Introduction à l'Informatique (INF 311)

F. Morain

Amphi 9 : listes

26 juin

I. Les listes par l'exemple.

II. Listes d'entiers.

III. Retour aux fichiers.

IV. Une autre façon d'implanter les piles.

V. Les pales machine.

0. Où en sommes-nous?

- Amphi 1: introduction.
- Amphi 2: programmer en Java.
- Amphi 3: fonctions/fonctions récursives.
- Amphi 4: tableaux/String.
- Amphi 5: classes.
- Amphi 6: tables.
- Amphi 7: algorithmes et complexité.
- Amphi 8: Internet.
- Amphi 9: listes.
- Amphi 10: système/sécurité.

I. Les listes par l'exemple

Pb: Comment organiser et répartir les fichiers sur un disque pour permettre de les créer, agrandir et supprimer.

Principes: disque = suite de cases mémoires; un fichier est une zone mémoire sur le disque. D'où deux classes, **`Disque`** et **`Fichier`**.

Spécifications:

- On se donne la taille du disque, et on retourne une erreur quand il est plein.
- On modélise le disque comme étant un tableau qui référence les parties libres ou utilisées du disque. À chaque nouvelle demande de création ou d'extension d'un fichier, on cherche un espace libre et on le réserve pour le fichier.
- On veut pouvoir également supprimer un fichier.
- (Dans la vraie vie, on doit aussi gérer les blocs corrompus.)


```
public class Disque{

  final static int TAILLE = 10240;
  // simulation de la mémoire
  static byte[] mem;
  // pour savoir si une case est libre
  static boolean[] libre;

  public static void initialiser(){
    mem = new byte[TAILLE];
    libre = new boolean[TAILLE];
    for(int i = 0; i < TAILLE; i++)
      libre[i] = true;
  }
}
```

L'allocation d'un morceau du disque

```
public static int allouer(int nb){
  int adr = chercherEspaceLibre(nb);

  if(adr == -1)
    // plus de mémoire
    return -1;
  else{
    // on marque l'espace réservé
    for(int j = adr; j < adr+nb; j++)
      libre[j] = false;
    return adr;
  }
}
```



```
public static int chercherEspaceLibre(int nb){
  int adr;
  boolean ok;

  // est-ce que l'intervalle [adr, adr+nb-1]
  // est libre?
  for(adr = 0; (adr+nb-1) < TAILLE; adr++){
    ok = true;
    for(int j = adr; j < adr+nb; j++)
      if(!libre[j]){
        ok = false;
        break;
      }
    if(ok)
      return adr;
  }
  return -1;
}
```



```
public class Fichier{
  String nom;
  int adr, taille;
  public Fichier(String s, int t){
    this.nom = s;
    this.taille = t;
    this.adr = Disque.allouer(t);
    if(this.adr == -1)
      System.out.println("Plus de mémoire");
    for(int i=this.adr; i<this.adr+t; i++)
      Disque.mem[i] = 1; // données
  }

  public static void afficher(Fichier f){
    System.out.print("Le fichier "+f.nom);
    System.out.println(" a pour taille="+f.taille);
    System.out.print("et est stocké à partir de ");
    System.out.println("l'adresse "+f.adr);
  }
}
```



```
public static void main(String[] args){
  Fichier f1, f2;

  Disque.initialiser();
  f1 = new Fichier("tata", 512);
  afficher(f1);
  f2 = new Fichier("tete", 1024);
  afficher(f2);
}
```

```
Le fichier tata a pour taille=512
et est stocké à partir de l'adresse 0
Le fichier tete a pour taille=1024
et est stocké à partir de l'adresse 512
```

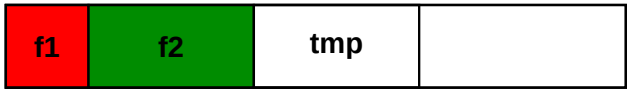
Visualisation

```
f1 = new Fichier("tata", 512);
```

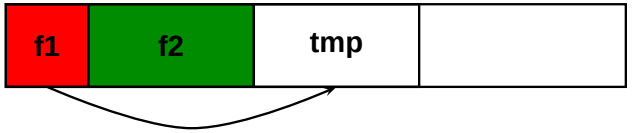

```
f2 = new Fichier("tete", 1024);
```

Pb: que se passe-t-il quand on veut agrandir `f1`?

