Master Sciences, Technologies, Santé
Mention Mathématiques, spécialité Enseignement des mathématiques

N1MA0011

Algorithmique et graphes, thèmes du second degré

COMPLEMENTS D'ALGORITHMIQUE

Éric SOPENA
Eric.Sopena@labri.fr

Ce document fait suite au document « Algorithmique de base pour le lycée », et présente des notions se situant au-delà du programme actuel des classes de seconde, première et terminale.

SOMMAIRE

Chapitre 1. Actions et fonctions 3
1.1. Actions 4
1.2. Fonctions 5
1.3. Visibilité des variables 6
1.4. La récursivité 6
Chapitre 2. Constructeurs de types....10
2.1. Les intervalles....10
2.2. Les énumérations....10
2.3. Les agrégats 11
2.4. Les tableaux....11
2.5. Les fichiers texte 15
2.6. Autres types de fichiers 15
Chapitre 3. Les types abstraits 17
3.1. La notion de type abstrait 17
3.2. Piles et files....18

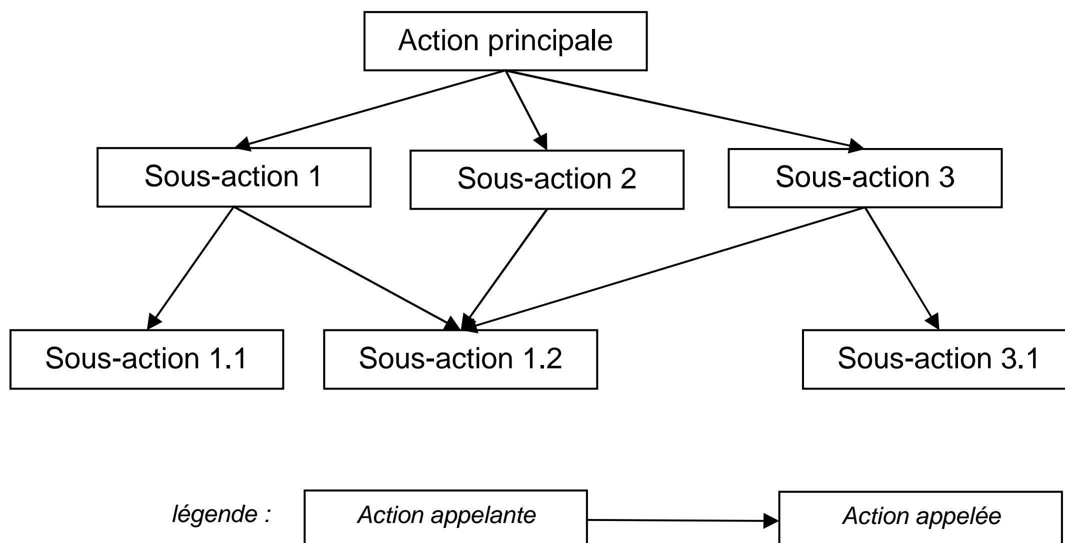
3.2.1. Les piles 18
3.2.2. Les files 20
3.2.3. Implémentation des piles et des files....................20
3.3. Les listes 20
3.4. Les arbres 22
3.4.1. Arbres binaires de recherche 24
3.4.2. Implémentation des arbres 27
3.4.3. Algorithme de Huffman....................29
Chapitre 4. Complexité des algorithmes....................33
4.1. Expression de la complexité d'un algorithme 33
4.2. Détermination de la complexité d'un algorithme....................34
4.3. Quelques exemples de calcul de complexité....................35

Chapitre 1. Actions et fonctions

La conception d'un algorithme procède en général par des affinements successifs : on décompose le problème à résoudre en différents sous-problèmes, puis les sous-problèmes à leur tour si nécessaire, jusqu'à obtenir des problèmes « faciles à résoudre » ou dont la solution est déjà connue (*méthode d'analyse descendante*).

Ainsi, l'algorithme obtenu se composera d'une action principale, qui réalise le traitement principal, et d'un certain nombre de sous-actions. L'action principale aura généralement pour but d'organiser l'enchaînement de plusieurs sous-actions, les sous-actions correspondront aux sous-problèmes dégagés lors de la phase de décomposition. Selon la complexité des sous-problèmes, les sous-actions correspondantes peuvent elles-mêmes faire appel à d'autres sous-actions.

D'une façon générale, on obtient un schéma, dit *graphe des appels*, tel que par exemple celui-ci :



Au niveau algorithmique, on ne fera pas de distinction entre la notion d'action principale et de sous-action : toutes deux sont des actions (une action principale un jour, peut devenir sous-action le jour suivant, bien qu'il s'agisse toujours de la même action). C'est souvent différent en programmation. Par ailleurs, on distinguera les notions d'action et de fonction :

- les actions réalisent un certain traitement (e.g. lecture de données complexes, tri d'un fichier étudiant par ordre alphabétique, etc.),
- les fonctions effectuent un calcul et retournent un résultat (e.g. périmètre d'une figure géométrique, racine d'une équation, etc.), ce sont simplement des *actions particulières*, dont l'utilisation sous forme de fonction est plus « souple » que sous forme d'action.

En Python, on ne retrouve que la notion de fonction. L'action principale est alors composée de l'ensemble des instructions non contenues dans des fonctions. La notion de fonction ou d'action n'existe pas en AlgoBox[1].

Le découpage en actions et/ou fonctions facilite le *travail en équipe* à condition

[1] La notion de *fonction* en AlgoBox est restreinte à la notion de fonction mathématique du type *f(x)* = … On peut définir de telles fonctions pour les utiliser ensuite dans des expressions numériques.

- de bien spécifier chacune des actions (données de départ, traitement à réaliser, résultats à fournir),
- de bien préciser les communications entre les différentes actions (transmission de données : les résultats, les données d'entrées, le type de ces données, etc.).

Les données permettant la communication entre actions peuvent être classées en trois catégories :

- Les paramètres d'entrée (E) sont des données fournies à l'action appelée par l'action appelante. Elles sont disponibles dès le début de l'action appelée.
- Les paramètres de sortie (S) correspondent aux résultats dont les valeurs sont renvoyées à l'action appelante lorsque l'action appelée se termine.
- Les paramètres d'entrée-sortie (ES), sont des données fournies à l'action appelée par l'action appelante, et donc disponibles dès le début de l'action appelée, puis modifiées (éventuellement) par l'action appelée et renvoyées à l'action appelante lorsque l'action appelée se termine. Ces paramètres peuvent ainsi être vus comme étant à la fois en entrée et en sortie.

1.1. Actions

Le format général de déclaration d'une action est le suivant :

```
Action nomAction ( <liste_paramètres> )
# spécification
variables   <variables locales>
début
   <corps>
fin
```

Le commentaire « spécification » permet d'indiquer ce que fait l'action et de préciser le rôle des paramètres.

Pour chaque paramètre, on précisera son nom, son type et sa catégorie (E = entrée, S =sortie, ES = entrée-sortie). La valeur d'un paramètre d'entrée ou d'entrée-sortie est disponible dès le début de l'action. La valeur d'un paramètre de sortie est calculée par l'action, celle d'un paramètre d'entrée-sortie est modifiée par l'action. Voici un exemple complet permettant d'en préciser la syntaxe :

```
Action nettoieListe (    ES maListe : Liste d'entiers naturels ;
                         E elem : entier naturel ;
                         S nbSupprimés : entier naturel )
# cette action supprime toutes les occurrences de elem dans maListe
# et renvoie dans nbSupprimés le nombre d'éléments supprimés
variables   i, nbEléments : entier naturel
            nouvelleListe : liste d'entiers naturels
début
       # initialisations
   nouvelleListe ← [ ]    # pour calculer la liste résultat
   nbSupprimés ← 0
   nbEléments ← NombreEléments (maListe)
       # boucle de traitement
   pour i de 0 à nbEléments - 1
   faire    si ( maListe[i] = elem )
            alors        # l'élément n'est pas recopié
                 nbSupprimés ← nbSupprimés + 1
            sinon        # l'élément est recopié
                 nouvelleListe ← nouvelleListe + [ maListe[i] ]
            fin_si
   fin_pour
       # on met à jour la liste paramètre si nécessaire
   si ( nbSupprimés > 0 )
```

```
  alors    maListe ← nouvelleListe
  fin_si
fin
```

On ne doit jamais modifier un paramètre d'entrée dans le corps d'une action : l'effet d'une telle opération peut en effet varier d'un langage de programmation à l'autre. Si besoin est, il suffit de recopier le paramètre en question dans une variable de travail.

Une action s'utilise ensuite comme s'il s'agissait d'une instruction prédéfinie, en spécifiant les *arguments* d'appel. Le lien entre paramètres et arguments se fait selon leur position (le premier argument est associé au premier paramètre, etc.).

Un argument associé à un paramètre d'entrée peut être n'importe quelle expression (variable, constante, littéral, etc.) *de type compatible*, et sa valeur n'est pas modifiable par l'action. Un argument associé à un paramètre de sortie ou d'entrée-sortie est *nécessairement* une variable car, à l'inverse, on doit pouvoir lui affecter une valeur.

Voici quelques appels corrects de l'action `nettoieListe` :

```
  variables     liste1 : liste d'entiers naturels
                entier1, nb : entiers naturels
...
  nettoieListe ( liste1, entier, nb )
...
  nettoieListe ( liste1, 14, nb )
...
  nettoieListe ( liste1, 4*entier1 + 7, nb )
```

1.2. Fonctions

Une fonction est une action particulière, possédant un certain nombre de paramètres d'entrée et *un seul* paramètre de sortie[2]. La syntaxe d'une fonction est définie pour permettre une utilisation souple de celle-ci :

```
Fonction nomFonction ( <liste_paramètres_entrée> ) : <type_du_résultat>
# spécification
variables   <variables locales>
début
  <corps>   # doit contenir au moins un Retourner(<valeur>)
fin
```

Les paramètres de la fonction étant *nécessairement* des paramètres d'entrée, il sera inutile de spécifier le « E » correspondant. Le calcul de la valeur retournée par la fonction est généralement effectué grâce à une variable locale. L'instruction `Retourner(<valeur>)` permet d'interrompre l'exécution de la fonction et de renvoyer une valeur.

Voici l'exemple d'une fonction calculant le maximum de deux entiers :

```
Fonction maximum ( a, b : entiers ) : entier
# cette fonction détermine le plus grand des entiers a et b
variables   max : entier
début
       # détermination du maximum
  si ( a > b )
  alors max ← a
  sinon max ← b
  fin_si
       # retour du résultat
  Retourner ( max )
fin
```

[2] Notons qu'en Python, une fonction peut retourner *plusieurs* résultats grâce au « return multiple ».

Notons que sur cet exemple, il n'est pas nécessaire d'utiliser une variable locale pour calculer le résultat de la fonction. On peut en effet écrire cette fonction de la façon suivante :

```
Fonction maximum ( a, b : entiers ) : entier
# cette fonction détermine le plus grand des entiers a et b
début
       # détermination et retour du maximum
   si ( a > b )
   alors Retourner ( a )
   sinon Retourner ( b )
   fin_si
fin
```

Les règles concernant la nature des arguments d'appel sont identiques à celles vues pour les actions mais, cette fois, tous les arguments correspondent à des paramètres d'entrée. Lorsqu'un appel de fonction est évalué, on obtient la valeur retournée par la fonction. Un tel appel peut ainsi être utilisé à l'intérieur d'une expression quelconque, pourvu que les types soient respectés. Voici par exemple quelques utilisations correctes d'un appel à la fonction `maximum` :

```
   variables     x, y, z : entiers
   ...
   Afficher ( maximum (x, y) )
   Afficher ( maximum (17, 2*y) )
   z ← maximum (2*z, y-1) + 5 * maximum (x, y)
```

On comprend bien sur ces exemples que la définition des fonctions permet une utilisation « souple » de celles-ci…

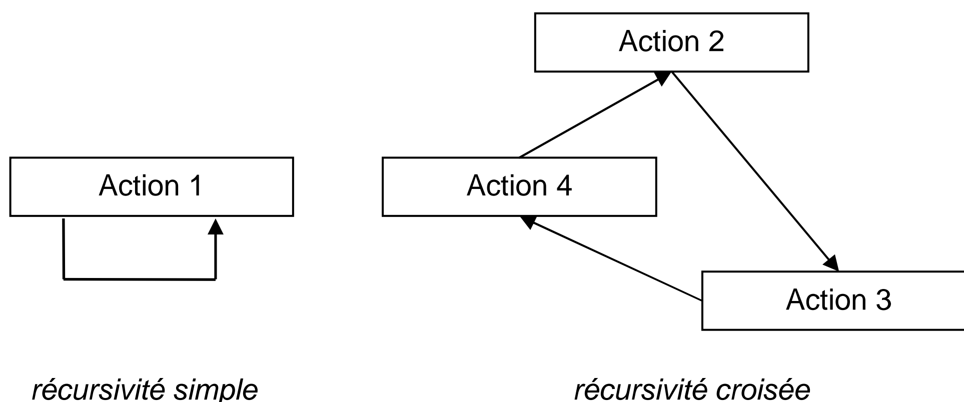
1.3. Visibilité des variables

Les variables locales, définies au sein d'une action ou d'une fonction, ne sont visibles (c'est-à-dire connues) qu'au sein de l'action ou de la fonction. Deux variables ayant même identificateur mais définies au sein d'actions différentes sont donc distinctes.

Les règles de visibilité des langages de programmation sont similaires. Il est généralement possible d'utiliser également des variables *globales* (visibles dans toutes les actions/fonctions), mais cela réduit considérablement la réutilisabilité des actions/fonctions qui utiliseraient de telles variables…

1.4. La récursivité

Une action ou une fonction est dite *récursive* lorsqu'elle s'utilise elle-même. On parle de *récursivité croisée* lorsque la récursivité fait intervenir au moins deux actions ou fonctions (e.g une action A appelle une action B qui, elle, appelle l'action A). Ainsi, il y a récursivité dès que le graphe des appels contient un circuit :



récursivité simple *récursivité croisée*

Les algorithmes récursifs sont en règle générale plus concis et plus simples à concevoir, et sont bien adaptés à l'écriture d'algorithmes sur les objets définis de façon inductive[3]. Dans le cas de l'écriture de fonctions, la récursivité permet d'avoir une expression « proche » de la définition mathématique de la fonction.

