Responsable de cette UE à Metz :
Christian Minich
christian.minich@univ-lorraine.fr
+33 3 87 54 77 99

Responsable de cette UE à Nancy :
Horatiu Cirstea
horatiu.cirstea@univ-lorraine.fr
+33 3 54 95 84 06

Langage support
C

Licence 2 Informatique
Christian MINICH (Metz)
Horatiu CIRSTEA (Nancy)
christian.minich@univ-lorraine.fr
horatiu.cirstea@univ-lorraine.fr

UNIVERSITÉ DE LORRAINE

Algorithmique et programmation III

Support de cours

Introduction

Prérequis : AP 1 et AP 2, c'est-à-dire:

- *maîtrise* de la programmation structurée (structures de contrôle, sous-programmes) ;

  Rappel : quand on conçoit un sous-programme :
  - on décide de ce qu'il doit faire
  - on établit son paramétrage (nom, type et nature de chaque paramètre)
  - on en déduit si le sous-programme est une procédure ou une fonction
  - on trouve une méthode et on la traduit en un algorithme
- *maîtrise* des listes (chaînées) en tant que TDA (SDD) ;

  Rappel :
  - un TDA (type de données abstrait ou ADT (Abstract Data Type) en anglais) est une collection de données et un ensemble d'opérations sur ces données.
  - une SDD (structure de données) est une construction d'un langage de programmation qui stoque une collection de données.
- *maîtrise* des notions de base sur les pointeurs ;
- *maîtrise* de la programmation récursive.

Pour concevoir des programmes de grande taille, fiables et efficaces, il faut :

- pouvoir comparer les mérites de plusieurs algos réalisant la même tâche, notamment leur complexité
- connaître d'autres approches (paradigmes) de la programmation → récursivité
- connaître les mérites des SDD classiques (listes, tableaux, …) afin de choisir la SDD la plus adaptée à son problème (ou d'en inventer une, au besoin) → arbres, arbres ordonnés, arbres ordonnés équilibrés, tas
- connaître des versions efficaces d'algos classiques comme le tri → tri par tas.

Usage d'un pseudo-langage :

- on distingue les procédures des fonctions
- les paramètres des sous-programmes sont qualifiés par le type et un symbole parmi : ↓ (paramètre en entrée), ↑ (paramètre en sortie) ou ↕ (paramètres en entrée-sortie)
- les variables locales ne sont parfois pas déclarées → dans un algo, toute variable qui n'est pas un paramètre est une variable locale (ou une variable globale, ce qui est interdit pour l'instant)

Exemple :

```
// Restitue la plus grande des valeurs présentes dans le tableau T
// à N éléments et dans la liste L, qui contiennent tous les deux
// des entiers par hypothèse.
// On suppose que l'une au moins des deux structures est non vide.
fonction MaxTabListe(↓tab : TabEnt ; ↓n : entier; ↓l : ListeEnt) : entier
variables
  max: entier
début
  si (n > 0) alors // tableau non vide
       max ← tab[0]  // on suppose que l'indiçage des tableaux commence à 0
  sinon  // tableau vide
      // Remarquons que L ne peut pas être vide, à ce stade donc Tete(L) est sans danger
       max ← Tete(l) ;
       l ← Reste(l) ;
  fsi // on pourrait initialiser max à -∞
  pour i de 1 à N-1 faire
       si (tab[i] > max) alors
            max ← tab[i]
       fsi
  fpour
  tant que non EstVide(l) faire
       si (tete(l) > max) alors
            max ← Tete(l)
       fsi
       l ← Reste(l)
  ftq
  retourner max
fin

TabEnt = tableau d'entiers[100]
ListeEnt = Liste[entier]
```

Rappel :

Type : `Liste[E]`

Opérations :

```
ConsVide : → Liste[E]
Cons : E x Liste[E] → Liste[E]
Tete: Liste[E] → E
Reste: Liste[E] → Liste[E]
EstVide: Liste[E] → booléen
```

Préconditions :

`Tete(l)` défini ssi non `EstVide(l)`

`Reste(l)` défini ssi non `EstVide(l)`

Axiomes :

```
Tete(Cons(e,l)) = e
Reste(Cons(e,l)) = l
EstVide(ConsVide) = vrai
EstVide(Cons(e,l)) = faux
```

Bibliographie :

- Introduction à l'algorithmique, T.H. Cormen, C.E. Leiserson, R.L. Rivest, Dunod, Paris, 1994, (MIT Press 1990), plusieurs exemplaires à la bibliothèque

Plan :

1 Rappels

Pour concevoir un programme (ou fonction ou procédure) il faut tout d'abord identifier clairement ses fonctionnalités. Il faut ensuite déterminer les entrées et les sorties ainsi que les structures de données les plus adaptées. Les algorithmes correspondants doivent être clairs et efficaces (voir section suivante pour l'analyse de la complexité).

Exemple :

```
// Calculer la moyenne, le min et le max des douze dernières températures
procedure Temperatures(↓tab : tableau de réels[12];
                       ↑moy : réel ; ↑min : réel ; ↑max : réel)
début
      moy ← tab [0]
      min ← tab [0]
      max ← tab [0]
      pour i de 1 à 11 faire
            moy ← moy + tab [i]
            si tab[i] > max alors
                  max ← tab[i]
            sinon
                si tab[i] < min alors
                  min ← tab[i]
                fsi
            fsi
      fpour
      moy ← moy/12
fin

// Ajouter une nouvelle température – la plus ancienne température est effacée.
procedure Remplacer(↓tab: tableau de réels[12]; nTemp : réel )
Début
      pour i de 0 à 10 faire
            tab[i] ← tab[i+1]
      fpour
      tab[11] ← NTemp
Fin
```

2 Introduction à l'analyse de la complexité d'un programme

2.1 Complexité en temps

Evaluer la complexité en temps = compter le nombre d'instructions élémentaires exécutées + en donner un ordre de grandeur.

2.1.1 Compter le nombre d'instructions exécutées

Evaluer la complexité *en temps* d'un programme consiste à exprimer le nombre d'instructions qu'il va exécuter (qui n'est PAS le nombre d'instructions qu'il contient) en fonction du nombre de données à traiter.

Exemple :

```
fonction Somme (tab : tableau d'entiers[N] ; N : entier) : entier
début
    somme ← 0                      // 1 instruction
    pour i allant de 0 à N-1 faire      // N fois …
         somme ← somme + tab[i] //     1 instruction
    fpour
    retourner somme              //     1 instruction
fin
```

Le nombre de données à traiter est N.
Le nombre d'instructions exécutées est environ de 1+N*1+1 = N+2 (alors que le programme contient environ 3 instructions).

On parle de complexité en temps parce que, si on considère que chaque instruction dure un temps constant (10^{-9} s par exemple), la complexité permet d'approcher la durée de l'exécution.

Concrètement, comment calculer le nombre d'instructions exécutées ? Il y a 2 difficultés :

- *estimer la taille* du problème : *la taille est le nombre de données à traiter. Parfois, la taille est définie par plusieurs quantités*.
  Exemples :
  - pour un algorithme de tri d'un tableau, la taille est la dimension N du tableau ;
  - pour l'algo de multiplication de deux entiers, la taille est le nombre N de bits dans la codification d'un entier ;
  - pour la recherche du plus court chemin entre deux villes d'un réseau routier, la taille est constituée du nombre total de villes N et du nombre total de routes M ;

- compter les instructions (dans le pire cas) et exprimer le résultat en fonction de la taille. Pour cela, nous choisirons la ***convention*** suivante :
  - pour une affectation : 1
  - pour un test : 1 + coût du plus gros des deux blocs (si et sinon)
  - pour une boucle : 1 (gestion de la boucle) + nombre de répétitions*coût du corps
    Parfois, le nombre d'itérations n'est pas connu et c'est plus compliqué.
  - pour un appel de sous-programme : 1 + coût du sous- programme

C'est assez approximatif mais on s'en contente. Souvent, on est obligé de surestimer.
Pour un nombre N de données, le nombre d'instructions exécutées n'est pas toujours le même. Par exemple, s'il faut trier N données avec un tri *par insertion*, le nombre d'instructions effectivement exécutées dépend grandement du fait que les données sont déjà triées par ordre croissant (cas le plus favorable) ou triées par ordre décroissant (cas le plus défavorable) : pour le même nombre N de données, le seul nombre de décalages varie de 0 à $\frac{1}{2}N^2$.
Concrètement, la complexité en temps est donc comprise entre deux bornes : le meilleur et le pire des cas. Dans ce cours, on ne considère que la *complexité dans le pire des cas*.

Exemple 1 : maximum d'un tableau de N valeurs
Taille du problème : nb de valeurs du tableau donc N

```
fonction maxTab(tab : tableau d'entiers[N]; N : entier) : entier
début
    max ← tab [0]                 // 1 instruction
    pour i de 1 à N-1 faire       // 1 instr. pour gérer le pour + N-1 fois :
         si (tab[i] > max) alors  // 1 test et
              max ←  tab[i]       //  1 instruction (au pire)
         fsi
    fpour
    retourner max                 // 1 instruction
fin
```

soit une complexité dans le pire des cas de C(N) = 2+(N-1)*(1+1)+1 = 2*N+1.

Exercice *:* Donner la complexité asymptotique de l'insertion d'une valeur V dans un tableau trié T de N valeurs. On suppose le tableau assez grand pour tolérer une valeur de plus.

Exemple 2 : tri par échange d'un tableau de N valeurs

Taille du problème : nb de valeurs du tableau donc N

```
procédure tri(↕tab : tableau d'entiers[N]; N : entier)
début
    // On échange chaque élément avec le plus petit de ceux qui
    // suivent (lui compris)
    pour i de 0 à N-2 faire
        // recherche du min entre i et N
        posMin ← i
        pour j de i+1 à N-1 faire
            si (tab[j] < tab[posMin]) alors
                posMin ← j
            fsi
        fpour
        echanger(tab, N, i, posMin)
    fpour
fin

fonction echanger(↕tab:tableau d'entiers[N]; N:entier; p1:entier; p2:entier)
début
    temp ← tab[p1]
    tab[p1] ← tab[p2]
    tab[p2] ← temp
fin
```

Le corps de la boucle principale coûte :

- une instruction (affectation), plus
- 1 instruction (gestion du "pour" interne) + (N-i-1)*2 instructions (test et affectation), plus
- 3 instructions (échange)

soit 2*N – 2*i + 3

Ceci est réalisé pour i valant 0, puis pour i valant 1, etc. jusqu'à i = N-2.

Le coût global est donc :

C (N) = 1 (gestion du "pour" externe) + Σ (2*N – 2*i + 3), i variant de 0 à N-2

= 1 + Σ 2*N – Σ 2*i + Σ 3

= 1 + 2*N*(N-1) – 2*(N-2)*(N-1)/2 + 3*(N-1)

= N^2 - 2*N + 2

Exemple 3 : recherche dichotomique de la position d'une valeur dans un tableau trié de N éléments

Taille du problème : nb de valeurs du tableau donc N

```
fonction chercher(tab:tableau d'entiers[N]; N:entire; val:entier) : entier
début
    // On conserve la première ou la deuxième moitié de l'intervalle
    // [deb,fin] selon que V est > ou < au milieu.
    // Si V est égale au milieu, on arrête.
    deb ← 1 ; fin ← N
    trouve ← faux
    pos ← –1
    tant que (deb <= fin) et non trouve faire // test possible sur pos
                                              // au lieu de trouve
        mil ← (deb+fin) / 2
        si (tab[mil] = val) alors
            trouve ← vrai
            pos ← mil
```

```
            sinon si (val < tab[mil]) alors
                fin ← mil - 1
            sinon
                deb ← mil + 1
            fsi fsi
        ftq
        retourne pos
    fin
```

Le pire des cas survient si `val` est absente. Le corps de la boucle coûte donc 4 (calcul de mil, 2 tests et modif de deb ou de fin).
Combien de fois la boucle est-elle répétée? Autant de fois que l'intervalle [1,N] peut être divisé en 2, autrement dit $\log_2 N$ (en effet, 1*2*2* … *2 ($\log_2 N$ fois) font $2^{\log_2(N)}$ qui fait N car $b^{\log_b(N)} = N$).
Le coût de la recherche dichotomique est donc (dans le pire des cas) de : 4* $\log_2 N$ (+4).

Notez bien **l'extraordinaire** performance de la dichotomie. Si on a un million de valeurs triées (N=1000000), 80 instructions suffisent à vérifier la présence de v dans cet ensemble : 80 pour un million ! En effet, en vingt itérations, on passe d'un intervalle de 1 000 000 à 500 000 à 250 000 à 125 000 à 64 000 à 32 000 à 16 000 à 8 000 à 4 000 à 2 000 à 1 000 à 500 à 250 à 125 à 64 à 32 à 16 à 8 à 4 à 2 à 1. Autrement dit, en 20 itérations (dans le pire des cas), on arrive à un intervalle dans lequel la réponse est immédiate. Par comparaison, la version qui parcourt le tableau d'un bout à l'autre risque de tester 1 000 000 valeurs (même si le tableau est trié) et peut donc être jusqu'à 25 000 fois plus lente !

2.1.2 Complexité asymptotique

Quand on évalue la complexité, on ne s'intéresse en général qu'à son ordre de grandeur. Par exemple, si la complexité C (N) = $223*N^3 + 25*N^2 + \log_2(N)$, on ne conserve que le terme de plus haut degré, on néglige la constante multiplicative et on note $C(N) = O(N^3)$. C'est ce que l'on appelle la complexité asymptotique, ou complexité pour les grandes valeurs de N (quand N tend vers l'infini).

Pour être exact :
Dire que la complexité $C(N) = 223*N^3 + 25*N^2 + \log_2(N)$ est en $O(N^3)$ signifie qu'il existe une constante multiplicative K telle que C(N) est dominée par $K*N^3$ à partir d'un certain stade.

Plus formellement, $C(N) = O(f(N)) \Leftrightarrow \exists\ N_0 > 0$ et $K > 0$ tels que $\forall\ N > N_0,\ \ C(N) \leq K * f(N)$

C(N) = O(f(N)) est un abus de notation. On devrait noter : $C(N) \in O(f(N))$ car O(f(N)) est l'ensemble des fonctions dont l'ordre de grandeur est inférieur ou égal à celui de f (N)

Deux autres symboles sont utilisés : Ω et θ.

