Chap. 4: Structures de données*

Un exercice pour commencer:

“Ecrire un algorithme qui vérifie qu’un texte contenant des caractères standards est syntaxiquement correct du point de vue des parenthèses. Les parenthèses sont de trois types (, [et { et leurs parenthèses fermantes correspondantes sont respectivement),] et }. La correction syntaxique implique qu’à chaque parenthèse ouvrante corresponde, plus loin dans le texte, une parenthèse fermante du même type. Le texte compris entre ces deux parenthèses doit également être correct au point de vue des parenthèses: une parenthèse ouverte doit y être refermée.”

*avec l’aimable collaboration de Gilles Falquet

Solutions + commentaires

Solution avec une structure de données “pile”

Supposons que nous disposons d’une structure de données “pile de caractères” avec les opérations habituelles “initialiser pile”, “empiler”, “dépiler”, “sommet” et “est vide”: --> l’algorithme devient très simple

Données:
p:pile, c:tableau de N caractères qui contient le texte

Algorithme:

```
p.InitPile
pour i de 0 à N-1
    si (c[i] est une prarenthèse ouvrante) p.empiler(c[i])
    sinon si (c[i] est une paranthèse fermante) {
        si (p.estVide() ) stop “erreur: il manque une
                                parenthèse ouvrante”
        si (p.sommet() est du même type que c[i])
            p.dépiler
        sinon stop “erreur: la parenthèse fermante
                                n’est pas du bon type”
    }
si (p.estVide) affiche“syntaxe parenthèses OK”
sinon affiche “il manque une parenthèse fermante
```

Remarques: cet algorithme est écrit en pseudo-langage orienté objet inspiré d’Obéron (et de Java pour les {}). Les parties de l’algorithme qui ne posent pas de problème particulier sont écrites en français.

Structures de données

Motivations:

- représentation appropriée des données
  -> algorithmes moins complexes
  -> algorithmes plus efficaces
- on retrouve les mêmes structures de données dans de nombreux algorithmes

Nous verrons 7 structures de données “célèbres”:

- pile
- queue (file d’attente)
- séquence (suite ordonnée)
- ensemble et multi-ensemble
- fonction (ou tableau ou encore table)
- arbre (arbre général et arbre de recherche)
- graphe

Approche: type abstrait (données + opérations), orientée objet

- **But du chapitre 4**: définition des structures, utilisation, spécification de l’interface en Oberon
- **Chapitre 5** et au séminaire: techniques d’implémentation (éventuellement plusieurs)

Structures de données (remarques)

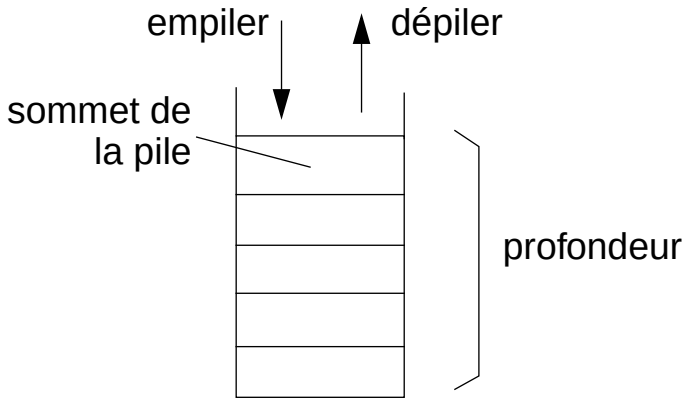
- Par abus de langage, on a coutume d'appeler ces structures de données célèbres (pile, queue, séquence, etc.) "structures de données"
- Propriétés remarquables:
  - les structures de données joue un rôle clé dans la réutilisation de logiciels
  - la séparation de l'interface et de l'implémentation est une application du principe d'abstraction des types abstraits
- Les opérations sur les structures seront regroupées en trois catégories:
  - constructeurs / modifieurs (modifient l'état de la structure)
  - sélecteurs (renseignent sur l'état de la structure)
  - itérateurs (parcourent la structure dans sa globalité)

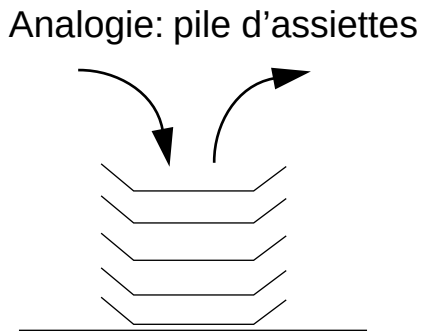
Remarques: comme les mêmes itérateurs sont valables pour plusieurs types de structures de données, nous les étudierons isolément.

Pile

Définition: “Une pile est une collection d’éléments à laquelle l’accès est limité à l’élément ajouté en dernier. On appelle cet élément le sommet de la pile.”

On dit aussi que la pile obéit au protocole LIFO (Last In First Out)Illustration:





Utilisation de la pile en informatique:

- journalisation des mises à jour pour réaliser la fonction “undo” (logiciels de bureautique p.e.)
- analyse des expressions par un compilateur
- allocation de la mémoire pour les variables locales lors de l’exécution des procédures

Spécification de la pile

Constructeurs / modifieurs de pile

opération	paramètre	résultat	description
new		p: Pile	crée une nouvelle pile p
initPile			initialise la pile à vide
empiler (push)	e: élément		dépose l'élément e au sommet de la pile
dépiler (pop)			enlève l'élément qui se trouve au sommet de la pile

Sélecteurs de pile

opération	paramètre	résultat	description
sommet (top)		élément	retourne la valeur de l'élément qui est au sommet de la pile
estVide		booléan	retourne vrai si la pile est vide
profondeur		entier	retourne la profondeur de la pile

Interface de la pile en Oberon:

```
DEFINITION Pile;
	TYPE
		Elément = ...; (* CHAR par exemle *)
		Pile = POINTER TO RECORD
			(p:Pile)PROCEDURE Dépiler, NEW;
			(p:Pile)PROCEDURE Empiler(el: Elément), NEW;
			(p:Pile)PROCEDURE EstVide(): BOOLEAN, NEW;
			(p:Pile)PROCEDURE InitPile, NEW;
			(p:Pile)PROCEDURE Profondeur(): INTEGER, NEW;
			(p:Pile)PROCEDURE Sommet(): Elément, NEW;
		END;
END Pile.
```

Pourquoi n’utilise-t-on que rarement les piles ?

Question: Pourquoi les programmes écrits dans un langage de haut niveau (Oberon, C, Java etc) n’utilisent que rarement une pile explicite ?

Réponse: Car ils utilisent la récursivité, qui elle-même est réalisée à l’aide d’une pile -> les programmes utilisent implicitement une pile.

Plus précisément:

L’appel à une procédure provoque le création au sommet de la pile de l’**environnement d’exécution** de la procédure, càd:

(1) les paramètres et les variables locales de la procédure

(2) l’adresse de retour

La fin de l’exécution de la procédure provoque:

(3) la reprise de l’exécution indiquée par l’adresse de retour, l’instruction qui suit l’appel de la procédure

(4) la suppression du sommet de la pile

->on retrouve donc l’environnement d’exécution appelant

-> rien d’autre n’est nécessaire pour implémenter la récursivité

Exemple

Reprenons l'exemple du vérificateur syntaxique de parenthèse et implémentons-le à l'aide d'une procédure **récursive**:

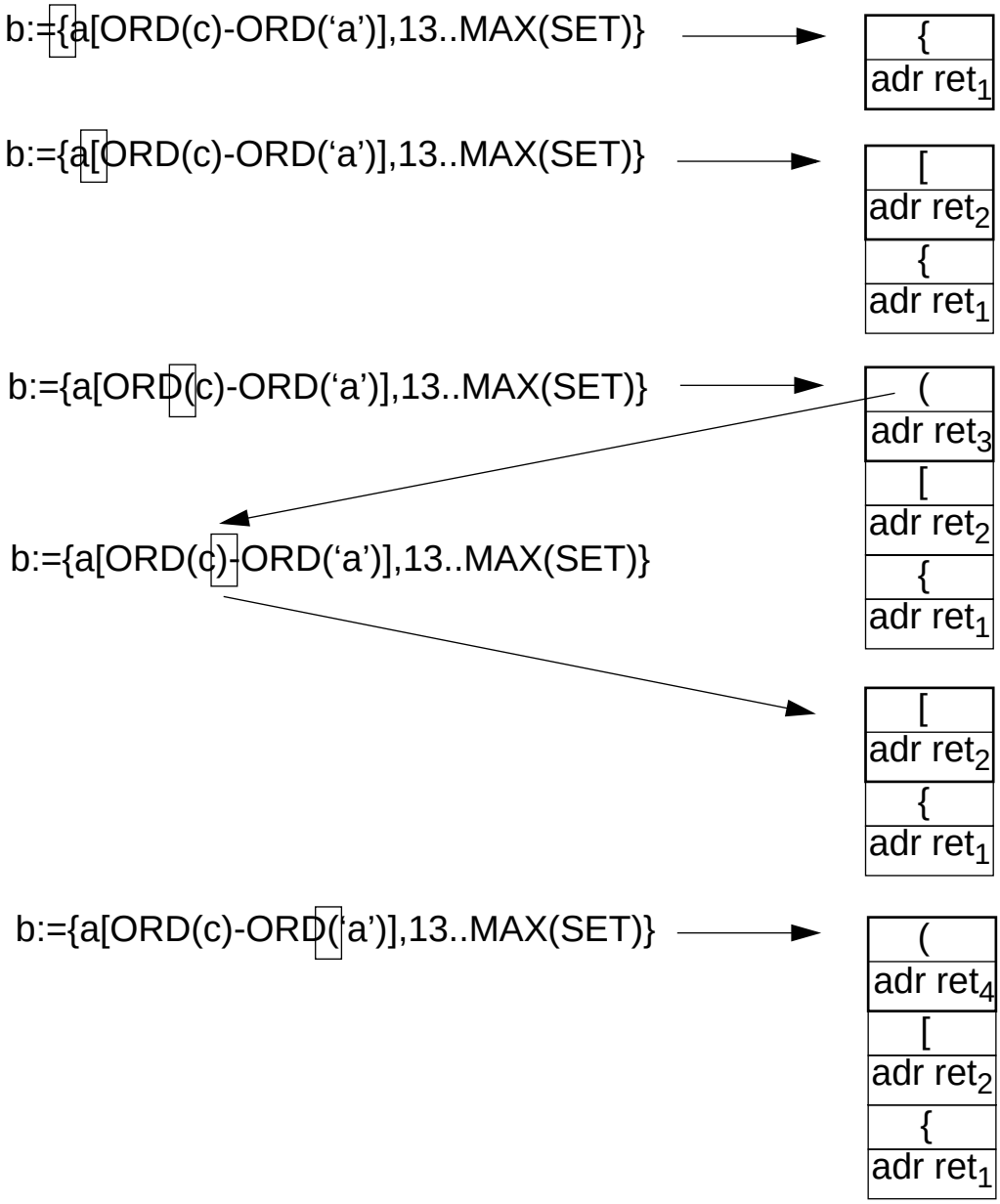
Données c:tableau de N caractères qui contient le texte

```
procédure vérifieParenthèse( p: caractère) {
    répéter
        incrémenter i
    jusqu'à ce que (i > N) ∨ (c[i] est une parenthèse)
    si i > N - 1 stop "erreur: il manque au moins une
                        parenthèse fermante"
    sinon
        si (c[i] est une parenthèse ouvrante)
            vérifieParenthèse(c[i])
        sinon si (c[i] n'est pas du même type que p)
                stop "erreur: la parenthèse fermante
                        n'est pas du bon type"
}
```

