Structures de Données

2019-20

Tounwendyam Frédéric Ouédraogo
Université Norbert Zongo

Volume horaire

- CM = 24h
- TD /TP = 12h

Objectifs

Acquérir des bases de développement méthodique de logiciels en s'appuyant sur trois niveaux d'approche:

- **Niveau fonctionnel** exprimant de façon formelle le comportement sous forme de TAD (type abstrait de données).
- **Niveau logique** exprimant sous une forme algorithmique les principes de réalisation.
- **Niveau physique** exprimant dans un langage de programmation la solution retenue.

Plan du cours

A. Généralités et rappel
B. Structures linéaires
C. Structures arborescentes
D. Gestion des fichiers

A Généralités

A.1 Notion d'algorithme

→ Abu Ja Far Mohamed Ibn Al-rhowâ-rismî

- Mathématicien arable du 9ème siècle

→ Définitions

- Un algorithme est la composition d'un ensemble fini d'étapes (tâches) dont chacune est:
  - →Définie de façon rigoureuse et non ambiguë
  - →Effective (= pouvant être réalisée en un temps fini)

A Généralités

A.1 Notion d'algorithme

→ Définitions

- Un algorithme est une procédure de calcul bien définie, qui prend en **entrée** une valeur(ensemble de valeurs) et qui produit en **sortie**, une valeur (ensemble de valeurs)
- C'est une séquence d'étapes de calcul permettant de passer de la valeur d'entrée à la valeur de sortie

A.1 Notion d'algorithme

Exemple d'algorithme

→ Calcul du PGCD de 2 entiers n et m

```
fonction pgcd(m : entier, n : entier) : entier
var r : entier
début
  r reçoit n
  tant que r non nul faire
    soit r, le reste de la division de m par n
    si r est non nul alors
      m reçoit n
      n reçoit r
    fsi
  ftq
  retourne n
fin
```

→ Exemple avec 35 et 10

n	m	r
10	35	10
35	10	5
10	5	0

A.2 Structures de données

- Ordinateur = Système de Traitement de l'Information
- Unité de base = le bit (0/1)
- 1 octet = 8 bits
- Pb = l'information n'est pas manipulable au même niveau par l'homme et par la machine
  - Agrégation et structuration de l'information
  - Construction de structures plus abstraites

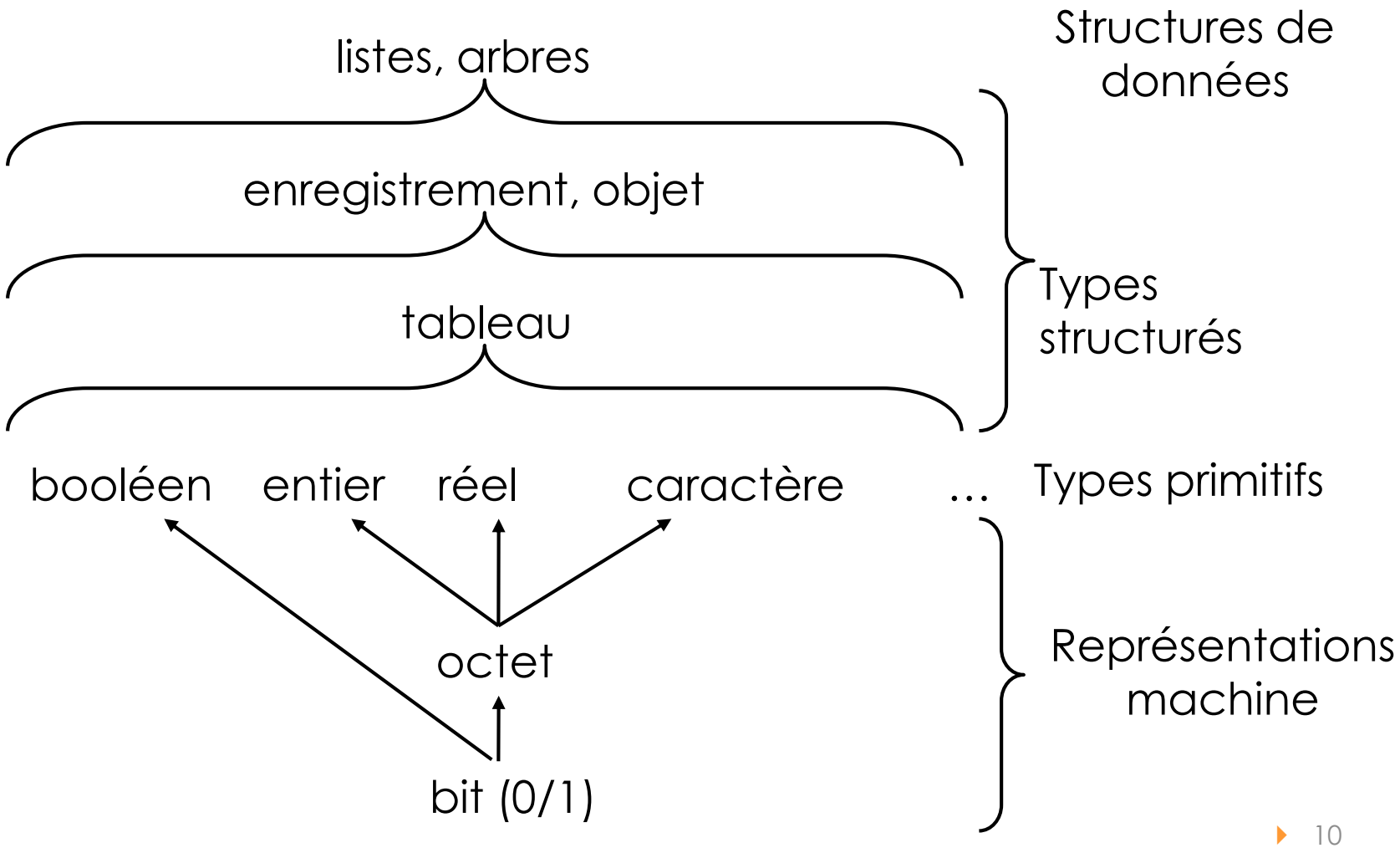
A.2 Structures de données

Structure de données =

- Un ensemble **structuré** de données construit par **agrégation** et **hiérarchisation** de données (ou structures de données) plus primitives c'est-à-dire moins abstrait.

A.2 Structures de données

Niveaux de représentation



A.2 Structures de données

Niveaux de représentation

Représentation des systèmes complexes :

- Système complexe briser en sous-système plus simple.
- Sous-systèmes peuvent représenter par des structures de données connues (piles, files, arbre,…)

Il est possible de résoudre un problème de deux façons, en utilisant des structures de données différentes.

Ceci implique des algorithmes différents qui sont liés aux SD → Etude de complexité

A Généralités

A.3 Programme

Programme = Structures de données + algorithmes

La partie algorithmique d'un programme décrit les **actions** que la machine doit effectuer alors que les données sont les **objets** sur lesquels portent les actions du programme.

Exemple en C:

Algorithme : boucles (while, for, do..while), conditions (if, switch), affectations (=), appels de fonctions, déclarations de fonctions.

Données : types prédéfinis (int, float, char...), tableaux, struct, arbres….

A Généralités

A.3 Programme

Programme = Structures de données + algorithmes

- Le pb = concevoir les structures de données de manière à ce que :
  - La mémoire utilisée soit **minimale**
  - les algorithmes soient les plus **simples** et les plus **efficaces** possible (2 besoins potentiellement contradictoires)

A Généralités

A.3 Programme

Programmation structurée

Une fois le modèle de données défini, le programme doit être écrit de manière structurée, c'est-à-dire être décomposé en petites entités ou sous-programmes (fonctions en C).

Chaque sous programme réalise une tâche précise, en ayant bien défini données nécessaires et celle retournées.

A.4 Récursivité

Introduction

Récursivité (informatique) <-> récurrence (maths)

Une démonstration par récurrence d'une propriété p(n) avec n un entier >0:

- Propriété vraie pour une valeur initiale n_0;
- Montrer que si la Propriété est vraie pour n alors vraie pour n+1

A.4 Récursivité

Introduction

Même principe pour le programme récursif mais en sens inverse.

Ainsi, pour réaliser une tâche sur des données de taille n, on va devoir d'abord la réaliser sur des données de taille n-1.

Appeler de façon récursive le même programme en diminuant la taille des arguments.

L'équivalent de l'initialisation d'une preuve par récurrence est une condition d'arrêt de l'appel récursif.

Lorsque la taille est suffisamment petite alors s'exécute la partie non récursive du programme.

A.4 Récursivité

Définition

Paradigme de calcul semblable au raisonnement par induction.

Une fonction ou un algorithme est dit récursif lorsqu'il est **auto-référent** : elle fait référence à elle-même (directement ou indirectement)dans sa définition.

