

Algorithmique et structures de données

Aziz Moukrim

aziz.moukrim@utc.fr

Pré-requis

- Témoignages

- « ... Honnêtement, je me suis "réveillée" après le médian, et je regrette de ne pas avoir mieux écouté la première partie car après le médian, mes notes de cours étaient complètes et détaillées. »

- Révisions

- Théorie des ensembles : intersection, union, inclusion, complémentaire, partition, produit cartésien, etc.
- Séries numériques : arithmétique, géométrie
- Relations d'équivalence
- Relations d'ordre
- Notion d'ensemble dénombrable

Fonctionnement

- Evaluation : Final (50%), médian(30%), TP(20%)
- Alternances : groupes A et B
- TD :
 - 5 séances
 - Corrigé(s) de médian
 - 6 séances
 - Corrigé(s) de final
 - Les sujets de médian corrigés ne seront pas nécessairement les mêmes : travail intra et inter groupes !
- TP :
 - TP3 : listes chaînées
 - TP4 : arbres binaires
 - Examen machine

Références

- Brassard, P. Bratlez, "Algorithmique, conception et analyse", Masson, 1987
- Les Goldschlager, Andrew Lister, "Informatique et algorithmique", InterEdition, Paris, 1986
- B. Meyer, C. Baudoin, « Méthodes de programmation », Eyrolles, 1984
- J. Courtin, I. Kowavski, "Initiation à l'algorithmique et aux structures de données", Tome 1, 2, 3, Dunod, 1989
- **Robert Sedgewick, “Algorithms”, Addison-Wesley, 1988**
- **D. Beauquier, J. Berstel, Ph. Chrétienne “Eléments d’algorithmique”, Masson, 1992**
- **T. Cormen, C. Leiserson, R. Rivest “Introduction à l’algorithmique”, Dunod, 1994**
- Brassard, P. Bratlez, "Algorithmique, conception et analyse", Masson, 1987
- Baynat, Chrétienne, Hanen, Kedad-Sidhoum, Munier-Kordon et Picouleau (2003) Exercices et problèmes d'algorithmique (Dunod)

Algorithmique et structures de données

NF16

Analyse des algorithmes

Analyse des algorithmes

- Exemples :
 - P1 : Multiplication de deux nombres
 - P2 : Problème des tours de Hanoï
 - P3 : Problème du voyageur de commerce
 - P4 : Problème du meilleur coup à jouer aux échecs
 - P5 : stratégie gagnante Table / pièces pour deux joueurs

Analyse des algorithmes

- Spécifications d'un algorithme :
 - La spécification d'un algorithme décrit ce que l'algorithme fait, sans détailler comment il le fait.
- Exemple :
 - {Ce programme recherche la place d'un élément X dans une liste L . Si X n'est pas dans la liste le résultat est zéro. La liste est représentée par un tableau. Elle comporte n éléments.}
 - $j:=1$;
 - Tant que $(j \leq n)$ et $(L[j] \neq X)$ faire $j:=j+1$;
 - Si $j>n$ alors $j:=0$;

Analyse des algorithmes

- Cette spécification est imprécise :
 - Est-ce que la liste peut être vide ?
 - Que fait-on s'il y a deux occurrences de X ?
 - Quels sont les données et les résultats ?
- Spécification précise :
 - {Les données sont L et X. L est un tableau de n éléments qui représente une liste. Cette liste peut être vide ($n=0$). On recherche l'indice de l'élément X dans L. L'algorithme termine avec la variable j égale à l'indice de la première occurrence de X dans L, s'il y en a une. *Si X n'est pas dans la liste, la variable j vaut zéro à la fin de l'algorithme.*}

Analyse des algorithmes

- Performances croissantes des ordinateurs en rapidité et en taille-mémoire
- « coût » de l'algorithme
 - Nombre d'opérations élémentaires
 - Quantité de mémoire requise
- Certains problèmes peuvent demander des temps prohibitifs quels que soient l'algorithme « connu » et la machine utilisés
- Certains problèmes, par contre, demandent un temps prohibitif avec certains algorithmes de résolution et « raisonnable » avec d'autres.
 - Il faut pouvoir comparer les performances de différents algorithmes disponibles pour résoudre un problème particulier.
 - Comment évaluer de telles performances ?
 - Ces évaluations ne devraient dépendre ni du langage ni de la machine utilisés

Analyse des algorithmes

- Définition : Un algorithme est un ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations
- Instance : Une instance d'un problème est définie par la fourniture des diverses valeurs attendues
 - 58×36 est une instance du problème P1
- Taille du problème : il existe toujours une donnée entière positive n caractérisant le volume de données manipulées :
 - Nombre de disques de la tour de Hanoï
 - Nombre d'éléments à trier par un algorithme de tri

Analyse des algorithmes

- Complexité temporelle : on appelle complexité temporelle d'un algorithme une fonction mesurant le temps nécessaire à l'exécution de cet algorithme pour une instance de taille n
- Complexité spatiale : on appelle complexité spatiale d'un algorithme une fonction mesurant la place utilisée en mémoire par celui-ci pour une instance de taille n .

Comportement asymptotique

- Calcul de la complexité d'un algorithme : somme du
 - Nombre d'affectations,
 - Nombre d'opérations arithmétiques,
 - Nombre d'opérations logiques.
- Exemple :
 - Plus grand élément dans un tableau
 - Recherche séquentielle d'un élément dans un tableau

Analyse d'un algorithme

- **Invariants de boucle**

Tant que C faire

 Inst;

Fin tantque

Une propriété A est un invariant de boucle ssi

-- A est vraie avant d'entrer dans la boucle

-- Si A et C sont vraies au début d'une itération k alors A est encore vraie à la fin de l'itération k

Donc à l'arrêt de la boucle A et (non C) sont vraies.

Analyse d'un algorithme

Invariants de boucle. Un invariant de boucle est souvent de la forme « telle variable admet pour valeur telle quantité (associée au problème étudié ».

Exemple. Trier un tableau T de taille n : $T[1 \dots n]$.

Algorithme Tri_par_selection ($T[1 \dots k]$);

var y : ...

Debut

Tant que $k > 1$ faire

// invariant : le sous-tableau $T[k+1 \dots n]$ est trié

$y := \text{le_plus_grand_element_de } T[1 \dots k]$;

échanger y et $T[k]$;

$k := k - 1$;

fin_tant_que

Fin

Analyse d'un algorithme

- **Conditions d'arrêt**

Une quantité de contrôle m restant ≥ 0

- montrer que $(m \geq 0)$ est un invariant de la boucle
- montrer que chaque exécution de la boucle fait décroître m au moins de 1.

Cette quantité m prend donc des valeurs entières, strictement décroissantes, minorées par zéro : cette suite de valeurs est finie. Il en est de même du nombre d'exécution de la boucle.

Complexité

- Soit D_n l'ensemble des données de taille n
- $\text{Cout}(d)$ la complexité en temps (affectations, opérations) sur la donnée d
- Complexité dans le meilleur des cas :
 - $\text{Min}(n) = \min \{ \text{cout}(d), d \in D_n \}$
- Complexité dans le pire des cas :
 - $\text{Max}(n) = \max \{ \text{cout}(d), d \in D_n \}$

Comportement asymptotique

- Paramètres ne pouvant influencer que d'un facteur multiplicatif :
 - Machine,
 - langage,
 - compilateur,
 - génie du programmeur.