- $f(N) = O(g(N)) \Leftrightarrow$ f est d'un ordre de grandeur inférieur ou égal à g
- $f(N) = \Omega(g(N)) \Leftrightarrow$ f est d'un ordre de grandeur supérieur ou égal à g
- $f(N) = \theta(g(N)) \Leftrightarrow$ f est du même ordre de grandeur que g.

Exemples : est-ce que f =O(g) ?

- f(N) = 2N, g(N) = 4N : ???
- f(N) = 4N, g(N) = 2N : ???
- $f(N) = N^3$, $g(N) = (2^{\log_2(N)})^2$: ???

Exemple : si f(N) > g(N) pour tout N > 0, est-ce que f = O (g) ? …… car ???

Les ordres de grandeur classiques :

- O (1) : algo en temps constant (accès au $i^{ème}$ élément d'un tableau)
- O ($\log_2$ (N)) : algo logarithmique (recherche dichotomique dans un tableau de N nombres)
- O (N) : linéaire (la somme des éléments d'un tableau de N nombres)
- O (N.log (N)) : henneloguedehenne (un bon algo de tri de N nombres)
- O (N^2) : quadratique (un mauvais algo de tri de N nombres)
- O (N^3) : cubique (le produit de deux matrices de rang N)
- O (N^k) : polynomial
- O (e^N) : exponentiel (plus courte tournée par toutes les villes d'un réseau=pb du voyageur de commerce)

ATTENTION : l'évolution de la complexité entre deux ordres de grandeur consécutifs n'est pas **DU TOUT** linéaire !!! Voici la durée d'exécution présumée d'un programme en fonction de sa complexité sur une machine extrêmement rapide (hypothèse optimiste : 1 instruction = 10^{-10} s et constante multiplicative = 1) :

C(N) \ N	100	1000	10000	100000	10^6
$\log_2$	ε	ε	ε	ε	$2\ 10^{-9}$ s
N	10^{-8} s	10^{-7} s	10^{-6} s	10^{-5} s	10^{-4} s
N.$\log_2$ (N)	$6\ 10^{-8}$ s	10^{-6} s	10^{-5} s	10^{-4} s	10^{-3} s
N^2	10^{-6} s	10^{-4} s	10^{-2} s	1 s	100 s
N^3	10^{-4} s	10^{-1}	100 s	30 h	3 a
e^N	10^{25} a	∞	∞	∞	∞

→ on ne peut espérer résoudre le problème du voyageur de commerce dans un réseau routier de 100 villes dans le temps d'une vie humaine (sauf cas particuliers ! Heureusement, il y a des *heuristiques*).

ATTENTION : il est maladroit de comparer les performances de deux algorithmes en ne se fondant *que* sur la complexité asymptotique mais c'est tout de même un bon indicateur.

Exemple :

l'algo 1 a une complexité de $N\log_2(N)$ donc est en $O(N\log_2(N))$,
l'algo 2 a une complexité de 10N donc est en O(N),
a priori, le deuxième est donc meilleur.
Pourtant, il faut attendre $N = 2^{10} = 1024$ pour que $N\log_2(N) > 10N$!!!

Synthèse des exercices

Coût d'une boucle :

- 1 + Nb itérations * Coût d'une itération
  ou
- 1 + cout itération 1 + coût itération 2 + … + coût itération N
  = 1 + ∑ Coûts itérations

2.2 Complexité en mémoire

Elle consiste à exprimer la quantité de mémoire utilisée par un algorithme en fonction de la taille du problème. Cette notion est hors-programme.

3 La programmation récursive

3.1 Définition d'un sous-programme récursif

Définition : un programme récursif est un programme qui s'appelle (directement ou indirectement) lui-même. Des mécanismes doivent être mis en place pour éviter de boucler à l'infini.

Soit la procédure P :

```
procédure P(↓i : entier)
début
      afficher(i)
      P(i+1)                                  ← ceci est un appel « récursif »
fin
```

L'appel
```
  P(1)
```
provoquerait l'affichage de : ???

Soit la procédure P :

```
procédure P (↓i : entier)
début
      afficher(i)
      si (i < 3) alors
            P (i+1)                           ← appel récursif
      fsi
fin
```

L'appel
```
   P(1)
```
provoquerait l'affichage de : ???

Soit la procédure P :

```
procédure P (↓i : entier)
début
      afficher(i)
      i ← i + 2
      si (i < 4) alors
          P(i+1)                              ← appel récursif
      sinon
          afficher("Hello")
      fsi
      afficher (2*i-1)
fin
```

L'appel
```
   P(1)
```
provoquerait l'affichage de : ???

3.2 Résolution récursive de problèmes

Définition : résoudre de manière récursive un problème portant sur un certain nombre de données consiste à appuyer sa résolution sur la résolution de problèmes identiques mais concernant un nombre plus petit de données.

Quand on met au point la formulation récursive, on NE se préoccupe PAS de la manière de résoudre le ou les sous-problème(s) : il *faut supposer que c'est fait* et *seulement* chercher comment cela permet de résoudre le problème initial.

ATTENTION : il y a toujours des cas particuliers où la méthode générale ne peut pas être appliquée. On les appelle « cas d'arrêt » ou « cas simples » ou « cas triviaux ». Pour ces cas-là, il faut une méthode de résolution différente (elle est toujours très simple).
ATTENTION : il faut être certain que les diminutions successives du nombre de données mènent vers un des cas triviaux (règle de réduction).
ATTENTION : certains problèmes ne se prêtent pas à une résolution récursive.

Exemples informels :

1. On veut une procédure qui affiche les entiers de Deb à Fin (Deb ≤ Fin)
   Afficher les entiers de Deb **à** Fin, c'est
   Afficher les entiers de Deb *à* Fin-1 (← récursivité)
   Ecrire (Fin)

   Mais si Deb = Fin, on exprime l'affichage de Deb à Deb (donc l'affichage de 1 valeur) à l'aide de l'affichage de Deb à Deb-1 ; ceci est faux à double titre : ce n'est pas un intervalle valide ET, à supposer que l'on considère cet intervalle comme valide, il contient 2 valeurs, ce qui viole la règle de réduction. Autrement dit, la méthode ne s'applique que si Deb > Fin, pour les autres cas, il faut une méthode différente !!

   Donc la bonne formulation est :
   Afficher les entiers de Deb **à** Fin, c'est
   SI (Deb = Fin) ALORS
   Ecrire (Fin) (← cas trivial)
   SINON
   Afficher les entiers de Deb *à* Fin-1 (← récursivité)
   Ecrire (Fin)
   FSI

2. **Afficher les entiers de** Deb **à** Fin (Deb ≤ Fin), c'est aussi
   SI (Deb = Fin) ALORS
   Ecrire (Deb) (← cas trivial)
   SINON
   Ecrire (Deb)
   Afficher les entiers de Deb+1 *à* Fin (← récursivité)
   FSI

   Remarquez que l'ordre des instructions est important ! Que se passe-t-il si on échange les deux lignes dans le sinon ?
   Remarquer aussi que la réduction d'échelle, dans ces deux premiers cas, consiste à diminuer le nb de données de un à chaque étape.

3. **Afficher les entiers de** Deb **à** Fin (Deb ≤ Fin), c'est aussi

```
SI (Deb = Fin) ALORS
        Ecrire (Deb)                                (← cas trivial)
SINON
        Milieu ← (Deb+Fin) div 2
        Afficher les entiers de Deb à Milieu        (← récursivité)
        Afficher les entiers de 1+ Milieu à Fin     (← récursivité)
FSI
```

   Remarquez que le cas général ne contient cette fois aucun affichage et que la méthode fonctionne pour des intervalles de 3 ou 2 valeurs (il y a bien réduction dans ces deux cas) et 1 valeur (cas trivial).

4. Cette fois, on souhaite gèrer aussi les intervalles dont la borne de début est supérieure à la borne de fin (dont on affiche les valeurs par ordre décroissant).

   Afficher les entiers de Deb **à** Fin, c'est aussi

```
SI (Deb = Fin) ALORS
        Ecrire (Deb)                                (← cas trivial)
SINON SI (Deb < Fin) ALORS
        Ecrire (Deb)
        Afficher les entiers de Deb+1 à Fin-1       (← récursivité)
        Ecrire (Fin)
SINON       // c'est donc que Deb > Fin
        Ecrire (Deb)
        Afficher les entiers de Deb-1 à Fin+1       (← récursivité)
        Ecrire (Fin)
FSI FSI
```

   Où est l'erreur ? (essayer avec Deb = 2, Fin = 4 puis avec Deb = 2 et Fin = 3)

   On n'a pas garanti d'arriver dans un cas trivial → il faut soit changer la méthode générale, soit augmenter le nombre de cas triviaux → y réfléchir.

5. On veut une *fonction* calculant le $n^{ème}$ terme de la suite $u_n = 2*u_{n-1} - 3$ ($u_0 = 1$, $n \geq 0$)

   Le $n^{è}$ terme de la suite $u_n = 2*u_{n-1} - 3$ ($u_0 = 1$) est

```
SI n = 0 ALORS
        Le résultat est 1
SINON
        u ← le (n-1)è terme de la suite             (← récursivité)
        Le résultat est 2*u - 3
FSI
```

   Remarquer que la formulation récursive du problème diffère de celle utilisée pour les 4 premiers exemples, qui étaient des procédures :

   - on utilise "le résultat est" dans chaque cas
   - la formulation du problème ne commence pas par un verbe à l'infinitif mais par « le (résultat souhaité) est ».

6. On veut une fonction calculant le plus grand des N premiers éléments d'un tableau T ($N \geq 0$)

   Le max des N **premiers éléments du tableau** T est :
   SI (N=1) ALORS
   Le résultat est T [0]
   SINON
   MaxTemp ← *Le max des* N-1 *premiers éléments du tableau* T

   // le résultat est le plus grand entre T [N] et MaxTemp
   SI T [N] < MaxTemp ALORS
   Le résultat est MaxTemp
   SINON
   Le résultat est T [N]
   FSI
   FSI

   Cette version réduit l'intervalle d'étude d'une case à chaque appel récursif.
   Faut-il craindre que la récursivité ne s'arrête jamais ?

7. Idem mais en réduisant le nombre de valeurs de 2

   Le max des N **premiers éléments du tableau** T est
   SI (N=1) ALORS
   le résultat est T [0]
   SINON
   MaxTemp ← *Max des* N-2 *premiers éléments du tableau* T

   le résultat est le plus grand entre T [N], T [N-1] et MaxTemp
   FSI

   Cette version réduit l'intervalle d'étude de *deux* cases à chaque appel récursif mais elle est fausse …
   Pourquoi ?
   Comment la corriger ?

3.3 Rédiger un programme récursif

Pour écrire un programme récursif, il faut :
- définir son rôle
- établir le paramétrage (→ procédure ou fonction)
- établir la (les) méthode(s) de résolution du cas général (exigeant une récursivité)
- chercher quand les méthodes générales ne s'appliquent pas et en déduire les cas triviaux
- traduire la méthode en un sous-programme récursif. Cette étape est simple, la structure du sous-programme étant (et devant être) absolument identique à celle de la méthode.

Reprise de l'exemple 1 **(c'est une procédure)** :

Afficher les entiers de Deb **à** Fin, c'est
SI (Deb = Fin) ALORS
Ecrire (Fin) (← cas trivial)
SINON
Afficher les entiers de Deb *à* Fin-1 (← récursivité)
Ecrire (Fin)
FSI

devient

```
procedure afficherEntiers( ↓deb : entier ; ↓fin : entier)
début
      si(deb = fin) alors
            afficher(fin)
      sinon
            afficherEntiers(deb,fin-1)    // notez l'usage de la
                                          // procédure récursive
            afficher(fin)
      fsi
fin
```

Reprise de l'exemple 6 **(c'est une fonction)** :
Le max des N **premiers éléments du tableau** T est :
SI (N=1) ALORS
le résultat est T [0]
SINON
MaxTemp ← *Le max des* N-1 *premiers éléments du tableau* T
// le résultat est le plus grand entre T [N] et MaxTemp
Si T [N] < MaxTemp ALORS
Le résultat est MaxTemp
Sinon
Le résultat est T [N]
FSi
FSI

Le programme correspondant (on suppose que les éléments sont des réels):

```
// Fonction qui prend un tableau de M éléments et restitue le plus grand
// élément parmi ses premiers N éléments (N < M)
fonction MaxTab (↓tab : tableau de réels[M] ; ↓N : entier) : réel
début
      si (N = 1) alors
            maxTab ← tab[0]
      sinon
            maxTemp ← maxTab(tab,N-1)     // notez l'usage de la
                                          // fonction récursive

            si tab[N-1] > maxTemp alors
                  maxTab ← tab[N-1]
            sinon
                  maxTab ← maxTemp
            fsi
      fsi
      retourne maxTab
fin
```

Synthèse des exercices :

- parfois plus d'un appel récursif dont un seul est exécuté (cf. la dichotomie), parfois plusieurs appels qui sont tous exécutés (cf. certains tris), parfois plusieurs appels et on ne peut prédire lesquels seront exécutés (cf. chemin dans un damier) ;
- dans le cas général (par opposition au cas d'arrêt), il y a des instructions supplémentaires en plus de l'appel récursif et il faut être attentif à la manière de les placer par rapport à (aux) appel(s) récursif(s) ;
- il faut parfois ajouter aux sous-programmes des paramètres autres que les paramètres « naturels » (voir (3.4)).

Avantages / inconvénients de la programmation récursive :

- ☺ Récursivité → résolution simplifiée des problèmes naturellement récursifs (c'est le cas de nombreux algorithmes sur les arbres et les graphes).
- ☺ Concision
- ☺ Lecture *donc* compréhension *donc* mise au point facilitées
- ☹ Plus lente (parfois beaucoup plus) que la version itérative à cause : 1) des appels récursifs 2) du fait que l'on résout parfois plusieurs fois un même sous-problème (cf le nombre de combinaisons). Cet inconvénient est très relatif car le temps perdu en exécution par comparaison avec la version itérative est le plus souvent **très** inférieur au temps passé à élaborer la version itérative, qui est bien moins intuitive
- ☹ Souvent plus gourmande en mémoire (jusqu'à la fin de l'appel récursif initial)

Pour information

- En principe, dans la version récursive, il y a une boucle de moins que dans la version itérative. *EN L2, dans les exercices que vous ferez seuls, il n'arrivera jamais qu'un appel récursif se trouve dans une boucle. Par contre il pourra arriver qu'un appel récursif soit précédé ou suivi d'une boucle* ;

- Tout programme récursif peut être transformé en un programme itératif. On parle de dérécursion.