Direct.

```
int fonct(…)   {
      x = fonct(…);
}
```

Indirect.

```
Int f1(…){
x=f2(…);
}
```

```
Int f2(…){
x=f1(…);
}
```

A.4 Récursivité

Motivation

Cela simplifie la résolution du problème.

Diviser pour régner : réappliquer un même traitement sur un échantillon de données d'une taille de plus en plus petite.

- La Récursivité sert aussi pour définir des structures de données : arbres, expressions arithmétiques, grammaires…
- La Récursivité est utilisable dans tous les langages de programmation modernes. En particulier, les langages **fonctionnels**.

A.4 Récursivité

Propriétés

Trois éléments à considérer lors de l'élaboration d'une procédure récursive :

Expression récursive du problème :

L' « équation » de la récursivité.

Condition d'arrêt :

Quand est-ce qu'on arrête les appels récursifs?

Convergence (vers la condition d'arrêt):

Une petite « preuve» et les conditions qui nous assure qu'un jour on va atteindre la condition d'arrêt.

A.4 Récursivité

Propriétés

Une fonction récursive doit posséder les deux propriétés suivantes:

- il doit exister certains critères, appelés critères d'arrêt ou conditions d'arrêt, pour lesquels la fonction ne s'appelle pas elle-même;
- chaque fois que la procédure s'appelle elle-même (directement ou indirectement), elle doit converger vers ses conditions d'arrêt.

Une fonction récursive possédant ces deux propriétés est dite bien définie.

A.4 Récursivité

Exemple La factorielle (n!)

Expression récursive du problème :

n!= n (n –1) … (2) (1) = n (n –1)!

Condition d'arrêt:

n = 1 ou n = 0

Convergence (vers la condition d'arrêt):

Si n = 1 ou n = 0, alors on a «convergé »!

Si n > 1, alors la soustraction à l'étape suivante nous approche de n = 1

si n est une entier non négatif ça converge!

A.4 Récursivité

Exemple La factorielle (n!)

```
long factorielle(int n)
{
        if (n < 0)/* hypothèse de convergence : n ≥ 0 */
                exit(1);

        if (n == 0 || n == 1)
                return  1; /* condition d'arrêt */

        else /* appel récursif */
                return  n * factorielle(n - 1);
}
```

Exercice d'application: proposer un algorithme récursif pour calculer le pgcd(m,n).

A.4 Récursivité

Itératif vs récursif

- Exemple : suite de *Fibonacci*
  - $U_0 = U_1 = 1$
  - $Un = U_{n-1} + U_{n-2}$
-

```
fonction fibo(n : entier) : entier
début
  si (i = 0) ou (i = 1)
  alors retourne 1
  sinon retourne fibo(n-1) + fibo(n-2)
fin
```

A.4 Récursivité

Itératif ou récursif

Solution itérative

```
fonction fibo(n : entier) : entier
var un, un_1, un_2, i : entier
début
  un_1 = un_2 = 1
  pour i = 2 jusqu'à i = n faire
    un = un_2 + un_1
    un_2 = un_1
    un_1 = un
  fait
  retourne un
fin
```

B- structures linéaires

Définition

On appelle **structure linéaire** une structure conçue de telle sorte que chaque fois qu'on passe du début à la fin de la structure, on parcourt tous ses éléments, toujours dans le même ordre.

B1- Les Listes

Définition

Une **liste** est ensemble ordonné d'éléments dans lequel on peut accéder à n'importe quel élément.

Une liste est une structure linéaire.

Exemples de liste dans la vite courante : liste de passage à un examen, file d'attente à un guichet.

B1- Les listes

Position des éléments

Exemple de liste d'entiers

- L = <8,1,5,4,6>
- $L_1 = 8$, $L_2 = 1$, $L_3 = 5$, $L_4 = 4$, $L_5 = 6$
- <8,1,5,4,6> ≠ <8,1,4,6,5>

Dans une liste on parle de **position**, donc le premier élément est en position 1. Il ne faut pas la confondre avec un tableau et commencer à l'indice 0.

Quelques opérations de base sur les listes.

- |L| Quel est le nombre d'éléments ?
- L = ∅ ? Est-ce que la liste est vide ?
- x ∈ L ? Est-ce que l'élément x appartient à L?
- Li Avoir accès à l'élément en $i^{ième}$ position
- x = Li ? Quelle est la position de l'élément x ?
- L +i x → L Ajouter l'élément x en $i^{ième}$ position
- L – Li → L Supprimer l'élément en $i^{ième}$ position
- L – x → L Supprimer la première occurrence de l'élément x
- L = L' ? Est-ce que les deux listes sont identiques ?
- L ⊆ L' ? Est-ce que la liste L est incluse dans L' ?
- → L Créer une liste vide

Réalisation ou implantation d’une liste par tableau

B1- Les listes

a)implantation par tableau

Ce premier modèle d'implantation retenu est le plus simple : l'utilisation d'un tableau de taille fixe.

Comme il s'agit d'un tableau statique, il faut décider quelle est sa taille.

Modèle d'implantation

```
DÉBUT
   TypeEl tab[TAILLE_MAX]
   int taille
FIN
```

a) Implantation par tableau

Insertion d'un élément

Faire un ajout se traduit par une insertion dans un tableau. Il faut donc, à partir de la fin jusqu'à la position à laquelle nous devons insérer, décaler tous les éléments d'une place vers la droite et ajouter l'élément à la position voulue.

Complexité linéaire

a) Implantation par tableau

Suppression d'un élément

Pour faire la suppression d'une position, il faut simplement décaler d'une place vers la gauche les éléments situés à droite de la position à supprimer.

Complexité linéaire

Recherche d'un élément

Pour faire une recherche, il suffit maintenant parcourir séquentiellement le tableau.

Complexité linéaire

a) Implantation par tableau

Les limites du modèle

- Lorsque la taille données dépasse celle du tableau.
- Quand il y a une grande fluctuations des données. L'occupation de la mémoire n'est pas optimisée
- l'opération de décalage des éléments vers la gauche ou vers la droite cause une lenteur dans l'exécution :

Pour contourner ce problème, il faut se tourner vers un modèle d'implantation qui a une grande importance dans les structures de données : le chaînage !

B1- Les listes

b) Implantation par chaînage

une liste chaînée est un ensemble fini d'élément de même type qui se suivent logiquement (mais non physiquement).

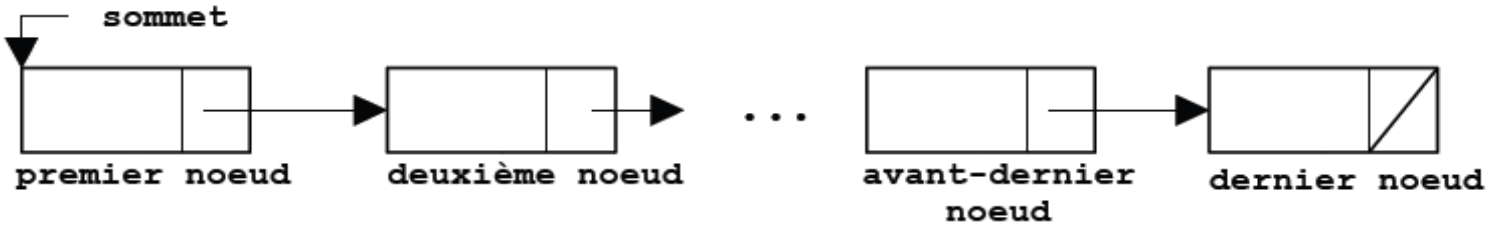
Dans une liste *simplement chaînée*, chaque élément comporte un et un seul pointeur vers un autre élément du même type.

C'est l'un des modèles d'implantation les plus importants en informatique.

b) Implantation par chaînage

Le chaînage simple

le chaînage simple consiste à inclure dans chaque élément de la liste un autre champ qui sert à pointer l'élément qui le suit logiquement. Les éléments d'une liste sont souvent appelés des nœuds.



b) Implantation par chaînage

Le chaînage simple

- Un élément comportera donc au minimum une partie « information » et une partie « pointeur sur le prochain élément ».
- Le dernier élément, n’a pas de successeur la référence est **null**
-

b) Implantation par chaînage

Le chaînage simple

Les nœuds d'une chaîne sont généralement modélisés par le type structure (enregistrement).

NOUVEAU TYPE Noeud

DEBUT

TypeEl info

Noeud *suivant

FIN

Info: contient l'information du nœud

b) Implantation par chaînage

Le chaînage simple

On a représenté un pointeur nommé **sommet**. Il est fourni au moment de la création du premier nœud de la liste.

