

Module Informatique 2

Algorithmique et Programmation Avancées

Gilles TALADOIRE

Juillet 2001

Université de La Nouvelle Calédonie

Plan

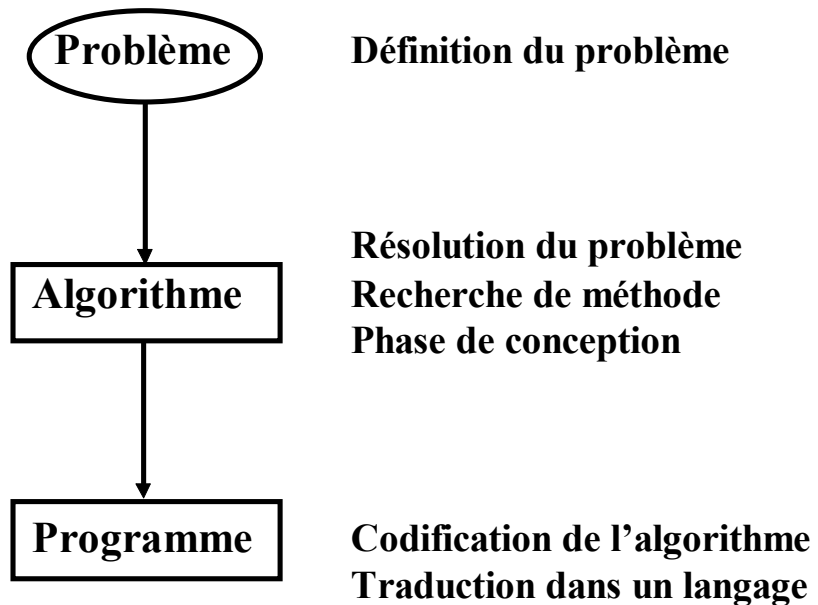
Programmation	3
Structures de données	4
Type abstrait algébrique	7
Langages et Types de données abstraits	13
Structures séquentielles	15
Les listes	15
Liste itérative	16
Liste récursive	19
Extensions du type liste	21
Récursivité	23
Représentation des listes	25
Représentation contigüe	25
Pointeurs et variables dynamiques	26
Représentation chaînée	31
Les piles	36
Les files	38
Structure arborescente : Les arbres	44
Définitions	48
Les arbres binaires	53
Propriétés	56
Représentation des arbres binaires	57
Parcours des arbres binaires	60
Arbres binaires de recherche	73
Recherche d'un élément	74
Ajout d'un élément	74
Ajout aux feuilles	74
Ajout à la racine	76
Suppression d'un élément	79

Références :

Problèmes classiques sur les structures de données de base que l'on retrouve dans la plupart des livres d'algorithmique

Programmation

Programmer consiste à rechercher et à exprimer, pour un problème donné, un algorithme compréhensible (exécutable) par l'ordinateur



Structures de données

Algorithms

+

Data Structures

=

Programs

(Livre de Wirth - Prentice Hall 1976)
(Algorithmes et Structures de données - Eyrolles 1987)

Structure de données

Définition

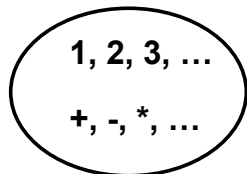
Modèle décrivant le comportement d'un ensemble d'éléments (d'informations) muni de propriétés logiques

Organisation entre les éléments choisie pour permettre de créer, exploiter, modifier cet ensemble au meilleur coût en place et en temps.

Spécifications

Vue externe
(Utilisateur)

Valeurs
Opérations applicables



Vue interne
(Concepteur)

Codage
Implantation

00000001
00000010
00000011

ET, OU, NON

Exemple : pour les entiers

----> 3 niveaux de spécification :

Fonctionnelle	<---->	Problème
Logique	<---->	Algorithme
Physique	<---->	Programme

Structure de données

Spécification fonctionnelle

Description des opérations possibles sur la structure avec leurs propriétés et leurs restrictions
(Cahier des charges, ce qu'il faut faire)

Description logique

Construction de l'organisation de la structure pour que les primitives définies soient réalisables et efficaces.

Construction des algorithmes des primitives avec contrôle de leur conformité aux spécifications.

Représentation physique

Implantation de la structure et des opérations dans un langage de programmation

Spécifications fonctionnelles par

Type abstrait algébrique

Spécifier : construire un modèle abstrait et formuler une description précise

----> recours aux Types Abstrait Algébriques (T.A.A.)

Spécifier une structure de donnée comme un type abstrait consiste à définir mathématiquement :

- le domaine des valeurs,
- l'ensemble des opérations applicables sur ces valeurs

Un type abstrait est algébrique si la sémantique de ses opérateurs (leur comportement) s'exprime par un système d'équations explicitant les liens entre les diverses opérations (utilisé ultérieurement pour vérifier la validité de l'implémentation).

Signature du type

Interface des opérations
Description syntaxique

Axiomes

Système d'équations

Type Abstrait algébrique

Deux catégories d'opérateurs (ou de primitives) :

- les générateurs : résultat du type spécifié,
- les fonctions de consultation ou d'accès (ou observateurs) : résultat d'un autre type

Les constructeurs sont des générateurs particuliers grâce auxquels toute valeur du type peut être obtenue (ils initialisent ou "augmentent" de manière élémentaire une donnée du type).

Une opération sans argument est une constante.

Les axiomes définissent le comportement des autres opérateurs à l'aide des constructeurs.

Le système d'axiomes doit être :

- non contradictoire (consistance),
- complet (complétude suffisante).

Type Abstrait algébrique

Exemple : Type abstrait algébrique "entier infini"

Type	Entier
Opérations (signature) (Interface)	
	ZERO → entier
	SUCC(entier) → entier
	+(entier, entier) → entier
Axiomes	
	zéro + x = x
	succ(x) + y = succ(x + y)

Tous les opérateurs sont des générateurs.

ZERO et SUCC sont des constructeurs.

On trouve différentes notations pour décrire la signature :

ZERO	:	→ entier
SUCC	:	entier → entier
+	:	entier X entier → entier

Exercice : Ecrire la signature du type Booléen

Type Abstrait algébrique

Réutilisation et hiérarchie dans les types abstraits

On se donne la possibilité, quand on définit un type, de réutiliser des types déjà définis.

La signature du type défini est l'union des signatures des types utilisés enrichie des nouvelles opérations.

On dit aussi que ce type hérite des propriétés des types qui le constituent (Notion d'Héritage).

On obtient donc une hiérarchie de types.

Opérations partiellement définies

Une opération peut ne pas être définie partout.

Cela dépend donc de son domaine de définition.

Ces "exceptions" seront traitées dans les axiomes sous différentes formes possibles.

Exemple : Si on définit la division entière
DIV(entier, entier) → entier

On écrira :

- soit l'axiome :

x div zéro = Erreur

- soit l'axiome (ou pré-condition) :

x div y est_défini_ssi y ≠ 0

Exemple

Type Vecteur

Utilise Entier, Element, Booléen

Opérations

vect : Entier X Entier → Vecteur
changer_ième : Vecteur X Entier X Elément → Vecteur
ième : Vecteur X Entier → Elément
init : Vecteur X Entier → Booléen
borneinf : Vecteur → Entier
bornesup : Vecteur → Entier

Pré_conditions

ième(v, i) **est_défini_ssi**

borneinf(v) ≤ i ≤ bornesup(v) & init(v, i) = vrai

Axiomes

borneinf(v) ≤ i ≤ bornesup(v)

⇒ ième(changer_ième(v, i, e), i) = e

borneinf(v) ≤ i ≤ bornesup(v)

& borneinf(v) ≤ j ≤ bornesup(v) & i ≠ j

⇒ ième(changer_ième(v, i, e), j) = ième(v, j)

init(vect(i, j), k) = faux

borneinf(v) ≤ i ≤ bornesup(v)

⇒ init(changer_ième(v, i, e), i) = vrai

borneinf(v) ≤ i ≤ bornesup(v) & i ≠ j

⇒ init(changer_ième(v, i, e), j) = init(v, j)

borneinf(vect(i, j)) = i

borneinf(changer_ième(v, i, e)) = borneinf(v)

bornesup(vect(i, j)) = j

bornesup(changer_ième(v, i, e)) = bornesup(v)

Avec v : Vecteur; i, j, k : Entier; e : Elément

Quels sont les générateurs, constructeurs, observateurs ?

Type Abstrait algébrique

Critère pour savoir si on a écrit suffisamment d'axiome (Complétude) :

Peut-on déduire de ces axiomes le résultat de chaque observateur sur son domaine de définition ?

Pour vérifier que ces axiomes ne sont pas contradictoire (Consistance) :

Le résultat de chaque observateur est-il unique ?

Langages et Types de données abstraits

Il existe des outils informatiques d'aide à la spécification pour vérifier complétude et consistance.

Mais la plupart des langages ne permettent pas de traduire l'ensemble des spécifications formelles.

Certains langages sont plus adaptés que d'autres, nous pouvons citer :

- le langage MODULA 2 (et ses modules),
- le langage ADA (et ses packages),
- les langages orientés objets : Smalltalk, Eiffel mais surtout **C++** et **JAVA**
- Turbo-Pascal (et ses unités).

Il faudra donc établir une correspondance entre les types abstraits utilisés pour concevoir l'algorithme et les types du langage de programmation.

Nous étudierons les représentations concrètes classiques :

- les structures séquentielles (listes, piles, files) en représentation contigüe ou chaînée,
- les arbres.

Structure de données complexe qui restera à étudier : les graphes.

Langages et Types de données abstraits

Les spécifications d'un type abstrait utilisent une notation fonctionnelle.

Il faudra parfois transformer ces fonctions en procédures lors du passage aux algorithmes

Exemple :

changer_ième : Vecteur X Entier X Élément → Vecteur

Fonction Changer_ième

(v : Vecteur; i : Entier; e : Element) : Vecteur
où v, i et e sont des paramètres données

se transformera en

Procédure Changer_ième

(v : Vecteur; i : Entier; e : Element)
où v est un paramètre donnée/résultat,
i et e des paramètres données

Car

- d'une part une fonction ne rend qu'une valeur de type élémentaire,
- d'autre part pour une question d'efficacité, cette formulation minimise les transferts en mémoire.

Structures séquentielles

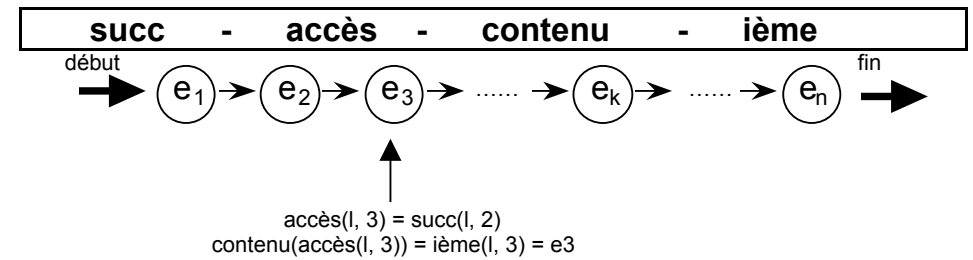
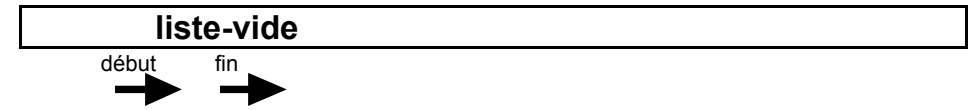
Les listes

Liste

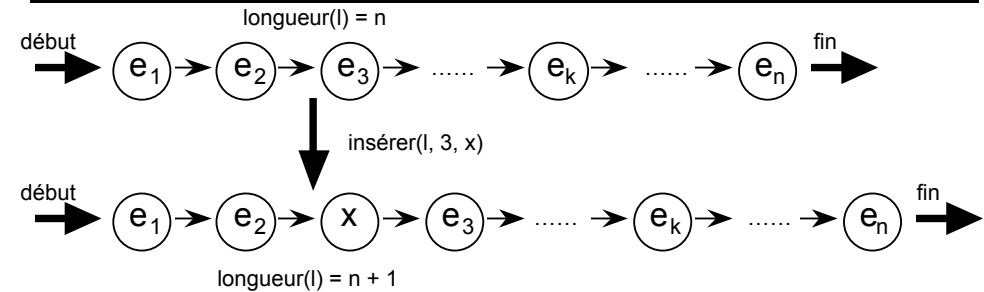
Suite finie d'éléments ordonnés selon leur place



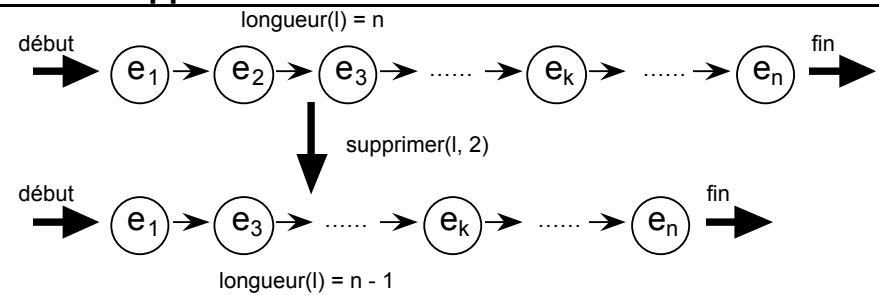
Liste itérative



longueur Insérer



Supprimer



Liste itérative

Type Liste, Place

Utilise Entier, Element

Opérations

Liste-vide : \rightarrow Liste
Accès : Liste X Entier \rightarrow Place
Contenu : Place \rightarrow Elément
ième : Liste X Entier \rightarrow Elément
Longueur : Liste \rightarrow Entier
Supprimer : Liste X Entier \rightarrow Liste
Insérer : Liste X Entier X Elément \rightarrow Liste
Succ : Place \rightarrow Place