Comportement asymptotique

10^6 opérations /seconde

	10	20	30	40	50	60
n	.00001s	.00002s	.00003s	.00004s	.00005s	.00006s
n^2	.0001s	.0004s	.0009s	.0016s	.0025s	.0036s
n^3	.001s	.008s	.027s	.064s	.124s	.216s
n^5	.1s	3.2s	24s	1.7mn	5.2mn	13mn
2^n	.001s	1s	18mn	13j	36a	366s.
3^n	.06s	58mn	6.5a	3800s.	10^8 s.	10^{13} s.
$n!$	3.6s	770s.	10^{17} s.	10^{32} s.	10^{49} s.	10^{66} s.

Comportement asymptotique

$f(n)$	Avec les ordinateurs d'aujourd'hui	100 fois plus rapide	1000 fois plus rapide
n	N1	100 N1	1000 N1
n^2	N2	10 N2	31.6 N2
n^3	N3	4.64 N3	10N3
2^n	N4	N4 + 6.64	N4 + 9.97
3^n	N5	N5 + 4.19	N5 + 6.29

Tableau rapportant la taille des problèmes résolus avec une heure de calcul.

19

Comportement asymptotique

- Grandes valeurs du paramètre n qui mesure la taille d'une instance d'un problème
- **Complexité dans le pire cas : pour l'instance pour laquelle notre algorithme fonctionne le moins bien**

Comportement asymptotique

- La notation O (« de l'ordre de » ou « grand O ») définit une borne asymptotique supérieure.
- Pour une fonction $f(n)$ donnée, on note $O(f(n))$ l'ensemble des fonctions tel que :
 - $O(f(n)) = \{T(n) : \text{il existe des constantes strictement positives } c \text{ et } n_0 \text{ telles que } 0 \leq T(n) \leq c.f(n) \text{ pour tout } n \geq n_0\}$
- Exemples :
 - $f(n) = n^2$ et $T(n) = 13n^2 + 7n + 11$
 - $f(n) = n^2$ et $T(n) = 1/2n^2 - 3n$

Comportement asymptotique

- Règle du maximum :
Si $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ alors $O(f(n)+g(n)) = O(\max(f(n), g(n)))$
- La relation « $\in O$ » est réflexive et transitive
- Si $\lim_{n \rightarrow +\infty} f(n)/g(n) \in \mathbb{R}_+^*$ alors $f(n) \in O(g(n))$ et $g(n) \in O(f(n))$
- Si $\lim_{n \rightarrow +\infty} f(n)/g(n) = 0$ alors $f(n) \in O(g(n))$ mais $g(n) \notin O(f(n))$
- Si $\lim_{n \rightarrow +\infty} f(n)/g(n) = +\infty$ alors $g(n) \in O(f(n))$ mais $f(n) \notin O(g(n))$

Comportement asymptotique

- Un algorithme de l'ordre de $n \log(n)$ est dans $O(n^2)$, dans $O(n^3)$, etc...
- La notation Ω définit une borne asymptotique inférieure.
- Pour une fonction $f(n)$ donnée, on note $\Omega(f(n))$ l'ensemble des fonctions tel que :
 - $\Omega(f(n)) = \{T(n) : \text{il existe des constantes strictement positives } c \text{ et } n_0 \text{ telles que } 0 \leq cf(n) \leq T(n) \text{ pour tout } n \geq n_0\}$

Comportement asymptotique

- La notation Θ définit l'ordre exact d'une fonction.
- Pour une fonction $f(n)$ donnée, on note $\Theta(f(n))$ l'ensemble des fonctions tel que :
 - $\Theta(f(n)) = \{T(n) : \text{il existe des constantes strictement positives } c_1, c_2 \text{ et } n_0 \text{ telles que } 0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n) \text{ pour tout } n \geq n_0\}$
- Exemples :
 - $f(n) = n^2$ et $T(n) = 1/2n^2 - 3n$

Comportement asymptotique

- Analyse du temps d'exécution d'un algorithme
 - Affectation ou opération élémentaire : $O(1)$
 - Instruction composée :
 - Si $T_1(n) \in O(f_1(n))$ et $T_2(n) \in O(f_2(n))$ avec $f_2(n) \in O(f_1(n))$ alors $T_1(n)+T_2(n)$ est en $O(f_1(n))$.
 - Bloc d'instructions : règle de sommation
 - « Si »
 - Si $O(f_{\text{Si}})$ et $O(f_{\text{Sinon}})$ sont des bornes supérieures des blocs « Si » et « Sinon » alors $O(\max\{f_{\text{Si}}(n), f_{\text{Sinon}}(n)\})$ est une borne supérieure du temps d'exécution de l'instruction « Si »
 - « Pour, Tant_que, repeter »
 - Si $O(f(n))$ est une borne supérieure du corps de la boucle et $O(g(n))$ est une borne supérieure du nombre d'itérations alors $O(f(n)g(n))$ est une borne supérieure de la boucle.

Récurtivité

- La définition d'un objet X est récursive si elle contient une référence à l'objet X lui même (récursivité directe) ou à un objet Y qui fait référence (directement ou indirectement) à l'objet X (récursivité croisée ou indirecte).
- Exemple : Un ascendant d'une personne est :
 - Soit son père
 - Soit sa mère
 - Soit un ascendant de son père ou de sa mère

Récurtivité

- On appelle profondeur de la récursivité à un instant donné de l'exécution de l'algorithme, le nombre d'appels imbriqués.
- Tout algorithme récursif doit contenir une condition qui assure la finitude du nombre d'appels emboîtés (cas trivial).
- Exemple :
 - $N!$
 - Tour de Hanoi
 - Suite de Fibonacci

Récurtivité

- Mécanisme de la récursivité :
 - pile des environnements
- Arbre des appels d'un programme récursif :
 - $N!$ $T(n) =$
 - Tour de Hanoi $T(n) =$
 - Suite de Fibonacci $T(n) =$

Récurtivité

- Calculs redondants
 - Méthode de marquage
 - Analyse du graphe des appels
- Algorithmes récursifs admettant une version itérative

Dérécursification

Algo Rec(N)

Début

Action a1;

Si $p(N)$ alors... //cas triviaux

Sinon

action a2 ;

$R := \dots$ //fonction de N

Rec(R) ;

//rien à exécuter

Fin si ;

Fin Rec

Algo Iter(N)

Début

Action a1;

Tant que not $p(N)$ faire

action a2 ;

$N := \dots$ //fonction de N

action a1;

Fin tant que ;

actions correspondant aux
cas triviaux ;

Fin Iter

Dérécursification

```
Algo Tri_par_selection_Rec (T[1...k]);  
  var y : ...  
  Debut  
    Si k=1 alors rien_a_executer  
    sinon  
      y := le_plus_grand_element_de T[1...k];  
      échanger y et T[k];  
      // ici T[k] est en place  
      Tri_par_selection_Rec (T[1...k-1]);  
    fin_si  
  Fin
```

Nous pouvons en déduire la version déjà vue plus haut.

