M11 : Algorithmique

Jean-François Berdjugin

M11 : Algorithmique

Jean-François Berdjugin

Publication date 2011

Table of Contents

1. Préambule 1
    1. Programme 1
    2. Évaluations 1
2. Cours 2
    1. Complexité 2
        1.1. Définition de l'ordre de grandeur asymptotique 2
        1.2. Exemples de complexités courantes 2
        1.3. Règles de simplification 3
    2. Structures de données 3
        2.1. Les listes simplement chaînées 3
        2.2. Les listes doublement chaînées 3
        2.3. Les files 4
        2.4. Les piles 4
        2.5. Les arbres binaires 5
        2.6. Les graphes 5
    3. Algorithme de tri 5
        3.1. Rappel sur les tableaux 5
            3.1.1. Déclaration 6
            3.1.2. Instanciation (ou construction) 6
            3.1.3. Accès aux données 6
        3.2. Généralités sur les tris 8
        3.3. Tri par insertion 8
        3.4. Tri par selection 8
        3.5. Tri à bulle 9
3. Exercices 10
    1. Cellule 10
        1.1. Présentation 10
        1.2. Codage 10
    2. Liste simplement chaînée en version itérative 10
        2.1. Présentation 11
        2.2. Codage 11
            2.2.1. Variable(s) d'instance(s) 11
            2.2.2. Constructeur sans paramètre 11
            2.2.3. Méthode estVide 11
            2.2.4. Méthode getPremier 12
            2.2.5. Méthode getDernier 12
            2.2.6. Méthode insereTete 12
            2.2.7. Méthode insereQueue 12
            2.2.8. Méthode suppressionTete 12
            2.2.9. Méthode suppressionQueue 12
            2.2.10. Exercices complémentaires 12
    3. Pile 12
        3.1. Présentation 12
        3.2. Codage 13
    4. Liste simplement chaînée en version récursive 13
        4.1. Rappel sur la récursivité 13
            4.1.1. Définition 13
            4.1.2. Exemples 13
        4.2. Exercice 16
            4.2.1. getTaille 16
            4.2.2. getVal 16
            4.2.3. setVal 16
            4.2.4. insertionPosition 17
            4.2.5. suppressionPosition 17
    5. Les arbres binaires. 17

5.1. Création de la classe ... 17
5.2. Les feuilles et les noeuds ... 18
5.3. La taille ... 18
5.4. La hauteur ... 18
5.5. Parcours en préordre ... 18
5.6. Parcours en postordre ... 18
A. Import et export de projet sous eclipse ... 19
1. Export ... 19
2. Import d'un projet dans un Workspace existant ... 19

List of Figures

2.1. Liste simplement chaînée 3
2.2. Liste doublement chaînée 3
2.3. File 4
2.4. Pile 4
2.5. Arbre binaire 5
2.6. Graphe 5
2.7. Déclaration d'un tableau 6
2.8. Construction d'un tableau 6
2.9. Accès aux éléments d'un tableau 7
2.10. Dépassement des bornes 7
2.11. Alias 8
3.1. Cellule 10
3.2. Diagramme de classe de Cellule 10
3.3. Liste simplement chaînée 11
3.4. Liste simplement chaînée réelle 11
3.5. Diagramme UML de la classe Liste 11
3.6. Diagramme de classe de Pile 13
3.7. Appel récursifs de la factorielle 14
3.8. Appel récursif de la somme des éléments d'un tableau d'entiers 15
3.9. Appel récursif de la somme des éléments d'un tableau d'entiers avec un index. 15
3.10. Liste récursive 16
3.11. Classe ArbreB 17
3.12. Exercice arbre binaire 17

List of Tables

2.1. Complexités courantes 2

List of Examples

3.1. Factorielle récursive 14
3.2. Somme récursive des éléments d'un tableau d'entiers 15

Chapter 1. Préambule

Un bon algorithme obéit à plusieurs critères. Il doit pouvoir être maintenu (facile à lire, à coder et à déboguer), il doit également être sûr. Nous allons maintenant ajouter une nouvelle contrainte, il doit-être rapide. La rapidité est vue au travers de la notion de complexité. Chaque algorithme a un coup en temps et en espace (occupation mémoire). Ce coup prend un sens particulier dans les systèmes embarqués où le processeur et la mémoire sont souvent moindre. Nous introduirons la notion de complexité temporelle et nous l'appliquerons aux structures de données.

Les structures de données sont des structures logiques destinées à contenir des données. Le choix de la bonne structure de données permet un traitement plus efficace et plus rapide. L'utilisation des bonnes structures données permet de baisser la complexité de l'application et ainsi à la rendre plus sûre.

1. Programme

Nous disposons de 12 heures soit 9,5 heures de cours et 1,5 heure de DS, nous allons étudier les listes simplement chaînées, les piles et les arbres binaires. Les listes et les arbres binaires seront implementés en utilisant une approche récursive. Les tableaux seront vus pendant le devoir maison et les algorithmes de tri en cours.

2. Évaluations

Il y aura deux évaluations : un devoir machine et un devoir maison.