Pré_conditions

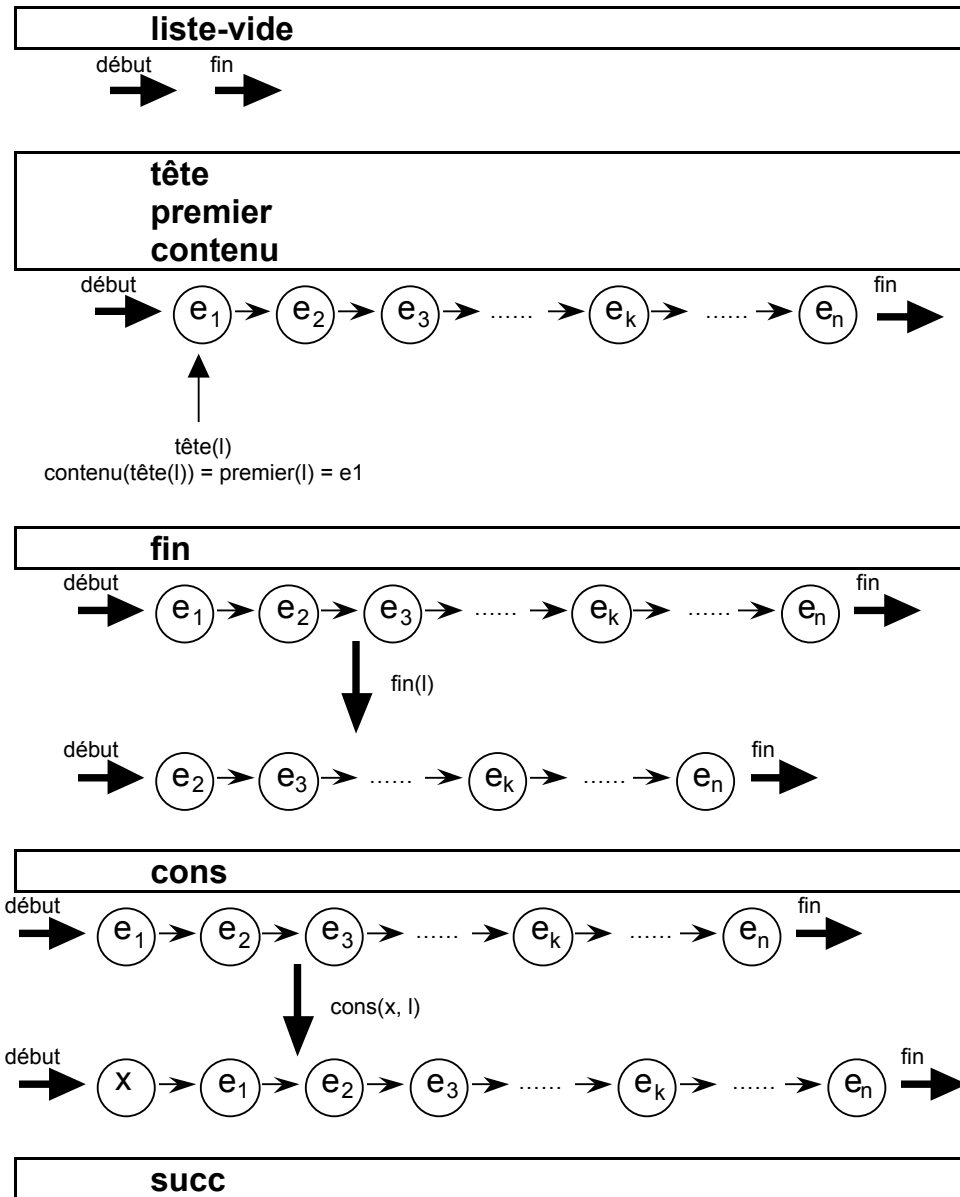
accès(l, k) **est défini ssi** $1 \leq k \leq \text{longueur}(l)$
supprimer(l, k) **est défini ssi** $1 \leq k \leq \text{longueur}(l)$
insérer(l, k, e) **est défini ssi** $1 \leq k \leq \text{longueur}(l) + 1$
{k = longueur(l) + 1 correspond à l'ajout en fin de liste}
ième(l, k) **est défini ssi** $1 \leq k \leq \text{longueur}(l)$

Liste itérative

Axiomes

$\text{longueur}(\text{liste-vide}) = 0$
 $l \neq \text{liste-vide} \ \& \ 1 \leq k \leq \text{longueur}(l)$
 $\Rightarrow \text{longueur}(\text{supprimer}(l, k)) = \text{longueur}(l) - 1$
 $1 \leq k \leq \text{longueur}(l) + 1$
 $\Rightarrow \text{longueur}(\text{insérer}(l, k, e)) = \text{longueur}(l) + 1$
 $l \neq \text{liste-vide} \ \& \ 1 \leq k \leq \text{longueur}(l)$
 $\Rightarrow \text{ième}(l, k) = \text{contenu}(\text{accès}(l, k))$
 $l \neq \text{liste-vide} \ \& \ 1 \leq k \leq \text{longueur}(l) \ \& \ 1 \leq i < k$
 $\Rightarrow \text{ième}(\text{supprimer}(l, k), i) = \text{ième}(l, i)$
 $l \neq \text{liste-vide} \ \& \ 1 \leq k \leq \text{longueur}(l) \ \& \ k \leq i \leq \text{longueur}(l) - 1$
 $\Rightarrow \text{ième}(\text{supprimer}(l, k), i) = \text{ième}(l, i + 1)$
 $1 \leq k \leq \text{longueur}(l) + 1 \ \& \ 1 \leq i < k$
 $\Rightarrow \text{ième}(\text{insérer}(l, k, e), i) = \text{ième}(l, i)$
 $1 \leq k \leq \text{longueur}(l) + 1 \ \& \ k = i$
 $\Rightarrow \text{ième}(\text{insérer}(l, k, e), i) = e$
 $1 \leq k \leq \text{longueur}(l) + 1 \ \& \ k < i \leq \text{longueur}(l) + 1$
 $\Rightarrow \text{ième}(\text{insérer}(l, k, e), i) = \text{ième}(l, i - 1)$
 $l \neq \text{liste-vide} \ \& \ 1 \leq k \leq \text{longueur}(l)$
 $\Rightarrow \text{succ}(\text{accès}(l, k)) = \text{accès}(l, k + 1)$

Liste récursive



Liste récursive

Type Liste, Place

Utilise Élément, Entier

Opérations

liste-vide	:	\rightarrow Liste
tête	:	Liste \rightarrow Place
fin	:	Liste \rightarrow Liste
cons	:	Élément X Liste \rightarrow Liste
premier	:	Liste \rightarrow Élément
contenu	:	Place \rightarrow Élément
succ	:	Place \rightarrow Place

Pré_conditions

tête(l) **est défini ssi** $l \neq$ liste-vide
 fin(l) **est défini ssi** $l \neq$ liste-vide
 premier(l) **est défini ssi** $l \neq$ liste-vide

Axiomes

$l \neq$ liste-vide \Rightarrow premier(l) = contenu(tête(l))
 fin(cons(e, l)) = l
 premier(cons(e, l)) = e
 $l \neq$ liste-vide \Rightarrow succ(tête(l)) = tête(fin(l))

Exercice

Ces deux formulations sont équivalentes :

- Exprimer premier, fin et cons en fonction de ième, insérer et supprimer,
- Exprimer ième, insérer, supprimer et longueur en fonction de premier, fin et cons.

Extensions du type liste

Copie d'une liste

Obtenir une copie d'une liste, dupliquer une liste

copier : Liste \rightarrow Liste

Ne pouvant effectuer d'affectations sur des listes, nous avons besoin de cette fonction.

Concaténation de deux listes

Construire une liste en mettant deux listes bout à bout.

concaténer : Liste x Liste \rightarrow Liste

définie par les axiomes

avec les listes itératives :

$\text{longueur}(\text{concaténer}(l, l')) = \text{longueur}(l) + \text{longueur}(l')$
 $1 \leq i \leq \text{longueur}(l) \Rightarrow \text{ième}(\text{concaténer}(l, l'), i) = \text{ième}(l, i)$
 $\text{longueur}(l) + 1 \leq i \leq \text{longueur}(l) + \text{longueur}(l')$
 $\Rightarrow \text{ième}(\text{concaténer}(l, l'), i) = \text{ième}(l', i - \text{longueur}(l))$

avec les listes récursives :

$\text{concaténer}(\text{liste-vide}, l') = l'$
 $\text{concaténer}(\text{cons}(e, l), l') = \text{cons}(e, \text{concaténer}(l, l'))$

Extensions du type liste

Recherche d'un élément dans une liste

Rechercher si un élément est présent dans une liste et, dans ce cas, à retourner la place de cet élément

rechercher : Liste x Elément \rightarrow Place

Opération non définie si l'élément recherché n'est pas présent dans la liste.

On se donne donc l'opération :

est_présent : Liste x Elément \rightarrow Booléen

définie par les axiomes

$\text{est_présent}(\text{liste-vide}, e) = \text{faux}$
 $e = e' \Rightarrow \text{est_présent}(\text{cons}(e, l), e') = \text{vrai}$
 $e \neq e' \Rightarrow \text{est_présent}(\text{cons}(e, l), e') = \text{est_présent}(l, e')$

Rechercher est donc définie par les axiomes :

$\text{rechercher}(l, e) \text{ est_défini_ssi } \text{est_présent}(l, e) = \text{vrai}$
 $\text{est_présent}(l, e) = \text{vrai} \Rightarrow \text{contenu}(\text{rechercher}(l, e)) = e$

Exercice

En cas de répétition de e dans l, la spécification de rechercher ne précise pas de quelle occurrence de e on retourne la place.

Ecrire les axiomes pour rechercher la première occurrence dans les deux cas de liste étudiés.

Récurtivité

Une procédure (ou une fonction) est dite "réursive" quand elle effectue un appel à elle-même.

Exemple 1 : Fonction Factorielle

```
Fonction Fact(n : Entier) : Entier
Début
  Si n = 1
  Alors Rendre(1)
  Sinon Rendre( n * Fact(n - 1) )
Finsi
Fin
```

Exemple 2 : Fonction Longueur d'une liste chaînée

```
Fonction Longueur(L : Liste) : Entier
Début
  Si L = nil
  Alors Rendre( 0 )
  Sinon Rendre( 1 + Longueur(L↑.suivant) )
Finsi
Fin
```

Récurtivité

Principe :

Utiliser, pour décrire l'algorithme sur une donnée D, l'algorithme lui-même appliqué à un sous-ensemble de D ou une donnée D' plus petite.

ATTENTION

- il ne faut pas réappliquer l'algorithme à des données plus grandes,
- il faut un test de terminaison, qui correspond à un cas où la donnée est élémentaire et peut être traitée directement.

La complexité d'un algorithme récursif se démontre en général par récurrence.

Il existe toujours une version itérative d'un algorithme récursif, mais la transformation est parfois simple, parfois très compliquée.

Un algorithme récursif masque souvent la complexité en place (et en temps) de l'algorithme :

celui-ci utilise la pile qui va contenir tous les paramètres et toutes les variables locales, autant de fois qu'il y aura eu d'appels.

-----> N'utiliser la récurtivité que quand celle-ci est indispensable (Algorithme itératif trop compliqué)

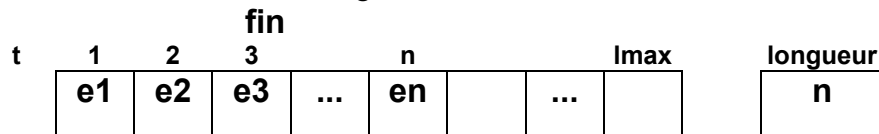
Exercice : Ecrire une fonction qui calcule le PGCD de 2 nombres a et b sous forme récursive et itérative.

Représentation des listes Représentation contigüe

Représentation d'une liste à l'aide d'un tableau :
la $i^{\text{ème}}$ case correspond à la $i^{\text{ème}}$ place de la liste

Remarque : La longueur maximale de la liste, l_{max} , doit être déterminée.

Type LISTET = type composé de
t : Tableau de 1 à l_{max} de Elément;
longueur : 0 .. l_{max}



Méthode relativement bien adaptée aux listes itératives,
Méthode mal adaptée aux listes récursives

Avantages

- Accès facile au $i^{\text{ème}}$ élément
- Insertion et suppression du dernier élément faciles
- Longueur facile à obtenir

Inconvénients

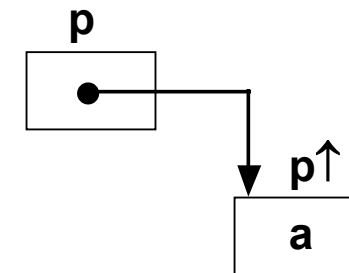
- Connaissance de la longueur maximale de la liste
- Insertion et suppression coûteuses (sauf à la fin)

Exercice : Ecrire les primitives des listes itératives à l'aide de cette représentation (Liste vide, Longueur, lème, Insérer, Supprimer).

Pointeurs et variables dynamiques

Un objet est repéré en mémoire par son adresse.

Nous allons maintenant utiliser des objets qui vont contenir des adresses, on les appelle des variables de type pointeur ou pointeurs.



p est une variable de type pointeur qui contient l'adresse de l'information **a**.

p : pointeur sur TypObjet

où TypObjet est le type (simple ou composé) de l'objet pointé.

Notations :

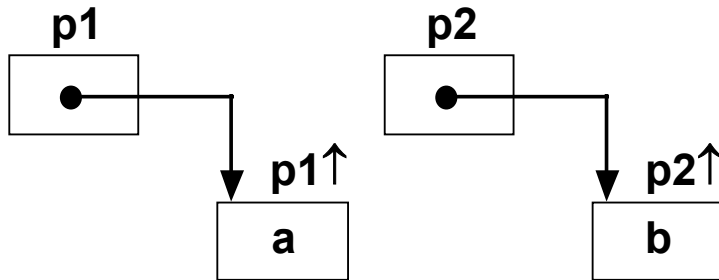
- **p** est donc une variable qui désigne l'adresse d'une autre variable.
- **p↑** désigne l'objet dont l'adresse est rangée dans **p**.

p et **p↑** peuvent être utilisées comme n'importe quelle variable mais ATTENTION !!!

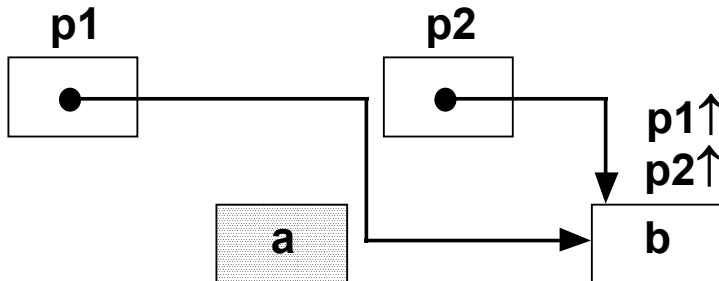
Pointeurs et variables dynamiques

Manipulation des pointeurs

Soient p1 et p2 deux variables de type pointeur.



Après $p1 \leftarrow p2$ on obtiendra :



Remarques :

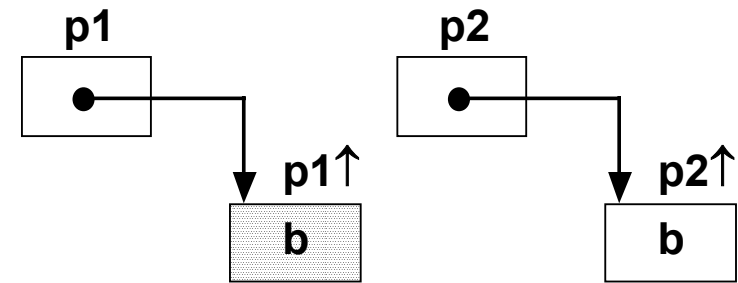
La donnée a n'est plus accessible car on a perdu son adresse qui se trouvait dans p1.