Algorithmique et structures de données

NF16

Structures linéaires

Ensembles dynamiques

- Les ensembles manipulés par les algorithmes peuvent croître ou diminuer. On dit qu'ils sont dynamiques.
- Chaque élément d'un ensemble est représenté généralement par un objet dont l'un des champs est une clé servant d'identifiant et dont les autres champs contiennent les données satellites.

Opérations

- Rechercher (S, k) :
 - Etant donné un ensemble S et une valeur de clé k , retourne un pointeur x sur un élément de S tel que $cle[x] = k$ ou NIL si l'élément en question n'est pas dans S
- Insérer (S, x) :
 - Ajout de l'élément pointé par x à l'ensemble S
- Supprimer(S, x) :
 - Suppression de l'élément pointé par x de l'ensemble S
- Un dictionnaire est un ensemble dynamique supportant ces opérations.

Opérations

- Minimum(S)
- Maximum(S)
- Successeur(S, x)
- Prédécesseur(S, x)

Tableau

- Un tableau est un ensemble d'emplacements mémoire contigus en nombre **fixé** contenant le **même type** de donnée, chacun d'eux étant accessible via un indice.
- Exemples : vecteurs, matrices, ...

Tableau

- Recherche
 - $j:=1$;
 - Tant que $(j \leq n)$ et $(T[j] \neq X)$ faire $j:=j+1$;
 - Si $j>n$ alors $j:=0$;
- Supprimer, insérer, minimum, maximum, successeur, prédécesseur

Listes chaînées

- Une liste consiste en une liste d'éléments contenant un champ clé et un champ pointeur, qui pointe sur l'élément qui suit
- Cellule : donnée élémentaire, informations de chaînage
- création de cellule, gestion de la mémoire

Listes chaînées

- $\text{succ}[x]$: successeur de la cellule x
- $\text{pred}[x]$: prédécesseur de la cellule x
- $\text{tete}[L]$: première cellule de la liste L
- $\text{cle}[x]$: valeur de clé de la cellule x

Conventions

- Notion d'absence d'information : NIL
 - valeur particulière dans le système de gestion du chaînage des cellules, dépend du système de gestion de l'information choisi
- Quand une liste est vide $tete[L] = NIL$
 - cette convention peut être remplacée par une autre plus avantageuse (Cf. sentinelle)

Liste chaînée simple

- Recherche_liste(L, k)
 $x := \text{tete}[L]$
 Tant que $x \neq \text{NIL}$ et $\text{cle}[x] \neq k$
 faire $x := \text{succ}[x]$
 Retourner(x)
- Insérer_tete_liste(L, x)
 $\text{succ}[x] := \text{tete}[L]$
 $\text{tete}[L] := x$

Liste chaînée simple

- Insérer_queue_liste(L, y)

Si tete[L] = NIL

alors tete[L] := y

sinon

x := tete[L]

Tant que succ[x] ≠ NIL

faire x := succ[x]

succ[x] := y

Liste chaînée simple

- Insérer_trier_liste(L, z)

x := tete[L]

si (x = NIL) ou (cle[z] < cle[x])

alors Insérer_tete_liste(L, z)

sinon Tant que succ[x] ≠ NIL et cle[z] > cle[succ[x]]

 faire x := succ[x]

 succ[z] := succ[x]

 succ[x] := z

Liste chaînée simple

- Pred(x)

```
si x = tete[L]
    alors retourner(NIL)
sinon y := tete[L]
    Tant que succ[y] ≠ x
    faire  y := succ[y]
    retourner(y)
```

Liste chaînée simple

- Suppression(L, x)

y := pred(x)

si y = NIL

alors tete[L] := succ[x]

sinon succ[y] := succ[x]

Liste chaînée double

- Insérer_tete_liste(L, x)

succ[x] := tete[L]

si tete[L] \neq NIL

alors pred[tete[L]] := x

tete[L] := x

pred[x] := NIL

Liste chaînée double

- Supprimer_liste(L, x)

Si $\text{pred}[x] \neq \text{NIL}$

alors $\text{succ}[\text{pred}[x]] := \text{succ}[x]$

sinon $\text{tete}[L] := \text{succ}[x]$

Si $\text{succ}[x] \neq \text{NIL}$

alors $\text{pred}[\text{succ}[x]] := \text{pred}[x]$

Sentinelles

- Il est souvent pratique de définir un nœud factice appelé sentinelle qui remplace les valeurs NIL du prédécesseur de la tête ainsi que du successeur de la queue de la liste. Ainsi, une liste doublement chaînée avec sentinelle devient circulaire.
- Objectif : simplifier les conditions limites.

Sentinelles

- conventions :
 - $\text{succ}[\text{nil}[L]] = \text{tete}[L]$
 - $\text{pred}[\text{nil}[L]] = \text{queue}[L]$
 - $\text{succ}[\text{queue}[L]] = \text{nil}[L]$
 - $\text{pred}[\text{tete}[L]] = \text{nil}[L]$
- $\text{ListeInsérer}(L, x)$
- $\text{ListeSupprimer}(L, x)$

Tableau trié

- Recherche dichotomique :
 - Dicho(X, T, g, d, res)
 - {cette procédure récursive recherche par dichotomie l'élément X dans le tableau T[1...n] dont les éléments sont triés en ordre croissant ; le résultat de la procédure est contenu dans la variable res : c'est 0 si X n'appartient pas au tableau, et c'est $i \in \{1, \dots, n\}$ si X se trouve à l'indice i du tableau}

En exercice.

Tableau trié

- Recherche dichotomique :
 - Dicholter(X, T, res)
 - $g:=1$; $d:=n$
 - Tant que $g < d$ faire
 - debut
 - $m := (g+d) / 2$
 - If $X \leq T[m]$ alors $d := m$
 - sinon $g:=m+1$
 - Fin
 - Si $T[g]=X$ alors $res:=g$ sinon $res := 0$

Tableau trié

- Dicholter :
 - Invariant
 - La boucle se termine
- Recherche dichotomique : Complexité

Tableau trié

- Recherche par interpolation
 - Si les nombres forment une progression régulière
 - X sera recherché autour de la place
 - $p = g + ((d-g).(X-T[g]) \text{ div } (T[d]-T[g]))$
 - Si $X = T[p]$, on a terminé
 - Si $X < T[p]$, on recommence une recherche par interpolation de X sur $T[g] \dots T[p-1]$
 - Si $X > T[p]$, on recommence une recherche par interpolation de X sur $T[p+1] \dots T[d]$

Tableau trié

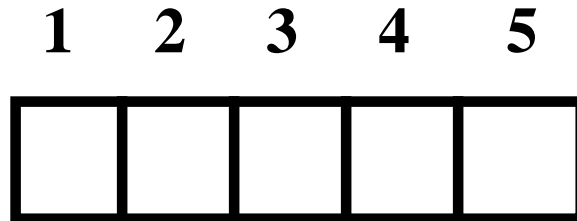
- Algorithmes et complexité :
 - Recherche
 - Insertion
 - Suppression