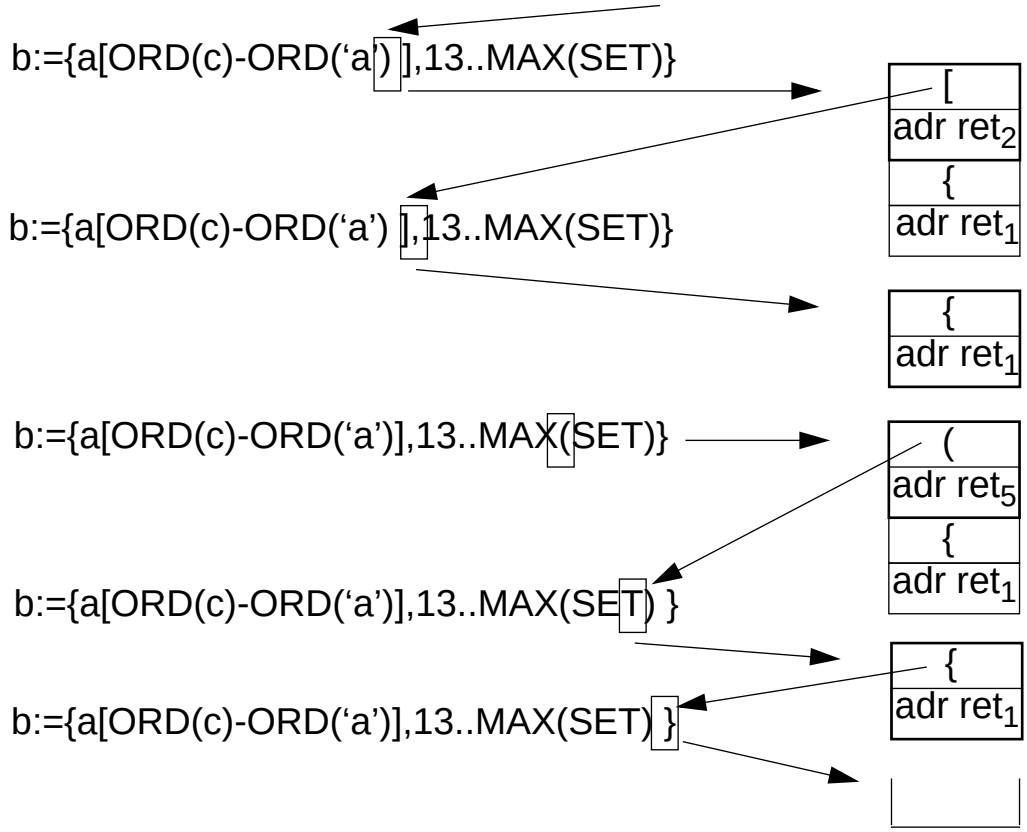
Exécution

Soit le texte “ b:={a[ORD(c)-ORD(‘a’)],13..MAX(SET)} “

Evolution de la pile de l’environnement d’exécution en fonction de l’analyse du texte d’entrée:



Exécution (suite)



Commentaire:

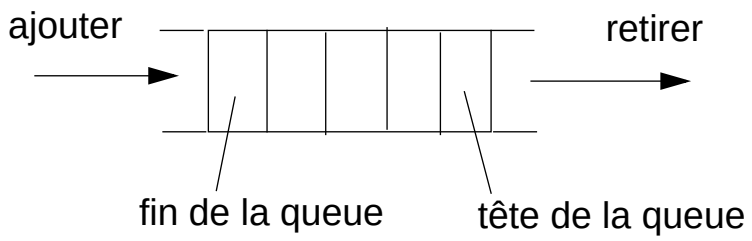
- chaque parenthèse ouvrante donne lieu à un appel récursif, provocant le stockage de la parenthèse et de l'adresse de retour au sommet de la pile
- chaque parenthèse fermante est comparée avec c (qui est stocké au sommet de la pile); si elles sont de même type, c'est la fin de la procédure -> l'environnement d'exécution est retiré du sommet de la pile
- la pile gérée explicitement par l'algorithme de la page 4-2 se serait comportée de manière similaire (mais, bien sûr, elle ne stocke pas d'adresse de retour)

Queue (ou file)

Queue: “Séquence d’éléments dans laquelle les éléments sont ajoutés à une extrémité (appelée fin de la queue) et rétirés de l’autre (appelée tête de la queue)”.

On procède donc selon un protocole de premier arrivé, premier servi. En Anglais, on appelle ce protocole First In First Out (FIFO).

Illustration:



Remarques:

- Un seul élément est accessible à un instant donné: l’élément en tête de la queue.
- Différence avec la pile: dans le cas de la queue, les éléments sont retirés dans l’ordre d’insertion alors que dans cas de la pile, les élément sont retirés dans l’ordre inverse de leur insertion.

Utilisation

Nombreux parallèles avec le monde réel:

- Queue devant un guichet (poste, banque etc.)
- Avions en attente d'atterissage.

Utilisation en informatique:

- Queue d'impression de fichiers
- "Scheduler": allocation du CPU à plusieurs processus concurrents
- Support au protocole "producteur - consommateur" entre deux processus assynchrones: un processus P produit des données envoyées à C. Une file stocke les données lorsque P produit plus vite que C ne consomme.
- Simulation de queues réelles (guichets, trafic aautomobile, etc) lorsqu'il nexiste pas d'approche analytique.

D'une manière générale: on utilise une queue chaque fois qu'il s'agit de gérer l'allocation d'une ressource à plusieurs clients

Spécification de la queue

Constructeurs / modifieurs de queue

opération	paramètre	résultat	description
new		q: Queue	crée une nouvelle queue q
initQueue			initialise la queue à vide
ajouter	e: élément		ajoute l'élément à la fin de la queue
retirer			enlève l'élément qui se trouve en tête de queue

Sélecteurs de queue

opération	paramètre	résultat	description
tête		élément	retourne la valeur de l'élément qui est en tête de queue
estVide		booléan	retourne vrai si la queue est vide
longueur		entier	retourne la longueur de la queue

Interface de la queue en Oberon:

```
DEFINITION Queue;
	TYPE
		Elément = ...; (* CHAR par exemle *)
		Queue = POINTER TO RECORD
			(q:Queue)PROCEDURE Ajouter (el: Elément), NEW;
			(q:Queue)PROCEDURE EstVide(): BOOLEAN, NEW;
			(q:Queue)PROCEDURE InitQueue, NEW;
			(q:Queue)PROCEDURE Longueur(): INTEGER, NEW;
			(q:Queue)PROCEDURE Retirer(), NEW;
			(q:Queue)PROCEDURE Tête(): Elément, NEW;
		END;
END Queue.
```

Exemple d’utilisation: simulation d’une file d’attente

- Simulation discrète
- But: simulation d’un système complexe dont on ne connaît pas de formule analytique
- discrète -> le temps avance par saccade
- on s’intéresse à certains instants -> points d’observation

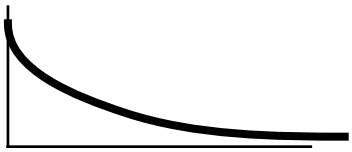
Simulation d’une file d’attente à un guichet (suite)

Paramètres

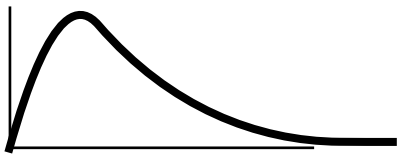
- temps moyen d’arrivée entre deux clients
- temps moyen de service
- nombre de guichets

Modélisation

- file d’attente des clients -> queue (temps de service)
- loi de probabilité des arrivées des clients: exponentielle négative



- loi de probabilité des temps de service: loi de poisson



Mesures

- temps moyen d’attente, temps maximumd’attente
- % d’occupation des guichets
- simulation graphique des files
- etc.