NOUVEAU TYPE Liste

DEBUT

Noeud *sommet

FIN

b) Implantation par chaînage

Le chaînage simple

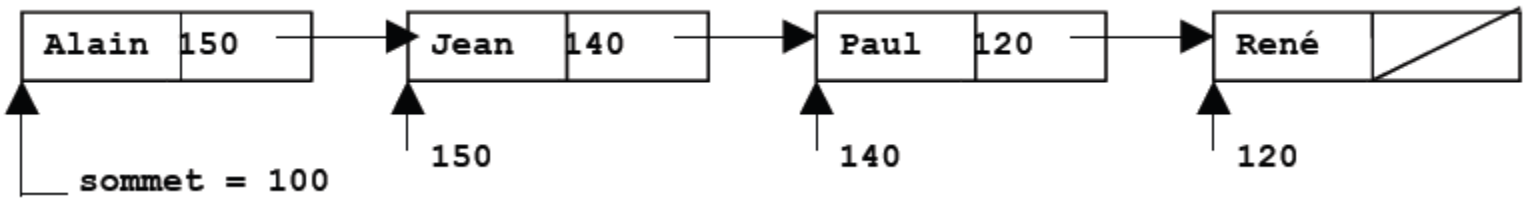
Le champ sommet est un pointeur sur la tête de la liste. Il est d'ailleurs possible (et même utile) d'ajouter un autre champ taille pour y garder la longueur de la liste et éviterait d'avoir à la calculer à chaque fois.

Perdre le pointeur sur une liste équivaut à perdre la liste et entraîne donc la création

b) Implantation par chaînage

Le chaînage simple

La figure suivante montre une liste de quatre articles rangés aux adresses physiques 100, 150, 140 et 120.



b) Implantation par chaînage

Initialisation

La fonction d'initialisation d'une telle **Liste** ne fera que mettre le pointeur sommet à **NULL**.

DEBUT

Liste l

l.sommet <- NULL {Initialiser le pointeur a NULL}

b) Implantation par chaînage

Nombre d'éléments

Cette opération va parcourir la liste chaînée en commençant par le sommet et compter le nombre de nœuds.

nbElementsListe

DEBUT

Courant ← l.sommet *{le pointeur courant pour se déplacer dans la liste }*

Cpt ← 0 *{Pour compter le nombre d'elements}*

TANT QUE courant != NULL

DEBUT

cpt ← cpt + 1

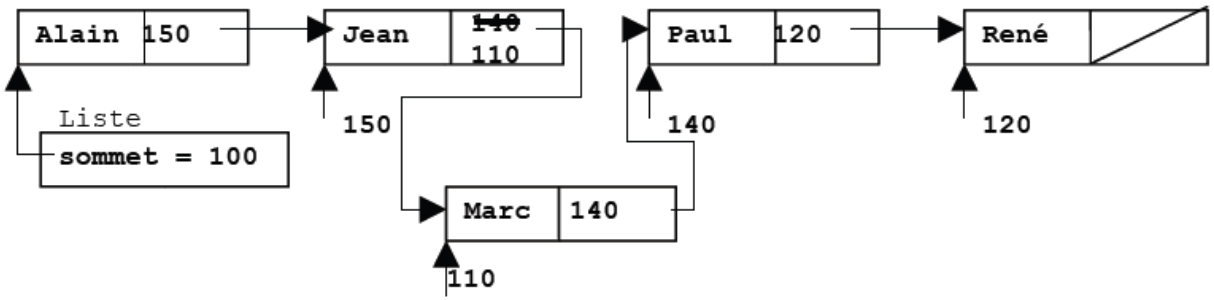
courant ← courant->suivant

FIN *{A: cpt contient le nombre d'elements de la Liste l}*

b) Implantation par chaînage

Insertion d'un élément.

Il suffit de changer les valeurs de quelques pointeurs.
Considérez cet exemple :



Inséré « Marc » entre le deuxième et le troisième nœuds.

On suppose que le nouveau nœud occupe déjà la case mémoire d'adresse 110. Le lien du second

article est changé de 140 à 110. Le lien du nouvel article reçoit la valeur 140.

Comment Ajouté marc au début de la liste?

b) Implantation par chaînage

Insertion d'un élément

ajoutListe(l,x,pos)

DEBUT

Nouveau ← nouveauNoeud(x) *{nouveau est différent de NULL, sinon PAM}*

SI pos = 1 ALORS FAIRE

DEBUT

nouveau-> suivant ← l.sommet

l.sommet ← nouveau

FIN

SINON

DEBUT

precedent ← l.sommet

REPETER pour **i □ [2, pos[**

DEBUT

prededent ← prededent->suivant

FIN *{precedent est le noeud en pos-1}*

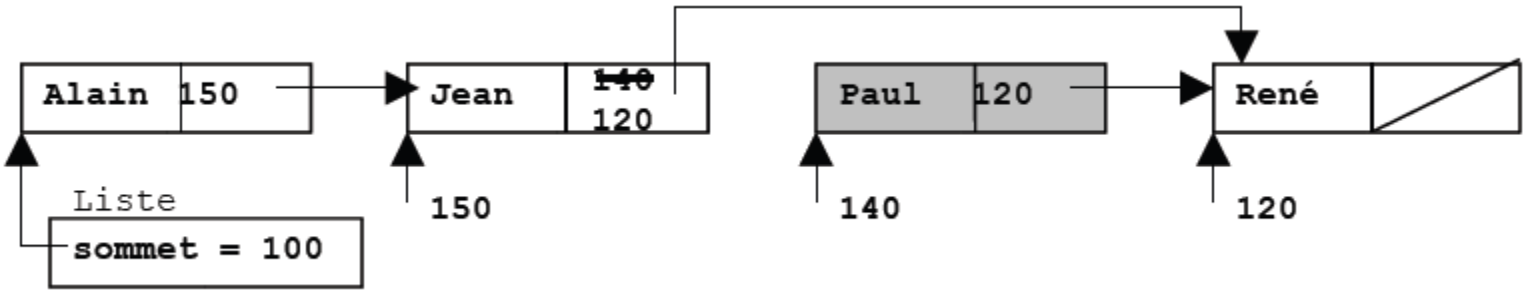
nouveau-> suivant ← precedent->suivant

precedent -> suivant ← nouveau

b) Implantation par chaînage

Suppression d'un élément

La suppression d'un article se fait de façon similaire.



S

donner la valeur NULL au pointeur de Paul. Si l'on avait éliminé le premier article (Alain), il aurait fallu donner à sommet la valeur 150.

b) Implantation par chaînage

Suppression d'un élément

enleverPosListe

DEBUT

SI pos = 1 ALORS FAIRE

aSupprimer → l.sommet

l.sommet → aSupprimer->suivant

SINON

precedent → l.sommet *{precedent est le noeud en pos-1}*

REPETER pour **i** ∈ **[2, pos[**

DEBUT

prededent → prededent->suivant

FIN

aSupprimer → precedent->suivant

precedent->suivant → aSupprimer->suivant

LIBERER aSupprimer *{La fonction FREE en C}*

B2- Les piles

Définition

Une pile est une structure linéaire qui se caractérisent par la manière d'insertion et d'extraction d'un élément qui se font toujours à la **même extrémité de la pile.**

Les éléments ne peuvent pas être retirés dans un ordre différent de celui de leur insertion dans la pile.

Ce principe est appelé en anglais LIFO (last in, first out)

B2- Les piles

Exemple de traitement utilisant les piles.

Traitement des structures emboîtées. par exemple, analyse d'un programme :

- boucles imbriquées,
- procédures locales,
- parcours d'arbres,
- récursivité,
- etc.)

pour lesquelles il faut que les sous-structures soient traitées avant la structure qui les contient.

B2- Les piles

a) Opérations sur les piles

- Ajouter un nouvel élément (***empiler***)
- Enlever un élément (**dépiler**)
- Regarder le sommet de la pile (**top**)
- Est-ce que la pile est vide ?
- Est-ce que l'élément x appartient à la pile ?

Il existe d'autres opérateurs

b) Implantation d'une pile

Implantation par tableau

Le modèle d'implantation par tableau est le plus simple.

Le meilleure choix de gestion du flux est d'ajouter toujours à la fin du tableau, autrement dit d'empiler et de dépiler toujours à la fin du tableau.

NOUVEAU TYPE Pile

DEBUT

TypeEl tab[TAILLE_MAX]

int nbElements

FIN

b) Implantation d'une pile

Implantation par tableau

empiler

DEBUT

{A : p.nbElements < TAILLE_MAX, sinon Pile Pleine}

p.tab[nbElements] <- x

p.nbElements <- p.nbElements + 1

FIN

Encore une fois, le problème avec ce modèle d'implantation, c'est la taille fixe du tableau

Une autre façon de faire l'implantation de la pile est d'utiliser le chaînage de structures.

b) Implantation d'une pile

Implantation par chaînage

NOUVEAU TYPE Noeud
DEBUT
TypeEl info
Noeud *suivant
FIN

NOUVEAU TYPE Pile
DEBUT
Noeud *sommet
FIN

```
Empiler(p,x)
DEBUT
 nouveau ← nouveauNoeud(x)
 nouveau->suivant ← p.sommet
  p.sommet ← nouveau
FIN
```

b) Implantation d’une pile

Exercice d’application

Des parenthèses de différentes formes peuvent être présentes dans une expression mathématique : **' ( ) ' , '{ }' et ' [ ]‘**

Pour que l'expression soit correcte, il faut que les parenthèses soient bien équilibrées, c’est-à-dire qu'une parenthèse ouverte par '[' doit être fermée par ']' et pas par ')' ou '}'.

