étant-donné un tableau de n entiers dans $[0, k]$, créer un structure de donnée qui permet de répondre à la question suivante en deux opérations arithmétiques : combien y a-t-il d'entiers entre a et b ? Cette structure doit être calculée en temps $O(n + k)$.

1.4 Le hachage ou l'adressage dispersé

Exercice 20 – Table de hachage avec résolution des collisions par chainage

Vous écrirez les différentes opérations possibles sur une table de hachage avec résolution des collisions par chaînage. :

— Insertion d'une clé
— Suppression d'une clé
— Recherche d'une clé

Vous préciserez pour chacune de ces opérations, la complexité en temps de celle-ci. Vous utiliserez les deux types de fonctions de hachage possible :

La méthode de la division $h(k) = k \mod m$ avec m le nombre de cases de la table de hachage

La méthode de la multiplication $h(k) = \lfloor m(kA \mod 1) \rfloor$ avec $A = \frac{\sqrt{5}-1}{2}$.

Nous allons maintenant considérer un exemple précis dans lequel la table a une longueur $m = 11$, et nous allons insérer dans cet ordre les clés : 10,22,31,4,15,28,17,88,59. Vous afficherez le contenu de la table de hachage pour chacune des fonctions de hachage possible.

Exercice 21 – Table de hachage avec adressage ouvert

Dans un premier temps vous écrirez les différentes opérations possibles sur une table de hachage :

— Insertion d'une clé
— Suppression d'une clé
— Recherche d'une clé

Dans ce contexte, il existe 3 types de fonctions de sondage usuelles :

Le sondage linéaire $h(k,i) = (h'(k)+i) \mod m$ avec i le nombre de sondages déjà effectués et h' une fonction de hachage classique vue dans le cas précédent.

Le sondage quadratique $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ avec c_1 et c_2 des constantes quelconques.

Le double hachage $h(k,i) = (h_1(k) + i h_2(k))$

Vous utiliserez les trois fonctions possibles, en considérant :
— $h'(k) = k \mod m$
— $c_1 = 1$ et $c_2 = 3$
— $h_1(k) = k \mod m$ et $h_2(k) = 1 + (k \ mod \ (m-1))$

Nous allons maintenant considérer un exemple précis dans lequel la table a une longueur $m = 11$, et nous allons insérer dans cet ordre les clés : 10,22,31,4,15,28,17,88,59. Vous afficherez le contenu de la table de hachage pour chacune des fonctions de hachage possible.

Exercice 22 – Hachage coalescent

Le hachage coalescent est une méthode de hachage qui résoud les collisions par chaînage. Les éléments chaînés se trouve dans la table de hachage elle-même. Chaque élément contiendra ainsi, l'indice du prochain élément dans la chaîne. La table de hachage aura deux parties distinctes, la première contient les éléments directement hachés, et la deuxième partie contient les éléments dont on a résolu la position par chaînage.

On notera m' la taille de la table de hachage. La taille de la partie qui contient les clés directement hachées sera de taille m et la seconde partie sera de taille $m' - m$.

Lors de l'insertion et de la suppression d'un élément on maintiendra deux valeurs :
— n qui contient le nombre d'éléments dans la table
— $nextfree$ qui contient l'indice de la prochaine alvéole vide.

1. Écrire une fonction d'insertion d'un élément dans le tableau, vous pouvez utiliser la fonction $h(k) = k \mod m$ qui renvoie l'indice dans la table en fonction de la clé d'un élément.
2. Écrire une fonction de suppression d'un élément dans le tableau.
3. Écrire une fonction de recherche d'un élément dans le tableau.

1.5 Les structures de données abstraites

Exercice 23 – Utilisation d'une pile

Vous disposez des 4 fonctions suivantes :

`créer_nouvelle_pile(), pile_vide(p), dépiler(p) et empiler(p,e)`

où p est une pile et e un élément.

1. Donner un algorithme qui supprime un élément sur deux d'une pile en utilisant uniquement ces quatres opérations.
2. Proposer un algorithme qui étant donné deux piles d'entiers p_1 et p_2, déplacent leurs éléments pour avoir tous les éléments impairs dans p_1 et pairs dans p_2.
3. Pouvez vous le faire avec une seule pile supplémentaire ? Avec aucune autre pile ? Et si p_1 et p_2 sont des files ?

Exercice 24 – Une Pile

Une pile est une structure de données dans laquelle on a accès à un seul élément : celui qui est au sommet de la pile. La pile met en œuvre l'adage "dernier arrivé, premier servi". En effet, lorsque l'on ajoute un élément on va l'ajouter au sommet de la pile. Par contre, lorsque l'on va vouloir retirer un élément on ne pourra enlever que celui qui est au sommet de la pile, c'est-à-dire le dernier élément inséré.

On veut mettre en œuvre une pile d'au plus n éléments à l'aide d'une liste. La structure de données que nous allons utiliser est donc la suivante :

```
Enregistrement L_Pile {
  val : entier;

   suivant :↑ L_Pile;
}
```

Vous écrirez les algorithmes suivants et donnerez leur complexité :

1. Un algorithme qui vérifie si la pile est vide.
2. Un algorithme qui vérifie si la pile est pleine.
3. Un algorithme pour empiler (insérer) un élément sur la pile.
4. Un algorithme pour dépiler (supprimer) un élément de la pile.

Proposer une implémentation alternative à l'aide d'un tableau.

Exercice 25 – Une File

Une file est une structure de données dans laquelle le prochain élément à supprimer est l'élément qui est présent depuis le plus longtmeps dans la file. La file met en œuvre l'adage "premier arrivé, premier servi". En effet, lorsque l'on ajoute un élément on va l'ajouter en fin de file. Par contre, lorsque l'on va vouloir retirer un élément on enlevera celui qui est en tête de file.

On veut mettre en œuvre une file à l'aide d'une liste.

1. Vous définirez le type de données que vous allez utiliser.
2. Vous écrirez les algorithmes suivants et donnerez leur complexité :
   a) Un algorithme qui vérifie si la file est vide.
   b) Un algorithme qui vérifie si la file est pleine.
   c) Un algorithme pour insérer un élément dans la file.
   d) Un algorithme pour supprimer un élément de la file.

Proposer une implémentation alternative à l'aide d'un tableau.

Exercice 26 – Tri par tas

C'est un algorithme découvert par Williams en 1964. La procédure de construction du tas est dûe à Floyd en 1964. La technique utilisée ici consiste à utiliser une structure de données spécifiques (le tas) pour résoudre le problème de tri. Un tas peut être représenté sous forme de tableau (A) avec 2 paramètres supplémentaires :

— longueur(A) : nombre d'éléments du tableau
— taille(A) : nombre d'éléments du tas rangés dans ce tableau

Un tas peut être vu comme un arbre. La racine de l'arbre est $A[1]$ et connaissant l'indice i d'un élément :

— père$(i) = \lfloor \frac{i}{2} \rfloor$
— gauche$(i) = 2i$
— droit $(i) = 2i + 1$

La propriété fondamentale des tas est la suivante :
Pour chaque nœud i autre que la racine on a :

$$A[\text{père}(i)] \geq A[i]$$

1. Voici une liste de tableaux, vous direz pour chacun d'eux s'ils satisfont la propriété de tas et si tel n'est pas le cas vous argumenterez votre réponse.

Tableau 1	16	14	10	8	7	9	3	2	4
Tableau 2	11	6	10	8	7	5	3	2	4
Tableau 3	18	16	15	13	14	1	2	12	11
Tableau 4	18	16	15	13	14	1	2	19	11
Tableau 5	16	14	15	8	7	9	3	2	4

2. Ecrivez une fonction qui étant donné un tableau et un indice vérifie que l'arbre logique enraciné en cet indice vérifie bien la propriété de tas. Vous donnerez la complexité de votre fonction.
3. On suppose que les sous arbres enracinés en $2i$ et $2i+1$ vérifient la propriété de tas. On veut que l'arbre enraciné en i la vérifie aussi. Donner une fonction qui prend en argument A et i et qui modifie A afin que le sous-arbre enraciné en i vérifie la propriété de tas. Le principe est simple : on regarde pour $A[i]$ si ces deux fils sont plus petit que lui. Si tel est le cas pas de problème pour $A[i]$ par contre si l'un des deux fils est le plus grand, admettons que ce soit l'élément d'indice $2i$, il faut alors échanger cet élément avec l'élément $A[i]$ et vérifier que le nouvel arbre logique enraciné en $A[2i]$ est bien un tas. Il est conseillé d'écrire un algorithme récursif.
4. A partir de la fonction précédente, donner une fonction qui transforme n'importe quel tableau en tas. Quelle est sa complexité ?
5. Comment utiliseriez-vous les deux fonctions précédentes pour faire un algorithme de tri ? Ecrivez cet algorithme et donnez-en sa complexité ? Est-ce un tri par comparaison optimal ?
6. Comment utiliser la structure de tas pour calculer les k plus grands éléments d'un tableau de taille n. Quelle est la complexité de votre algorithme ?

1.6 Les arbres

Exercice 27 – Arbres Binaires (1)

On considère la structure de données `arbre binaire`. Il vous faut donc écrire les fonctions suivantes :

— `NombreNoeuds` qui calcule le nombre de nœuds d'un arbre binaire
— `Descendants` qui calcule et stocke en chaque nœud de l'arbre, le nombre de ses descendants.

Exercice 28 – Arbres Binaires (2)

A un sous-arbre d'un arbre binaire on peut associer la liste des choix binaires (*gauche* ou *droite*), appelée *liste de direction*, qui sont faits pour aller de la racine de l'arbre binaire à la racine du sous-arbre binaire considéré

1. Si l'on note un déplacement à gauche par la lettre g et un déplacement à droite par la lettre d, quel est le sous-arbre considéré par la liste de direction suivante (g, d, g) dans l'arbre de la figure 1.1.
2. Ecrire une fonction *direction* qui à partir d'un arbre binaire et d'une liste de direction, renvoie le sous-arbre correspondant (s'il existe).

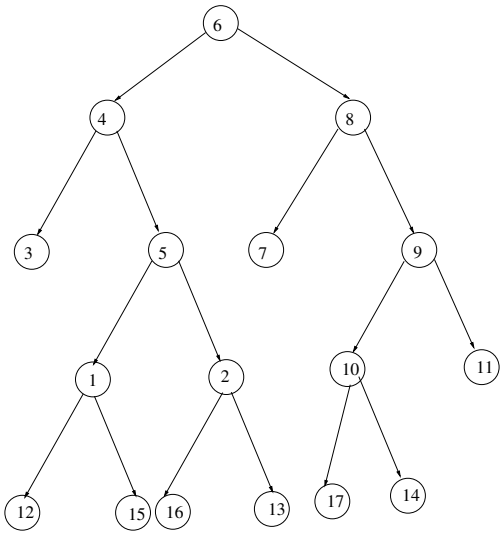


Figure 1.1 – Un arbre binaire

Exercice 29 – Parcours d'arbres

La liste des valeurs suivantes :

$$5, 9, 12, 2, 4, 11, 3, 7, 8, 1, 6, 10$$

a été mémorisée sous forme d'arbre binaire. La première valeur a donné la racine. La seconde son fils gauche, la troisième valeur son fils droit et ainsi de suite (sans se soucier de trier ou d'ordonner les valeurs).

