Cours d'Algorithmique
Partie 2: Structures de données

Georges Edouard KOUAMOU

Ecole Nationale Supérieure Polytechnique

Algorithme et structures de données

- La plupart des bons algorithmes fonctionnent grâce à une méthode astucieuse pour organiser les données.
- Quatre grandes classes de structures de données sont explorés :
  - Les structures de données séquentielles (tableaux);
  - Les structures de données linéaires (liste chaînées);
  - Les structures hiérarchiques (arbres);
  - Les tables de hachage
- Les graphes ont été vus dans un cours précédent.

Concept

- Une structure de données est une manière d'organiser et de stocker l'information pour en faciliter l'accès ou dans d'autres buts
- Une structure de données a une interface et l'implémentation
  - l'interface consiste en un ensemble de procédures pour ajouter, effacer, accéder, réorganiser, etc. les données.
- Une structure de données conserve des données et éventuellement des méta-données
  - Par exemple : un tas utilise un tableau pour stocker les clés et une variable A.heap-size pour retenir le nombre d'éléments qui sont dans le tas.
- Conséquence: Une structure de données est indépendante de la technologie (langage) d'implémentation

Structure de données vs Types Abstraits

- Une **structure de données** est une mise en œuvre concrète d'un **type abstrait**.
- En informatique, un type abstrait (en anglais, abstract data type ou ADT) est une spécification mathématique :
  - d'un ensemble de données
  - et de l'ensemble des opérations qu'on peut effectuer sur elles.
- On qualifie d'abstrait ce type de données car il correspond à un **cahier des charges** qu'une structure de données doit ensuite mettre en œuvre.

Structures séquentielles

Tableau

- En anglais : array, vector.
- Un tableau est une structure de donnée T qui permet de stocker un certain nombre d'éléments T[i] repérés par un index i. Les tableaux vérifient généralement les propriétés suivantes :
  - tous les éléments ont le même type de base ;
  - le nombre d'éléments stockés est fixé ;
  - l'accès et la modification de l'élément numéro i est en temps constant $\Theta(1)$, indépendant de i et du nombre d'éléments dans le tableau.

Opérations de base

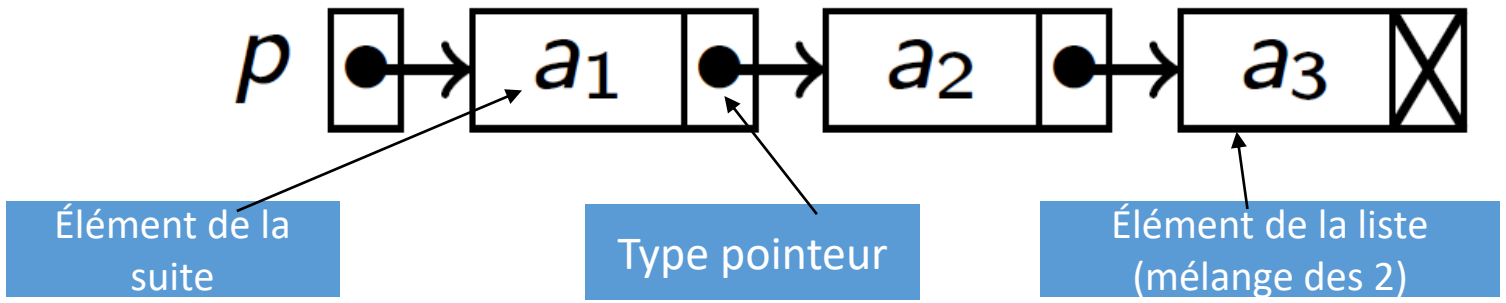
- Hypothèses :
  - tableau de taille max_taille alloué
  - éléments *0 ≤ i ≤ taille ≤ max_taille initialisés*
- Opérations de base
  - accès au premier élément : $\Theta(1)$
  - accès à l'élément numéro i : $\Theta(1)$
  - accès au dernier élément : $\Theta(1)$
  - insertion d'un élément au début : $\Theta(taille)$
  - insertion d'un élément en position i : $\Theta(taille - i) \in \Theta(taille)$
  - insertion d'un élément à la fin : $\Theta(1)$

Problème de taille

- On essaye d'insérer un élément dans un tableau où taille = max_taille
  - Il n'y a plus de place disponible.
- Comportements possibles :
  - Erreur (arrêt du programme, exception)
  - Ré-allocation du tableau avec recopie, coût : $\Theta(taille)$

Chaînages dynamiques

- Listes Dynamiques simplement chaînées (**LDSC**)
  - le pointeur p repère la LDSC qui implante la suite $\langle a_1; a_2; a_3 \rangle$
- Dans le schéma, trois types sont à distinguer :
  - le type des éléments de la suite
  - le type des pointeurs
  - le type des éléments de la liste dynamique, composition des deux types précédents



Définitions

LDSC définie itérativement

- La suite $\langle a_1; a_2; \ldots; a_n \rangle$ est implantée par la LDSC repérée par le pointeur p lorsque :
  - $p(\rightarrow next)^{k-1} \neq null, \qquad pour\ 1 \leq k \leq n$
  - $p(\rightarrow next)^{k-1} \rightarrow val = a_k, \qquad pour\ 1 \leq k \leq n$
  - $p(\rightarrow next)^{n} = null$

LDSC définie récursivement

- La suite U est implantée par la LDSC repérée par le p lorsque :
  - soit $U = \langle\ \rangle\ et\ p = NULL;$
  - soit U est de la forme $U = \langle a \rangle \cdot V$ avec :
    - $p \neq NULL\ et\ p \rightarrow val = a;$
    - la suite V est implantée par la LDSC pointée par $p \rightarrow next$.