Première stratégie: on alloue un nouvel espace mémoire avec la nouvelle taille:



On recopie l'ancien fichier dans le nouveau:



Puis on efface l'ancien, ce qui redonne de l'espace libre:


```
public static boolean agrandir(Fichier f, int q){
  int a = Disque.allouer(f.taille + q);

  if(a == -1)
    // on a rempli le disque!
    return false;
  else{
    // on transfère le contenu de f
    Disque.transferer(a, f.adr, f.taille);
    // on libère la mémoire
    Disque.liberer(f.adr, f.taille);
    f.adr = a;
    f.taille += q;
    return true;
  }
}
```


Dans la classe **`Disque`**:

```
// on recopie la mémoire
public static void transferer(int dest,
                              int src,
                              int taille){
  for(int i = 0; i < taille; i++)
    mem[dest+i] = mem[src+i];
}
// on marque que la mémoire est libérée
public static void liberer(int adr, int nb){
  for(int i = adr; i < adr+nb; i++)
    libre[i] = true;
}
```

Pbs.

- Le transfert de fichiers sur disque est coûteux (temps de recopie; usure de la tête de lecture, du bras, etc.).
- Fragmentation: si on veut ajouter un fichier plus grand que le max des espaces libres, alors que la somme de la place libre aurait été suffisante.

Meilleure idée: le fichier est formé de plusieurs blocs, et à la fin de chacun, on trouve l'adresse du suivant en mémoire:

Comment réaliser cela concrètement: un fichier va être formé d'une suite de blocs. Chaque bloc contient une adresse de début de zone mémoire, d'une taille (comme dans la stratégie 1) et l'adresse du bloc suivant (ou un indicateur qu'il n'y a pas de suivant). On dit que chaque bloc est chaîné au précédent.
Un fichier va seulement contenir le premier bloc de la suite.

La classe pour gérer les blocs est:

```
public class ListeDeBlocs{
  int adr, taille;
  ListeDeBlocs suivant;
}
```

L'adresse d'un bloc est simplement la référence de l'objet correspondant! C'est Java qui va se charger de l'allocation, à l'aide de **new**!

On dit que la classe **ListeDeBlocs** a une structure récursive puisque la classe **ListeDeBlocs** fait référence à elle-même. Notons qu'un objet de la classe **ListeDeBlocs** ne fait bien sûr pas référence à lui-même (en général).

II. Listes d'entiers

On commence par considérer une structure simple: on relie entre elles des cellules contenant des entiers. Une liste est une suite de cellules. On se livre à un jeu de piste dans la mémoire.

							⇓		
···	@1000	@1004	···	@2000	@2004	···	@3000	@3004	···
···	12	@2000	···	3	@0	···	4	@1000	···

@0 est la convention pour signifier que la liste s'arrête là.

Évidemment, ça rappelle furieusement ce qu'on a déjà vu avec les objets. . .

Déclaration d'une liste chaînée en Java

```
public class Liste{
  int contenu;
  Liste suivant;

  public Liste(int c, Liste suiv){
    this.contenu = c;
    this.suivant = suiv;
  }
  public static void main(String[] args){
    Liste l; // l = @0

    l = new Liste(3, null); // l = @2000
    l = new Liste(12, l);   // l = @1000
    l = new Liste(4, l);    // l = @3000
  }
}
```

Le grand dessin

Pour comprendre les listes, il faut faire des dessins!

```
Liste l; // déclaration
```

@10
l
@0 (**null**)

```
l = new Liste(3, null);
```

@10
l
@2000

@2000	@2004
.contenu	**.suivant**
3	@0

```
l = new Liste(12, l);
```

@10	@2000	@2004	@1000	@1004
l	**.contenu**	**.suivant**	**.contenu**	**.suivant**
@1000	3	@0	12	@2000

@10	@2000	@2004	@1000	@1004
l	**.contenu**	**.suivant**	**.contenu**	**.suivant**
@1000	3	@0	12	@2000

```
l = new Liste(4, l);
```