1. Dessiner l'arbre obtenu.
2. Quel est l'ordre des valeurs que l'on obtient quand on fait un parcours infixe ?
3. Et avec des parcours préfixe et postfixe ?

Exercice 30 – Calculette

On veut utiliser un arbre binaire pour représenter une expression arithmétique.

1. Proposer une variante de l'arbre binaire pour représenter une expression arithmétique.
2. Proposer un algorithme qui prend une expression arithmétique et qui renvoie sa valeur
3. Au lieu d'un arbre on a une pile qui contient les éléments d'une expression arithmétique en notation polonaise inversée. Peut-on écrire un algorithme d'évaluation en utilisant uniquement les fonctions empiler, dépiler et pilevide ? Vous pourrez vous inspirer de la question précédente.

Exercice 31 – Bee Trees

Une abeille mâle est produite de manière asexuée à partir d'une abeille femelle. Par contre, une abeille femelle a deux parents : un mâle et une femelle.

1. Représenter l'arbre généalogique d'une abeille mâle jusqu'à la quatrième génération.
2. Combien une abeille mâle a-t-elle d'ancêtres de niveau 1 ? de niveau 2 ? de niveau n ?

Exercice 32 – Arbres équilibrés

On appelle ici *arbre équilibré* un arbre binaire tel que, en tout nœud, la hauteur des sous-arbres gauches et droits diffère d'au plus 1.

1. Donner plusieurs exemples possibles d'arbres équilibrés de 10 nœuds au total.
2. Quelles sont les hauteurs minimale et maximale d'un arbre équilibré de 20 nœuds au total.
3. Ecrire un algorithme qui vérifie qu'un arbre binaire quelconque est bien un arbre équilibré. Quelle est la complexité de cet algorithme ?

Exercice 33 – Nombre de Strahler

On définit récursivement une fonction entière ϕ sur l'ensemble des arbres binaires. Appelons $sag(A)$ et $sad(A)$ le sous-arbre gauche et le sous-arbre droit d'un arbre binaire A :

$$\phi(A) = \begin{cases} 0 & \text{si } A \text{ est l'arbre vide} \\ \max(\phi(sag(A)), \phi(sad(A)) & \text{si} \phi(sag(A)) \neq \phi(sad(A)) \\ \phi(sad(A)) + 1 & \text{si} \phi(sag(A)) = \phi(sad(A)) \end{cases}$$

Cette fonction ϕ permet d'associer à tout arbre binaire un nombre dit de Strahler.

1. Quel est le nombre de Strahler d'un arbre binaire complet de hauteur n ? le nombre de Strahler d'un arbre filiforme de hauteur n ?
2. Donner une procédure récursive qui calcule le nombre de Strahler d'un arbre binaire.

Exercice 34 – Codage d'Huffmann

Le *codage d'Huffmann* a pour but de coder un texte en binaire en respectant deux propriétés :

1. Coder chaque lettre par un mot sur 0,1 (toujours le même pour une lettre).
2. Ce code est préfixe.

On dispose donc de caractères à coder. Pour chaque caractère, on connait sa proportion d'occurences (i.e. le pourcentage) dans le texte à coder.

L'algorithme récursif est le suivant :

- Pour débuter, chaque lettre est un arbre ramené à un sommet étiqueté par la proportion d'occurences de cette lettre dans le texte.

Tant qu'il y a plus d'un arbre faire

- *Considérer T_1 et T_2 les deux arbres dont les racines portent les plus petites étiquettes e_1 et e_2*
- *Construire un nouvel arbre T dont la racine r a pour fils les racines de T_1 et T_2.*
- *r est étiqueté par $e_1 + e_2$.*

Dans cet arbre, pour chaque noeud, l'arête vers son fils gauche est étiquetée 0 et celle vers son fils droit 1.

1. Construire l'arbre obtenu pour l'exemple suivant.

Lettre	A	C	E	I	N	O	R	S	T
Prop.	18	12	24	6	7	2	11	17	3

2. Expliquer comment utiliser cet arbre pour coder et décoder un texte en s'aidant de l'étiquetage des arêtes. Coder le mot "CREATION".
3. Quel est l'intérêt que le code utilisé soit préfixe ?
4. Ecrire l'algorithme `EstUnCode (Suite, L, A)` qui vérifie si le tableau **Suite** de **L** bits passé en paramètre est un code selon l'arbre construit A et décode la suite de bits.
5. Donner un algorithme qui prend en entrée une suite de lettres et un arbre et qui produit le code associé à la suite de lettres.
6. Quelle est la complexité de la construction de l'arbre, de l'encodage et du décodage ?
7. Le langage *Morse* a été créé de cette façon en étudiant les occurrences des lettres dans des textes anglais notamment. Pourquoi certains pays pourraient-ils se sentir lésés ?
8. Donner une expression de la taille d'un texte encodé grâce au code de Huffman. Montrer que parmi tous les arbres possibles, celui obtenu par l'algorithme donne le plus petit texte (vous pourrez procéder par récurrence sur le nombre de lettres).

Exercice 35 – Les arbres binomiaux

1. Un arbre binomial est défini par récurrence : l'arbre binomial d'ordre 0 ne contient qu'un seul nœud, l'arbre binomial d'ordre k est construit à partir de deux arbres binomiaux d'ordre $k-1$ en plaçant la racine de l'un comme fils aîné de l'autre et l'ancien fils aîné devenant le frère. Dessinez les 5 premiers arbres binomiaux.
2. Montrer que la racine d'un arbre binomial est de degré k.
3. Montrer que si les fils de la racine d'un arbre binomial sont numérotés de $k-1$ à 0 en partant du fils aîné alors le fils i est un arbre binomial d'ordre i.
4. Montrer enfin qu'un arbre binomial d'ordre k possède 2^k nœuds et que la plus longue branche contient k nœuds.
5. Soit X un ensemble non vide à n éléments et soit $(n_0, n_1, \ldots, n_p)$ l'écriture binaire de n, où $n = n_0 2^0 + n_1 2^1 + \ldots + n_p 2^p$ avec $n_k \in \{0,1\}$, $k \in \{0, \ldots, p-1\}$ et $n_p = 1$. Une *forêt binomiale* sur X est formée :
   — de la suite $(\beta_0, \beta_1, \ldots, \beta_p)$ où β_k est l'arbre vide si $n_k = 0$ et l'arbre binomial d'ordre k sinon.
   — une bijection entre X et les sommets de la forêt constituée des arbres binomiaux non vide de la suite $(\beta_0, \beta_1, \ldots, \beta_p)$.

   Construire la forêt binomiale d'un ensemble à 6 éléments. Montrer que la hauteur de tout arbre β_k non vide est au plus $\lfloor \log_2 n \rfloor$ et que $p \leq \lfloor \log_2 n \rfloor$.
6. Soit $A =< r, (A_1, ..., A_f) >$ et B sont des arbres binomiaux de *même hauteur* h (définis sur deux ensembles disjoints E et F), on note `ArbreBin(A,B)` la fonction qui renvoie l'arbre binomial $A =< r, (A_1, ..., A_f), B >$ de hauteur $h+1$ sur $E \cup F$. On supposera cette fonction de complexité $O(1)$. Soit a une forêt binomiale sur X et soit b une forêt binomiale sur Y (X et Y sont disjoints). Proposer un algorithme `Union(a,b)` pour déterminer une forêt binomiale sur $X \cup Y$ en utilisant `ArbreBin`. Exprimer la complexité de cet algorithme en fonction du nombre n d'éléments de $X \cup Y$ si l'on suppose que chaque arbre binomial d'une forêt binomiale est représentée par un arbre binaire complet.
7. Appliquer `Union(a,b)` pour $a = (B_0, B_1, \emptyset)$ et $b = (B_0, B_1, B_2)$.
8. Soit E un ensemble dynamique dont chaque élément est muni d'une priorité $p(e)$. On représente le couple (E, p) par une *forêt binomiale tournoi* sur E pour gérer les opérations d'insertion, de recherche d'un élément de priorité minimale et de suppression d'un élément de priorité minimale. Une forêt binomiale tournoi pour (E, p) est une forêt binomiale sur E qui possède la propriété suivante : si le sommet x qui contient l'élément $e_1 \in E$ est le père du sommet y qui contient l'élément $e_2 \in E$, alors $p(e_1) \leq p(e_2)$.

   Dans ce contexte, un arbre binomial sera représenté par un pointeur sur un enregistrement contenant notamment les champs *objet* et *prio* destinés à stocker respectivement l'élément et la priorité de l'élément associé à sa racine r, et un champ l contenant un pointeur sur une liste $(l_0, \ldots, l_q)$ simplement chaînée d'enregistrements à deux chmaps l_i : le champ `arbrebin` qui contient un pointeur sur l'arbre binomial $i$ème fils de r et le champ `suiv` qui contient un pointeur sur l_{i+1}, et un champ `filsder` pointant sur le dernier enregistrement de cette liste, et permettant ainsi d'y ajouter un élément avec une complexité $O(1)$.

   De manière similaire, une forêt binomiale tournoi $\beta = (\beta_0, \ldots, \beta_q)$ pour (E, p) est représentée par un pointeur sur la liste simplement chaînée des arbres binomiaux qui la composent, comme ci-dessus.

   Ecrire une fonction `minimum(f)` de complexité $O(\log_2 n)$, qui renvoie un élément de priorité minimale de E (représenté par la forêt binomiale tournoi f).
9. Démontrer que si $B_p =< r, (A_1, \ldots, A_f) >$ est l'arbre binomial de hauteur p, alors $A_1, \ldots, A_f$ sont les arbres binomiaux $B_0, ..., B_{p-1}$.

   En déduire une fonction `Supprimer_Min(f)`, de complexité $O(\log_2 n)$, qui supprime un élément de priorité minimale de E (représenté par la forêt binomiale tournoi f).
10. Ecrire une fonction `Insérer(e,p,f)`, de complexité $O(\log_2 n)$, qui insère un élément e de priorité minimale de E (représenté par la forêt binomiale tournoi f).

Exercice 36 – Implémentation des ABR

1. Donner l'arbre binaire de recherche obtenu après les insertions de 4,3,5,7,9,2,1,2,6,8,19.
2. Supprimer la racine de l'arbre.
3. Donner des algorithmes pour l'insertion et la suppression.
4. Proposer une structure de donnée qui permette de maintenir un arbre équilibré.
5. Donner l'algorithme qui réalise une rotation gauche sur un noeud d'un arbre.
6. Comment utiliser cette structure de donnée pour maintenir un ensemble d'objets et son élément le plus fréquent, pendant une suite d'insertions et de suppressions ?
7. Prouver qu'un arbre binaire équilibré de hauteur h avec n noeuds vérifie $1,44\log(n)$. On pourra utiliser la construction des arbres de hauteur donnée ayant le minimum de sommets.

1.7 Introduction à la Programmation Dynamique

Exercice 37 – Fibonnaci mais rapide

1. Donner un algorithme récursif pour calculer la suite de Fibonacci.
2. Stocker les résultats intermédiaires dans un tableau pour accélérer le calcul (mémoïsation).

Exercice 38 – Retour sur le binôme

1. Rapeler l'algorithme récursif qui calcule $\binom{n}{p}$.
2. Quelle est sa complexité ?
3. Remplir un tableau avec les valeurs de $\binom{n}{p}$ pour $n < 8$ et $p < 5$.
4. Donner un algorithme qui remplit le tableau bidimensionnel $(m \times m)$ T de manière à ce que $T[n][p] = \binom{n}{p}$.
5. Quelle est sa complexité ?