Interface

- **Opérations** de base

Liste_vide : → Liste

Tête : Liste → Adresse

Fin : Liste → Liste

Cons : Elément x Liste → Liste

Premier : Liste → Elément

Elt : Adresse → Elément

Succ : Adresse → Adresse

- **Modifications**

Ajouter : Elément x Adresse x Liste → Liste

ajouter e après l'élément d'adresse p

Supprimer : Adresse x Liste → Liste

Elément : Elément x Liste → Booléen

- **Autres**

Place : Elément x Liste → Adresse

Position : Elément x Liste → Entier

Ième : Entier x Liste → Elément

Tri : Liste → Liste

- **Axiomes (exemples)**

Tête (L), Fin (L), Premier (L) définis ssi L ≠ **ε**

L ≠ **ε** ⇒ Premier (L) = Elt (Tête (L))

Fin (Cons (e, L)) = L Premier (Cons (e, L)) = e

L ≠ **ε** ⇒ Succ (Tête(L)) = Tête (Fin(L))

Techniques (1)

- LDSC avec pointeurs de tête et de queue
  - Pointeurs repérant respectivement la tête et la queue de la LDSC qui implante la suite $\langle a_1; a_2; \ldots; a_n \rangle$
  - Intérêts :
    - ajout en queue sans parcours de la LDSC ;
    - concaténation sans parcours des LDSC.
- LDSC circulaire
  - pointeur repérant la queue de la LDSC circulaire qui implante la suite
  - Intérêts :
    - ajout en queue et suppression en tête sans parcours de la LDSC ;
    - concaténation sans parcours des LDSC.

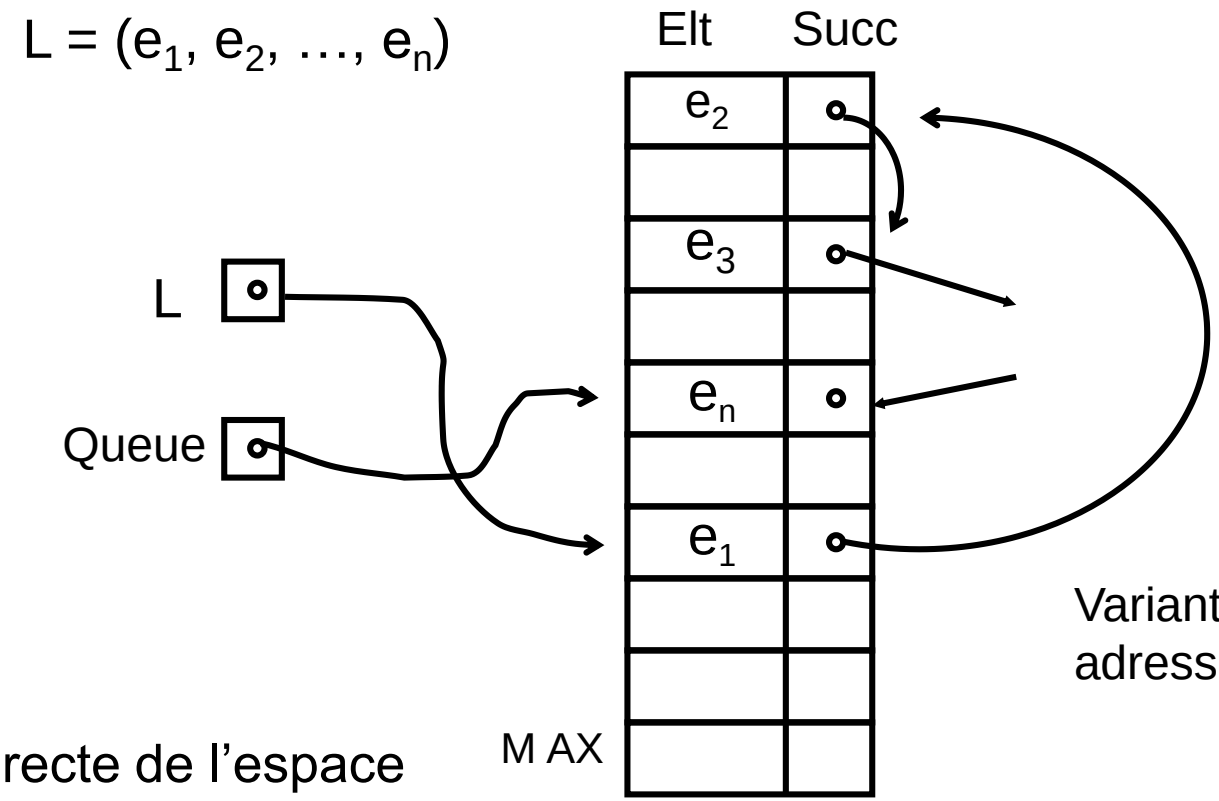
Techniques (2)

- Listes dynamiques doublement chaînées (LDDC)
  - pointeurs repérant respectivement la tête et la queue de la LDDC qui implante la suite
  - Intérêts :
    - marches avant et arrière ;
    - ajout en queue et suppression en tête sans parcours de la LDDC ;
    - concaténation sans parcours des LDD

Implémentations par pointeur

- **Avantages**
  - nombreuses opérations rapides
  - gestion souple, autorise plusieurs listes
  - accès à tout l’espace mémoire disponible
- **Inconvénients**
  - pas d ’accès direct
  - un peu gourmand en mémoire

Implémentation par curseur



Variante des pointeurs
adresses = indices

Avantage

Gestion directe de l'espace

Inconvénient

Choix de MAX ?

Applications

Implantation des types de données abstraits :

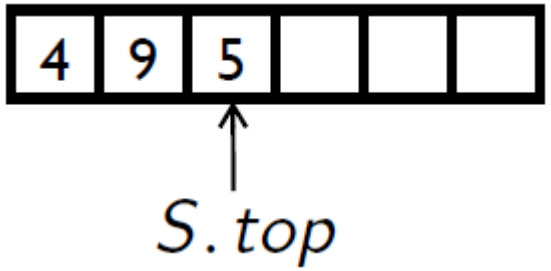
- piles (dernier entré - premier sorti)
- files d'attente (premier entré - premier sorti)
- double-queues
- ensembles, tables d'associations

Pile

- Ensemble dynamique d'objets accessibles selon une discipline LIFO ("Last-in first-out").
  - Liste avec accès par une seule extrémité
- Interface
  - **Stack-Empty(S)** renvoie vrai si et seulement si la pile est vide
  - **Push(S, x)** pousse la valeur x sur la pile S => *empiler*
  - **Pop(S)** extrait et renvoie la valeur sur le sommet de la pile S => *depiler*
- Implémentations :
  - avec un tableau (taille fixée a priori)
  - au moyen d'une liste liée (allouée de manière dynamique)
  - . . .

