Algorithmique et structures de données

CM 1 – Langage algorithmique et tableaux

Jean-Marie Le Bars
jean-marie.lebars@unicaen.fr

6 septembre 2022

Plan du CM 1

Plan du CM1

Objectif du cours

Algorithmes et structures de données

Avant de concevoir des algorithmes, il est essentiel de définir proprement les structures de données nécessaires.

L'objectif est d'étudier les structures de données les plus utilisées

On peut les regrouper en trois familles :

- les structures linéaires
- les structures arborescentes
- les structures de recherche

Comment choisir une structure de données

- chaque structure possède ses avantages et ses inconvénients
- le choix s'effectue selon les problèmes que l'on souhaite résoudre avec ces structures

Structures linéaires

Caractéristiques générales

- les éléments sont organisés dans une séquence
- de telles structures permettent de définir une fonction de successeur qui permet de passer d'un élément à un autre

Exemples de structures linéaires

- tableaux allocation statique continue, accès direct
- listes allocation dynamique
  - listes simplement chaînées, doublement chaînées, circulaires...
- files FIFO, First In First Out – Premier entré premier sorti
- piles LIFO, Last In First Out – Dernier entré premier sorti

les structures arborescentes

Structures arborescentes

- un élément peut accéder à plusieurs éléments.
- arbre binaire : un nœud peut accéder à son nœud gauche et son nœud droit.

Exemples de structures arborescentes

- arbres binaires
- arbres généraux
- forêts liste d'arbres

Structures de recherche

Définition informelle

- on dispose d'un ensemble d'éléments
- on les organise afin de pouvoir les retrouver facilement
- l'ensemble peut évoluer au cours du temps
  - insertion, suppression, réorganisation...

Exemples de structures de recherche

- arbres binaires de recherche
- arbres rouge-noir
- arbres des préfixes ou tries
- tables de hachage
- tas

Organisation du cours

CM - 15h

10 séances d'1h30 – Langage algorithmique (pseudo-langage).

TD - 25h

10 séances de 2*h*30 – Langage algorithmique

TP - 10h

5 séances de 2*h* – Langage C, plateforme caseïne

Contrôle des connaissances

- Contrôle continu : un devoir maison – compte pour 40%
  Deux notes
  1. une note sur la partie théorique à rédiger en langage algorithmique
  2. une note sur la partie pratique à écrire en langage C

- Examen terminal de 2h – compte pour 60%
  À rédiger en langage algorithmique, pas de programmation

Organisation du cours

TP

- Deux demi-groupes a et b pour chaque groupe de TD
- 5 séances de TP de 2h pour chaque demi-groupe
- Alternance groupe a, groupe b pendant 10 semaines
- à programmer en langage C
- à réaliser sur la plateforme caseïne

Organisation du cours

Planning (peut être amené à changer)

```
S1  29 août
S2   5 septembre    CM1
S3  12 septembre    CM2  TD1
S4  19 septembre    CM3  TD2  TP1 groupe a
S5  26 septembre    CM4  TD3  TP1 groupe b
S6   3 octobre      CM5  TD4  TP2 groupe a
S7  10 octobre      CM6  TD5  TP2 groupe b
S8  17 octobre      CM7  TD6  TP3 groupe a
S9  24 octobre      CM8  TD 7 TP3 groupe b
S10  7 novembre     CM9  TD8  TP4 groupe a
S11 14 novembre     CM10 TD9  TP4 groupe b
S12 21 novembre          TD10 TP5 groupe a
S13 28 novembre rattrapages   TP5 groupe b
S14  5 décembre rattrapages
```

Ecampus

les différentes ressources pour les CM, TD et TP sont disponibles sur ecampus.

Définition d'un algorithme

Définition informelle

- suite finie d'instructions réalisées dans un ordre fixé pour atteindre un but.