Exercice 39 –

Un voleur trouve dans un coffre-fort n types d'objets de tailles et de valeurs différentes mais n'a qu'un petit sac à dos de contenance m pour porter son butin. Le *problème du sac à dos* consiste à trouver la combinaison d'objets que le voleur doit choisir pour maximiser la valeur total de ce qu'il emporte.

Exemple :

type	1	2	3	4	5
valeur	4	5	10	11	13
taille	3	4	7	8	9

On suppose que chaque objet de type i, $i \in \{1, \ldots, n\}$, est en quantité illimitée dans le coffre-fort. De plus les types des objets sont triés tel que $v[1] < \cdots < v[n]$. On supposera enfin que les tailles sont des entiers.

1. Dans l'exemple précédent, quelle est la valeur maximale du butin qu'un voleur peut emporter dans un sac de contenance $m = 17$?
2. Proposer un algorithme utilisant la programmation dynamique permettant de donner une solution au problème du sac à dos, en notant $v[i]$ la valeur et $t[i]$ la taille de l'objet de type i.
3. Quelle est la complexité en temps de cet algorithme en fonction de n et m ?

Exercice 40 – If she weighs the same as a duck ...

Étant donné un ensemble de poids donné par un tableau de taille n, on veut les placer sur les plateaux d'une balance de façon à ce que la balance soit équilibrée. Proposer un algorithme de programmation dynamique qui trouve une solution si elle existe, sachant que les poids sont des

entiers entre 1 et 100.

Exercice 41 – On considère un escalier de m marches que l'on peut gravir à l'aide de sauts au choix de $\alpha_1, \alpha_2, \ldots, \alpha_p$ marches. On veut calculer $N(s, m)$ le nombre de façons différentes de gravir m marches en s sauts.

1. Donner une formule de récurrence pour $N(s, m)$.
2. Proposer un algorithme récursif pour calculer $N(s, m)$.
3. Etant donnés $p = 3, \alpha_1 = 2, \alpha_2 = 3, \alpha_3 = 5, m = 12$. Remplir le tableau N[1 à 6, 1 à 12].
4. Ecrire l'algorithme correspondant (qui remplit le tableau, étant donné s et m).
5. Quelle est la complexité de votre algorithme.

Exercice 42 – On considère un échiquier $n \times n$ et on s'intéresse au passage de la case $(n, 1)$ à la case $(1, n)$. Sachant que lorsque l'on est en (i, j) on peut aller en $(i, j + 1)$ ou en $(i - 1, j - 1)$ ou bien encore en $(i - 1, j + 1)$. On veut calculer le nombre de chemins ou trajets allant de la case $(n, 1)$ à la case $(1, n)$.

1. Trouver une formule de récurrence donnant $N[i, j]$, le nombre de chemins de la case (i, j) à la case $(1, n)$.
2. Comment le calcul va-t-il évoluer ? Donner l'algorithme.
3. Quelle est la complexité de cet algorithme ?

Chapitre 2

Conseils

Vous trouverez dans ce document quelques conseils pour écrire correctement des algorithmes en pseudo-code, quelques rappels de mathématiques élémentaires (utiles pour les calculs de complexité) et enfin quelques techniques usuelles de preuves. J'espère que vous ferez bon usage de ce mémento !

2.1 Pseudo-Code

Le pseudo-code est un mélange de langage naturel et de concepts de programmation de haut-niveau qui décrit les idées générales d'un algorithme. Le but du pseudo-code est de s'abstraire des contingences d'un langage de programmation particulier. Chacun écrit le pseudo-code un peu à sa manière, ce n'est en rien un langage avec des règles claires (une grammaire) que l'on peut appliquer de manière automatique. Néanmoins, il reste que la plupart des gens s'accorde sur un certain nombre de règles de présentation. Ce sont celles-ci sur lesquelles je veux insister ici.

2.1.1 Quelques règles générales

Types élémentaires Généralement, les types élémentaires de données sont les suivants : `Entier`, `Réel`, `Caractère`.

Variables Les variables sont typées et portent un nom facilement identifiable.

Affectation Il y a plusieurs façons de la noter soit $\leftarrow$ soit $=$ (à la manière du langage C) soit $:=$ (à la manière de Pascal).

Égalité Là encore deux façons de l'écrire soit $=$, soit $==$ (pour la différencier de l'affectation comme en C). On notera la différence (non égalité) $<>$ ou bien $\neq$.

Instructions Ce que j'appellerai Instructions de manière générale regroupe 2 réalités : Soit une instruction simple terminée par un `«;»` soit un bloc d'instructions (suite d'instructions généralement identifiable par l'indentation du programme).

Conditionnelle `Si` *condition* `alors` *Instructions si la condition est vraie* `sinon` *Instructions si la condition est fausse* `Fin Si`

Boucles Plusieurs types de boucles sont possibles :

- `Tant Que` *condition* `faire` *Instructions* `Fin Tant Que`
- `Répéter` *Instructions* `tant que` *condition* `Fin Répéter`
- `Pour` *variable* `de initialisation à` *valeur finale* `faire` *Instructions* `Fin Pour`

Retour de valeur `Retourner` *valeur*. Cette instruction stoppe l'exécution de la fonction en cours et retourne la valeur spécifiée.

Tableaux Un tableau A de type `tableau type1[n]` est une suite contiguë d n éléments (ou cases) notés $A[1], A[2], \ldots, A[n]$. Chaque élément $A[i]$ est une variable de type `type1`. Les tableaux à deux ou plusieurs dimensions sont définis de la même façon, c'est-à-dire `tableau type1[n1,n2, ..., n3]`. Par exemple pour un tableau bi-dimensionnel, $A[i, j]$ donne accès au $j$ème élément de la $i$ème ligne du tableau A et est du même type que A.

Enregistrements Ils permettent de définir des types structurés. On les représentera généralement de la façon suivante : `enregistrement` *nom* `{` *champ1 : type1 ; champ2 : type 2 ; ... champn : typen*`}` L'accès au champ d'une variable structurée se fait par l'opérateur . (point).

Pointeurs Généralement, on ne descend pas au niveau des pointeurs pour les descriptions d'algorithmes, cela peut cependant se produire dans certains cas. Pour déclarer une variable de type pointeur on utilise le symbole ↑ `NIL` représente un pointeur qui ne pointe sur rien.

Entête d'une fonction *Nom_de_la_fonction*(*liste de paramètres avec leurs types*) : *type renvoyé.* Il faut préciser ensuite quelles sont les variables modifiées et les variables non modifiées.

Corps d'une fonction Après l'entête de la fonction, on précise quelles sont les variables locales puis les Instructions de la fonction.

2.1.2 Quelques exemples

Voici quelques exemples d'algorithmes vus en cours et repris ici.

Algorithme 2.1 Somme des carrés des entiers entre m et n (version itérative)

```
SommeCarrés(m : entier, n : entier) : entier
▷ Entrées : m et n
▷ Sortie : somme des carrés des entiers entre m inclus et n inclus, si m ≤ n, et 0 sinon.
Début
▷ Variables locales
   i : entier ;
   som : entier ;

   som ← 0 ;
   pour i de m à n faire
      som ← som + i * i ;
   fin pour

   retourner som

Fin
```

Algorithme 2.2 Somme des carrés des entiers entre m inclus et n inclus (version récursive)

```
SommeCarrés(m : entier, n : entier) : entier
▷ Entrées : m et n
▷ Sortie : somme des carrés des entiers entre m et n.
▷ Pré-condition : m ≤ n

Début
    si (m <> n)
        retourner (m*m + SommeCarrés(m+1,n)) ;
    sinon
        retourner (m*m) ;
    fin si

Fin
```

Il y a plusieurs choses à remarquer dans les algorithmes 1 et 2 :

— Tout d'abord, dans l'algorithme 2, on remarque que l'on a précisé les pré-conditions. Autrement dit, les conditions qui doivent être remplies avant l'utilisation de cet algorithme. En effet, si ces conditions ne sont pas remplies alors l'algorithme rend un résultat erroné.

— Dans l'algorithme 1, aucune pré-condition n'est précisée, toutefois la boucle `Pour`, ne s'exécutera que si la variable `i` n'a pas une valeur plus grande que n. Donc les conditions de validité de l'algorithme seront ici encore bien remplies.

Algorithme 2.3 Recherche dans une liste chaînée

```
Recherche(L : ↑ Element, x : entier) : booléen
▷ Entrées : x (élément recherché), L (tête de liste)
▷ Sortie : vrai si l'élément x a été trouvé dans la liste L, faux sinon.
▷ Pré-condition : La liste L est triée par ordre croissant

Début
▷ Variable Locale
    p : ↑ Element
    p ← L ;
    tant que (p <> NIL)  faire
        si (p.val < x)
            p ← p.suiv
        sinon  si (p.val = x)
            retourner  vrai
        sinon
            retourner  faux
        fin si
    fin tant que
```

Dans ce dernier algorithme, il faut noter que j'ai fait appel à un type structuré `Element` défini de la manière suivante :

`Enregistrement Element { val : entier; suiv : ↑ Element}`. Attention ici, j'ai explicitement fait apparaître une boucle `tant que` avec une seule condition qui est que la liste ne soit pas vide.

On aurait pu écrire une boucle qui intègre les 2 conditions et donc avoir une boucle du type `tant que p <> NIL et p.val < x)`, toutefois cette écriture ne fonctionne que si le `et` est paresseux, à savoir si la première condition est fausse la seconde condition n'est pas évaluée. Ceci est vrai dans un certain nombre de langages de programmation mais pas dans tous. Si le `et` est passif alors la seconde condition est évaluée et dans ce cas, votre programme écrit à l'aide de l'algorithme ne pourra pas s'exécuter correctement. Il faut donc être très prudent lorsque l'on écrit de tels algorithmes. C'est pourquoi j'ai choisi de présenter ici un algorithme sans le `et` comme cela j'évite le piège du `et` actif ou passif. Cela n'a ici de fait, plus aucune importance.

Algorithme 2.4 Recherche dichotomique

Dicho(x : entier, S : tableau d'entiers, g : entier, d : entier) : booléen
▷ **Entrées :** ***x (élément recherché), S (espace de recherche), g (indice de gauche), d (indice de droite)***
▷ **Sortie :** ***vrai si l'élément x a été trouvé dans le tableau S entre les indices g et d, faux sinon.***
▷ **Pré-condition :** ***g et d sont des indices valides du tableau S.***

```
Début
▷ Variable Locale
  m : entier;
  si (g < d)
     m ← ⌊(g+d)/2⌋;
     si (x = S[m])
        retourner vrai;
     sinon si (x < S[m])
        retourner (Dicho(x,S,g,m-1));
     sinon
        retourner (Dicho(x,S,m+1,d));
     fin si
  sinon
     retourner faux;
  fin si

Fin
```

2.2 Mathématiques élémentaires

Voici quelques rappels de mathématiques élémentaires.

logarithmes

- $\log_b(xy) = \log_b x + \log_b y$
- $\log_b(x/y) = \log_b x - \log_b y$
- $\log_b x^\alpha = \alpha \log_b x$
- $\log_b x = \frac{\ln x}{\ln b} = \frac{\log_{10} x}{\log_{10} b}$

exposants

- $a^{(b+c)} = a^b a^c$
- $a^{bc} = (a^b)^c$
- $\frac{a^b}{a^c} = a^{(b-c)}$
- $b = a^{\log_a b}$
- $\sqrt[n]{x} = x^{\frac{1}{n}}$

Parties entières

- Partie entière inférieure : $\lfloor x \rfloor$ = le plus grand entier inférieur ou égal à x.