Algorithme de simulation discrète

```
tant que t < Max {
    si t_prochaine_arrivée < t_fin_service {
        t := t_prochaine_arrivée
        n := n+1
        file.ajouter(aléa_poisson) (* temps de service *)
        t_prochaine_arrivée := t + aléa_exponentielle
        si file.longueur = 1
            t_fin_de_service := t + file.tête
    }
    sinon {
        t := t_fin_service
        file.retirer
        si file.longueur > 0
            t_fin_de_service := t + file.tête
        sinon
            t_fin_service = infini
    }
}
```

- *aléa_poisson* et *aléa_exponentielle* sont des nombres pseudo-aléatoires générés conformément à leur loi de probabilité
- *infini* est un nombre très grand (pour que la condition à la 2ème ligne de l'algorithme soit vraie)

Queue prioritaire

- C’est une variante de la structure de queue (variante que nous avons déjà vue au chapitre 2).
- Rappel de la définition: “Structure à laquelle les éléments sont ajoutés dans n’importe quel ordre et retirés dans l’ordre de leur priorité.“

Exemples d’application en informatique:

- allocation du processeur en fonction de la priorité des processus. Par ex., un processus s’occupant d’une tâche critique aura une priorité élevée.
- queue d’impression donnant la priorité à des fichiers courts par rapport à des fichiers très longs

Dans la vie réelle:

- le service d’urgences d’un hôpital traitera les cas les plus graves en premiers
- un avion qui doit se poser rapidement en raison d’une avarie technique aura priorité pour l’atterissage

Spécification de la queue prioritaire

Remarque: les opérations sont les mêmes qu'au chapitre 2 à l'exception de

- "Vider" qui devient "initQueue"
- "Retirer" qui est éclatée en deux opérations "Retirer" et "Tête"
- nouvelle opération "EstVide" qui renseigne si la queue est vide

Constructeurs / modifieurs de queue prioritaire

opération	paramètre	résultat	description
new		q: QueueP	crée une nouvelle queue q
initQueue			initialise la queue à vide
ajouter	e: élément		ajoute l'élément à la queue
retirer			enlève l'élément qui a la plus grande priorité de la queue

Sélecteurs de queue prioritaire

opération	paramètre	résultat	description
tête		élément	retourne la valeur de l'élément qui a la plus grande priorité de la queue
estVide		booléan	retourne vrai si la queue est vide
longueur		entier	retourne la longueur de la queue

Interface de la queue prioritaire en Oberon:

```
DEFINITION QueuePrioritaire;
    TYPE
        Elément = ...; (* INTEGER par exemle *)
        QueueP = POINTER TO RECORD
            (q:QueueP)PROCEDURE Ajouter (el: Elément), NEW;
            (q:QueueP)PROCEDURE EstVide(): BOOLEAN, NEW;
            (q:QueueP)PROCEDURE InitQueue, NEW;
            (q:QueueP)PROCEDURE Longueur(): INTEGER, NEW;
            (q:QueueP)PROCEDURE Retirer(), NEW;
            (q:QueueP)PROCEDURE Tête(): Elément, NEW;
        END;
END QueuePrioritaire.
```

Séquence (ou liste ordonnée)

Séquence: “Une séquence est une collection d’objets qui sont placés selon un ordre. Les objets peuvent être insérés et retirés à n’importe quelle position, mais de manière à ce qu’un ordre linéaire stricte soit préservé”.

Remarques:

- Chaque objet possède une position.
- Le modèle mathématique d’une séquence d’objet est la fonction $\{1 \rightarrow o_1, 2 \rightarrow o_2, \ldots, n \rightarrow o_n\}$ de $\{1,2,..,n\}$ dans $\{o_1, o_2, \ldots, o_n\}$

Utilisation

- Toute collection de données organisée selon un ordre linéaire
- Exemple: un fichier de type texte

Spécification de la séquence

Constructeurs / modifieurs de séquence

opération	paramètres	résultat	description
new		s: Séquence	crée une nouvelle séquence s
initSéquence			initialise la séquence à vide
insérerEnI	e: élément i: entier		insère l'élément e à la position i
supprimerI	i: entier		supprime le i ème élément
insérerDébut	e: élément		ajouter l'élément e au début de la séquence
insérerFin	e: élément		ajouter l'élément e à la fin de la séquence
supprimer-Début			supprime l'élément du début de la séquence
supprimerFin			supprime l'élément de fin

Sélecteurs de séquence

opération	paramètre	résultat	description
élémentI	i: entier	élément	retourne la valeur qui se trouve à la position i
début		élément	retourne la valeur de l'élément du début de la séquence
fin		élément	retourne la valeur de l'élément de fin de la séquence
indice	e: élément	entier	retourne la position de l'élément
estVide		booléan	vrai si la séquence est vide
longueur		entier	retourne la longueur de la séq.

Interface de la séquence en Oberon:

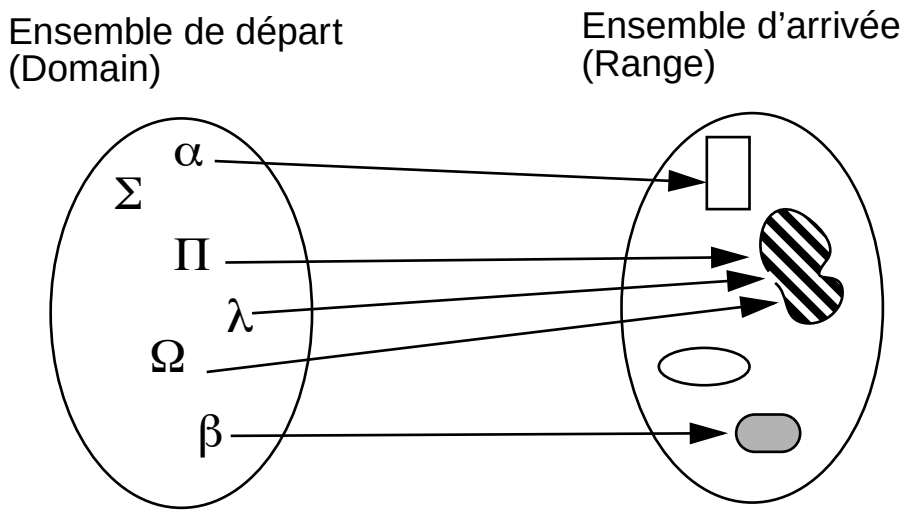
```
DEFINITION Séquence;
    TYPE
        Elément = ...;
        Séquence = POINTER TO RECORD
                (s:Séquence)PROCEDURE Début(): Elément, NEW;
                (s:Séquence)PROCEDURE ElémentI(i:INTEGER):
                                                Elément, NEW;
                (s:Séquence)PROCEDURE EstVide():
                                                BOOLEAN, NEW;
                (s:Séquence)PROCEDURE Fin(): Elément, NEW;
                (s:Séquence)PROCEDURE Indice(clé: Elément):
                                                INTEGER, NEW;
                (s:Séquence)PROCEDURE InitSéquence, NEW;
                (s:Séquence)PROCEDURE InsérerDébut(e: Elément), NEW;
                (s:Séquence)PROCEDURE InsérerEnI(e: Elément;
                                                i: INTEGER), NEW;
                (s:Séquence)PROCEDURE InsérerFin (e: Elément), NEW;
                (s:Séquence)PROCEDURE Longueur(): INTEGER, NEW;
                (s:Séquence)PROCEDURE SupprimerDébut, NEW;
                (s:Séquence)PROCEDURE SupprimerFin, NEW;
                (s:Séquence)PROCEDURE SupprimerI (I: INTEGER), NEW;
        END;
END Séquence.
```

Fonction

Fonction: “Une fonction partielle d’un ensemble de départ A dans un ensemble d’arrivée B relie chaque élément a de A à au plus un élément b de B”.

Une fonction est donc un ensemble dynamique de paires ordonnées (a -> b). Les paires (liens) peuvent être créés et supprimés pendant la vie de la fonction.

Illustration:



Remarques:

- Fonction en anglais: Map
- Ne pas confondre fonction ensemble de paires avec fonction au sens langage de programmation, càd suite d’instruction calculant un résultat à partir de paramètres.

Utilité

On peut se demander pourquoi il est nécessaire de définir une structure de données *fonction* alors qu'apparemment il suffirait d'ajouter un champ à A pour stocker le lien vers B.

Cette solution n'est pas bonne pour plusieurs raisons:

- elle nécessite la modification de la classe A -> le principe de modularité n'est pas respecté (dans le futur les objets de la classe A seront peut-être liés à des objets d'autres classe C, D etc. -> à chaque fois il faudra modifier A).
- les opérations globales (cardinalité, estVide) sont difficile à réaliser directement sur les instances des classes.
- il n'est pas possible de retrouver les objets de départ liés à un objet d'arrivée

Tout ceci justifie donc l'utilisation d'une structure de données *fonction.*

Tableaux et fonctions

Un tableau de taille N dont les éléments sont de type T (ARRAY N OF T) peut être vu comme une fonction de l'ensemble {0, 1, ..., N-1} vers l'ensemble des objets de type T

Un tableau est donc un cas particulier de fonction.

Spécification de la fonction

Constructeurs / modifieurs de fonction

opération	paramètre	résultat	description
new		f: Fonction	crée une nouvelle fonction f
initFonction			initialise la fonction à vide
lier	e1: départ e2: arrivée		lie l'élément e1 de l'ensemble de départ à l'élément e2 de l'ensemble d'arrivée
délier	e1: départ		enlève le lien de l'élément e1

Sélecteurs de fonction

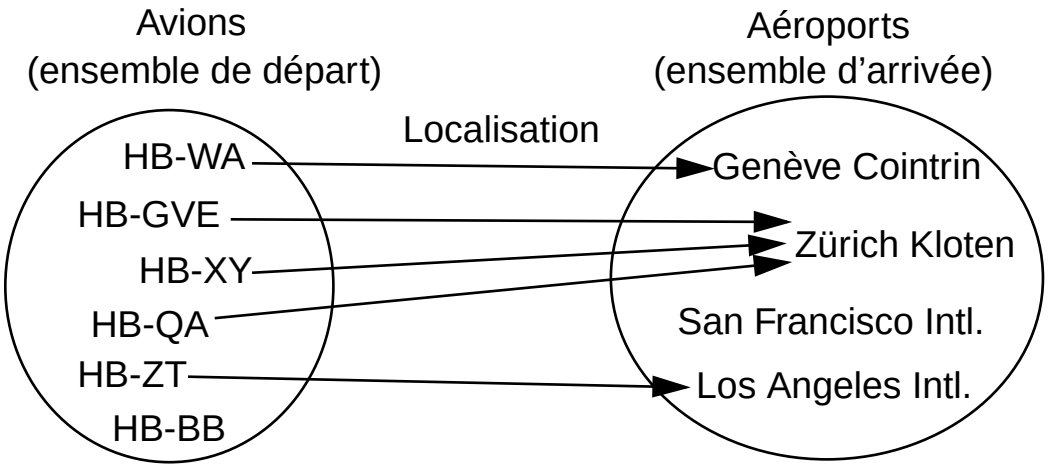
opération	paramètre	résultat	description
image	e: départ	arrivée	retourne la valeur de l'élément qui est lié à l'élément e
estVide		booléan	retourne vrai si la fonction est vide, i.e. ne contient aucun lien
cardinalité		entier	retourne la cardinalité de la fonction, i.e. le nombre de lien
(préimage)	e: arrivée	{départ}	retourne tous les éléments de l'ensemble de départ liés à l'élément e de l'ensemble d'arrivée