Chapter 2. Cours

1. Complexité

Commençons par un exemple :

```
int s = 0;
int i;
for (i=0; i < n; i++)
    s=s+i;
```

Cet algorithme coûte en mémoire deux entiers, c'est son coût spatial, mais quel est son coût temporel ? Si l'on ne prend pas en compte les déclarations, que chaque opération d'écriture, de lecture, de comparaison ou de calcul coûte 1, nous avons un coup de

$$2 + 8n + 1. \tag{2.1}$$

Nous observons ainsi que le coût temporel de notre problème dépend de la taille de l'entrée (n) et des hypothèses de consommation de temps.

Voici un deuxième exemple :

```
int i = 0;
for (i=0; i <n; i++)
    if (a[i]>i)
       a[i] = a[i] -1;
```

Cette fois-ci le coût temporel de l'algorithme est plus difficile à calculer, c'est pourquoi nous allons nous intéresser au plus mauvais temps d'exécution. Il aussi possible d'avoir le meilleur temps d'exécution et le temps d'exécution moyen.

Nous avons

$$T_{min}(n) \leq T_{moyen}(n) \leq T_{max}(n). \tag{2.2}$$

Maintenant comment choisir le bon algorithme, est-il besoin du calcul du plus mauvais temps de calcul ($T_{max}(n)$) ou un ordre de grandeur peut-il suffire.

Par exemple, si nous avons $2 + 8n + 1$ et $3 + 4n^2 + 5$, le premier algorithme a un ordre de grandeur en n et le second en n^2.

1.1. Définition de l'ordre de grandeur asymptotique

L'ordre de grandeur asymptotique, la notation O (grand O) permet la comparaison entre les temps d'exécution. O est défini comme suit :

Soit f(n) et T(n) deux fonction de N vers R*+, T=O(f) (T est en grand O de f) ssi il existe c appartenent à R*+, il existe n_0 tq quelque soit $n > n_0$, $T(n) \leq c\ f(n)$ (2.3)

Donc $2 + 7n + 1$ est en $O(n)$ et $3 + 4\ n^2 + 5$ est en $O(n^2)$.*Pour être plus formel, il nous faudrait une autre notion le grand Θ qui permet de définir des équivalences, par exemple 2n est en $\Theta(n)$ mais n'est pas en $\Theta(n^2)$.*

1.2. Exemples de complexités courantes

Le tableau suivant contient les complexités courantes et les valeurs pour n =1000.

Table 2.1. Complexités courantes

O	Dénomination	Valeur pour n=1 000 000
O(1)	constant	
O(ln n)	logarithmique	13.81
O(n)	linéaire	1 000 000

O(n ln n)	n log	13 815 510
$O(n^2)$	quadratique	1 0000 000 000 000
$O(n^3)$	cube	1 0000 000 000 000 000 000
$O(2^n)$	exponentiel	...

1.3. Règles de simplification

Il est possible de simplifier le temps d'exécution pour les ordres de grandeur :

1. Les constantes n'ont pas d'importance, T(n) = n + c => O(n)

2. Les coefficients n'ont pas d'importance, T(n) =c n => O(n)

3. Dans un polynome, les termes d'ordre inférieurs n'ont pas d'importance, $T(n) = 2^n + n^{1000} = O(2^n)$.

2. Structures de données

Il est possible de classer les structures de données : les structures finies (constantes, variables, enregistrements, structures composées finies), les structures indexées, les structures récursives.

Voici quelques exemples de structures de données, les quatre premières sont qualifiées de linéaires, la troisième est qualifiée d'arborescente et enfin la dernière est qualifiée de relationnelle.

2.1. Les listes simplement chaînées

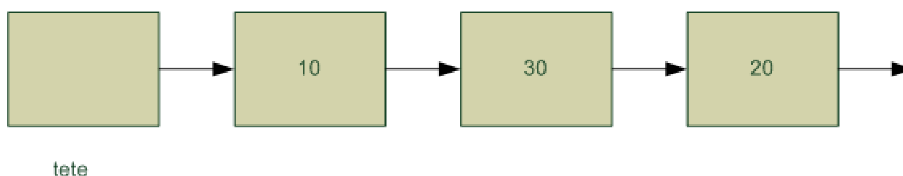
Les listes sont des structures séquentielles, elles peuvent-être implémentées sous forme de tableaux ou de sous forme chaînée.

Les opérations de base sont :

- l'insertion,
- la suppression,
- la recherche,
- la concaténation.

Les listes simplement chaînées sont constituées de maillons ou de cellules qui contiennent une valeur et une référence vers la cellule suivante.

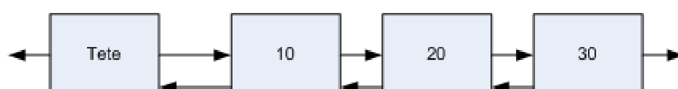
Figure 2.1. Liste simplement chaînée



2.2. Les listes doublement chaînées

La liste doublement chaînée est constituée de cellules ayant une référence vers la cellule suivante et la cellule précédente.

Figure 2.2. Liste doublement chaînée

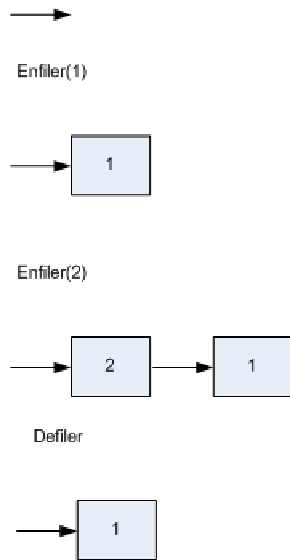


2.3. Les files

Les files possèdent deux opérations de base :

- enfiler,
- défiler.