La donnée b est accessible par p1 ou p2.

Pointeurs et variables dynamiques

Manipulation des pointeurs

Après $p1↑ \leftarrow p2↑$ on obtiendra :



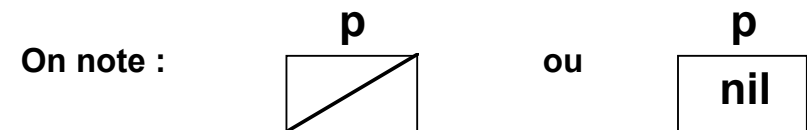
Remarques :

La donnée a n'existe plus.

b existe en deux exemplaires, à deux adresses différentes.

Valeur d'adresse "nil"

Quand un pointeur n'indique aucune adresse d'objet, on dit que $p = nil$



$p↑$ n'est défini que si $p \neq nil$

Pointeurs et variables dynamiques

Gestion de la mémoire dynamique
Utilisation des pointeurs

Grâce aux pointeurs, nous allons pouvoir créer et manipuler de nouvelles variables pendant l'exécution d'un algorithme, on parle alors de :

variables dynamiques

Pour cela, nous introduisons deux procédures pour gérer la mémoire dite dynamique (en opposition à la mémoire statique), appelée aussi le tas.

Procédure nouveau(p : pointeur)

où p est un paramètre résultat

Cette procédure a pour effet de réserver la place mémoire nécessaire pour une variable du type pointé et d'affecter cette adresse mémoire à p.

Si cette opération n'est pas possible (mémoire saturée), p aura la valeur nil.

Procédure libérer(p : pointeur)

où p est un paramètre donnée

Cette procédure a pour effet de libérer la place mémoire d'adresse p.

Remarques :

Après un libérer(p), p↑ n'a plus aucune signification.

Le programme doit absolument libérer les variables non utilisées sous peine d'épuisement de la mémoire.

Pointeurs et variables dynamiques

Exercice :

Faire un schéma avec les valeurs des différentes variables après chaque action.

p, q, r : pointeur sur caractère
c : caractères

début

nouveau(p)

nouveau(q)

nouveau(r)

c ← 'x'

p↑ ← 'y'

q↑ ← 'z'

r↑ ← c

p↑ ← q↑

q↑ ← r↑

ou libérer(q)

q ← p

q ← p

fin

Quelle est la différence entre les deux variantes ?

Représentation des listes

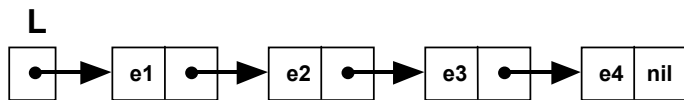
Représentation chaînée

On utilise des pointeurs pour chaîner entre eux les éléments successifs.

La liste sera alors déterminée par l'adresse de son premier élément.

```

Type    cellules =   type composé de
                        val : Elément;
                        lien : ↑cellules
                        end;
LISTEP = ↑cellules;
    
```



Avantages

- pas de longueur maximum
- opérations faciles à réaliser (parcours, insertion, suppression, concaténation)

Inconvénients

- besoin de place mémoire supplémentaire pour les pointeurs
- longueur et accès au i^{ème} élément nécessitent un parcours de la liste

Exercices : 1/ Ecrire les primitives des listes récursives à l'aide de cette représentation.

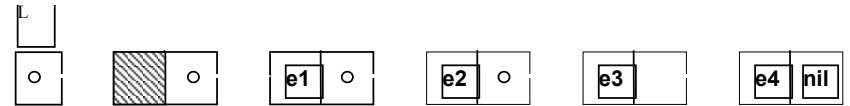
2/ Ecrire les primitives des listes itératives à l'aide de cette représentation.

Représentation des listes

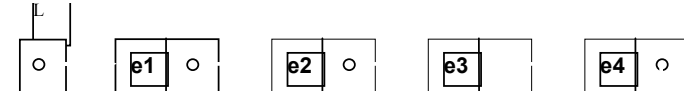
Représentation chaînée

Variantes

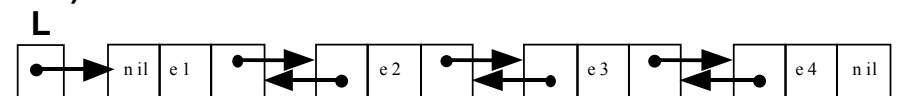
a) Définir la tête de la liste comme une cellule ordinaire : facilité pour l'insertion et la suppression en début de liste



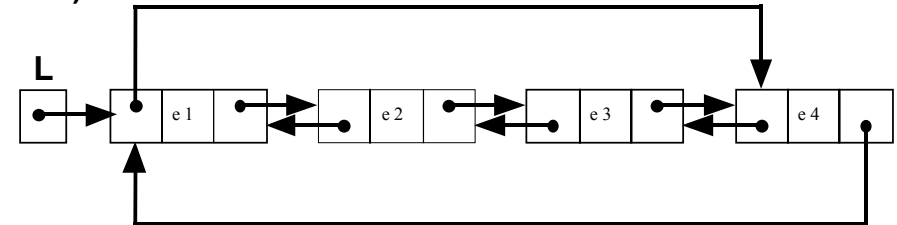
b) Listes circulaires



c) Listes doublement chaînées



d) Listes doublement chaînées circulaires



Augmentation de la place mémoire utilisée

Exercices sur les listes

Pour chacune des deux représentations des listes (contiguë et chaînée), écrire les procédures suivantes :

1/ Recherche d'éléments dans une liste

- Fonction Est_présent(L, E) : Booléen

Vrai si $E \in L$, Faux sinon

- Procédure Recherche(L, E, i) pour la représentation contiguë

Où i est l'indice trouvé, 0 si $E \notin L$

- Procédure Recherche(L, E, i, p) pour la représentation chaînée

Où i est le n° d'ordre de la cellule, p le pointeur sur la cellule,

Si $E \notin L$, $i = 0$ et $p = \text{nil}$

2/ Copie d'une liste

procédure Copier(L1, L2)

Copie de la liste L1 dans L2, L1 et L2 sont 2 listes identiques, L2 est supposé n'avoir aucune valeur

3/ Concaténation de deux listes

procédure Concaténer(L1, L2, L3)

$L3 = L1 + L2$ (les éléments de L1 suivis des éléments de L2)

Pour la représentation chaînée, deux versions sont possibles :

- sans duplication des éléments,
- avec duplication des éléments.

4/ Insertion et recherche dans une liste triée

Procédure Ins_trié(L, E)

Insertion de E 'à sa place' dans la liste L supposée triée en ordre croissant

Procédure Rech_trié(L, E, i) pour la représentation contiguë (cf. 1/)

Procédure Rech_trié(L, E, i, p) pour la représentation chaînée (cf. 1/)

5/ Fusion de deux listes triées

Procédure Fusion(L1, L2, L3)

L3 contient les éléments en ordre croissant des deux listes triées en ordre croissant L1 et L2. La procédure sera optimisée en tenant compte du fait que les listes d'origine (L1 et L2) sont triées.

Pour la représentation chaînée, deux versions sont possibles :

- sans duplication des éléments,
- avec duplication des éléments.

6/ Inversion d'une liste

Pour les deux représentations, écrire une procédure qui inverse l'ordre des éléments de cette liste : **Procédure Inversion(L)**

On donnera deux versions de cette procédure pour les listes chaînées (ni circulaire, ni doublement chaînée) :

- une avec duplication des cellules,
- une sans duplication des cellules.

Exemple :

$L = E1 E2 E3 E4$ devient $L = E4 E3 E2 E1$

POUR LES 3 EXERCICES SUIVANTS, PROPOSEZ LES STRUCTURES DE DONNEES LES PLUS ADAPTEES AU PROBLEME

7/ L'employé d'une banque saisit les opérations que vous avez effectuées depuis le début du mois. (on suppose que cette saisie consiste à rentrer les divers montants, nombres entiers, positifs ou négatifs selon la nature de l'opération). Lorsque vous consultez le distributeur de cartes bleues de la banque, vous voyez apparaître les 8 dernières opérations. On suppose de plus que les autres opérations sont perdues.

8/ Le problème du Amstramgram

On considère une ronde de n enfants. Ils veulent jouer à cache-cache, et donc déterminer qui devra coller, en utilisant le bon vieux principe du amstramgram.

Donnez deux structures de données permettant de répondre au problème (l'une contiguë et l'autre chaînée).

On suppose que la phrase magique comporte p syllabes.

Donnez pour chaque représentation une procédure permettant de récupérer le chat et la liste, dans l'ordre, des enfants tirés.

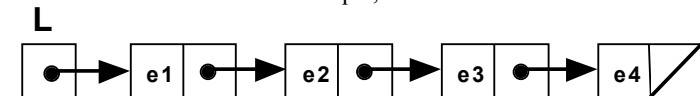
9/ Où l'on reparle de table ronde: Un certain nombre de personnes se réunissent autour d'une table. Chaque personne peut parler à son voisin de droite et de gauche, mais ne peut pas adresser directement la parole aux autres convives. Quelles est la structure de données adaptée à ce problème ? Donnez des procédures d'insertion et de suppression d'un convive autour de la table.

REVISIONS

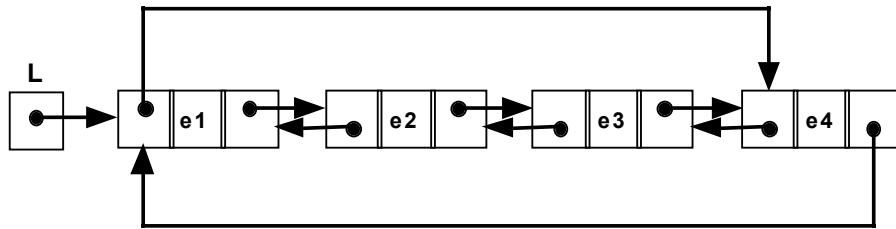
10/ Manipulation de listes chaînées (Examen 1ère session 1995)

a) Ecrire une fonction dernier qui rende le dernier élément d'une liste chaînée dans les deux cas suivants :

- utilisation d'une liste chaînée classique,



- utilisation d'une liste doublement chaînée circulaire.



Les types correspondants à ces deux représentations des listes seront rappelés.

On suppose maintenant qu'on utilise des listes circulaires doublement chaînées dans les questions suivantes.

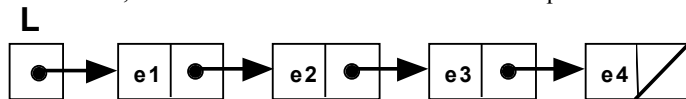
b) Ecrire les procédures pour effectuer les actions suivantes :

- ajouter un élément en tête de liste (procédure AjoutTête),
- ajouter un élément en fin de liste (procédure AjoutFin).

c) Ecrire une procédure qui recherche et affiche l'élément minimum de cette liste. La liste n'est pas ordonnée.

11/ Manipulation de listes chaînées (Examen 2ème session 1995)

Dans cet exercice, nous utiliserons une liste chaînée classique contenant des entiers.



a) Ecrire une procédure qui partage cette liste en deux listes :

- une qui contiendra les nombres pairs,
- une qui contiendra les nombres impairs.

On donnera deux versions de cette procédure "**Partage(L, Pair, Impair)**":

- une version avec duplication des cellules,
- une version sans duplication des cellules.

b) Ecrire une procédure qui supprime les multiples de 3 dans une liste d'entiers.

Les questions a) et b) sont indépendantes

Les piles

Cas particulier des listes :

Les insertions et suppressions se font à une seule extrémité.

Pile LIFO = Last In - First Out

Premier Entré - Dernier Sorti

Dernier Entré - Premier Sorti

Exemple : une pile d'assiette ...

Type Pile

Utilise Booléen, Elément

Opérations

pile-vidé :	→ Pile
empiler	: Pile X Elément → Pile
dépiler	: Pile → Pile
sommet	: Pile → Elément
est-vidé	: Pile → Booléen

Pré-conditions

dépiler(p)	est_défini_ssi	est-vidé(p) = faux
sommet(p)	est_défini_ssi	est-vidé(p) = faux

Axiomes

dépiler(empiler(p, e))	= p
sommet(empiler(p, e))	= e
est-vidé(pile-vidé)	= vrai
est-vidé(empiler(p, e))	= faux

Exemples d'utilisation en Informatique :

- Appels de procédures,
- Calcul d'expressions arithmétiques

Représentation des piles

Représentation contigüe

Type Pile = Type composé de
som : entier
elts : Tableau de 1 à lmax d'Elément
Fin

où som est l'indice du sommet (0 si la pile est vide).

Exercice :

Ecrire les primitives avec cette représentation.
(Attention aux débordements)

Représentation chaînée

Type Elt = Type composé de
val : Elément;
lien : ↑Elt
Fin
Pile = ↑Elt

où le sommet est le premier élément de la liste,
la liste vide est représentée par nil.

Exercice :

Ecrire les primitives avec cette représentation.

Les files

Cas particulier des listes : Les insertions se font à une extrémité, les accès et les suppressions à l'autre.

Pile FIFO = First In - First Out

Premier Entré - Premier Sorti

Dernier Entré - Dernier Sorti

Exemple : une file d'attente ...