- La récursion est un moyen (parmi d'autres) d'appliquer une approche classique de résolution de problèmes : diviser pour régner : on divise le problème principal en plusieurs sous-problèmes (diviser), on résout ces problèmes par des appels récursifs (à la condition qu'ils soient de même nature que le problème principal) et on combine les solutions des sous-problèmes pour obtenir celle du problème principal. D'autres méthodes de résolution de problèmes se nomment "programmation dynamique", "backtracking", "approche gloutonne", "heuristiques" → L3, M1

- Les risques de gâcher de la mémoire sont importants → de la rigueur dans les désallocations !!!

Exercices (travail personnel):
Donner une méthode récursive de résolution des problèmes suivants et écrivez les sous-programmes récursifs qui implémentent ces solutions (ne pas gâcher de mémoire):

a. Calculer le produit des entiers de a à b (soit N ! si a = 1 ou 2 et b = N) pour a ″ b
b. Calculer le nombre de combinaisons de p parmi n (soit C (n,p)) pour 0 ″ p ″ n
c. Vérifier la présence d'une valeur dans un tableau de N éléments
d. Inverser sur place de l'ordre des éléments d'un tableau
e. Vérifier la présence d'une valeur dans un tableau *trié*
f. Trier un tableau

3.4 Si on ne trouve pas la formulation récursive …

Parfois, après avoir défini le rôle du sous-programme et trouvé ses paramètres, on ne trouve pas de méthode récursive de résolution. On peut alors essayer d'ajouter des paramètres a priori inutiles.

Il est difficile d'avoir l'intuition de ces paramètres supplémentaires et dans ce module, ils sont souvent suggérés dans l'énoncé.

Exemple : transformation d'une expression arithmétique parenthésée infixée en une expression préfixée. Les deux sont dans des chaînes de caractères.

Par exemple,
la chaîne "x" deviendrait "x".
la chaîne "57" deviendrait "57".
la chaîne "(x*3)" deviendrait "*(x,3)".
la chaîne "(57+(x*3))" deviendrait "+(57,*(x,3))".

Tel qu'exprimé, le problème a 2 paramètres : la chaîne infixée (en entrée) et la chaîne préfixée (en sortie). Avec ces 2 seuls paramètres, il est difficile de trouver une formulation récursive.
Par contre, en introduisant 2 paramètres supplémentaires, les choses deviennent plus faciles : il y a un paramètre d'entrée, qui est la position à laquelle commence l'expression à traduire en préfixé ; et il y a un paramètre de sortie, qui indique où s'est arrêtée la traduction, c'est-à-dire la position où se termine l'expression. Il y a donc maintenant 4 paramètres, 2 données et deux résultats.
Par exemple, si les données sont "(57+(x*3))" et :
- 6 (la position de 'x'), alors les résultats sont la chaîne "x" et l'entier 6 ;
- 2 (le début de 57), alors les résultats sont la chaîne "57" et l'entier 3 ;
- 5 (la position de la deuxième parenthèse ouvrante), alors les résultats sont la chaîne "*(x,3)" et l'entier 9 ;
- 1, alors les résultats sont la chaîne "+(57,*(x,3))" et l'entier 10.

```
procedure InfixeEnPrefixe(     ↓Inf : TChaine ; ↓Deb : entier,
                               ↑Pref : TChaine ; ↑Fin : entier)
// On transforme en expression préfixée la partie de Inf qui commence en Deb.
// On restitue l'expression préfixée dans Pref et la position à laquelle
// s'est arrêté le parcours de Inf dans Fin
début
      // la position Deb correspond à une '('. On a donc une expression
      // du type (…+…) ou (…-…) ou …
      si (Inf [Deb] = '(') alors
            // On transforme récursivement la partie qui commence en Deb+1,
            // on obtiendra une version préfixée de l'opérande gauche de la
            // parenthèse et la position où il s'arête.
            InfixeEnPrefixe(Inf,Deb+1, OperG, FinOG)

            // FinOG est la fin de l'opérande gauche ; juste après, on trouve
            // donc l'opérateur et, en FinOG+2, l'opérande de droite.
            // On transforme donc récursivement la partie qui commence en FinOG+2
            // et on obtiendra une version préfixée de l'opérande droit de la
            // parenthèse et la position où il s'arête.
            InfixeEnPrefixe(Inf,FinOG+2, OperD, FinOD)

            // On assemble, dans cet ordre, l'opérateur, une (, l'opérande gauche en
            // préfixé, une virgule, l'opérande droit en préfixé et une ).
            Pref ← Inf[FinOG+1] + "(" + OperG + "," + OperD + ")" ;

            // L'expression qu'on vient de traduire se termine juste après son opérande droit.
            Fin ← FinOD + 1

      // cas de x
```

```
        sinon si (Inf [Deb] = 'x') alors
              Fin ← Deb ;
              Pref ← "x" ;

        // cas d'une constante numérique
        sinon
              // recherche de la fin de la constante numérique
              FinConstante ← Deb+1
              TrouveFinCste ← false ;
              tant que (FinConstante <= Longueur (Inf)) et
                        non TrouveFinCste
              faire
                    si (Inf [i] dans ['0' ..'9']) alors
                          FinConstante ++
                    sinon
                          TrouveFinCste ← true
                    fsi
              ftq

              Fin ← FinConstante -1

              // Le résultat est la partie de Deb à Fin
              Pref ← Portion de Inf de Deb à Fin
        fsi fsi fsi
fin
```

3.5 Complexité d'un sous-programme récursif

On peut souvent exprimer la complexité en temps d'un sous-programme récursif (taille N) à l'aide d'une formule récursive, par exemple :

$C(N) = a \times C(N - b) + c$ (cf l'exercice « présence d'une valeur dans une liste »)
ou
$C(N) = a \times C(N / b) + c$ (cf l'exercice « présence dans un tableau trié avec la dichotomie »)
ou, plus généralement,
$C(N) = a \times C(N / b) + f(N)$

Il faut alors « résoudre » cette récurrence. On le fait au cas par cas en L2, à l'aide du théorème maître en L3 (si on est dans le dernier cas).

4 Les listes, les piles et les files

4.1 Listes (lists)

Une liste est une séquence d'éléments d'un certain type E. Ci-dessous on rappelle une définition possible de l'ADT Liste. D'autres opérations peuvent être définies (taille de la liste, appartenance d'un élément, élément à une position donnée, etc.) – les axiomes correspondants utiliseront les opérations de base sur les listes et éventuellement sur d'autres types (par exemple, les entier pour la taille).

Type : `Liste[E]`
Opérations :

```
ConsVide : → Liste[E]
Cons : E x Liste[E] → Liste[E]
Tete: Liste[E] → E
Reste: Liste[E] → Liste[E]
EstVide: Liste[E] → booléen
```

Préconditions :

`Tete(l)` défini ssi non `EstVide(l)`

`Reste(l)` défini ssi non `EstVide(l)`

Axiomes :

```
Tete(Cons(e,l)) = e
Reste(Cons(e,l)) = l
EstVide(ListeVide) = vrai
EstVide(Cons(e,l)) = faux
```

Plusieurs structures de données implémentent l'ADT Liste. Ces implémentations doivent être fidèles à la spécification et efficace en terme de complexité d'algorithme. Nous étudions dans la suite essentiellement les structures chainées.

4.1.1 Listes chaînées

Rappel de la structure de données classique : un pointeur sur un maillon à deux membres dont le deuxième est un pointeur sur … Il faut donc bien garder à l'esprit que la liste ne désigne pas l'ensemble des valeurs mais seulement le pointeur de départ !!!
Rappel des primitives des listes chaînées + complexité de ces primitives :

CONSULTATION

```
fonction     EstVide (↓L : Liste[E]) : booléen
// vrai ssi la liste est vide
```
Complexité : O (1)

```
fonction     Tete (↓L : Liste[E]) : E
// Restitue la première valeur → la liste ne doit pas être vide
```
Complexité : O (1)

```
fonction     Reste (↓L : Liste[E]) : Liste[E]
// Restitue la liste privée de sa prmière valeur → la liste ne doit pas être vide
```
Complexité : O (1)

CONSTRUCTION

```
fonction     ConsVide () : Liste[E]
// Restitue une liste vide
```
Complexité : O (1)

```
fonction     Cons (↓X : E, ↓L : Liste[E]) : Liste[E]
// Restitue la liste obtenue en ajoutant X en tête de L. L est utilisée telle qu'elle, sans
// duplication.
```
Complexité : O (1)

MODIFICATION

```
procedure    ModifTete (↕L : Liste[E], ↓X : E)
// Remplace la première valeur de L par X → L ne doit pas être vide
```
Complexité : O (1)

```
procedure    ModifReste (↕L : Liste[E], ↓NouvReste : Liste[E])
// Remplace le reste par NouvReste → la liste ne doit pas être vide
// L'ancien reste n'est pas libéré, le nouveau reste n'est pas dupliqué
```
Complexité : O (1)

DESTRUCTION

```
procedure    Libérer (↓ L : Liste[E])
```
Complexité : O (N) où N = nb de valeurs de la liste

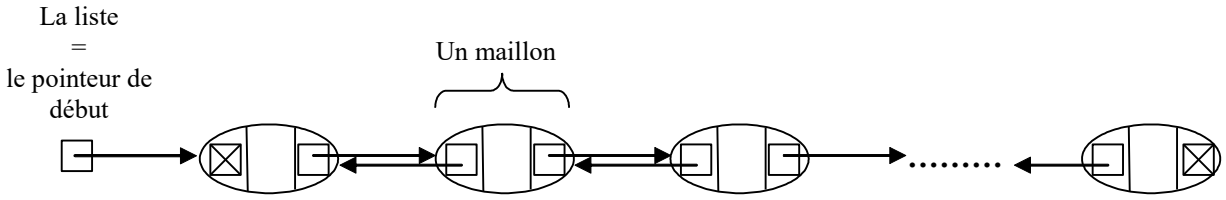
Elles ne pénalisent pas l'ordre de grandeur de la complexité des algos qui les utilisent sauf `Libérer`.

Exercices (travail personnel):
Donner une méthode récursive de résolution des problèmes suivants et écrivez les sous-programmes récursifs qui implémentent ces solutions (ne pas gâcher de mémoire):

g. Supprimer les K premiers éléments d'une liste (pas de création d'une nouvelle liste).
h. Supprimer un maillon sur deux dans une liste en commençant par le deuxième.
i. Ajouter les maillons d'une liste à l'arrière d'une autre (concaténation).
j. Faire la fusion de deux listes triées dans une troisième liste dont les maillons sont des copies de ceux des deux listes initiales.
k. Suppression des N dernières valeurs d'une liste. S'il y a moins de N valeurs, les supprimer toutes.

Des variantes des listes chaînées sont :

- les listes doublement chaînées : chaque maillon dispose d'un pointeur supplémentaire, initialisé à NULL pour le maillon de tête et pointant vers le maillon précédent pour les autres.



Les primitives portent les mêmes noms que celles des listes.
Elles gèrent le double chaînage.
Il y a également deux primitives supplémentaires : Pred (symétrique de Reste) et ModifPred (cf ModifReste).

- les listes circulaires : le suivant du dernier maillon est le premier.
Les primitives sont celles des listes chaînées.

Exercices :

- Ecrire une procédure récursive d'insertion d'une valeur à la position i d'une liste doublement chaînée. Si i <= 1, il s'agit d'une insertion en tête, si i > N, d'une insertion en queue. Le résultat de la fonction doit être la liste modifiée.
- Ecrire en pseudo-code une version itérative du calcul du nombre d'éléments d'une liste circulaire doublement chaînée.

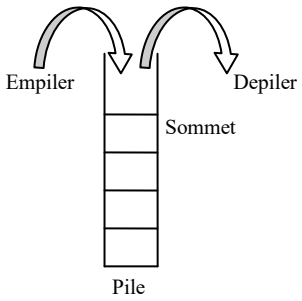
Ecrire la version C de la primitive Cons pour les listes doublement chaînées et circulaires.

4.2 Piles (stacks)

Une pile est une liste particulière, dans laquelle les seules opérations sont :

- la création d'une pile vide
- l'ajout en tête (on dit "empiler" ou « push »)
- la consultation de la valeur de tête (on dit le "sommet" ou « top »)
- la suppression de la valeur de tête (on dit "dépiler" ou « pop »)
- le test de vacuité.

Schématiquement, on la représente comme suit :



Comme la dernière valeur ajoutée est la seule que l'on puisse retirer, les piles sont aussi appelées des LIFO (Last In First Out).

Type : `Stack[E]`
Opérations :
```
ConsVide : → Stack[E]
Empiler: E x Stack[E] → Stack[E]
Sommet: Stack[E] → E
Depiler: Stack[E] → Stack[E]
EstVide: Stack[E] → booléen
```
Préconditions :
`Sommet(l)` défini ssi `non EstVide(l)`
`Depiler(l)` défini ssi `non EstVide(l)`
Axiomes :
```
Sommet(Empiler(e,l)) = e
Depiler(Empiler(e,l)) = l
EstVide(ListeVide) = vrai
EstVide(Empiler(e,l)) = faux
```

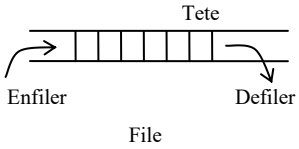
Elles servent entre autres :
- dans les dérécursions (transfo d'un algo récursif en un algo itératif),
- au cours de l'exécution d'un programme, pour conserver les paramètres, variables locales et adresse de retour d'un sous-programme qui est mis en veille par le déclenchement d'un autre sous-programme,
- dans certains algorithmes (comme le calcul de l'enveloppe convexe d'un nuage de points).

Exercice :
On dispose d'un type `Liste` pour les listes (chaînées) et des primitives associées. Réaliser une implantation du type `Stack` en utilisant celle des listes (chaînées) et en supposant que les valeurs conservées dans les piles et les listes chaînées sont d'un même type. Prononcez vous sur l'efficacité de votre réalisation des primitives des piles.

4.3 Files (queues)

Une file est une liste particulière, dans laquelle les seules opérations sont :
- la création d'une file vide
- l'ajout en queue (on dit "enfiler" ou « enqueue »)
- la consultation de la valeur de tête (on dit "enfiler" ou « front »)
- la suppression de la valeur de tête (on dit "défiler" ou « dequeue »)
- le test de vacuité.

Schématiquement, on la représente comme suit :



File

Comme la première valeur ajoutée est la première que l'on puisse retirer, les files sont aussi appelées des FIFO (First In First Out).