Implémentation avec un tableau

- S est un tableau (indicé à partir de 1) qui contient les éléments de la pile
- S.top est la position courante de l'élément au sommet de S
- Complexité des opérations: *O(1)*



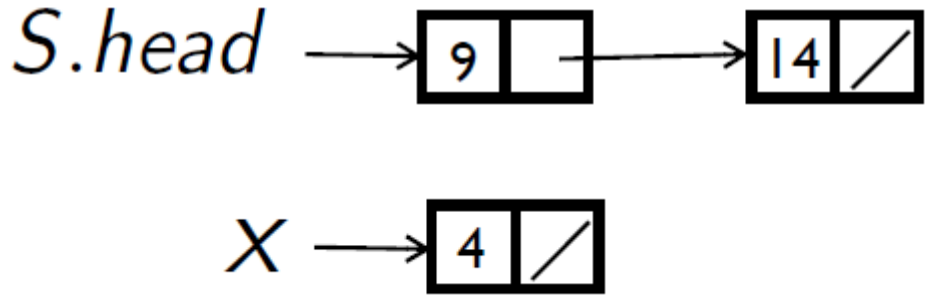

```
Push(S, x):void
1 if S.top == S.length
2     error "overflow"
3 S.top = S.top + 1
4 S[S.top] = x
```

```
Pop(S): element
1 if Stack-Empty(S)
2     error "underflow"
3 else S.top = S.top - 1
4     return S[S.top + 1]
```

```
Stack-Empty(S)
1 return S.top == 0;
```

Implémentation à l'aide d'une liste liée

- S est une liste simplement liée (S.head pointe vers le 1er élément de la liste)
- Les opérations se passent en tête de liste
- Complexité en temps des opérations: O(1)




```
Stack-Empty(S): boolean
1 if S.head == NIL
2     return true
3 else return false
```

```
Pop(S): element
1 if Stack-Empty(S)
2     error "underflow"
3 else x = S.head
4     S.head = S.head.next
5     return x
```

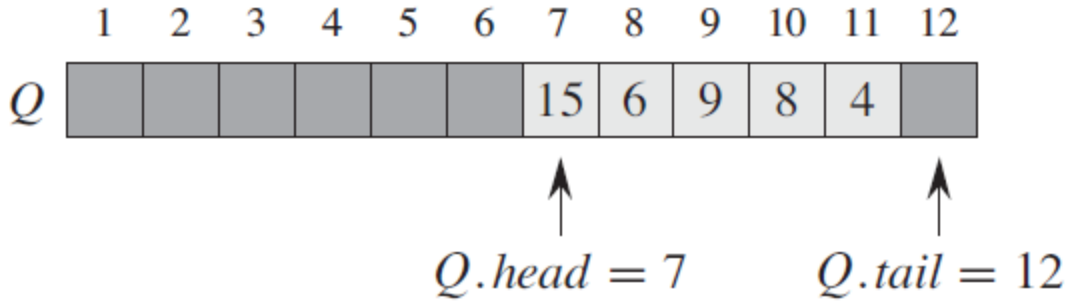
```
Push(S,x): void
1  x.next = S.head
2  S.head = x
```

File

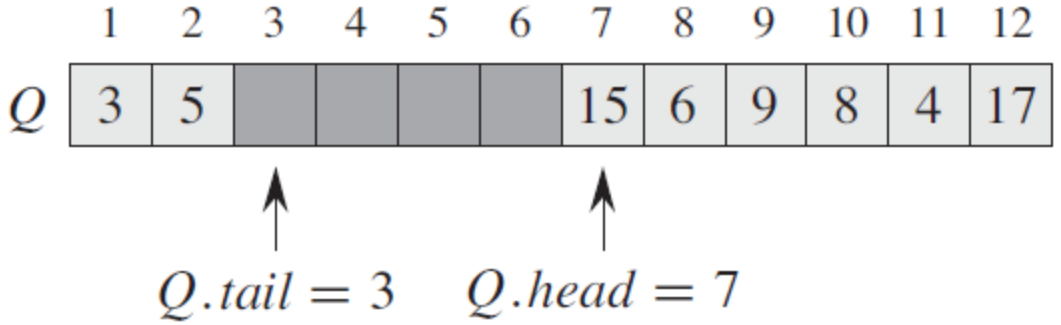
- Ensemble dynamique d'objets accessibles selon une discipline FIFO ("First-in first-out").
- Interface
  - Enqueue(Q, s) ajoute l'élément x à la fin de la file Q
  - Dequeue(Q) retire l'élément à la tête de la file Q
- Implémentation à l'aide d'un tableau circulaire
  - Q est un tableau de taille fixe Q. length
    - Mettre plus de Q.length éléments dans la file provoque une erreur de dépassement
  - Q.head est la position à la tête de la file
  - Q.tail est la première position vide à la fin de la file
  - Initialement : Q.head = Q.tail = 1