Piles (structure LIFO)

- Implantation par tableau
- Convention :
 - sommet[P] pointe sur le dernier élément ajouté et vaut 0 initialement
 - Longueur [P] (= MAX_P) donne la taille maximale de la pile
- Débordements par excès et défaut
- Algorithmes :
 - Créer_pile, Pile_vide, Pile_pleine, Empiler et Dépiler
 - Insérer

Pile

- Créer_Pile(MAX_P) crée et retourne une Pile de taille MAX_P éléments



Sommet [P] = 0 (initialement)

Pile

Pile_vide(P)

Si sommet[P] = 0
 alors retourner(vrai)
 sinon retourner(faux)

Fin_si

Pile_pleine(P)

Si sommet[P] = Longueur[P]
 alors retourner(vrai)
 sinon retourner(faux)

Fin_si

Pile

Empiler(P, x)

Si (Pile_pleine(P) = vrai)
 alors Erreur('débordement par excès')
 sinon sommet[P] := sommet[P] + 1
 P[sommet[P]] := x

Finsi

Pile

Dépiler(P)

Si Pile_vide(P)

alors Erreur('débordement par défaut')

sinon sommet[P] := sommet[P] - 1

Retourner(P[sommet[P] + 1])

Pile

Insérer(P, x)

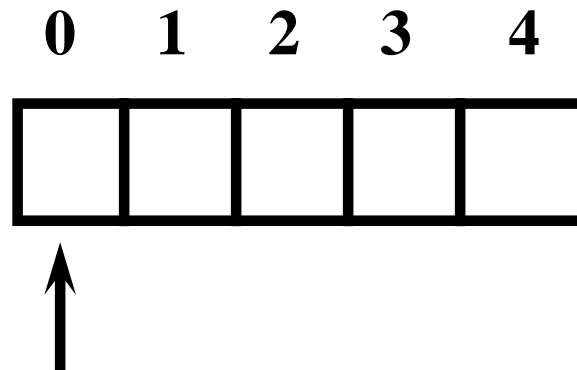
Si $\text{Pile_vide}(P)$ ou $x < P[\text{sommet}[P]]$
alors Empiler(P, x)
sinon $y := \text{Dépiler}(P)$
 Insérer(P, x)
 Empiler(P, y)

Files (structure FIFO)

- Implantation par tableau en file tournante
- Conventions :
 - Tête[F] pointe sur le premier élément à défiler
 - Queue[F] pointe sur le premier emplacement libre
 - Initialement $Tête[F] = Queue[F] = 0$
 - Longueur[F] retourne la taille maximale de F
 - On dit que la File est vide si $Tête[F] = Queue[F]$
 - On dit que la File est pleine si
 $Tête[F] = \text{modulo}(Queue[F] + 1, \text{longueur}[F])$
- Débordements par excès et défaut
- Algorithmes :
 - Créer_file, File_vide, File_pleine, Enfiler et Défiler

File

- Créer_file(MAX_F) crée et retourne une file de taille MAX_F éléments donc longueur[F] retournera la valeur MAX_F



Tête[F] = Queue[F] = 0

File

Enfiler(F, x)

```
Si   File_pleine(F)
alors Erreur('La file est pleine')
sinon   F[Queue[F]] := x
        Si Queue[F] = longueur[F] - 1
        alors   Queue[F] := 0
        sinon   Queue[F] := Queue[F] + 1
```

File

Défiler(F)

```
Si File_vide(F) = vrai
alors Erreur('La file est vide')
sinon    x := F[Tête[F]]
          Si Tête[F] = longueur[F] - 1
          alors    Tête[F] := 0
          sinon    Tête[F] := Tête[F] + 1
          Retourner(x)
```

Finsi

Algorithmique et structures de données

NF16

Structures arborescentes

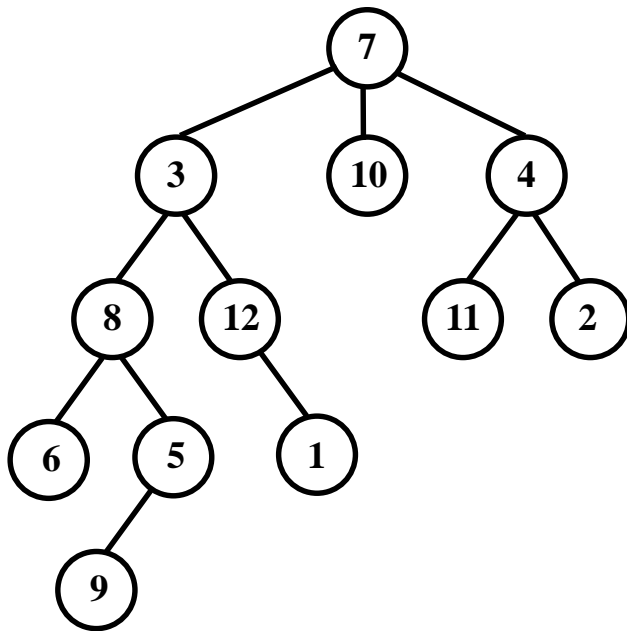
Définitions

- Graphe, sommets, arêtes
- Forêt : graphe non orienté acyclique
- Arbre : graphe non orienté connexe acyclique
- Arborescence : arbre orienté, chaque sommet (sauf un) a exactement un prédécesseur. Le sommet sans prédécesseur est la racine (r) de l'arborescence
- Ancêtre, Descendant, Père, fils, frères

Définitions

- Nœud externe (ou feuille) : nœud sans fils
- Nœud interne : n'est pas une feuille
- Degré d'un nœud x : nombre de fils de x
- Longueur d'un chemin = nombre d'arcs sur ce chemin
- Profondeur(x) : longueur du chemin entre la racine r et le nœud x
- Hauteur : plus grande profondeur d'un nœud

Définitions



- Hauteur = 4
- Profondeur de 7 est 0
- Profondeur de 3, 10 et 4 est 1

Définitions

- Arbre binaire : aucun nœud (arbre vide) ou un nœud racine et un sous-arbre gauche binaire et un sous arbre droit binaire
- Arbre k-aire : aucun nœud (NIL) ou un nœud racine et au plus k sous-arbres k-aires
- Arbre complet d'arité k : arbre d'arité k pour lequel toutes les feuilles ont la même profondeur et tous les nœuds internes ont pour degré k
- Nombre de nœuds internes d'un arbre binaire complet de hauteur h vaut $2^h - 1$.