Figure 2.3. File



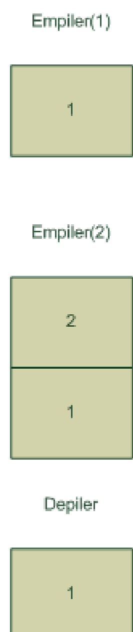
La file est qualifiée de FIFO (First In First Out). Il existe des variantes comme les files à priorité.

2.4. Les piles

La pile possède deux opérations de base :

- empiler,
- dépiler.

Figure 2.4. Pile



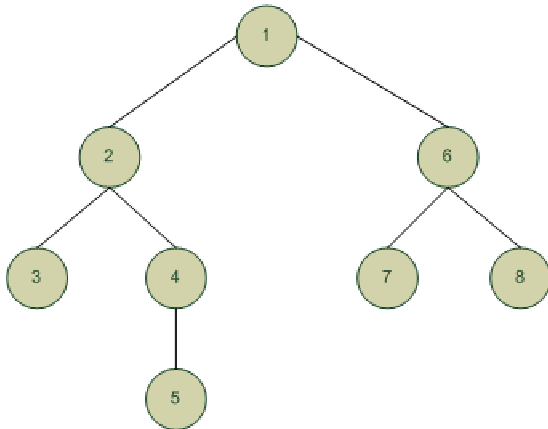
La pile est qualifiée de LIFO (Last In First Out).

2.5. Les arbres binaires

Les arbres sont constitués de noeuds, un noeud particulier est la racine. Les noeuds peuvent posséder des sous arbres.

Les arbres imposent une approche récursive.

Figure 2.5. Arbre binaire

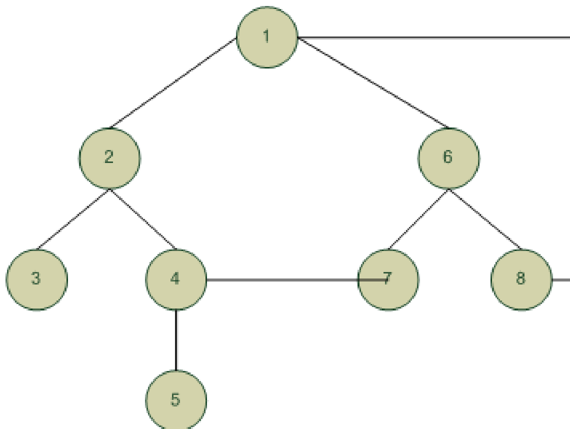


Les arbres binaires sont des sous-cas des arbres où les noeuds possèdent au plus deux fils (les fils sont les racines des sous-arbres).

2.6. Les graphes

Les graphes sont des structures où des cycles sont possibles.

Figure 2.6. Graphe



3. Algorithme de tri

Nous allons étudier des algorithme de tri sur les tableaux et calculer l'ordre de grandeur de leur plus mauvais temps d'exécution.

3.1. Rappel sur les tableaux

Les tableaux JAVA ™ correspondent à la représentation intuitive que vous vous faite d'un tableau ou d'un vecteur. Les éléments cases sont accessibles via un indice, dont la numérotation commence à 0. La première case d'un tableau est numérotée 0 est la dernière la longueur du tableau moins 1.

Nous allons apprendre comment déclarer un tableau, puis l'instancier et enfin le manipuler.

3.1.1. Déclaration

Pour déclarer un tableau les crochets*[] doivent être utilisés.*

typeTab[] tab; déclare *tab* comme étant une variable référencent un tableau de type *typeTab*. Tous les éléments de notre tableau sont de même type.

```
typeTab[] tab; //declaration de tab comme étant un tableau
               //de type typeTab
```

Figure 2.7. Déclaration d'un tableau

tab

3.1.2. Instanciation (ou construction)

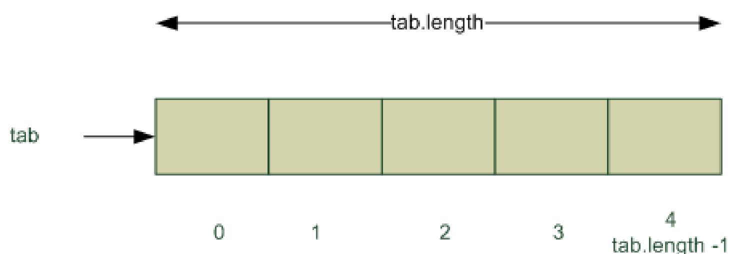
Après la déclaration notre tableau n'existe pas, nous ne disposons que de la référence (le nom), pour qu'il existe, il faut l'instancier (construire). Les tableaux ont une taille fixe (structure de donnée statique), celle taille doit être fixée lors de la construction. Le mot clef permettant la construction est new.

tab = new typeTab[taille]; permet de créé le tableau de type *typeTab*, de taille *taille* est accessible via la référence *tab*.

```
typeTab[] tab;

tab = new typeTab[taille]; //construction d'un tableau de taille : taille
                           // la première case est numérotée 0
                           // la dernière case est numérotée taille - 1
```

Figure 2.8. Construction d'un tableau



3.1.3. Accès aux données

Sous un même nom la référence, nous avons un ensemble de données accessible en utilisant un indice.