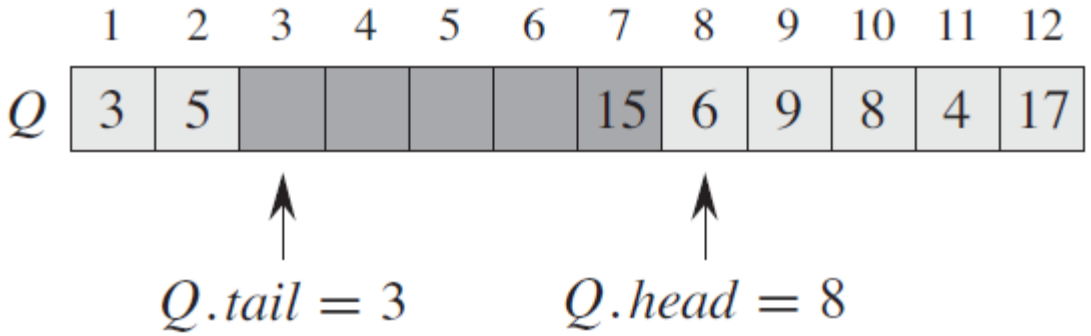
Les opérations: Enfiler et Défiler

Etat initial



Enqueue(Q, 17)
Enqueue(Q, 3)
Enqueue(Q, 5)





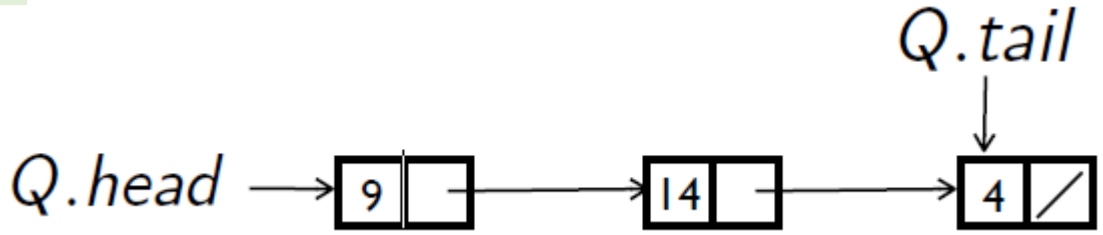
Dequeue(Q) -> 15

Implémentation à l'aide d'une liste liée

- Q est une liste simplement liée
- Q.head (resp. Q.tail) pointe vers la tête (resp. la queue) de la liste
- Complexité des opération enfiler et defiler en temps O(1), complexité en espace O(n) pour n opérations

```
Enqueue(Q,x)
1 x.next = NIL
2 if Q.head == NIL
3            Q.head = x
4 else Q.tail .next = x
5 Q.tail = x
```

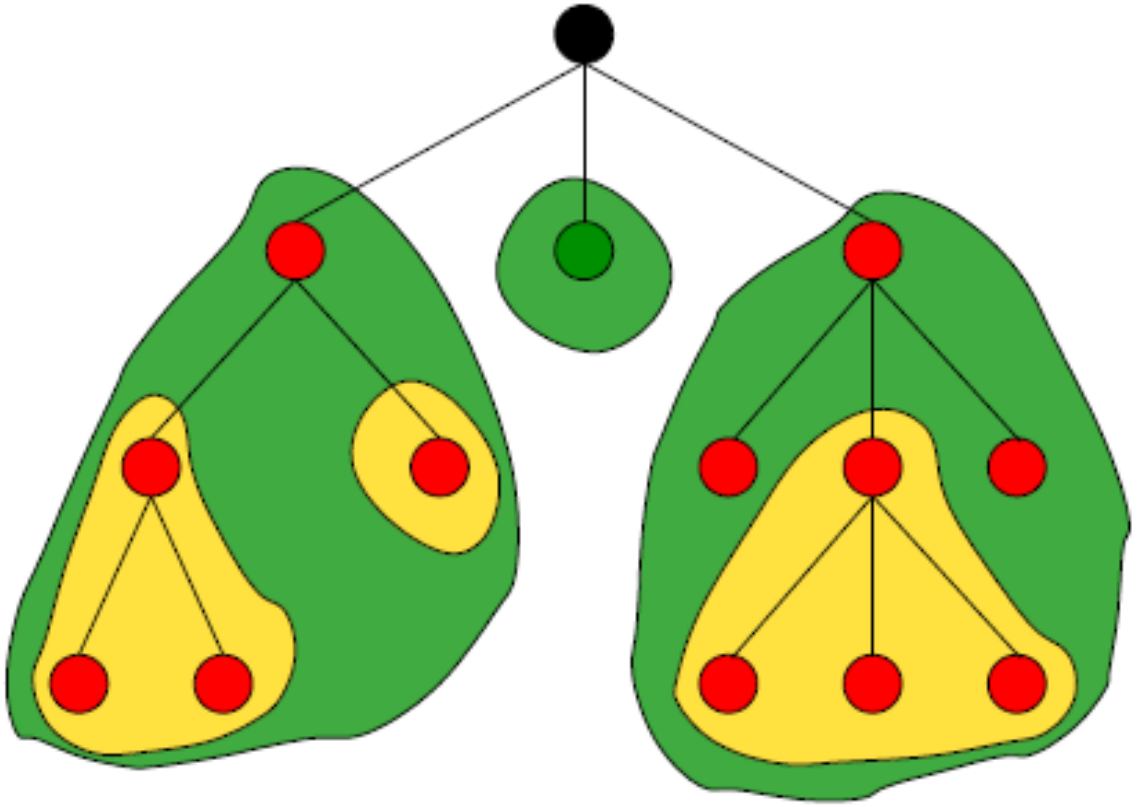
```
Dequeue(Q)
1 if Q.head == NIL
2            error "underflow"
3 x = Q.head
4 Q.head = Q.head.next
5 if Q.head == NIL
6            Q.tail = NIL
7 return x
```



Les arbres

Présentation

- C’est une Structure de données non linéaire, naturellement récursive, qui permet une organisation hiérarchique entre les données stockées
- Un arbre général est soit vide (Λ) soit de la forme $(v, \{B_1, \dots, B_n\})$ avec $B_1, \dots, B_n$ des arbres généraux et v une valeur.
- Un nœud qui contient une valeur est un nœud étiqueté.
- Un nœud qui n’a pas de sous-arbres est une feuille ou un nœud externe.