Type : `Queue[E]`
Opérations :
```
ConsVide : → Queue [E]
Enfiler: E x Queue [E] → Queue [E]
Tete: Queue [E] → E
Defiler: Queue [E] → Queue [E]
EstVide: Queue [E] → booléen
```
Préconditions :
`Tete(l)` défini ssi `non EstVide(l)`
`Defiler(l)` défini ssi `non EstVide(l)`
Axiomes :
```
EstVide(l) => Tete(Enfiler(e,l)) = e
non EstVide(l) => Tete(Enfiler(e,l)) = Tete(l)
EstVide(l) => Defiler(Enfiler(e,l)) = ConsVide()
non EstVide(l) => Defiler(Enfiler(e,l)) = Enfiler(e,Defiler(l))
EstVide(ListeVide) = vrai
EstVide(Enfiler(e,l)) = faux
```

4.4 Files de priorité (priority queues)

On utilise aussi souvent **la file de priorité**. Il s'agit d'un ensemble de valeurs dans lequel les éléments sont munis d'une information qui définit leur *priorité*. En général, on considère que l'élément le plus prioritaire est celui dont la priorité est la plus petite. Les files de priorités offrent les services (primitives) suivants :

- la création d'une file de priorité vide
- l'insertion d'une valeur
- la consultation de la valeur la plus prioritaire
- la suppression de la valeur la plus prioritaire
- le changement de la priorité d'une valeur
- le test de vacuité.

Les files de priorité servent par exemple :

- dans le tri par tas (cf 10.1.4),
- dans l'algorithme de Dijkstra, pour le calcul des plus courts chemins d'un sommet vers tous les autres sommets d'un graphe.

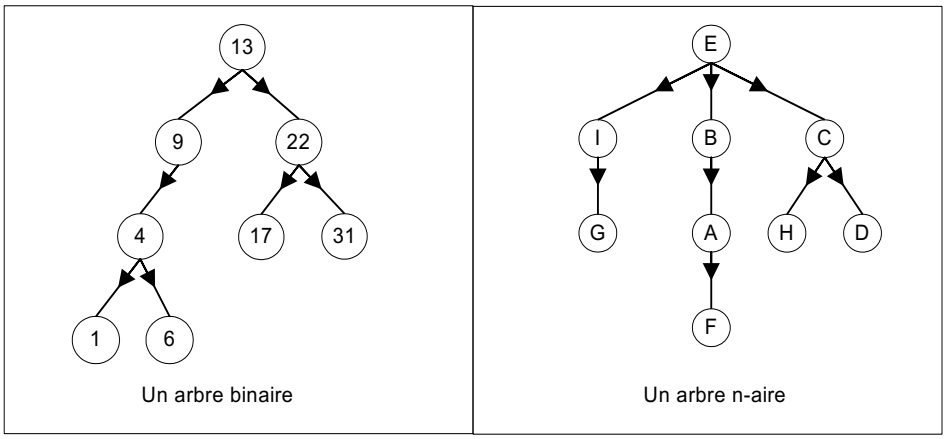
Exercice :
On dispose d'un type `Liste` pour les listes chaînées et des primitives associées. Réaliser une implantation du type `FilePrio` (il s'agit bien des *files de priorité*, et non *pas* des files simples) en utilisant celle des listes chaînées et en supposant que les valeurs conservées dans les files de priorité et les listes chaînées sont d'un même type. On considère qu'une valeur v_1 est plus prioritaire qu'une valeur v_2 si v_1.prio < v_2.prio.
Prononcez vous sur l'efficacité de votre réalisation des primitives des files de priorité.

5 Les arbres

5.1 Introduction

Les arbres sont une généralisation des listes : chaque maillon peut avoir plusieurs suivants et on ne parle plus de suivants mais de fils. De plus, on ne parle plus de maillons mais de nœuds et on dessine la racine (nœud de départ) en haut. La structure de données la plus immédiate est une extension de celle des listes chaînées : un arbre est un **pointeur** sur un nœud qui contient une ou plusieurs informations, des pointeurs vers les nœuds fils et, éventuellement, un pointeur vers le nœud père.

Exemples :



Comme on l'a fait pour les listes, à chaque fois que l'on s'intéressera à l'implantation des algorithmes, on considérera qu'un arbre est un pointeur sur le nœud racine (et non l'ensemble des nœuds) et que ce pointeur est NULL quand l'arbre est vide.

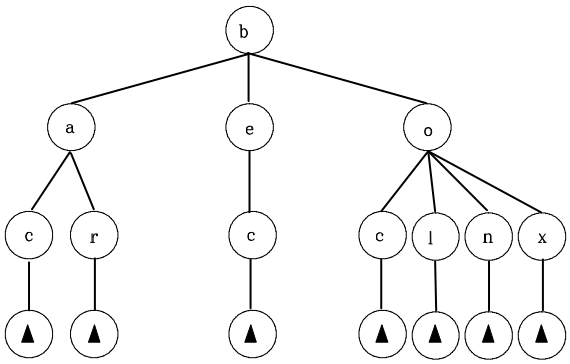
Ce qui suit ne concerne que les arbres *binaires*.

Vocabulaire : racine, fils gauche, fils droit, feuille, nœud interne, hauteur d'un arbre A (nombre de nœuds sur le chemin de la racine de A à la feuille la plus éloignée).

Motivations de l'étude des arbres

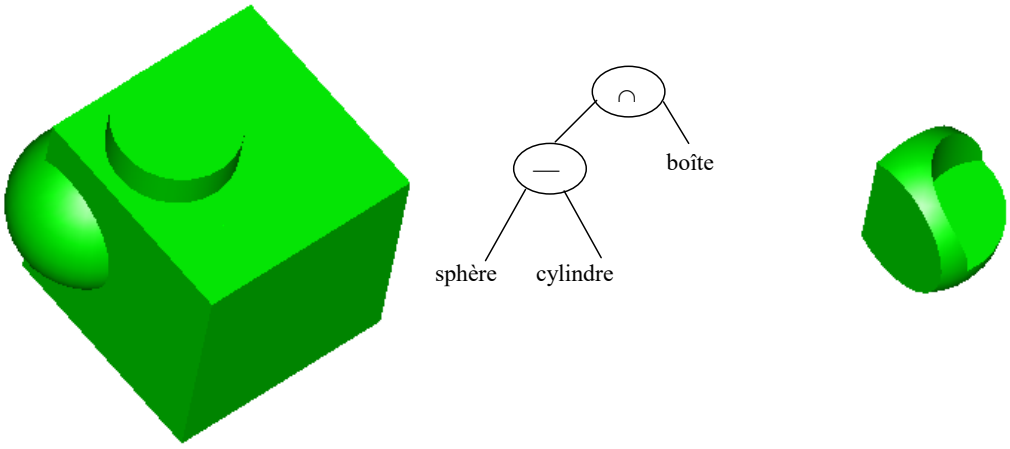
- un dictionnaire est un ensemble de valeurs munies chacune d'une clé unique. Dans cet ensemble, les trois opérations de base sont : l'ajout d'une valeur, le test de présence d'une valeur de clé donnée et le retrait d'une valeur. Certains arbres permettent une réalisation *extrêmement* efficace des dictionnaires (les 3 opérations de base ont des performances logarithmiques) avec une occupation mémoire très raisonnable → ce sont des outils incontournables pour la réalisation des ensembles et des BD ;

- ce sont des structures de données naturelles dans un certain nombre de situations :
    - mémorisation d'un dictionnaire lexicographique par un arbre n-aire (partage des préfixes communs → économie de mémoire, par comparaison avec une liste de mots – y réfléchir à la maison –)



- modélisation d'arbres (au sens végétal) par un arbre n-aire ou un arbre binaire équivalent

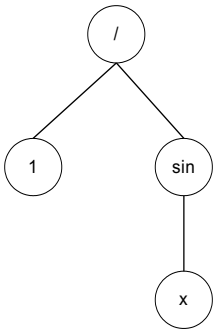
- Arbres CSG : représentation de la forme d'un objet



3 solides — Un arbre CSG — Le solide décrit par l'arbre CSG

- Représentation d'expressions arithmétiques (évaluation, dérivation) par des arbres syntaxiques :

1/sin(x) ➔



- Programmation des jeux de stratégies (et techniques d'élaguage)
- …

5.2 Type abstrait de données

Le TAD ArbreBinaire est similaire aux listes.

Type : `ArbreBinaire[E]`
Opérations :

```
ConsVide : → ArbreBinaire[E]
Cons : E x ArbreBinaire[E] → ArbreBinaire[E]
Racine : ArbreBinaire[E] → E
FG : ArbreBinaire[E] → ArbreBinaire[E]
```

```
    FD : ArbreBinaire[E] → ArbreBinaire[E]
    EstVide: ArbreBinaire[E] → booléen
```
Préconditions :
```
    Racine(l) défini ssi non EstVide(l)
    FG(l) défini ssi non EstVide(l)
    FD(l) défini ssi non EstVide(l)
```
Axiomes :
```
    Racine(Cons(e,fg,fd)) = e
    FG(Cons(e, fg,fd)) = fg
    FG(Cons(e,fg,fd)) = fd
    EstVide(ConsVide()) = vrai
    EstVide(Cons(e,fg,fd)) = faux
```

Comme pour les listes, plusieurs implémentations sont possible et on se focalisera ici essentiellement sur des structures de données basées sur des pointeurs.

5.3 Structure de données et primitives

La structure de données adoptée dans ce cours pour le type ArbreBinaire est un pointeur sur un nœud. Le nœud contient une information, qu'on supposera pour l'instant être un type nommé E, et des pointeurs vers les deux fils, donc deux arbres.

Primitives :

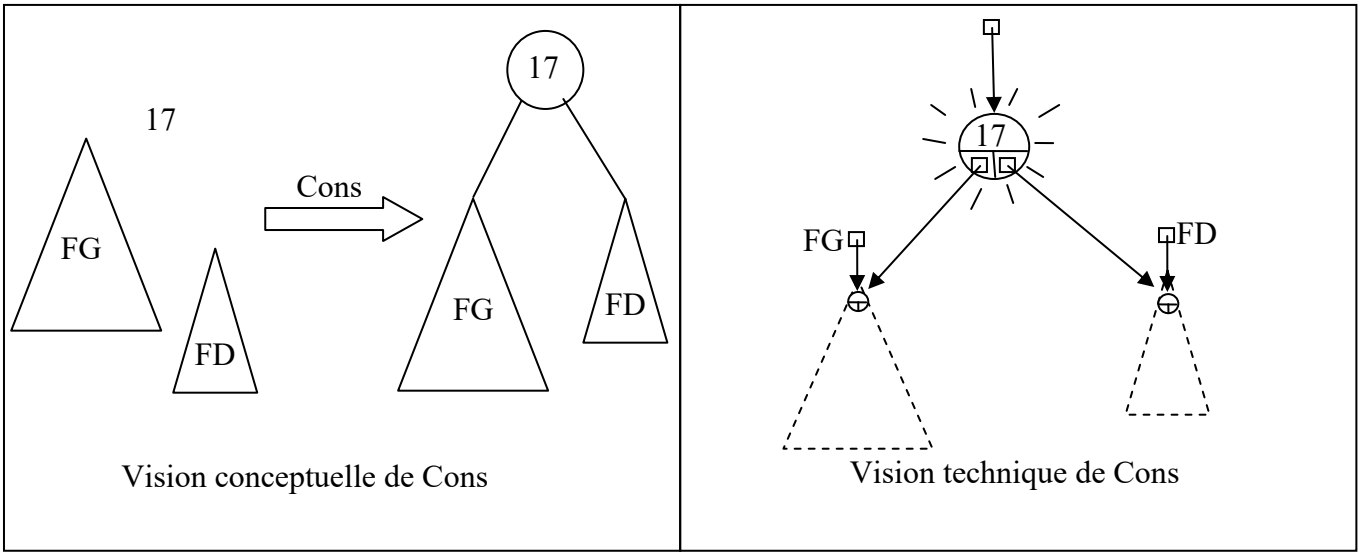
CONSTRUCTION
```
fonction     ConsVide() : ArbreBinaire[E]
// Restitue un arbre vide

fonction     Cons(↓X : E, ↓fg : ArbreBinaire[E], ↓fd : ArbreBinaire[E]) :
ArbreBinaire[E]
// Restitue un arbre de racine X et dont les fils sont fg et fd. Les deux fils sont
// utilisés tels quels, sans duplication.
// Voir la figure ci-dessous
```

CONSULTATION
```
fonction     EstVide(↓A : ArbreBinaire[E]) : booléen
// vrai ssi l'arbre est vide

fonction     Racine (↓A : ArbreBinaire[E]) : E
// Restitue la valeur du nœud racine → l'arbre ne doit pas être vide

fonction     FG (↓A : ArbreBinaire[E]) : ArbreBinaire[E]
// Restitue le fils gauche → l'arbre ne doit pas être vide

fonction     FD (↓A : ArbreBinaire[E]) : ArbreBinaire[E]
// Restitue le fils droit → l'arbre ne doit pas être vide
```

MODIFICATION
```
procedure    ModifRacine (↕ A : ArbreBinaire[E], ↓ X : TypeX)
// Remplace la valeur dans la racine par X → l'arbre ne doit pas être vide

procedure    ModifFG(↕A : ArbreBinaire[E], ↓NouvFG : ArbreBinaire[E])
// Remplace le fils gauche par NouvFG → l'arbre ne doit pas être vide

procedure    ModifFD(↕A : ArbreBinaire[E], ↓NouvFD : ArbreBinaire[E])
// Remplace le fils droit par NouvFD → l'arbre ne doit pas être vide
```

DESTRUCTION
```
procedure    Libérer(↓A : ArbreBinaire[E])
// Libère toute la mémoire dynamique occupée par l'arbre (donc destruction des noeuds).
// L'arbre n'est pas initialisé à NULL. Il ne doit plus être utilisé jusqu'à
// sa réinitialisation
```

Fonctionnement de Cons :



D'autres opérations (sur le TAD et la SDD)peuvent être définies à partir des opérations de base.