3.1.3.1. Lecture/Ecriture

La lecture et l'écriture sont réalisées en spécifiant entre crochet (*[]*) la position de la "case".

tab[i] permet d'accéder à la position *i* du tableau référence par *tab*.

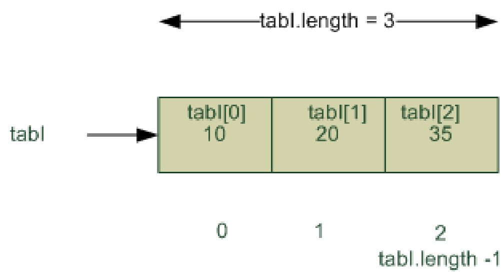
Voici un exemple d'écriture suivi d'un exemple de lecture sur un tableau d'entiers:

```
int[] tabI;
tabI = new int[3];

tabI[0]=10; //ecriture de la valeur 10 à la position 0
tabI[1]=20; //ecriture de la valeur 20 à la position 1
tabI[2]=35; //ecriture de la valeur 35 à la position 2

int i = tabI[0] + tabI[1] + tabI[2]; //i recoit la somme des elements du tableau
```

Figure 2.9. Accès aux éléments d'un tableau



3.1.3.2. Propriétés

Nous allons utiliser comme propriété du tableau sa longueur : *length*.

Comme nous l'avons vu l'accès au propriétés d'un objet se fait en utilisant l'opérateur . (point). Ainsi la longueur du tableau *tab* est accessible en utilisant *tab.length*. Le code suivant permet d'afficher la taille du tableau *tabI* ainsi que le dernier et le premier élément.

```
int[] tabI;
tabI = new int[3];
tabI[0]=10;
tabI[1]=20;
tabI[2]=35;

System.out.println(tabI.length);
//affiche la longueur du tableau ici 3
System.out.println(tabI[tabI.length - 1])
//affiche la valeur de la derniere case du tableau ici 35
System.out.println(tabI[0])
//affiche la valeur de la première case du tableau ici 10
```

3.1.3.3. Pièges

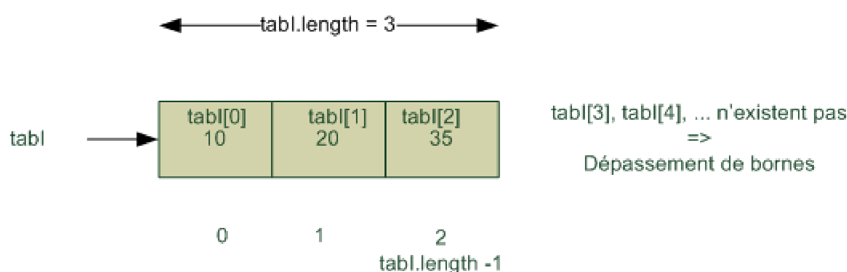
Les tableaux contiennent deux pièges intrinsèques : le dépassement des bornes et les alias involontaires.

3.1.3.3.1. Dépassement des bornes

Les tableaux sont des structures statiques dont la taille est choisie à la création, il est impossible d'accéder à un élément à l'extérieur des bornes. Si notre tableau *tabI* a pour taille 3 : t[-1] et t[3] conduisent par exemple au message *ArrayIndexOutOfBoundsException* (Erreur de dépassement des bornes par l'indice du tableau).

Vous rencontrez probablement se problème un jour ou l'autre aussi n'en oubliez pas la raison.

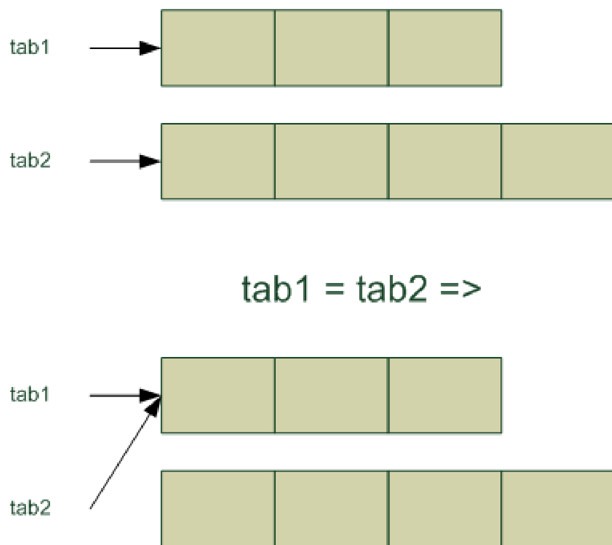
Figure 2.10. Dépassement des bornes



3.1.3.3.2. Alias

Nous accédons aux tableaux via une référence aussi si *tab1* et *tab2* sont deux références de tableaux : *tab1 = tab2* ne recopie pas le tableau référencé par *tab2* dans celui référencé par *tab1* mais recopie la référence de *tab2* dans *tab1*. *tab1* devient un alias sur *tab2*, ils référencent le même tableau et toute modification par l'un affecte en conséquence l'autre. Nous avons deux noms (*tab1*, *tab2*) pour une même chose (un même tableau).