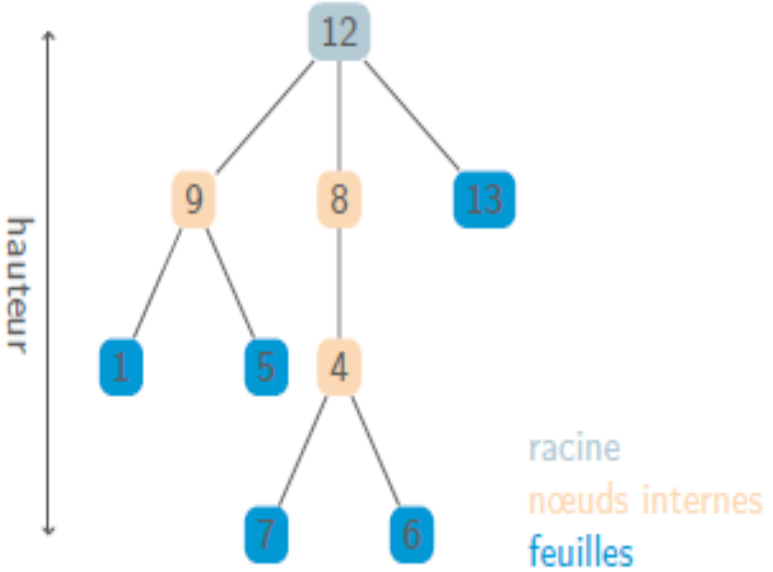


Domaine d’utilisation en informatique

- structure naturelle : arbre de décision, tournoi, généalogie, arborescence des répertoires, expressions arithmétiques, ...
- en algorithmique du texte (Huffamn, prefix-trees) : compression, recherche de motifs, détection de répétitions, ...
- en géométrie algorithmique (quadtrees, octrees, KD-trees, arbres rouge-noir, ...) ;
- en linguistique et en compilation (arbres syntaxiques) ;
- pour la gestion du cache, les bases de données, representation des documents XML, les systèmes de fichiers, ... (B-trees,).

Vocabulaire

- noeud : caractérisé par une donnée associée + un nombre fini de fils, possède un unique père
- feuille : noeud sans fils
- noeud interne : un noeud qui n'est pas une feuille
- Une forêt est un ensemble d'arbres.
- arité d'un noeud n : nombre de fils du noeud n
- arité d'un arbre a : nombre maximal de fils d'un noeud de a
- racine d'un arbre a : c'est le seul noeud sans père
- Un chemin est une suite de nœuds consécutifs.
- Une branche est un chemin de la racine à une feuille.
- profondeur d'un noeud *n* : nombre de noeuds sur la branche entre la racine et le noeud *n* exclu
- hauteur d'un arbre *a* : c'est le nombre de noeuds sur la branche qui va de la racine de *a* à la feuille de profondeur maximale

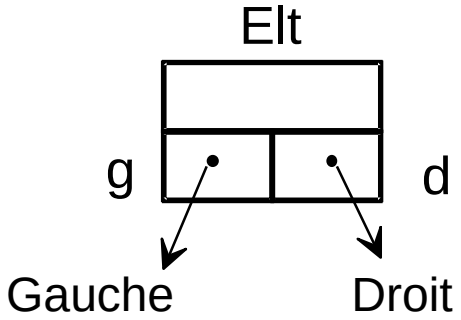


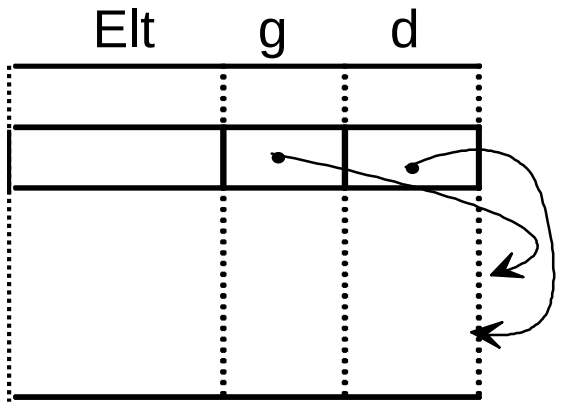
Arbres binaires

- **Définition informelle**
  - Dans un arbre binaire tout noeud a au plus deux fils
  - Un arbre binaire possède exactement deux sous-arbres (éventuellement vides)
- **Définition récursive** (et constructive). Un arbre binaire est
  - Soit vide
  - Soit composé d'une racine r de 2 sous arbres binaires G et D disjoints
    - G : sous Arbre Binaire Gauche
    - D : sous Arbre Binaire Droit
- Nœuds $(A) = \{r\} \cup$ Nœuds $(G) \cup$ Nœuds (D)

Type arbre binaire

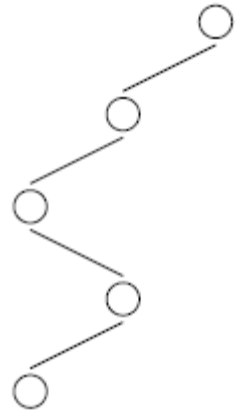
- Opérations
  - Arbre-vide : → arbre
  - Racine : arbre → nœud
  - Gauche, Droit : arbre → arbre
  - Cons : nœud x arbre x arbre → arbre
  - Elt : nœud → élément
  - Vide : arbre → booléen
- Implémentations par pointeurs ou curseurs

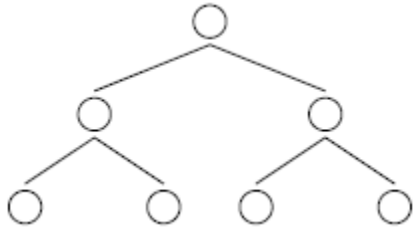


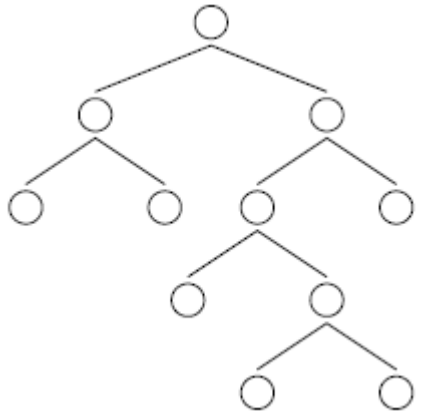


Arbres binaires particuliers

- Un arbre binaire est dégénéré si tout noeud n'a au plus qu'un fils.
- Un arbre binaire est complet si chaque niveau est complètement rempli.
- Un arbre binaire est localement complet si tout nœud interne a exactement deux fils.