@10	@2000	@2004	@1000	@1004
l	**.contenu**	**.suivant**	**.contenu**	**.suivant**
@3000	3	@0	12	@2000

@3000	@3004
.contenu	**.suivant**
4	@1000


```
public static void main(String[] args){
  Liste l;
  l = new Liste(3, null);
  l = new Liste(12, l);
  l = new Liste(4, l);
  System.out.println(l.contenu); // 4
  System.out.println(l.suivant.contenu); // 12
  System.out.println(l.suivant.suivant.contenu); // 3
}
```

Pour afficher un peu mieux:

```
public static void afficherContenu(Liste l){
  Liste ll = l;

  while(ll != null){
    System.out.print(ll.contenu + " -> ");
    ll = ll.suivant;
  }
  System.out.println("null");
}
```


Encore plus idiomatique

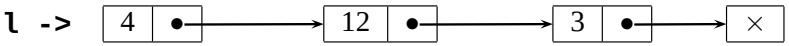
```
public static void afficherContenu(Liste l){
  while(l != null){
    System.out.print(l.contenu + " -> ");
    l = l.suivant;
  }
  System.out.println("null");
}
```

La liste originale (le **l** du **main**) n'est pas perdue, puisqu'on a juste recopié son adresse (sa référence) dans le paramètre de la fonction (le **l** de **afficherContenu**).

Abstractions

@10	@2000	@2004	@1000	@1004
l	**.contenu**	**.suivant**	**.contenu**	**.suivant**
@3000	3	@0	12	@2000

@3000	@3004
.contenu	**.suivant**
4	@1000



l -> 4 -> 12 -> 3 -> null

Longueur d'une liste

```
public static int longueur(Liste l){
  int lg = 0;

  while(l != null){
    lg++;
    l = l.suivant;
  }
  return lg;
}

public static int longueurRec(Liste l){
  if(l == null) // test d'arrêt
    return 0;
  else
    return 1 + longueurRec(l.suivant);
}
```


Faisons le point

Une liste chaînée permet de stocker des informations l'une après l'autre. On peut accéder directement à la cellule en tête de liste. Accéder aux autres cellules demande à parcourir toute la liste. On ne peut accéder facilement à la longueur de la liste.

On parle d'accès séquentiel par opposition à un accès indexé (aléatoire, ou encore direct).

Ex. Renvoyer l'entier contenu dans la i-ème cellule.

```
// on suppose i < longueur de la liste
public static int ieme(Liste l, int i){
  for(int j = 0; j < i; j++)
    l = l.suivant;
  return l.contenu;
}
```


Tableau ou liste?

On utilise un tableau quand:

- on connaît la taille (ou un majorant proche) à l'avance;
- on a besoin d'accéder aux différentes cases dans un ordre quelconque (accès direct à **t[i]**).

On utilise une liste quand:

- on ne connaît pas la taille *a priori*;
- on n'a pas besoin d'accéder souvent au i-ème élément;
- on ne veut pas gaspiller de place.

Un mauvais cas d'utilisation d'un tableau

Ex: supposons que l'on veuille créer un tableau qui simule une liste, ce qui nous permettrait d'avoir accès à la longueur de la liste.

```
public static int[] creerTableau(int n){
  int[] t, tmp;

  t = new int[1];
  t[0] = 1;
  for(int i = 2; i <= n; i++){
    tmp = new int[i];
    for(int j = 0; j < t.length; j++)
      tmp[j] = t[j];
    tmp[i-1] = i;
    t = tmp;
  }
  return t;
}
```

Coût: $O(n^2)$ recopies.

Ajout en queue

Principe: si la liste est vide, on crée une nouvelle cellule que l'on retourne; si la liste n'est pas vide, on cherche la fin de la liste et on rajoute une nouvelle cellule.

```
public static Liste ajouterEnQueue(Liste l,int c){
  if(l == null)
    return new Liste(c, null);
  // l a un suivant
  Liste ll = l; // cherchons la dernière cellule
  while(ll.suivant != null)
    ll = ll.suivant;
  ll.suivant = new Liste(c, null);
  return l; // pas ll...
}
```

Rem. On retourne une liste uniquement à cause du cas `null`.