Exemple :

```
// Un traitement sur les arbres : le nombre de feuilles :
// - si l'arbre est vide, c'est 0
// - si les 2 fils sont vides, c'est 1
// - sinon c'est la somme des nombres de feuilles des deux fils
fonction NombreFeuilles ( ↓A : ArbreBinaire[E]) : entier
début
     si EstVide (A) alors
          NombreFeuilles ← 0
     sinon
          si EstVide (FG (A)) et EstVide (FD (A)) alors
               // Ce test est nécessaire sinon, pour une feuille, le
               // résultat serait nul.
               NombreFeuilles ← 1
          sinon
               NombreFeuilles← NombreFeuilles(FG(A))+NombreFeuilles (FD(A))
          fsi
fin
```

Dans le langage C la structure de données ArbreBinaire peut être implémentée par la déclaration :

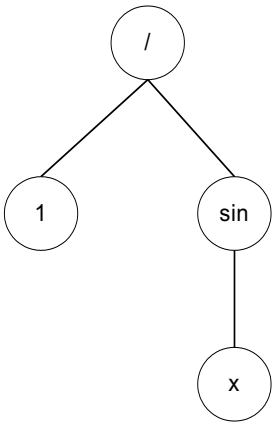
```
typedef struct Noeud {  TypeX              Valeur ;
                        struct Noeud  *fg,*fd ;
                     }
                     TNoeud, * TArbre ;
```

Exemple 1 : des arbres implémentés en C dont les valeurs sont des entiers

```
// Le type
typedef struct Noeud {  int           Valeur ;
                        struct Noeud  *fg,*fd ;
                     }
                     TNoeud, *TArbre ;

// Primitives
TArbre ConsVide ()
{
     return NULL ;
}
TArbre Cons (int v, TArbre fg, TArbre fd)
{
```

```
     TArbre    a = (TArbre) malloc (sizeof (TNoeud)) ;
     a->Valeur = v ;
     a->fg = fg;
     a->fd = fd ;
     return a ;
}

TArbre FG (TArbre A)
{
     return A->fg ;
}
void ModifFG (TArbre *PA, TArbre NouvFG)
{
     (*PA)->fg = NouvFG ;
}
…
// Exemple d'utilisation
TArbre    A1 = ConsVide (),
          A2 = Cons (2,A1,Cons (4,ConsVide (),ConsVide ())),
          A3,
          A4 = FG (A2) ;      // donc A4 = A1

A3 = Cons (6,A2,Cons (8,NULL,NULL)) ;      // déconseillé (NULL)

ModifFG (&A2, A3) ;          // FAUX : ceci introduit un cycle dans l'arbre
```

Exemple 2 : les arbres syntaxiques



Pour que tous les nœuds aient même type, il n'y a pas d'autre choix que de stocker cinq informations dans chaque nœud, certaines d'entre elles étant utilisées seulement pour un type de nœud donné :

- une valeur réelle
- un caractère
- des pointeurs vers les deux fils éventuels
- une information supplémentaire qui indique quelle est la nature du nœud et, donc, lesquelles des infos précédentes sont utiles. C'est la seule info qui est utilisée dans chaque nœud.

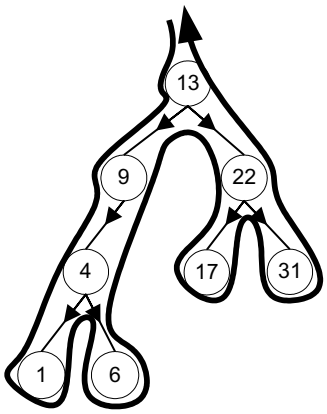
Si le nœud est une constante, l'autre info utilisée est le réel, pour conserver la valeur de la constante.
Si le nœud est un appel de fonction, le caractère sert à conserver la première lettre de la fonction et l'un des pointeurs donne l'accès à l'unique fils (l'argument).
Si le nœud est une opération binaire, le caractère sert à conserver l'opérateur et les pointeurs donnent accès aux deux fils (les deux opérandes).
Si le nœud est une variable, aucune des autres infos n'est utile.

Implémentation en C :

```
// Définition d'un type énuméré pour la nature du noeud
typedef enum {Constante,Variable,Binaire,Fonction} TNature ;

// Définition des types Nœud syntaxique et Arbre syntaxique.
// Le membre « nature » dit si le nœud est une constante, une variable etc etc …
// Si le nœud est binaire, l'opérateur est dans OperOuFct et les deux fils dans fg et fd.
// Si le nœud est une fonction, le premier caractère du nom de la fonction est dans OperOuFct
// et le fils est dans fg.
// Si le nœud est une constante, cette dernière est dans ValConst.
// Si le nœud est une variable, aucune autre info n'est utile.
typedef struct NoeudSynt {   TNature             Nature ;
                             double              ValConst ;
                             char                OperOuFct ;
                             struct NoeudSynt    *fg,*fd ;
                          }
                          TNoeudSynt, *TArbreSynt ;

// Pour la construction, le plus simple est d'avoir une fonction par type de noeud
TArbreSynt ConsVariable ()
{
     TArbreSynt    a = (TArbreSynt) malloc (sizeof (TNoeudSynt)) ;
     a->Nature = Variable ;
     // Aucun des autres membres n'est utile → on les laisse non initialisés

     return a ;
}

TArbreSynt ConsBinaire (char Op, TArbreSynt FilsG, TArbreSynt FilsD)
{
     TArbreSynt    a = (TArbreSynt) malloc (sizeof (TNoeudSynt)) ;
     a->Nature = Binaire ;
     a->OperOuFct = Op ;
     a->fg = FilsG ; a->fd = FilsD ;

     return a ;
}

// reste à définir ConsConstante et ConsFonction
…

// Exemple : création de l'arbre de : 1 / sin (x)
TArbreSynt    Un = ConsConstante (1),
              SinusX = ConsFonction ('s',ConsVariable ()),
              A = ConsBinaire ('/',Un,SinusX) ;
```

5.4 Les différents parcours d'un arbre

Un parcours en *profondeur* consiste à parcourir un arbre en privilégiant toujours la descente au déplacement latéral. La figure ci-dessous illustre le cheminement décrit par le parcours en profondeur d'un arbre :



L'algorithme du parcours en profondeur est très simple :

```
procedure Profondeur ( ↓a : ArbreBinaire[E])
{
     si non EstVide(a)
     {
          Profondeur(FG(a)) ;
          Profondeur(FD(a)) ;
     }
}
```

Habituellement, le parcours en profondeur permet de réaliser un traitement sur chaque nœud de l'arbre. Selon que le traitement est réalisé avant, entre ou après le parcours des deux fils, on parle de parcours préfixé, infixé ou postfixé :

- en préfixé : on traite d'abord la racine puis le fils gauche (dans l'ordre préfixé) puis le fils droit (dans l'ordre préfixé) ; on dit aussi préordre.
- en infixé : on traite d'abord le fils gauche (dans l'ordre infixé) puis la racine puis le fils droit (dans l'ordre infixé).
- en postfixé : on traite d'abord le fils gauche (dans l'ordre postfixé) puis le fils droit (dans l'ordre postfixé) puis la racine. ; on dit aussi postordre.

Le parcours *en largeur* (d'abord) consiste à parcourir l'arbre niveau par niveau, en commençant par celui de la racine. A l'inverse des précédents, la mise en œuvre est naturellement itérative.

Ces parcours sont tous en temps linéaire puisqu'ils parcourent tous les nœuds de l'arbre.

5.5 La hauteur d'un arbre

La hauteur H est le nombre de nœuds d'un chemin de la racine à la feuille la plus éloignée.

Si un arbre binaire contient N nœuds et *si il est parfaitement équilibré* (tous les niveaux sont saturés), on obtient le *moins* haut des arbres de N nœuds et on a :

$$N = 1 + 2 + 4 + 8 + \ldots + 2^{H-1}$$
$$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{H-1}$$
$$= 2^H - 1$$

donc

$$N + 1 = 2^H$$

donc

$$\log_2 (2^H) = \log_2(N+1)$$

donc

$$H = \log_2(N+1)$$

Si un arbre binaire contient N nœuds et *si il est dégénéré* (chaque nœud a exactement un fils sauf l'unique feuille), on obtient le *plus* haut des arbres de N nœuds et on a :

$$H = N$$

La hauteur d'un arbre binaire de N nœuds varie donc entre ces deux bornes :

$$\log_2(N+1) \leq H \leq N$$

Souvent, la complexité en temps des algorithmes sur les arbres est proportionnelle à la hauteur. Il est alors préférable que l'arbre soit aussi équilibré que possible.

Exercices :
Donner une méthode récursive de résolution des problèmes suivants et écrivez les sous-programmes récursifs qui implémentent ces solutions (ne pas gâcher de mémoire):

a) Calculer la taille (nombre de nœuds) d'un arbre.
b) Calculer le nombre de feuilles d'un arbre.
c) Calculer l'hauteur d'un arbre.
d) Tester la présence d'une valeur dans un arbre.
e) Retourner la liste infixée (resp. préfixée) des valeurs d'un arbre.

6 Arbres binaires ordonnés (ou arbres binaires de recherche)

Dans ce qui suit, on ne se sert jamais des valeurs mais seulement des clés qu'elles contiennent. On ne parle donc plus que de clés. Le type de la clé est appelé `TCle`.

Définition : un arbre binaire est ordonné ssi :

- il est vide

ou

- toutes les clés du fils gauche sont inférieures à la racine, la racine est inférieure à toutes les clés du fils droit et les fils sont eux-mêmes ordonnés

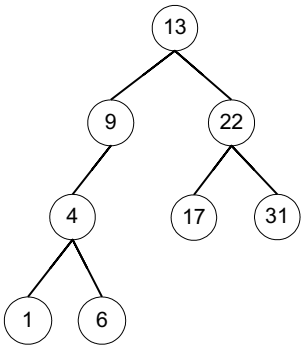
Définition équivalente : un arbre binaire est ordonné ssi :

- il est vide

ou

- la liste infixée de ses clés est ordonnée

Exemple :



Un arbre binaire ordonné

Pour la structure de données, on conserve celle des arbres binaires : un arbre est un pointeur sur une structure à trois membres : Clé (et non plus valeur), fg et fd. Comme la première information change de nom, on change le nom du type qui devient TABR (Type Arbre Binaire de Recherche).

Présence d'une clé c dans un ABR A :

Principe :

Si c < Racine (A), il n'y a aucune chance de la trouver dans le fils droit

→ on ne vérifie sa présence *que* dans le fils gauche

→ potentiellement, on diminue de moitié le nombre de valeurs à étudier à chaque étape.

Analyse :

si l'arbre est vide,
la clé est absente

sinon si la clé est égale à la racine,
elle est présente

sinon
on cherche la clé *soit* dans le fils gauche *soit* dans le droit selon qu'elle est < ou > à la racine (d'où appels récursifs).

D'où

```
fonction Presence (↓c : TClé, ↓a : TABR) : TABR
// Cette version restitue un pointeur sur le noeud qui contient la clé, donc
// un ABR, l'ABR vide à défaut
début
      si EstVide (a) alors          // Cas d'arrêt 1
            res ← ConsVide ()
      sinon si c = Racine (a) alors // Cas d'arrêt 2
            res ← a
      sinon si c < Racine (a) alors // On ne cherche qu'à gauche de la racine
            res ← Presence (c, FG(a))
      sinon                         // On ne cherche qu'à droite de la racine
            res ← Presence (c, FD(a))
      fsi fsi fsi
      retourne res
fin
```

Remarques :

- si EstVide (a) DOIT rester le premier test (sinon ça plante) ;
- puisque la recherche fait abstraction d'une moitié de l'arbre à chaque appel, les performances sont celles de la dichotomie, donc extraordinaires, sauf si la répartition des clés dans l'arbre n'est pas homogène. Dans le pire des cas, chaque nœud n'a qu'un fils, l'arbre dégénère donc en une liste et les performances sont celles de la recherche dans une simple liste triée. Si on a 1 000 000 de clés, la recherche exige donc de 20 (excellent) à 1 000 000 (moyen) de tests → les arbres ordonnés ne sont pas la panacée. D'autres arbres, en plus d'être ordonnés, sont aussi équilibrés, ce qui garantit une répartition homogène des clés dans tout l'arbre. On obtient alors des

performances extraordinaires, de l'ordre de 20 à 30 tests pour 1 000 000 de clés. De plus ces performances restent valables pour l'insertion ET la suppression !!!

- de manière plus formelle : la plupart des algorithmes ont une complexité en temps proportionnelle à la hauteur de l'arbre donc en O (h) (c'est le cas du test de présence). Si l'arbre est bien équilibré, la hauteur est proche de $\log_2$ (n), n étant le nombre de clés dans l'arbre et donc la complexité des algo est excellente : O ($\log_2$ (n)).
  Si l'arbre est dégénéré, la hauteur est égale à n donc la complexité des algo est médiocre : O (n) (bien que tout ce qui est polynomial est considéré par les algorithmiciens comme efficace).

- l'algo précédent suppose que l'on peut comparer directement deux clés du type TClé avec les opérateurs = = et <. Si ce n'est pas le cas, il faut écrire des fonctions de comparaison. Par exemple :

  ```
  if (c == Racine (a))
  ```

  devient

  ```
  if (Egal(c, Racine(a)).
  ```

- il est facile d'écrire une version itérative de cet algo : on descend à gauche ou à droite tant qu'on n'a ni épuisé l'arbre (auquel cas la clé est absente), ni trouvé la clé cherchée à la racine de l'arbre courant (auquel cas la clé est présente)

```
fonction PresenceVersionIterative (↓c : TClé, ↓a : TABR) : TABR
// Cette version restitue un pointeur sur le noeud qui contient la clé, donc
// un ABR, l'ABR vide à défaut
début
      Trouve ← faux
      tant que non EstVide(a) et non Trouve faire
            si c = Racine(a) alors
                  Trouve ← vrai
            sinon si c < Racine (a) alors
                  a ← FG(a)
            sinon
                  a ← FD(a)
            fsi fsi
      ftq
      retourne a
fin
```

Ajout d'une clé (supposée absente) dans un ABR A :

si A est vide,
le résultat est une feuille contenant la nouvelle clé
sinon
on ajoute la clé dans le fils gauche ou droit de A selon qu'elle est inférieure ou supérieure à la racine (→ récursivité).

Exemple :
Dessiner l'arbre obtenu par insertion des clés 4,2,5,1,3,6 dans un arbre initialement vide.

Bilan : les ajouts se font toujours « en bas » de l'arbre mais pas forcément sous une feuille.

Dessiner l'arbre obtenu par insertion des clés 3,1,2,5,4,6 dans un arbre initialement vide.

Bilan : l'arbre est différent bien que les valeurs soient les mêmes → la structure de l'arbre dépend de l'ordre d'insertion.