L’expression 3*(2-[4 / (2 +x)] * y) est bien équilibrée au niveau des parenthèses.

A l’aide d’une pile proposer un algorithme qui vérifie l’équilibre des parenthèses d’une expression mathématique.

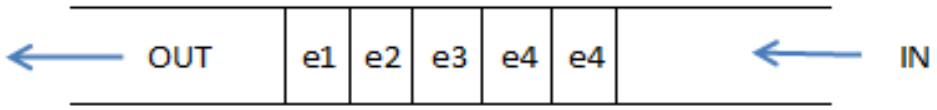
B3- Les Files

Définition

Cette structure linéaire se caractérise par un ce comportement particulier : **l'insertion se fait à une extrémité et l'extraction à l'autre extrémité.**

En anglais, on parle de **FIFO (*First In First Out*)**

Dans une file d'attente l'ordre d'arrivée des éléments est préservé.



File

B3- Les Files

Exemples d'application

- L'exécution des programmes dans un système multi-usagers. Puisque l'ordinateur n'a qu'une unité centrale de traitement (*CPU*), qu'une mémoire et qu'il ne peut traiter qu'un programme à la fois, il a fallu créer une file d'attente.
- processus d'impression pour une imprimante partagée dans un laboratoire. Un document envoyé pour impression est ajouté à une file d'attente.
- Etc

B3- Les Files

a) opérations sur les *files*

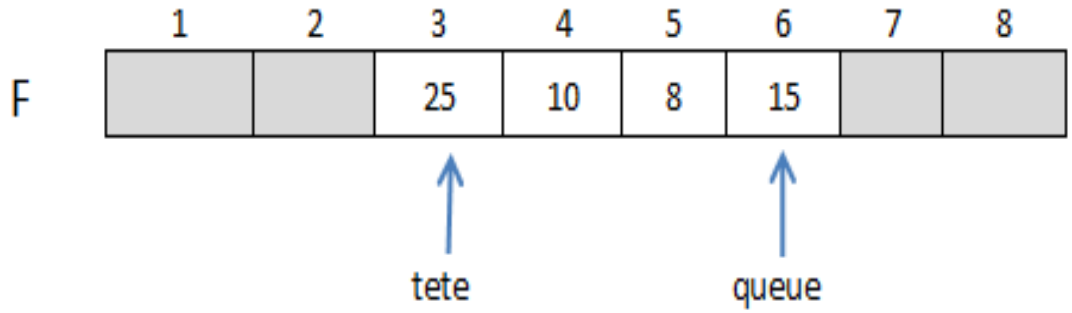
- Ajouter un nouvel élément (**enfiler**)
- Enlever un élément (**défiler**)
- Regarder le premier élément de la file (sans le défiler)
- Regarder le dernier élément de la file
- Est-ce que la file est vide ?
- Est-ce que l'élément x appartient à la file ?

b) Implantation d'une file

Implantation par tableau

On peut réaliser une file par l'entremise d'un tableau à une dimension et deux pointeurs (dans ce cas les indices du tableau), **tete** qui vise la tête de la file et **queue** qui vise la queue, le dernier élément de la file.

Exemple: une file F implémentée à l'aide d'un tableau de T de taille 8.

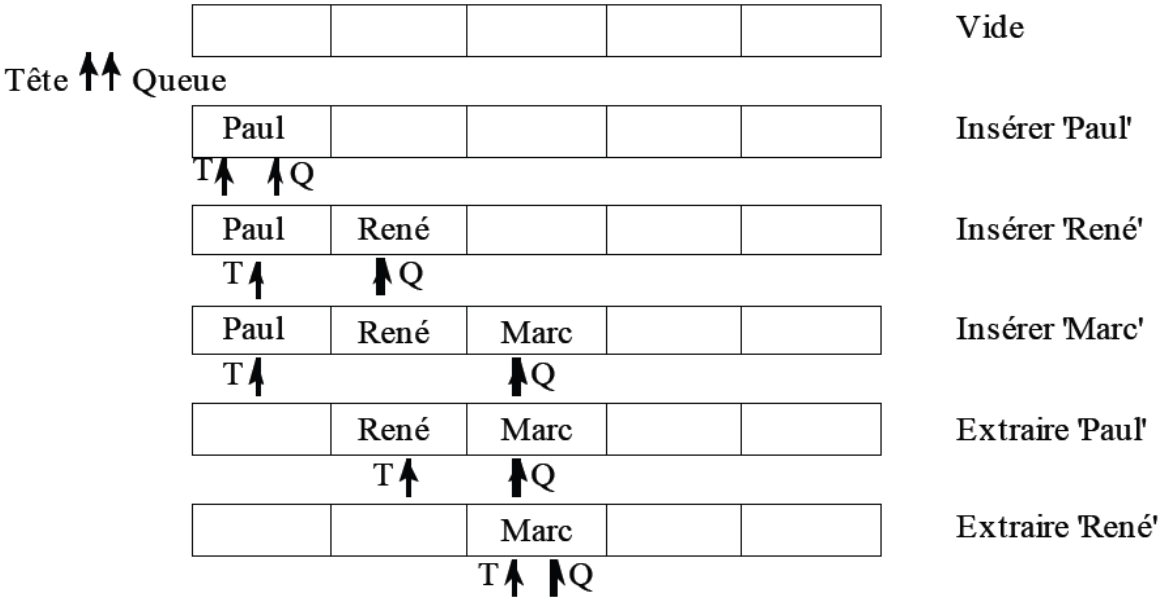


b) Implantation d'une file

Implantation par tableau

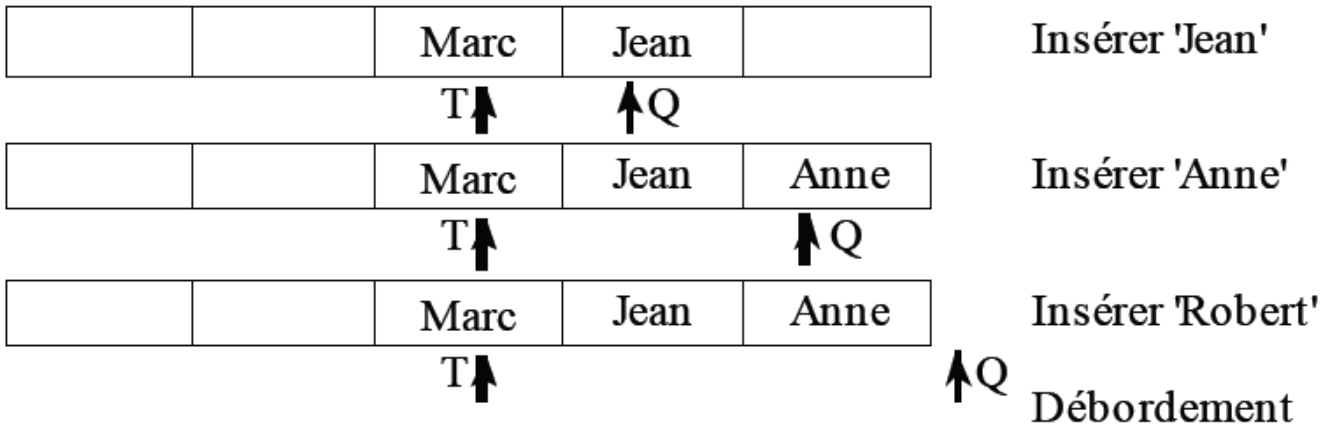
On peut épuiser la mémoire et ce, même si la file n'est jamais très chargée. Cela s'est produit parce que le déplacement des pointeurs se fait toujours dans le même sens et, entraînant plus tard un débordement du tableau.

Exemple



b) Implantation d'une file

Implantation par tableau



Pour éviter ce problème, on rend le tableau circulaire.

on fait un calcul modulo **n** (**n** est la longueur de la file).