— Partie entière supérieure : $\lceil x \rceil$ = le plus petit entier supérieur ou égal à x.
— $x - 1 \leq \lfloor x \rfloor \leq x \leq \lceil x \rceil \leq x + 1$
Par exemple, $\lfloor 3.8 \rfloor = 3$ et $\lfloor -3.8 \rfloor = -4$. De même $\lceil 3.8 \rceil = 4$ et $\lceil -3.8 \rceil = -3$.

Somme

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + (n-1) + n = \frac{n(n+1)}{2}$$

2.3 Techniques de preuves

2.3.1 Les techniques efficaces

Exemple

— Pour montrer qu'une propriété est fausse, il suffit de trouver un contre-exemple qui ne vérifie pas la propriété. Par exemple si la question est *Montrer que l'algorithme de coloration séquentielle est optimal* ce qui revient à dire que *pour tout graphe, cet algorithme permet de trouver une coloration optimale*, il suffit d'en trouver un pur lequel ça ne marche pas pour montrer que l'affirmation est fausse. Le contre-exemple du cours (un graphe biparti) montrer que dans une famille de cas, l'algorithme est non optimal.
— On peut montrer qu'une propriété d'existence est vraie à l'aide d'un exemple. Par exemple, si la question est *existe-t-il des graphes tels que leur indice chromatique est strictement supérieur au degré du graphe.* Vous pouvez répondre en exhibant un exemple de graphe : le cycle à 5 sommets qui est de degré 2 et qui nécessite 3 couleurs pour colorier ses arêtes.

Absurde

Il s'agit de montrer que la négation de l'énoncé aboutit à une absurdité. Le schéma de la démonstration est donc le suivant : on prend comme hypothèse la négation de l'énoncé (pour éviter tout risque d'erreur, on écrit cette hypothèse). En l'utilisant, on arrive à un résultat que l'on sait faux. On en conclut que l'hypothèse était fausse et donc que sa négation, l'énoncé est juste.

Par exemple dans la démonstration du théorème suivant

Théorème 1 *Un graphe orienté G est acyclique si et seulement si un parcours en profondeur de G ne rencontre jamais un arc (u, v) tel que u est un descendant de v dans une arborescence en profondeur.*

Preuve.

1. On va d'abord prouver que si on a un DAG alors il n'existe pas d'arc (u, v) tel que u est un descendant de v dans une arborescence en profondeur. Pour cela, on va supposer le contraire : il existe un arc (u, v) tel que u est un descendant de v dans une arborescence en profondeur. Dans ce cas là, le sommet v est un ancêtre du sommet u dans la forêt en profondeur. Ce qui implique qu'il existe un chemin de v à u dans G, donc avec l'arc (u, v) on a un circuit ! ce qui est absurde, donc il n'existe pas d'arc (u, v) tel que u est un descendant de v dans une arborescence en profondeur.
2. On va maintenant partir du fait qu'il n'existe pas d'arc (u, v) tel que u est un descendant de v dans une arborescence en profondeur pour montrer que le graphe G est sans circuit. De la même façon que dans la première partie de la preuve, on va partir de la supposition contraire, c'est-à-dire qu'il existe un circuit C dans le graphe. Soit v le premier sommet découvert par un parcours en profondeur dans le circuit C et soit (u, v) l'arc précédent v dans C. A la date $d[v]$, il existe un chemin composé de sommets blancs (non encore découverts) de v à u. D'après le théorème du chemin blanc, le sommet u devient un descendant de v dans la forêt en profondeur. L'arc (u, v) est donc un arc tel que u est un descendant de v dans l'arborescence en profondeur. C'est une absurdité puisque c'est contraire à notre hypothèse de départ, donc il ne peut pas exister de circuit dans le graphe G.

□

Induction (récurrence)

La preuve se fait en 2 temps :

1. Prouver le cas de base
2. En considérant que la propriété est vérifiée pour n, il faut montrer que la propriété est aussi vraie pour $n+1$.

Par exemple, pour montrer la propriété des arbres binaires suivante :

Lemme 1 *Un arbre binaire localement complet*[1] *ayant n nœuds internes a $(n+1)$ nœuds externes.*

Preuve. La preuve va se faire par récurrence sur le nombre de nœuds internes de l'arbre.

Cas de base Le plus petit arbre localement complet est l'arbre réduit à un seul nœud. Cet arbre a 0 nœud interne et 1 nœud externe. La propriété est donc vérifiée.

Récurrence Supposons cette propriété vraie pour tous les arbres ayant moins de n nœuds internes c'est-à-dire $\forall k \leq n$, tout arbre binaire localement complet ayant k nœuds internes a $k+1$ nœuds externe.

Soit B un arbre constitué d'une racine o et de deux sous-arbres l'un gauche B_1 et l'autre droit B_2. B a n nœuds internes. Soit n_1 le nombre de nœuds internes de l'arbre B_1 et n_2, le nombre de nœuds internes de l'arbre B_2. On a donc $n = n_1 + n_2 + 1$.

Par hypothèse de récurrence, B_1 (resp. B_2) a $n_1 + 1$ (resp. $n_2 + 1$) nœuds externes. Or les nœuds externes de B sont les nœuds externes de B_1 et de B_2, donc le nombre de nœuds externes de B est $n_1 + 1 + n_2 + 1 = n + 1$. Ce qui termine la preuve puisque nous avons montré que si la propriété était vraie pour tous les arbres ayant moins de n nœuds internes alors elle était vraie pour tous les arbres ayant n nœuds internes. De plus, la propriété est vraie pour les arbres les plus petits (avec un seul nœud). Donc la propriété est vraie pour tous les arbres.

□

2.3.2 Les techniques absolument inefficaces :-)

— Donner un exemple pour une propriété générale. Exemple : *montrer que tous les entiers impairs sont premiers.* Réponse : C'est vrai pour 3 donc c'est vrai pour tous!!!!

— Preuve par excès d'agitations des mains (cela peut éventuellement être utile pour un oral, mais généralement c'est très peu convaincant).

— Preuve par diagramme incompréhensible (très en vogue lors des examens écrits mais est tout aussi inefficace que le précédent).

— Preuve par intimidation : *"Cette preuve est tellement évidente que seul un idiot serait incapable de la comprendre."* La notation en général est tout aussi évidente :-)

1. Un arbre binaire est localement complet s'il est binaire non vide et chaque nœud a 0 ou 2 fils. On appelle nœud externe un nœud ayant 0 fils et nœud interne un nœud ayant 2 fils.

Chapitre 3

Résumés de cours

3.1 Introduction à la complexité des algorithmes

— *Mesure intrinsèque* de la complexité de l'algorithme indépendamment de l'implémentation.
— Permet la *comparaison* entre différents algorithmes pour un même problème.

Définition 1. Différentes Mesures

— Complexité en temps
— Complexité en espace

But : *« Sur toute machine, et quel que soit le langage utilisé, l'algorithme α est meilleur que l'algorithme β pour des données de grande taille. »*

3.1.1 Quelques règles

$cout(x)$: nbre d'op. élémentaires de l'ens. d'instructions x.

— **Séquence d'instructions** : $x_1; x_2;$

$$cout(x_1; x_2;) = cout(x_1;) + cout(x_2;)$$

— **Les boucles simples** : *tant que condition faire* x_i ;

$$cout(boucle) = \sum_{i=1}^{n} (cout(x_i) + cout(condition))$$

— **Conditionnelle** : *Si condition alors* x_{vrai}; *sinon* x_{faux};

$$cout(conditionnelle) \leq cout(condition) + \max(cout(x_{vrai}); cout(x_{faux}))$$

3.1.2 Grandeurs

— Caractérisation du comportement d'un algorithme $\mathcal{A}$ sur l'ensemble des données D_n de taille n.
— $Cout_{\mathcal{A}}(d)$: coût de l'algorithme $\mathcal{A}$ sur la donnée d.

— **Complexité dans le meilleur cas :**

$$Min_{\mathcal{A}}(n) = \min\{Cout_{\mathcal{A}}(d), d \in D_n\}$$

— **Complexité dans le pire cas :**

$$Max_{\mathcal{A}}(n) = \max\{Cout_{\mathcal{A}}(d), d \in D_n\}$$

— **Complexité en moyenne :**

$$Moy_{\mathcal{A}}(n) = \sum_{d \in D_n} p(d) \times Cout_{\mathcal{A}}(d)$$

Définition 2. O **« Borne Supérieure »**

Soient f et $g : \mathbb{N} \to \mathbb{R}_+$: $f = O(g)$ ssi $\exists c \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$ tels que :

$$\forall n > n_0, f(n) \leq c \times g(n)$$

Définition 3. Ω **« Borne Inférieure »**

Soient f et $g : \mathbb{N} \to \mathbb{R}_+ : f = \Omega(g)$ ssi $\exists c \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$ tels que :

$$\forall n > n_0, 0 \leq c \times g(n) \leq f(n)$$

Définition 4. Θ

$$f = \Theta(g) \text{ ssi } f = O(g) \text{ et } f = \Omega(g)$$

$\exists c, d \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$ tels que :

$$\forall n > n_0, d \times g(n) \leq f(n) \leq c \times g(n)$$

3.2 Etudes de complexité

3.2.1 Structures Linéaires

Eléments d'un même type stockés dans :
- un tableau
- une liste

Opérations sur les structures linéaires
- Insérer un nouvel élément
- Supprimer un élément
- Rechercher un élément
- Afficher l'ensemble des éléments
- Concaténer deux ensembles d'élements
- ...

Définition des structures
- Un tableau

```
Enregistrement Tab {
    T[NMAX] : entier;
    Fin     : entier;
}
```

- Une liste

```
Enregistrement Elément {
   Val     : entier;
   Suivant : ↑ Elément;
}
```

3.2.2 Recherche et insertion dans les structures non triées

Algorithme 3.1 Recherche dans un tableau non trié

```
Recherche(S : Tab, x : entier) : booléen
▷ Entrées : S (un tableau), x (élément recherché)
▷ Sortie : vrai si l'élément x a été trouvé dans le tableau S, faux sinon.
Début
▷ Variable Locale
    i : entier;
pour i de 1 à S.Fin faire
    si (S.T[i] = x)
        retourner vrai;
    fin si
fin pour
retourner faux;
Fin
```

— **Opération fondamentale :** comparaison
— **A chaque itération :**
  — 1 comparaison (`Si ... Fin Si`)
  — 1 comparaison (`Pour ... Fin Pour`)
— **Nombre d'itérations maximum :** nombre d'éléments du tableau
— **Complexité :** Si n est le nombre d'éléments du tableau **O(n)**.

Algorithme 3.2 Insertion dans un tableau non trié

```
Insertion(S : Tab, x : entier)
▷ Entrées : S (un tableau), x (élément recherché)
▷ Sortie : le tableau S dans lequel x a été inséré.
Début
    S.Fin ← S.Fin + 1;
    S.T[S.Fin] ← x;
Fin
```

— **Opération fondamentale :** affectation
— **Nombre d'opérations fondamentales** : 2 affectations.
— **Complexité : O(1)** (Temps constant).