Algorithme $\neq$ programme

- l'algorithme doit être indépendant du langage de programmation

Langage algorithmique

- la maîtrise d'un langage algorithmique (ou pseudo-langage) est nécessaire afin d'obtenir cette indépendance

Structures de données

- la conception d'un algorithme dépend de la façon dont sont représentées les données
- il faut choisir convenablement les structures de données

Qu'est ce qu'un bon algorithme ?

Bon algorithme vu du côté de certains étudiants

- ça donne le bon résultat
- le principal c'est que ça marche
- j'obtiens le même résultat que mon voisin
- pourquoi j'ai pas une bonne note ?

Qu'est ce qu'un bon algorithme ?
Bon algorithme vu du côté des enseignants

Les critères suivants sont essentiels

Efficacité

- complexité en temps (pire des cas, en moyenne)
- complexité en espace (mémoire allouée)

Compréhension

- méthode facile à comprendre
- pas d'obfuscation

Lisibilité

- code lisible
- code commenté
- nom des variables et des procédures significatif

Plan du CM1

Types

Variables et types

Les données sont représentées par des variables

- toute variable possède un nom/identifiant qui désigne un emplacement mémoire
- elle a un type qui détermine la place (le nombre d'octets) que cette variable occupe en mémoire

Types de base et types composés

- types de base ou simples (disponibles dans le langage)
  - booléen
  - entier
  - réel
  - caractère et chaîne de caractères
  - ...
- types composés (créés par l'utilisateur)
  - structure ou enregistrement
  - tableau
  - liste chaînée
  - arbre
  - ...

Typage statique

Définition

- le type est défini au départ
- avantage : offre une plus grande sûreté.
  - une erreur de type peut être détectée lors de la compilation
- désavantage : nous avons moins de liberté
- on retrouve le typage statique dans des langages compilés
  - C, C++, java, ...

Exemple en C

```
int foisDeux(int x){
  return 2*x;
    }
int main(){
  int a = 5;
  char * b = "Bonjour";
  printf("%d ",foisDeux(a));
  printf("%s ",foisDeux(b));
}
```

- dans la dernière instruction, b n'est pas de type int
- l'erreur de type est détectée à la compilation

Typage dynamique

Définition

- le typage est réalisé à la volée
- avantage : programmation plus souple
- désavantage : peut entrainer des erreurs selon le type utilisé
- on retrouve le typage dynamique dans des langages interprétés
  - python, php, ruby,...

Exemple en python

```
def foisDeux(x):
    return 2*x
a = 10
b = "Bonjour"
print(foisDeux(a))#on affiche 20
print(foisDeux(b))#on affiche BonjourBonjour
```

- x peut être de n'importe quel type
- il faut juste que l'opération $2 * x$ soit définie
- a étant un int, $2 * a$ est la multiplication sur les entiers
- b étant une chaîne de caractères, $2 * b$ est la duplication de b

Pour notre langage algorithmique, nous considérerons un typage statique

Langage algorithmique – opérateurs

Pour manipuler ces données, on dispose des opérateurs suivants

- opérateurs arithmétiques
  - $+$ l'addition
  - $-$ la soustraction ou la négation
  - $*$ la multiplication
  - $/$ la division
  - *div* le quotient de la division euclidienne
  - *mod* le reste de la division euclidienne
- opérateurs de comparaison
  - $=$ l'égalité
  - $<$ strictement inférieur
  - $<=$ inférieur ou égal
  - $>$ strictement supérieur
  - $>=$ supérieur ou égal
  - $<>$ ou $!=$ différent
- opérateurs booléens
  - et
  - ou
  - non

Langage algorithmique – instructions (1)

Déclaration d'une variable

```
a : entier
mot : chaîne de caractères
```

Séparations

- indentation comme en python pour définir un bloc d'instructions
- pas d'utilisation des mots clefs begin et end, ni de séparateurs comme les accolades

Suite d'instructions

On peut écrire plusieurs instructions sur une même ligne avec des ;

```
a : entier ; mot : chaîne de caractères
```

Langage algorithmique – instructions (2)