Dessiner l'arbre obtenu par insertion des clés 1,2,3,4,5,6 dans un arbre initialement vide.

Bilan : l'arbre est dégénéré donc les tests de présence et les insertions à venir seront en temps linéaire.

Conclusion :

Avec cet algorithme d'insertion, la nouvelle valeur est toujours accrochée **en bas** de l'arbre (ça peut paraître surprenant mais c'est normal).

D'où l'algo suivant :

```
procedure Insertion(↓C : TClé, ↕A : TABR)
début
     si EstVide (A) alors                  // Attention de commencer par ce test !!
          A ← Cons (C, ConsVide(), ConsVide())
     sinon
          si C < Racine (A)  alors         // Insertion à gauche
               f ← FG (A)
               Insertion (C,f)
               ModifFG (A,f)
          // attention, on peut pas faire Insertion(V,FG(A)) car la spécification
          // de l'ADT n'indique pas si FG(A) retourne l'arbre dans le fg ou une copie
          // (en C on pourrait Insertion (V,A->fg) ;
          sinon                            // Insertion à droite
               f ← FD (A)
               Insertion (C,f)
               ModifFD (A,f)
          fsi
     fsi
fin
```

Attention : une mise en œuvre fonctionnelle peut déboucher facilement sur une version qui gâche de la mémoire.

Exercice : Proposer une version fonctionnelle de l'algorithme.

Suppression d'une clé (supposée présente) :

Si elle est inférieure à la racine, on va supprimer à gauche,
Si elle est supérieure à la racine, on va supprimer à droite,
Si elle est égale à la racine, trois cas sont possibles :

- la racine est une feuille → le résultat est l'arbre vide
- la racine n'a qu'un fils → le résultat est ce fils
- la racine a deux fils :
  (approche déconseillée = remplacer l'arbre par son fils gauche et brancher le fils droit en dessous et à droite du plus à droite des nœuds du fils gauche)
  On remplace la clé de la racine par la plus grande clé du fils gauche et on supprime récursivement cette clé du fils gauche
  ou

on remplace la clé de la racine par la plus petite clé du fils droit et on supprime cette clé du fils droit.

Conclusion :

Les arbres binaires ordonnés ne sont pas adaptés à la réalisation de dictionnaires (a fortiori de BD) car on ne peut garantir la non dégénérescence de l'arbre en une liste → les performances peuvent être logarithmiques, si tous les niveaux sont saturés, ou seulement linéaires si l'arbre est dégénéré.

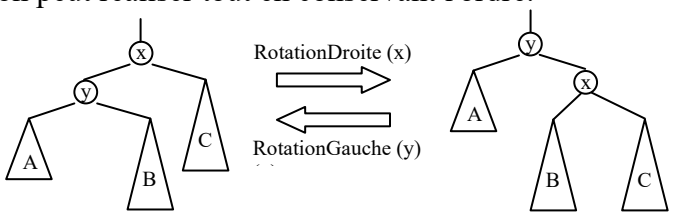
7 Quelques mots sur les arbres ordonnés équilibrés

Il y a diverses manières de réaliser un équilibrage d'un arbre ordonné :
- arbres binaires ordonnés à hauteurs équilibrées (arbres AVL) ;
- arbres binaires ordonnés à nœuds rouges et noirs (arbres RN) ;
- arbres quaternaires ordonnés (arbres 2-3-4) ;
- arbres n-aires ordonnés (B-arbres)
- et d'autres

Dans tous les cas, la hauteur reste un log du nombre de clés présentes ; on a ainsi la garantie que toutes les opérations qui sont en O (h) sont en réalité en O (log (n)). Et comme toutes ces SDD réalisent les 3 opérations d'un dictionnaire en O (h), ces opérations sont effectivement en O (log (N)) --> n'importe laquelle de ces SDD est une bonne réalisation d'un dictionnaire.

7.1 Deux opérations de base des arbres BINAIRES ordonnés : les rotations droite et gauche

Dans un arbre ordonné (équilibré ou non), il y a deux opérations, appelées rotations (vers la gauche ou vers la droite), que l'on peut réaliser tout en conservant l'ordre.



7.2 Définition des AVL

Un arbre ordonné est à hauteurs équilibrées si, en tout nœud,

$$\text{Hauteur (FG)} - \text{Hauteur (FD)} \in \{-1,0,1\}$$

Autrement dit, les deux fils d'un nœud ont toujours des hauteurs quasi identiques.

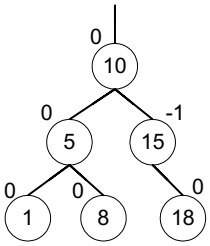
Les arbres AVL, ou arbres à *hauteurs équilibrées*, sont des arbres ordonnés qui ont cette propriété de rester équilibrés après insertion, suppression et fusion : la hauteur reste entre log (N+1) et 1.45*log (N+1).

Le nom d'AVL vient des deux concepteurs de ces arbres : **A**del'son-**V**el'skii et **L**andis

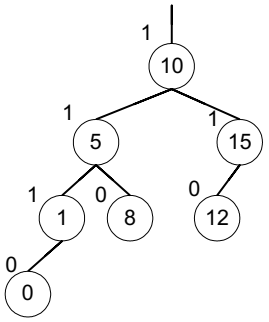
En plus des trois opérations de base, fusion, min, max, prédécesseur et successeur sont aussi en O (log (N)).

Structure de données : tout nœud est muni d'un Facteur d'Equilibre, qui est la différence de hauteur entre ses deux fils. Le FE est donc toujours dans {-1,0,1}.

Exemples :



Un arbre AVL



Un arbre AVL de hauteur 4
aussi déséquilibré que possible

Conséquences de la présence du FE :

- un paramètre de plus dans Cons :

```
a ← Cons (v,fe,fg,fd) ;
```

- deux primitives de plus :

```
fe ← FE (a)
```

et

```
ModifFE (a,NouvFE)
```

7.3 Présence d'une clé dans un AVL

C'est exactement le même algo que celui de la présence dans un ABR.

7.4 Insertion dans un AVL

Le schéma de base est le même que celui de l'insertion dans un ABR : on insère à gauche ou à droite selon que la valeur à insérer est inférieure ou pas à la racine.

Il y a deux différences :

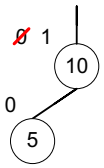
- l'insertion informe du fait que la hauteur de l'arbre augmente ou pas suite à l'insertion,
- après insertion dans le fils, si ce dernier grandi, il faut mettre à jour le facteur d'équilibre, voire rééquilibrer l'arbre à l'aide de rotations.

Exemple : insérer les clés 10, 5, 15, 2,1 dans un arbre initialement vide.

Insertion de 10 : on crée une feuille de FE nul (l'arbre grandit)

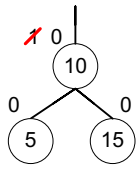
Insertion de 5 :

- 5 < 10 donc on insère à gauche
  - on insère 5 dans l'arbre vide : on crée une feuille de FE nul
- de retour sur 10, le fils où l'on a inséré a grandi → le FE passe de 0 à 1 (et l'arbre grandit)



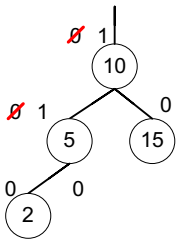
Insertion de 15 :

- 15 > 10 donc on insère à droite
    - on insère 15 dans l'arbre vide : on crée une feuille de FE nul
- de retour sur 10, le fils où l'on a inséré a grandi → le FE passe de 1 à 0 (mais l'arbre ne grandit pas)



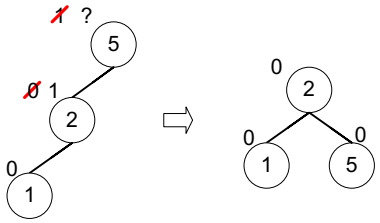
Insertion de 2 :

- 2 < 10 donc on insère à gauche
    - 2 < 5 donc on insère à gauche
        - on insère 2 dans l'arbre vide : on crée une feuille de FE nul (et l'arbre grandit)
    - de retour sur 5, le fils où l'on a inséré a grandi → le FE passe de 0 à 1 (et l'arbre grandit)
- de retour sur 10, le fils où l'on a inséré a grandi → le FE passe de 0 à 1 (et l'arbre grandit)

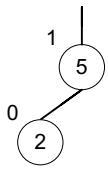


Insertion de 1 :

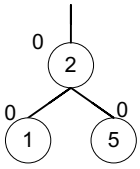
- 1 < 10 donc on insère à gauche
    - 1 < 5 donc on insère à gauche
        - 1 < 2 donc on insère à gauche
            - on insère 1 dans l'arbre vide : on crée une feuille de FE nul (et l'arbre grandit)
        - de retour sur 2, le fils où l'on a inséré a grandi → le FE passe de 0 à 1 (et l'arbre grandit)
    - de retour sur 5, le fils où l'on a inséré a grandi (la hauteur passe de 1 à 2) → le déséquilibre dépasse le seuil toléré → on doit ré-équilibrer l'arbre (une rotation droite sur 5 suffit).



On constate qu'après équilibrage, l'arbre qui avait 5 pour racine retrouve sa hauteur d'avant insertion :

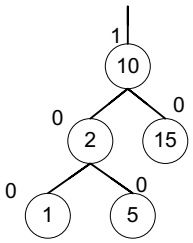


Avant insertion de 1 dans l'arbre de racine 5 : Hauteur = 2



Après insertion de 1 dans l'arbre de racine 5 : Hauteur = 2

- de retour sur 10, le fils où l'on a inséré a été restructuré mais il n'a pas grandi → le FE ne change pas et l'arbre ne grandit pas.

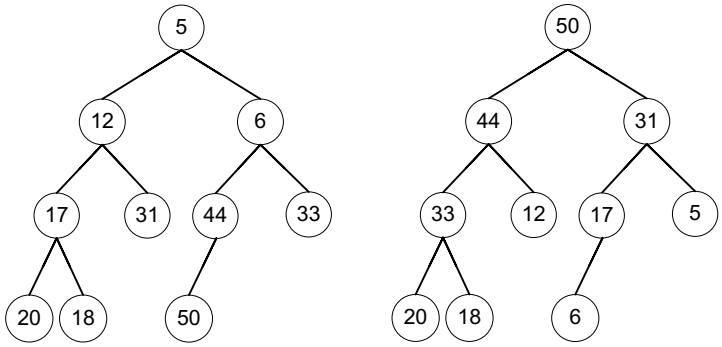


8 Les tas

8.1 Définition

Un tas est un arbre binaire muni d'un ordre *partiel* :
- pour un tas *minimier* (type TTasMin), chaque valeur est inférieure à sa descendance (les enfants, petits-enfants, arrières petits-enfants …)
- pour un tas *maximier* (type TTasMax), chaque valeur est supérieure à sa descendance

Exemple



Un tas minimier — Un tas maximier

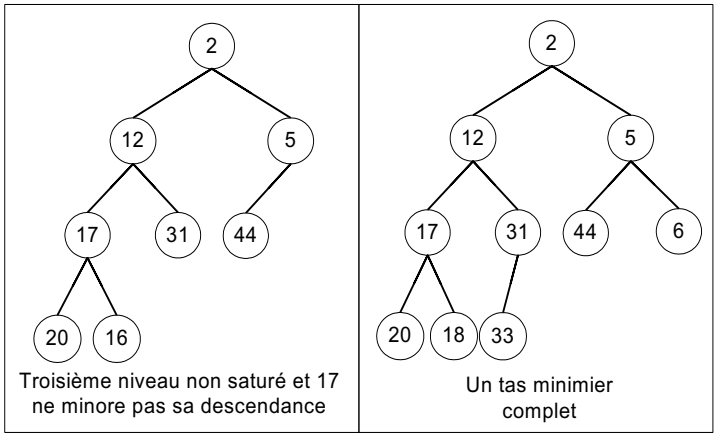
Les primitives sont les mêmes que celles des arbres.

C'est une structure très utilisée pour retrouver rapidement le minimum d'un ensemble (algo rapide de tri, réalisation efficace d'une file de priorité, recherche rapide d'un itinéraire dans un réseau). Mais elle est médiocre pour la vérification de la présence car la seule optimisation possible est d'arrêter la descente dans l'arbre quand la valeur cherchée est supérieure au nœud courant.

8.2 Tas minimiers complets

Un tas minimier complet est un tas minimier dont tous les niveaux sont saturés, sauf éventuellement le dernier. Dans ce cas, les valeurs du dernier niveau doivent être groupées à "gauche" du niveau.

Exemple :



Soit H la hauteur d'un tas minimier complet. Les niveaux saturés contiennent $1+2+2^2+\ldots+2^{H-2}$ nœuds = $2^{H-1} - 1$ noeuds. Le dernier niveau contient de 1 à 2^{H-1} valeurs. Donc

$$2^{H-1} - 1 + 1 \leq N \leq 2^{H-1} - 1 + 2^{H-1}$$

$$\Leftrightarrow \quad 2^{H-1} \leq N \leq 2^{H} - 1$$

$\Rightarrow \quad 2^{H-1} \leq N < 2^{H}$

$\Leftrightarrow \quad H-1 \leq \log_2 (N) < H$

donc

$\log_2 (N) < H \leq \log_2 (N) + 1 \rightarrow \boxed{H \approx \log_2 (N)}$

Tout arbre complet (donc un tas minimier complet) peut être représenté par un tableau dans lequel les valeurs sont rangées niveau par niveau.

Exemple : le tas minimier complet de la figure ci-dessus peut être décrit par le tableau suivant

0	1	2	3	4	5	6	7	8	9
2	12	5	17	31	44	6	20	18	33

Remarque :

- dans ce chapitre, les indices des tableaux commencent à 0 (les formules à venir changent si les indices commencent à un) ;

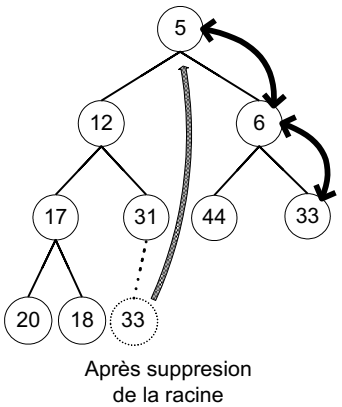
Remarques :

- appelons N le nombre de valeurs dans le tas (N=10 sur l'exemple) ;
- la seule valeur que l'on peut supprimer sans perdre le critère "complet" est la plus à droite des valeurs du dernier niveau. Elle se trouve en N-1 ;
- la position du père d'un nœud d'indice f est : (f-1) / 2 ;
- les nœuds des indices N/2 à N-1 sont des feuilles ;
- le fils gauche d'un nœud d'indice p (p < N/2) est en 2*p+1
- le fils droit d'un nœud d'indice p (p < (N-1)/2) est en 2*p+2

Suppression de la plus petite valeur d'un tas minimier complet.