Type File

Utilise Booléen, Elément

Opérations

file-vide	:	→ File
ajouter	:	File X Elément → File
retirer	:	File → File
premier	:	File → Elément
est-vide	:	File → Booléen

Pré-conditions

premier(f)	est_défini_ssi	est-vide(f) = faux
retirer(f)	est_défini_ssi	est-vide(f) = faux

Axiomes

est-vide(f) = vrai ⇒ premier(ajouter(f, e)) = e
est-vide(f) = faux ⇒ premier(ajouter(f, e)) = premier(f)
est-vide(f) = vrai ⇒ retirer(ajouter(f, e)) = file-vide
est-vide(f) = faux
⇒ retirer(ajouter(f, e)) = ajouter(retirer(f), e)
est-vide(file-vide) = vrai
est-vide(ajouter(f, e)) = faux

Exemple d'utilisation en Informatique :

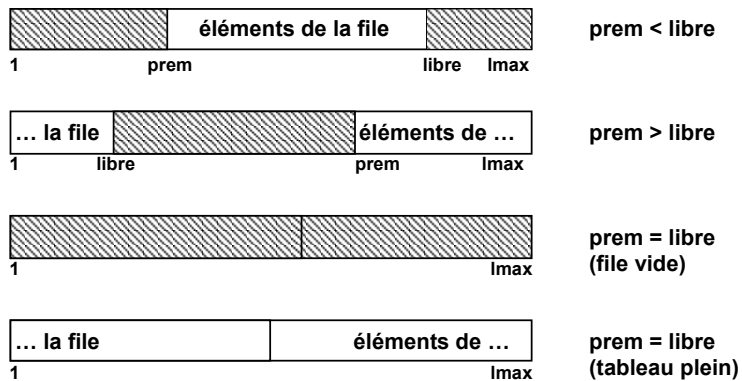
- Liste d'attente pour l'imprimante

Représentation des files

Représentation contigüe

Type File = Type composé de
 prem : entier
 libre : entier
 elts : Tableau de 1 à lmax d'Elément
Fin

où **prem** est l'indice du premier élément,
libre est l'indice de la prochaine case vide



Exercice :

Ecrire les primitives avec cette représentation.
(Attention aux débordements)

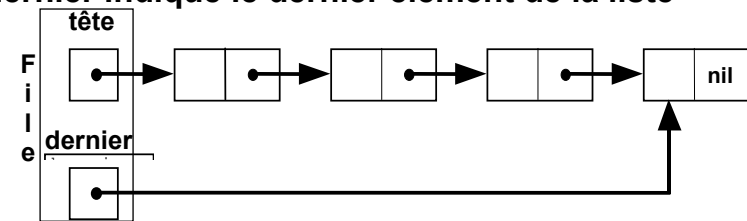
Il faudra différencier les deux cas où "prem = vide". Pour cela, si le cas *prem = libre* se produit pour une file vide (après un 'retirer'), celle-ci sera toujours initialisée comme une file vide avec *prem = 0* et *libre = 1*. Le cas où *prem = libre* correspondra donc toujours à une file pleine.

Représentation des files

Représentation chaînée

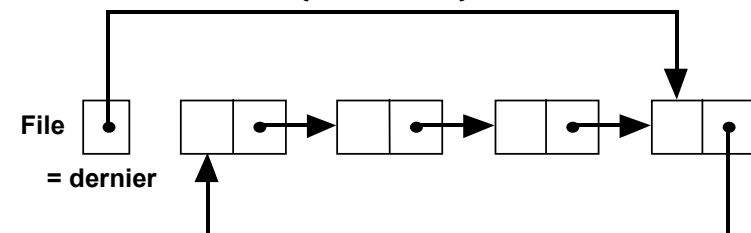
Type Elt = Type composé de
 val : Elément;
 lien : ↑Elt
Fin
File = Type composé de
 tête : ↑Elt;
 dernier : ↑Elt
Fin

où **tête** indique le premier élément de la liste,
dernier indique le dernier élément de la liste



OU variante

File = ↑Elt ; { = dernier }



Exercice :

Ecrire les primitives avec cette représentation.

Exercices de manipulation de piles et de files

1/ Editeur de texte

Un éditeur de texte doit mémoriser le texte sur lequel il travaille ainsi que l'emplacement du curseur. L'ensemble texte - curseur est parfois représenté à l'aide de deux piles G et D :

- G contient les caractères situés en début de texte (avant le curseur), le dernier étant en sommet de la pile,

- D contient les caractères de fin de texte, le premier en sommet de pile.

On suppose de plus que l'on dispose d'un caractère de fin de ligne, soit \$ ce caractère.

En utilisant les opérations du type abstrait pile (sommet, empiler, dépiler, est_vide), décrivez les opérations suivantes :

- Insérer un caractère C et positionnement du curseur après le caractère inséré,
- Effacer le caractère qui suit le curseur,
- Avancer le curseur d'un caractère,
- Reculer le curseur d'un caractère,
- Rechercher la première occurrence d'un caractère donné C à partir de la position courante, et positionner le curseur juste après cette occurrence si elle existe, et en fin de texte sinon,
- Aller au début de la ligne (curseur devant le premier caractère de la ligne),
- Effacer la ligne dans laquelle se trouve le curseur et se placer au début de la ligne suivante.

2/ Editeur de texte

Lorsqu'on utilise un éditeur de texte, on dispose d'une touche qui permet d'effacer le caractère que l'on vient de frapper (par exemple "BackSpace").

Notons '#' ce caractère.

Une autre touche permet de tout effacer jusqu'au début de la ligne.

Notons '%' ce caractère.

La fin d'une ligne sera indiquée par la touche '\$'

Exemple :

```
"Jem# m'euh%Je m'#euh## suit#s trop#mp&#é$"
```

sera lu comme

```
"Je me suis trompé"
```

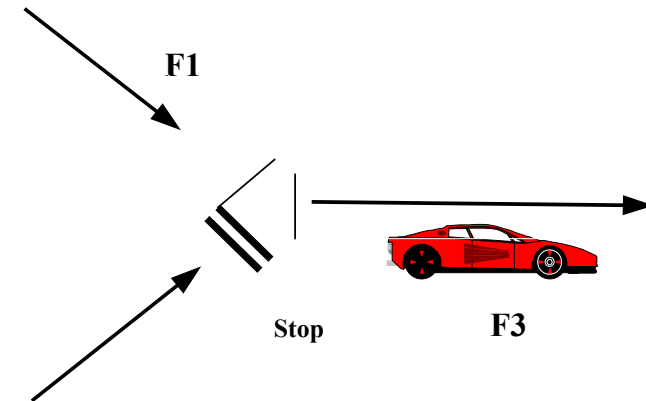
Ecrire un algorithme qui permette de lire une ligne de texte avec ce mécanisme en utilisant une pile. On écrira deux procédures une procédure LireLigne(P) et une procédure EcrireLigne(P).

On supposera que l'on ne saisit qu'une seule ligne de texte, de longueur quelconque.

3/ Utilisation des files : Simulation d'un croisement

Pour simuler un croisement routier, à sens unique, on utilise 3 files F1, F2 et F3 représentant, respectivement, les voitures arrivant sur les routes R1 et R2, et les voitures repartant sur la route R3. Chaque élément d'une file est donc une voiture.

La route R2 a un STOP, les voitures de la file F2 ne peuvent avancer que si il n'y a aucune voiture sur la route R1, donc dans la file F1.



En utilisant les primitives sur les files (est-vidé, ajouter, retirer, premier), écrire un algorithme qui simule ce carrefour.

L'état des files pourra être modifié à tout moment par un processus extérieur (voiture qui arrive) dont on ne s'occupera pas. On suppose donc qu'il existe une procédure *arrivée(F1, F2)*, dans laquelle F1 et F2 sont des paramètres données/résultats. L'appel à cette procédure simulera l'arrivée d'une ou plusieurs voitures sur les routes R1 et R2.

Le processus étant infini, on utilisera une boucle sans fin pour la simulation. Il sera fait un appel à la procédure *arrivée* par itération c'est à dire que la structure du programme sera la suivante :

```
Tantque Vrai faire
    arrivée(F1, F2)
    ...
fintantque
```

4/ Utilisation des piles

En utilisant les procédures et fonctions de manipulation de piles vues en cours, écrire une procédure qui enlève le $i^{\text{ème}}$ élément d'une pile et laisse les autres éléments inchangés : **Procédure Enlever(P, i)**

Exemple :

Après un Enlever(P, 3), la pile	devient la pile
sommet → E1	
E2	sommet → E1
E3	E2
E4	E4

Remarques :

Attention, i peut avoir une valeur incorrecte (au delà de la taille de la pile).

Une variable locale Q de type pile pourra être utilisée.

Attention, la méthode d'implantation de la pile (tableaux ou listes chaînées) n'est pas connue, il faut obligatoirement passer par les opérations de manipulation de piles. Le type *Pile* est supposé connu.

Les piles contiennent des éléments de type *Elément*.

Rappels : Procédures et fonctions de manipulation de piles

Procédure Pilevide(P)

Fonction Estvide(P) : Booléen

Procédure Empiler(P, x)

Procédure Dépiler(P)

Procédure Sommet(P, x)

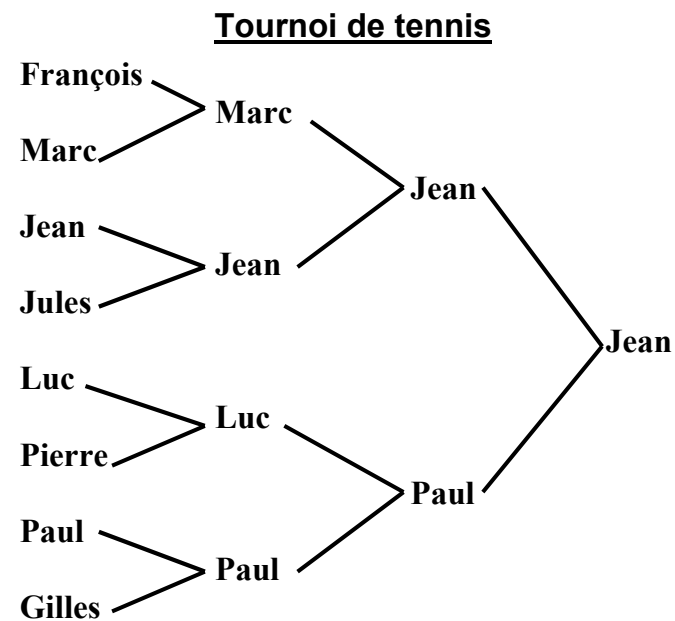
Structure arborescente :

Les arbres

Un arbre est un ensemble de **noeuds** organisés de façon hiérarchique à partir d'un noeud particulier appelé **racine**.

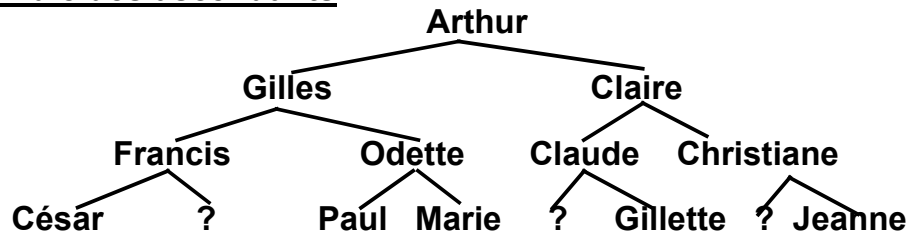
Très utilisés, surtout en Informatique

Exemples :

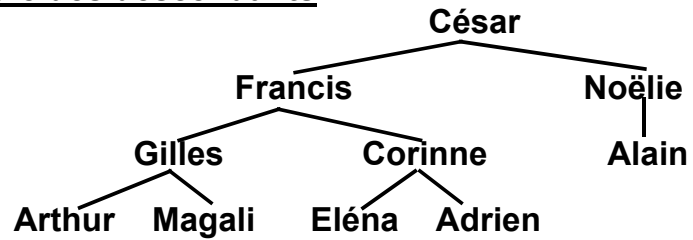


Arbre généalogique

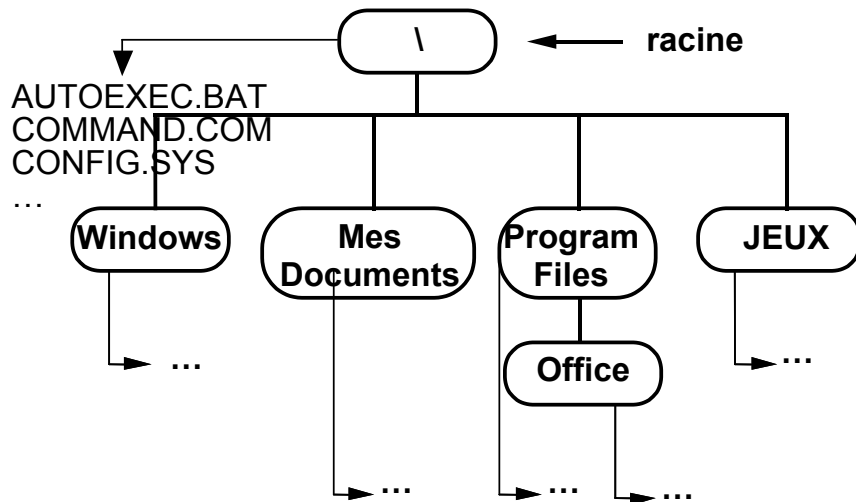
Arbre des ascendants



Arbre des descendants



Hiérarchie de Répertoires (pour tous les systèmes d'exploitation)

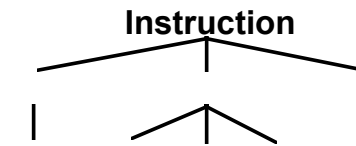


Structure arborescente Exemples

Instruction Pascal "IF" (Compilation)

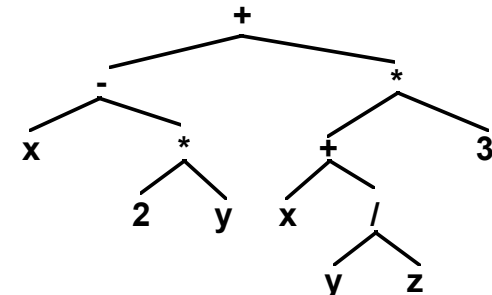
```

IF C
THEN BEGIN
        A1; A2; A3
      END
ELSE B
  
```



Expression arithmétique (Compilation)

$(x - (2 * y)) + ((x + (y / z)) * 3)$



Structure arborescente Exemples

Structure de données (Compilation)

Type Etudiant = Type composé de

Scolarité : Type composé de
 Université : ...
 Dipl-prep : ...
 Année : ...

Fin

Identité : Type composé de

Nom : ...
 Prénom : ...
 Naiss : Type composé de

Date : Type composé de
 Jour : ...
 Mois : ...
 Année : ...

Fin

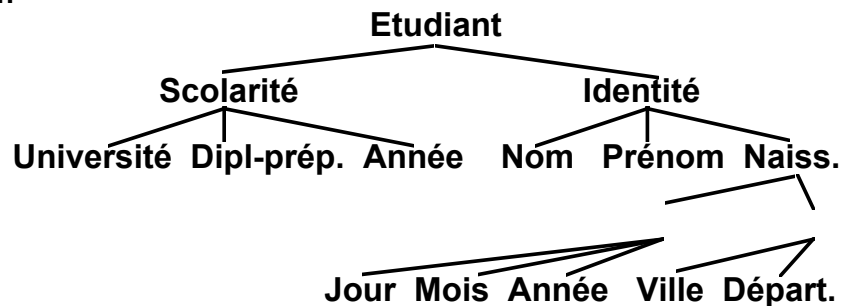
Lieu : Type composé de
 Ville : ...
 Départ. : ...

Fin

Fin

Fin

Fin



Structure arborescente

Définitions

Une structure d'arbre d'un type de base T est :

- soit la structure vide,
- soit un noeud de type T associé à un nombre fini de structures d'arbre disjointes du type de base T appelées sous-arbres.

Définition récursive ----> récursivité : propriété des arbres et des algorithmes qui les manipulent

Une liste est un cas particulier des arbres (arbre dégénéré) : tout noeud a au plus un sous-arbre.