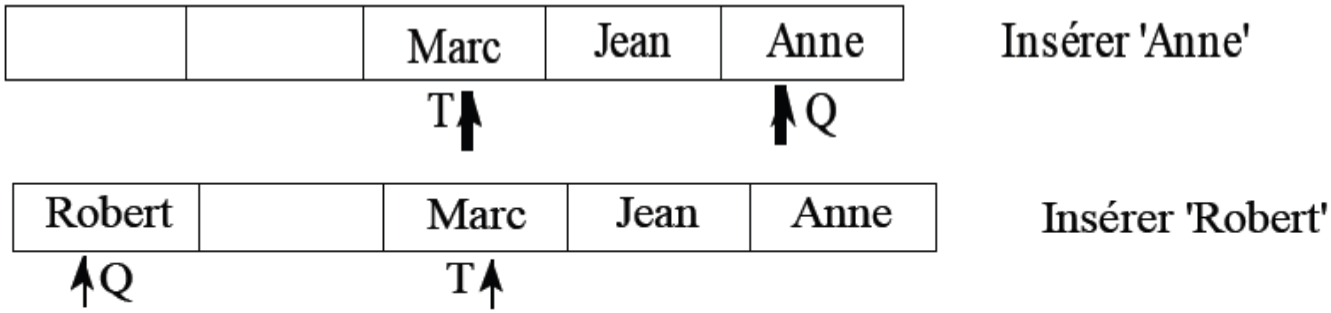
Lorsque nous voulons avancer un pointeur

d'une position, on fait : **p = p mod n +1**

b) Implantation d'une file

Implantation par tableau

Dans le précédent exemple, on aurait eu:



Attention à ce que les pointeurs de tête et de queue ne se dépassent pas .

Comment vérifier que la file est pleine?

C- Structures arborescentes ou arbres

Introduction

La structure d'arbre est une structure de données primordiale en informatique.
Diverses variantes sont utilisées : arbres binaires, B-arbres, arbres AVL, forêts d'arbres ...
Un arbre est une structure homogène dont chaque élément, appelé nœud, contient de l'information et plusieurs liens (pointeurs) vers des élément s du même type appelés nœuds fils ou successeurs.

C- Structures arborescentes ou arbres

Introduction

Exemples:

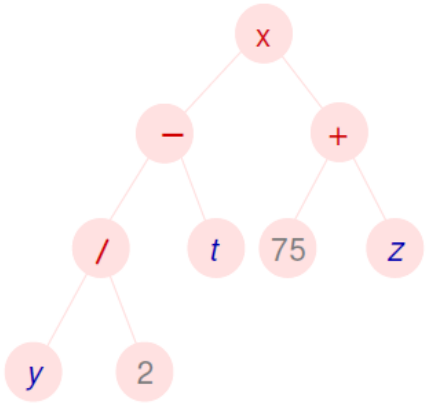
- Représenter les distances de Ouaga à toutes les autres villes du Burkina.
- Représenter votre arbre généalogique à partir de votre grand-père.

C- Structures arborescentes ou arbres

Introduction

Expressions arithmétiques

Un arbre arithmétique permet de représenter des expressions arithmétiques. Il est constitué soit de feuilles qui sont des entiers, des nœuds qui contiennent un opérateur arithmétique (+, −, ∗ ou /), un sous arbre gauche et un sous arbre droite.



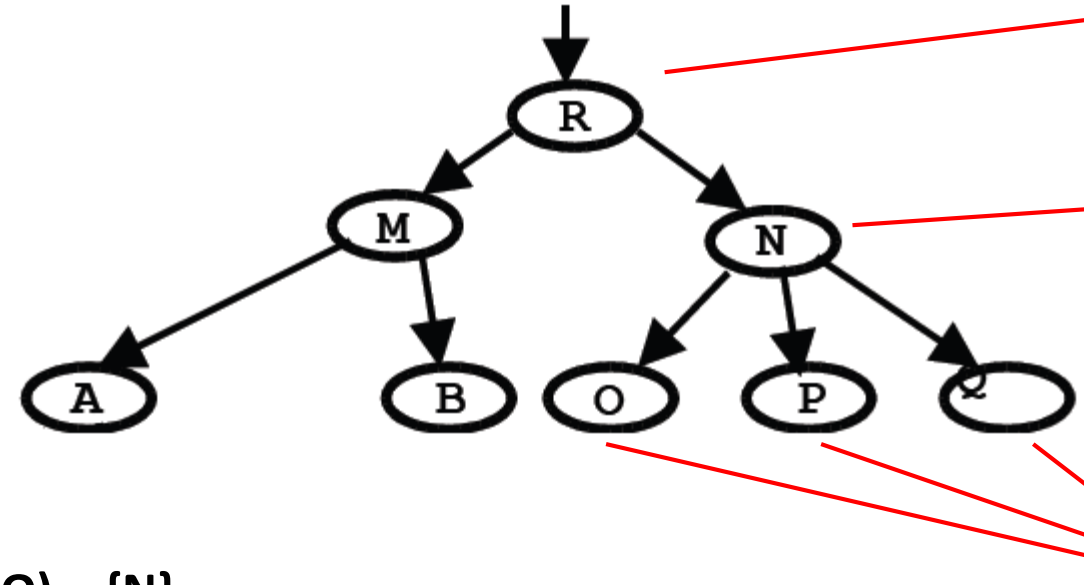
C- Structures arborescentes ou arbres

1) Terminologie

Un arbre (orienté ou enraciné) est un ensemble de nœuds muni d'une relation de « parenté » qui vérifie les conditions suivantes :

1. Il y a un seul nœud qui n'a pas de prédécesseur (père). Ce nœud s'appelle ***racine***.
2. Tous les nœuds, sauf la racine, n'ont qu'un seul père (prédécesseur).
3. Il existe **un chemin unique** de la racine à chaque nœud.

1) Terminologie



Racine

Nœuds internes

Feuilles ou nœuds externes

père(O) = {N}

père(R) = {}

fils(M) = {A,B}

fils(B) = {}

1) Terminologie

Les nœuds terminaux, qui n'ont pas de fils, sont appelés *feuilles ou nœuds externes*.

Les nœuds qui ne sont pas des feuilles ou racine sont appelés *nœuds internes*.

Le prédécesseur d'un nœud s'appelle le ***père*** ou l'*ancêtre* direct du nœud.

Le successeur est appelé le ***fils*** ou le *descendant* direct du nœud.

Un nœud peut avoir plusieurs fils.

1)Terminologie

Degré de nœud

Le nombre de fils d'un nœud n dans un arbre est appelé le **degré** de n.

Un arbre **ordonné** est un arbre dont dans lequel les fils de chaque nœuds sont ordonnées. En d'autres termes: si nœuds à k fils, il y a un 1^{er} fils, 2^{e} fils … $K^{ième}$ fils.

1) Terminologie

Chemin et branche

Un **chemin** de longueur k entre nœuds **u** et **v** d'un **arbre** est la suite de nœuds $\{n_0, n_1, ..., n_k\}$ telle que n_0=u, n_k=v et pour tout i appartenant à l'intervalle [1..p], n_{i-1} = père(n_i)

Une **branche** de l'arbre est un chemin de la racine à l'une de ses feuilles.

C- Structures arborescentes ou arbres

1)Terminologie

Bords

*Le **bord gauche*** de l'arbre est le plus long chemin depuis la racine en ne suivant que les fils gauches

*Le **bord droit*** de l'arbre est le plus long chemin depuis la racine en ne suivant que les fils droits

D2 Structures arborescentes (Arbre)

1) Terminologie

Sous arbre

Soit *A* arbre et ***n*** nœud de *A*.

- Un **sous arbre** d'arbre ayant pour racine **n** est la partie de l'arbre constitué des descendant de n.
- Le(s) **sous arbres d'un nœud x** sont le(s) sous-arbes dont les racines sont des fils de x.

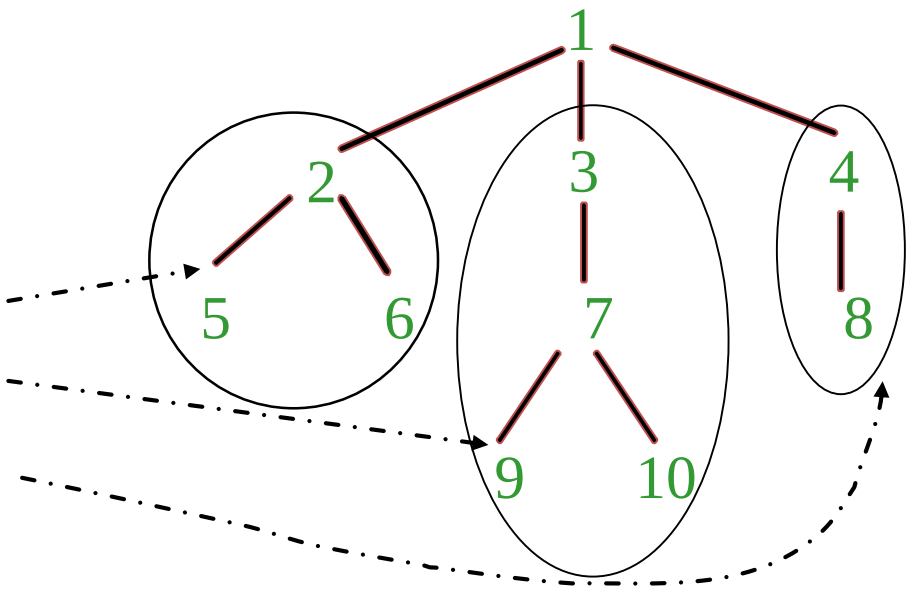
Sous-arbre

A arbre, x nœud de *A*

SA(x) = sous-arbre de *A* qui a racine x

A = SA(racine(*A*))

SA (2)
SA (3)
SA (4)



SA(2), SA(3) et SA(4) sont des sous-arbres de 1

1) Terminologie

Hauteur

La *hauteur* d'un nœud est le nombre ***maximum*** **de liens** qu'il faut parcourir pour aller à une feuille.