Affectation

- l'instruction d'*affectation* est notée $=$
- La variable doit d'abord être déclarée
- l'espace mémoire réservé dépend du type de la variable

```
a : entier
mot : chaîne de caractères
a = 10
mot = "Bonjour"
```

Instructions sur une ligne

```
a : entier ; mot : chaîne de caractères ; a = 10 ; mot = "Bonjour"
```

Affectations en parallèle comme en Python

```
a, b = 2, 3   équivaut à la suite d'instructions a = 2 ; b = 3
a, b = b, a équivaut à la suite d'instructions c = a ; a = b ; b = c
```

Langage algorithmique – instructions (3)

Lecture et affichage

- lire *x*
- afficher *x*

Condition

```
si a < b alors
    afficher a
sinon
    afficher b
```

Remarque : la partie *sinon*... est facultative.

Il peut donc n'y avoir aucune instruction à exécuter lorsque la condition n'est pas vérifiée.

Langage algorithmique – instructions (4)

On dispose des trois types de boucle suivants :

Tant que

```
x : entier ; x = 1
Tant que x > 0 faire
     afficher x
     lire x
```

Répéter jusqu'à

```
x : entier ; lire x
Répéter
   afficher x
   x = x div 2
jusqu'à x mod 2 = 1
```

Pour i de a à b faire (par pas de 1)

```
n, res : entier ; n = 57 ; res = 1
pour i de 2 à n faire
     res = res * i
afficher res
```

Langage algorithmique – instructions (5)

Exemple de programme en langage algorithmique

```
prenom : chaîne de caractères
a,b : entier
afficher "Entrez votre prénom"
lire prenom
a,b = -1,1
tant que a*b < 0 faire
   si a < b alors
      affichez a, b
   sinon affichez b, a
   afficher prenom, "entrez deux entiers"
   lire a,b
```

Plan du CM1

Les procédures ou fonctions

Procédure

La procédure ne retourne pas de valeur.

```
affichePuissanceDeux (n : entier)
   res : entier ; res = 1
   pour i de 1 à n faire
       afficher res
       res = res*2
```

Fonction

- La fonction retourne une valeur
- On précise dans la signature le type de la valeur retournée

```
puissanceDeux (n : entier) : entier
   res : entier ; res = 1
   pour i de 1 à n faire
       res = res*2
   retourner res
```

Les procédures ou fonctions
Passage par valeur

Passage par valeur

les paramètres d'une procédure sont passés par valeur.

Cela a pour conséquence que toute variable qui est paramètre d'une procédure qu'on appelle dans un programme n'est jamais modifiée par l'appel (l'exécution) de la procédure.

Exemple

```
pluscinq(n : entier) : entier
   n = n + 5
   retourner n

m : entier ; m = 3
afficher pluscinq(m) // on affiche 8
afficher m           // on affiche 3
```

- une nouvelle variable locale m' à la fonction `pluscinq` prend la valeur de m et est augmentée de 5.
- c'est cette variable m' qui est retournée
- elle a une portée limitée à l'exécution de `pluscinq`

Fonction mathématique et fonction informatique

Fonction informatique vs fonction mathématique

- une fonction mathématique spécifie le résultat retourné
  Par exemple, $f(x) = x^2$
- une fonction informatique donne la méthode (le programme) pour arriver au résultat
- Il existe plusieurs façons de calculer x^2
- une fonction mathématique correspond à plusieurs fonctions informatiques

Convention

Afin de ne pas faire de confusion entre une fonction mathématique et une fonction informatique, nous essaierons de parler lors des CM et TD de

- procédure pour les procédures et les fonctions informatiques
- fonction pour les fonctions mathématiques

Procèdures récursives

Une procédure récursive est une procédure qui s'appelle elle-même [1].