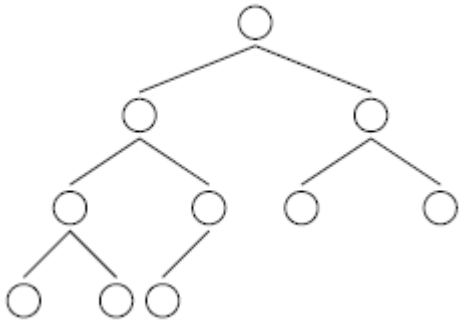


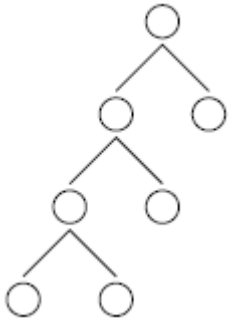




Arbres binaires particuliers

- Un arbre binaire est parfait si tous les niveaux sont complètement remplis sauf le dernier mais tous les noeuds du dernier niveau sont groupés à gauche.
- Un peigne à gauche (resp. droite) est un arbre binaire localement complet dont tous les fils droits (resp. gauches) sont des feuilles.



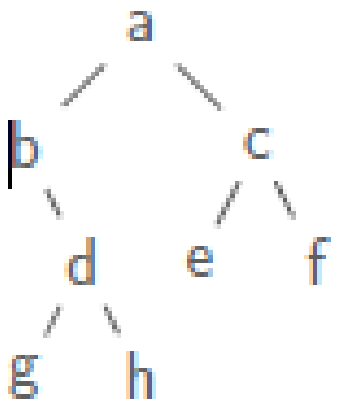


Propriétés des arbres binaires

- Pour tout arbre binaire non vide ayant n noeuds et une hauteur h, $\lfloor log_2 n \rfloor \leq h \leq n - 1$
- Pour tout arbre binaire non vide ayant n noeuds et f feuilles, $f \leq \frac{n+1}{2}$
- Tout arbre binaire localement complet ayant n noeuds internes a (n+1) feuilles.
- La hauteur d'un arbre binaire localement complet ayant n feuilles est $\lceil log_2 n \rceil$

Parcours en profondeur

- On traite récursivement les nœuds de l'arbre.
- Trois sens de parcours :
  - **préfixe** : traiter la racine, puis le sous-arbre gauche, puis le sous-arbre droit ;
  - **postfixé** : traiter le sous-arbre gauche, puis le sous-arbre droit, puis la racine ;
  - **infixé** : traiter le sous-arbre gauche, puis la racine, puis le sous-arbre droit.



Infixé:	b, g, d, h, a, e, c, f
Postfixé:	g, h, d, b, e, f, c, a
préfixé:	a, b, d, g, h, c, e, f

Parcours en profondeur (version récursive)

```
procedure AffichagePrefixe(a)
    si a n'est pas vide alors
        (* traitement de la racine *)
        afficher l'étiquette de la racine
        (* traitement des fils gauche et droit *)
        AffichagePrefixe(Gauche(a))
        AffichagePrefixe(Droit(a))
    fin si
fin procedure
```

Note : pour changer le type de parcours il suffit d'échanger l'ordre des trois instructions du if.

Comment dérécursiver le parcours ?

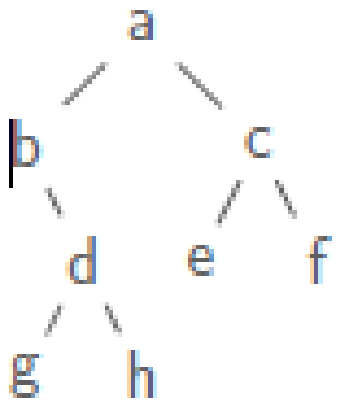
Parcours en profondeur (version itérative)

```
procedure AffichagePrefixe(a)
    soit p une pile d'arbres
    Empiler(a, p)
    Tant que p n'est pas vide faire
        (* on traite l'arbre au sommet de la pile *)
        s = sommet(p)
        Dépiler(p)
        si s n'est pas vide alors
            afficher l'étiquette de la racine de s
            Empiler(Droit(s),p)
            Empiler(Gauche(s), p)
        fin si
    fin tant que
Fin procedure
```

Il ne faut pas se tromper et bien empiler le sous-arbre droit avant le sous-arbre gauche car la pile est une structure LIFO.

Parcours en largeur (itératif)

On traite les noeuds, des moins profonds aux plus profonds, par strates.



a,b,c,d,e,f,g,
h

```
procedure AffichageEnLargeur(a)
    soit f une file d'arbres
    Enfiler(a, f)
    tant que f n'est pas vide faire
        s = Défiler(f)
        si s n'est pas vide alors
            afficher la racine de s
            Enfiler(Gauche(s), f)
            Enfiler(Droit(s), f)
        fin si
    fin tant que
fin procedure
```

Opérations de mesure

Calcul de la taille d'un arbre binaire

```
procedure Taille(a)
    si a est vide alors
        renvoyer 0
    sinon
        tg = Taille(Gauche(a))
        td = Taille(Droit(a))
        renvoyer 1 + tg + td
    fin si
fin procedure
```

Calcul de la hauteur d'un arbre binaire

```
procedure Hauteur(a)
    si a est vide alors
        renvoyer −1
    sinon
        hg = Hauteur(Gauche(a))
        hd = Hauteur(Droit(a))
        renvoyer 1 + max(hg , hd )
    fin si
fin procedure
```