Fils ou Descendant direct

Père ou Ascendant (ou Ancêtre) direct

Frères : deux noeuds ayant le même père

Si un noeud n'a pas de fils (donc pas de descendant), c'est un noeud terminal (ou extérieur) ou une feuille.

Un noeud qui n'est pas terminal est appelé noeud interne (ou intérieur).

Le degré d'un noeud est le nombre de fils de ce noeud.

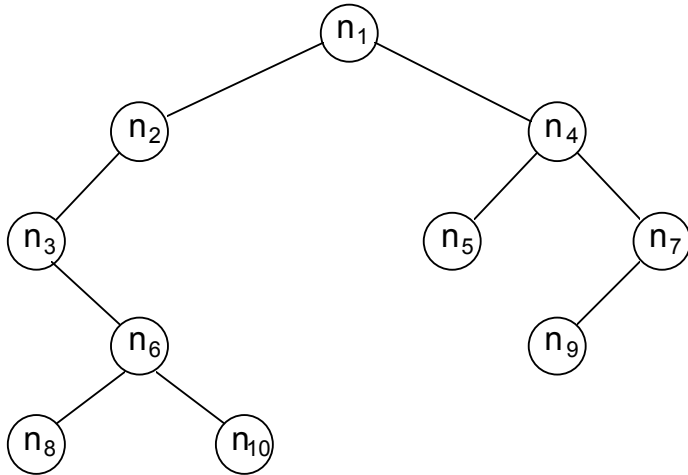
Le degré d'un arbre est le plus grand degré de ses noeuds.

La taille d'un arbre est le nombre de ses noeuds.

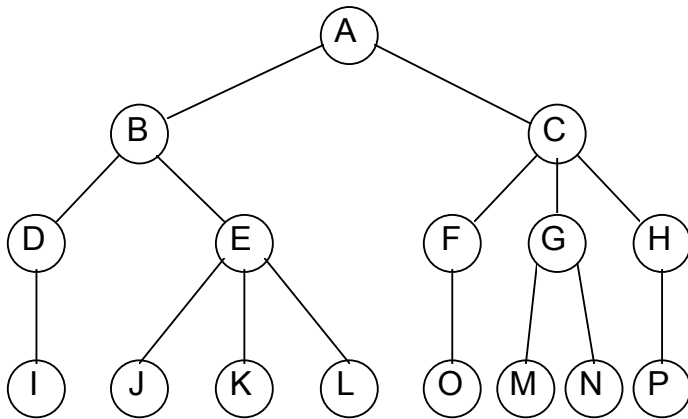
Structure arborescente

Exemples

Exemple 1 :



Exemple 2 :



Exercice : Donnez le degré et la taille de ces arbres

Structure arborescente

Définitions

La hauteur (ou la profondeur, le niveau, la longueur de chemin) d'un noeud est le nombre de branches ou arcs qu'il faut traverser pour aller de la racine à ce noeud.

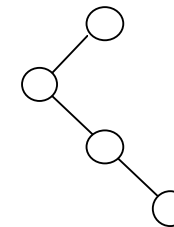
La hauteur (ou la profondeur) d'un arbre est la plus grande profondeur de ses noeuds.

La longueur de chemin (ou de cheminement) d'un arbre est la somme des longueurs de chemin de ses noeuds.

La longueur de cheminement interne est la somme des longueurs de cheminement de ses noeuds non terminaux.

La longueur de cheminement externe est la somme des longueurs de cheminement de ses feuilles.

Arbre dégénéré ou filiforme : Arbre dont chaque noeud a au plus un fils.

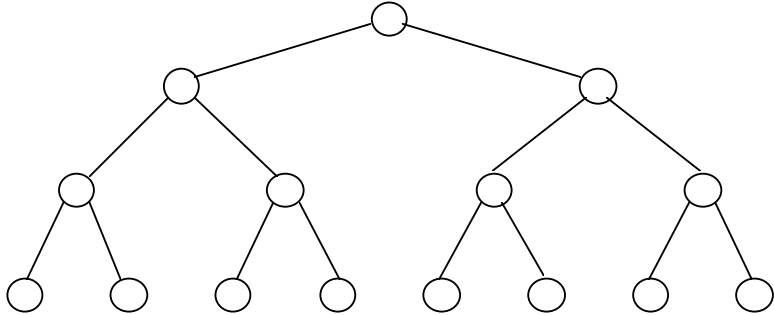


Exercice : Pour les arbres de la page précédente, donnez la hauteur de chaque noeud, la hauteur de l'arbre, la longueur de cheminement, la longueur de cheminement interne, la longueur de cheminement externe

Structure arborescente

Définitions

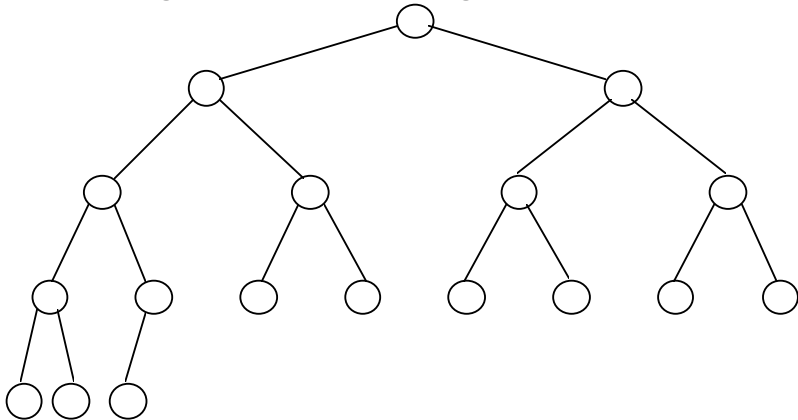
Arbre complet : Arbre dont chaque niveau est rempli.



Exercice : Donnez le nombre total de noeud pour un arbre complet de degré 2 de hauteur h

n =

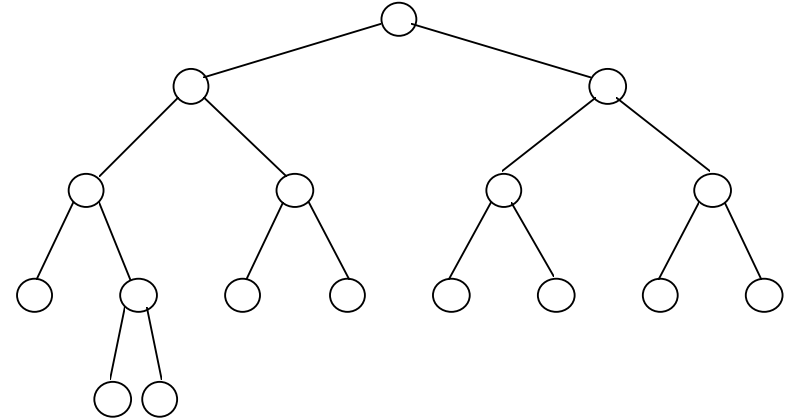
Arbre parfait : Arbre dont chaque niveau est rempli sauf éventuellement le dernier, dans ce cas les noeuds (feuilles) sont groupés le plus à gauche possible.



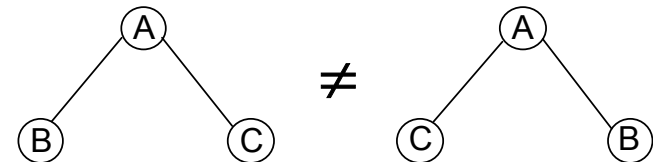
Structure arborescente

Définitions

Arbre localement complet : Arbre A dont tous les noeuds non terminaux ont degré(A) fils.



Un **arbre ordonné** est un arbre dans lequel les branches de chaque noeud sont ordonnées.



Un **arbre binaire** est un arbre ordonné de degré 2.

Structure très utilisée, on peut même représenter de cette manière des arbres plus généraux.

Structure arborescente

Les arbres binaires

Définition

Un **arbre binaire** est soit vide \emptyset , soit de la forme $B = \langle o, B_1, B_2 \rangle$ où B_1 et B_2 sont des arbres binaires disjoints (appelés resp. sous-arbre gauche et sous-arbre droit), o est un noeud appelé racine.

$$B = \emptyset + \langle o, B, B \rangle$$

Spécification du type abstrait arbre binaire

Type Arbre

Utilise Noeud, Elément

Opération

Arbre-Vide : \rightarrow Arbre
 $\langle _, _, _ \rangle$: Noeud x Arbre x Arbre \rightarrow Arbre
Racine : Arbre \rightarrow Noeud
Gauche : Arbre \rightarrow Arbre
Droite : Arbre \rightarrow Arbre
Contenu : Noeud \rightarrow Elément

Pré-conditions

Racine(B) **est défini ssi** $B \neq$ Arbre-Vide
Gauche(B) **est défini ssi** $B \neq$ Arbre-Vide
Droite(B) **est défini ssi** $B \neq$ Arbre-Vide

Axiomes

Racine($\langle o, B_1, B_2 \rangle$) = o
Gauche($\langle o, B_1, B_2 \rangle$) = B_1
Droite($\langle o, B_1, B_2 \rangle$) = B_2

Structure arborescente

Les arbres binaires

Autres opérations :

Taille : Arbre \rightarrow Entier

Axiomes

Taille(Arbre-vidé) = 0

Taille($\langle o, B_1, B_2 \rangle$) = 1 + Taille(B_1) + Taille(B_2)

Feuille : Arbre \rightarrow Booléen

Axiomes

Feuille(A) = vrai ssi $A \neq$ Arbre-vidé
& g(A) = Arbre-vidé
& d(A) = Arbre-vidé

Hauteur (ou niveau) d'un noeud x

$h(x) = 0$ si x est racine de B
 $h(x) = 1 + h(y)$ si y est le père de x

Hauteur (ou profondeur) d'un arbre B

$h(B) = \text{Max} \{ h(x) ; x \text{ noeud de B} \}$

Longueur de cheminement d'un arbre B

$LC(B) = \sum h(x) , x \text{ noeud de B}$

Longueur de cheminement externe

$LCE(B) = \sum h(f) , f \text{ feuille de B}$

Longueur de cheminement interne

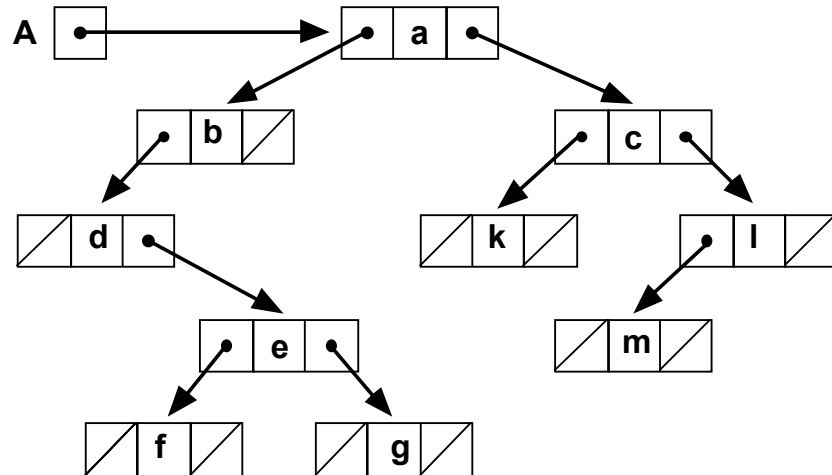
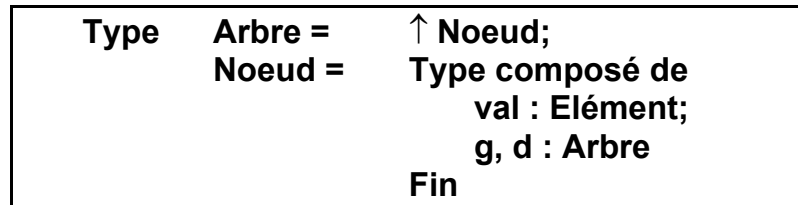
$LCI(B) = \sum h(x) , x \text{ noeud interne de B}$

Remarque : $LC(B) = LCE(B) + LCI(B)$

Structure arborescente

Représentation des arbres binaires

En utilisant les pointeurs :



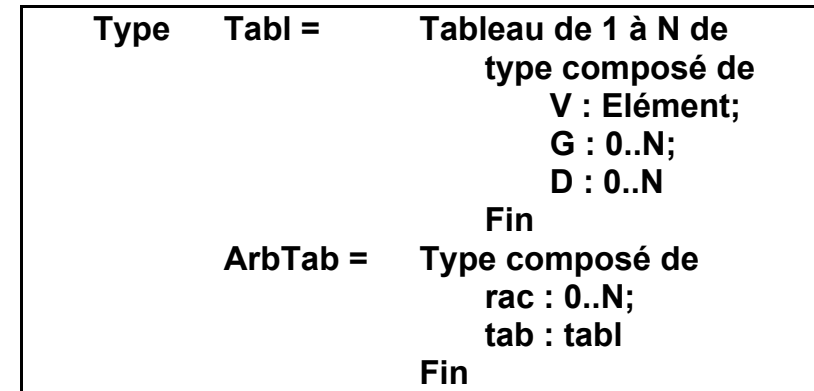
Exercice :

Comment s'écrivent les opérations Arbre-Vide, Feuille, Taille ?

Structure arborescente

Représentation des arbres binaires

En utilisant les tableaux :



rac : 3

tab :

	V	G	D
1			
2	d	0	10
3	a	5	6
4	g	0	0
5	b	2	0
6	c	13	11
7			
8	f	0	0
9	m	0	0
10	e	8	4
11	l	9	0
12			
13	k	0	0

Exercice :

Comment s'écrivent les opérations Arbre-Vide, Feuille, Taille ?