La hauteur d'un arbre est la hauteur de sa racine.

Définition récursive

Soit A arbre et *n* nœud de *A*

$$h_A(n) = \begin{cases} 0 \quad \text{si } n = \textit{feuille} \\ 1+\max\{\, h_A(e), \text{ où } e \text{ fils de } n \,\} \end{cases}$$

Hauteur

Soit A un arbre de 10 nœuds

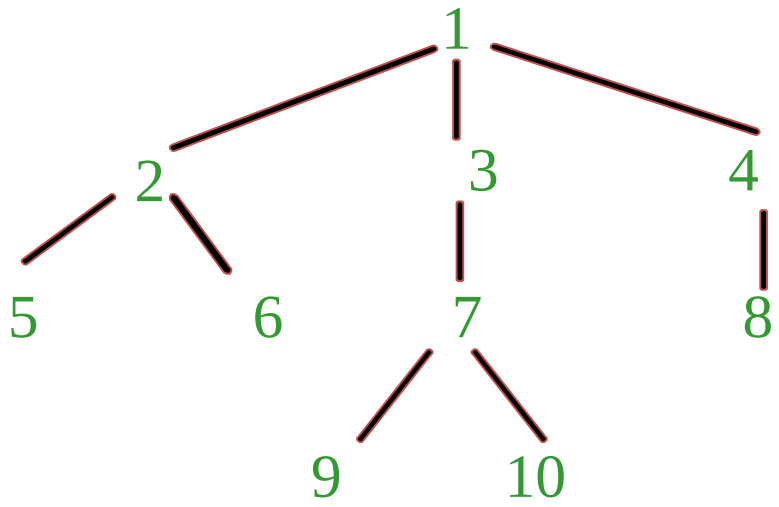
Hauteur 3

$h_A(8) = 0$

$h_A(7) = 1$

$h_A(3) = 2$

$h(A) = h_A(1) = 3$



C- Structures arborescentes ou arbres

2) Parcours

Il y a deux catégorie de parcours dans un arbre :

parcours en profondeur (branche après branche)

- préfixe,
- suffixe,
- symétrique

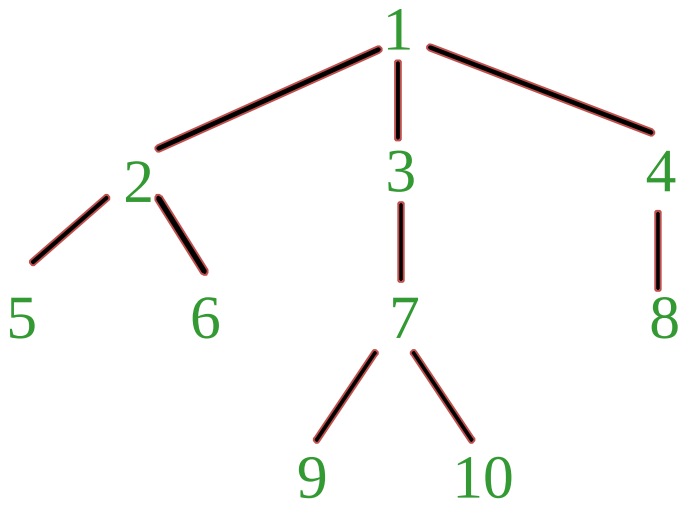
parcours en largeur (niveau après niveau)

2) Parcours

Considérons la définition suivante d'arbre A non vide et dont SAi sous les sous arbre de la racine r

$A = (r, SA_1, SA_2, \ldots, SA_k)$

SA_i = sous-arbre de r

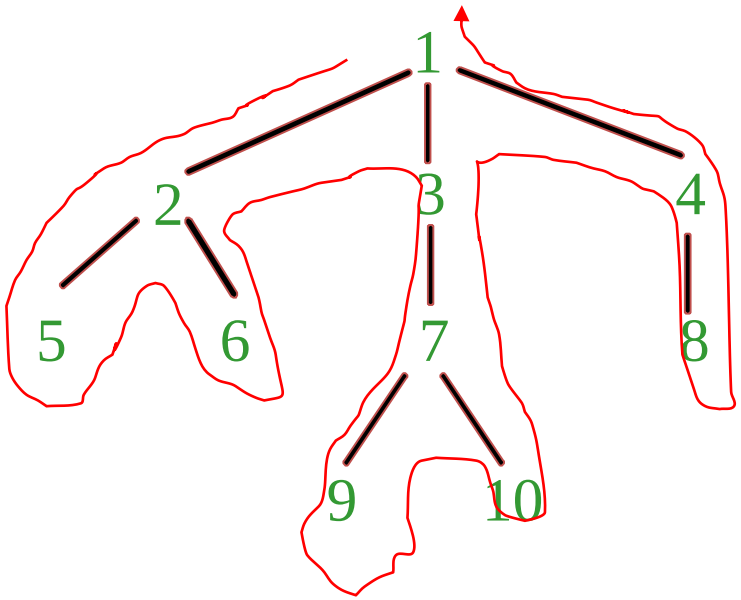


Parcours préfixe

Cette visite donne la priorité au père.

- Visiter la racine **r**;
- Visite les sous arbres SA_1, SA_2, …, SA_k

P(A) = (r).P(SA_1). ... .P(SA_k) : (1, 2, 5, 6, 3, 7, 9, 10, 4, 8)



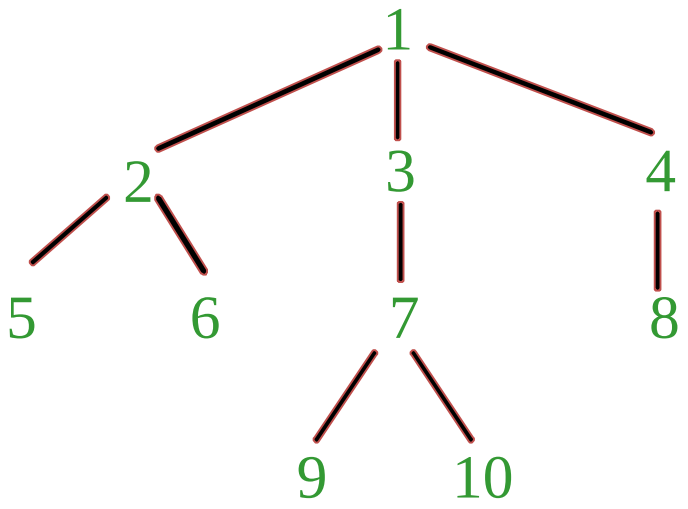
Parcours suffixe

Le parcours suffixe donne la priorité aux fils

- Visite les sous arbres SA_1, SA_2, …, SA_k
- Visiter la racine **r**;

$S(A) = S(SA_1). \ldots .S(SA_k).(r)$

(5, 6, 2, 9, 10, 7, 3, 8, 4, 1)