Figure 2.11. Alias



3.2. Généralités sur les tris

Les tris peuvent-être internes ou externe (il utilisent d'autre structures de données), séquentiels ou parallèles. Nous allons aborder des algorithmes de tri internes, séquentiels avec une approche itérative.

La compléxité des tris pour des données quelconques ne peut-être inférieur à *n ln* pour les tris par comparaison et est souvant en n^2.

3.3. Tri par insertion

L'idée est de prendre les éléments les uns après les autres et de les insèrer parmi les éléments déjà triés => Une boucle pour parcourir le tableau, une boucle pour insérer à la bonne place. La complexité est en n^2.

```
public static void triInsertion(int tableau[])
    {
  int i,j, m;
  int l = tableau.length -1;
  for (i=1; i <= l; i++)
  {
   m = tableau[i];
   j=i;
   while (j>0 && tableau[j-1]>m)
   {
    tableau[j]=tableau[j-1];
    j=j-1;
   }
   tableau[j]=m;
  }
    }
```

3.4. Tri par selection

L'Idée est de trouver le plus petit et le placer en premier puis trouver le plus petit parmi les éléments non placés et le placer en second, etc. => Une boucle de parcour du tableau Une boucle pour trouver le minimum. La complexité est en n^2.

```
public static void triSelection(int tableau[])
 {
  int i,j,m, min;
  int l = tableau.length - 1;
  for (i=0; i< l; i++)
  {
```

```
   min = i;
   for (j=i+1; j<=l; j++ )
   {
    if (tableau[j]< tableau[min])
     min=j;
   }
   m=tableau[i];
   tableau[i] = tableau[min];
   tableau[min] = m;
  }
 }
```

3.5. Tri à bulle

L'iIdée est de permuter les éléments adjacents mal placé et de parcourir autant de fois que nécessaire le tableau => Une boucle pour selectionner les éléments et une boucle pour les permutations. La complexité est en n^2.

```
public static void triBulle(int tableau[])
 {
  int i,j,m;
  int l = tableau.length -1;
  for ( i=l; i>=0; i--)
   for (j=1; j<=i; j++)
    if (tableau[j-1] > tableau[j])
    {
     m = tableau[j-1];
     tableau[j-1] = tableau[j];
     tableau[j]=m;
    }
 }
```

Chapter 3. Exercices

Nous commencerons par une approche itérative avec la liste, la pile pour nous reviendrons sur la liste pour en proposer une approche récursive.

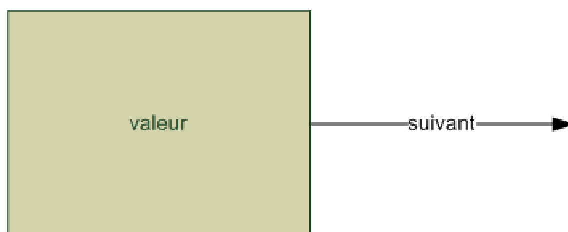
1. Cellule

Nous allons commencer par définir une cellule. Les cellules nous permettrons plus tard de créer des listes simplement chaînées.

1.1. Présentation

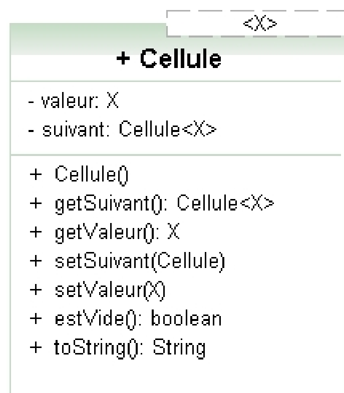
Une cellule est définie par une valeur et une référence vers une autre cellule.

Figure 3.1. Cellule



Une référence qui n'a pas encore reçu de valeur a la valeur *null*.

Figure 3.2. Diagramme de classe de Cellule



Le que vous observé est un type passé en paramètre, ainsi la classe Cellule est indépendante du type, on parle de généricité.

1.2. Codage

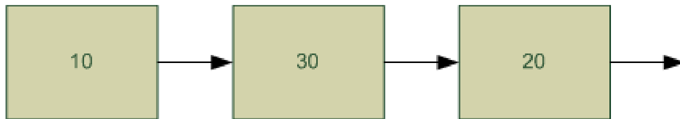
Donner le code la classe Cellule sachant que le constructeur construit une cellule vide.

2. Liste simplement chaînée en version itérative

Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, une ou des références vers les éléments qui lui sont logiquement adjacents dans la liste.

Une liste simplement chaînée est une liste qui ne possède qu'une référence, pour nous elle sera vers la cellule de droite.

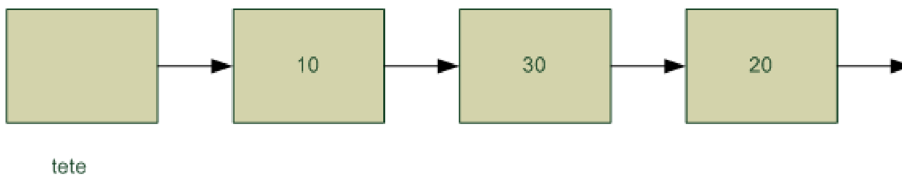
Figure 3.3. Liste simplement chaînée



2.1. Présentation

Nous souhaitons que notre classe Liste soit accessible via une cellule particulière nommée "tete". Cette cellule ne contient pas de valeur mais permet simplement de connaître le début de la liste donc la liste entière.