Attention cependant au problème de complexité, en temps et en espace, car un algorithme récursif n'est jamais plus efficace (et parfois beaucoup moins efficace !) que le même algorithme en version itérative.

Voici par exemple la version récursive de la fonction `factorielle` :

```
Fonction factorielle ( n : entier naturel ) : entier naturel
   # cette fonction calcule la factorielle d'un entier naturel n
début
   si ( n < 2 )
   alors Retourner (1)
   sinon Retourner (n * factorielle(n-1))
   fin_si
fin
```

et sa version itérative :

```
Fonction factorielle ( n : entier naturel ) : entier naturel
   # cette fonction calcule la factorielle d'un entier naturel n
variables   fact, i : entier naturel
début
   fact ← 1
   pour i de 2 à n
   faire fact ← fact * i
   fin_pour
   Retourner (fact)
fin
```

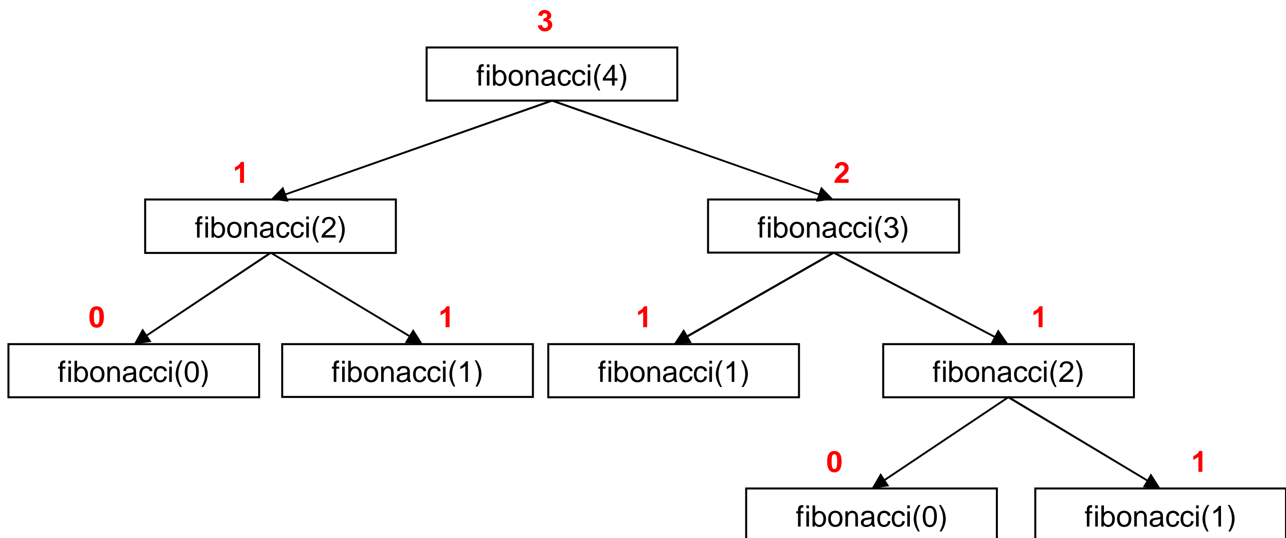
Concernant la complexité en temps, ces deux versions sont équivalentes (le calcul de n! nécessite n-1 multiplications). Par contre, pour ce qui est de la complexité en espace, la version récursive est beaucoup plus gourmande, car plusieurs appels sont simultanément en cours 'et donc en mémoire).

Considérons maintenant le calcul du n-ième nombre de Fibonacci. En version récursive :

```
Fonction Fibonacci ( n : entier naturel ) : entier naturel
   # cette fonction calcule le n-ième nombre de Fibonacci
début
   selon que
   n = 0 : Retourner (0)
   n = 1 : Retourner (1)
   sinon : Retourner (fibonacci(n-2) + fibonacci(n-1))
   fin_selon
fin
```

Le graphe des appels engendré par l'appel `fibonacci(3)` est le suivant :

3 Voir le document « Types abstraits inductifs ».



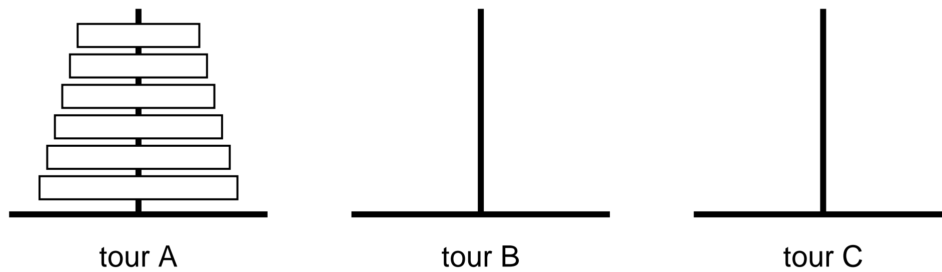
On observe que le nombre d'appels générés par `Fibonacci(n)` est de l'ordre de 2^n, et que de nombreux appels sont effectués plusieurs fois ! Cette version est donc bien moins efficace que la version itérative qui calcule cette valeur en effectuant de l'ordre de 2n additions :

```
Fonction Fibonacci ( n : entier naturel ) : entier naturel
  # cette fonction calcule le n-ième nombre de Fibonacci
variables   fibo, prec, i : entiers naturels
début
       # initialisations, prec = valeur précédente
  fibo ← 1
  prec ← 0
       # test si cas simple ( n = 0 )
  si ( n = 0 )
  alors    Retourner ( 0 )
  sinon        # boucle de calcul pour le cas général
           pour i de 2 à n
           faire    fibo ← fibo + prec     # f(n) <- f(n-1) + f(n-2)
                    prec ← fibo - prec     # prec <- f(n-1)
           fin_pour
               # retour résultat
           Retourner ( fibo )
  fin_si
fin
```

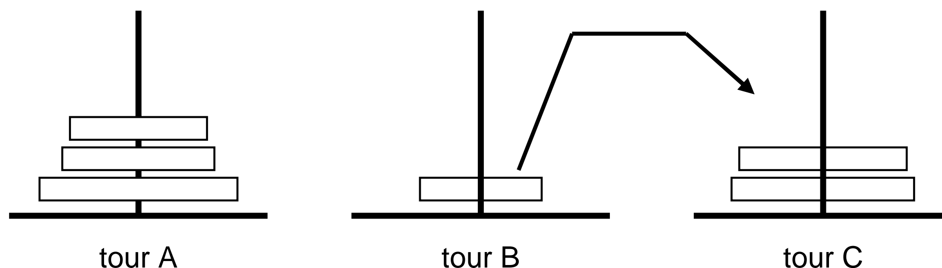
Le problème du calcul du n-ième nombre de Fibonacci est un problème de complexité linéaire. La solution récursive, de complexité exponentielle, n'offre donc pas une solution intéressante. Mais pour certains problèmes de nature exponentielle, une solution récursive est souvent beaucoup plus facile à concevoir. C'est le cas notamment du célèbre problème des *Tours de Hanoï*.

On dispose de trois tours A, B et C. La tour A comprend N disques, de diamètres deux à deux distincts, rangés par diamètres décroissants. Un déplacement élémentaire consiste à déplacer le disque se trouvant au sommet d'une tour vers une autre tour, à condition que cette autre tour soit vide ou que le disque se trouvant à son sommet soit de diamètre supérieur à celui du disque déplacé. Le but du jeu est alors de déplacer l'ensemble des disques de la tour A vers la tour C en effectuant une séquence de déplacements autorisés.

Configuration initiale :



Un exemple de déplacement autorisé :



Une solution récursive s'obtient facilement en observant que « *pour déplacer N disques d'une première tour vers une troisième, il suffit de déplacer les N-1 premiers disques sur la deuxième tour, puis de déplacer le dernier disque sur la troisième tour et, enfin, de déplacer les N-1 disques restants de la deuxième vers la troisième tour* »… On obtient alors l'algorithme suivant :

```
Action ToursHanoi (  E n : entier naturel,
                     E tour1, tour2, tour3 : caractères )
   # cette action affiche la liste des déplacements élémentaires à
   # effectuer pour déplacer n disques de la tour1 vers la tour3
   # (les tours sont codées par un caractère, e.g. 'A', 'B' et 'C')
début
   si ( n = 1 )
   alors    # cas simple
        Afficher ( tour1, ' ---> ', tour3 )
   sinon    # la récursivité permet de déplacer n-1 disques aisément
        ToursHanoi ( n-1, tour1, tour3, tour2 )
        Afficher ( tour1, ' ---> ', tour3 )
        ToursHanoi ( n-1, tour2, tour1, tour3 )
   fin_si
fin
```

Un appel à cette action aura la forme suivante :

```
tour1 ← 'A'
tour2 ← 'B'
tour3 ← 'C'
ToursHanoi(6, tour1, tour2, tour3)
```

Il suffit d'essayer d'en fournir une version itérative pour se convaincre de la difficulté du problème[4] !

[4] Le document « Types abstraits récursifs » indique comment supprimer « automatiquement » la récursivité dans un algorithme pour en obtenir une version itérative.

Chapitre 2. Constructeurs de types

Nous avons jusqu'ici utilisé essentiellement des types élémentaires, que l'on retrouve généralement sous forme prédéfinie dans les langages de programmation : entier naturel ou relatif, réel, booléen, caractère, chaîne.

Il existe également des types dont la structure est plus complexe (on parle de types *structurés*), tels que les listes par exemple. Ces types sont construits par l'utilisateur, à l'aide de *constructeurs de types*. Ainsi, lorsque nous écrivions :

```
   variable maListe : liste d'entiers relatifs
```

nous utilisions le *constructeur* `liste`, appliqué au type de base (élémentaire) `entier relatif`.

Nous allons présenter dans ce chapitre les principaux constructeurs de types, que l'on retrouve dans la plupart des langages de programmation. Ces constructeurs vont nous permettre de définir des *types utilisateur* qui seront alors utilisables comme s'ils étaient prédéfinis. Ces types seront définis dans une rubrique `types` de la façon suivante :

```
   types
        TMonType : <definition du type>
   variables
        v1, v2 : TMonType
```

On préfixera systématiquement les types utilisateur par un « `T` ».

Un type utilisateur sera défini à l'aide d'un constructeur spécifique et d'un ou plusieurs *types de base* (types déjà définis, élémentaires ou structurés). À chaque constructeur de type correspond un ensemble d'opérations *primitives*, utilisables sur les objets de type ainsi construit. Cependant, il se peut que dans certains langages de programmation, certaines primitives, voire certains constructeurs, n'existent pas. Il faudra dans ce cas les « simuler ». Simuler un constructeur, c'est représenter l'objet d'une façon équivalente à l'aide des constructeurs disponibles (il faudra dans ce cas simuler également l'ensemble de toutes les primitives...). Simuler une primitive, c'est réaliser celle-ci sous forme d'une action ou d'une fonction.

Les constructeurs que nous présentons ici sont toutefois très répandus dans les langages de programmation et il ne sera que très rarement nécessaire de les simuler.

2.1. Les intervalles

Le constructeur `intervalle` permet de réduire l'ensemble des valeurs possibles d'un type scalaire (entier ou caractère) et est noté à l'aide du symbole « .. ». On écrira par exemple :

```
types
        TMois = 1..12           # mois = entier compris entre 1 et 12
        TReponse = 'a'..'f'     # réponse 'a', 'b', 'c', 'd', 'e' ou 'f'
```

Les primitives utilisables sur les objets de type intervalle sont identiques à celles définies sur le type de base.

2.2. Les énumérations

Le constructeur `énumération` permet de définir en extension l'ensemble des valeurs possibles :

```
types
        TJour = { lun, mar, mer, jeu, ven, sam, dim }
        TCouleur = { trèfle, carreau coeur, pique }
```

Les valeurs ainsi définies sont ordonnées (selon l'ordre de leur définition). On dispose alors des primitives Successeur et Prédécesseur qui permettent de récupérer la valeur suivante ou précédente, et des opérateurs de comparaison : =, ≠, <, >, ≤ et ≥. Un type énuméré est un type scalaire, il est donc possible d'en parcourir un intervalle à l'aide de la boucle `pour` :

```
variables   j1, j2 : TJours

   j1 ← lun
   j2 ← Successeur(j1)         # j2 reçoit mar

   si ( j1 < ven )
   alors ...

   pour j1 de lun à ven
   faire ...
```

2.3. Les agrégats

Le constructeur `agrégat` permet de regrouper des objets de nature distincte pour constituer des objets plus complexes. Une date est par exemple composée de trois informations, le jour le mois et l'année. On écrira alors :

```
types
        TDate = agrégat {  jour : 1..31
                           mois : 1..12
                           année : entier }
```

Les composants de l'agrégat (ici jour, mois et année) sont appelés *champs*. On accède aux champs d'une variable de type agrégat en utilisant la « notation pointée » :

```
variables   date : TDate
...
   date.jour ← 14
   date.mois ← 2
   date.année ← 2011

   Afficher (2*date.mois + 6*date.année)
```

Aucune primitive particulière n'est associée aux objets de types agrégat. Leur intérêt sera essentiellement d'être « repris » par d'autres constructeurs (tableaux, liste ou fichiers par exemple).