La plus petite valeur est la racine. Pour la supprimer, on retire la dernière valeur DV du tas (on conserve ainsi le critère "complet") et on place DV à la racine → le min disparaît. Si nécessaire, on rétablit ensuite le critère "minimier" en échangeant DV avec le plus petit de ses fils tant que c'est nécessaire.

Exemple : suppression de la racine du tas minimier précédent



Implementation en C :

```
TValeur   RestitueEtSupprimeMinDuTas (TValeur Tas [], int *PtN)
{
          // Suppression de la racine
          TValeur   Min = Tas [0] ;
          Tas [0] = Tas [*PtN-1] ;
          (*PtN)-- ;

          // Rétablissement du critère "tas minimier"
          EchangeBas (Tas,0,*PtN) ;

          return Min ;
}

void      EchangeBas (TValeur Tas [], int i, int N)
// i est l'indice de la valeur en mouvement vers le bas. Si
// elle a au moins un fils (i < N/2) et si elle ne minore
// pas sa descendance, on l'échange avec le min de ses fils
// et on poursuit l'échange vers le bas.
{
          if (i < N/2)  // i a au moins un fils
          {
               // Détermination du plus petit fils
               int  fg = 2*i + 1,
                    fd = fg + 1,
                    FilsMin ;

               if (fd < N)    // fd existe donc il y a 2 fils
                    FilsMin = (Tas [fg] < Tas [fd]) ? fg : fd ;
               else           // Y a qu'un fils gauche donc il est le min
                    FilsMin = fg ;

               // Echange avec le fils min si nécessaire
               if (Tas [FilsMin] < Tas [i])
               {
                    Echange (Tas, FilsMin, i) ;

                    EchangeBas (Tas, FilsMin, N) ;
               }
          }
}
```

Complexité de la suppression du min :

Le coût de `RestitueEtSupprimeMinDuTas` se résume à celui de `EchangeBas`. Ce dernier est en O (H), H étant la hauteur du tas (puisque la nouvelle racine descend au pire jusqu'en bas). Et comme $H \approx \log_2 (N)$, `EchangeBas` est en O ($\log_2$ (N)).

Remarques :

- bien garder ce sous-programme EchangeBas en tête : il servira dans la mise en œuvre du très efficace tri par tas ;

9 Les tables (ou dictionnaires)

Une table/un dictionnaire est un ensemble S de valeurs tel que :

- chaque valeur est munie d'une information particulière, appelée clé,
- la clé est unique (deux valeurs différentes ont des clés différentes),
- les requêtes sur cet ensemble sont :
    - Rechercher (S, c) qui restitue la valeur de clé c si elle est présente dans S ou qui informe de l'échec de la recherche sinon (plusieurs prototypes possibles selon le contexte)
    - Inserer (S, V) qui insère une valeur V dans S (en principe, on a déjà vérifié que V est absente ; là encore, plusieurs protos) ;
    - Supprimer (S, V) qui supprime V de S (en principe, on a déjà vérifié que V est présente ; là encore, plusieurs protos).

Le dictionnaire est une SDD très utilisée en informatique. Par exemple, pour représenter un ensemble d'étudiants : la clé est le n° d'étudiant, les autres infos sont le nom, la date de naissance, les notes etc.

Structures de données possibles pour un dictionnaire :

- Tableau de valeurs indicé par les clés
  OK seulement si les clés sont entières (ou convertibles en entiers)

  Performances : excellentes : O (1) pour les trois opérations

  Encombrement : de parfait (clés consécutives) à inconcevable (clés très espacées)

  Conséquence : les tableaux ne sont pas la bonne solution

- Table d'adressage direct : un tableau, indicé par les clés, de pointeurs sur les valeurs
  OK seulement si les clés sont entières (ou convertibles en entiers)

  Perf : excellentes : O (1) pour les trois opérations

  Encombrement : moins mauvais que tableau mais reste intolérable si les clés sont très espacées

- Tableau de valeurs, trié sur les clés

  OK pour tous types de clés, à condition d'avoir un ordre sur les clés

  Perf : très bonnes pour la recherche : O (log (N)), moyennes pour insertion et suppression – O (N) – à cause des décalages et des réallocations si le tableau est dynamique.

  Encombrement : idéal si le tableau est dynamique, potentiellement mauvais si le tableau est surdimensionné.

- Liste (doublement) chaînée triée sur les clés

  OK pour tous types de clés, à condition d'avoir un ordre sur les clés

  Perf : recherche en O (N), insertion et suppression en O (1) (hors recherche).

Encombrement : presque idéal (1 ou 2 pointeurs par maillon en plus du strict nécessaire)

- Table de hachage. C'est un tableau de K listes chaînées de valeurs. Quand on ajoute une valeur, on calcule d'abord, à partir de sa clé, dans quelle liste elle doit être ajoutée. Ceci est fait par la fonction de *hachage*, dont l'entrée est la clé et dont la sortie est un indice dans le tableau (entre 0 et K-1). Ensuite, la valeur est ajoutée en tête de la liste choisie.
  Si la fonction de hachage est bonne, il y a presque autant de valeurs dans chaque liste (environ N / K).

  OK pour tous types de clés, à condition d'avoir une fonction de hachage qui peut les transformer en un indice avec une équiprobabilité pour chaque indice (sinon, certaines listes sont beaucoup plus longues que d'autres).

  Perf : supposons que la fonction de hachage est en O (1)
  - La recherche est en : O (1) + O (N/K) (parcours d'une liste).
    Si K = N/100 (donc K = Θ (N)) alors la recherche est en O (1) + O (100) = O (1) !!
    Si K << N (par exemple K = √N) alors la recherche devient lente

  - insertion en O (1) + O (1) donc O (1)

  - suppression en O (1) + O (N / K) donc même raisonnement que pour la recherche.

  Encombrement : bon (1 ou 2 pointeurs par maillon en plus du strict nécessaire plus le tableau de K pointeurs)
  ➔ super structure si le nombre de valeur (N) n'est pas trop grand devant le nombre de listes (K) et si une bonne fonction de hachage peut être trouvée.

Conclusion : aucune de ces réalisations n'offre à la fois compacité et efficacité garantie des trois opérations de base.

10 Algorithmes de tri

On distingue les tris internes (toutes les valeurs sont en mémoire vive, donc jusqu'à quelques dizaines ou centaines de milliers) et les tris externes (les valeurs sont dans un fichier et sont trop nombreuses pour tenir en mémoire vive). Les stratégies sont très différentes dans les deux cas : dans le premier, on se préoccupe essentiellement du nombre de comparaisons engendrées par le tri. En effet, le nombre total d'opérations est directement corrélé au nombre de comparaisons (par un facteur multiplicatif) et, donc, à la rapidité de l'algorithme. Ce nombre de comparaisons est une fonction du nombre de valeurs à trier. Pour un tri externe, le critère d'efficacité est la réduction du nombre d'accès au disque. Il faut réussir à trier le fichier complet en n'ayant toujours en mémoire qu'une partie du fichier et en optimisant surtout le nombre d'accès au fichier.

En l'absence de toute contrainte sur les N valeurs à trier, il est démontré que le meilleur algorithme de tri provoque au maximum O ($N.\log_2 N$) comparaisons. Les plus simples (tri-bulle, tri par échanges, tri par insertion …) sont en O (N^2) comparaisons. Cet écart n'est pas négligeable : si on a 10^6 valeurs à trier, le nombre de comparaisons réalisées, donc la complexité en temps, est en $k_1.20.10^6$ dans un cas, $k_2.10^{12}$ dans l'autre. A supposer que chaque opération dure 10^{-9} secondes, ça fait $k_1.20$ ms dans un cas, $k_2.10^3$ s dans l'autre, soit environ $k_2.17$ mn !!!

Si les valeurs à trier satisfont certaines contraintes (par exemple ce sont tous des entiers de l'intervalle [1..K], K étant une constante), il existe des tris internes en temps linéaire !

Complexité moyenne et complexité dans le meilleur des cas
Soit N valeurs à trier et soit un algorithme de tri donné.
Le pire des cas est la permutation de ces N valeurs qu'il est le plus long de trier.
Le meilleur des cas est la permutation de ces N valeurs qu'il est le plus rapide de trier.
Il est souvent facile d'exhiber ces deux cas et, donc, d'en déduire la complexité dans le meilleur et dans le pire des cas.
La complexité moyenne est celle correspondant à une permutation dont le tri représente une difficulté "moyenne". Il est extrêmement difficile de choisir une telle permutation. Mais, dans le cas particulier des tris, il existe d'autres approches pour calculer la complexité moyenne : par exemple, on somme les temps de tri de chaque permutation des N valeurs et on divise par le nombre de permutations (N !). On obtient le temps moyen.

10.1 De bons et de mauvais tris internes

10.1.1 Un mauvais algorithme de tri d'un tableau : le tri par échanges

Entrée : nombre N de valeurs à trier
Entrée/Sortie : le tableau T à trier

Principe de fonctionnement
Pour i de 0 à N-1, on échange la i^e valeur avec la plus petite valeur de l'intervalle [i..N].
A la fin de la i^e étape, le tableau est trié de 0 à i.

Exemple

6	10	1	2	4

Algorithme
(1) **pour** i **de** 0 **à** N-2 **faire**
(2) chercher la position PosMin du min entre i et N-1
(3) échanger T [i] et T [PosMin]
(4) **fpour**

Le pire des cas :
Aucune entrée n'est pire qu'une autre

Complexité du pire des cas :
(2) est en O (N - i)
donc l'algo est en $\sum_{1}^{N-1}(N-i)$ = somme des N-1 premiers entiers = O (N^2)

10.1.2 Un mauvais algorithme de tri d'un tableau: le tri par insertion

Entrée : nombre N de valeurs à trier
Entrée/Sortie : le tableau T à trier

Principe de fonctionnement

Au départ, T est trié de 0 à 0.

Pour i de 1 à N-1,

on insère T [i] dans l'intervalle [1..i] de manière à ce que cet intervalle soit trié.

→ A la fin de l'étape i, le tableau est trié de 0 à i.

Exemple

6	10	1	2	4

Algorithme

```
(1)   pour i de 1 à N-1 faire
(2)         V ← T [i], k ← i, Trouve ← faux
            // décalage à droite jusqu'à trouver une valeur inférieure à V
(3)         tant que (k > 0) et non Trouve faire
(4)               si (V >= T [k-1]) alors
(5)                     Trouve ← vrai
(6)               sinon
(7)                     T [k] ← T [k-1] et k ← k – 1
(8)               fsi
(9)         ftq

(10)        T [k] ← V
(11)  fpour
```

Complexité du pire des cas :

(3)..(9) le décalage concerne i éléments au maximum donc i*3 + 1

(2)..(10) = 3*i + 5

donc l'algo est en $1 + \sum_{2}^{N} 3.i + 5 = O(N^2)$

Le pire des cas :

le pire des cas est celui qui allonge (3), càd qui allonge le décalage → il faut que l'insertion se fasse toujours en tête → une séquence décroissante de valeurs

Le meilleur des cas :

(travail personnel)

10.1.3 Un mauvais algorithme de tri d'un tableau : le tri bulle

Entrée : nombre N de valeurs à trier

Entrée/Sortie : le tableau T à trier

Principe de fonctionnement

Parcourir le tableau de 0 à N-2 en échangeant T [i] et T [i+1] si T [i] > T [i+1].

A chaque fois que ce parcours a provoqué au moins un échange, le recommencer mais en arrêtant à chaque fois une case plus tôt

Exemple

6	10	1	2	4

Algorithme

```
(0)  Fin ← N-2

(1)  répéter
(2)       Echange ← faux
(3)       pour i de 0 à Fin faire
(4)            si (T [i] > T [i+1] )  alors
(5)                 Echanger T [i] et T [i+1]
(6)                 Echange ← vrai
(7)            fsi
(8)       fpour
(9)       Fin ← Fin-1
(10) jusque non Echange
```

Complexité du pire des cas :

Au pire, Fin décroît jusque 0. En effet, lorsque Fin atteint 0, il n'est plus possible que Echange devienne vrai donc la boucle "répéter" s'arrête. Les N parcours que cela engendre coûtent respectivement O (N-1), O (N-2) … O (1) soit O (N^2).

Le pire des cas :

Il survient si, dans chaque parcours, il y a au moins un échange → valeurs initialement par ordre décroissant.

Le meilleur des cas :

(travail personnel)

10.1.4 Un excellent algorithme de tri d'un tableau : le tri par tas

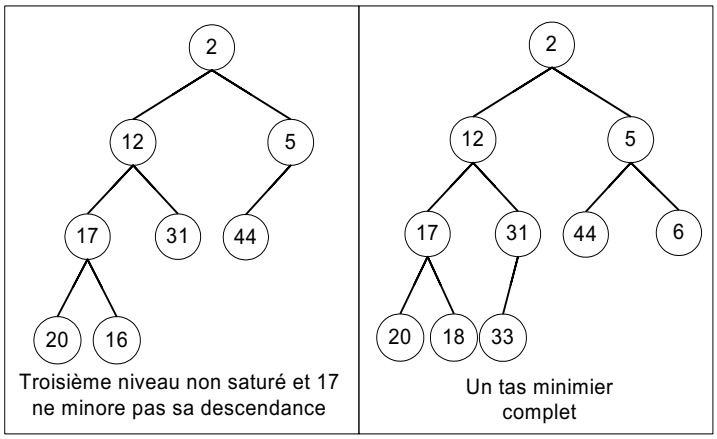
Rappels :

- un tas minimier complet est un arbre tel que :
  - tous les niveaux sont saturés, sauf éventuellement le dernier. Dans ce cas, les valeurs du dernier niveau doivent être groupées à "gauche" du niveau (critère "complet"),
  - tout nœud est un minorant de sa descendance (critère "tas minimier") ;
- la hauteur d'un tas minimier complet varie de $\log_2 (N)$ à $1+ \log_2 (N)$;
- tout arbre complet (donc un tas minimier complet) peut être représenté par un tableau dans lequel les valeurs sont rangées niveau par niveau.