Interface de la fonction en Oberon:

```
DEFINITION Fonction;
    TYPE
        ElémentDépart = ...;
        ElémentArrivée = ...;
        EnsembleDépart = ...;
        Fonction = POINTER TO RECORD
                (f:Fonction)PROCEDURE Cardinalité(): INTEGER, NEW;
                (f:Fonction)PROCEDURE Délier(e: ElémentDépart), NEW;
                (f:Fonction)PROCEDURE EstVide(): BOOLEAN, NEW;
                (f:Fonction)PROCEDURE Image(e: ElémentDépart):
                                                ElémentArrivée, NEW;
                (f:Fonction)PROCEDURE InitFonction, NEW;
                (f:Fonction)PROCEDURE Lier(e1: ElémentDépart;
                                        e2: ElémentArrivée), NEW;
                (f:Fonction)PROCEDURE PréImage(e: ElémentArrivée):
                                                EnsembleDépart, NEW;
        END;
END Fonction.
```

Fonction: exemple d'utilisation

Localisation des avions d'une compagnie



Ebauche de code Oberon:

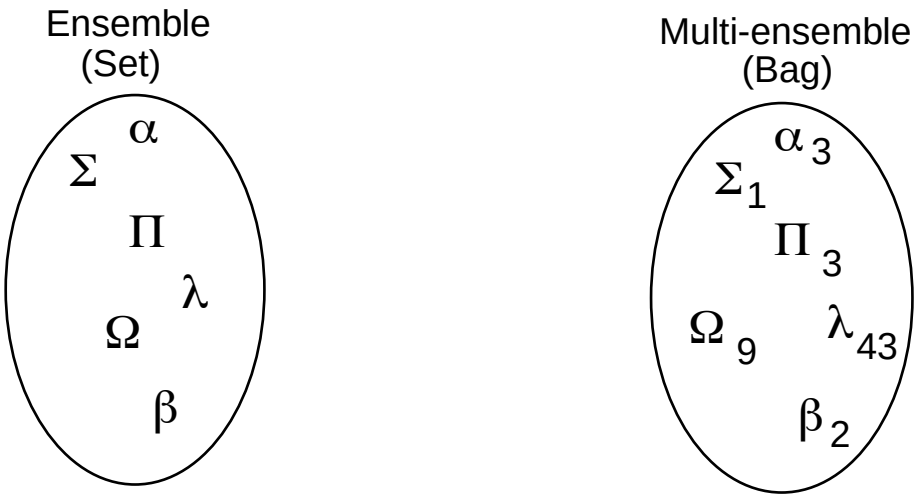
```
TYPE Avion = POINTER TO RECORD
                immatriculation, modèle: String;
             END;
     Aéroport = POINTER TO RECORD
                nom, abréviation: String;
             END;
VAR avion: ARRAY N OF Avion;
     aéroport: ARRAY M OF Aéroport;
     localisation : Fonction;
...
NEW(localisation); localisation.initFonction();
NEW(avion[0]); avion[0].immatriculation:=”HB-WA”;
NEW(aéroport[0]); aéroport[0].nom:=”Genève Cointrin”;
(* lier l'avion HB-WA à Genève Cointrin: *)
localisation.lier(avion[0], aéroport[0]);
```

Ensemble et multi-ensemble

Ensemble: “Collection d’objets distincts non ordonnés”.

Un ensemble ne peut pas contenir d’objet dupliqué.

Multi-ensemble: “Ensemble ou chaque élément peut apparaître plusieurs fois”.



Remarques:

- pour représenter les multi-ensembles on ajoute un indice à chaque élément pour indiquer son nombre d’occurences
- un ensemble dont tous les éléments appartiennent au même type ou à la même classe est dit homogène

Spécification de l’ensemble

Constructeurs / modifieurs d’ensemble

opération	paramètre	résultat	description
new		A: Ensemble	crée un nouvel ensembleA
initEnsemble			initialise l’ensemble à vide
ajouter	e: élément		ajoute l’élément e à l’ensemble
retirer	e: élément		retirer l’élément e de l’ensemble

Sélecteurs d’ensemble

opération	paramètre	résultat	description
appartient	e: élément	booléen	retourne vrai si l’élément e appartient à l’ensemble
union	B: ensemble	ensemble	retourne l’union de l’ensemble avec l’ensemble B
intersection	B: ensemble	ensemble	retourne l’intersection de l’ensemble avec l’ensemble B
différence	B: ensemble	ensemble	retourne la différence de l’ensemble avec l’ensemble B
estSous Ensemble	B: ensemble	booléan	retourne vrai si l’ensemble est le sous-ensemble de B
estVide		booléan	retourne vrai si l’ensemble est vide
cardinalité		integer	retourne la cardinalité de l’ensemble (nb. d’éléments appartenant à l’ensemble)

Spécification du multi-ensemble

Constructeurs / modifieurs de multi-ensemble

opération	paramètre	résultat	description
idem ensemble	...	...	...

Sélecteurs d'ensemble

opération	paramètre	résultat	description
idem ensemble +	...	...	...
nombre Occurences	e: élément	entier	retourne le nombre d'occurences de l'élément e dans le multi-ensemble
cardinalité Unique		integer	retourne le nombre d'éléments distincts du multi-ensemble

Interface de l’ensemble en Oberon:

```
DEFINITION Ensemble;
    TYPE
        Elément = ...;
        Ensemble = POINTER TO RECORD
                (A:Ensemble)PROCEDURE Ajouter(e: Elément), NEW;
                (A:Ensemble)PROCEDURE Appartient(e: Elément):
                                                BOOLEAN, NEW;
                (A:Ensemble)PROCEDURE Cardinalité(): INTEGER, NEW;
                (A:Ensemble)PROCEDURE Différence(B: Ensemble):
                                                Ensemble, NEW;
                (A:Ensemble)PROCEDURE EstSousEnsemble
                                        (B: Ensemble): BOOLEAN, NEW;
                (A:Ensemble)PROCEDURE EstVide(): BOOLEAN, NEW;
                (A:Ensemble)PROCEDURE InitEnsemble, NEW;
                (A:Ensemble)PROCEDURE Intersection(B: Ensemble):
                                                Ensemble, NEW;
                (A:Ensemble)PROCEDURE Retirer(e: Elément), NEW;
                (A:Ensemble)PROCEDURE Union(B: Ensemble):
                                                Ensemble, NEW;
        END;
END Ensemble.
```

Ensemble: exemple d’utilisation

“Ecrire un algorithme qui affiche la liste des balises apparaissant dans un document HTML.” Remarque: les balises HTML sont entourées par les caractères < et >.

Exemple de document HTML:

```
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="Adobe PageMill 2.0 Mac">
  <TITLE>Luka Nerima</TITLE>
</HEAD>
<BODY>
<H2>Luka Nerima</H2>
<DL>
  <DT>Charg&eacute; d'enseignement en informatique, <A HREF="http://
www.unige.ch/lettres/linge/linge.html">D&eacute;partement
  de linguistique g&eacute;n&eacute;rale</A> facult&eacute; des lettres et
  <A HREF="http://cui.unige.ch/">Centre Universitaire d'Informatique</A>
  d&eacute;partement de Syst&egrave;mes d'information, facult&eacute; des
  SES.
  <DT> 
</DL>
<H3>Contact</H3>
<DL>
  <DT>E-mail: nerima@uni2a.unige.ch
  <DT>T&eacute;l: +(41 22) 705 73 63 / 78 16
  <DT>Adresse: Centre Universitaire d'Informatique, 24, rue G&eacute;n&eacute;ral
```

Ensemble: solution de l’exemple d’utilisation

Données:
A:ensemble, c:tableau de N caractères qui contient le texte source du document HTML, b: chaîne de caractères (séquence de caractères) pouvant contenir une balise

Algorithme:

```
A.initEnsemble
tant que i <= N faire {
    si (c[i] = ‘<‘ ) {
        b.initSéquence
        répéter
            b.ajouterFin(c[i])
            i := i+1
        jusqu’à ce que c[i] = ‘>’
        A.ajouter(b)
    }
    sinon incrémenter i
}
```

Remarques:

- l’algorithme devient (presque) trivial par l’utilisation d’une structure de données ensemble
- pour calculer en plus le nombre d’apparitions de chaque balise, il suffit d’utiliser un multi-ensemble

Multi-ensemble: exemple d'utilisation

Voir TP1, vérification de la loi de Zipf

Ah, si j'avais les multi-ensembles !

Données:
A:multi-ensemble, c:tableau de N caractères qui contient le texte source, m: chaîne de caractères (séquence de caractères) qui contient un mot, sep: ensemble des séparateurs (" ", ",", "." etc)

Algorithme:

```
A.initEnsemble
initialiser i à 0
tant que i <= N faire {
    si c[i] ∉ sep {
        m.initSéquence
        répéter
            m.ajouterFin(c[i])
            i ← i + 1
        jusqu'à ce que c[i] ∈ sep
        A.ajouter(m)
    }
    sinon incrémenter i
}
```

Il ne reste plus qu'à trier les mots par ordre décroissant de leur fréquence -> suite de l'algorithme p. 4-41

Les itérateurs sur les collections

But: très souvent dans les algorithmes nous avons besoin d'examiner tous les éléments d'une structure de données sans perturber l'état de la structure elle-même.