Structure arborescente

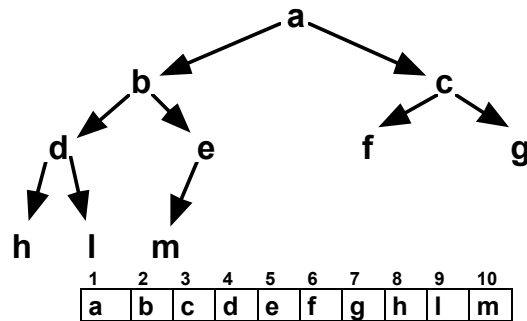
Représentation des arbres binaires

Autre représentation avec les tableaux reposant sur l'ordre hiérarchique (numérotation des noeuds) :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
a	b	c	d		k	l		e					m					f	g		

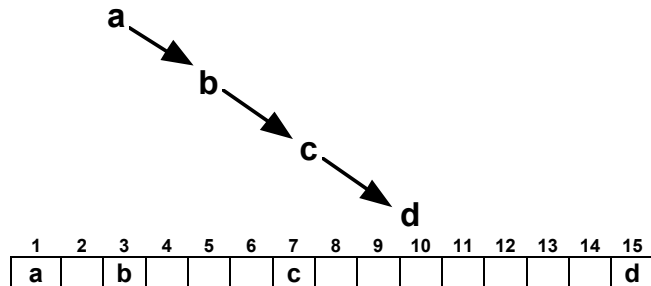
Idéal pour les arbres parfaits

---> très compact



Mauvais pour les arbres dégénérés

---> beaucoup de perte de place

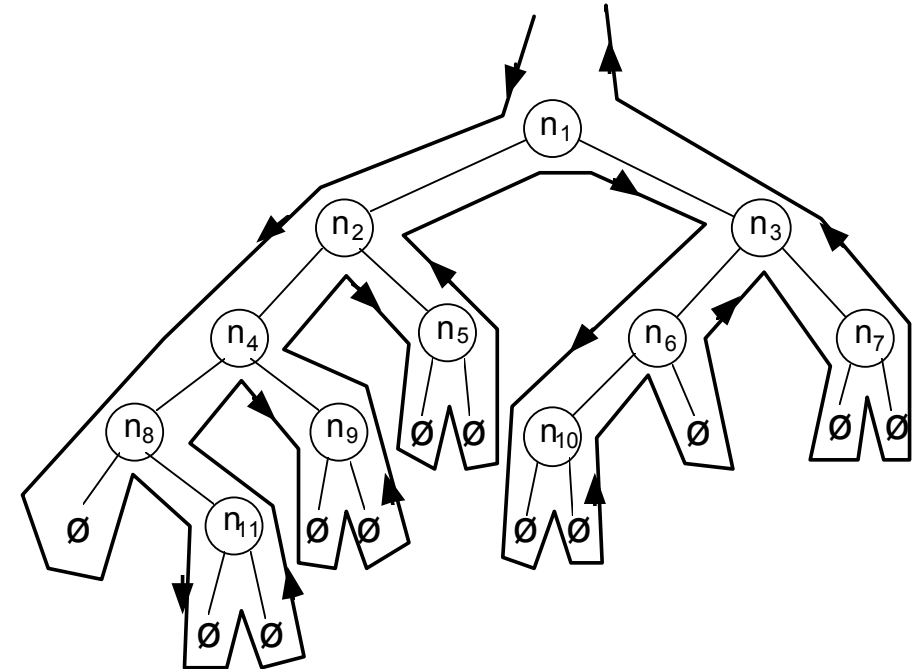


Structure arborescente

Parcours des arbres binaires

Parcours en profondeur à main gauche

---> traitement de l'arbre gauche en premier (le plus courant)



---> on passe trois fois à un noeud : Trois types de parcours suivant le moment où on traite le noeud :

Ordre préfixe ou Pré-ordre : Noeud, Gauche, Droite

Ordre infixé ou En-ordre : Gauche, Noeud, Droite

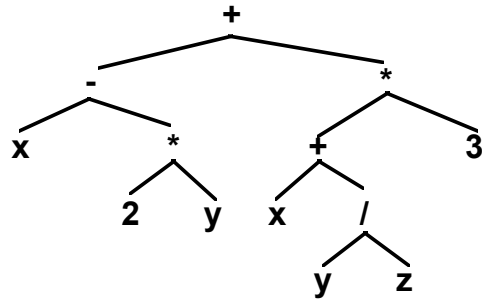
Ordre suffixe ou Post-ordre : Gauche, Droite, Noeud

Exercice : Ecrire les différents ordres de traitement des noeuds de l'arbre ci-dessus.

Structure arborescente

Parcours des arbres binaires

Exemple : Expression arithmétique



Lecture de l'arbre en

Ordre préfixe

$+ - x * 2 y * + x / y z 3$

Ordre infixe

$x - 2 * y + x + y / z * 3$

---> sans parenthèses, il y a des ambiguïtés

$(x - (2 * y)) + ((x + (y / z)) * 3)$

Ordre suffixe

$x 2 y * - x y z / + 3 * +$

Structure arborescente

Parcours des arbres binaires

Algorithme général de parcours

Version récursive

Procédure Parcours(A : Arbre)

A	paramètre donnée	Arbre à parcourir
Début		
Si	A = Arbre-vide	
Alors	Terminaison	
Sinon	Traitement1	
	Parcours(g(A))	
	Traitement2	
	Parcours(d(A))	
	Traitement3	
Finsi		
Fin		

Exercices :

A quoi correspondent Traitement1, Traitement2 et Traitement3 dans les 3 cas de parcours ?

Ecrire la procédure d'affichage d'une expression arithmétique dans les 3 cas.

Exercices sur les arbres

1/ Ecrire les procédures et fonctions suivantes pour la représentation chaînée des arbres binaires

- a) Procédure ArbreVide(A)
- b) Fonction Feuille(A) : Booléen
- c) une fonction qui calcule la taille (=nombre de nœuds) d'un arbre binaire :
 Fonction Taille(A) : Entier
 une fonction qui compte le nombre de feuilles (ou noeuds internes) d'un arbre

binaire :

- Fonction NbFeuilles(A) : Entier
 une fonction qui compte le nombre de noeuds internes d'un arbre binaire (sans utiliser les fonctions précédentes) :
 Fonction NbInternes(A) : Entier
- d) une fonction qui calcule la hauteur d'un arbre binaire :
 Fonction Hauteur(A) : Entier
- e) une fonction qui calcule la longueur de cheminement d'un arbre binaire :
 Fonction LC(A) : Entier
 Cette fonction utilisera une des deux procédures suivantes :
 Fonction LC(A, N) : Entier
 Où N est la profondeur du nœud racine de A
 Procédure LCT(A, LC, T)

Cette procédure calculera la longueur de cheminement LC et la taille T de l'arbre A.

2/ Copie d'un arbre

En utilisant la représentation chaînée des arbres, écrire une procédure "CopieArb" qui duplique un arbre passé en paramètre.

Procédure CopieArb(A1, A2)

A1	Arbre	Donnée	Arbre à dupliquer
A2	Arbre	Résultat	= A1

3/ Recherche dans un arbre binaire

Ecrire une fonction App(A, x) : Booléen qui rend Vrai si $x \in A$, Faux sinon.

4/ Quel est le nombre maximal de noeuds à profondeur k d'un arbre binaire ?

5/ Donnez tous les arbres binaires dont la liste des noeuds en ordre infixe coïncide avec l'expression arithmétique : $2 + 3 * 4 + 5$

6/ Parcours d'arbres

Ecrire les trois procédures de parcours d'arbre.
 Procédure ParcoursPref(A)
 Procédure ParcoursInf(A)
 Procédure ParcoursSuff(A)

7/ Ordre hiérarchique

La numérotation en ordre hiérarchique d'un arbre binaire consiste à numéroter en ordre croissant à partir de 1, tous les noeuds possibles d'un arbre binaire, à partir de la racine, niveau par niveau, et de gauche à droite sur chaque niveau.

- a) Si un noeud est numéroté i, quels sont les numéros de son fils gauche et de son fils droit ? de son père ?
- b) Ecrire une procédure de parcours d'un arbre en ordre hiérarchique (utilisation d'une file).

8/ Arbres binaires localement complets (Contrôle continu n°2 - 1994)

On suppose que l'on utilise la représentation chaînée des arbres binaires.

Ecrire une fonction booléenne qui indique si un arbre est localement complet :
 Fonction LocComplet(A : Arbre) : Booléen
 Vrai si A est localement complet,
 Faux sinon

Rappel : Un arbre binaire localement complet est un arbre dont tous les noeuds non terminaux sont de degré 2.

9/ Moyenne des valeurs d'un arbre binaire (Contrôle continu n°2 - 1994)

On suppose que l'on utilise la représentation chaînée des arbres binaires.

Ecrire une fonction moyenne qui rende la moyenne des éléments d'un arbre binaire contenant des entiers :
 Fonction Moyenne(A : Arbre) : Réel

10/ Arbres binaires dégénérés

On suppose que l'on utilise la représentation chaînée des arbres binaires.

Ecrire une fonction booléenne qui indique si un arbre est dégénéré.
 Fonction Dégénéré(A : Arbre) : Booléen

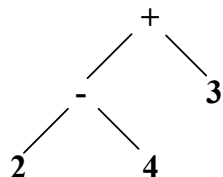
Rappel : Un arbre dégénéré est un arbre dont chaque noeud a au plus un fils.

11/ Evaluation d'expressions arithmétiques

On considère les expressions arithmétiques ne comportant que des opérateurs d'arité 2 (à deux opérands) soit +, -, / (division entière), *, et dont les opérands sont des valeurs entières positives. Un entier lui même est considéré comme une expression arithmétique.

Une représentation adaptée à l'évaluation de l'expression est une représentation en arbre binaire dont les noeuds sont les opérateurs et les feuilles les entiers.

Exemple :



L'évaluation de cette expression est la valeur 1.

On utilise un arbre binaire pour représenter une expression arithmétique en utilisant le type suivant :

```

Type      noeud =   type composé de
                    opérateur : caractère;
                    opérande  : entier;
                    G, D : expr
                    fin
                    expr =   pointeur sur noeud
  
```

Pour un noeud, un seul champ (opérateur ou opérande) sera utilisé.

Ecrire une fonction "EvalExpr" qui prend comme paramètre une expression sous forme d'arbre et rend comme résultat sa valeur (entière, positive ou négative).

```

Fonction EvalExpr(A) : Entier
A      Expr      Donnée      Expression à évaluer
  
```

Remarque : On supposera les expressions sans erreurs, donc que l'on a des arbres localement complets

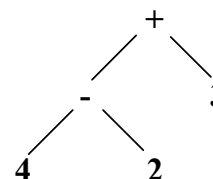
12/ Expressions arithmétiques Calcul formel : Dérivation de fonctions

On considère les expressions arithmétiques ne comportant que des opérateurs d'arité 2 (à deux opérands) soit +, -, /, *, et dont les opérands sont soit des valeurs entières positives entre 0 et 9, soit une variable X. Un entier lui même est considéré comme une expression arithmétique.

Une telle expression peut être représentée de diverses façons : préfixée, postfixée, infixée complètement parenthésée et autres ...

Par exemple : + - 4 2 3 4 2 - 3 + ((4 - 2) + 3), 4 - 2 + 3

sont diverses représentations de la même expression dont la valeur est 5. Une représentation adaptée à l'évaluation de l'expression est une représentation en arbre binaire dont les noeuds sont les opérateurs et les feuilles les entiers. L'expression ci-dessus devient :



On choisit donc de représenter une fonction arithmétique à une variable par un arbre en utilisant les types suivants :

```

Type fonct =   pointeur sur noeud
noeud      =   type composé de
                    val : caractère;
                    D, G : fonct
                    fin
  
```

La valeur d'un noeud pourra donc être soit un opérateur (+, -, *, /), soit un chiffre de 0 à 9, soit une variable "X".

a) Ecrire une procédure qui, à partir d'une expression donnée sous forme d'une chaîne de caractères, construise l'arbre de cette expression.

On donnera les trois versions possibles de cette procédure correspondantes aux trois représentations préfixée, infixée complètement parenthésée et postfixée.

b) Ecrire une procédure qui rend sous forme d'une chaîne de caractères l'expression. On donnera les trois versions possibles (forme préfixe, infixée complètement parenthésée et postfixe).

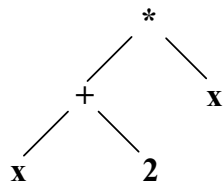
c) Ecrire une procédure "Copie" qui crée un arbre identique à celui passé en paramètre.

Procédure Copie(A, C)			
A	Arbre	Donnée	Arbre à dupliquer
C	Arbre	Résultat	Copie de A

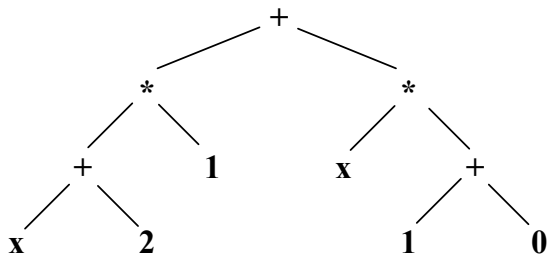
d) Ecrire une procédure qui, à partir d'une fonction, génère sa fonction dérivée sous forme d'un nouvel arbre.

Exemple :

Si la fonction f est



la fonction dérivée f' générée sera



Rappels :

$$(i)' = 0 \text{ si } i \in \{0..9\}$$

$$(x)' = 1$$

$$(u + v)' = u' + v'$$

$$(u - v)' = u' - v'$$

$$(u * v)' = u' * v + u * v'$$

$$(u / v)' = \frac{u' * v - u * v'}{v * v}$$

On ne s'occupera pas des simplifications éventuelles du type :

$$1 * u = u, \quad 0 * u = 0, \quad 0 + u = u, \quad \dots$$

13/ Expressions arithmétiques (Examen 1ère session 1995)

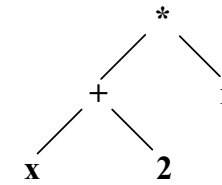
Calcul formel : Simplification d'expressions arithmétiques

Même introduction que l'exercice précédent

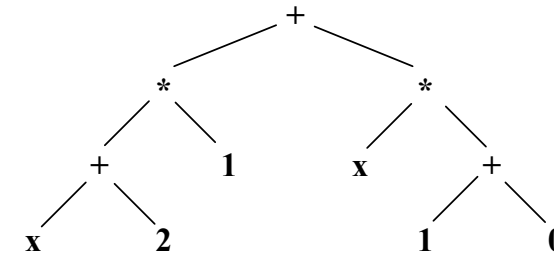
On peut, à partir d'une fonction, générer sa fonction dérivée sous forme d'un nouvel arbre.

Exemple :

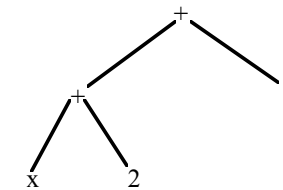
Si la fonction f est



la fonction dérivée f' générée sera



Nous remarquons alors que de nombreuses simplifications peuvent être effectuées, la fonction devient alors :



$$\text{car } (x + 2) * 1 = x + 2, \quad 1 + 0 = 1, \quad x * 1 = x$$

Ecrire une procédure qui simplifie une fonction passée en paramètre en modifiant l'arbre qui la représente.

Simplifications à mettre en oeuvre :

$u + 0 = 0 + u = u$
$u - 0 = u$
$0 - i = -i$ si i feuille et $0 \leq i \leq 9$ feuille
$u * 0 = 0 * u = 0$
$u * 1 = 1 * u = u$
$u / 1 = u$
$u / 0 = \text{erreur}$
$0 / u = 0$

On s'aidera d'une **procédure ToutLibérer(A)** qui, comme son nom l'indique, libère l'ensemble des nœuds d'un arbre.