Algorithme 3.3 Recherche dans une liste non triée

```
Recherche(L : ↑ Elément, x : entier) : booléen
▷ Entrées : L (tête de la liste), x (élément recherché)
▷ Sortie : vrai si l'élément x a été trouvé dans la liste L, faux sinon.
Début
    si (L = NIL)
        retourner faux;
    sinon si (L.Val = x)
        retourner vrai;
    sinon
        retourner Recherche(L.Suivant,x);
    fin si
Fin
```

Algorithme 3.4 Recherche dans une liste non triée

```
Recherche(L : ↑ Elément, x : entier) : booléen
▷ Entrées : L (tête de la liste), x (élément recherché)
▷ Sortie : vrai si l'élément x a été trouvé dans la liste L, faux sinon.
▷ Variable Locale
    p : ↑ Element ;
Début
    p ← L ;
    tant que (p ≠ NIL)  faire
        si (p.Val ≠ x)
            p ← p.Suivant ;
        sinon
            retourner  vrai ;
        fin si
    fin tant que
    retourner  faux ;
Fin
```

— **Opération fondamentale :** comparaison
— **A chaque itération :**
  — 1 comparaison (`Si ... Fin Si`)
  — 1 comparaison (`Tant que ... Fin Tant Que`)
— **Nombre d'itérations maximum :** au pire le nombre d'éléments de la liste
— **Complexité :** Si n est le nombre d'éléments de la liste $\mathbf{O(n)}$.

Algorithme 3.5 Insertion dans une liste non triée

```
Insertion(L : ↑ Element, x : entier)
▷ Entrées : L (tete de liste), x (élément recherché)
▷ Sortie : la liste L dans laquelle x a été inséré.
▷ Variable Locale
    p : ↑ Element ;
Début
    p.Val ← x ;
    p.Suivant ← L ;
    L ← p ;
Fin
```

— **Opération fondamentale :** affectation
— **Nombre d'opérations fondamentales** : 3 affectations.
— **Complexité : O(1)** (Temps constant).

3.2.3 Recherche et insertion dans les structures triées

Algorithme 3.6 Insertion dans un tableau trié

```
Insertion(S : Tab, x : entier)
▷ Entrées : S (un tableau), x (élément recherché)
▷ Sortie : le tableau S dans lequel x a été inséré.
▷ Pré-condition : le tableau S trié par ordre croissant.
▷ Variable Locale
    i,k :entiers;
Début
    si (S.Fin = -1)
        S.Fin ← 0;
        S.T[S.Fin] ← x;
    sinon
        i ← 0;
        tant que (i < S.Fin et S.T[i] < x)
            i ← i + 1;
        fin tant que
        si (i = S.Fin et S.T[i] < x)
            k ← S.Fin + 1;
        sinon
            k ← i;
        fin si
        pour i de S.Fin + 1 â k en décroissant  faire
            S.T[i] ← S.T[i-1];
        fin pour
        S.T[k] ← x;
        S.Fin ← S.Fin + 1;
    fin si
Fin
```

- **Opération fondamentale :** affectation
- **Recherche de la bonne position :** k affectations
- **Décaler à droite :** $n - k$ affectations
- **Insérer élément :** 1 affectation
- **Total :** $n + 2$ affectations
- **Complexité : O(n)** si n est le nombre d'éléments du tableau.

Algorithme 3.7 Recherche dichotomique

```
Recherche(x : entier, S : tableau, g : entier, d : entier) : booléen
▷ Entrées : x (élément recherché), S (espace de recherche), g (indice de gauche), d (indice de droite)
▷ Sortie : vrai si l'élément x a été trouvé dans le tableau S entre les indices g et d, faux sinon.
▷ Pré-conditions : g et d sont des indices valides du tableau S et S est trié par ordre croissant.
▷ Variable Locale
    m : entier;
Début
    si (g < d)
        m ← ⌊(g+d)/2⌋;
        si (x = S.T[m])
            retourner  vrai;
        sinon  si (x < S.T[m])
            retourner (Recherche(x,S,g,m-1));
        sinon
            retourner (Recherche(x,S,m+1,d));
        fin si
    sinon
        retourner  faux;
    fin si
Fin
```

- **Opération fondamentale :** comparaison
- ***A chaque appel récursif, on diminue l'espace de recherche par 2*** et on fait au pire 2 comparaisons
- **Complexité :** Au pire on fera donc $\mathbf{O}(\log_2 \mathbf{n})$ appels et la complexité est donc en $\mathbf{O}(\log_2 \mathbf{n})$.

Algorithme 3.8 Recherche dans une liste chaînée

```
Recherche(L : ↑ Element, x : entier) : booléen
▷ Entrées : x (élément recherché), L (tête de liste)
▷ Sortie : vrai si l'élément x a été trouvé dans la liste L, faux sinon.
▷ Pré-condition : La liste L est triée par ordre croissant
▷ Variable Locale
    p : ↑ Element ;
Début
    p ← L ;
    tant que (p ≠ NIL) faire
        si (p.Val < x)
            p ← p.Suivant ;
        sinon si (p.Val = x)
            retourner vrai ;
        sinon
            retourner faux ;
        fin si
    fin tant que
Fin
```

— **Opération fondamentale :** comparaison
— **A chaque itération :**
  — Une comparaison (`Si ... Fin Si`)
  — Une comparaison (`Tant que ... Fin Tant Que`)
— **Nombre d'itérations :**au pire le nombre d'éléments de la liste.
— **Complexité :**si n est le nombre d'éléments de la liste : $\mathbf{O(n)}$.

Algorithme 3.9 Insertion dans une liste chaînée

```
Insérer(L : ↑ Element, x : entier)
▷ Entrées : x (élément à insérer), L (tête de liste)
▷ Sortie : la liste L dans laquelle l'élément x a été inséré à sa place..
▷ Pré-condition : La liste L est triée par ordre croissant
▷ Variable Locale
    p : ↑ Element ;
Début
    si (L = NIL ou L.Val >= x)
        p.Val ← x ;
        p.Suivant ← L ;
        L ← p ;
    sinon
        L.Suivant ← Insérer(L.Suivant,x) ;
    fin si
Fin
```

— **Opération fondamentale :** comparaison
— **A chaque itération :** 2 comparaisons
— **Nombre d'itérations :** au pire il faut parcourir tous les éléments de la liste.
— **Complexité :** si n est le nombre d'éléments de la liste : $\mathbf{O(n)}$.

3.2.4 Résumé

Complexité de l'insertion

	Eléments triés	Eléments non triés
Tableau	$O(n)$	$O(1)$
Liste	$O(n)$	$O(1)$

Complexité de la recherche

	Eléments triés	Eléments non triés
Tableau	$O(\log_2 n)$	$O(n)$
Liste	$O(n)$	$O(n)$

3.3 Les tris

3.3.1 tri par insertion

Algorithme 3.10 Tri par insertion

```
TriInsertion(T : tableau d'entiers, TailleMax : entier)
▷ Variables Locales
    TC, i, p, temp : entiers
Début
pour TC de 1 à TailleMax - 1 faire
    temp ← T[TC + 1]
    p ← 1
    tant que T[p] < temp faire                    ▷ Chercher la position p
        p ← p + 1
    fin tant que
    pour i de TC en décroissant à p faire         ▷ Décaler les éléments
        T[i+1] ← T[i]
    fin pour
    T[p] ← temp
fin pour
Fin
```

Complexité pour n éléments

— Le corps de la boucle est exécuté $n-1$ fois
— Une itération :
  — Recherche de la position : p
  — Décalage des éléments : $TC - p$
  — Total : TC
— Au total :

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

La complexité du tri par insertion est en $O(n^2)$.

3.3.2 Tri par permutation

Algorithme 3.11 Tri par permutation

```
TriPermutation(T : tableau d'entiers, TailleMax : entier)
▷ Variables Locales
    i, TC : entiers
Début
pour TC de 2 à TailleMax faire
    pour i de TailleMax en décroissant à TC faire
        si T[i-1] > T[i] faire
            T[i-1] ↔ T[i]
        fin si
    fin pour
fin pour
Fin
```

Complexité pour n éléments

— Boucle externe : $n-2$ fois
— Boucle interne : $TailleMax - TC$ fois
— Total : $\frac{(n-1)(n-2)}{2}$

La complexité du tri par permutation est en $O(n^2)$.

3.3.3 Tri fusion

Algorithme 3.12 Tri Fusion

```
TriFusion(T : tableau d'entiers, p : entier, r : entier)
▷ p et r sont les indices entre lesquels on veut trier le tableau. On suppose p ≤ r.
 Début
     si p < r  faire
        q ← ⌊(p+r)/2⌋
        TriFusion(T,p,q)
        TriFusion(T,q+1,r)
        Fusion(T,p,q,r)

     fin si
 Fin
```

Algorithme 3.13 Tri Fusion

```
Fusion(T : tableau d'entiers, p : entier, q : entier, r : entier)
▷ Entrées : T : tableau d'entiers. p, q et r : indices entre lesquels on veut trier le tableau avec p ≤ q ≤ r.
▷ Sortie : T : tableau trié entre les indices p et r.
▷ Pré-condition : T tableau trié entre les indices p et q et T trié entre les indices q + 1 et r
▷ Variables locales : i,j,k : entiers et B : tableau d'entiers

 Début
i ← p ; k ← p ; j ← q + 1 ;
 tant que (i ≤ q et j ≤ r)  faire
     si T[i] < T[j]  faire
        B[k] ← T[i]
        i ← i + 1

     sinon
        B[k] ← T[j]
        j ← j + 1

     fin si
    k ← k + 1
 fin tant que
 tant que i ≤ q  faire
    B[k] ← T[i]
    i ← i + 1
    k ← k + 1
 fin tant que
 tant que j ≤ r  faire
    B[k] ← T[j]
    j ← j + 1
    k ← k + 1
 fin tant que
T ← B
 Fin
```

Complexité pour n éléments

— Intuitivement il faut résoudre :

$$Tri(n) = 2 \times Tri(\frac{n}{2}) + \Theta(n)$$

— $\Theta(n)$: complexité de la fusion

La complexité du tri fusion est en $\Theta(n \log_2 n)$.

3.3.4 Tri rapide

Algorithme 3.14 Tri Rapide

```
TriRapide(T : tableau d'entiers, p : entier, r : entier)
▷ p et r sont les indices entre lesquels on veut trier le tableau. On suppose p ≤ r.
Début
    si p < r faire
        q ← partitionner(T,p,r)
        TriRapide(T,p,q)
        TriRapide(T,q+1,r)

    fin si
Fin
```

Algorithme 3.15 Partitionner

```
Partitionner(T : tableau d'entiers, p : entier, r : entier)
▷ p et r sont les indices entre lesquels on veut trier le tableau. On suppose p ≤ r.
▷ Variables locales : i,j,pivot : entiers
Début
i ← p ; j ← r ; pivot ← T[p] ;
tant que (i < j) faire
    tant que (T[i] < pivot) faire i ← i + 1 fin tant que
    tant que (T[j] > pivot) faire j ← j - 1 fin tant que
    si (i < j) faire
        T[i] ↔ T[j]
        i ← i + 1
        j ← j - 1

    fin si
fin tant que
retourner j
Fin
```

Complexité pour n éléments

— Partitionner n éléments coûte $\Theta(n)$.