Cette opération est:

- itérer

que l'on peut systématiquement ajouter à toute les structures de données collection que nous avons vues jusqu'à présent: pile, queue, séquence, fonction, ensemble (et multi-ensemble)

Remarques:

- dans le cas de la pile et de la queue l'itérateur viole le protocole d'accès à un seul élément (le sommet de pile, rsp. la tête de la queue); dans le cas exceptionnel de l'itérateur, on accepte cette violation.
- pour les collections ordonnées (pile, queue et séquence) l'ordre de parcours est imposé (du sommet au fond, du premier au dernier, du début à la fin)
- pour les collections non-ordonnées, l'ordre de parcours est arbitraire (l'ordre est choisi par l'itérateur)
- il y a deux manière de spécifier l'itérateur: l'itérateur actif et l'itérateur passif

(1) Itérateur actif

On peut voir un itérateur comme un objet d'une classe caractérisée par les opérations suivantes:

Itérateur actif

opération	paramètre	résultat	description
init	S: Structure	itr: Itérateur	associe un itérateur itr à la structure de donnée S
courant		élément	retourne la valeur de l'élément courant
suivant			avance l'itérateur sur l'élément suivant
fini		booléen	retourne vrai si la collection a été entièrement visitée

Itérateur actif: exemple

Suite de l'algorithme "vérification de la loi de Zipf": tri des mots par ordre décroissant des fréquences

Données:
c: itérateur sur le multi-ensemble A, T: tableau des mots avec leur fréquence trié selon les fréquences

Algorithme:

```
itr.init(A)
T[0].freq := A.nombreOccurences(itr.courant)
T[0].mot := itr.courant
p := 1 ; itr.suivant
tant que ¬itr.fini { (* itérateur actif *)
    freq := A.nombreOccurences(itr.courant)
    p := p + 1
    pour i de p à 1 descendant { (* tri par insertion *)
        j := i
        tant que j > 0 et freq > T[j-1].freq {
            T[j] := T[j-1]
            j := j-1
        }
        T[j].freq := freq
        T[j].mot := itr.courant
    }
    itr.suivant
}
```

(2) Itérateur passif

Dans ce cas, au lieu d'exporter un type Itérateur avec ses opérations associées, on ajoute directement à la spécification de la structure de données elle-même l'opération *itérer* ainsi qu'une *procédure générique*.

A l'appel de l'itérateur, cette procédure sera appliquée systématiquement à tous les éléments de la structure. C'est au client de la structure de définir *effectivement* la procédure générique.

Itérateur passif

opération	paramètre	résultat	description
itérer	p: procédure		applique la procédure p successivement à chaque élément de la structure

Itérateur passif: exemple

- Dans l'exemple des balises HTML, pour afficher les balises collectées dans l'ensemble A, on ajoutera à la fin de l'algorithme:
  ...
  A.itérer(afficher)

- et dans l'exemple de la vérification de la loi de Zipf, :
  ...
  T.itérer(afficher)

- dans les deux cas, il faudra déclarer et écrire le code d'une procédure *afficher* qui affiche la balise, resp. le mot et sa fréquence d'apparition dans le texte.

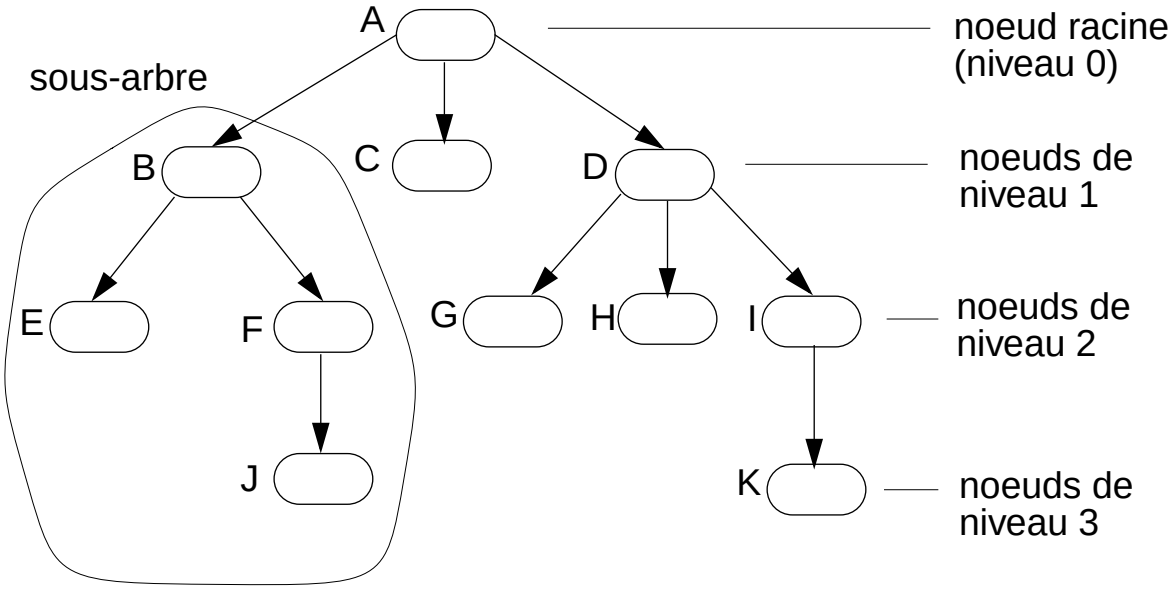
ARBRE

La structure d'arbre possède un double intérêt:

- dans de nombreux problèmes les données sont naturellement structurées en arbres; exemples:
  - arbre généalogique, organigramme
  - arbre syntaxique
  - arbre de décision
  - document structuré (livre->chapitres->sections->..)
- les structures d'arbres permettent d'implémenter efficacement la structure de données **ensemble** ou **fonction**; on parle dans ce cas d'**arbres de recherche** ("search tree" en anglais) que nous étudierons dans quelques semaines.
- Définition: "Un arbre est composé de deux ensembles N et A appelés respectivement l'ensemble des noeuds et l'ensemble des arcs, et d'un noeud particulier appelé racine de l'arbre. Les éléments de A sont des paires (n1,n2) d'éléments de N. Un arc (n1,n2) établit une relation entre n1, appelé noeud parent, et n2, appelé noeud enfant de n1, A doit être tel que chaque noeud, sauf la racine, a exactement un parent. "

Knuth dixit: "L'arbre est la structure non linéaire la plus importante que l'on trouve dans les algorithmes informatiques."

Illustration



(le noeud A est le noeud racine, les noeuds B,F,D et I sont les noeuds internes et les noeuds E,J,C,G,H et K sont les noeuds feuille de l'arbre)

Trois types de noeuds:

- **Racine**: le seul noeud de l'arbre qui n'a pas de parent.

- **Feuille**s de l'arbre: les noeuds qui n'ont pas d'enfants.

- **Noeuds** intérieurs: les noeuds qui ne sont ni des feuilles ni la racine.

Autres caractéristiques de l’arbre

Chaque noeud possède:

- un **degré**: le nombre de ses enfants.

- un **niveau**: nombre d’arcs qu’il faut remonter pour atteindre la racine.

- A chaque noeud est aussi associé une élément qu’on appelle ***contenu*** du noeud ou ***valeur*** du noeud

Le **degré** de l’arbre est le plus grand degré qu’on trouve parmi ses noeuds. On peut fixer un degré maximum pour un arbre. On parlera alors d’arbre **unaire**, **binaire**, **ternaire**...

La **hauteur** de l’arbre est le plus grand niveau qu’on trouve parmi ses noeuds.

Si l’ordre entre les sous-arbres enfant est pris en compte, on parlera d’**arbre ordonné** (à ne pas confondre avec un arbre trié).

Définition récursive d’une structure d’arbre

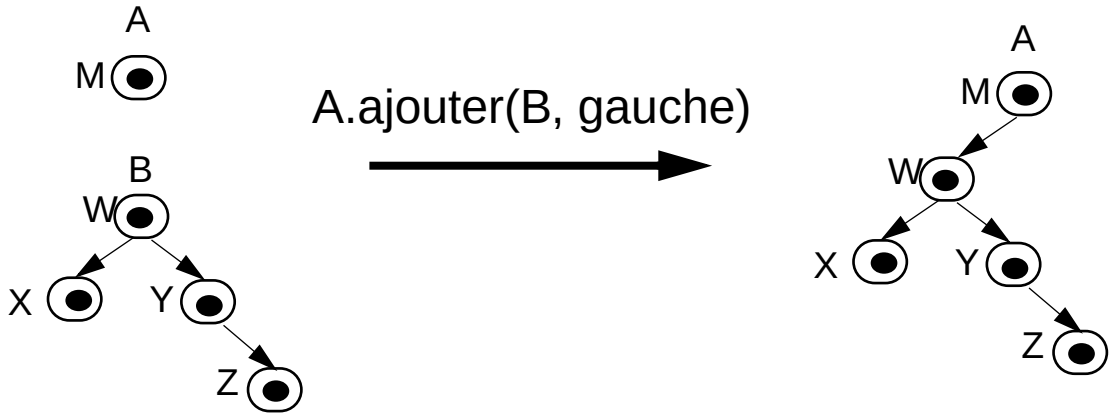
En tant que structure, on peut définir un arbre comme:

- un arbre vide;
- un noeud racine uniquement;
- ou bien un noeud racine qui possède une valeur et qui est lié à 0, 1 ou plusieurs sous arbres éventuellement vides.

