Algorithmique et structures de données

CM 1

Langage algorithmique et tableaux

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 le problème à résoudre

Structures linéaires

Caractéristiques générales

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

Exemples de structures linéaires

- tableaux structures dynamiques, accès direct
- 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 ensemble 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

exemples de structures de recherche

- arbres binaires de recherche
- arbres des préfixes ou tries
- tables de hachage

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*

Contrôle des connaissances

- Contrôle continu : un devoir, 1/3
- Examen terminal de 2h, 2/3 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
- langage de programmation
  - L2 Info et L2 Maths Programmation en C et en python
  - L2 MIASHS Programmation en python

Organisation du cours

Planning

S1	2 septembre	CM1			
S2	9 septembre	CM2	TD1		
S3	16 septembre	CM3	TD2	TP1 groupe a	
S4	23 septembre	CM4	TD3	TP1 groupe b	
S5	30 septembre	CM5	TD4	TP2 groupe a	
S6	7 octobre	CM6	TD5	TP2 groupe b	
S7	14 octobre	CM7	TD6	TP3 groupe a	
S8	21 octobre		TD7	TP3 groupe b	
S9	4 novembre	CM8	TD8	TP4 groupe a	
S10	18 novembre	CM9	TD9	TP4 groupe b	
S11	25 novembre	CM10	TD10	TP5 groupe a	
S12	1 décembre			TP5 groupe b	
S13	8 décembre	semaine de rattrapage			

- Planning garanti pour le L2 Info
- Pour le L2 Maths et le L2 MIASH voir le responsable de l'année

Ecampus

Documents pour les CM, TD et TP disponibles sur ecampus.

Définition d'un algorithme

Définition

Suite finie d'instructions réalisées dans un ordre fixé pour arriver à un résultat

Algorithme $\neq$ programme

L'algorithme doit être indépendant du langage de programmation

Langage algorithmique

Un langage algorithmique (ou pseudo-langage) est nécessaire pour avoir cette indépendance

Structures de données

La conception d'un algorithme dépend de la façon dont sont représentées les données

$\Longrightarrow$ il faut choisir convenablement les structures de données

Qu'est ce qu'un bon algorithme ?

Bon algorithme vu du côté des étudiants

- ça donne le bon résultat
- le principal c'est que ça marche
- j'ai 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 est 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
  - ...

Typage statique

Définition

- le type est défini au départ
- avantage plus grande sûreté.
  $\Longrightarrow$ une erreur de type peut être détectée lors de la compilation
- Pour des langages compilés comme 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));
}
```

- 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 produire des erreurs de type
- 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 est un int, $2 * a$ est la multiplication sur les entiers
- b est une chaîne de caractères, $2 * b$ est la duplication de b

Pour notre langage algorithmique, nous prendrons un typage statique

Langage algorithmique – données

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

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 (espace mémoire réservé en fonction du type)

```
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 n'y avoir aucune instruction lorsque la condition n'est pas vérifiée.

Langage algorithmique – instructions (4)

On dispose des trois boucles suivantes :

Tant que

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

Répéter jusqu'à

```
x : entier ; x = 29800
Répéter
   afficher x
   x = x div 2
jusqu'à x mod 2 = 1
```

Pour i de a à b faire

```
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 ou plusieurs valeurs.

```
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 m' dans 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

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

Convention

Afin de ne pas confondre entre une fonction mathématique et une fonction informatique, nous essaierons de parler 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

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

Récursif $\Longleftrightarrow$ itératif

Tout algorithme récursif peut être réécrit en algorithme itératif et vice-versa.
On préférera un algorithme itératif lorsque c'est possible (pas plus difficile à écrire).

```
factorielle (entier n) : entier
   res, i : entier ; res = 1 ; i = 2
   Tant que i <= n faire //on rentre dans la boucle pour n >= 2
      res = res*i
      i = i + 1
   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.
Dans ce cas, le compilateur ou l'interpréteur peut optimiser l'exécution de la procédure.

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

Récursivité non terminale

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

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

Ici la multiplication s'effectue après l'appel récursif.
Il faut mémoriser les instructions qui seront effectuées après cet appel (voir TD).

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 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 fait généralement faire intervenir des piles ou des files.

Choisir entre un algorithme récursif et un algorithme itératif

Il faut suivre les consignes données en TD, TP et à l'examen terminal !

Plan du cours

Structures linéaires – les tableaux

Définition 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.

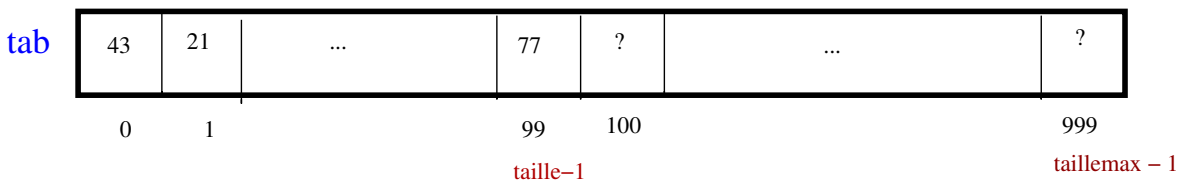
Structures linéaires – les tableaux

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 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 parcourir tout le tableau.

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

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, taille : entier) : entier
   i : entier ; i = 0
   Tant que i < taille et Tab[i] <> 0 faire
      i = i + 1
   Si i = taille alors retourner -1
   Sinon retourner i
```