2.4. Les tableaux

Un tableau est une succession d'un nombre donné de « cases » pouvant contenir des objets de même type. Chaque case est repérée par un *indice*, la première case ayant pour indice 0. Voici par exemple un tableau de 8 chaînes de caractères, dont seules les 6 premières sont pour l'instant utilisées :

	0	1	2	3	4	5	6	7
tab :	'alain'	'lucie'	'anne'	'jean'	'léa'	'tony'		

Une telle structure se déclare de la façon suivante :

```
   constante     NB = 8
   type          TTableauNbChaines = tableau de NB chaînes

   variables     tab : TTableauNbChaines
                 nbElem : entier
```

Notons ici l'utilisation d'une constante `NB` pour donner la taille du tableau (ce qui permet, en cas de nécessité d'augmenter cette taille, de ne modifier que cette ligne dans notre algorithme), et de la variable `nbElem` qui permet de dire combien de cases du tableau `tab` sont effectivement utilisées (`nbElem` vaut 6 dans notre exemple).

L'objet contenu dans la case d'indice `i` du tableau `tab` est noté `tab[i]`. On peut ainsi écrire par exemple :

```
tab[2] ← 'antoine'

pour i de 0 à nbElem – 1
faire Afficher ( tab[i] )
fin_si
```

Le constructeur tableau est présent dans de nombreux langages de programmation. On trouve cependant souvent (c'est le cas en Python) un constructeur liste ayant des primitives plus puissantes. Un tableau est en effet une structure *figée*, dont la taille maximale est fixée a priori, alors que la taille d'une liste est non bornée a priori. Par ailleurs, pour insérer des valeurs dans un tableau, il est nécessaire de décaler les valeurs déjà présentes (et de modifier la variable « nombre d'éléments »), alors que les listes disposent de primitives d'insertion.

Voici un exemple d'insertion d'un élément dans un tableau :

insertion de la valeur 'loïc' dans la case 3 :

	0	1	2	3	4	5	6	7
avant :	'alain'	'lucie'	'anne'	'jean'	'léa'	'tony'		

nbElem : 6

	0	1	2	3	4	5	6	7
après :	'alain'	'lucie'	'anne'	'loïc'	'jean'	'léa'	'tony'	

nbElem : 7

Il y a deux principales catégories d'algorithmes sur les tableaux : ceux qui nécessitent un parcours séquentiel du tableau (éventuellement partiel), et « les autres »…

Les algorithmes de parcours sont basés sur l'un des deux « squelettes » suivants, à adapter selon le problème considéré :

```
<parcours d'un tableau tab à nbElem éléments en totalité>
  ...
  pour i de 0 à nbElem – 1
  faire
      <traiter tab[i]>
  fin_pour
```

```
<parcours partiel d'un tableau tab à nbElem éléments>
  ...
  i ← 0
  encore ← Vrai
  tantque ( (i < nbElem-1) et encore )
  faire
      si ( <fin du parcours> )
      alors encore ← Faux
      sinon
          <traiter tab[i]
          i ← i + 1
      fin_si
  fin_pour
```

La fonction suivante permet de déterminer la plus petite chaîne (ordre alphabétique) contenue dans un tableau de type `TTableauNbChaines` (si le tableau est « vide », on retourne une chaîne vide…). Elle nécessite donc un parcours du tableau en totalité (avec une adaptation) :

```
Fonction PlusPetit ( tab : TTableauNbChaines, nb : entier ) : chaîne
   # cette fonction retourne la plus petite chaîne contenue dans le
   # tableau tab, et la chaîne vide si nb = 0 (tableau vide)
variables   i, posMinimum : entiers naturels
début
   si ( nb = 0 )
   alors     # tableau vide
        Retourner ('')
   sinon     # initialisation
        posMinimum ← 0
             # boucle de traitement
        pour i de 1 à nb - 1
        faire     # on met à jour la position du minimum si nécessaire
             si ( tab[i] < tab[posMinimum] )
             alors posMinimum ← i
             fin_si
        fin_pour
             # retour du résultat
        Retourner (tab[posMinimum])
   fin_si
fin
```

La fonction suivante permet de dire si un tableau de type `TTableauNbChaines` est ou non trié par ordre alphabétique croissant. Le parcours s'arrête dès qu'une « rupture » est détectée.

```
Fonction EstTrié ( tab : TTableauNbChaines, nb : entier ) : booléen
   # cette fonction détermine si le tableau tab est trié ou non par
   # ordre croissant
variables   i : entier naturel
            trié : booléen
début
        # initialisation
   trié ← Vrai
   i ← 1
        # boucle de parcours
   tantque (i < nbElem)
   faire
             # si tab[i] et tab[i-1] non ordonnés, le tableau n'est pas
             # trié,sinon on avance
        si ( tab[i] < tab[i-1] )
        alors trié ← Faux
        sinon i ← i + 1
        fin_si
   fin_tantque
   Retourner (trié)
fin
```

On peut également, puisqu'il s'agit d'une fonction, interrompre le parcours en retournant `Faux` dès qu'une rupture est détectée (on a alors plus besoin de variable booléenne) :

```
Fonction EstTrié ( tab : TTableauNbChaines, nb : entier ) : booléen
   # cette fonction détermine si le tableau tab est trié ou non par
   # ordre croissant
variables   i : entier naturel
début
        # initialisation
```

```
  i ← 1
      # boucle de parcours
  tantque (i < nbElem)
  faire
          # si tab[i] et tab[i-1] non ordonnés, réponse = faux,
          # sinon on avance
      si ( tab[i] < tab[i-1] )
      alors Retourner(Faux)
      sinon i ← i + 1
      fin_si
  fin_tantque
      # on a franchi le tant que : le tableau est trié...
  Retourner (Vrai)
fin
```

La fonction suivante permet de rechercher la position d'un élément dans un tableau, et retourne `-1` lorsque l'élément est absent. La recherche s'effectue de façon séquentielle, et fonctionne donc que le tableau soit trié ou non.

```
Fonction Position (  tab : TTableauNbChaines, nb : entier
                     elem : Chaîne ) : entier
  # cette fonction détermine la position de elem dans tab et retourne
  # la valeur -1 si elem n'est pas présent
variables   i : entier naturel
début
      # initialisation
  i ← 1
      # boucle de parcours
  tantque (i < nbElem)
  faire
          # si tab[i] = elem, on retourne la position i
          # sinon on avance
      si ( tab[i] = elem )
      alors Retourner(i)
      sinon i ← i + 1
      fin_si
  fin_tantque
      # on a franchi le tant que : elem n'est pas présent...
  Retourner (-1)
fin
```

Lorsque le tableau est trié, il est plus efficace (en temps) d'effectuer une recherche dichotomique, dont le principe est le suivant. On utilise deux indices, `deb` et `fin`, initialisés à `0` et `nbElem-1` (les limites du tableau). On compare l'élément du milieu, d'indice `(deb + fin) div 2`, avec l'élément recherché. S'ils sont égaux, on a trouvé, si l'élément recherché est plus petit, il ne peut se trouver que dans la moitié gauche du tableau, et s'il est plus grand, dans la moitié droite. On modifie les indices `deb` et `fin` en conséquence, et on itère ce processus jusqu'à trouver l'élément ou jusqu'à ce que les deux indices `deb` et `fin` « se croisent » (ce qui signifie que l'élément n'est pas présent).

La fonction correspondante est la suivante :

```
Fonction PositionDichotomie (  tab : TTableauNbChaines, nb : entier
                               elem : Chaîne ) : entier
  # cette fonction détermine la position de elem dans tab en effectuant
  # une recherche dichotomique et retourne la valeur -1 si elem n'est
  # pas présent
variables   deb, fin : entiers
début
      # initialisation
  deb ← 0
  fin ← nbElem - 1
```

```
        # boucle de recherche
    tantque (deb <= fin)
    faire
        selon que
        tab[(deb + fin) div 2] = elem :      # on a trouvé
            Retourner ((deb + fin) div 2)
        tab[(deb + fin) div 2] > elem :      # on part à gauche
            fin ← ((deb + fin) div 2) - 1
        tab[(deb + fin) div 2] < elem :      # on part à droite
            deb ← ((deb + fin) div 2) + 1
        fin_selon

    fin_tantque

        # on a franchi le tant que : elem n'est pas présent...
    Retourner (-1)

fin
```

Il existe naturellement de nombreux algorithmes sur les tableaux, notamment les algorithmes permettant de modifier le contenu d'un tableau par ajout ou suppression d'éléments, qui nécessitent d'effectuer des décalages d'éléments.

Le document *Algorithmes de tri* présente différents algorithmes permettant de trier un tableau.

2.5. Les fichiers texte

Il s'agit ici de fichiers que l'on peut créer ou relire à l'aide d'un éditeur de texte : ce sont donc des fichiers de caractères, mais organisés en "lignes". Ils s'utilisent comme un écran (en sortie) ou un clavier (en entrée) : on peut ainsi y lire ou y écrire des valeurs de type quelconque. Ces fichiers sont utiles pour conserver des résultats afin éventuellement de les réutiliser ou de les imprimer. En entrée, ils permettent de « préparer à l'avance » des réponses clavier. Ils se déclarent ainsi :

```
variable    fic : fichier texte ;
```

Pour utiliser un fichier texte, il est nécessaire que celui-ci soit « ouvert » selon un *mode d'ouverture* précisant la façon dont on accédera aux informations qu'il contient. Après utilisation, un fichier doit être « fermé ». Le format de ces primitives est le suivant :

```
    OuvrirFichier ( fichier, nomExterne, modeOuverture )
    FermerFichier ( fichier )
```

Le Nom_externe correspond au nom du fichier sur le disque. Les modes d'ouverture principaux sont :

- création : crée un nouveau fichier ; s'il existait déjà, l'ancien contenu est écrasé,
- rajout : permet de « rallonger » un fichier existant,
- lecture : permet de récupérer les enregistrements contenus dans un fichier.

Il est possible de rajouter un enregistrement dans un fichier en utilisant la primitive `EcrireFichier` et de récupérer un enregistrement en utilisant la primitive `LireFichier`. Ces primitives ont le format suivant :

```
    EcrireFichier ( fichier, <Liste d'expressions> )
    LireFichier ( fichier, <liste de variables)
```

Ces deux primitives fonctionnent exactement comme s'il s'agissait d'une écriture à l'écran (`Afficher`) ou d'une lecture clavier (`Entrer`).

Pour savoir si une opération s'est correctement déroulée, on utilise la primitive `EtatFichier (fichier)` qui retourne les valeurs `FdF` (lorsqu'une opération de lecture « lit » la marque de fin de fichier), `Echec` ou `Succès` (après toute opération).

2.6. Autres types de fichiers

Il existe d'autres types de fichiers, dont le contenu est homogène (tous les éléments contenus dans le fichier ont la même structure, souvent des agrégats) : les fichiers séquentiels (l'accès aux éléments est

séquentiel), les fichiers relatifs (avec accès direct par numéro d'ordre) et les fichiers indexés (avec accès direct par clé). Ces structures dépasse le cadre de ce document et nous n'en parlerons donc pas.

Chapitre 3. Les types abstraits

3.1. La notion de type abstrait

La conception d'un algorithme complexe se fait toujours en plusieurs étapes correspondant à des affinements successifs (analyse descendante). Les premières versions d'un algorithme (concernant un problème complexe) sont autant que possible indépendantes de toute *implémentation* (réalisation effective). En particulier, la représentation des données n'est pas encore définitivement fixée.

A ce premier stade, les données sont considérées de manière abstraite. On se donne :

- une notation pour les décrire,
- un ensemble d'opérations que l'on peut leur appliquer (les primitives),
- les propriétés de ces opérations (permettant de définir leur sémantique).

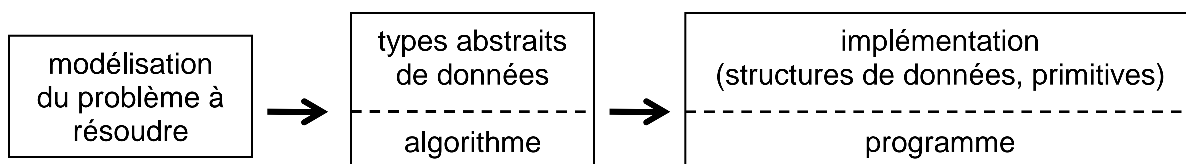
On parle dans ce cas de *type abstrait de données*. L'algorithme va alors être conçu en utilisant les opérations définies sur le type abstrait. Pour obtenir une version effective de cet algorithme il est ensuite nécessaire d'implémenter le type abstrait si celui-ci n'existe pas déjà tel quel dans le langage de programmation choisi, c'est-à-dire :

- en donner une représentation à l'aide des structures de données usuelles (et disponibles),
- traduire les différentes primitives sous forme d'actions ou de fonctions agissant sur cette représentation.

Il est généralement possible de proposer des implémentations différentes pour un même type abstrait de données : en particulier, on peut proposer diverses représentations d'un objet, ce qui entraîne évidemment diverses implémentations des primitives de l'objet. Selon l'implémentation choisie, on obtiendra ainsi différentes réalisations de notre algorithme, conduisant éventuellement à des algorithmes de complexités différentes.

On voit ici l'importance de ce que l'on appelle « indépendance d'un algorithme vis à vis de l'implémentation des données » : on peut substituer une implémentation efficace (parfois complexe) à une implémentation peu efficace mais rapidement obtenue (utilisée en phase de mise au point) sans pour autant modifier l'algorithme considéré. La notion de type abstrait est alors une composante essentielle de cette indépendance.

Le processus de résolution d'un problème est ainsi décrit par le schéma suivant :



L'utilisation des types abstraits est nécessaire lorsque :

- on utilise des objets d'une nature très spécifique, n'existant dans aucun langage de programmation, nécessitant ainsi d'être « traduits » à l'aide de structures disponibles,
- on utilise des objets d'un type bien connu mais qui n'est malheureusement pas offert par le langage de programmation utilisé (par exemple des entiers de plusieurs centaines de chiffres).