14/ Expressions arithmétiques : Evaluation de fonctions(Examen 2ème session 1995)

Même introduction que l'exercice précédent

a) Ecrire une fonction Valeur(C) : Entier qui pour un caractère compris entre '0' et '9' rend la valeur entière correspondante.

On pourra utiliser la fonction ASCII(c : caractère) : Entier qui rend le code ASCII d'un caractère quelconque.

b) Ecrire une fonction qui effectue une évaluation de la fonction pour une valeur de x donnée :

Fonction Eval(F, x) : réel

On pourra utiliser la fonction du a).

15/ Nombre d'additions (Contrôle continu n°2 - 1995)

On suppose qu'une expression arithmétique est stockée sous forme d'un arbre binaire.

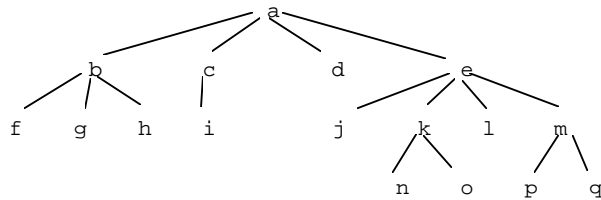
Ecrire une fonction qui rende le nombre d'additions présentes dans cette expression arithmétique.

Fonction NbAdd(A):Entier

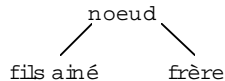
16/ Arbres généraux (Extrait examen 1994 1ère session)

On désire utiliser et représenter des arbres de degré quelconque.

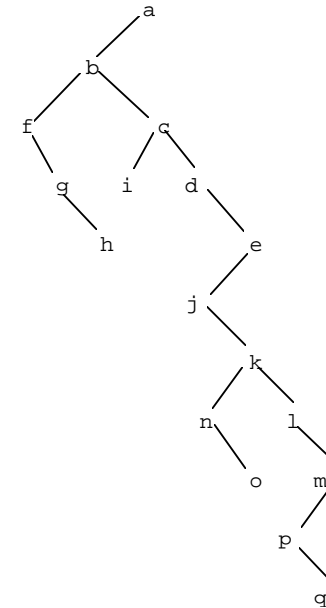
Exemple d'arbre général :



Pour cela, il existe différentes représentations possibles. Une de ces représentations est d'utiliser un arbre binaire **fils aîné-frère droit** dans lequel le fils gauche sera le fils aîné du noeud courant et le fils droit un frère du noeud courant.



Exemple : représentation en arbre binaire de l'arbre précédent :



Le type utilisé pour représenter un arbre général est donc :

Type ArbreG = **pointeur sur** Noeud;
Noeud = **Type composé de**
 val : Elément;
 Fils : ArbreG;
 Frère : ArbreG;
Fin

- a) Ecrire une fonction qui rende le nombre de frères cadets d'un noeud :
 Fonction Nbfrères(A : ArbreG) : Entier
- b) Ecrire une fonction qui rende le degré d'un noeud :
 Fonction DegréNoeud(A : ArbreG) : Entier
 On pourra utiliser la fonction du a)
- c) Ecrire une fonction qui rende le degré d'un arbre général :
 Fonction Degré(A : ArbreG) : Entier
 On pourra utiliser la fonction du b)

Autres questions sur les arbres généraux (2ème session)

a) Avec les arbres généraux, on peut effectuer des parcours à main gauche comme pour les arbres binaires. Ces parcours peuvent être en ordre préfixe ou suffixe (on ne peut pas faire d'ordre infixe).

Ecrire les noeuds dans l'ordre où ils sont consultés en ordre préfixe ou en ordre suffixe pour l'arbre général de l'exemple.

Quels ordres de parcours faut-il utiliser dans l'arbre binaire correspondant pour arriver aux mêmes résultats ?

b) Ecrire deux procédures :

Procédure ParcoursPref(A) et **Procédure ParcoursSuff(A)**

où A, de type ArbreG, paramètre Donnée, est l'arbre à parcourir

17/ Arbres binaires - Files (Examen 1ère session 1995)

Procédure ADEVINER(A : Arbre)

A Arbre paramètre donné

F File d'Arbre local

p Arbre

Début

File_Vide(F);

Ajouter(F, A)

Tantque Non Est_Vide(F) faire

Début

Premier(F, p)

Retirer(F)

Afficher p↑.val

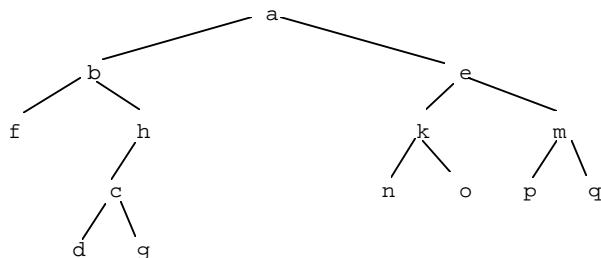
Si p↑.G ≠ Nil alors Ajouter(F, p↑.G) Finsi

Si p↑.D ≠ Nil alors Ajouter(F, p↑.D) Finsi

Fin

Fin

a) En utilisant la procédure ADEVINER, donnez l'ordre d'affichage des noeuds de l'arbre suivant :



b) Que fait la procédure ADEVINER ? Dans quel ordre affiche-t-elle les noeuds d'un arbre binaire ?

18/ Arbres binaires - Piles

(Examen 2ème session 1995)

Procédure ADEVINER(A : Arbre)

A Arbre paramètre donné

P Pile d'Arbre local

q Arbre local

Début

Pile_Vide(P);

Empiler(P, A)

Tantque Non Est_Vide(P) faire

Début

Sommet(P, q)

Dépiler(P)

Afficher q↑.val

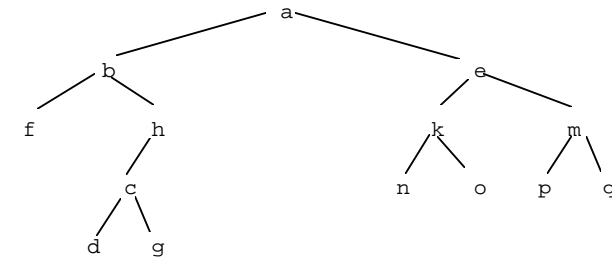
Si q↑.G ≠ Nil alors Empiler(P, q↑.G) Finsi

Si q↑.D ≠ Nil alors Empiler(P, q↑.D) Finsi

Fin

Fin

a) En utilisant la procédure ADEVINER, donnez l'ordre d'affichage des noeuds de l'arbre suivant :



b) Que fait la procédure ADEVINER ? Dans quel ordre affiche-t-elle les noeuds d'un arbre binaire ?

Arbres binaires de recherche

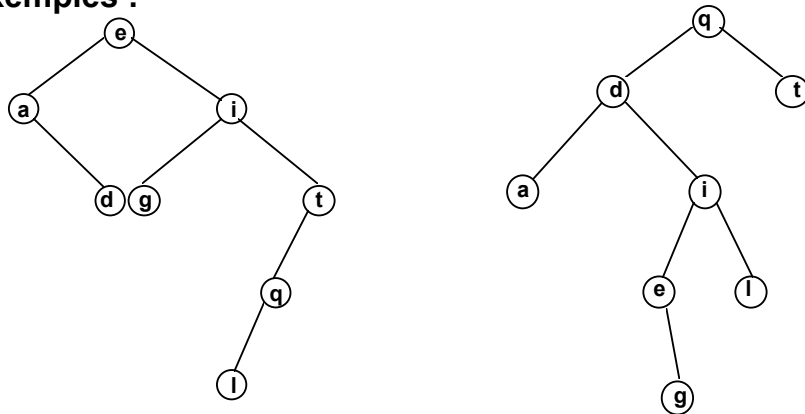
Un arbre binaire de recherche est un arbre binaire tel que pour tout noeud :

- les éléments de tous les noeuds du sous-arbre gauche sont inférieurs ou égaux à l'élément du noeud,
- les éléments de tous les noeuds du sous-arbre droit sont supérieurs à l'élément du noeud.

Remarque :

Quel parcours (préfixe, infixe ou suffixe) nous donne les éléments triés en ordre croissant ?

Exemples :



Remarque :

Pour une suite d'éléments, il existe plusieurs arbres binaire de recherche.

Arbres binaires de recherche

Recherche d'un élément

Recherche : Élément x Arbre \rightarrow Booléen

Axiomes

Rechercher(x, Arbre-vide) = Faux

$x = r \Rightarrow$ Rechercher(x, <r, G, D>) = Vrai

$x < r \Rightarrow$ Rechercher(x, <r, G, D>) = Rechercher(x, G)

$x > r \Rightarrow$ Rechercher(x, <r, G, D>) = Rechercher(x, D)

Exercice :

Ecrire les versions récursive et itérative de cette fonction.

Ajout d'un élément

2 étapes :

- recherche de la place de l'élément à insérer,
- insertion de l'élément.

Ajout aux feuilles

Ajouter-feuille : Élément x Arbre \rightarrow Arbre

Axiomes

Ajouter-feuille(x, Arbre-vide) = < x, Arbre-vide, Arbre-vide >

$x \leq r \Rightarrow$

Ajouter-feuille(x, <r, G, D>) = < r, Ajouter-feuille(x, G), D >

$x > r \Rightarrow$

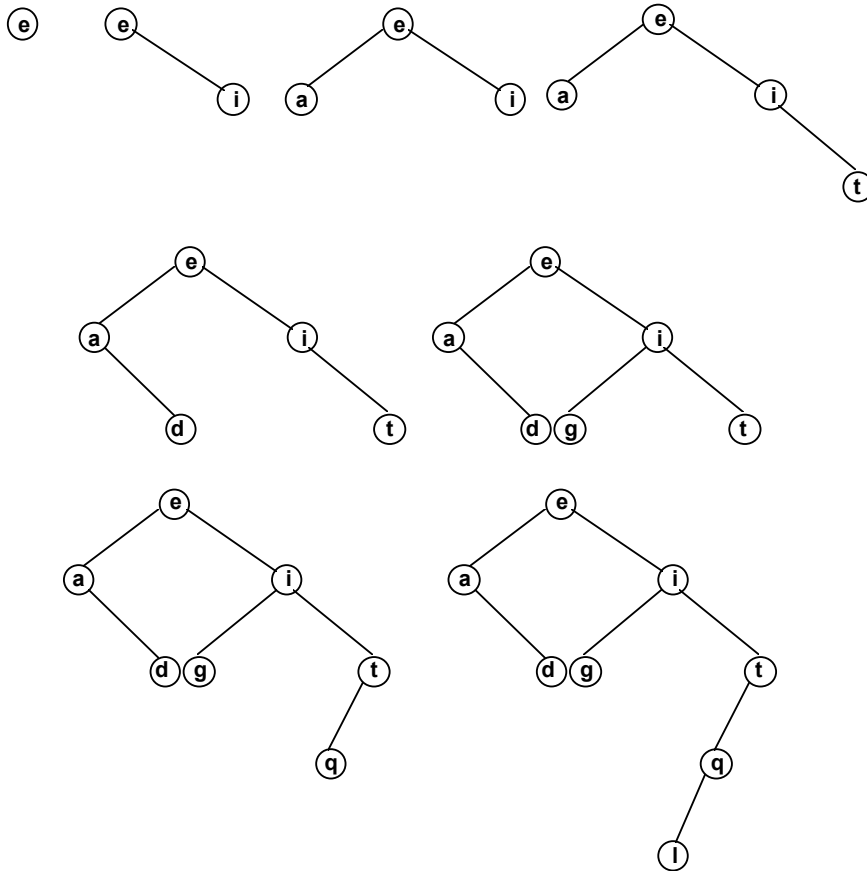
Ajouter-feuille(x, <r, G, D>) = < r, G, Ajouter-feuille(x, D) >

Exercice : Ecrire les versions récursive et itérative de cette procédure.

Arbres binaires de recherche

Exemple d'ajout aux feuilles :

Ajout successif de e, i, a, t, d, g, q, l



Exercice :

Ecrire les différentes étapes de construction de l'arbre pour les suites :

- e, a, d, i, g, t, q, l
- q, d, t, i, a, l, e, g

Arbres binaires de recherche

On peut ajouter des éléments, non seulement aux feuilles, mais à n'importe quel niveau, en particulier à la racine, ce qui peut être utile quand c'est après l'ajout d'un élément que l'on fait le plus de recherches le concernant.

Ajout à la racine

Ceci va nécessiter une réorganisation de l'arbre : il faut couper l'arbre initial en deux sous-arbres, l'un contenant les éléments inférieurs, l'autre les éléments supérieurs. C'est l'étape de coupure qui est la plus délicate.