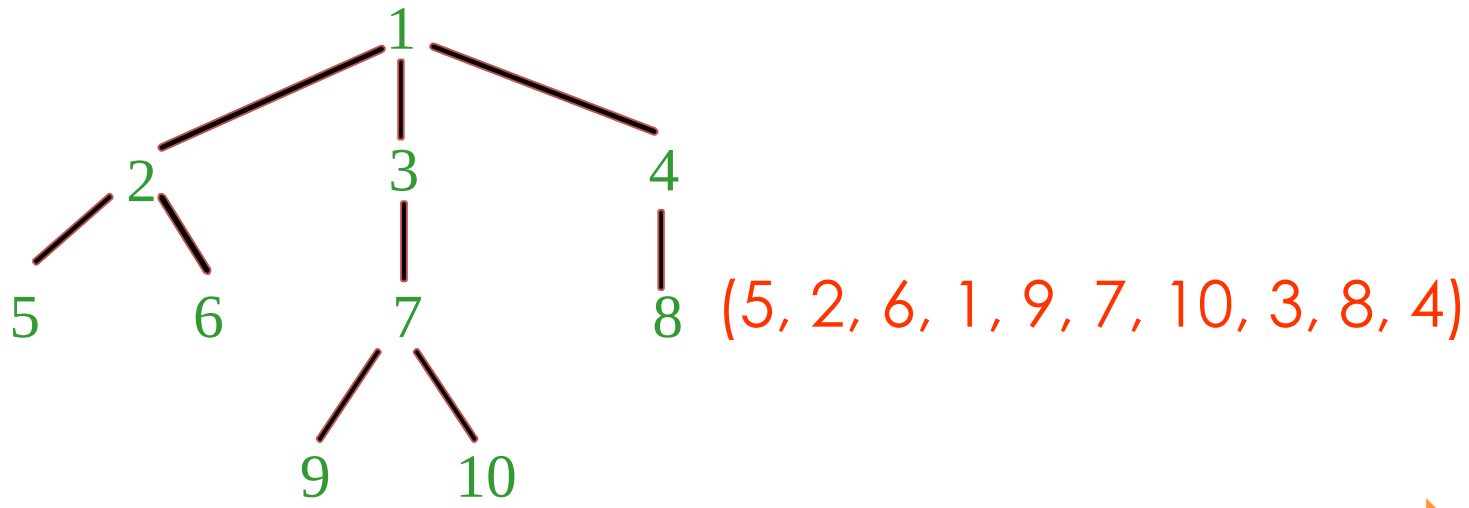


Parcours symétrique

Ce parcours se fait selon l'ordre suivant

- Visite le sous arbres SA_1
- Visiter la racine **r**;
- , SA_2, …, SA_k

$I(A) = I(A_1).(r).I(A_2). \ldots .I(A_k)$



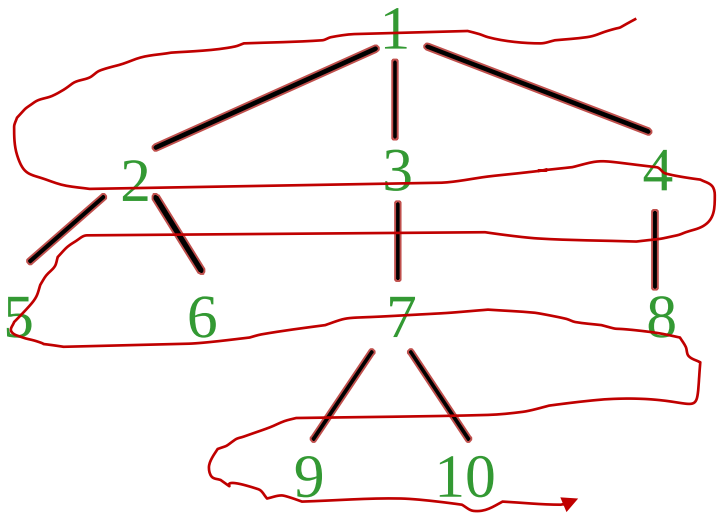
Parcours en largeur

- **Parcours en largeur ou hiérarchique**

$$H(A) = (r, x_1, \ldots, x_i, x_{i+1}, \ldots x_j, x_{j+1}, \ldots, x_n)$$

- nœuds de niveau 0, 1, 2, ...

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)



3) Arbres binaires

Un arbre de degré 2 est appelé **arbre binaire**.

Un *arbre binaire* est un arbre orienté qui vérifie les conditions suivantes :

- Chaque fils d'un nœud est soit fils gauche, soit fils droit.
- Aucun nœud n'a plus d'un fils gauche ni plus d'un fils droit.

Un arbre de degré ***n*** supérieur à 2 est appelé arbre ***n***-aire.

3) Arbres binaires

Définition récursive

Un arbre binaire non vide $A = <r, A_1, A_2>$

- r = nœud racine,
- A_1 = *arbre binaire, sous-arbre gauche* de r,
- A_2 = *arbre binaire, sous-arbre droit* de r,

- *Un arbre binaire qui ne contient aucun nœud est appelé* ***arbre vide***
- *Si A1 n'est pas vide sa racine est fils gauche de r, de même si A2 n'est pas vide sa racine est fils droit de r.*
- *Si un sous-arbre est vide on dit que le fils est* ***absent*** *ou* ***manquant***

3) Arbres binaires

•• Un arbre binaire **complet** est un arbre binaire dont chaque nœud est soit une feuille soit de d'un degré égal à 2. Autrement dit tous les nœuds n'ont pas de fils manquants sauf la feuille.

• Le nombre maximal de nœuds dans un arbre binaire est:

$$\sum_{i=0}^{p-1} 2^i = 2^p\text{-}1$$

3) Arbres binaires

Exercice d'application:

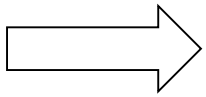
Dessiner tous les arbres binaires ayant les nœuds a, b, et c avec a pour racine.

3) Arbres binaires

Parcours ou visites

Accéder à l'information contenue dans les nœuds d'un arbre pour

- faire une recherche,
- compter des éléments,
- faire une insertion,
- faire une suppression,
- …



Parcourir ou visiter les nœuds

Parcours ou visites

Dans le cas d'une structure linéaire, le choix est facile : on y passe en ligne droite, du début à la fin.

Cependant, dans les arbres comme dans les graphes, la ligne droite n'existe pas. Alors, comment peut-on passer par tous les nœuds ?

Parcours ou visites

Deux types de parcours

- En largeur d'abord
- En profondeur d'abord
  - Préfixé (ordre de priorité au père)
  - Postfixé (ordre de priorité aux fils)
  - Infixé (l'ordre de priorité symétrique)

3- Arbres binaires

Parcours en profondeur

- Ces trois ordres viennent de la définition récursive d'un arbre : un arbre est vide ou est une racine avec un arbre à gauche et un arbre à droite.
- Il y a donc trois arbres et c'est l'ordre de visite de ces trois arbres qui change.

3- Arbres binaires

Parcours Préfixé (ordre de priorité au père)

Cet parcours se fait dans ce sens :

- Visiter la racine **r**;
- Visiter selon la priorité au père les sous arbres SAG, SAD de r;

```
Parcprefixe(Racine)
Début
   SI Racine ≠ Nil ALORS FAIRE
   debut
         Traiter Racine
         Parcprefixe (filsGauche)
         Parcprefixe(filsDroit)
   fin
Fin
```

3- Arbres binaires

Parcours Postfixé (ordre de priorité au fils)

Cet parcours se fait dans ce sens :

- Visiter les sous arbres SAG, SAD de **r** selon la priorité au fils ;
- Visiter la racine **r**;

```
Parcpostfixe(Racine)
Début
   SI Racine ≠ Nil ALORS FAIRE
   debut
        Parcpostfixe (filsGauche)
        Parcpostfixe(filsDroit)
        Traiter Racine
   fin
Fin
```

Parcours infixé (ordre de priorité symétrique)

Cet parcours se fait dans ce sens :

- Visiter selon la priorité symétrique le fils gauche de r;
- Visiter la racine **r**;
- Visiter selon la priorité symétrique le fils droit de r.

```
Parcinfixe(Racine)
Début
    Sl Racine ≠ Nil ALORS FAIRE
    debut
            Parcinfixe (filsGauche)
            Traiter Racine
            Parcinfixe(filsDroit)
    fin
Fin
```

3- Arbres binaires

Implantation Par tableau

On crée une tableau de **N** éléments, un élément par nœud, et ensuite on fait une correspondance des nœuds de l'arbre sur les éléments du tableau.

Pour ce faire, on identifie chaque nœud de l'arbre :

- La racine a comme identificateur **1;** **0**
- Un fils gauche a comme identificateur **2D;** **2D+1**
- Un fils droit a comme identificateur **2D + 1;** **2D+2** où **D** est l'identificateur du père.

3- Arbres binaires

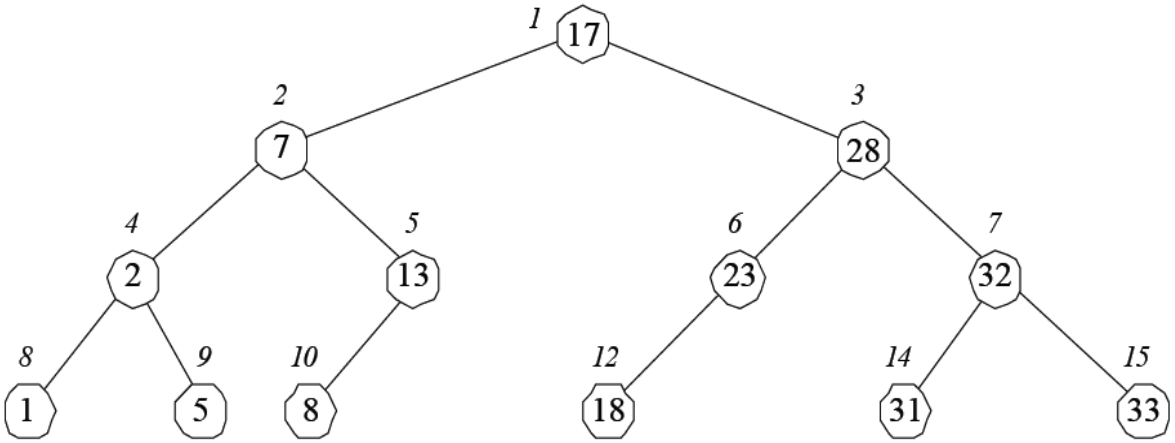
Implantation Par tableau

Si on ne connaît pas exactement la taille de l'arbre, on ajouter une marge de sécurité dans l'allocation de la mémoire.