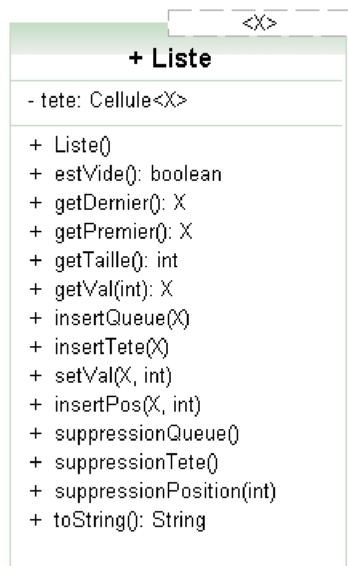
Figure 3.4. Liste simplement chaînée réelle



2.2. Codage

Vous aller réaliser l'implémentation d'une liste répondant au diagramme de classe UML suivant :

Figure 3.5. Diagramme UML de la classe Liste



Donner pour chacune des étapes qui vont suivre le code java correspondant et l'ordre de grandeur de la complexité temporelle.

2.2.1. Variable(s) d'instance(s)

La ou les variables d'instance. Justifier de ces ou de cette variable.

2.2.2. Constructeur sans paramètre

Donner le code java correspondant.

2.2.3. Méthode estVide

Une méthode pour savoir si la liste est vide.

2.2.4. Méthode getPremier

Accès au premier élément de la liste.

2.2.5. Méthode getDernier

Accès au dernier élément de la liste.

2.2.6. Méthode insereTete

On insère une nouvelle valeur en tête de liste.

2.2.7. Méthode insereQueue

On insère une nouvelle valeur en queue de liste.

2.2.8. Méthode suppressionTete

On supprime la valeur en tête de liste.

2.2.9. Méthode suppressionQueue

On supprime la valeur en queue de liste

2.2.10. Exercices complémentaires

Nous pouvons enrichir notre liste avec des méthodes, pour pouvoir l'utiliser comme un tableau de taille variable.

2.2.10.1. Méthode getTaille

Renvoie le nombre d'éléments de la liste que nous recomptons à chaque fois.

> **Note**
>
> Il existe bien entendu, une meilleures solution qui consiste à rajouter une variable d'instance incrémentée à chaque insertion et décrémentée à chaque suppression dans la liste.

2.2.10.2. Méthode getVal

Renvoie l'élément à une position quelconque.

2.2.10.3. Méthode setVal

Modifie un élément à une position quelconque.

2.2.10.4. Méthode insertPosition

Insère un élément à une position quelconque.

2.2.10.5. Méthode suppressionPosition

Supprime l'élément à une position quelconque.

3. Pile

Une pile est une structure de données de type LIFO (Last In First Out). Ce qui signifie que l'élément inséré en dernier dans une pile est le premier à en être extrait. L'analogie avec une pile d'assiette nous amène facilement à définir les méthodes de la Pile.

3.1. Présentation

Nous allons utiliser une liste pour implémenter notre Pile.

La pile possède trois méthodes :

1. estVide qui permet de savoir si la pile est vide.
2. push qui permet de rajouter un élément
3. pop qui permet de retire un élément

Figure 3.6. Diagramme de classe de Pile

3.2. Codage

Donner le code de la classe Pile.

4. Liste simplement chaînée en version récursive

Nous allons reprendre l'exemple précédant et l'implémenter sous forme récursive.

4.1. Rappel sur la récursivité

Il est des problèmes en algorithmique comme les parcours d'arbres qui ne peuvent être résolus simplement sans récursivité. La récursivité consiste a s'appeler soit même. Le principe se retrouve en art, en linguistique et bien évidement en informatique.

4.1.1. Définition

On appelle récursivité le fait pour un sous-programme (méthode) de s'appeler au moins une fois. La vision récursive s'oppose bien souvent à la vision itérative.

4.1.2. Exemples

Pour réaliser une méthode récursive, il faut :

- un point d'arrêt
- trouver un sous problème identique au problème et exprimer le résultat en fonction de ce sous problème.

4.1.2.1. Factorielle

Le calcul d'une factorielle peut s'exprimer de deux manières :

- n! = n * n - 1 * ... * 1
- n! = n * (n -1)!

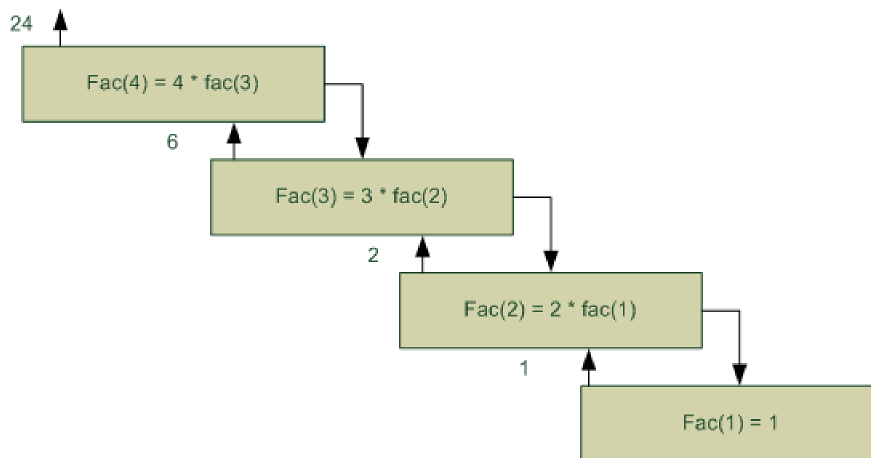
La première version permet l'implantation d'une fonction itérative. La deuxième permet une implantation récursive de la fonction factorielle.