— Temps d'exécution dépend de l'équilibre ou non du partitionnement :

  — S'il est équilibré : **aussi rapide que le tri fusion**

  — S'il est déséquilibré : **aussi lent que le tri par insertion**

3.4 Les structures de données abstraites

— Mise en œuvre d'un ensemble dynamique

— Définition de données (**structuration**)

— Définition des opérations pour **manipuler** les données.

Quelques structures classiques

1. Pile
2. File
3. *Tables de hachage*
4. Tas
5. Files de priorité
6. Arbres
7. .....

3.4.1 Les Piles

Analogie avec une pile d'assiette :

LIFO (Last In First Out ou **Dernier Arrivé Premier Servi**)

— On ne peut rajouter un élément qu'au dessus de la pile
— On ne peut prendre que l'élément qui est au dessus de la pile (élément le plus récemment inséré).

Mise en œuvre à l'aide d'un tableau (*nombre maximum d'éléments dans la pile fixé*)

```
Enregistrement Pile {
   T[NMAX] : entier;
   Sommet : entier;
}
```

Algorithme 3.16 La pile est-elle vide ?

```
PileVide(p : Pile) : booléen
▷ Entrée : P (une pile)
▷ Sortie : vrai si la pile est vide, faux sinon.
 Début
 si (p.Sommet = NIL)
     retourner vrai ;
 sinon
     retourner faux ;
 fin si
 Fin
```

Complexité : $O(1)$

Algorithme 3.17 La pile est-elle pleine ?

```
PilePleine(p : Pile) : booléen
▷ Entrée : P (une pile)
▷ Sortie : vrai si la pile est pleine, faux sinon.
 Début
 si (p.Sommet = NMAX-1)
     retourner vrai ;
 sinon
     retourner faux ;
 fin si
 Fin
```

Complexité : $O(1)$

Algorithme 3.18 Insertion d'un élément

```
Insertion(p : Pile, elt : entier)
▷ Entrée : p (une pile) et elt (un entier)
▷ Sortie : la pile p dans laquelle elt a été inséré
 Début
 si (PilePleine(p) =  faux )
    p.Sommet ← p.Sommet + 1 ;
    p.T[p.Sommet] ← elt ;
 sinon
    Afficher un message d'erreur
 fin si
 Fin
```

Complexité : $O(1)$

Algorithme 3.19 Suppression d'un élément

```
Suppression(p : Pile) : entier
▷ Entrée : p (une pile) e
▷ Sortie : renvoie l'élément qui était au sommet de la pile p et supprime l'élément de la pile
▷ Variable locale :
    elt : entier ;
 Début
```

```
si (PileVide(p) =  faux )
    elt ← p.T[p.Sommet] ;
    p.Sommet ← p.Sommet - 1 ;
     retourner elt ;
sinon
    Afficher un message d'erreur
fin si
Fin
```

Complexité : $O(1)$

3.4.2 Les Files

Analogie avec une file d'attente :
FIFO (First In First Out ou **Premier Arrivé Premier Servi**)
— On rajoute un élément à la fin de la file
— On supprime l'élément qui est en tête de file.

Mise en œuvre à l'aide d'un tableau (*nombre maximum d'éléments dans la file fixé*)

```
Enregistrement File {
    T[NMAX] : entier;
    Début : entier;  Indice de l'élément la plus ancien de la file
    Fin : entier;    Indice du prochain élément à insérer dans la file
}
```

Algorithme 3.20 La file est-elle vide ?

```
FileVide(f : File) : booléen
▷ Entrée : F (une file)
▷ Sortie : vrai si la file est vide, faux sinon.
 Début
 si (f.Début = f.Fin)
     retourner  vrai ;
 sinon
     retourner  faux ;
 fin si
 Fin
```

Complexité : $O(1)$

Algorithme 3.21 La file est-elle pleine ?

```
FilePleine(f : File) : booléen
▷ Entrée : f (une file)
▷ Sortie : vrai si la file est pleine, faux sinon.
 Début
 si (f.Début = (f.Fin +1) mod NMAX)
     retourner  vrai ;
 sinon
     retourner  faux ;
 fin si
 Fin
```

Complexité : $O(1)$

Algorithme 3.22 Insertion d'un élément

```
Insertion(f : File, elt : entier)
▷ Entrée : f (une file) et elt (un entier)
▷ Sortie : la file f dans laquelle elt a été inséré
 Début
```

```
si (FilePleine(f) = faux )
    f.T[f.Fin] ← elt ;
    f.Fin ← (f.Fin + 1) mod NMAX ;
sinon
    Afficher un message d'erreur
fin si
Fin
```

Complexité : $O(1)$

Algorithme 3.23 Suppression d'un élément

```
Suppression(f : File) : entier
▷ Entrée : f (une file)
▷ Sortie : renvoie l'élément le plus ancien de la file f et supprime l'élément de la file
▷ Variable locale :
    elt : entier ;
Début
si (FileVide(f) = faux )
    elt ← f.T[f.Début] ;
    f.Début ← (f.Début + 1) mod NMAX ;
     retourner elt ;
sinon
    Afficher un message d'erreur
fin si
Fin
```

Complexité : $O(1)$

3.4.3 Les tas binaires

Un tableau T qui peut être vu comme un arbre A.

1. La **racine** du tas est en $T[1]$
2. Sachant l'indice i d'un élément (un **nœud**) du tas :
   — **Pere(i)** : élément d'indice $\lfloor \frac{i}{2} \rfloor$.
   — **Gauche(i)** : élément d'indice $2i$.
   — **Droit(i)** : élément d'indice $2i + 1$.

Propriété de tas Pour chaque nœud i autre que la racine

$$T[Pere(i)] \geq T[i]$$

Algorithme 3.24 Entasser

```
Entasser(T : tableau, i : entier)
▷ Entrée : T (un tableau) et i un indice du tableau
▷ Sortie : l'élément d'indice i est la racine d'un tas binaire
▷ Pré-Conditions : Les éléments d'indices Gauche(i) et Droit(i) sont les racines de deux tas binaires.
▷ Variables locales :
    l,r,max : entier ;

Début
l ← Gauche(i) ;
r ← Droit(i) ;
si (l ≤ taille (T) et T[l] > T[i])
    max ← l;
sinon
    max ← i ;
fin si
si (r ≤ taille (T) et T[r] > T[max])
    max ← r ;
fin si
si (max ≠ i)
    T[i] ↔ T[max] ;
    Entasser(T,max) ;
fin si
Fin
```

Complexité : $O(\log n)$

Algorithme 3.25 Construction d'un tas binaire à partir d'un tableau

```
Construire_Tas(T : tableau, i : entier)
▷ Entrée : T (un tableau)
▷ Sortie : T est un tas binaire
▷ Variables locales :
    i : entier ;
Début
pour i de ⌊ longueur(T)/2 ⌋ à 1  faire
    Entasser(T,i) ;
fin pour
Fin
```

Complexité : $O(n)$

3.4.3.1 Les files de priorités

Un ensemble dans lequel chaque élément possède une valeur et une priorité.
— **Insérer** : insère un nouvel élément dans l'ensemble.
— **Maximum** : renvoie l'élément de plus grande priorité.
— **Extraire_Max** : supprime de l'ensemble et renvoie l'élément de plus grande priorité.

Un tas permet de modéliser une file de priorité.

Algorithme 3.26 Extraire l'élément maximum

```
Extraire_Max(T : tableau) : entier
▷ Entrée : T (un tas binaire)
▷ Sortie : T est un tas binaire sans l'élément maximum et retourne l'élément maximum
▷ Variables locales :
    max : entier ;
Début
    si (taille(T) < 1)
        Afficher un message d'erreur ;
    fin si
    max ← T[1] ;
    T[1] ← T[taille(T)] ;
    taille(T) = taille(T) - 1 ;
    Entasser(T,1) ;
    retourner max ;
Fin
```

Complexité : $O(\log n)$

Algorithme 3.27 Insérer un élément dans un tas

```
Insérer(T : tableau, p :entier)
▷ Entrée : T (un tas binaire)
▷ Sortie : T est un tas binaire contenant l'élément de priorité p
▷ Variables locales :
    i : entier ;
Début
    Taille(T) ← Taille(T) + 1 ;
    i ← Taille(T) ;
    tant que (i > 1 et T[pere(i)] < p)  faire
        T[i] ← T[Pere(i)] ;
        i ← Pere(i) ;
    T[i] ← p ;
Fin
```

Complexité : $O(\log n)$

3.5 Les arbres

3.5.1 Définitions

Définition 5.

— Un **arbre** : un ensemble de nœuds reliés entre eux par des arêtes.
— Trois propriétés pour les arbres **enracinés** :

1. Il existe un nœud particulier nommé **racine**.
2. Tout nœud c autre que la racine est relié par une arête à un nœud p appelé **père** de c.
3. Un arbre est **connexe**.

— Un nœud peut avoir 0 ou plusieurs fils.
— Un nœud a exactement un père.

Une définition récursive :

— **Base :**
  — Un nœud unique n est un arbre
  — n est la racine de cet arbre.
— **Récurrence :**
  — Soit r un nouveau nœud
  — $T_1, T_2, \ldots, T_k$ sont des arbres ayant pour racine $r_1, r_2, \ldots, r_k$.
  — Création d'un nouvel arbre ayant pour racine r et on ajoute une arête entre r et r_1 r et r_2, …, r et r_k.

Définition 6.

Les **ancêtres** d un nœud : *Nœuds trouvés sur le **chemin unique** entre ce nœud et la racine.*

Définition 7.

Le nœud d est un **descendant** de a si et seulement si a est un ancêtre de d.

Définition 8.

Longueur d'un chemin = nombre d'arêtes parcourues.

Définition 9.

Les nœuds ayant le même père = **frères**.

Définition 10.

Un nœud n et tous ses descendants = **sous-arbre**

Définition 11.

Une `feuille` est un nœud qui n'a pas de fils Un nœud intérieur est un nœud qui a au moins 1 fils. Tout nœud de l'arbre est :
— Soit une feuille
— Soit un nœud intérieur

Définition 12.

La **hauteur d'un nœud** n, notée $h(n)$, est la longueur du chemin depuis la racine jusqu'à n. La **hauteur de l'arbre** T, notée $h(T)$:

$$h(T) = \max_{x \text{ nœud de l'arbre}} h(x)$$

Définition 13.

Taille de l'arbre T, notée *taille(T)* = nombre de nœuds.

Définition 14.

Nombre de feuilles noté *nf(T)*.

Définition 15.

Longueur de cheminement de l'arbre T, notée *LC(T)* = somme des longueurs de tous les chemins issus de la racine.

$$LC(T) = \sum_{x \text{ nœud de } T} h(x).$$

Longueur de cheminement externe de l'arbre T, notée *LCE(T)* = somme des longueurs de tous les chemins aboutissant à une feuille issus de la racine.

$$LCE(T) = \sum_{x \text{ feuille de } T} h(x).$$

Définition 16.

Profondeur moyenne de l'arbre T, notée *PC(T)*= moyenne des hauteurs de tous les nœuds.

$$PC(T) = \frac{LC(T)}{taille(T)}$$

Profondeur moyenne externe de l'arbre T, notée *PCE(T)* = moyenne des longueurs de tous les chemins issus de la racine et se terminant par une feuille.

$$PCE(T) = \frac{LCE(T)}{nf(T)}$$

3.5.2 les arbres binaires

Définition 17.