Ajouter-rac : Élément x Arbre \rightarrow Arbre

Axiomes

Ajouter-rac(x, Arbre-vide) = $\langle x, \text{Arbre-vide}, \text{Arbre-vide} \rangle$

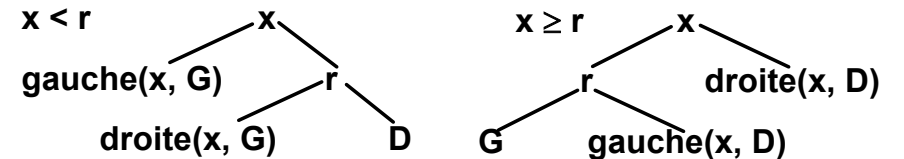
Racine(Ajouter-rac(x, $\langle r, G, D \rangle$)) = x

$x < r \Rightarrow$ Gauche(Ajouter-rac(x, $\langle r, G, D \rangle$))
= Gauche(Ajouter-rac(x, G))

Droite(Ajouter-rac(x, $\langle r, G, D \rangle$))
= $\langle r, \text{Droite(Ajouter-rac(x, G))}, D \rangle$

$x \geq r \Rightarrow$ Gauche(Ajouter-rac(x, $\langle r, G, D \rangle$))
= $\langle r, G, \text{Gauche(Ajouter-rac(x, D))} \rangle$

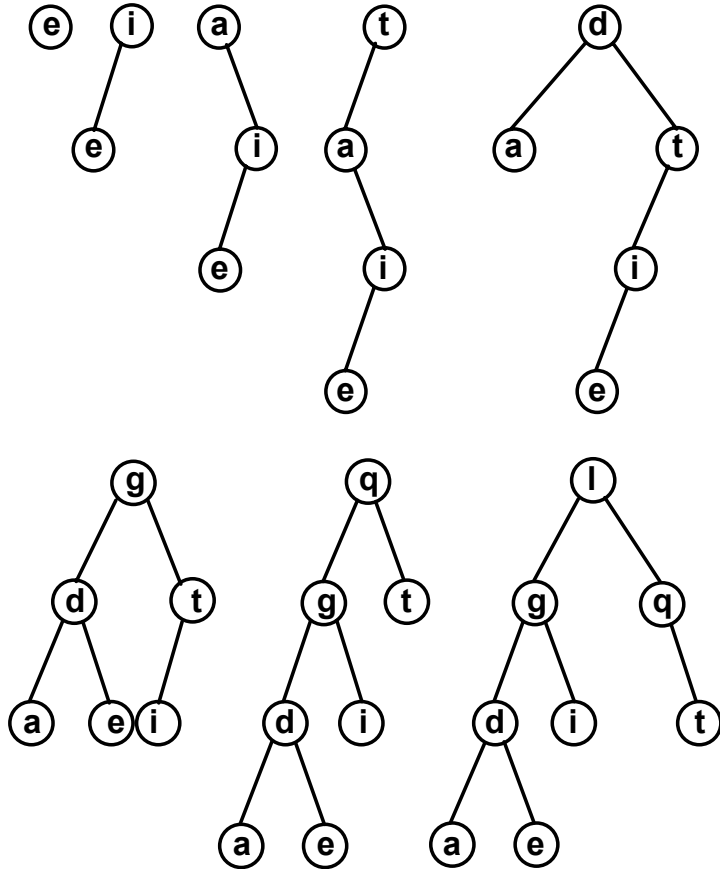
Droite(Ajouter-rac(x, $\langle r, G, D \rangle$))
= Droite(Ajouter-rac(x, D))



Arbres binaires de recherche

Exemple d'ajout à la racine :

Ajout successif de e, i, a, t, d, g, q, l



Exercice :

Ecrire les différentes étapes de construction de l'arbre pour les suites :

- e, a, d, i, g, t, q, l
- q, d, t, i, a, l, e, g

Arbres binaires de recherche

Ajout à la racine

Procédure AjouterRac(x, A)			
x	Elément Donnée	Elément à insérer	
A	Arbre	Donnée/Résultat	Arbre où ajouter x
R	Arbre	Local	Nouvelle racine
Début			
Nouveau(R)			
$R \uparrow .val \leftarrow x$			
Couper(x, A, $R \uparrow .g$, $R \uparrow .d$)			
$A \leftarrow R$			
Fin			

Procédure récursive de coupure

Procédure Couper(x, A, G, D);			
x	Elément Donnée	Valeur selon laquelle couper l'arbre	
A	Arbre	Donnée/Résultat	Arbre à couper
G	Arbre	Résultat	Eléments de A inférieurs ou égaux à x
D	Arbre	Résultat	Eléments de A sup. à x
Début			
Si $A = \text{nil}$			
Alors $G \leftarrow \text{nil} ; D \leftarrow \text{nil}$			
Sinon Si $x < A \uparrow .val$			
Alors $D \leftarrow A$			
Couper(x, $A \uparrow .g$, G, $D \uparrow .g$)			
Sinon $G \leftarrow A$			
Couper(x, $A \uparrow .d$, $G \uparrow .d$, D)			
Finsi			
Finsi			
Fin			

(Ecrire la version itérative de cette procédure)

Suppression d'un élément

- chercher l'élément,
- supprimer l'élément,
- réorganiser l'arbre.

Spécification de la suppression

Max rend l'élément maximal d'un arbre

Dmax rend l'arbre privé de cet élément

Supprimer : Élément x Arbre \rightarrow Arbre

Max : Arbre \rightarrow Élément

Dmax : Arbre \rightarrow Arbre

Pré-conditions

Max(G) est défini ssi $G \neq$ Arbre-vide

Dmax(G) est défini ssi $G \neq$ Arbre-vide

Axiomes

Supprimer(x, Arbre-vide) = Arbre-vide

$x = r \Rightarrow$ Supprimer(x, $\langle r, G, \text{Arbre-vide} \rangle$) = G

$x = r \ \& \ D \neq$ Arbre-vide

\Rightarrow Supprimer(x, $\langle r, \text{Arbre-vide}, D \rangle$) = D

$x = r \ \& \ G \neq$ Arbre-vide $\ \& \ D \neq$ Arbre-vide

\Rightarrow Supprimer(x, $\langle r, G, D \rangle$) = $\langle \text{Max}(G), \text{Dmax}(G), D \rangle$

$x < r \Rightarrow$ Supprimer(x, $\langle r, G, D \rangle$) = $\langle r, \text{Supprimer}(x, G), D \rangle$

$x > r \Rightarrow$ Supprimer(x, $\langle r, G, D \rangle$) = $\langle r, G, \text{Supprimer}(x, D) \rangle$

Max($\langle r, G, \text{Arbre-vide} \rangle$) = r

Dmax($\langle r, G, \text{Arbre-vide} \rangle$) = G

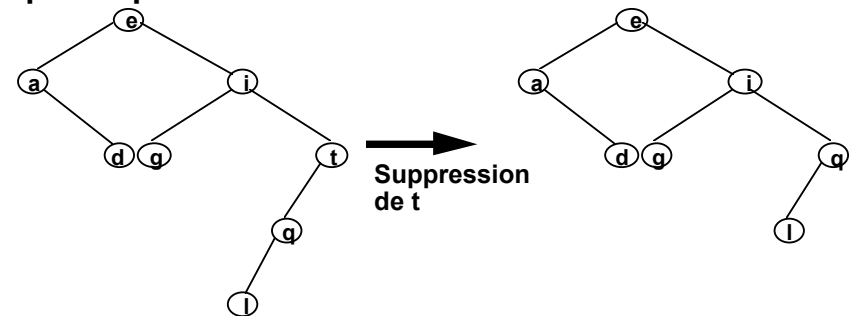
$D \neq$ Arbre-vide \Rightarrow Max($\langle r, G, D \rangle$) = Max(D)

$D \neq$ Arbre-vide \Rightarrow Dmax($\langle r, G, D \rangle$) = $\langle r, G, \text{Dmax}(D) \rangle$

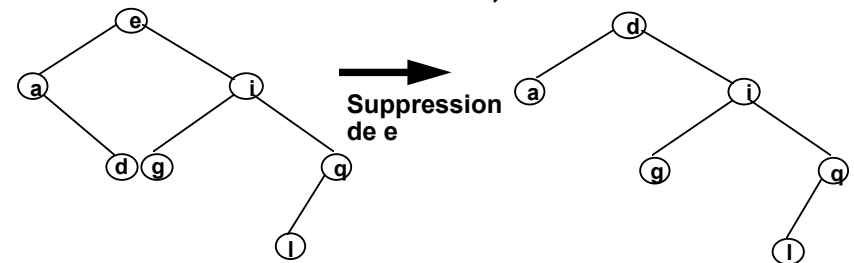
Suppression d'un élément

Réorganiser l'arbre :

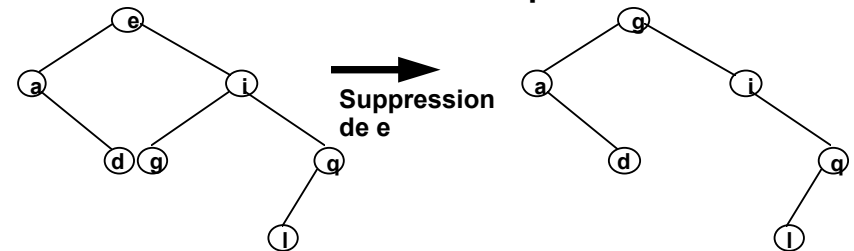
- pour un noeud sans fils, suppression immédiate.
- pour un noeud avec un seul fils, il suffit de le remplacer par ce fils.



- pour un noeud qui a deux fils : remplacer le noeud contenant l'élément à supprimer par celui qui lui est immédiatement soit inférieur,



soit supérieur.



Remarque : Solutions équivalentes si il n'y a pas d'éléments égaux et sinon que se passe-t-il ?

Arbres binaires de recherche
Suppression d'un élément

Procédure Supprimer(x, A)			
x	Elément	Donnée	valeur à supprimer
A	Arbre	Donnée/Résultat	Arbre où supprimer l'élément
maxArbre	Local		Maximum de l'arbre
p	Arbre	Local	Variable auxiliaire
Début			
Si A ≠ nil			
Alors Si x < A↑.val			
Alors Supprimer(x, A↑.g)			
Sinon Si x > A↑.val			
Alors Supprimer(x, A↑.d)			
Sinon { x = A↑.val }			
Si A↑.g = nil			
Alors p ← A			
A ← A↑.d			
Libérer(p)			
Sinon Si A↑.d = nil			
Alors p ← A			
A ← A↑.g			
Libérer(p)			
Sinon {A↑.d ≠ nil et A↑.g ≠ nil}			
Supmax(max, A↑.g)			
A↑.val ← max			
Finsi			
Finsi			
Finsi			
Finsi			
Fin			

Arbres binaires de recherche
Suppression d'un élément

Procédure de suppression du maximum
(Rôle de Max et Dmax)

Procédure Supmax(max, A)			
maxElément	Résultat		Maximum de A
A	Arbre	Donnée/Résultat	Arbre où supprimer le maximum
p	Arbre	Local	Variable auxiliaire
Début			
Si A↑.d = nil			
Alors Max ← A↑.val			
p ← A			
A ← A↑.g			
Libérer(p)			
Sinon Supmax(max, A↑.d)			
Finsi			
Fin			

Exercices sur les arbres binaires de recherche

1/ Ordre croissant et décroissant

Quel est l'ordre de parcours d'un arbre binaire de recherche qui affiche la liste des éléments en ordre croissant ? en ordre décroissant ?

Ecrire les procédures correspondantes.

Procédure Croissant(A : Arbre) qui affiche les valeurs des noeuds

Procédure Décroissant(A : Arbre) qui affiche les valeurs des noeuds

2/ Ecrire les procédures et fonctions vues en cours :

Fonction Recherche(A, x):Booléen Version itérative et récursive

Procédure AjoutFeuille(A, x) Version itérative et récursive

Procédure AjoutRac(A, x) + procédure récursive de coupure

Procédure Supprimer(A, x) + procédure Supmax(A, max)

Procédure Supprimer(A, x) pour les arbres binaires sans duplicats
+ procédure Supmin(A, max)

3/ Recherche du minimum (Extrait examen 1994 2ème session)

Comme nous l'avons vu en cours, un arbre binaire peut être représenté, au moins, de trois manières :

Chaînée

Type Arbre = pointeur sur Noeud;

Noeud = type composé de
 val : Elément;
 G, D : Arbre

 {pointeurs sur les noeuds Gauche et Droit}
 fin

Contigüe chaînée dans un tableau

Type Arbre = type composé de
 rac : entier;
 T : tableau de 1 à MAX de Noeud

 fin
Noeud = type composé de
 val : Elément;
 G, D : entier
 {indices dans T du noeud Gauche et du noeud Droit}
 fin

Contigüe dans un tableau en ordre hiérarchique

Type Arbre = Tableau de 1 à Max de Elément
 {un élément est rangé dans la case du tableau correspondant à son
 ordre hiérarchique }
 { Rappels : les fils d'un noeud d'indice i sont à l'indice 2i et 2i+1 }

Ecrire une fonction qui recherche le minimum d'un arbre binaire de recherche :
Fonction Minimum(A : Arbre) : Elément

On écrira les trois versions de cette fonction.

4/ Affichage des nombres pairs (Contrôle continu n°2 - 1995)

Ecrire une procédure qui affiche en ordre croissant les nombres pairs dans un arbre binaire de recherche A.

Procédure AffPair(A)

5/ Echange des fils d'un arbre binaire (Contrôle continu n°2 - 1995)

Ecrire une procédure qui échange les sous-arbres gauche et droit de tous les noeuds d'un arbre binaire.

On en écrira deux versions : une qui modifie l'arbre initial (sans duplication), une qui conserve l'arbre initial.

Procédure Echange1(A)

Procédure Echange2(A1,A2)

Quel est l'effet de ces procédures sur un arbre binaire de recherche ?

6/ Affichage des nombres inférieurs ou égaux à x (Contrôle continu n°2 - 1995)

Ecrire une procédure qui affiche, en ordre croissant, les nombres inférieurs ou égal à un nombre x. Les nombres sont stockés dans un arbre binaire de recherche A. Il faudra tenir compte des performances et ne tester que le minimum de noeuds. Une autre procédure pourra être écrite.

Procédure AffInfEg(A, x)

7/ Fusion de deux arbres binaires de recherche

L'arbre binaire de recherche U est la fusion des arbres binaires de recherche A et B, si U contient tous les éléments de A et tous les éléments de B. En utilisant la procédure de coupure, écrire une procédure de fusion de deux arbres binaires de recherche.

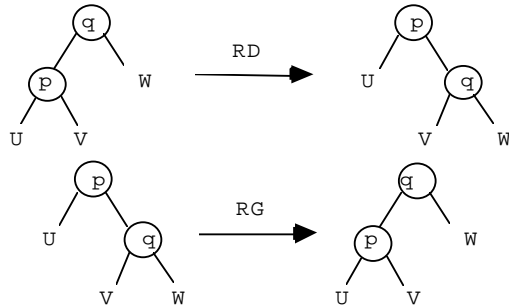
8/ Rotations

On définit sur les arbres binaires de recherche des transformations qui sont appelées des rotations. Ces opérations sont utilisées dans les algorithmes de rééquilibrage d'arbres. Les arbres obtenus sont des arbres binaires de recherche si les arbres sont sans duplicats.

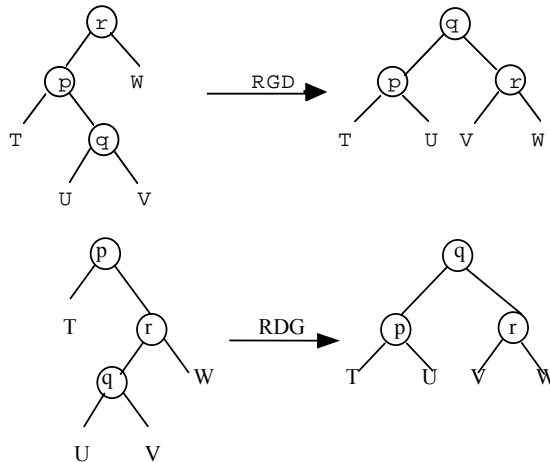
Elles consistent à faire basculer vers la droite (resp. gauche) un sous-arbre qui est trop déséquilibré vers la gauche (resp. droite).

Il y en a quatre types :

les rotations simples, à droite (RD) et à gauche (RG),



les rotations doubles, gauche-droite (RGD) et droite-gauche (RDG)



En utilisant la représentation des arbres binaires utilisant les pointeurs (vue en cours), écrire les algorithmes de ces quatre procédures.

Procédure RD(A) et **Procédure RG(A)**

Procédure RGD(A) et **Procédure RDG(A)**

où A est un paramètre Donnée/Résultat de type Arbre

Remarque : Les rotations doubles pourront être écrites en utilisant les procédures de rotation simple.