Exemple : le tas minimier complet de la figure ci-dessous peut être décrit par le tableau suivant

0	1	2	3	4	5	6	7	8	9
2	12	5	17	31	44	6	20	18	33

Exemple :



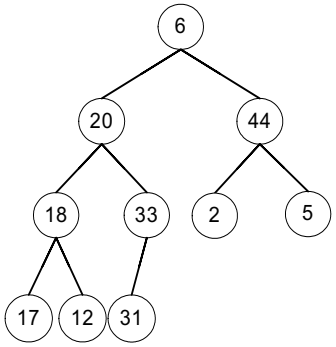
Le principe du tri par tas est de créer un tas minimier complet contenant les N valeurs à trier puis de retirer N fois la plus petite valeur du tas. On obtient ainsi une succession ordonnée de valeurs.

On a déjà vu que la suppression de la plus petite valeur (ExtraireMin) est en O (log (N)) donc la deuxième partie est en O (N.log (N)).

Constitution d'un tas minimier complet à partir d'un tableau de valeurs

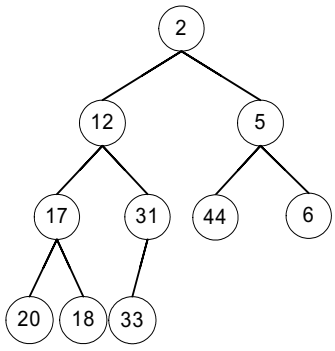
On veut réorganiser les valeurs d'un tableau de manière à ce qu'elles forment un tas minimier complet. Par exemple, il faut passer de :

0	1	2	3	4	5	6	7	8	9
6	20	44	18	33	2	5	17	12	31



à

0	1	2	3	4	5	6	7	8	9
2	12	5	17	31	44	6	20	18	33



Pour cela, on constate que les valeurs de N/2 à N-1 sont des feuilles donc des tas minimiers (d'une seule valeur).
Pour que la valeur en N/2 – 1 devienne un tas minimier, il suffit de lui appliquer un EchangeBas.
Idem pour N/2 – 2, N/2 – 3 etc jusqu'à 0.

Implementation en C :

```
void      TableauEnTas (TValeur Tab [], int N)
{
          int i ;

          // A ce stade, Tab est un arbre complet
          for (i = N/2 - 1 ; i >= 0 ; i--)
               EchangeBas (Tab, i, N) ;

          // A ce stade, Tab est un tas minimier complet
}
```

Complexité de la construction du tas :

`EchangeBas` est en O ($\log_2$ (N)), on l'appelle N/2 fois ➔ `TableauEnTas` est en O (N. $\log_2$ (N))

Tri par tas d'un tableau T de N valeurs

On transforme T en un tas

On supprime la plus petite valeur du tas. Ceci libère une place à la fin du tableau, qui passe de N à N-1 éléments → on y met le min

On répète ceci N-1 fois de plus → on obtient un tableau trié de manière **décroissante**.

Implementation en C :

```
void      TriParTas (TValeur Tab [], int N)
{
          int i ;

          // Tab est un tableau simple, donc aussi un arbre complet

          TableauEnTas (Tab,N) ;
          // Tab est maintenant un tas min complet

          // N fois, on supprime le min et on le stocke dans la case libérée
          // (N-1 fois suffiraient)
          for (i = N-1 ; i >= 0 ; i--)
               Tab [i]   = RestitueEtSupprimeMinDuTas (Tab, &N) ;
}
```

Complexité :

Construction du tas = O (N. $\log_2$ (N))

N suppressions = N.O ($\log_2$ (N)) = O (N. $\log_2$ (N))

D'où une complexité totale en O (N. $\log_2$ (N))

Maison : faire les adaptations pour que le tableau soit trié de manière *croissante* sans utiliser de structure de données annexe.

10.1.5 Le meilleur algorithme de tri d'un tableau : le tri rapide (quicksort)

*Le quicksort est paradoxal : sa complexité dans le pire des cas est O (N^2), soit nettement moins bien que le tri par tas. Pourtant, le pire des cas survient si peu souvent qu'**en moyenne, il est meilleur**.*

Principe :

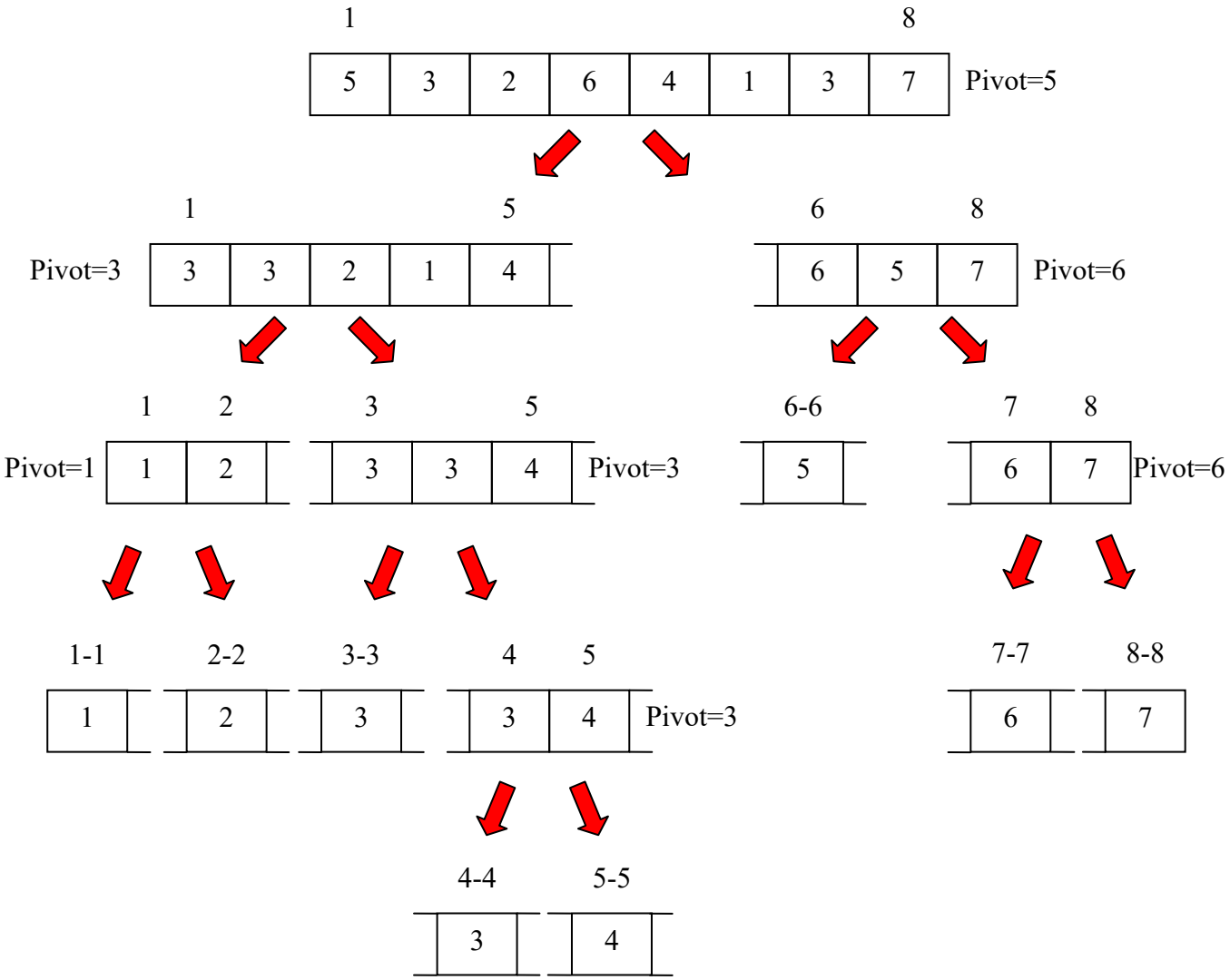
Les paramètres sont le tableau T et les indices de début et de fin de la partie à trier. Initialement deb = 1 et fin = N.

Si deb < fin

1. on choisit une valeur dite pivot P, par exemple P = T [deb]
2. on réorganise les valeurs de l'intervalle [deb,fin] de manière à n'avoir d'abord que des valeurs ≤ au pivot (appelons k l'indice de fin de cette partie) puis que des valeurs ≥ P (donc de k+1 à fin), c'est la *partition*.
   Cette étape est délicate, n'importe quelle procédure de partition ne convient pas.
3. on trie récursivement les parties [deb..k] et [k+1..fin]
4. le tableau est alors trié.

Sinon
    Rien à faire
Fsi

Exemple :



Si on n'a pas de chance, toutes les valeurs sont d'un même côté du pivot et, au lieu de couper le tableau en deux parties de tailles semblables, on a une partie réduite à une valeur et une autre avec toutes les autres.

Si cette malchance se poursuit à chaque étape, on tombe dans le pire des cas qui se traite en O (N^2). En effet, considérons une étape donnée et supposons qu'elle porte sur K valeurs :

- il faut parcourir les K valeurs de l'intervalle pour faire la partition,
- on traite ensuite récursivement un intervalle de taille 1 (immédiat) et un autre de taille K-1

La complexité totale est donc

$C(N) = N + C(N-1)$ avec $C(1) = 1$ d'où

d'où

$$C(N) = \frac{N(N+1)}{2} = O(N^2)$$

Pour information, voici la procédure de partition :

```
fonction Partition ( ↕tab : Tableau; ↓deb : entier; ↓fin : entier) : entier
     // Le résultat est l'indice auquel s'arrête la séquence de valeurs <=
     // au pivot
     début
     i ← deb – 1
     j ← fin + 1
     Pivot ← tab[deb]

     tant que Vrai faire
                 répéter
                 j ←□□□
                 jusque tab[j] <= Pivot

                 répéter
                 i ← i+1
                 jusque tab[i] >= Pivot

                 si (i < j) alors
                 Echanger tab[i] et tab[j]
                 sinon
                 retourner j
                 fsi
     ftq
fin
```

10.1.6 Conclusion sur les tris internes

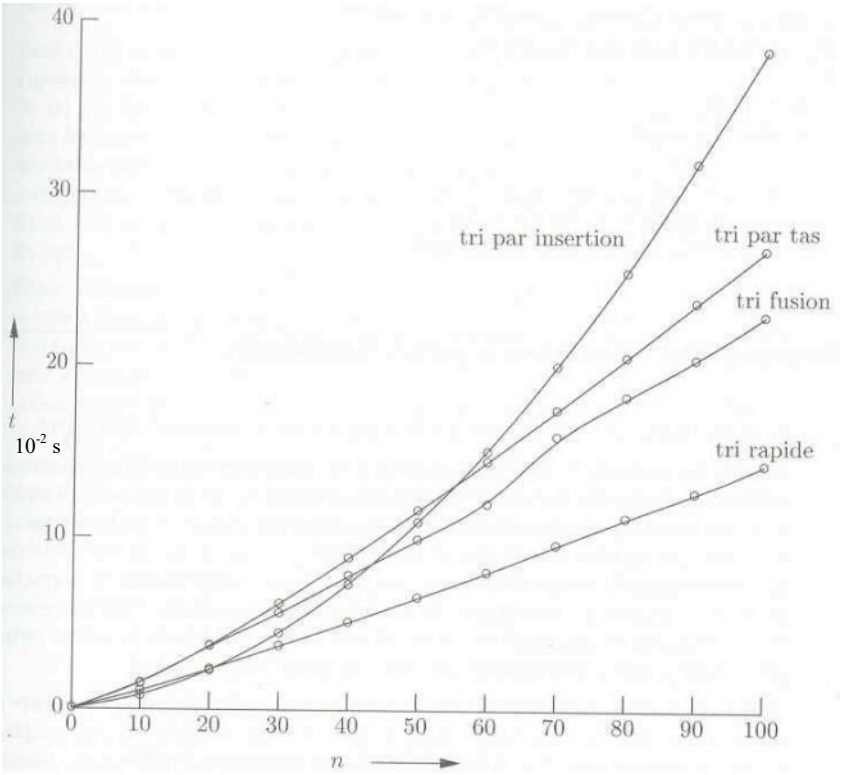


Figure tirée du livre de E. Horowitz, S. Sahni, S. Anderson-Freed, *l'essentiel des structures de données en C* (Dunod)

10.2 Un tri externe : le tri "fusion"

L'algo qui suit est une adaptation d'un tri qu'on appelle le tri fusion.

Problème

Trier les N valeurs d'un fichier F dont une fraction seulement peut être chargée en mémoire et en minimisant le nombre d'accès au disque (qui est LE facteur influant sur les performances, dans le cas d'un tri externe).

Principe

Phase 1 : tant qu'on n'a pas épuisé F, on charge une partie des valeurs, on la trie et on la sauve sur un fichier F_i

Phase 2 : on fusionne ensuite les fichiers obtenus

Présentation détailée

Soit :

- "Bloc" la quantité d'informations saisie en un accès fichier (>> à une unique valeur)
- NB le nombre de blocs qu'il est possible de charger en mémoire vive

La phase 1 est facile : on lit NB blocs de F, on trie l'ensemble avec un bon tri interne (quicksort ou tri par tas), on sauve le tout dans un fichier intermédiaire et on recommence

jusqu'à avoir épuisé tous les blocs du fichier. Soit NF le nombre de fichiers intermédiaires ainsi générés.

La phase 2 est plus subtile : on ne peut pas se contenter d'ouvrir les NF fichiers intermédiaires et de procéder à une fusion : ceci oblige à lire un bloc dans chaque F_i et cela fait trop de blocs pour la mémoire vive → on fusionne F_1, F_2, F_3 … F_{NB-1} puis F_{NB}, F_{NB+1}, F_{NB+2} … $F_{2.NB-1}$ puis … (cf ci-dessous) ; ceci engendre NF / (NB-1) fichiers intermédiaires plus gros, que l'on fusionne sur le même principe. On poursuit jusqu'à ce qu'il ne reste qu'un fichier, que l'on renomme en F après avoir détruit le F original.

Pour fusionner NB-1 fichiers

On lit un bloc dans chaque fichier → NB-1 blocs

On parcourt "en parallèle" (comme le veut la fusion) ces NB-1 blocs et on stocke les valeurs retenues dans un $NB^{ème}$ bloc.

A chaque fois que le bloc NB est plein, on l'écrit dans le fichier de sortie.

A chaque fois que l'un des NB-1 premiers blocs est épuisé, on lit un nouveau bloc dans le fichier correspondant, s'il en reste.

Tout ceci se poursuit jusqu'à épuisement des NB-1 fichiers à fusionner.