Tous les nœuds d'un arbre binaire ont $0, 1$ ou 2 fils.

— **Fils gauche** de n = racine du sous-arbre gauche de n.
— **Fils droit** de n = racine du sous-arbre droit de n.

Définition 18.

Bord gauche de l'arbre = le chemin depuis la racine en ne suivant que des fils gauche.
Bord droit de l'arbre = le chemin depuis la racine en ne suivant que des fils droits.

Quelques arbres binaires particuliers :

1. Arbre binaire *filiforme*
2. Arbre binaire *complet* :
   — 1 nœud à la hauteur 0

— 2 nœuds à la hauteur 1
— 4 nœuds à la hauteur 2
— ...
— 2^h nœuds à la hauteur h.
— Nombre total de nœuds d'un arbre de hauteur h :

$$2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$$

3. Arbre binaire *parfait* :
   — Tous les niveaux sont remplis sauf le dernier.
   — Les feuilles sont le plus à gauche possible.
4. Arbre binaire *localement complet* : chaque nœud a 0 ou 2 fils.

Lemme 2

$$h(T) \leq taille(T) - 1$$

Preuve. Egalité obtenue pour un arbre filiforme. □

Lemme 3 *Pour tout arbre binaire T de taille n et de hauteur h on a :*

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Preuve.
— Arbre filiforme : arbre de hauteur h ayant le plus petit nombre de nœuds : $n = h + 1$ (seconde inégalité).
— Arbre complet : arbre de hauteur h ayant le plus grand nombre de nœuds : $n = 2^{h+1} - 1$ (première inégalité).

□

Corollaire 1 *Tout arbre binaire non vide T ayant f feuilles a une hauteur $h(T)$ supérieure ou égale à $\lceil \log_2 f \rceil$.*

Lemme 4 *Un arbre binaire localement complet ayant n nœuds internes a $(n+1)$ feuilles.*

Mise en œuvre

```
Enregistrement Nœud {
   Val     : entier;
   Gauche : ↑ Nœud;
   Droit : ↑ Nœud;
}
```

Algorithme 3.28 Parcours en largeur d'un arbre binaire

```
ParcoursEnLargeur(r : Nœud)
▷ Entrée : r (la racine d'un arbre)
▷ Sortie : traitement de tous les nœuds de l'arbre enraciné en r
▷ Variables locales :
   ce_niveau, niveau_inférieur : File;
   o : Nœud;
Début
   ce_niveau ← { r };
   tant que (ce_niveau est non vide) faire
      niveau_inférieur = { };
      pour chaque nœud o de ce_niveau faire
         traiter o;
         niveau_inferieur ← niveau_inférieur ∪ enfants de o.
      fin pour
      ce_niveau ← niveau_inférieur;
   fin tant que
Fin
```

Algorithme 3.29 Parcours en profondeur d'un arbre binaire

```
ParcoursEnProfondeur(r : Nœud)
▷ Entrée : r (la racine d'un arbre)
▷ Sortie : traitement de tous les nœuds de l'arbre enraciné en r
Début
    si r = ∅
        traitement de l'arbre vide
    sinon
        traitement_prefixe(r) ;
        ParcoursEnProfondeur(r.Gauche) ;
        traitement_infixe(r) ;
        ParcoursEnProfondeur(r.Droit) ;
        traitement_postfixe(r) ;
    fin si
Fin
```

3.5.3 Arbres généraux et forêts

— **Arbre général** : arbre où les nœuds peuvent avoir un nombre quelconque de fils.
— **Forêt** : collection d'arbres en nombre quelconques

Mise en œuvre

— Tableau de fils
— Fils ainé et frère droit (bijection avec les arbres bianires).

Algorithme 3.30 Parcours prefixe d'un arbre général

```
Prefixe(r : Nœud)
▷ Entrée : r (la racine d'un arbre)
▷ Sortie : traitement de tous les nœuds de l'arbre enraciné en r
▷ Variable locale :
    t : Nœud ;
Début
    traiter le nœud r
    t ← r.FilsAine ;
    tant que t ≠ NIL faire
        Prefixe(t) ;
        t ← t.FrereDroit ;
    Ftque
Fin
```

Algorithme 3.31 Parcours postfixe d'un arbre général

```
Postfixe(r : Nœud)
▷ Entrée : r (la racine d'un arbre)
▷ Sortie : traitement de tous les nœuds de l'arbre enraciné en r
▷ Variable locale :
    t : Nœud ;
Début
    t ← r.FilsAine ;
    tant que t ≠ NIL faire
        Postfixe(t) ;
        t ← t.FrereDroit ;
    Ftque
    traiter le nœud r
Fin
```

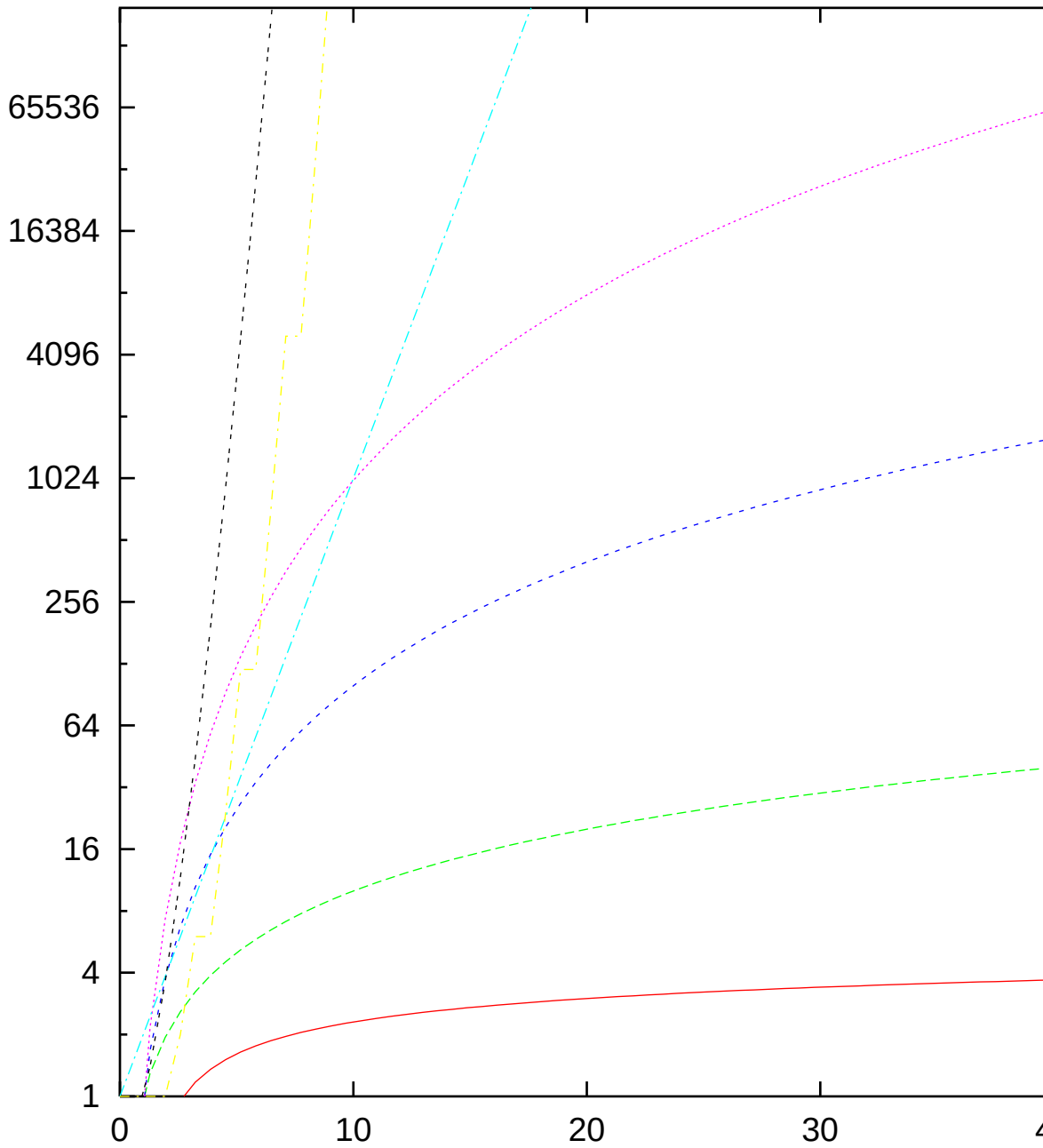


FIGURE 3.1 – Représentation des différentes fonctions $\log x, x, x^2, x^3, 2^x, x!, x^x$ en échelle logarithmique