Les types de données que nous avons vus jusqu'ici, de même que les différents constructeurs que nous avons présentés, sont en fait eux-mêmes des types abstraits. Leur implémentation peut en effet différer d'un langage à l'autre sans que cela ait une influence sur la façon de les manipuler. Les types de données que nous allons maintenant présenter ne sont présents que dans les langages les plus récents. Pour cette raison, le terme « type abstrait de données » leur a longtemps été réservé car leur utilisation nécessitait une réelle implémentation. L'implémentation d'un type abstrait de données est une opération importante et doit être bien maîtrisée.

Les principaux types abstraits de données sont généralement regroupés de la façon suivante :

- les structures séquentielles : en particulier les listes ainsi que deux cas particuliers de celles-ci : les piles et les files,
- les structures arborescentes : arbres binaires et arbres dessinés généraux,
- les structures relationnelles que sont les graphes,
- les structures à accès par clé que sont les tables (internes ou externes).

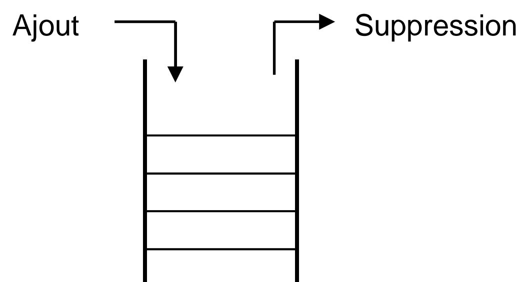
Dans cette partie, nous nous intéressons uniquement aux trois premières catégories.

3.2. Piles et files

La forme la plus commune d'organisation de données devant être traitée séquentiellement est la liste linéaire. Une liste est évolutive : on peut rajouter ou supprimer des informations n'importe où au sein d'une liste. Si l'on restreint les possibilités d'ajout et de suppression dans une liste, on obtient deux cas particuliers que sont les piles et les files, dont l'étude est justifiée par l'importance du rôle que jouent ces deux structures en Informatique.

3.2.1. Les piles

Une pile est une structure séquentielle pour laquelle les ajouts et les suppressions ne peuvent concerner que le dernier élément, appelé sommet de pile. Graphiquement, une telle structure se représente donc ainsi :



Cette structure est donc utile chaque fois que l'on souhaite stocker séquentiellement des objets pour les récupérer dans l'ordre inverse où ils ont été stockés. C'est le cas notamment en Informatique pour le traitement des appels d'actions (gestion des adresses de retour).

Une pile d'objets de type `TInfo` se déclarera de la façon suivante :

```
Type    TPile = pile de TInfo
Variable
        p : TPile
```

Pour manipuler un objet de type `TPile`, on utilisera les primitives suivantes :

```
Action CréerPile ( S p : TPile )
Fonction PileVide ( p : TPile ) : Booléen
Fonction ValeurSommet ( p : TPile ) : TInfo
Action Empiler ( ES p : TPile ; E elem : TInfo )
Action Dépiler ( ES p : TPile )
```

L'action `CréerPile` permet d'initialiser une pile à vide. La valeur (ou contenu) d'une pile est initialement indéterminée. Cette action doit donc être appelée avant toute manipulation d'une pile. La fonction `PileVide` permet de savoir si une pile est vide ou non. La fonction `ValeurSommet` permet de récupérer la valeur de l'élément se trouvant en sommet de pile (cet élément reste dans la pile). L'action `Empiler` permet de rajouter un élément dans une pile. Ce nouvel élément devient sommet de pile. L'action `Dépiler` supprime de la pile l'élément se trouvant en sommet de pile.

Les primitives `PileVide`, `ValeurSommet`, `Empiler` et `Dépiler` ne sont pas définies sur une pile dont la valeur est *indéterminée*. Les primitives `ValeurSommet` et `Dépiler` ne sont pas définies sur une pile *vide*.

Cette structure de pile est utilisée pour la manipulation d'expressions arithmétiques (en particulier par certains interpréteurs). Elle permet notamment de convertir une expression usuelle en notation postfixée, ou d'évaluer une expression postfixée.

Le principe de la notation postfixée est le suivant : les opérandes d'un opérateur binaires sont donnés avant l'opérateur lui-même. Ainsi, l'expression `3 + 28` sera notée `3 28 +`. Ainsi, lorsqu'un opérateur apparait, nous connaissons déjà la valeur de ses opérandes et pouvons donc évaluer l'expression au fur et à mesure.

Cette notation a de plus l'avantage de ne plus nécessiter l'utilisation de parenthèses. Ainsi par l'exemple, l'expression

```
6 * ( 12 + 5 * ( 4 - 3 * 5 ) + 6 * ( 11 + 3 ) )
```

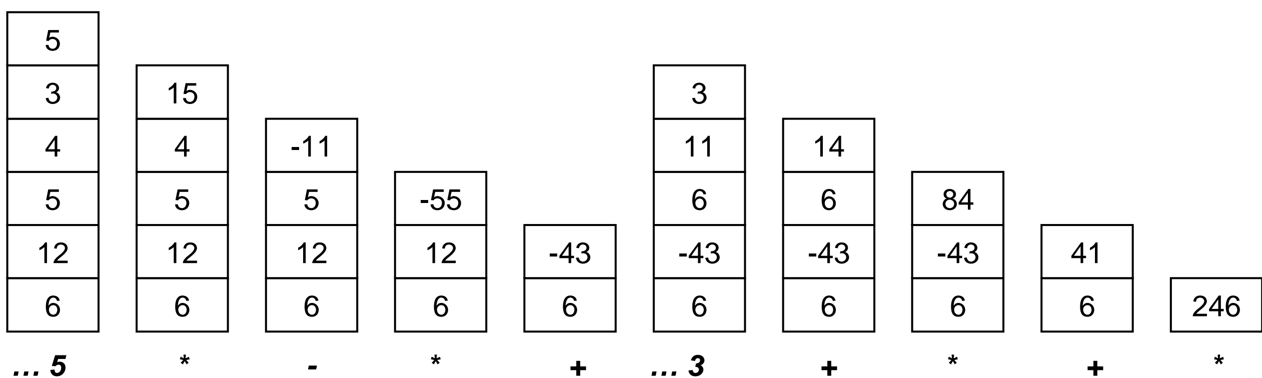
sera notée

```
6  12  5  4  3  5  *  -  *  +  6  11  3  +  *  +  *
```

De façon informelle, l'algorithme d'évaluation d'une expression postfixée (en supposant que l'expression est syntaxiquement correcte) est le suivant :

```
variable    p : pile d'entiers
…
   CréerPile ( p )
   <lire élément de l'expression>
   tantque <lecture réussie>          # échec lecture lorsque l'expression
                                      # est terminée
   faire
         si <element de type nombre>
         alors Empiler ( p, element )
         sinon
               operande2 ← ValeurSommet ( p )
               Dépiler ( p )
               operande1 ← ValeurSommet ( p )
               Dépiler ( p )
               resultat ← <operande1 element operande2>
               Empiler ( p , resultat )
         fin_si
   fin_tantque
   resultat ← ValeurSommet ( p )
```

Sur l'expression précédente, l'évolution de la pile est la suivante (le dernier élément d'expression lu est indiqué sous la pile, le résultat de l'évaluation est 246) :

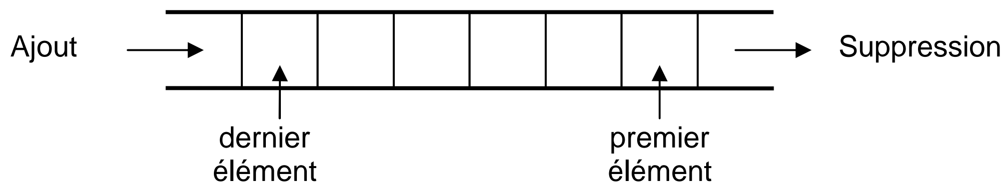


3.2.2. Les files

Cette notion correspond à la notion bien connue de *file d'attente*. Elle est également appelée structure FIFO (pour *First-In First-Out*). Il s'agit d'une structure que l'on rencontre fréquemment en Informatique. Pour gérer l'accès à une ressource partageable entre plusieurs utilisateurs, on utilise une file des processus en attente de cette ressource (par exemple, liste des fichiers en attente d'impression). Dans une telle structure, les ajouts se font « à la fin » et les suppressions « au début » : le premier arrivé est le premier servi.

Cette structure est donc utile chaque fois que l'on souhaite stocker séquentiellement des objets pour les récupérer dans l'ordre où ils ont été stockés.

On représente généralement une file de la façon suivante :



Une file de `TInfo` se déclarera de la façon suivante :

```
Type    TFile = file de TInfo
Variable
        f : TFile
```

Pour manipuler un objet de type `TFile`, on utilisera les primitives suivantes :

```
Action CréerFile ( S f : TFile )
Fonction FileVide ( f : TFile ) : Booléen
Fonction ValeurPremier ( f :TFile ) : TInfo
Action Enfiler ( ES f : TFile ; E elem : TInfo )
Action Défiler ( ES f : TFile )
```

L'action `CréerFile` permet d'initialiser une file à vide. La valeur (ou contenu) d'une file est initialement indéterminée. Cette action doit donc être appelée avant toute manipulation d'une file. La fonction `FileVide` permet de savoir si une file est vide ou non. La fonction `ValeurPremier` permet de récupérer la valeur du premier élément d'une file (cet élément reste dans la file). L'action `Enfiler` permet de rajouter un élément dans une file. Ce nouvel élément se retrouve en dernière position de la file. L'action `Défiler` supprime de la file l'élément se trouvant en première position.

Les primitives `FileVide`, `ValeurPremier`, `Enfiler` et `Défiler` ne sont pas définies sur une file dont la valeur est *indéterminée*. Les primitives `ValeurPremier` et `Défiler` ne sont pas définies sur une file *vide*.

3.2.3. Implémentation des piles et des files

Les piles et les files ont longtemps été implémentées en utilisant le constructeur tableau. L'inconvénient majeur de ce type d'implémentation est que les tableaux sont des structures *statiques*, dont la taillée est bornée à priori, ce qui n'est pas le cas des piles et des files.

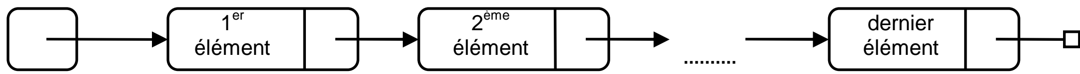
On rencontre maintenant dans la plupart des langages des structures dynamiques, de type liste, qu'il est possible d'utiliser pour représenter une pile ou une file. Il suffit alors de se limiter à des ajouts en fin de liste et des suppressions en fin de liste (pour les piles) ou en début de liste (pour les files).

3.3. Les listes

Une liste est une suite ordonnée d'éléments, ce qui signifie qu'à chaque élément est associé un certain rang dans la liste (1er, 2ème, etc.). La structure de liste est plus riche que les structures pile ou file car nous pouvons ajouter ou supprimer un élément se trouvant en n'importe quelle position dans la liste. Une telle opération de mise à jour se fera sans modifier (ni déplacer) les autres éléments de la liste : on

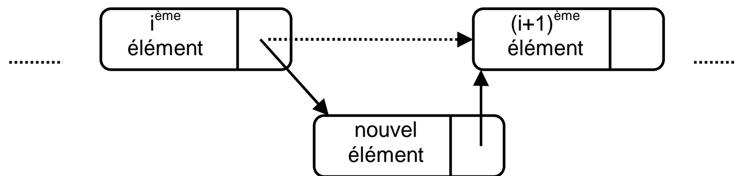
évite ainsi l'inconvénient majeur des mises à jour dans un tableau, qui nécessitent d'effectuer des décalages.

Graphiquement, nous représenterons une liste de la façon suivante :

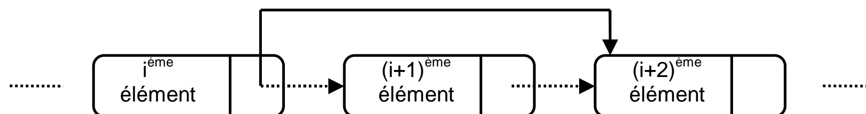


La « flèche » représente ici un certain mécanisme permettant de passer d'un élément de la liste à l'élément suivant. Attention, le dernier élément de la liste n'a bien sûr pas de suivant.

L'insertion d'un élément « au milieu » d'une liste peut alors se faire de la façon suivante :



Et la suppression d'un élément :



Nous voyons qu'ainsi les opérations de mise à jour se traduisent par une modification du mécanisme des flèches, et que les éléments ne sont pas déplacés. Il est fondamental de se convaincre que la position physique des éléments est absolument indépendante de leur ordre logique au sein d'une liste : on peut représenter une liste différente en modifiant simplement les flèches, tout en laissant les éléments physiquement à leur place.

Comme précédemment, on se fixe le type `TInfo` des objets composant la liste. Une liste séquentielle de `TInfo` se déclarera alors de la façon suivante :

```
Type    TListe = liste de TInfo
Variable
        l : TListe
```

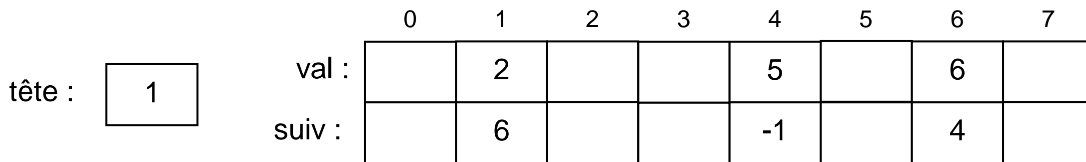
Les primitives généralement utilisées sur les listes sont les suivantes :

- création d'une liste (initialisée à vide) : `CréerListe ( l )`
- accès à l'élément de rang i : `l[i]`
- nombre d'éléments d'une liste : `nb ← NombreElements ( l )`
- concaténation de listes : `l3 ← l1 + l2`
- ajout d'un élément au rang i : `AjouterElement (l, elem, i)`
- suppression de l'élément de rang i : `SupprimerElement (l, i)`

Notons que ces primitives peuvent être différentes dans les langages proposant le type liste comme type prédéfini (c'est notamment le cas du langage Python).