Fonction factorielle

```
factorielleRec (entier n) : entier
   Si n = 0 ou n = 1 alors //condition terminale
      retourner 1
   Sinon retourner n*factorielleRec(n-1)
```

Récursif $\Longleftrightarrow$ itératif

- tout algorithme récursif peut être réécrit en algorithme itératif et vice-versa.
- on préférera un algorithme itératif lorsque ce n'est pas plus difficile à écrire

```
factorielleIt (entier n) : entier
   res, i : entier ; res = 1 ;
   Pour i de 2 à n faire//on rentre dans la boucle lorsque n >= 2
      res = res*i
   retourner res
```

1. cas de la récursivité directe

Récursivité terminale et non terminale

Récursivité terminale

Une procédure est récursive terminale lorsqu'il n'y a pas d'instruction après l'appel récursif.
Une telle procédure nécessite l'utilisation d'un argument supplémentaire appelé **accumulateur**.

```
factorielleRecT (n : entier, acc : entier) : entier
   Si n = 0 ou n = 1 alors //condition terminale
      retourner acc
   Sinon retourner factorielleRecT(n-1,acc*n)

afficher factorielleRecT(10,1)
```

Récursivité non terminale

Dans le cas contraire, la procédure est récursive non terminale.

```
factorielleRecNT (entier n) : entier
   Si n = 0 ou n = 1 alors //condition terminale
      retourner 1
   Sinon retourner n*factorielleRecNT(n-1)
```

L'exécution de cette procédure nécessite une pile.

Algorithme récursif ou itératif ?

Le choix dépend souvent de la structure de données

Structures linéaires

Pour les structures linéaires comme les tableaux et les listes chaînées, les algorithmes itératifs sont souvent mieux adaptés.

Structures arborescentes

- pour les structures arborescentes comme les arbres binaires, les arbres généraux et les ABR (arbres de recherche), les algorithmes récursifs sont mieux adaptés.
- pour obtenir des algorithmes itératifs, il faut généralement faire intervenir des structures auxiliaires comme les piles ou les files.

Consignes

- il faut respecter les consignes données en TD, TP et à l'examen terminal
- si ce n'est pas précisé vous pouvez choisir entre récursif et itératif

Plan du cours

Structures linéaires – les tableaux

Définition d'un tableau

- un tableau est constitué de cases
- `tab[i]` correspond au contenu de la case *i*
- pour un tableau de *n* cases, nous allons de la case 0 à la case $n-1$
- le contenu de toutes les cases sont d'un même type

Instructions élémentaires

```
tab[2] = 10//on affecte la valeur 10 à la case 2
x : entier;x = tab[5]//on affecte à x la valeur de la case 5
```

Structures linéaires – les tableaux

Déclaration d'un tableau

```
tab[tailleMax] : tableau de <type>

tab[1000] : tableau d'entiers
```

Remarques

- le typage est statique, comme dans le langage C
- tailleMax doit être une constante, pas une variable
- tailleMax permet de définir l'allocation mémoire, pas le nombre d'éléments du tableau
- on utilise souvent une variable taille pour mémoriser le nombre d'éléments que l'on a rentrés.

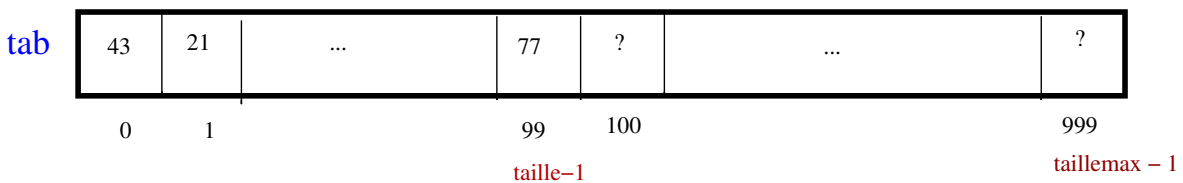
Structures linéaires – les tableaux

Remplissage d'un tableau

```
remplissageTableau(tab : tableau d'entiers, taille : entier)
   Pour i de 0 à taille-1 faire
      tab[i] = i

tab[1000] : tableau d'entiers
remplissage(tab,100)
```

Ici tailleMax vaut 1000 et taille vaut 100.
Le tableau peut contenir 1000 entiers, mais il n'en contient actuellement que 100.