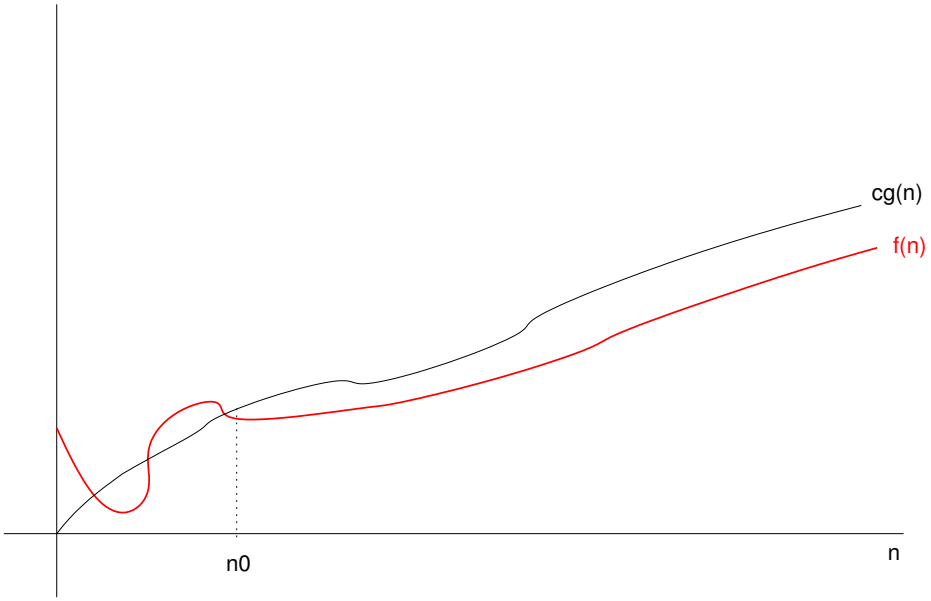


Figure 3.2 – $f(n) = O(g(n))$

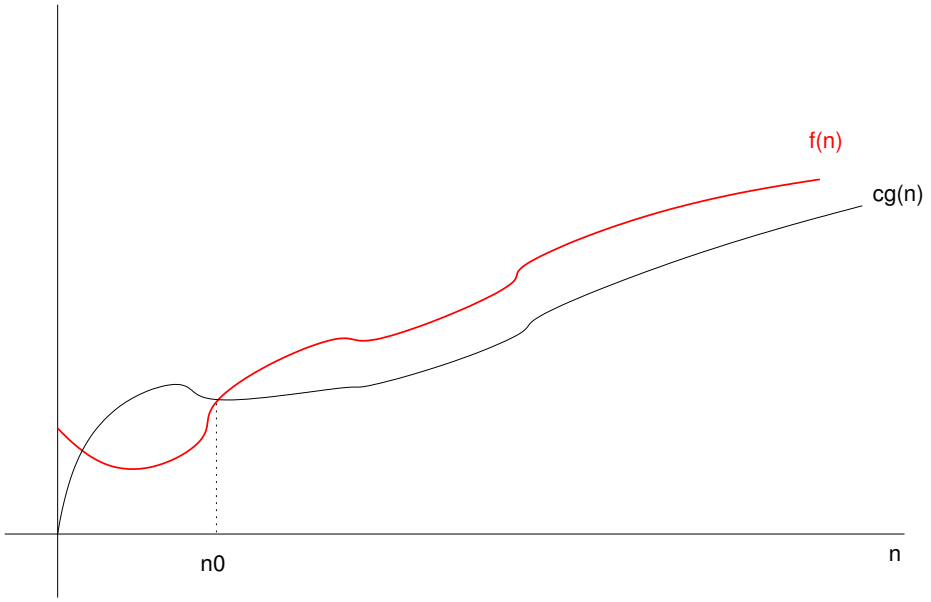


Figure 3.3 – $f(n) = \Omega(g(n))$

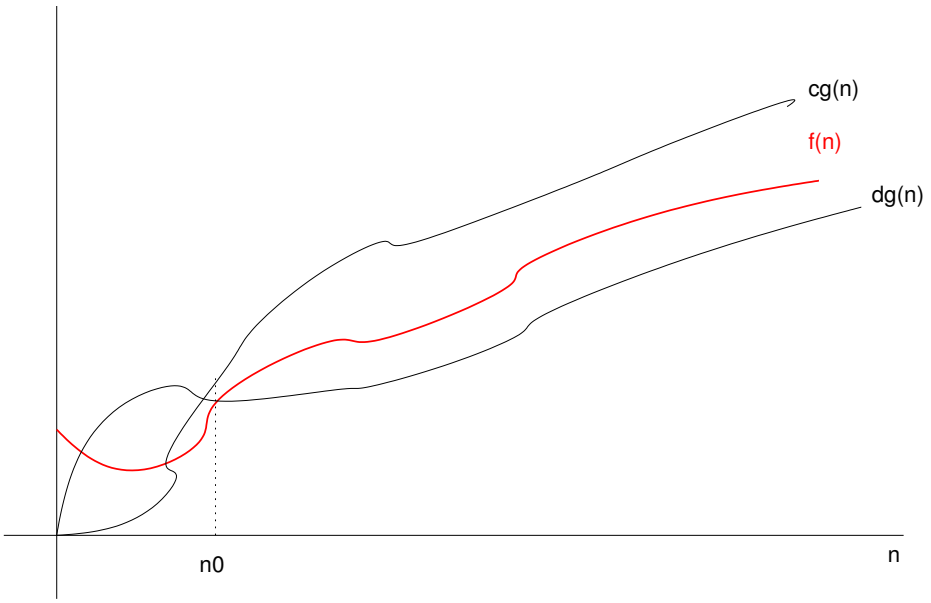


FIGURE 3.4 – $f(n) = \Theta(g(n))$

3.6 Les Arbres Binaires de Recherche

3.6.1 Définition

Définition 19.

> Un **Arbre Binaire de Recherche** est un arbre binaire tel que pour tous les noeuds de l'arbre, tous les noeuds de son sous-arbre gauche ont une valeur plus petite que le nœud lui-même et tous les nœuds de son sous-arbre droit ont une valeur plus grande que le nœud lui-même.

Mise en œuvre

```
Enregistrement Nœud {
   Val     : entier;
   Gauche : ↑ Nœud;
   Droit : ↑ Nœud;
   Pere : ↑ Nœud;
}
```

3.6.2 Algorithmes

Algorithme 3.32 Element Maximum dans un ABR

```
ABR-Max(r : Nœud) : Entier
▷ Entrée : r (la racine d'un arbre)
▷ Sortie : l'élément maximum de l'ABR enraciné en r
 Début
     tant que x.Droit ≠ NIL  faire
         x ← x.Droit ;
     fin tant que
     retourner x ;
 Fin
```

Algorithme 3.33 Recherche d'un élément dans un ABR

```
Recherche(r : Nœud, c : Entier) : Booleen
▷ Entrée : r (la racine d'un arbre), c (l'élément recherché)
▷ Sortie : renvoie vrai si c est dans l'arbre enraciné en r, faux sinon
 Début
     si r ≠ NIL
         si r.val = c  retourner Vrai ;
         sinon  si r.val > c  retourner Recherche(r.Gauche,c) ;
         sinon  retourner Recherche(r.Droit,c) ;
     fin si
     retourner Faux ;
 Fin
```

Algorithme 3.34 Successeur dans un ABR

```
ABR-Successeur(r : Nœud) : Noeud
▷ Entrée : r (la racine d'un arbre)
▷ Sortie : renvoie le nœud dont la valeur est immédiatement supérieure à celle de r
▷ Variables locales :
    y : Noeud ;
     Début
         si r.Droit ≠ NIL
             retourner ABR _Min(r.Droit) ;
         fin si
        y ← r.Pere ;
         tant que y ≠ NIL et r = y.Droit
             r ← y ;
             y ← y.Pere ;
         fin tant que
         retourner y ;
     Fin
```

Algorithme 3.35 Insertion d'un nœud aux feuilles

```
ABR-Inserer(r : Nœud, z : Noeud) : Noeud
▷ Entrée : r (la racine d'un ABR), z (un nouveau à insérer)
▷ Sortie : renvoie la racine de l'ABR dans lequel le nœud z a été inséré
 Début
```

```
    si r = NIL
        retourner z ;
    sinon
        si z.val ≤ r.val
            r.Gauche ← ABR-Inserer(r.Gauche, z) ;
             retourner r ;
        sinon
            r.Droite ← ABR-Inserer(r.Droite,z) ;
             retourner r ;
        fin si
    fin si
Fin
```

Complexité au pire : $O(hauteur\ de\ l'ABR)$

Ajout d'un nœud à la racine de l'arbre

Soit un arbre `a = <o',g,d>`. Ajouter le nœud `o` à `a` c'est construire l'arbre `<o,a1,a2>` tel que :
— `a1` contienne tous les nœuds dont la clé est inférieure à celle de `o`
— `a2` contienne tous les nœuds dont la clé est supérieure à celle de `o`.
Si la valeur de `o` est plus petite que la valeur de `o'` alors `a1 = g1` et `a2 = <o',g2,d>`, avec :
— `g1` = nœuds de `g` dont la valeur est inférieure à la valeur de `o`
— `g2` = nœuds de `g` dont la valeur est supérieure à la clé de `o`.
Si la valeur de `o` est plus grande que la valeur de `o'` alors `a1 = <o',g,d1>` et `a2 = d2`, avec :
— `d1` = nœuds de `d` dont la valeur est inférieure à la valeur de `o`
— `d2` = nœuds de `d` dont la valeur est supérieure à la clé de `o`.

Complexité au pire : $O(hauteur\ de\ l'ABR)$

Suppression d'un nœud

— Si c'est une feuille : suppression simple
— Si c'est un nœud avec un seul fils : suppression du nœud et raccordement de son sous-arbre à son père.
— Si le nœud a deux fils : remplacement par son successeur (et suppression du successeur).

Complexité au pire : $O(hauteur\ de\ l'ABR)$

3.6.3 les arbres AVL

Définition 20.

Soit a un nœud dans un ABR

$$desequilibre(a) = h(a.Gauche) - h(a.Droit)$$

Définition 21.

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

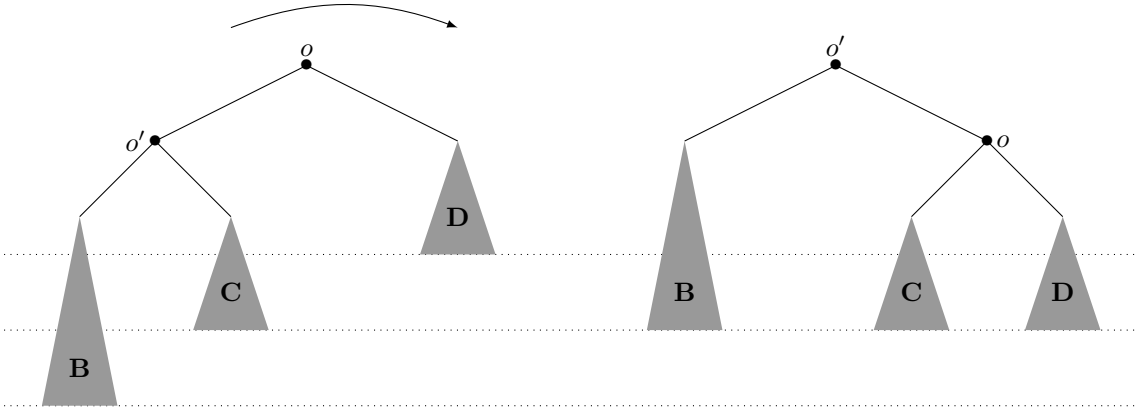
$$desequilibre(b) \in \{-1, 0, 1\}$$

Définition 22.

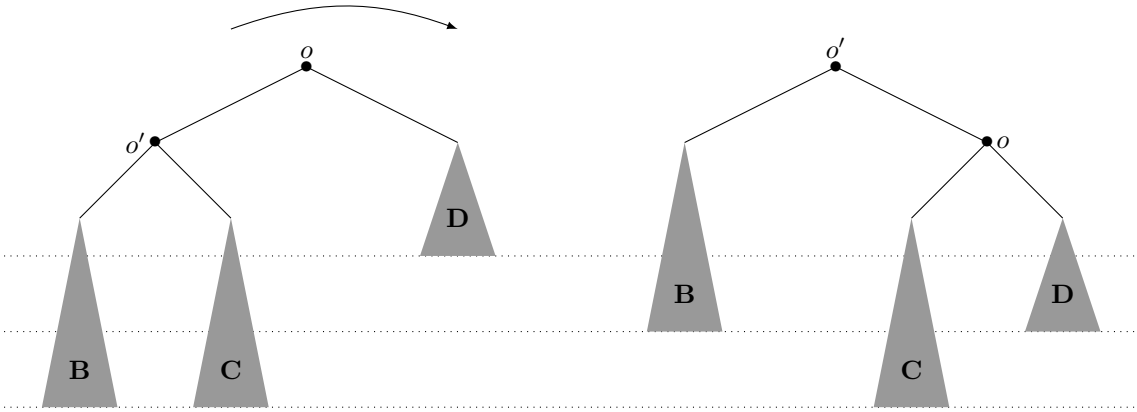
Un **arbre AVL** est un ABR H-équilibré.

Après une insertion ou une suppression d'un nœud dans un arbre AVL, il peut y avoir des nœuds qui ont un déséquilibre de +2 ou -2. Il faut alors rééquilibrer l'arbre par des opérations de rotations. Ci-dessous voici les opérations de rotations lorsqu'il existe un déséquilibre de +2. Il est facile d'en déduire les rotations lorsque le déséquilibre est de -2. Il y a trois cas de figures :

1. Déséquilibre de +1 sur le fils gauche : **Rotation Droite**



2. Déséquilibre de 0 sur le fils gauche : **Rotation Droite**



3. Déséquilibre de -1 sur le fils gauche : **Rotation Gauche-Droite**