Rem. L'ajout en queue est beaucoup plus coûteux que l'ajout en tête.

Version récursive:

```
public static Liste ajouterEnQueueRec(Liste l,
                                      int c){
  if(l == null)
    return new Liste(c, null);
  else{
    // les modifications vont avoir
    // lieu dans la partie
    // "suivante" de la liste
    l.suivant=ajouterEnQueueRec(l.suivant, c);
    return l;
  }
}
```


Copier une liste

```
public static Liste copier(Liste l){
  Liste ll = null;

  while(l != null){
    ll = new Liste(l.contenu, ll);
    l = l.suivant;
  }
  return ll;
}
```

mais cela copie à l'envers. Si

```
l : 1 -> 2 -> 3 -> null
```

la fonction crée:

```
ll : 3 -> 2 -> 1 -> null
```

Copie à l'endroit

```
public static Liste copierALEndroit(Liste l){
  if(l == null)
    return null;
  else{
    Liste tmp = copierALEndroit(l.suivant);
    return new Liste(l.contenu, tmp);
  }
}
```

Ex. Récrire cela de manière itérative.

III. Retour aux fi chiers

On modifie la structure de fichier de sorte à gérer une liste de blocs. Quand le fichier grandit, on rajoute un bloc à la fin de la liste.

Comme l'ajout en queue est coûteux (proportionnel à la longueur de la liste), on a intérêt à rajouter un champ qui repère toujours la dernière cellule utilisée, de façon à faire l'ajout de bloc en temps constant.


```
public class ListeDeBlocs{
  int adr; // adresse de début de bloc
  int taille; // taille du bloc alloué
  ListeDeBlocs suivant;

  public ListeDeBlocs(int a, int t,
                      ListeDeBlocs suiv){
    this.adr = a;
    this.taille = t;
    this.suivant = suiv;
  }
}
```



```
public class Fichier{
  String nom;
  ListeDeBlocs origine, dernier;

  public Fichier(String s, int t){
    int adr;

    this.nom = s;
    adr = Disque.allouer(t);
    this.origine=new ListeDeBlocs(adr,t,null);
    this.dernier = this.origine;
    for(int i = 0; i < t; i++)
      Disque.mem[adr + i] = 1;
  }
}
```

Création d'un **Fichier**:

```
Fichier f = new Fichier("toto", 512);
```

@320
f
@10

@10	@20	@30
.nom	**.origine**	**.dernier**
@100	@200	@200

@200	@210	@220
.adr	**.taille**	**.suivant**
0	512	@0

Calcul de la taille d'un fi chier

Principe: on doit parcourir toute la liste pour cumuler les tailles des différents blocs.

```
public static int taille(Fichier f){
  ListeDeBlocs l = f.origine;
  int t = 0;

  while(l != null){
    t += l.taille;
    l = l.suivant;
  }
  return t;
}
```


Affi chages

```
public static void afficher(Fichier f){
  System.out.print("Le fichier "+f.nom);
  System.out.println(" a pour taille="+taille(f));
  System.out.print("et est stocké à partir de ");
  System.out.println("l'adresse "+f.origine.adr);
}
```

On affiche le contenu de tous les blocs dans l'ordre:

```
public static void afficherContenu(Fichier f){
  ListeDeBlocs l;
  for(l=f.origine; l!=null; l=l.suivant)
    for(int i=l.adr; i<l.adr+l.taille; i++)
      System.out.print(Disque.mem[i]);
}
```


Agrandir un fi chier: c'est tout simple!

```
// on veut rajouter q octets à f
public static boolean agrandir(Fichier f, int q){
  ListeDeBlocs l;
  int a;

  a = Disque.allouer(q);
  if(a == -1)
    return false;
  else{
    l = new ListeDeBlocs(a, q, null);
    f.dernier.suivant = l;
    f.dernier = f.dernier.suivant;
    for(int i = 0; i < q; i++)
      Disque.mem[a + i] = 1;
    return true;
  }
}
```

Agrandissement d'un **`Fichier`**:

@320
f
@10

@10	@20	@30
.nom	**.origine**	**.dernier**
@100	@200	@500

@200	@210	@220
.adr	**.taille**	**.suivant**
0	512	@500

@500	@510	@520
.adr	**.taille**	**.suivant**
1024	512	@0

On vient de réinventer le format FAT (*File Allocation Table*) de Microsoft! Dans la vraie vie:

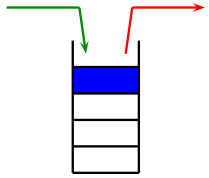
FAT12: créé pour les disques souples (floppy) en 1980; adresses de blocs (clusters, 512 octets alloués quoi qu'il arrive) sur 12 bits (limité à 2 Mo, mais floppy 360 ko...).

FAT16: 1983, adresses sur 16 bits (32 Mo); puis taille des clusters portée à 8192 octets (512 Mo), mais problème de fragmentation augmente.

VFAT: pour Windows 95 (noms de fichier plus longs); FAT32; FAT utilisé dans les clefs USB. Pas très fiable.

Windows NT: NTFS (secret).

IV. Une autre façon d'implanter les piles



Propriétés abstraites: on veut pouvoir disposer des opérations suivantes: tester si la pile est vide, empiler un élément, le dépiler.

Exemple d'utilisation

```
public class TestPile{
  public static void main(String[] args){
    Pile p = new Pile(100);

    for(int i = 20; i >= 0; i--)
      Pile.empiler(p, i);
    while(!Pile.estVide(p))
      System.out.println(Pile.depiler(p));
  }
}
```

```
0
1
...
20
```

Implantation avec un tableau

```
public class Pile{
  int[] p;
  int h; // dernier indice rempli
  public Pile(int hmax){
    this.p = new int[hmax]; this.h = -1;
  }
  public static boolean estVide(Pile pile){
    return (pile.h == -1);
  }
  public static void empiler(Pile pile, int a){
    pile.p[++pile.h] = a;
  }
  public static int depiler(Pile pile){
    return pile.p[pile.h--];
  }
}
```


Avec des listes: le sommet de la pile est en tête de la liste.

```
public class Pile{
  Liste l;
  public Pile(){
    this.l = null;
  }
  public static boolean estVide(Pile p){
    return (p.l == null);
  }
  public static void empiler(Pile p, int a){
    p.l = new Liste(a, p.l);
  }
  public static int depiler(Pile p){
    int a = p.l.contenu;
    p.l = p.l.suivant;
    return a;
  }
}
```


C'est le même programme! et le même résultat si les piles sont correctement implantées.

```
public class TestPile{
  public static void main(String[] args){
    Pile p = new Pile(); // pas de taille

    for(int i = 20; i >= 0; i--)
      Pile.empiler(p, i);
    while(!Pile.estVide(p))
      System.out.println(Pile.depiler(p));
  }
}
```

Rem. On a donné deux implantations d'un même type abstrait de données (TDA).

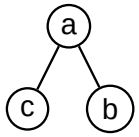
Conclusions provisoires

Les listes sont une des structures fondamentales de l'informatique, plus riches que tableau ou objet simple.
Des langages basés sur les listes existent (Lisp): concept de S-expression.
Permet de représenter des données sous forme creuse: on peut stocker le polynôme

$$X^{100} + X + 1$$

dans un tableau de 101 coefficients, ou bien dans une liste à 3 éléments; idem pour des matrices avec peu de coefficients $\neq 0$.

Aller plus loin dans les structures récursives




```
public class Arbre{
  int racine;
  Arbre filsg, filsd;

  public Arbre(int sommet, Arbre fg, Arbre fd){
    this.racine = sommet;
    this.filsg = fg;
    this.filsd = fd;
  }
}
```

Cf. cours INF 421.

V. Les pales machine

Derniers mots

Résumé du cours: introduction aux listes; pale machine.

Prochains rendez-vous :

Groupes	TD 9
1–6	13h30–15h30
7–12	15h45–17h45

Prochain amphi: A10 le lundi 3 juillet à 10h30 en amphi Arago.

11 juillet: pale papier.