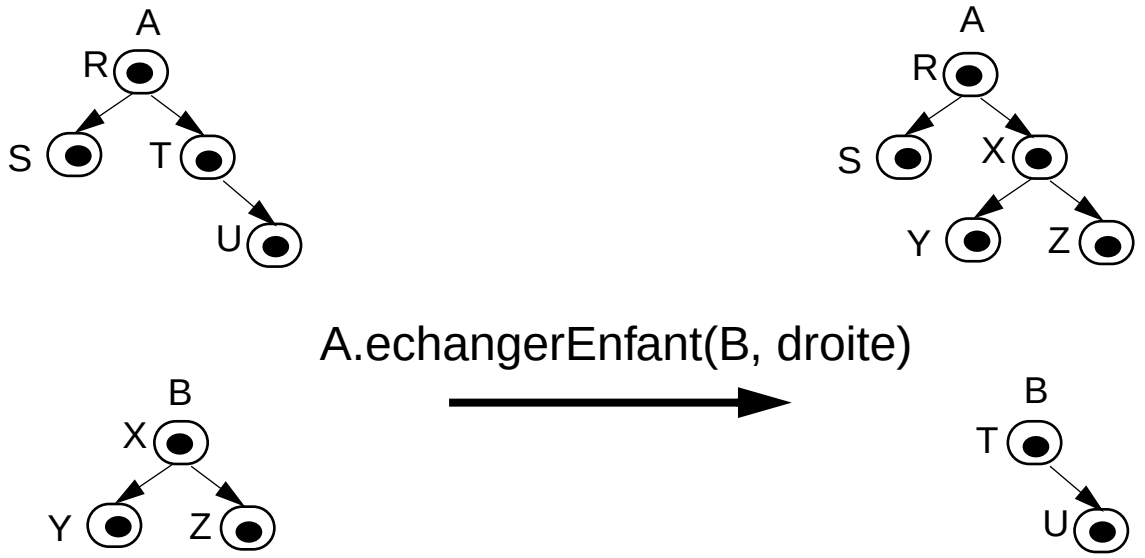
Les 2 modifieurs d’arbre les plus importants

Pour simplifier, nous considérons un arbre binaire

- Ajouter : ajouter un élément à la racine de l’arbre; l’arbre ajouté devient l’enfant de la racine.



- EchangerEnfant: échanger un arbre enfant donné avec un arbre complet.



Spécification de l’arbre (ordonné)

Ici l’arbre est de degré arbitraire -> on numérote les enfants: L’enfant le plus à gauche est numéroté 1, son voisin immédiat 2 et l’enfant le plus à droite *n*

Constructeurs / modifieurs d’arbre

opération	paramètre	résultat	description
new		A: Arbre	crée une nouvel arbre A
initArbre			initialise l’arbre à vide (aucun noeud)
ajouter (voir p 4-48)	B: Arbre i: entier		ajoute l’arbre B comme i-ème enfant de la racine de l’arbre
échangerEn-fant (p 4-48)	B: Arbre i: entier		échanger l’enfant i de la racine de l’arbre avec l’arbre B
modifierVal	v: élément		modifie la valeur de la racine de l’arbre
supprimer	i: entier		supprime le i-ème enfant de la racine de l’arbre
copier		Arbre	faire une copie de l’arbre

Sélecteurs d’arbre

opération	paramètre	résultat	description
enfant	i: entier	Arbre	retourne l’arbre qui est le i-ème enfant de la racine
nombreDen-fants		entier	retourne le degré de la racine
valeur		élément	retourne la valeur de la racine
estVide		booléen	vrai si l’arbre est vide
égal	B: Arbre		vrai si l’arbre a le même état B

Itérateurs d’arbre:

- On utilisera le sélecteur “enfant” pour parcourir récursivement un arbre à partir de sa racine.
- Contrairement à la structure de liste ou de queue, il n’y a pas d’ordre canonique pour parcourir un arbre, même ordonné. On distingue deux grandes catégories d’ordre de parcours:
  - le parcours en profondeur
  - le parcours en largeur

Parcours en profondeur

Dans un parcours en profondeur, à partir de la racine on descend dans un sous-arbre, qu'on explore complètement, puis on passe à un autre sous-arbre, et ainsi de suite jusqu'au dernier sous-arbre de la racine.

```
parcourirEnProfondeur(Arbre a) {
    parcourir la racine de a
    pour chaque enfant e de a
        parcourirEnProfondeur(e)
}
```

Parcours en largeur

Dans le parcours en largeur, on commence par parcourir tous les noeuds de niveau 1(les enfants de la racine), puis tous les noeuds de niveau 2, puis tous les noeuds de niveau 3 et ainsi de suite.

```
parcourirEnLargeur(Arbre a) {
    niveau_courant := {a}
    tant que niveau_courant est non vide {
        niveau_inférieur := {}
        pour chaque arbre s de niveau_courant {
            parcourir la racine de s
            niveau_inférieur := niveau_inférieur ∪ les
                                    enfants de s
        }
        niveau_courant := niveau_inferieur
    }
}
```

Représentation linéaire des arbres

L'affichage des arbres en deux dimensions est difficile à réaliser par programme. C'est pourquoi on recourt souvent à une représentation linéaire des arbres sous forme d'une séquence de symboles comprenant des parenthèses. Le principe consiste à représenter l'arbre sous la forme:

(racine enfant1 enfant2 ... enfantN **)**

Les enfants étant eux-même des arbres, on obtient une séquence avec des parenthèses imbriquées. Exemples:

(* (carré (+ 5 1)) (carré (* 5 2)))

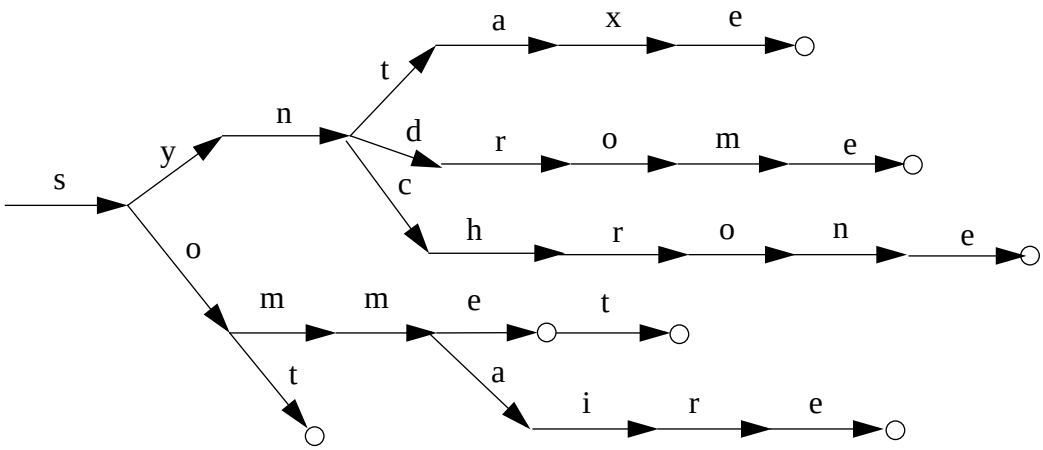
[TP[DP Marie][T' a [VP donné [DP le [NP chat]][PP à [DP Jean]]]]]

Exemple 1: Les arbres à lettres

Un arbre à lettre est une manière compacte de représenter un ensemble de mots (par exemple un lexique). L'idée est de regrouper tous les mots en un arbre dont chaque arc représente une lettre. Un mot est représenté par un chemin de la racine à un noeud contenant la valeur "fin de mot". Prenons par exemple les mots:

syndrome
synchrone
syntaxe
sommaire
somme
sommet
sot

L'arbre à lettres correspondant sera:



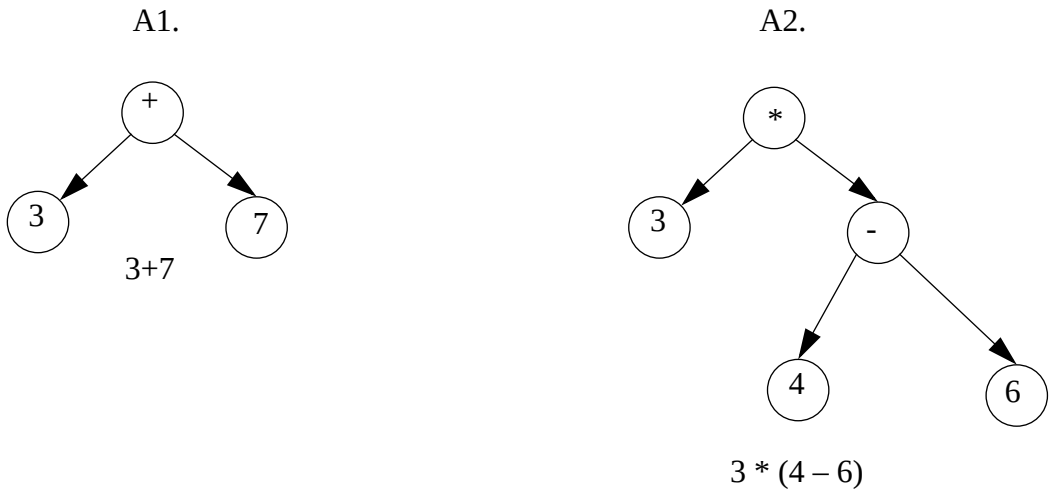
Les ○ sont des noeuds de fin de mot

Intérêt de l’arbre à lettre

L’intérêt de l’arbre à lettres réside dans la rapidité de la reconnaissance d’un mot dans le lexique. Pour tester si un mot formé des lettres c1, c2,...,ck se trouve dans le lexique, on part de la racine et on suit les arcs étiquetés par les lettres c1, c2 etc. Le mot appartient au lexique ssi on arrive sur un noeud “fin de mot”. Le temps de recherche est donc proportionnel à la longueur du mot et est indépendant de la taille du dictionnaire!

Exemple 2: La représentation et la manipulation d'expressions arithmétiques

On peut représenter une expression arithmétique par un arbre dont la racine contient un opérateur et les sous-arbres les opérandes. Par exemple:



Une telle représentation permet toutes sortes de manipulations des expressions:

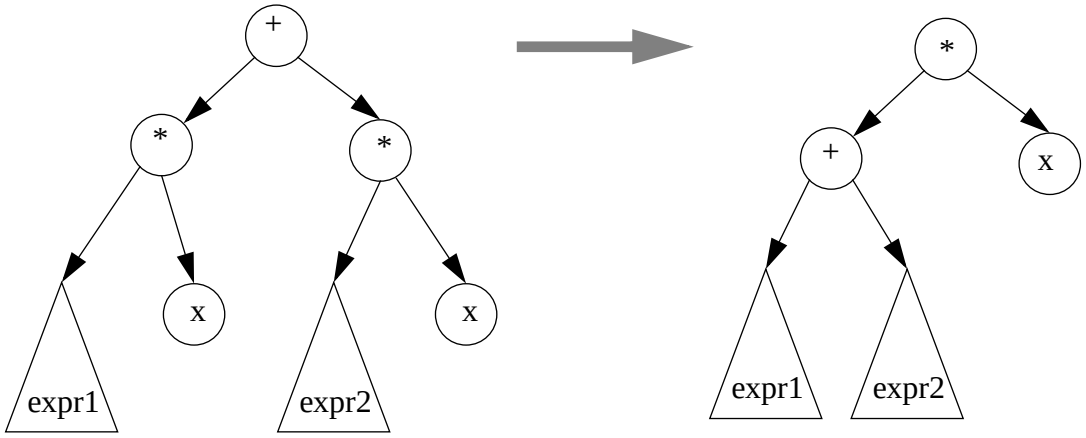
Evaluation:

L'évaluation consiste à remplacer un (sous-)arbre par la valeur de l'expression qu'il représente. Ainsi l'arbre A1 ci-dessus sera remplacé par l'arbre constitué du seul noeud (10); pour l'arbre A2 on évalue d'abord le sous-arbre (- 4 6) qui donne -2, puis (* 3 -2) qui donne -6.