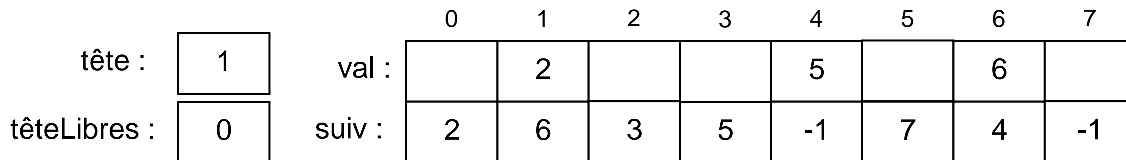
Lorsque le type liste n'est pas proposé par un langage de programmation, il est possible d'en proposer une implémentation « simple » à l'aide d'un tableau (mais dans ce cas, les listes auront un nombre maximum d'éléments à ne pas dépasser). L'idée est de représenter la notion de flèche à l'aide d'indices précisant la position dans le tableau de l'élément suivant, la valeur particulière `-1` signifiant « pas de suivant ». La liste est alors donnée par l'indice de son premier élément (`-1` si la liste est vide).

La liste [2, 6, 5] serait alors représentée ainsi :



	0	1	2	3	4	5	6	7
val :		2			5		6	
suiv :		6			-1		4	

Pour pouvoir gérer les « cases libres » de ce tableau, la technique la plus simple consiste à les regrouper dans une seconde liste, dite « liste des cases libres ». On obtient alors le schéma suivant :



	0	1	2	3	4	5	6	7
val :		2			5		6	
suiv :	2	6	3	5	-1	7	4	-1

pour lequel la liste des cases libres est [0, 2, 3, 5, 7].

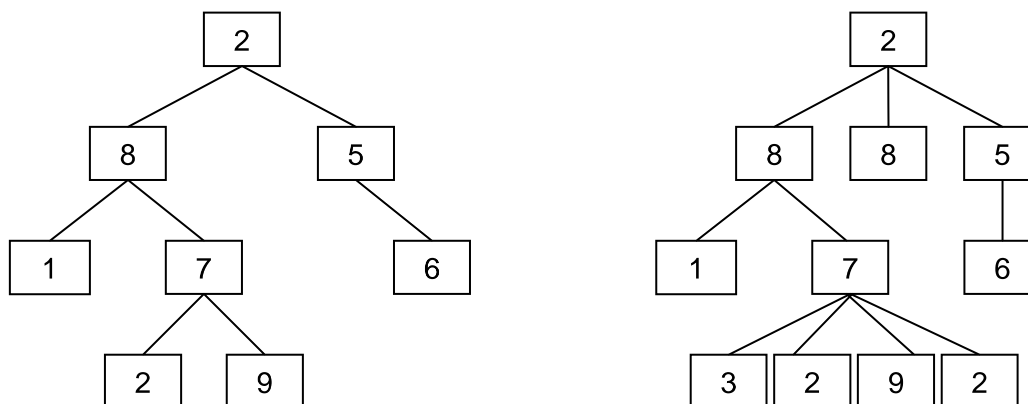
La déclaration du type `liste de TInfo` est alors :

```
constante    MAX = 20  # par exemple
type         TCellule = agrégat {
                  val : TInfo
                  suiv : entier }
             TListe = agrégat {
                  tête : entier
                  têteLibres : entier
                  tab : tableau de MAX TCellule }
```

3.4. Les arbres

La notion d'arbre est une extension naturelle de la notion de liste. Alors que dans une liste, chaque élément (sauf le dernier) possède un unique successeur, les éléments d'un arbre (appelés *nœuds* ou *sommets*) peuvent avoir un nombre quelconque de successeurs (appelés *fils*), chaque élément n'ayant toutefois qu'un seul prédécesseur (appelé *père*), à l'exception d'un élément (appelé *racine*) qui est le « point d'entrée » de la structure. Un élément n'ayant aucun successeur est une *feuille*, un élément non-feuille est un *sommet interne*.

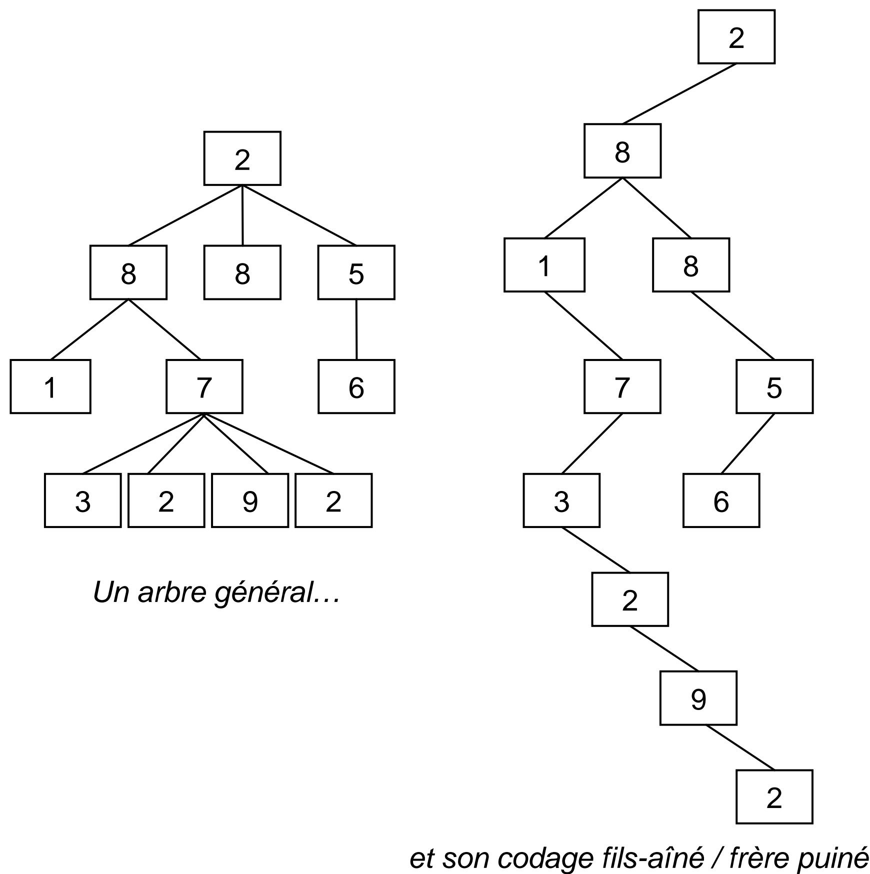
Un arbre sera dit *binaire* si ses sommets ont au maximum deux fils. Dans ce cas, on distinguera les fils *gauche* et *droit* (l'arbre est donc *dessiné*). Voici par exemple un arbre binaire et un arbre quelconque (non binaire) :



Dans l'arbre binaire de gauche, le sommet de valeur 5 n'a qu'un seul fils, de valeur 6, et il s'agit de son fils droit.

En pratique, on s'intéresse essentiellement aux arbres binaires. On peut en effet remarquer que tout arbre quelconque peut être « codé » par un arbre binaire, en associant à chaque sommet son *fils aîné*

comme fils gauche d'une part, et son *frère puiné* comme fils droit d'autre part. L'arbre général précédent est alors codé ainsi :



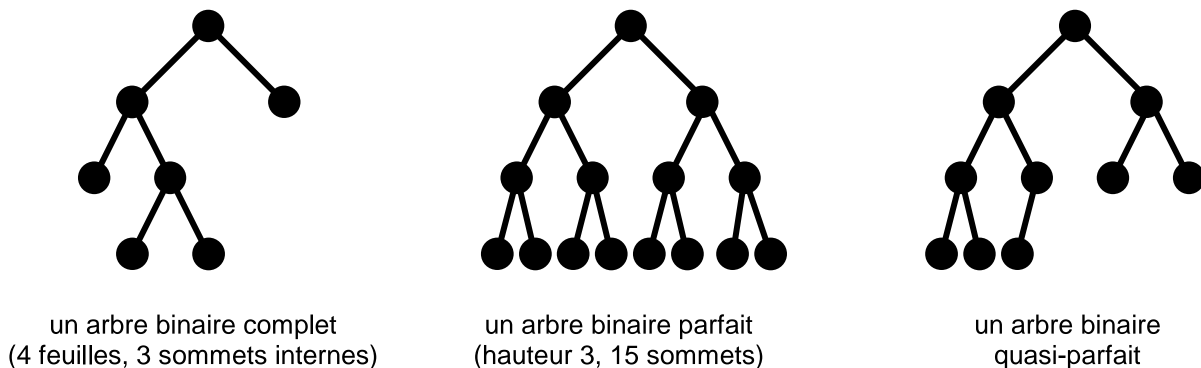
Un arbre général…

et son codage fils-aîné / frère puiné

Parmi les arbres binaires, on distinguera notamment :

- les arbres binaires *complets*, dont tous les sommets ont exactement 0 ou 2 fils (on peut vérifier que dans ce cas il y a une feuille de plus que de sommets internes),
- les arbres binaires *parfaits*, qui sont complets et dont tous les niveaux sont « remplis » (on peut vérifier que dans ce cas, un arbre complet de hauteur `n` contient exactement 2^{n+1} sommets, la hauteur correspondant à la distance maximale entre un sommet et la racine),
- les arbres binaires *quasi-parfaits*, dont tous les niveaux sont remplis, à l'exception éventuelle du dernier qui est « rempli sur la partie gauche ».

La figure suivante illustre ces trois catégories :



un arbre binaire complet (4 feuilles, 3 sommets internes) | un arbre binaire parfait (hauteur 3, 15 sommets) | un arbre binaire quasi-parfait

Un arbre binaire de `TInfo` se déclarera alors ainsi :

```
Type   TArbreBinaire = arbre binaire de TInfo
```

```
Variable
       a : TArbreBinaire
```

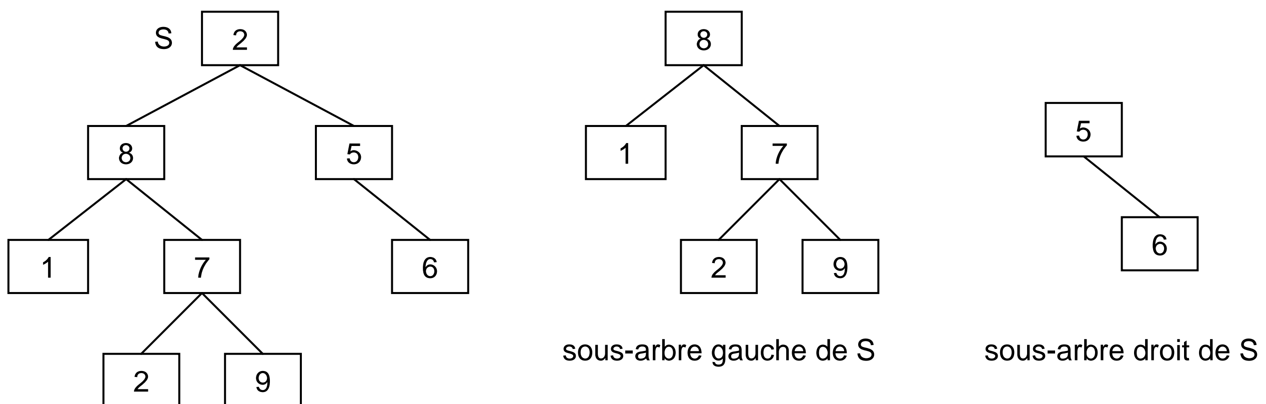
Les primitives utilisées sur les arbres dépendent généralement du problème traité. On retrouve en principe les primitives suivantes :

- initialisation d'un arbre à vide (aucun sommet),
- accès à la racine de l'arbre,
- tester si un arbre est vide,
- tester si un sommet possède un fils gauche ou droit,
- accès aux fils gauche ou droit d'un sommet,
- accès au père d'un sommet.

Concernant les primitives de mise à jour, la situation est plus délicate. On peut aisément ajouter une feuille ou supprimer une feuille en préservant la structure de l'arbre. Par contre, pour insérer ou détruire un sommet interne, il n'y a pas de façon naturelle de « recoller les morceaux »[5]. La façon de procéder dépendra alors du problème considéré, c'est-à-dire de la *sémantique* associée à l'arbre. L'algorithme du tri par tas[6], ou la gestion des arbres binaires de recherche (section suivante) illustrent cette particularité.

3.4.1. Arbres binaires de recherche

Soit A un arbre et S un sommet de A. Le *sous-arbre* de A enraciné en S est l'arbre ayant pour racine S et composé de l'ensemble de la descendance de S dans A. Le *sous-arbre gauche* (resp. *sous-arbre droit*) d'un sommet S est le sous-arbre enraciné en son fils gauche (resp. en son fils droit). Si S ne possède pas de tel fils, le sous-arbre correspondant est vide.

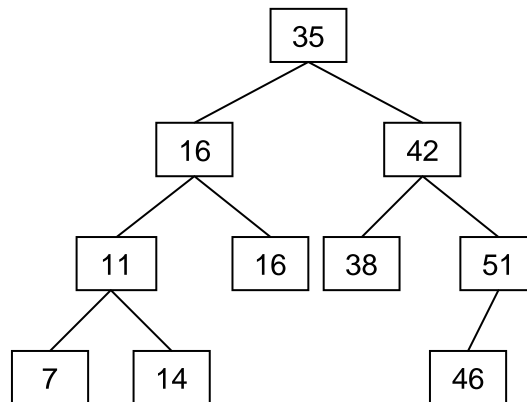


Un arbre binaire de recherche (*binary search tree*) est un arbre binaire dont les sommets sont organisés (ou ordonnés) de façon telle que la valeur de tout sommet est *supérieure* aux valeurs des sommets de son sous-arbre gauche et *inférieure ou égale* aux valeurs des sommets de son sous-arbre droit.