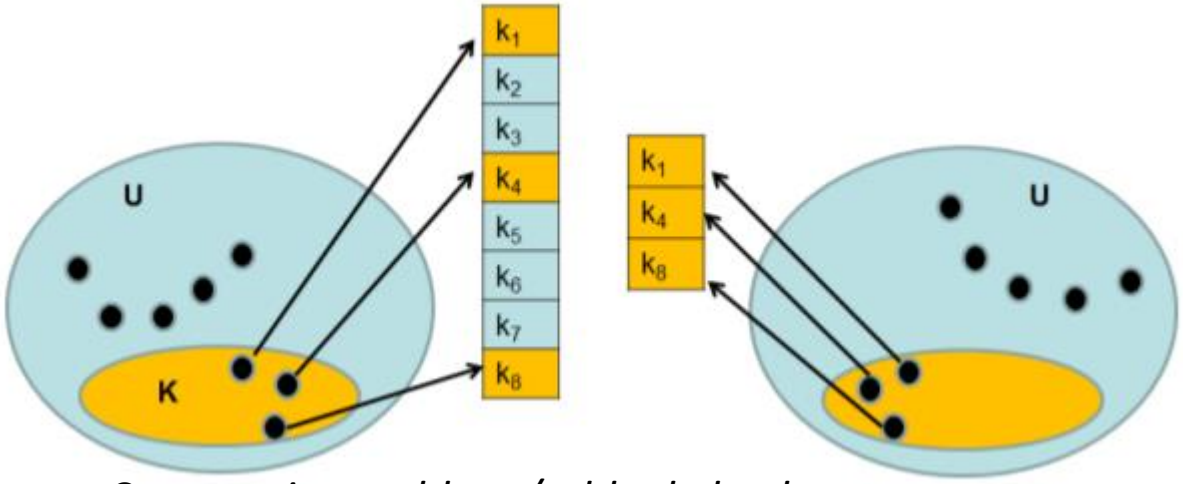
Table de hachage

Position du problème

- Soit U l'ensemble des valeurs possibles et soit K l'ensemble des valeurs effectivement utilisées.
- Une table de hachage est une table (tableau) dont :
  - Les éléments sont placées dans des cases.
  - Le choix de la case est effectué en appliquant une fonction sur une valeur : c'est la *fonction de hachage*.

Remarque

- Si U est grand, la taille du tableau est problématique.
- Si K est petit, il y a gaspillage de ressources



Comparaison tableau/table de hachage

Définitions

- Une fonction de hachage h établit une relation entre U (univers des valeurs) et un sous-ensemble fini : $h: U \rightarrow \{0,1, \ldots, m-1\}$
  - h(k) est appelé valeur de hachage de k.
  - On place notre élément dans la case h(k).
  - *h est non injective en général*
- **Bonne nouvelle** : on doit gérer m valeurs au lieu de |U|.
- **Mauvaise nouvelle** : les collisions, plusieurs valeurs peuvent avoir une même valeur de hachage.
  - Résolution des collisions: hachage ouvert (chaînage) ou hachage fermé (adressage ouvert)
- Si chaque élément a la même chance d'être haché dans n'importe laquelle des alvéoles, indépendamment des éléments précédents, la fonction de hachage est dite *uniforme simple*.

Exemples de fonction de hachage

$$h(id) = \left(\sum_{c_i \in id} c_i \right) \ mod \ m$$

```
fonction h (x : mot) : indice //fonction de hachage
début
        somme : = 0 ;
        pour i ← 1 à longueurmaxi faire
              somme ← somme + ord (x [i]) ;
        retour (somme mod B) ;
fin
```

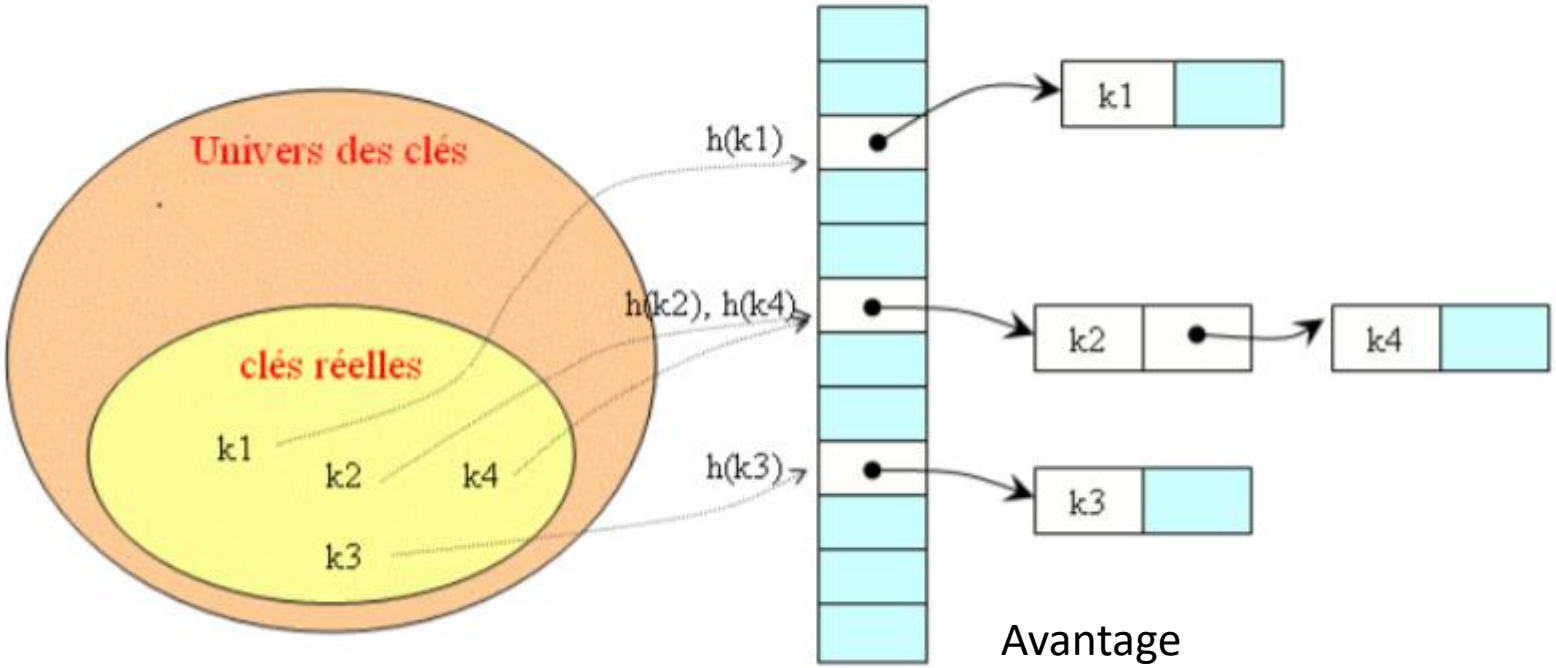
$$h = \left(\sum_{i=1}^{n} \alpha^{n-i} c_i \right) \ mod \ \ m$$

```
#define     SIZE  …//un nombre premier
#define     SHIFT 4 //une puissance de 2

int hash( char *key) {
    int temp = 0 ;
    int i = 0 ;
    while (key[ i ] != '\0') {
        temp = ((temp << SHIFT) + key[ i ]) ;
        ++i;
    }
    return temp % SIZE;
}
```