Simplification:

Si l'expression contient des variables comme opérandes, on ne peut pas l'évaluer complètement. Par contre, on peut appliquer des simplifications du genre:

- remplacer (* 1 expr) ou (* expr 1) par (expr)
- remplacer (+ 0 expr) ou (+ expr 0) par (expr)
- remplacer (+ (* expr1 x) (* expr2 x))
  par (* (+ expr1 expr2) X)



Exemple de simplification d'expression

Dérivation:

On peut calculer la dérivée (selon x) d'une formule en appliquant des transformations telles que:

- d(* constante x) = (constante)
- d(** x n) = (* n(** x(- n 1)))
- d(+ f g) = (+ d(f) d(g))
- d(* f g) = (+ (* d(f) g) (* f d(g)))

Remarques:

- Cette représentation, avec quelques variantes, est la base des systèmes de manipulation et de résolution symbolique d'équation mathématiques (Mathematica, Maple, MathLab etc.).
- Elle peut également être utilisée dans les compilateurs pour vérifier le typage des expressions et pour les optimiser.
- Bien que nous n'ayons montré que des opérations mathématiques, cette représentation peut aussi s'appliquer à d'autres domaines où la notion d'opération et d'opérandes existe.

Implémentation des arbres en Oberon

Implémentation: simple et directe

- on définit une classe Noeud comprenant le contenu d'un noeud ainsi que les liens vers les enfants de ce noeud
- chaque noeud de l'arbre sera représenté par un objet de cette classe

```
MODULE Arbre;
CONST degreMax = 10; (* Le nombre 10 est arbitraire.
Par ex,pour l'arbre à lettres, on aurait degreMax=26*)

TYPE Element* = ...; (* type du contenu des noeuds*)
     Arbre* = POINTER TO Noeud;
     Noeud  = RECORD
                    contenu : Element;
                    enfant : ARRAY degreMax OF Arbre;
              END;
(* ... constructeurs, sélecteurs, itérateurs ...*)

END Arbre.
```

Remarques:

- si le nombre maximum d'enfant (degré de l'arbre) est fixé on utilisera un tableau pour représenter les enfants, une séquence de noeuds sinon.
- Si l'arbre est binaire, on utilisera les champs "gauche" et "droite" à la place d'"enfant".

ARBRE DE RECHERCHE

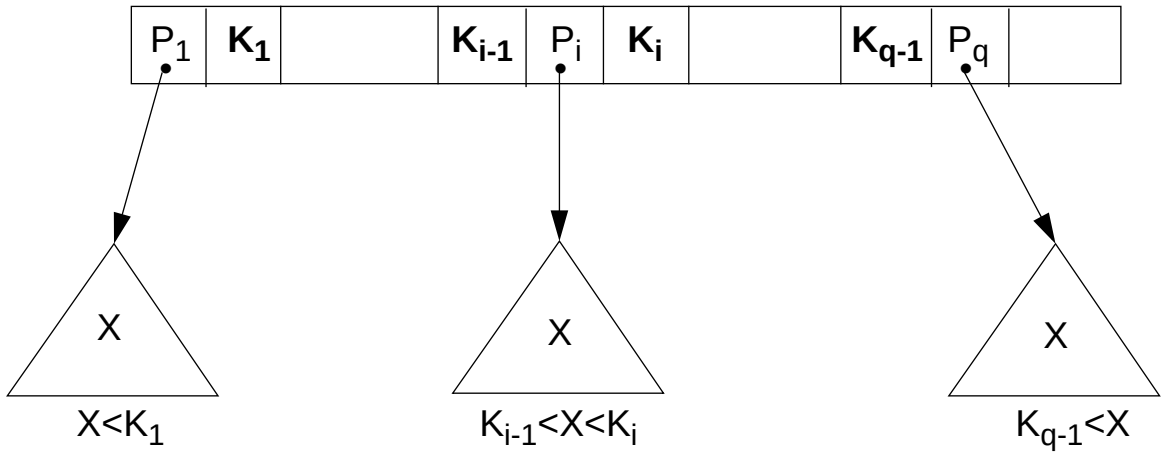
- Arbre spécial pour guider la recherche d'un élément (enregistrement) en fonction d'une clé et d'une valeur de clé.
- Ce n'est pas une structure de données du même niveau conceptuel que celle que nous avons vu jusqu'à présent (pile, queue, etc) mais une structure permettant de représenter efficacement d'autres structures de données (ensembles, fonctions).
- Condition: Il faut toutefois que les données possèdent une relation d'ordre ($\leq$).

Arbres de recherche

Arbre spécial pour guider la recherche d'un enregistrement en fonction d'une clé et d'une valeur de clé.

Arbre de recherche d'ordre p:

- Chaque noeud contient q-1 valeurs de clé et q pointeurs dans l'ordre $<P_1,K_1,P_2,K_2,...,P_{q-1},K_{q-1},P_q>$ avec $q \leq p$ et $K_1 < K_2 < ... < K_{q-1}$
  ($P_1,P_2,...,P_{q-1},P_q$ sont les pointeurs vers les noeuds descendant)



Un noeud peut donc avoir au maximum p descendants

Hauteur pour n valeurs: $\log_p(n)$ –> complexité de recherche $O(\log_p(n))$

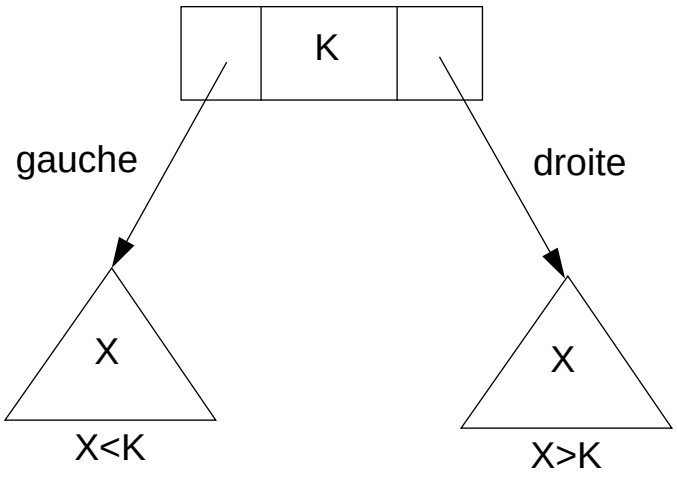
Remarque:

Nous supposons que la clé est unique. Cette restriction peut être levée, mais il faut changer légèrement les formules.

Arbre de recherche binaire

Chaque noeud contient:

- une clé (K)
- un pointeur gauche et un pointeur droit t.q.

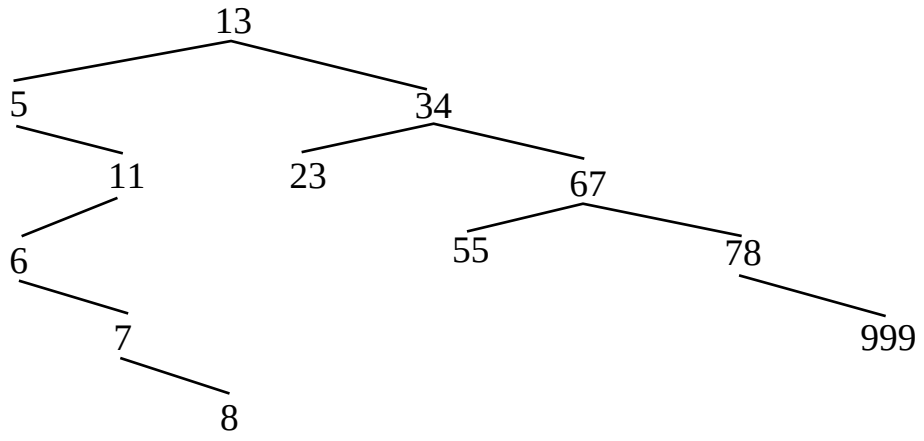


Arbre de recherche binaire

- Arbre de degré 2 -> chaque noeud a au maximum deux enfants, appelés gauche et droite.
- L'arbre binaire est construit de tel manière que pour tous les noeuds de l'arbre, (i) tous les éléments contenus dans le sous-arbre gauche d'un noeud ont une valeur de clé inférieure à celle du noeud considéré, (ii) tous les éléments du sous-arbre droit ont une valeur de clé supérieure à celle du noeud considéré.