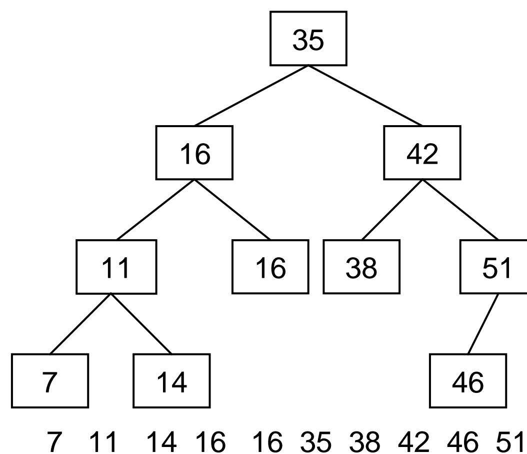
Voici par exemple un arbre binaire de recherche :

[5] La situation est différente dans le cas de la définition *inductive* d'un arbre, mais cela sort du cadre de ce document…

[6] Voir le document *Algorithmes de tri*.

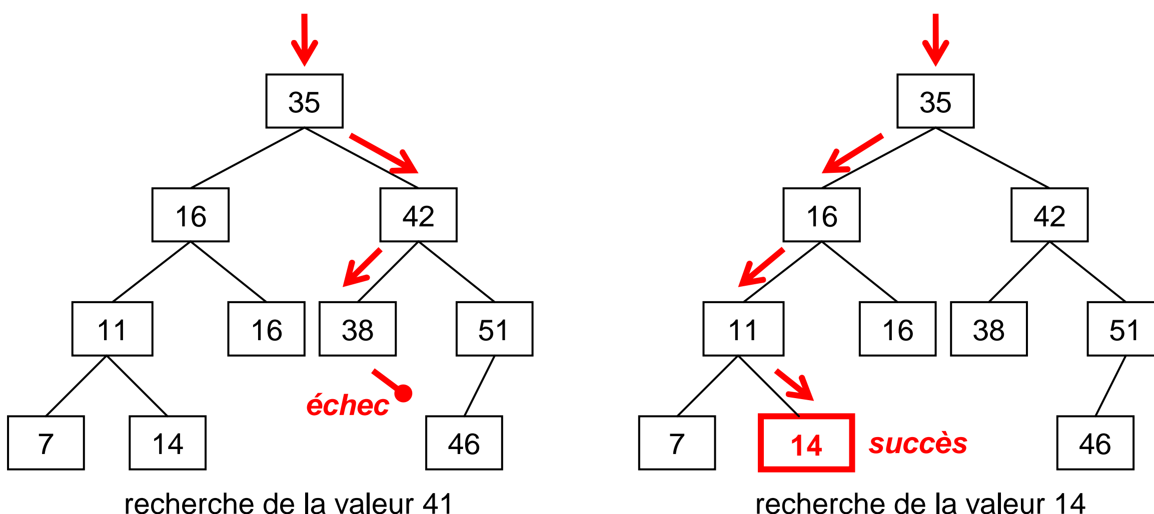


L'un des intérêts d'une telle structure est qu'elle permet d'obtenir aisément la liste *triée* des informations qu'elle contient. Il suffit en effet de parcourir l'arbre selon un ordre précis (dit *parcours infixe*) : le sous-arbre gauche de la racine, puis la racine, et enfin le sous-arbre droit de la racine (et ce, récursivement). Cela revient en fait à « aplatir » l'arbre :



On peut en effet voir un tel arbre comme le « schéma d'organisation » d'un tableau de valeurs selon la méthode du pivot (voir l'algorithme de tri rapide dans le document *Algorithmes de Tri*).

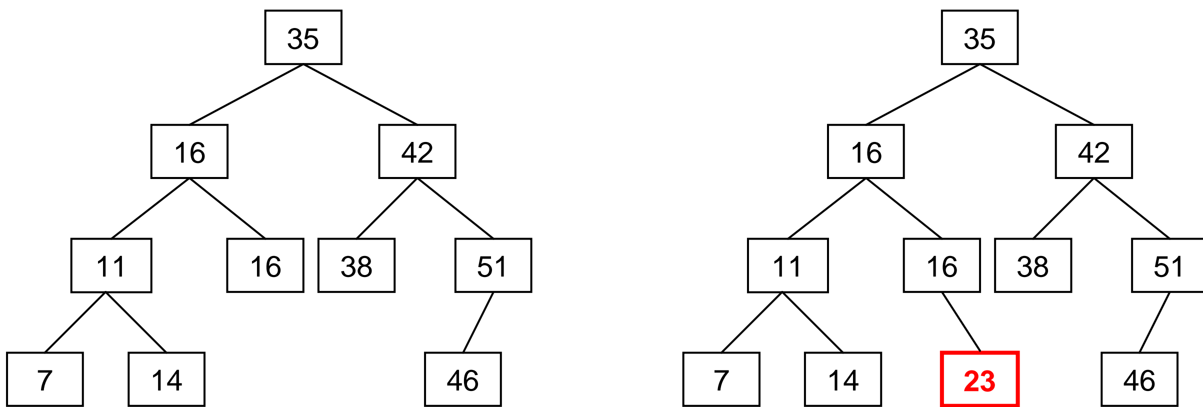
Cette structure permet également de rechercher de façon efficace une information : pour chercher une valeur VAL dans un tel arbre, il suffit de partir de la racine et, si la valeur recherchée n'est pas celle du sommet visité, de descendre du côté gauche ou du côté droit selon le résultat de la comparaison entre VAL et la valeur du sommet visité (il n'est ainsi pas nécessaire de parcourir la totalité des sommets) :



recherche de la valeur 41 recherche de la valeur 14

Intéressons-nous maintenant aux opérations de mise à jour, ajout et suppression.

La façon la plus simple de rajouter un élément consiste à créer une nouvelle feuille « bien placée » qui contiendra cet élément (la position d'insertion peut être déterminée à l'aide d'un algorithme similaire à l'algorithme de recherche vu précédemment). La notion d'ordre définie sur les sommets des arbres binaires de recherche permet d'assurer qu'il n'existe qu'une seule possibilité pour créer une telle feuille :

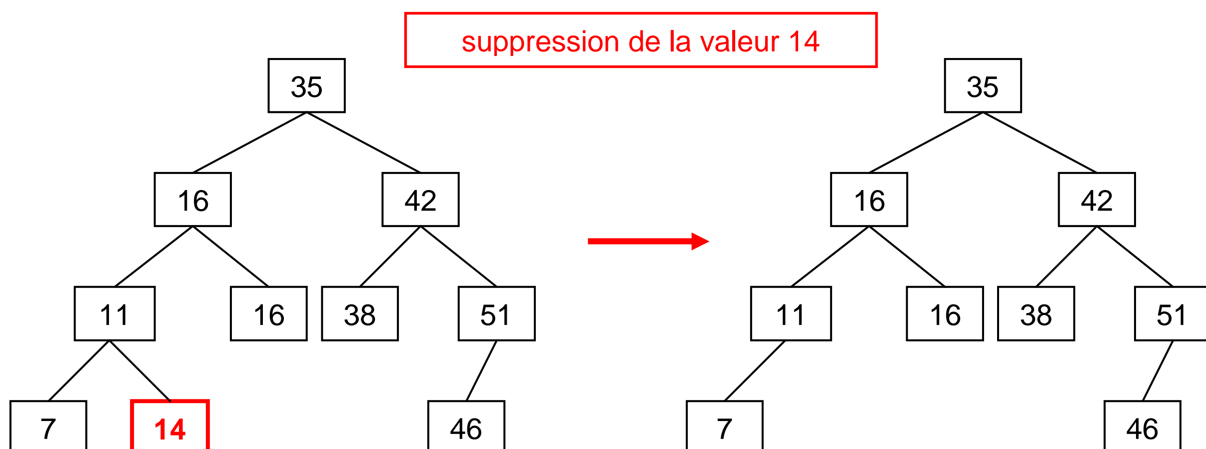


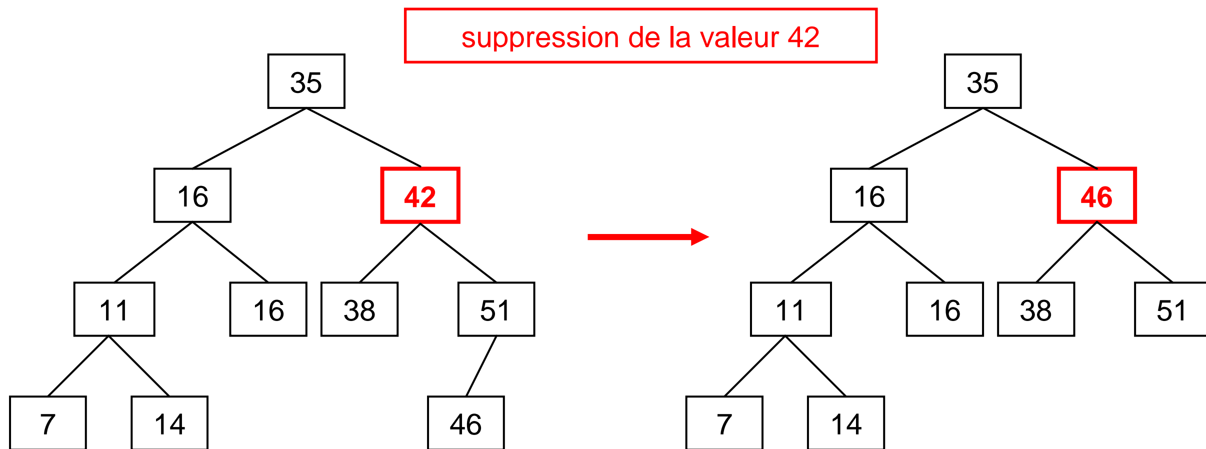
insertion de la valeur 23

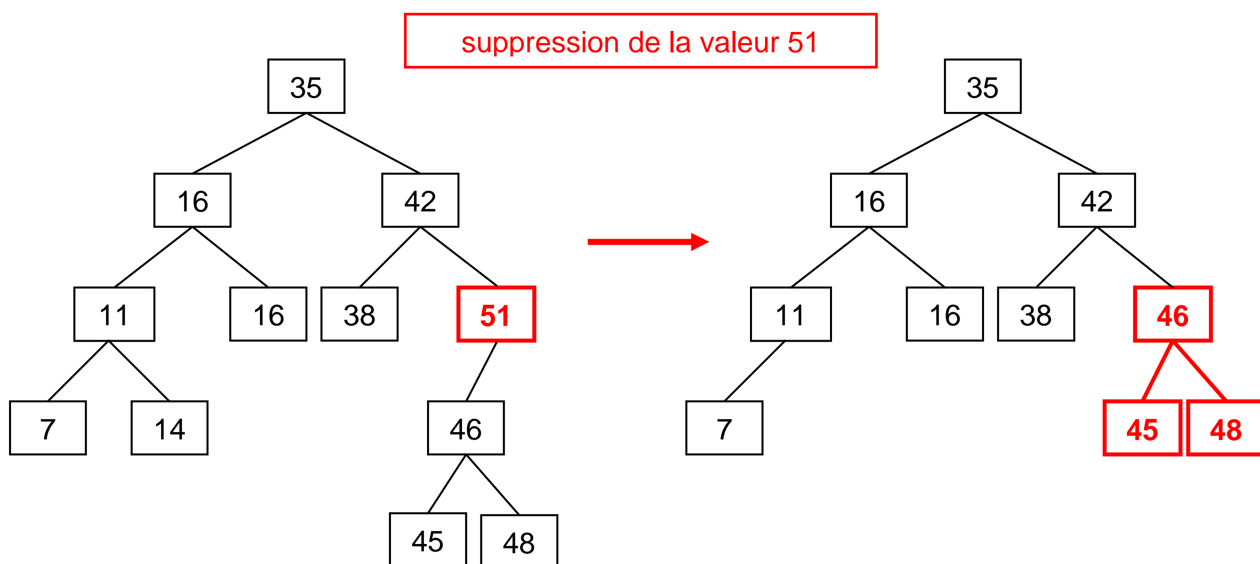
L'opération de suppression est plus délicate… Il va être en effet nécessaire de « remplir » le vide laissé par l'élément supprimé. Pour cela, on procède de la façon suivante :

- si l'élément à supprimer est une feuille, on supprime la feuille,
- sinon, si l'élément possède un sous-arbre droit, on remplace la valeur supprimée par la plus petite valeur de son sous-arbre droit (ce qui éventuellement va créer un nouveau vide, qui sera comblé de façon similaire…),
- sinon (dans ce cas l'élément possède un sous-arbre gauche et n'a pas de fils droit), on fait « remonter » le sous-arbre gauche à la place de l'élément supprimé.

Ces différents cas de figure sont illustrés ci-dessous :







Quelle est la complexité de ces opérations ? Si l'arbre est « bien équilibré », sa hauteur est de l'ordre de log n s'il contient n valeurs. Ainsi, on peut montrer que toutes ces opérations sont en $\Theta(\log n)$.