Exemple d'un arbre de 15 nœuds :

3- Arbres binaires

Implantation Par tableau



17	7	28	2	13	23	32	1	5	8		18		31	33
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

L'arbre par excellence à implanter dans un tableau est l'**arbre binaire *complet***. Un arbre binaire complet remplit tout le tableau.

3- Arbres binaires

Implantation Par tableau

Les avantages de ce modèle d'implantation sont surtout au niveau de la manipulation de

l'arbre : il s'agit de la manipulation des indices.

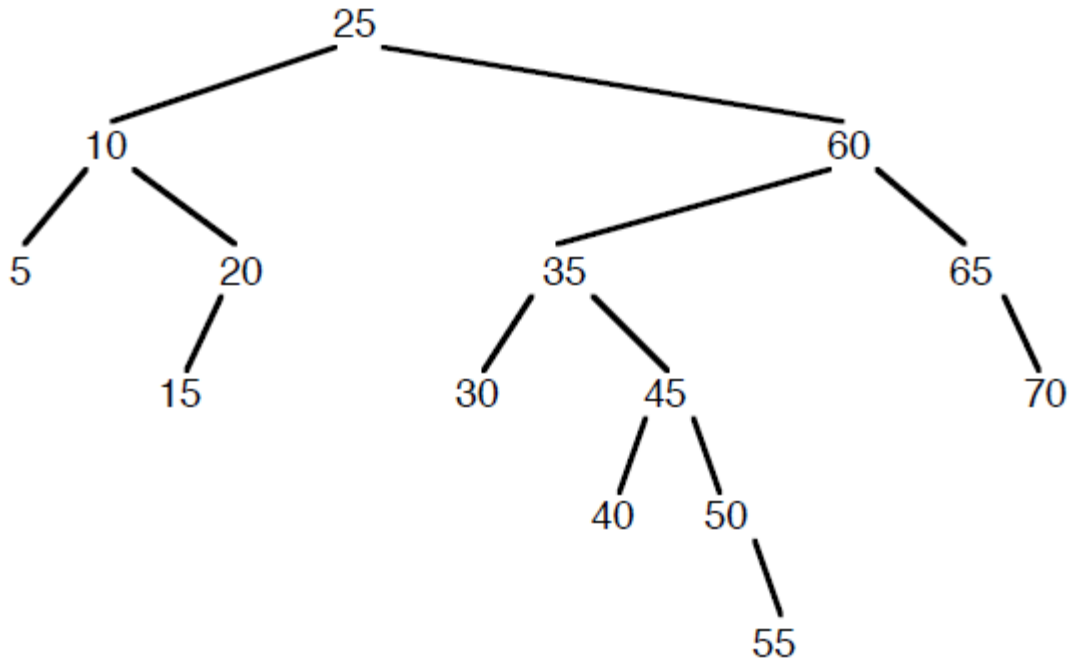
- Pour trouver le père est simple. Il suffit de prendre la partie entière de l'indice diviser par 2 et nous avons son indice dans le tableau.
- Avec le modèle d'implantation dynamique, retrouver le père est un peu plus compliqué que cette opération:

indiceDuPereDe(i) = [i/2]

3- Arbres binaires

Exercices d'application.

Simuler tous les parcours sur cette arbre binaire.



Gestion des fichiers

D- Gestion de fichiers

Introduction

Nous n'avons manipulé que des données en mémoire jusqu'à présent, ce qui nous empêchait de les stocker de manière permanente.

Maintenant nous allons voir comment conserver des informations de manière permanente à l'aide des fichiers.

D-Gestion de fichiers

Introduction

Pour pouvoir manipuler (lire/ecrire) un fichier un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit de la mémoire où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture).

- Ces informations sont rassemblées dans une structure **FILE** , est défini dans ***stdio.h.***

Une objet de type FILE * est appelé *flot de données* (en anglais, stream).

D-Gestion de fichiers

Introduction

- Pour manipuler un fichier, on utilise une variable **pointeur** de type « **FILE** ».

FILE *p_fichier;

La variable « p_fichier » contiendra l'adresse en mémoire-tampon du début du fichier.

D-Gestion de fichiers

Introduction

Trois flots standard peuvent être utilisés en C sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- stdin (standard input) : unité d'entrée (par défaut, le clavier) ;
- stdout (standard output) : unité de sortie (par défaut, l'écran) ;
- stderr (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

D-Gestion de fichiers

Ouverture de fichiers

La fonction fopen

Cette fonction ouvre un fichier et renvoie un pointeur FILE sur le fichier ouvert. **fopen** est définie dans le fichier stdio.h par :

FILE *fopen (char ***nom**, char ***mode**)

- **nom** est une chaîne de caractères contenant le nom du fichier sur la mémoire permanente

D-Gestion de fichiers

Ouverture d'un fichier

- **mode** chaîne de caractère qui désigne le mode d'accès du fichier.
  - **"r"** (read) : lecture (si le fichier existe)
  - **"w"** (write) : écriture (le fichier est écrasé s'il existe et s'il n'existe pas, il est crée)
  - **"a"** (append) : écriture à la fin d'un fichier existant. Le fichier peut ne pas exister, dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

D-Gestion de fichiers

Ouverture d'un fichier

Exemple

```
FILE *fp, fl*;
fp  = fopen("C:/Users/Documents/myData.txt", "w");
fl  = fopen("info.txt", "r");
```

- Ouverture en écriture du fichier myData.txt dans le dossier Documents.
- Ouverture en mode lecture du fichier info.txt dans le répertoire courant.

Gestion de fichiers

Fermeture d'un fichier avec fclose

La fonction fclose permet de fermer le flot qui a été associé à un fichier par la fonction fopen. Sa définition dans le fichier stdio.h est.

int fclose(FILE *pfichier) ;

Il faut toujours fermer un fichier après l'avoir utilisé Afin de libérer la mémoire

D-Gestion de fichiers

Écriture-lecture de fichiers textes

- La fonction **fprintf,** analogue à printf, permet d'écrire des données dans un fichier. Le prototype de la fonction est:

int fprintf(FILE *_fich_, char *_format_,…) ;

Les spécifications de format sont ici les mêmes que celles de printf .

Exemple
```
// a est une variable entier
fprintf(fp, "%d", a);
```

```
#include <stdio.h>
int main(){
  int a;   float b;
   a=5;  b = 8.34;
  // Déclaration d'un pointeur FILE  qui recevra l'adresse du fichier ouvert
   FILE *monfichier;
    /* ouverture du fichier en mode écriture    SYNTAXE GENERALE
      File *fopen(char *nom, char * mode )    */
      monfichier = fopen("mesDonnees.txt","w");
```

```
 if (monfichier != NULL){
        printf("Success! Fichier Ouvert\n");
      //Ecriture du fichier a l'aide du flot monfichier
        fprintf(monfichier,"%d\n",  a);
        fprintf(monfichier,"%f\n", b);
        fclose(monfichier);   //fermeture du fichier
}
else {
        printf("Echec d'ouverture du Fichier\n");
}
return 0;
}
```

D-Gestion de fichiers

Écriture-lecture de fichiers textes

- La fonction **fscanf,** analogue à scanf, permet de lire des données dans un fichier. Le prototype de la fonction est:

int fscanf(FILE *_fich_, char *_format_,…) ;

Les spécifications de format sont ici les mêmes que celles de scanf.

Exemple:

```
//a est un entier
fscanf(fichier,"%d",&a);
```

```
#include<stdio.h>
#include<stdlib.h>
int main() {
FILE *fp;
int a;
float b;
fp = fopen("data.txt", "r");
if(fp == NULL) {
        printf("Erreur Ouverture du fichier\n");
        exit(1);
 }
```

/*lecteur selon le format des données dans le fichier Data.txt, EOF est la valeur retournée par fscanf pour indiquer la fin du fichier.*/

```
while( fscanf(fp, " %d   %f \n" , &a, &b) != EOF ) {
        printf("%d  %f\n", a,b);
}
fclose(fp);
return 0;
}
```

TP

Déclarer un type qui permettent de stocker un étudiant caractérisé par son nom, prénom, sa date de naissance.

- Ecrire une fonction **LireEtudiant()** qui permet de lire au clavier et retourner les informations d'un étudiant.
- Dans la fonction main(), l'utilisateur saisie d'abord le nombre d'étudiants puis entre les données.
- Faire une fonction **WriteFile()** qui enregistre les données étudiants lues dans un fichier donnees.txt. Chaque ligne du fichier contiendra les informations d'un étudiant.