Affichage d'un tableau

```
affichageTableau(tab : tableau d'entiers, taille : entier)
   Pour i de 0 à taille-1 faire
      afficher tab[i]

affichageTableau(tab,50)
```

Recherche dans un tableau

Recherche complète

Pour trouver le maximum du tableau, il faut obligatoirement parcourir tout le tableau.

```
maxTableau(tab : tableau d'entiers, n : entier) : entier
   maxi : entier ; maxi = 0
   Pour i de 0 à n-1 faire
      Si tab[i] > tab[maxi] alors maxi = i
   retourner maxi
```

Coût de la procédure

- on définit le coût comme le nombre de comparaisons entre deux entiers
- ici le coût est égal à n
- le coût est toujours le même quelque soit le contenu du tableau.
- il ne dépend que de la variable n.

Recherche dans un tableau

Recherche conditionnelle

On parcourt le tableau jusqu'à trouver un zéro.
On retourne -1 lorsque le tableau ne contient pas de zéro.

```
premierZero(tab : tableau d'entiers, n : entier) : entier
   i : entier ; i = 0
   Tant que i < n et tab[i] <> 0 faire
      i = i + 1
   Si i = n alors retourner -1
   Sinon retourner i
```

Coût de la procédure

- comme précédemment, on définit le coût comme le nombre de comparaisons entre deux entiers
- le coût varie de 1 à n
- il dépend de la position du premier zéro

Tableau - remplissage contigu

Objectif

- on souhaite mettre n éléments (ici entiers) dans un tableau
- on peut insérer ou supprimer un élément, donc n varie.
- on veut une représentation contiguë
  - les éléments sont donc regroupés de la case 0 à la case $n - 1$.

Structure de tableau contigu

```
structure tableauContigu
   tableau[tailleMax] : tableau d'entiers
   taille : entier
```

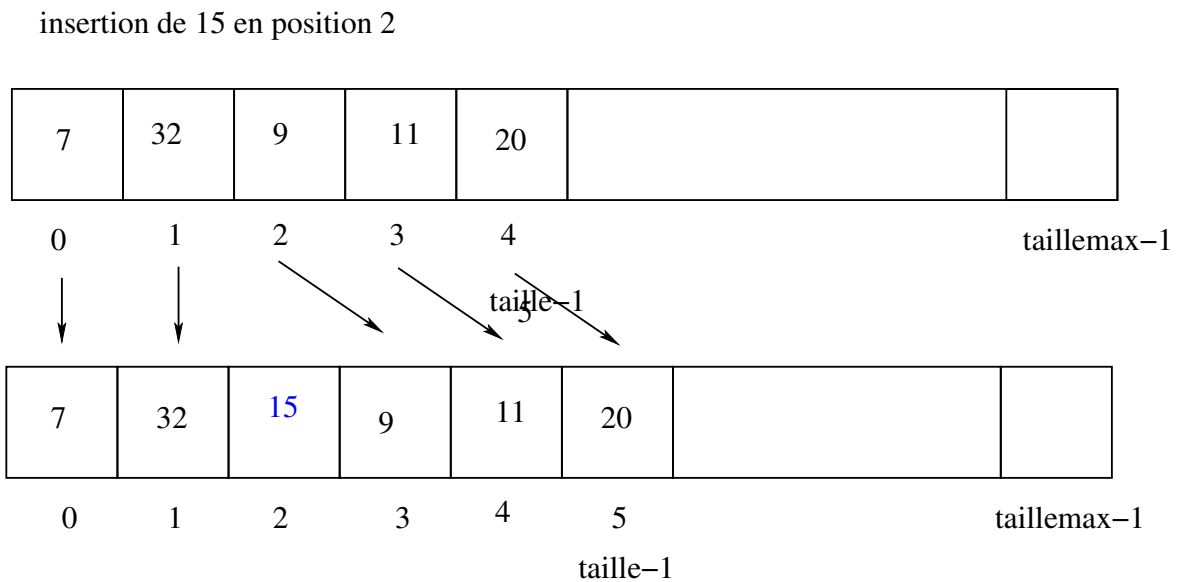
Les structures seront vues plus en détail au CM 2.