Définitions

- Arbre parfait : Arbre binaire dont les feuilles sont situées sur deux niveaux au plus, l'avant dernier niveau est complet, et les feuilles du dernier niveau sont regroupées le plus à gauche possible
- Les informations (data) gérées dans un arbre peuvent comprendre une clé
- Les nœuds peuvent avoir une étiquette (nom, label) qui peut être une valeur numérique ou alphanumérique

Parcours d'arbres binaires

- Parcours d'un arbre : accéder une fois et une seule fois à tous les nœuds de l'arbre
- Parcours_Prefixe(x)
 - Si $x \neq \text{nil}$ alors
 - Imprimer clé[x]
 - Parcours_Prefixe(gauche[x])
 - Parcours_Prefixe(droit[x])

Parcours d'arbres binaires

- **Parcours_Postfixe(x)**
 - Si $x \neq \text{nil}$ alors
 - **Parcours_Postfixe(gauche[x])**
 - **Parcours_Postfixe(droit[x])**
 - Imprimer clé[x]
- **Parcours_Infixe(x)**
 - Si $x \neq \text{nil}$ alors
 - **Parcours_Infixe(gauche[x])**
 - Imprimer clé[x]
 - **Parcours_Infixe(droit[x])**
- **Versions non récursives : Piles et Files (voir TD/TP)**

Arbres Binaires de Recherche

- Arbre Binaire de Recherche
 - Soit x est nœud d'un ABR. Si y est un nœud du sous-arbre gauche de x , alors $\text{clé}[y] < \text{clé}[x]$. Si y est un nœud du sous-arbre droit de x , alors $\text{clé}[x] < \text{clé}[y]$.
- ABR_Rechercher(x , k)
 - Si $x = \text{nil}$ ou $k = \text{clé}[x]$ alors retourner x
 - Si $k < \text{clé}[x]$ alors retourner ABR_Rechercher(gauche[x], k)
sinon retourner ABR_Rechercher(droit[x], k)

Arbres Binaires de Recherche

- ABR_Rechercher_Itératif(x, k)
 - Tant_que $x \neq \text{nil}$ et $k \neq \text{clé}[x]$ faire
 - Si $k < \text{clé}[x]$ alors $x := \text{gauche}[x]$
sinon $x := \text{droit}[x]$
 - Retourner (x)
- ABR_Minimum(x)
 - Tant_que $\text{gauche}[x] \neq \text{nil}$ faire
 - $x := \text{gauche}[x]$
 - Retourner (x)

Arbres Binaires de Recherche

- ABR_Maximum(x)
 - Tant_que droit[x] \neq nil faire
 - $x := \text{droit}[x]$
 - Retourner (x)
- ABR_Successeur(x)
 - Si droit[x] \neq nil alors retourner ABR_Minimum(droit[x])
 - $y := \text{pere}[x]$
 - Tant_que $y \neq \text{nil}$ et $x = \text{droit}[y]$ faire
 - $x := y$
 - $y := \text{pere}[y]$
 - Retourner (y)

Arbres Binaires de Recherche

- ABR_Insérer(T, z)
 - $y := \text{nil}$
 - $x := \text{racine}[T]$
 - Tant_que $x \neq \text{nil}$ faire
 - $y := x$
 - Si $\text{clé}[z] < \text{clé}[x]$ alors $x := \text{gauche}[x]$
sinon $x := \text{droit}[x]$
 - $\text{Pere}[z] := y$
 - Si $y = \text{nil}$ alors $\text{racine}[T] := z$
sinon si $\text{clé}[z] < \text{clé}[y]$ alors $\text{gauche}[y] := z$
sinon $\text{droit}[y] := z$

Arbres Binaires de Recherche

- Suppression d'un nœud z d'un ABR
 - Cas 1 : z n'a pas de fils
 - Détacher z en modifiant son père
 - Cas 2 : z n'a qu'un seul fils
 - On détache z en créant un nouveau lien entre son fils et son père
 - Cas 3 : z possède deux fils
 - On détache le successeur de z , noté y
 - On remplace la clé de z par celle de y

Algorithmique et structures de données

NF16

Arbres équilibrés

Arbres équilibrés : AVL

- Construire des ABR :
 - 1, 2, 3, 4, 5, 6, 7
 - 7, 6, 5, 4, 3, 2, 1
 - 4, 2, 6, 1, 3, 5, 7
- Un arbre est dit H-équilibré ssi en tout nœud x ,
$$|\text{hauteur}(\text{droit}(x)) - \text{hauteur}(\text{gauche}(x))| \leq 1$$
- $\text{équilibre}(x) = \text{hauteur}(\text{droit}(x)) - \text{hauteur}(\text{gauche}(x))$

Arbres équilibrés : AVL

- Rotations simples
 - À droite
 - À gauche
- Rotations doubles
 - Gauche - droite
 - Droite – gauche
- Les opérations de rotations conservent la propriété d'ABR.

Arbres équilibrés : AVL

- Tout arbre H-équilibré ayant n nœuds a une hauteur h vérifiant :
 - $\log_2 (n+1) \leq h+1$
 - $h + 1 < 1,44 \log_2 (n + 2)$

Arbres équilibrés : AVL

- Insertion dans un AVL :
 - Réorganisation pour une insertion gauche – gauche dans le cas d'un arbre de racine P : **Rotation droite**
 - $P1 := \text{gauche}[P]$;
 - $\text{gauche}[P] := \text{droit}[P1]$;
 - $\text{droit}[P1] := P$;
 - $\text{équilibre}[P] := 0$;
 - $P := P1$;
 - $\text{équilibre}[P] := 0$;

Arbres équilibrés : AVL

- Insertion dans un AVL :
 - Réorganisation pour une insertion gauche – droite dans le cas d'un arbre de racine P :

Rotation gauche - droite

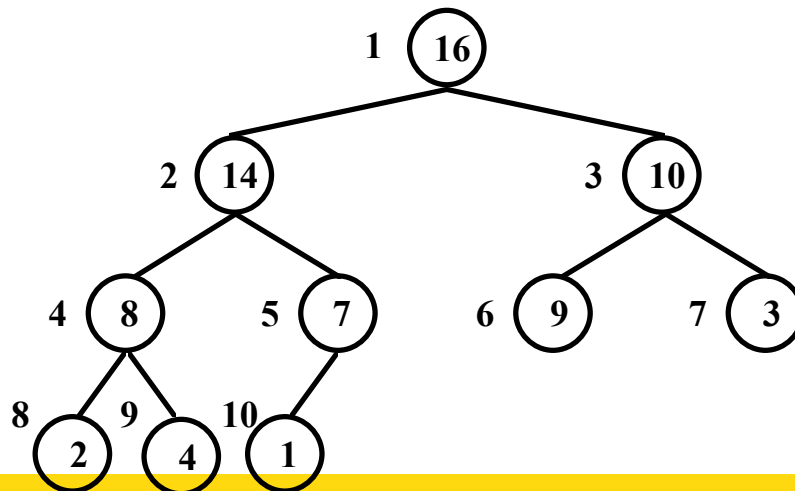
Algorithmique et structures de données

NF16

Structure de tas

Structure de tas

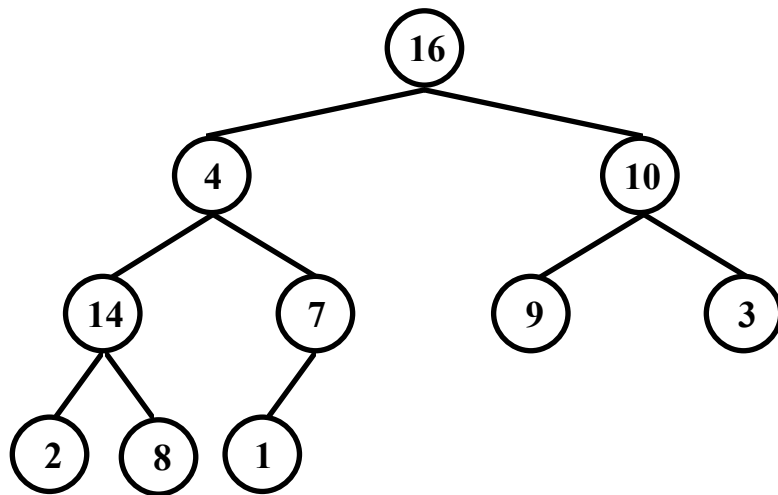
- Un tas est un objet qui peut être vu comme un arbre binaire stocké de manière compacte et trié verticalement.
- Un tableau A sert à stocker les données
- Une donnée possède une clé
- Le tableau A a pour taille maximale longueur[A]
- Le nombre de données dans le tas est de taille[A] (qui est \leq longueur[A] nombre d'éléments du tas)