On parcourt tout le tableau lorsque le tableau ne contient pas de zéro ou lorsque le premier zéro est en dernière position.

Tableau - remplissage contigu

Objectif

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

Structure de tableau contigu

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

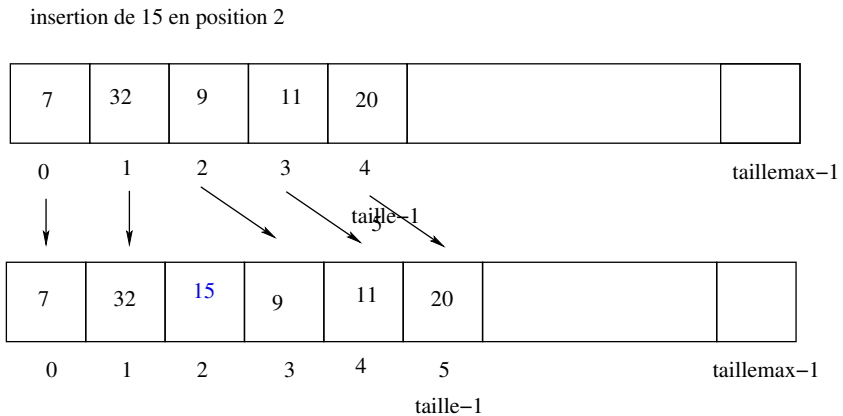
Les structures seront vues au CM 2.

On rappelle que *tailleMax* est une constante.

Insertion dans un tableau contigu

Insertion – méthode

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



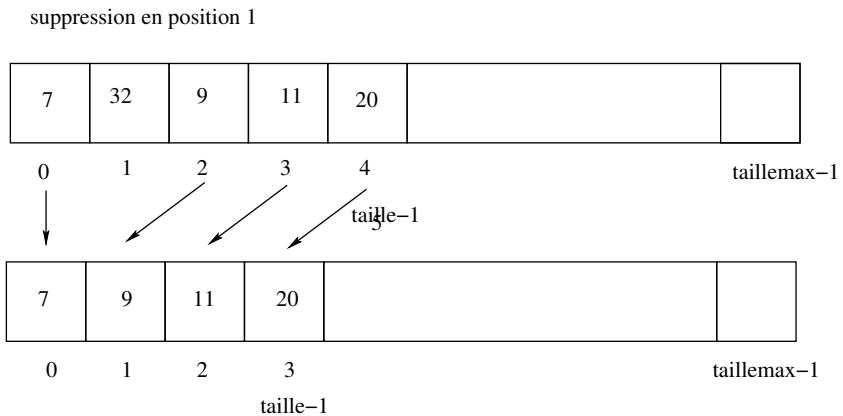
Insertion – procédure

```
insérerTableau( T : tableauContigu, k : entier, x : entier)
   si T.taille = tailleMax alors
      afficher "Débordement"
   sinon
        pour i de k+1 à T.taille faire
           T.tableau[i] = T.tableau[i-1]
        T.tableau[k] = x
        T.taille = T.taille + 1
```

Suppression dans un tableau contigu

Suppression – méthode

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



Suppression – procédure

```
suppressionTableau(T : tableauDynamique, k : entier, x : entier)
    si T.taille = 0 alors
       afficher "Suppression impossible, le tableau est vide"
    sinon pour i de k+1 à T.taille faire
             T.tableau[i-1] = T.tableau[i]
          T.taille = T.taille - 1
```

Tableau contigu
Pile et file

Implémentation d'une file ou d'une pile

Nous verrons que cette structure de tableau contigu permet d'implémenter une file ou une pile avec un tableau.