On rappelle que *tailleMax* est une constante, pas une variable.

Insertion dans un tableau contigu

Insertion de la valeur *x* à la case *k* – méthode

- on décale à droite les éléments qui doivent se retrouver après *x*
- on insère l'élément *x* à la case *k*
- on incrémente la taille



Insertion dans un tableau contigu

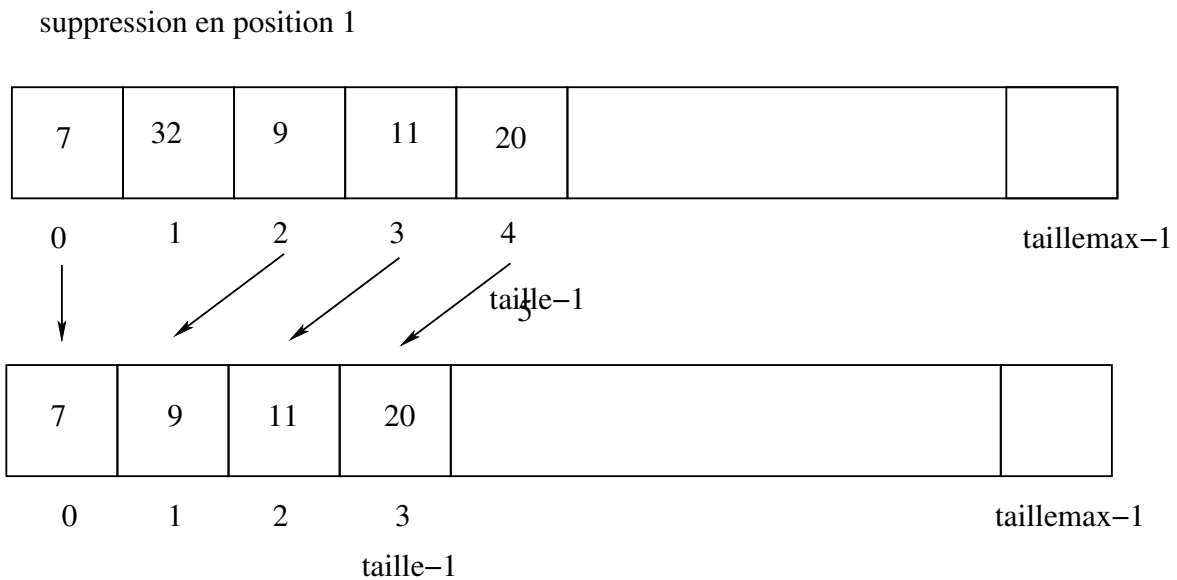
Insertion de la valeur *x* à la case *k* – procédure

```
insérerTableau( tab : tableauContigu, k : entier, x : entier)
   si k > tab.taille alors
      afficher "impossible d'accéder à la case " k
   si tab.taille = tailleMax alors
      afficher "Débordement"
   sinon
        pour i de T.taille à k+1 par pas de -1 faire
           tab.tableau[i] = tab.tableau[i-1]
        tab.tableau[k] = x
        tab.taille = tab.taille + 1
```

Suppression dans un tableau contigu

Suppression du contenu de la case *k* – méthode

- on décale à gauche les éléments se trouvant après l'élément à supprimer
- on décrémente la taille



Suppression dans un tableau contigu

Suppression du contenu de la case *k* – procédure

```
suppressionTableau(tab : tableauContigu, k : entier)
    si k >= tab.taille alors
      afficher "impossible d'accéder à la case " k
    sinon pour i de k+1 à tab.taille faire
            tab.tableau[i-1] = tab.tableau[i]
          tab.taille = tab.taille - 1
```