Propriétés

- **Racine(A) \leftarrow Retourner(1)**
- **Pere(i) \leftarrow Retourner($\lfloor i/2 \rfloor$)**
- **Gauche(i) \leftarrow Retourner(2i)**
- **Droit(i) \leftarrow Retourner(2i + 1)**
- **$A[i] \leq A[\text{Pere}(i)]$ (dans ce cours)**
- **Les éléments entre les indices $\lfloor \text{taille}[A]/2 \rfloor + 1$ et $\text{taille}[A]$ sont des feuilles**

Entasser(A, i)



ex : Entasser(A, 2)

$l \leftarrow \text{Gauche}[i]$

$r \leftarrow \text{Droit}[i]$

Si $l \leq \text{taille}[A]$ et $A[l] > A[i]$

alors $\text{max} \leftarrow l$

sinon $\text{max} \leftarrow i$

Si $r \leq \text{taille}[A]$ et $A[r] > A[\text{max}]$

alors $\text{max} \leftarrow r$

Si $\text{max} \neq i$

alors $\text{echange}(A[i], A[\text{max}])$

Entasser(A, max)

Construire_tas(A)

taille[A] \leftarrow longueur[A]
Pour $i \leftarrow \lfloor \text{longueur[A]} / 2 \rfloor$ à 1
faire
Entasser(A, i)

Trier_tas (A)

Construire_tas(A)

Pour $i \leftarrow \text{longueur}[A]$ à 2

faire échange($A[1]$, $A[i]$)

taille[A] \leftarrow taille[A] - 1

Entasser(A, 1)

File de priorité

Extraire_Max_tas(A)

Si $\text{taille}[A] < 1$
 alors **Erreur**('débordement négatif')
max $\leftarrow A[1]$
A[1] $\leftarrow A[\text{taille}[A]]$
taille[A] $\leftarrow \text{taille}[A] - 1$
Entasser(A, 1)
Retourner(max)

File de priorité

Inserer_tas(A, cle)

taille[A] \leftarrow taille[A] + 1

i \leftarrow taille[A]

Tant que i > 1 et A[Pere(i)] < cle

faire A[i] \leftarrow A[Pere(i)]

i \leftarrow Pere(i)

A[i] \leftarrow cle

Algorithmique et structures de données

NF16

Méthodes de tri

Définitions

- Tri interne : s'effectue sur des données présentes en mémoire centrale. L'accès aux informations prenant très peu de temps on ne considère que le nombre de comparaisons et de permutations pour évaluer le temps de calcul.
- Tri externe : s'effectue sur des données résidant en mémoire secondaire. Dans ce cas le temps d'accès aux informations devient crucial, et l'on cherche donc à minimiser le nombre d'accès à la mémoire secondaire.

Définitions

- Un tri est stable s'il ne modifie pas l'ordre initial de deux éléments de clés égales. Un tri stable est intéressant dans le cas d'éléments multiclés. Il permet de conserver l'ordre établi pour une clé, en cas d'ex æquo dans un tri sur une autre clé.
- Complexité d'un tri comparatif :
 - Une borne inférieure pour le pire des cas : $\Omega(n \log(n))$

Méthodes de tri interne

- Tri par insertion.
 - L'insertion de l'élément frontière est effectuée par décalage ou par permutation.
 - Stable.
 - Complexité :
 - $O(n^2)$.
 - $O(n)$ si le tableau est déjà trié.
- Tri par bulles.
 - Dans cette méthode, on effectue un certain nombre de parcours du tableau à classer. Un parcours consiste à aller d'un bout à l'autre du tableau en effectuant la comparaison de deux éléments successifs et en les permutant s'ils ne sont pas classés.
 - Complexité : $O(n^2)$.

Tri rapide

- $\text{Tri_Rapide}(A, p, r)$
Si $p < r$ alors
 $q := \text{Partitionner}(A, p, r)$
 $\text{Tri_Rapide}(A, p, q)$
 $\text{Tri_Rapide}(A, q+1, r)$
- $\text{Partitionner}(A, p, r)$
 $x := A[p] ; i := p - 1 ; j := r + 1 ;$
 Tant que vrai faire
 répéter $j := j - 1$ jusqu'à $A[j] \leq x$
 répéter $i := i + 1$ jusqu'à $A[i] \geq x$
 si $i < j$ alors échanger $(A[i], A[j])$
 sinon retourner j

Tri rapide

- Dans Partitionner :
 - Les indices i et j ne font jamais référence à un élément de A hors de l'intervalle $[p..r]$.
 - L'indice j n'est pas égal à r quand Partitionner se termine : le découpage n'est jamais trivial.
 - Chaque élément de $A[p..j]$ est inférieur ou égal à chaque élément de $A[j + 1 .. r]$ quand Partitionner se termine.
- Complexité :
 - Pire des cas : $O(n^2)$.
 - Meilleur des cas : $O(n \log(n))$.
 - Tableau trié par ordre croissant ?
 - Tableau trié par ordre décroissant ?

Tri par dénombrement

- **Tri_Dénombrement(A, B, k)**
 Pour $i = 1$ à k faire $C[i] := 0$
 Pour $j = 1$ à $\text{longueur}[A]$ faire
 $C[A[j]] := C[A[j]] + 1$
 // $C[i]$ contient ici le nombre d'éléments de A qui sont égaux à i
 Pour $i = 2$ à k faire
 $C[i] := C[i] + C[i - 1]$
 // $C[i]$ contient ici le nombre d'éléments de A qui sont inférieurs ou égaux à i
 Pour $j = \text{longueur}[A]$ à 1 faire
 $B[C[A[j]]] := A[j]$
 $C[A[j]] := C[A[j]] - 1$

Algorithmique et structures de données

Graphes

Définitions

Graphe orienté :

- Un graphe orienté $G = (S, A)$ est un couple où S est un ensemble de sommets (nœuds) et A une partie de $S \times S$ dont les éléments sont appelés arcs.