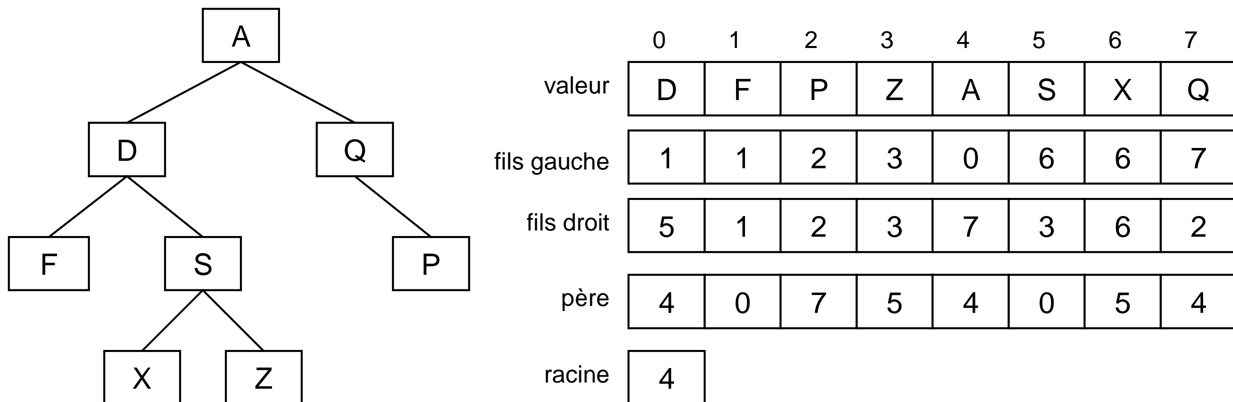
Par contre, si l'arbre n'est pas équilibré, la complexité peut retomber en $\Theta(n)$. Il existe cependant des techniques de mise à jour, plus sophistiquées que celles que nous avons présentées, qui permettent d'assurer que l'arbre est à tout instant bien équilibré.

3.4.2. Implémentation des arbres

Nous avons vu précédemment que la définition d'un jeu de primitives permettant de manipuler les arbres dépendait du problème considéré. De la même façon, l'implémentation de la structure d'arbre dépendra fortement du jeu de primitives souhaité, car elle devra permettre d'implémenter aisément, et efficacement, les primitives en question.

Il est possible d'implémenter la structure d'arbre binaire à l'aide de tableaux, en s'inspirant de ce qui a été fait pour les listes. Il suffit en effet de considérer que chaque élément possède deux « suivants », son fils gauche et son fils droit, et un prédécesseur, son père. Par convention, la racine est son propre père, et un élément n'ayant pas de fils gauche (resp. droit) est son propre fils gauche (resp. droit). Il est également nécessaire de stocker l'indice de la case contenant la racine.

Voici un exemple de représentation :

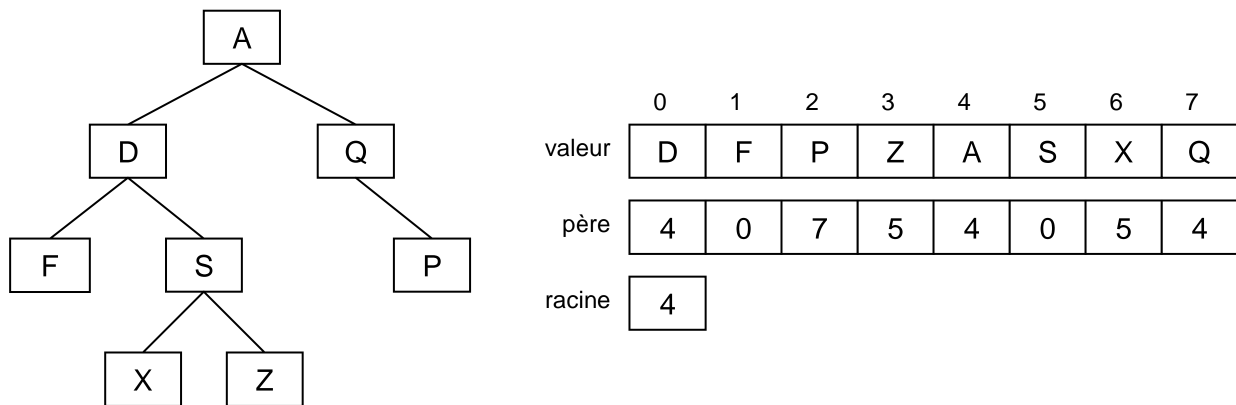


	0	1	2	3	4	5	6	7
valeur	D	F	P	Z	A	S	X	Q
fils gauche	1	1	2	3	0	6	6	7
fils droit	5	1	2	3	7	3	6	2
père	4	0	7	5	4	0	5	4
racine	4							

Notons que si la structure de l'arbre évolue par ajout et/ou suppression de sommets, il sera nécessaire, comme dans le cas des listes, de gérer la liste des cases libres…

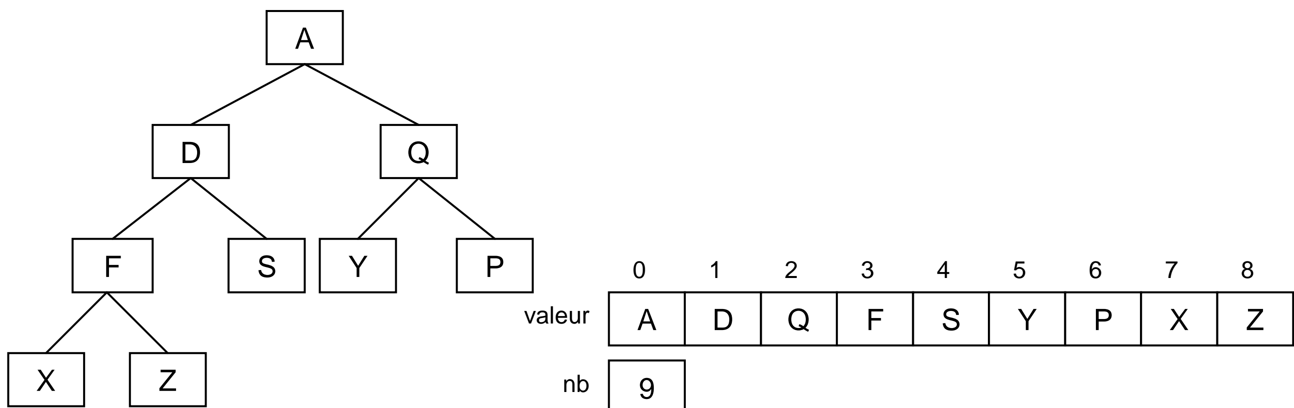
Une autre représentation, plus synthétique mais moins performante pour certaines primitives, consiste à associer à chaque sommet simplement l'indice de son père. On perd dans ce cas la distinction gauche / droite pour les fils mais, dans certains cas (voir Algorithme de Huffman, section suivante), il n'est pas nécessaire de coder cette distinction.

Cette représentation est illustrée ainsi :



	0	1	2	3	4	5	6	7
valeur	D	F	P	Z	A	S	X	Q
père	4	0	7	5	4	0	5	4
racine	4							

Dans le cas des arbres quasi-parfaits, il existe une représentation simple, consistant à stocker les valeurs des éléments niveau par niveau, de gauche à droite. On stocke également le nombre de sommets de l'arbre (pour connaître les nombre d'éléments effectivement utilisés dans le tableau) :



	0	1	2	3	4	5	6	7	8
valeur	A	D	Q	F	S	Y	P	X	Z
nb	9								

On s'aperçoit aisément que :

- la racine est en position 0,

- le sommet en position `i` a pour fils gauche le sommet en position `2i+1` et pour fils droit le sommet en position `2i+2` (à condition que ces indices soient strictement inférieurs à `nb`…),
- le sommet en position `i > 0` a pour père le sommet en position `(i-1) div 2`.

Cette représentation permet donc de parcourir l'arbre de haut en bas ou de bas en haut.

Notons enfin que cette représentation peut permettre de représenter un arbre binaire quelconque. Il suffit en effet de rajouter des sommets « fictifs » de façon à le rendre quasi-parfait (on utilisera une valeur particulière pour ces sommets fictifs afin de les distinguer des « vrais » sommets de l'arbre). Cette représentation est cependant coûteuse si l'arbre est vraiment « éloigné » (en nombre de sommets) de la structure d'un arbre quasi-parfait…

3.4.3. Algorithme de Huffman

L'algorithme de David Albert Hufman est un algorithme simple de compression de données sans perte d'information, développé dans sa thèse de doctorat au MIT en 1953. L'idée de base est la suivante :

- déterminer la fréquence de chaque caractère (octet) de la donnée à comprimer (fichier texte par exemple),
- construire à partir de ces fréquences un codage performant, c'est-à-dire pour lequel les caractères les plus fréquents ont un code « court »,
- coder la donnée (fichier), et y inclure le code construit.

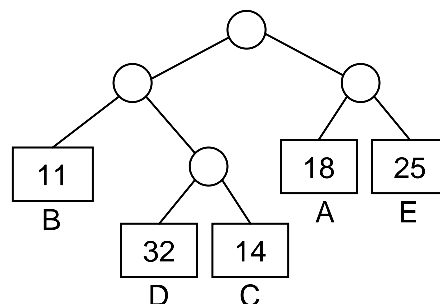
Pour construire un code optimal, nous allons « ranger » les fréquences dans les feuilles d'un arbre binaire complet, de façon telle que les plus grandes valeurs soient le plus près possible de la racine...

À chaque feuille f_i de l'arbre, on associe sa valeur (fréquence) v_i et sa distance à la racine (nombre d'arêtes à traverser) d_i. La « qualité » d'un tel arbre peut alors être mesurée par la somme Q des produits v_i x d_i.

Considérons par exemple les fréquences suivantes :

caractère	A	B	C	D	E
fréquence	18	11	14	32	25

Ces fréquences peuvent par exemple être organisées ainsi :



Q = 2x(11+18+25) + 3x(32+14) = 246

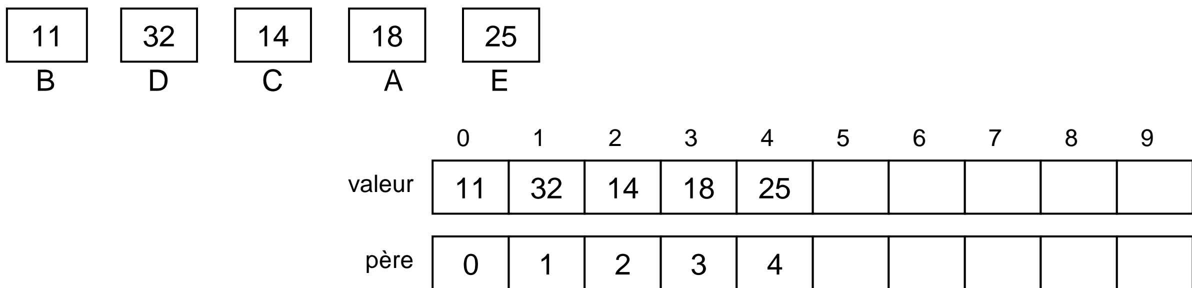
L'algorithme de Huffman va nous permettre de construire un tel arbre *optimal*. Le principe de cet algorithme est le suivant :

- Au départ, on considère que toutes les valeurs sont isolées, n'ayant encore aucun père dans l'arbre qui va être construit. L'arbre va alors se construire de bas en haut…

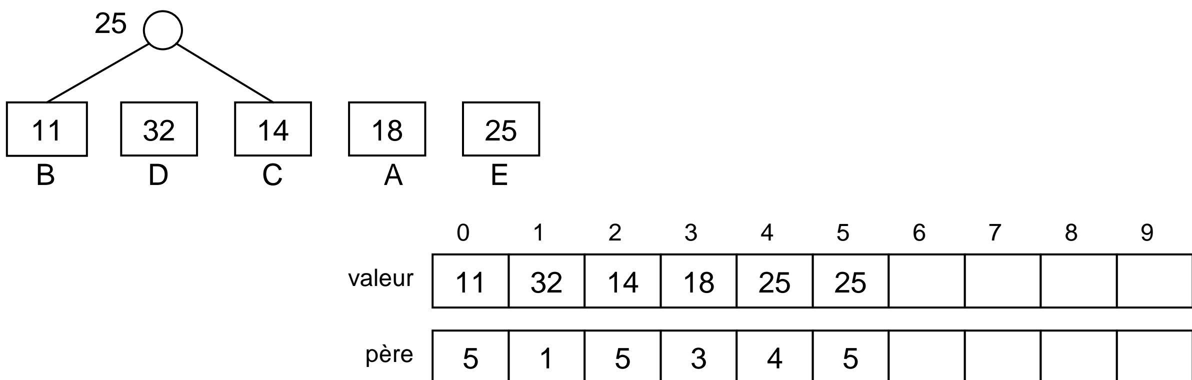
- À chaque étape, on cherche les deux plus petites valeurs (fréquences) parmi celles n'ayant pas encore de père, et on les associe à un nouveau sommet qui va devenir leur père. Ce nouveau sommet aura pour valeur la somme des deux valeurs de ses fils.

Cet algorithme est parfaitement adapté à la représentation par père d'un arbre binaire, vue précédemment. Sur notre exemple, cet algorithme fonctionnera de la façon suivante (on recherche les minima uniquement sur les valeurs qui sont leur propre père…) :

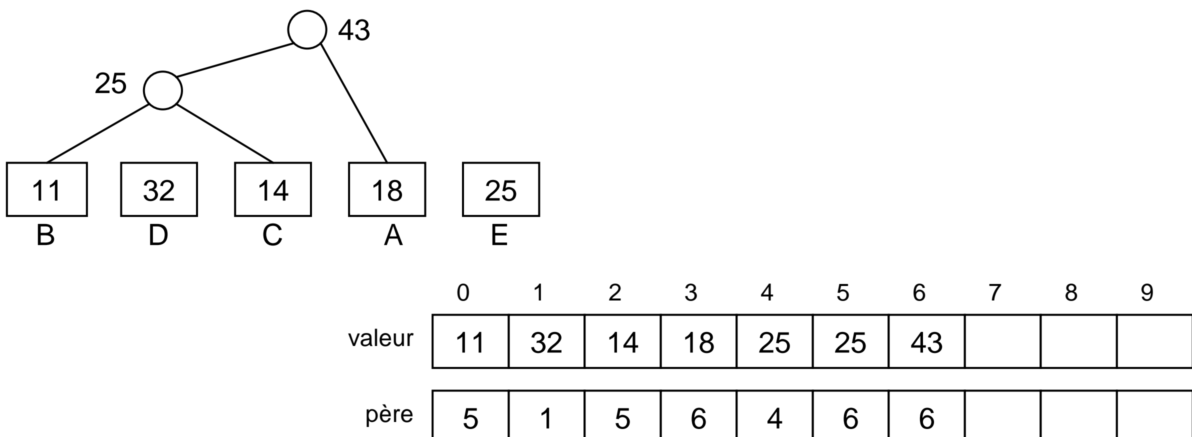
Étape 0 : initialisation (tous les sommets sont leur propre père).