Le cas d'arrêt et 0! = 1, le sous problème (n-1)!, et le moyen de relier le problème n! au sous problème.

Example 3.1. Factorielle récursive

```
public static int facR(int n)
 {
  int res;
  if (n==0 || n==1) //Cas d'arret
  {
   res = 1;
  }
  else
  {
   res = n * facR(n-1); //Appel recursif
  }
  return res;
 }
```

Figure 3.7. Appel récursifs de la factorielle



4.1.2.2. Somme d'un tableau d'entiers

Le cas d'arrêt est le tableau à une case, nous pouvons conclure que sa somme est celle de sa case. L'appel récursifs est obtenu en constatant que la somme des éléments est égale au premier élément plus la somme du sous tableau restant. Nous supposons la méthode extraction qui permet d'extraire un sous tableau est donnée

> ## Important
>
> Ce premier exemple est a oblier aussitôt car a chaque appel, il y a recopie du tableau, une version plus performante suivra.

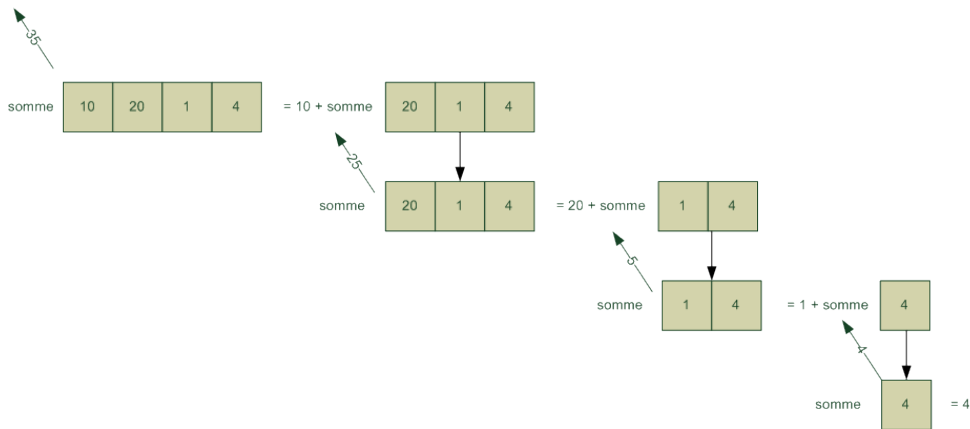
:

```
public static int[] extraction(int[] t, int debut, int fin)
 {
  int[] res;
  if (0<fin-debut && fin-debut < t.length)
  {
   res = new int[fin-debut+1];
   for (int i=0; i < res.length; i++)
   {
    res[i]=t[debut+i];
   }
  }
  else
  {
   res = null;
  }
  return res;
 }
```

Example 3.2. Somme récursive des éléments d'un tableau d'entiers

```
public static int sommeR(int[] t)
 {
  int res = 0;
  if (t.length ==1)
  {
   res = t[0];
  }
  else
  {
   res = t[0]+sommeR(extraction(t, 1, t.length-1));
  }
  return res;
 }
```

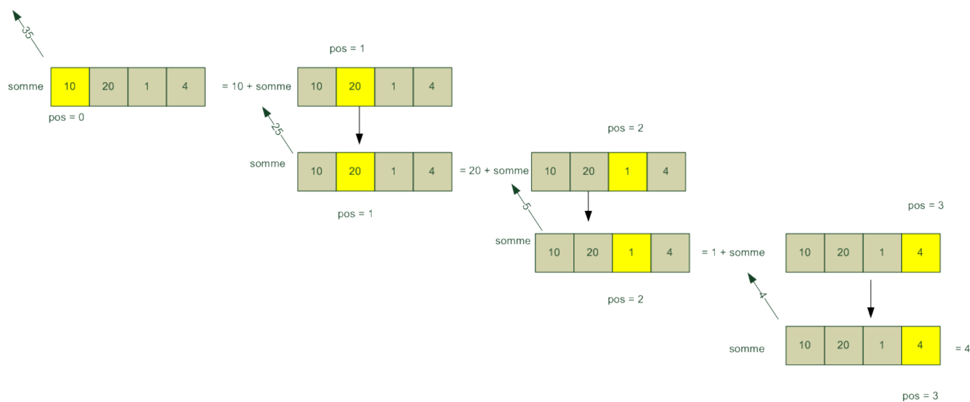
Figure 3.8. Appel récursif de la somme des éléments d'un tableau d'entiers



La solution précédente consomme de la mémoire, à chaque appel récursif un tableau est créé, nous pouvons proposer une nouvelle version avec un marqueur qui indique la position du sous-tableau à traiter :

```
public static int sommeR2(int[] t, int pos)
 {
  int res;
  if (pos == t.length -1)
  {
   res = t[pos];
  }
  else
  {
   res = t[pos] + sommeR2(t, pos +1);
  }
  return res;
 }
```

Figure 3.9. Appel récursif de la somme des éléments d'un tableau d'entiers avec un index.