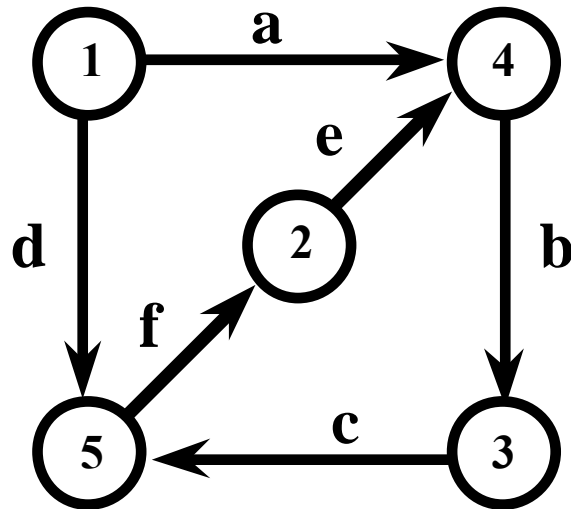
Graphe non orienté :

- Un graphe non orienté $G=(S, A)$ est un couple où S est un ensemble de sommets (nœuds) et A une famille de paires de S dont les éléments sont appelés arêtes.

Graphes

- Les graphes constituent un outil puissant de modélisation :
 - Réseau routier, téléphonique, ...
 - Plus court chemin, plus long chemin, ...
 - Gestion de projet, problèmes d'ordonnancement
 - Minimiser la durée du projet, réalisation à moindre coût, ...
 - etc.

Graphes



Orienté :

$$S = \{ 1, 2, 3, 4, 5 \}$$

$$A = \{ (1, 4), (1, 5), (2, 4), (3, 5), (4, 3), (5, 2) \}$$

Non Orienté :

$$S = \{ 1, 2, 3, 4, 5 \}$$

$$A = \{ [1, 4], [1, 5], [2, 4], [3, 5], [4, 3], [5, 2] \}$$

Dans l'exemple :

- les noeuds ont des labels numériques
- les arcs ont des labels alphanumériques

Définitions

- **Graphe valué** : un triplet $G = (S, A, v)$ où (S, A) est un graphe et v une application de A dans l'ensemble des réels.
- Soit un arc (i, j) . i est dit **extrémité initiale** de l'arc (i, j) et j **extrémité terminale**. On dit aussi que j est un **successeur** de i , et i un **prédécesseur** de j .
- Un **chemin** reliant deux sommets i et j est une suite d'arcs commençant en i et se terminant en j telle que les sommets intermédiaires parcourus soient tous distincts. Un **circuit** est un chemin se terminant sur le sommet de départ.
- La **longueur** d'un chemin est le nombre d'arcs qui le constituent.
- **Degré, Chaîne, cycle, connexité, composantes connexes.**

Graphes

matrice d'adjacence

		j				
		1	2	3	4	5
i	1	0	0	0	1	1
	2	0	0	0	1	0
	3	0	0	0	0	1
	4	0	0	1	0	0
	5	0	1	0	0	0

$M(G)$ est une matrice carrée
indicée par S

$$m_{ij} = \begin{cases} 1 & \text{si } [i, j] \in A \\ 0 & \text{sinon} \end{cases}$$

- successeur (i), prédécesseur(i)

Graphes

Matrice d'incidence

$\Delta(G)$ est une matrice définie par :

$$\delta_{x a} = \begin{cases} 1 & \text{si } x \text{ est origine de l'arc } a \\ -1 & \text{si } x \text{ est extrémité de l'arc } a \\ 0 & \text{sinon} \end{cases}$$

		arcs					
		a	b	c	d	e	f
noeud	1	1	0	0	1	0	0
	2	0	0	0	0	1	-1
	3	0	-1	1	0	0	0
	4	-1	1	0	0	-1	0
	5	0	0	-1	-1	0	1

- successeur (i) , prédécesseur(i)

Graphes

File (ou liste) de successeurs dit " $\alpha\beta$ "

Alpha

1	3	4	5	6	7
---	---	---	---	---	---

Beta

4	5	4	5	3	2
---	---	---	---	---	---

Successeurs du noeud 3 :

**de Alpha[3] à Alpha[4]-1
dans le tableau Beta donc
de Beta[4] à Beta[4]**

- **successeur (i) , prédécesseur(i)**

Parcours en largeur (G, s)

- **Pour tout sommet $u \neq s$ faire**
 - **Couleur[u] := Blanc; d[u] := infini; ArbrePere[u] := Nil;**
- **Couleur[s] := Gris; d[s] := 0; ArbrePere[s] := Nil;**
- **F := {s};**
- **Tant que F est non vide faire**
 - **u := Tête[F];**
 - **Pour chaque v dans Adj[u] faire**
 - **Si couleur[v] = Blanc alors**
 - **Couleur[v] := Gris; d[v] := d[u] + 1; ArbrePere[v] := u; Enfiler(F, v)**
 - **Defiler(F)**
 - **Couleur[u] := Noir**