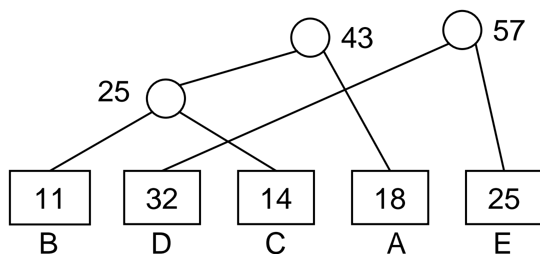
Étape 1 : apparition du sommet 25



Étape 2 : apparition du sommet 43

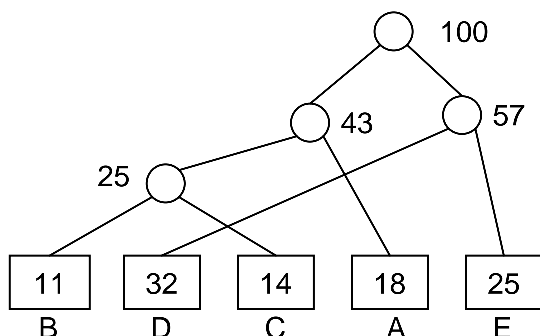


Étape 3 : apparition du sommet 57



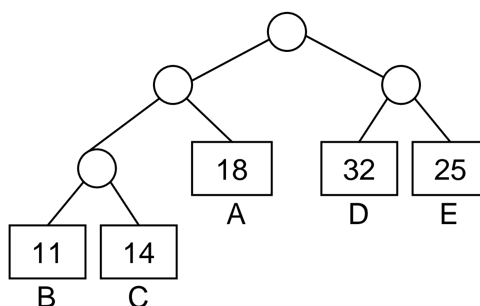
	0	1	2	3	4	5	6	7	8	9
valeur	11	32	14	18	25	25	43	57		
père	5	7	5	6	7	6	6	7		

Étape 4 : fin de la construction…



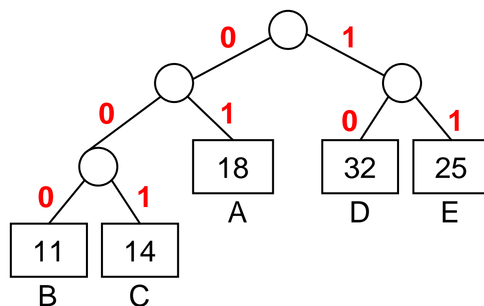
	0	1	2	3	4	5	6	7	8
valeur	11	32	14	18	25	25	43	57	100
père	5	7	5	6	7	6	8	8	8

Redessiné correctement, l'arbre obtenu est le suivant, et on se convainc aisément qu'il s'agit d'une solution optimale.



$$Q = 2x(18+32+25) + 3x(11+14) = 225$$

Maintenant, cet arbre va nous permettre de définir un *code*. Pour cela, on va associer un 0 aux branches gauches et un 1 aux branches gauches de la façon suivante :

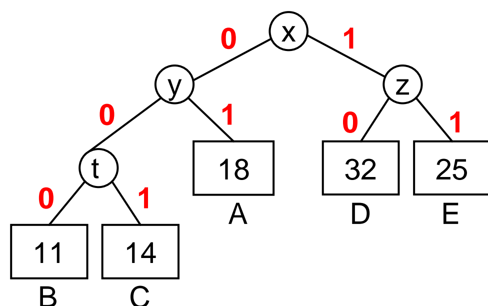


Maintenant, le code de chaque lettre « se lit » sur l'arbre, en suivant le chemin menant à sa fréquence. Ce code est donc le suivant :

lettre	A	B	C	D	E
code	01	000	001	10	11

Ainsi, le texte ACADBEDDA par exemple, sera codé par la séquence 01001011000011101001.

Pour décoder cette séquence, il suffit d'utiliser l'arbre : on part de la racine et on « suit » le chemin indiqué par les caractères de la séquence ; lorsqu'on atteint une feuille, on génère la lettre associée et on repart au niveau de la racine. Sur notre exemple, nous obtenons alors (nous avons nommé les sommets internes pour plus de clarté) :



0 1	0 0 1	0 1	1 0	0 0 0	1 1	1 0	1 0	0 1
x y	x y t	x y	x z	x y t	x z	x z	x z	x y
A	C	A	D	B	E	D	D	A

Maintenant, la version comprimée du fichier d'origine contiendra le code (c'est-à-dire l'arbre permettant de le retrouver, stocké à l'aide du tableau père), suivi de la séquence codée, ce qui permettra le décodage ultérieur (décompression).

Cet algorithme est particulièrement bien adapté aux fichiers dont les caractères ont des fréquences fortement distinctes. C'est le cas notamment des fichiers texte, car on n'utilise environ que 90 caractères sur les 256 disponibles.

On pourrait également définir un code « universel », pour la langue française par exemple, basé sur les fréquences habituellement rencontrées. Cela éviterait la nécessité de stocker le code, mais le taux de compression ne serait plus optimal (le code de Hufmann est directement calculé à partir des fréquences *réelles* des caractères au sein du fichier).

Chapitre 4. Complexité des algorithmes

La complexité d'un algorithme est une mesure permettant d'évaluer la performance de cet algorithme en fonction de la taille des données du problème à traiter. On distingue :

- la *complexité en temps*, qui mesure le temps nécessaire à l'exécution d'un algorithme,
- la *complexité en espace*, qui mesure la taille mémoire nécessaire à l'exécution d'un algorithme.

L'analyse de la complexité consiste à déterminer ces deux grandeurs, dans le but de *comparer* différents algorithmes résolvant le *même problème*. Cette analyse porte sur l'algorithme et doit être indépendante de toute implémentation de celui-ci.

La complexité en temps représente le nombre d'opérations élémentaires (opérations arithmétiques, comparaisons, affectations) nécessaires à l'exécution de l'algorithme sur une donnée en entrée de taille N. La complexité en espace représente l'espace mémoire (nombre d'octets) nécessaire à l'exécution de l'algorithme sur une donnée en entrée de taille N (on considère généralement la taille de la donnée elle-même, fonction de la représentation choisie, ainsi que la taille des données propres à l'algorithme).

Lorsqu'on réalise un algorithme pour résoudre un problème donné, il faut toujours s'assurer que sa complexité est « la meilleure possible » pour le problème en question.

Soit **A** un algorithme ; notons *coût(d)* la complexité (en temps ou en espace) de l'algorithme **A** sur la donnée *d*, et D_n l'ensemble des données de taille *n*. On s'intéresse alors aux quantités suivantes :

- la complexité dans le meilleur des cas : $Min(n) = \min \{ coût(d) \ / \ d \in D_n \}$
- la complexité dans le pire des cas : $Max(n) = \max \{ coût(d) \ / \ d \in D_n \}$
- la complexité en moyenne: $Moy(n) = \sum_{d \in Dn} P(d) \times cout(d)$,

  où *P(d)* représente la probabilité d'avoir la donnée *d* en entrée.

En particulier, si toutes les données sont équiprobables, on a :

$$Moy(n) = \frac{1}{|D_n|} \sum_{d \in Dn} cout(d) .$$

Lorsqu'on s'intéresse à la complexité d'un algorithme, on cherche avant tout à connaître la *rapidité de croissance* de cette fonction lorsque la taille des données augmente. Ainsi, une simple approximation de la fonction de complexité suffit : si pour une donnée de taille *n*, la complexité en temps et de *3n+7*, l'important est ici que cette fonction est *linéaire*. Intuitivement, si la taille des données est multipliée par deux, le temps nécessaire à l'exécution de l'algorithme sera également multiplié par deux (à peu de choses près).

4.1. Expression de la complexité d'un algorithme

On introduit les notations suivantes, qui permettent de comparer *les ordres de grandeur asymptotiques* des fonctions :

- $f = O(g)$, ssi il existe une constante C et un rang n_0, tels que $n>n_0 \Rightarrow f(n) \leq C.g(n)$ (f est dominée par g)
- $f = \Theta(g)$, ssi $f = O(g)$ et $g = O(f)$ (f et g sont asymptotiquement équivalentes)

en d'autres termes, il existe deux constantes C1 et C2 et un rang n_0 tels que

$$n < n_0 \Rightarrow C1.g(n) \leq f(n) \leq C2.g(n)$$

On a ainsi par exemple :

$$3n + 7 = \Theta(n) \qquad 5n^2 + 6n - 11 = \Theta(n^2)$$

On voit ici que dans le cas de fonctions polynomiales, seul le terme de plus grand poids nous intéresse.

Les ordres de complexité les plus fréquents sont les suivants :

- $\Theta(1)$: complexité constante (ex : somme des N premiers entiers… grâce à la formule associée),
- $\Theta(\log n)$: complexité logarithmique (ex : recherche dichotomique),
- $\Theta(n)$: complexité linéaire (ex : recherche séquentielle),
- $\Theta(n \log n)$: complexité subquadratique (ex : tri rapide),
- $\Theta(n^2)$: complexité quadratique (ex : tri simple),
- $\Theta(n^k)$: complexité polynomiale,
- $\Theta(2^n)$: complexité exponentielle (ex : tours de Hanoï).

Les tableaux suivants[7] permettent d'illustrer l'intérêt de la complexité. Le premier tableau donne une estimation du temps d'exécution d'un algorithme suivant sa complexité et la taille des données en entrée (en prenant comme unité de temps 1µs = temps d'exécution d'une opération). Le second tableau donne une estimation de la taille maximale du problème que l'on peut traiter en un temps donné.

Complexité Taille	**1**	**logN**	**N**	**NlogN**	**N^2**	**N^3**	**2^N**
10^2	1µs	6,6µs	0.1ms	0,66ms	10ms	1s	4×10^{16}a
10^3	1µs	9,9µs	1ms	9,9ms	1s	16,5mn	∞
10^4	1µs	13,3µs	10ms	0,13s	1,5mn	11,5j	∞
10^5	1µs	16,6µs	0,1s	1,64s	2,7h	31,7a	∞
10^6	1µs	19,9µs	1s	19,9s	11,5j	$31,7\times10^6$a	∞

Complexité Temps	**1**	**logN**	**N**	**NlogN**	**N^2**	**N^3**	**2^N**
1 s	∞	∞	10^6	$6,3\times10^4$	10^3	10^2	19
1 mn	∞	∞	6×10^7	$2,8\times10^6$	7×10^3	4×10^2	25
1 h	∞	∞	36×10^8	$1,3\times10^8$	6×10^4	15×10^2	31
1 jour	∞	∞	$8,6\times10^{10}$	$2,7\times10^9$	3×10^5	44×10^2	36

On a noté ∞ lorsque la valeur dépassait 10^{100}. Ces tableaux permettent de constater que certains algorithmes sont inutilisables.

En pratique, il est parfois possible d'améliorer la complexité en temps au détriment de la complexité en espace. On s'adapte alors à la situation considérée, selon que l'espace est « critique » ou non…

4.2. Détermination de la complexité d'un algorithme

La complexité en espace d'un algorithme est généralement facile à déterminer. Si l'algorithme ne manipule que des données statiques, il suffit en effet de considérer la taille des variables nécessaires à son exécution[8].

[7] Extraits de M.C. Gaudel, M. Soria, C. Froidevaux, *Types de données et Algorithmes*, Vol. II : *Recherche, Tri, Algorithmes sur les graphes*, INRIA, Coll. didactique.

[8] Si l'algorithme utilise l'allocation dynamique (à l'aide de pointeurs) ce calcul peut s'avérer plus délicat.

Pour déterminer la complexité en temps d'un algorithme il est nécessaire de caractériser les *opérations fondamentales*, c'est-à-dire les opérations qui influeront le plus sur la complexité (une affectation d'objet de grande taille est par exemple bien plus coûteuse qu'une incrémentation d'indice).

Ces opérations fondamentales peuvent être par exemple :

- dans le cas de la recherche d'un élément dans une liste : le nombre de comparaisons entre cet élément et les éléments de la liste,
- dans le cas de la recherche d'un élément dans un fichier : le nombre d'accès disque,
- dans le cas d'un tri : le nombre de comparaisons et le nombre de transferts d'éléments.

Remarquons qu'il est toujours possible de considérer que toutes les opérations sont fondamentales... Pour compter le nombre d'opérations fondamentales, il suffit ensuite d'appliquer quelques règles de calcul plus ou moins simples :

- Enchaînement séquentiel : les valeurs s'ajoutent, naturellement... Cela étant, si l'on s'intéresse à l'ordre de grandeur (notation Θ), il suffit de prendre l'ordre maximal.
- Structures conditionnelles : l'idéal serait bien sûr de savoir quelle branche de la structure l'algorithme va emprunter... C'est généralement impossible et on se contente de prendre un *majorant*, en considérant la branche la plus coûteuse... Si l'on dispose d'informations plus précises, il est possible d'en calculer la moyenne. En termes d'ordre de grandeur, on prendra le maximum des deux branches.
- Structures répétitives : il suffit bien sûr de multiplier le nombre d'opérations contenues dans le corps par le nombre d'exécutions de la structure ! Ce nombre est parfois difficile à déterminer... Là encore, on prendra souvent un majorant...
- Algorithmes récursifs : il s'agit certainement des algorithmes pour lesquels la complexité est la plus délicate à déterminer... On peut être amené à résoudre des équations de récurrence.

4.3. Quelques exemples de calcul de complexité

Le document *Algorithmes de tri* présente différents algorithmes de tris ainsi que leur complexité.