Résolution des collisions par chainage

Avec cette technique, on place dans une liste chaînée tous les éléments ayant la même valeur de hachage



Pour rechercher un élément, parcourir la liste chainée dont l'accès se trouve dans la case d'indice h(x).

Avantage

- Implémentation aisée
- Pas de limitation (théorique) au nombre de clés.

=> Toutefois, il faut veiller à prendre un tableau assez grand pour que la taille des listes reste réduite, le but étant de minimiser les parcours.

Interface

- Implémente le Type abstrait « dictionnaire »
- Opérations principales
  - inserer(x) : insertion d'un élément x en tête de la liste chaînée h(x.valeur)
  - rechercher(k) : recherche d'un élément de valeur k dans la liste chaînée h(k)
  - supprimer(x) : suppression d'un élément x dans la liste chaînée h(x.valeur)
- Complexité
  - ***Hypothèse:*** opération de hachage = O(1)
  - insertion est en O(1)
  - recherche et de suppression sont indéfinies car dépend de la longueur des listes
    - On introduit la notion de ***facteur de remplissage***.

Facteur de remplissage

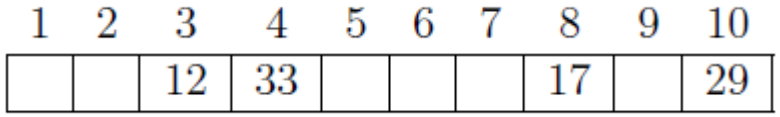
- Donnée: Table de hachage T avec m cases et n éléments
- Facteur de remplissage: $\alpha = n/m$
- Comportement au pire cas = Θ(n), si tous les objets sont dans la même case.
- Complexité en moyenne
  - Insertion :$\Theta(1)$
  - Recherche : $\Theta(1 + \alpha)$
  - Suppression : $\Theta(1 + \alpha)$

Adressage ouvert

- **L’idée** est la suivante : lorsqu'on veut utiliser un emplacement de la table et que celui-ci est occupé, on va voir ailleurs.
- Dans la technique la plus simple, on va simplement continuer le parcours du tableau, à partir de la position de départ, à la recherche d’un « trou ».
- Pour la recherche, on procède de même : *la fonction de hachage nous indique où commencer à chercher* et on poursuit jusqu’à trouver l'élément ou un « trou ».

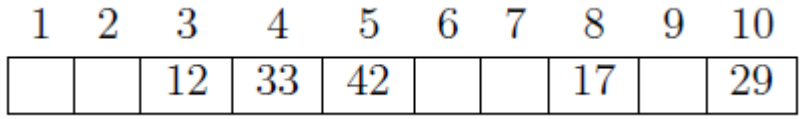
Exemple

Configuration initiale, on veut insérer la clé 42



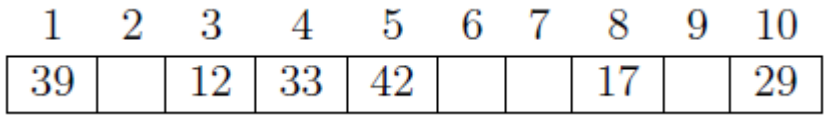
1	2	3	4	5	6	7	8	9	10
		12	33				17		29

la clé 42 devrait occuper la case d'indice 3. Comme celle-ci est occupée, on parcourt le tableau jusqu'à la prochaine case libre, i.e 5



1	2	3	4	5	6	7	8	9	10
		12	33	42			17		29

Ce parcours de recherche est circulaire: lorsqu'on arrive à la dernière case, on revient au début.
Ainsi, la clé 39 qui ne peut pas être mise à la position 10 (occupée par 29) sera placée en première position



1	2	3	4	5	6	7	8	9	10
39		12	33	42			17		29

La **suppression** est plus délicate à mettre en œuvre: s'il y a eu une collision, on ne retrouverait plus l'autre valeur de clé. Il faut donc reboucher le trou avec la dernière clé donnant la même valeur de hachage.

A retenir..

- Les tables de hachage ont de très bonnes propriétés :
  - **Rapidité** : probablement la structure la plus rapide pour l'insertion quand le nombre de données est grand
  - **Efficacité** : nul besoin d'ajouter des informations ou d'envelopper les données pour l'adressage ouvert.
- Mais elles souffrent de limitations :
  - Elles permettent juste de savoir si un élément est présent, pas de rechercher le plus petit.
  - Il faut prévoir la taille de la table avant exécution, contrairement aux autres structures dynamiques.
  - Toutes leurs propriétés de complexité dépendent du bon fonctionnement de la fonction de hachage.