Algorithmique et structures de données

NF16

Table de hachage

Tables de hachage

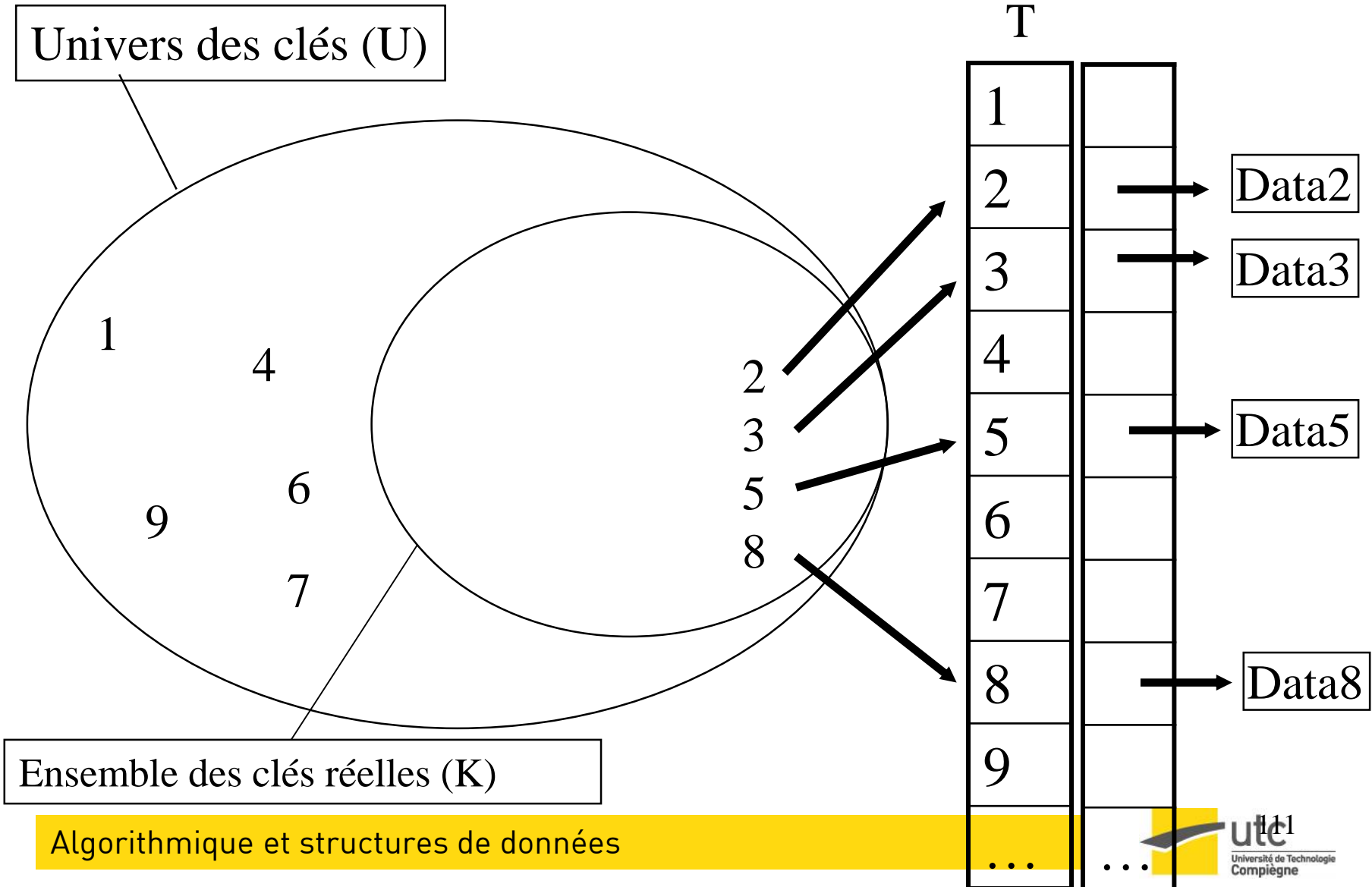
Objectifs

- **Gestion d'ensemble dynamique de données**
- **Calculer à partir de la clé k l'indice i de la case où se trouve potentiellement l'information à stocker, consulter ou détruire**
- **la correspondance entre l'ensemble des clés et l'ensemble des informations est univoque**

Tables à adressage direct

- Si l'univers U des clés n'est pas trop important, on utilise alors un tableau T de même cardinal
- T est appelé table à adressage direct
- à une alvéole (case) de T correspond une clé k de U
- Une alvéole pointe vers un élément de l'ensemble ayant pour clé k
- Opérations d'accès (recherche, insertion, suppression)

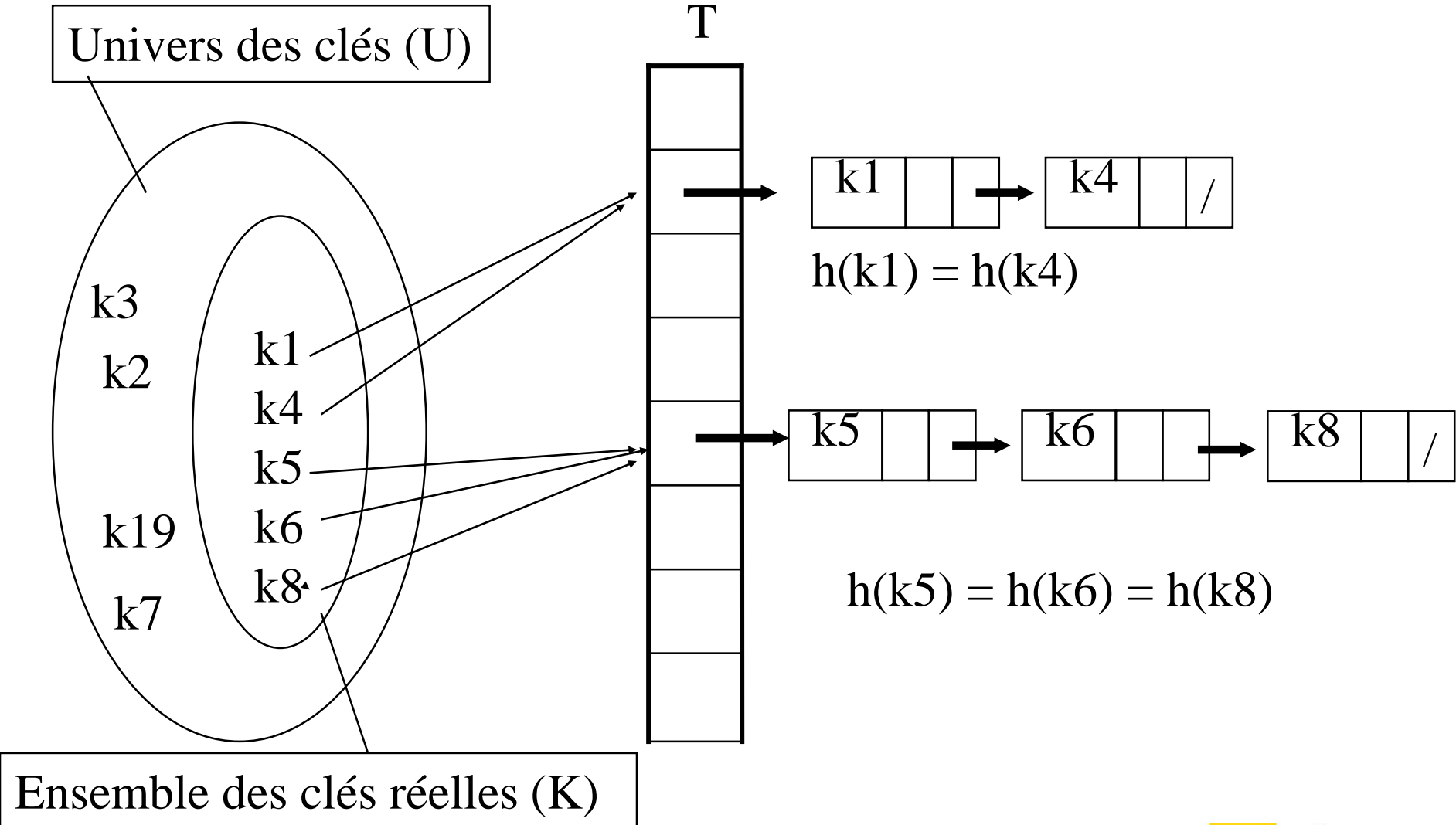
Tables à adressage direct ($|T| = |U|$)



Tables de hachage

- Soit U l'univers des clés et K l'ensemble des clés réellement utilisées et $|U| \gg |K|$
- Utiliser une fonction de hachage $h(k)$ qui retourne un numéro d'alvéole de la table de hachage T . On pose $|T| = m$ et n le nombre de clés.
- Une collision survient si deux clés différentes ont même valeur de hachage

Tables de hachage ($|T| \ll |U|$)



Gestion des collisions par chaînage

- **Résolution des collisions par chaînage**
- **Opérations de base**
- **Soit n le nombre de clés à gérer, on définit le facteur de remplissage par $\alpha = n/m$**
- **Gestion en zone de débordement et en interne**
- **La performance moyenne est en $\Theta(1+\alpha)$**

Opérations de base

- **Inserer_H_chaine(T, x)**
 - insérer en tête de la liste $T[h(cle[x])]$
- **Recherche_H_Chaine(T, k)**
 - rechercher un élément de clé k dans la liste $T[h(k)]$
- **Supprimer_H_chaine(T, x)**
 - Supprimer x de la liste $T[h(cle[x])]$