4.2. Exercice

Voici trois méthodes pour commencer la récursivité :

```
private X car()
 {
  X res = null;
  if (!estVide())
   res = this.tete.getSuivant().getValeur();
  return res;
 }

 private List<X> cdr()
 {
  List<X> res = new List<X>();
  if (!estVide())
   res.tete = tete.getSuivant();
  return res;
 }

 public boolean estVide()
 {
  return this.tete.getSuivant() == null;
 }
```

`car(x)` renvoie le premier élément de la liste, `cdr()` renvoie la liste privée du premier élément.

Figure 3.10. Liste récursive

Coder dans l'ordre suivant les méthodes suivantes.

4.2.1. getTaille

`public int getTaille()` renvoie la taille de la liste.

4.2.2. getVal

`public X getVal(int pos)` renvoie la valeur en position *pos*.

4.2.3. setVal

`public void setVal(X val,int pos)` modifie la valeur de l'élément de position *pos*.

4.2.4. insertionPosition

`public void insertionPosition(X val, int pos)` insert *val* en position *pos*.

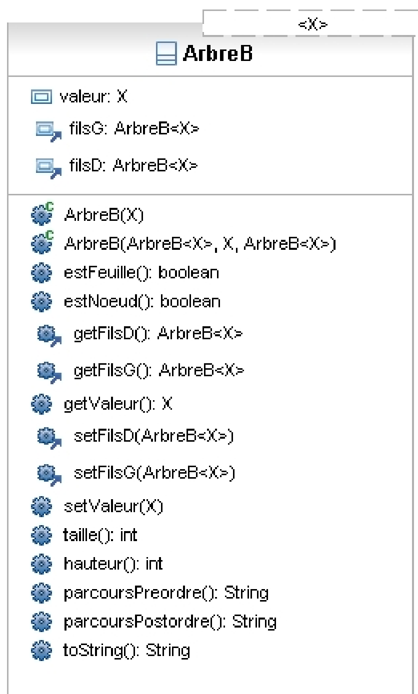
4.2.5. suppressionPosition

`public void suppressionPosition(int pos)` supprime l'élément de position *pos*.

5. Les arbres binaires.

Nous allons implémenter un arbre binaire, à savoir un arbre dont chaque noeud possède au plus un fils gauche et un fils droit.

Figure 3.11. Classe ArbreB

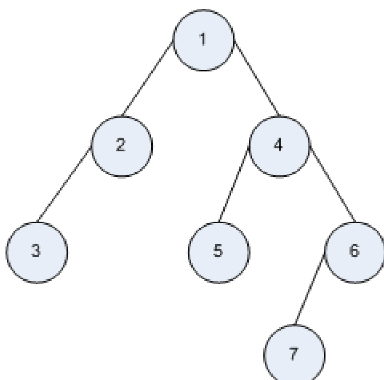


Notre arbre binaire ne peut-être construit vide. Pour les exercices qui suivent, il vous faudra à chaque fois, au plus tôt tester vos méthodes.

5.1. Création de la classe

Créer la classe arbre binaire avec pour le moment, les attributs et les constructeurs, puis dans un fichier de test séparer reproduire l'arbre suivant :

Figure 3.12. Exercice arbre binaire



5.2. Les feuilles et les noeuds

Un noeud est un élément de l'arbre, un noeud particulier est la racine, les noeuds qui n'ont pas de fils sont appelés feuilles.

Coder la méthode `public boolean estFeuille()` qui renvoie `true` si l'arbre est une feuille et `false` sinon, coder de même la méthode `public boolean estNoeud()`.

5.3. La taille

La taille est le nombre de noeuds de notre arbre, coder : public int taille().

Ici, vous devez trouver sept.

5.4. La hauteur

La profondeur d'un noeud, est le est la longueur du chemin allant de ce noeud à la racine.

Ici la profondeur du noeud contenant 3 est 2.

La hauteur est la profondeur maximum, ici 3.

Coder la méthode `public int hauteur()`.

5.5. Parcours en préordre

Le parcours préordre est un parcours d'arbre qui repose sur la méthode suivante :

- examiner la racine
- parcourir en préordre le sous-arbre gauche
- parcourir en préordre le sous-arbre droite

Coder la méthode `public String parcoursPreordre()`.

Vous devez trouver : 1234567.

5.6. Parcours en postordre

Le parcours postodre est un parcours d'arbre qui repose sur la méthode suivante :

- parcourir en postordre le sous-arbre gauche
- parcourir en postordre le sous-arbre droite
- examiner la racine

Coder la méthode `public String parcoursPostordre()`.

Vous devez trouver : 3257641.

Appendix A. Import et export de projet sous eclipse

L'unité de travail sous eclipse est le projet, voici deux moyens pour échanger vos données en exportant et en important vos données.

1. Export

Pour exporter un projet, vous pouvez suivre la procédure suivante :

1. Clic-droit sur le projet puis "export"
2. choisir "general" et "archive file"
3. choisir les sources à exporter

2. Import d'un projet dans un Workspace existant

Pour importer un projet dans un workspace existant :

1. choisir "file" puis "import"
2. choisir "general" puis "existing projects into workspace"