Dans l'exemple d'arbre de recherche binaire ci-dessous, les éléments de l'arbre sont de type entier et jouent en-même temps le rôle de clé:

Séquence d'insertion: 13 34 5 67 78 55 23 11 999 6 7 8



Efficacité de la recherche

- La recherche d'un élément dans un arbre de recherche binaire qui contient *n* éléments requiert en moyenne $\log_2 n$ comparaisons.
- Le gain par rapport à une recherche linéaire (*n*/2 comparaisons) est d'autant plus important que n est grand
- Exemple:

  n=100'000. Nombre moyen de comparaisons:

  -> recherche linéaire: 50'000
  -> arbre binaire: 17

Spécification de l’arbre de recherche

Constructeurs / modifieurs d’arbre de recherche

opération	paramètre	résultat	description
new		A: Arbre	crée une nouvel arbre A
initArbre			initialise l’arbre à vide
insérer	e: Elément		insérer l’élément e dans l’arbre
supprimer	c: clé		supprime l’élément dont la clé vaut c
copier		Arbre	faire une copie de l’arbre

Sélecteurs d’arbre de recherche

opération	paramètre	résultat	description
rechercher	c: clé	Elément	retourne l’élément dont la clé vaut c
estVide		booléen	vrai si l’arbre est vide
égal	B: Arbre		vrai si l’arbre a le même état que l’arbre B

Itérateurs d’arbres de recherche

- Dans le cas des arbres de recherches, on ne considère que des parcours en profondeur
- Trois manières habituelles pour parcourir un arbre:
  - *en-ordre* :
    sous-arbre gauche → racine → sous-arbre droit
  - *pré-ordre* :
    racine → sous-arbre gauche → sous-arbre droit
  - *post-ordre* :
    sous-arbre gauche → sous-arbre droit → racine

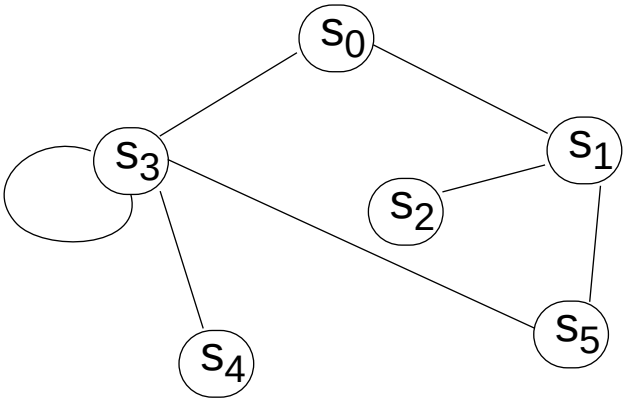
Utilisation:

- Le parcours *en-ordre* fournit une visite des éléments de l’arbre dans l’ordre croissant de leur clé.
- les parcours en *pré-ordre* et *post-ordre* conviennent bien à la construction d’un arbre; on l’utilisera par exemple pour faire une copie d’arbre.

Graphe

“Le graphe est composé de deux ensembles: S l’ensemble des sommets (ou noeuds) et A l’ensemble des arêtes. Les éléments de A sont des paires {s,u} d’éléments de S représentant un lien entre s et u ou des singletons {s} représentant un lien de s avec lui-même.”

Illustration:



$S=\{s_0, s_1, s_2, s_3, s_4, s_5\}$

$A=\{\{s_0, s_1\}, \{s_0, s_3\}, \{s_3\}, \{s_3, s_4\}, \{s_3, s_5\}, \{s_1, s_2\}, \{s_1, s_5\}\}$

- Le degré d’un sommet est le nombre d’arêtes qui contiennent ce sommet
- Deux sommets s,u sont adjacents s’il existe une arête {s,u}
- Un chemin de u_0 à u_n est une séquence d’arrêtes de la forme $\{u_0,u_1\}, \{u_1,u_2\}, ..., \{u_{n-2},u_{n-1}\}, \{u_{n-1},u_n\}$
- Un graphe est connexe si tous les sommets sont réliés deux à deux par au moins un chemin

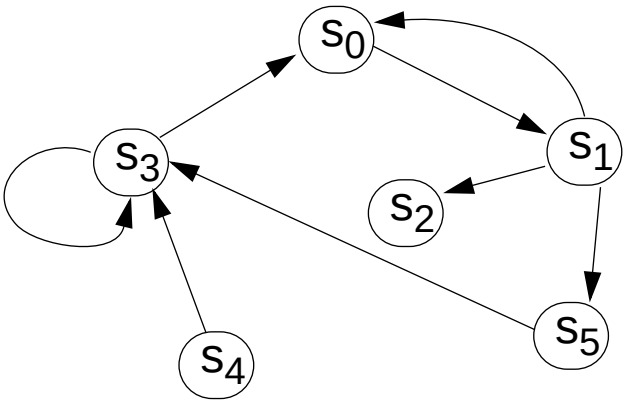
Graphe orienté

"Un graphe orienté est composé d'un ensemble de sommets et d'**arcs**. Les arcs sont des paires de sommets dont le premier est appelé *origine* et le second *destination*."

En anglais: directed graph

Notation: on notera les paires (s,u) au lieu de {s,u}

Graphiquement: les arcs sont fléchés



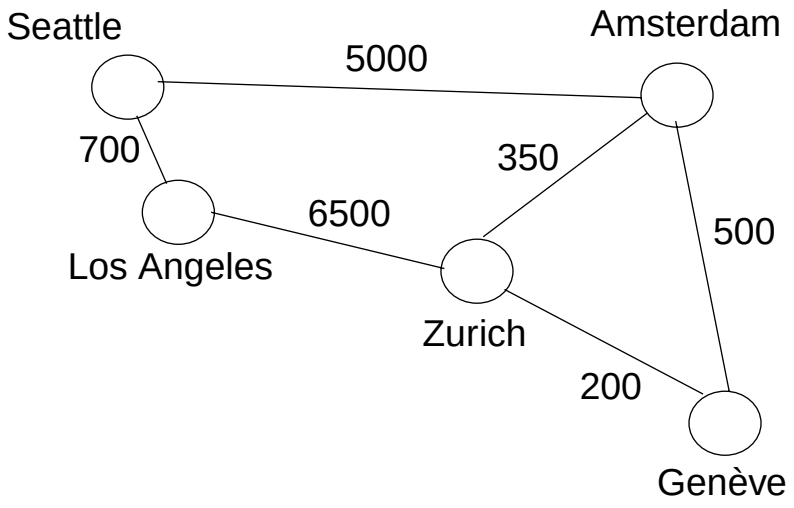
$S=\{s_0, s_1, s_2, s_3, s_4, s_5\}$

$A=\{(s_0, s_1),(s_1, s_0),(s_3, s_0)$
$(s_3), (s_4, s_3), (s_5, s_3),$
$(s_1, s_2), (s_1, s_5)\}$

- Un chemin de u_0 à u_n est une séquence d'arcs de la forme $(u_0,u_1), (u_1,u_2), ..., (u_{n-2},u_{n-1}), (u_{n-1},u_n)$

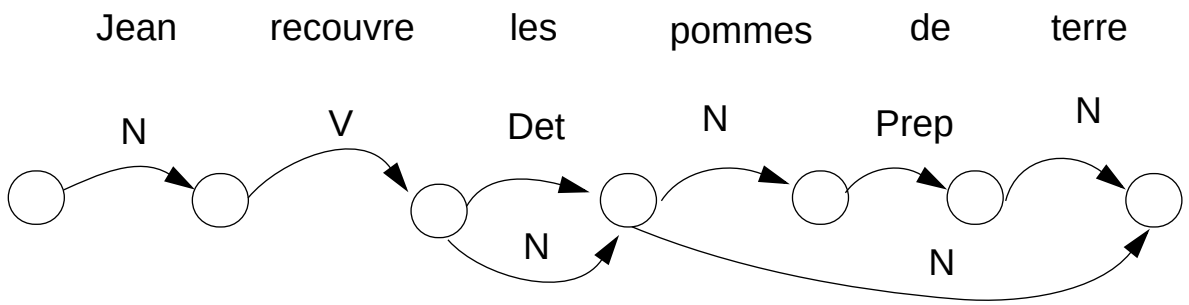
Graphe étiqueté

On peut également associer une valeur à chaque sommet et à chaque arête (par exemple pour indiquer un poids, un coût, une distance):



ou pour attribuer une valeur symbolique:

Pour représenter les ambiguïtés lexicales, à partir de la phrase “Jean recouvre les pommes de terre” on construit le graphe:



Utilité des graphes

La structure de graphe convient bien à la solution de très nombreux problèmes en informatique

En fait, on a une structure de graphe chaque fois qu'il y a une relation binaires entre des objets d'un même ensemble.

Exemples: réseaux de transport et de communication (physique ou électronique), relations entre personnes ou institutions, relations entre objets.

-> de très nombreux problèmes peuvent se ramener à des problèmes classique de la théorie des graphes.

Les graphes ont été l'objet de très nombreux travaux (Dijkstra, Knuth, Kruskal, ...). Parmi les problèmes les plus célèbres:

- recherche du plus court chemin
- recherche de composantes connexes
- arbre de recouvrement minimal
- Problèmes NP-complet:
  . couverture de sommets
  . recherche de cliques
  . circuit Hamiltonien
  . etc

Spécifiction du graphe orienté

Constructeurs / modifieurs de graphe

opération	paramètre	résultat	description
new		G: Arbre	crée une nouveau graphe G
initGraphe			initialise le graphe à vide
insérSom-met	s: Sommet		insérer le sommet s
insérerArc	so: Sommet sd: Sommet		insérer un arcs reliant le som-met origine so au sommet destination sd
supprimer-Sommet	s: Sommet		supprimer le sommet s
supprime-rArc	so: Sommet sd: Sommet		supprimer l'arc (so,sd)
copier		Graphe	faire une copie du graphe

Spécifiction du graphe orienté (suite)

Sélecteurs de graphe

opération	paramètre	résultat	description
arcs		ensemble d'arcs	retourne l'ensemble des arcs du graphe
sommets		ensemble de sommets	retourne l'ensemble des sommets du graphe
entrants	s: sommet	ensemble d'arcs	retourne l'ensemble des arcs qui ont pour destination s
sortants	s: sommet	ensemble d'arcs	retourne l'ensemble des arcs qui ont pour origine s

Itérateur de graphe

- Parcours d'un graphe: consiste à visiter tous les sommets une et une seule fois
- Comme pour les arbres, on peut effectuer un parcours en profondeur ou en largeur à partir d'un sommet.
- Toutefois, même pour le parcours en profondeur il faudra explicitement gérer l'ensemble des sommets déjà visités

Algorithme du parcours en profondeur:

```
procédure parcourirGraphe(G Graphe) {
    procédure parcourirDepuis(s sommet,SV ensemble) {
        visiter s
        SV := SV ∪ {s}
        pour toute arête {s, s'}
            si (s' ∉ SV} parcourirDepuis(s',SV)
    }
Vus := {} -- ensemble des sommets déjà visités
tant que (vus ≠ G.sommets) {
    choisir un sommet s ∈ G.sommets - Vus
    parcourirDepuis(s, Vus)
    }
}
```