Algorithmique Avancée

Mounir T. El Araki

mounir.elarakitantaoui@uic.ac.ma

CPI 1

Motivation

- Représentation des ensembles
- Un contenu rarement statique, besoin de le mettre à jour périodiquement
- Plusieurs possibilités de représentation
  - Aucune n'est meilleure dans l'absolu
  - En prenant en compte un besoin de traitement, une représentation peut être meilleure qu'une autre

Opérations élémentaires

- Généralement, ce type de structures, on définit les opérations suivantes:

  - **RECHERCHE(S, k)** : étant donné un ensemble S et une clé k, le résultat de cette requête est un pointeur sur un élément de S de clé k, s'il en existe un, et la valeur NULL s'il n'appartient pas à S.
  - **INSERTION(S, x)** : ajoute à l'ensemble S l'élément pointé par x. (tout les champs de l'élément de x sont supposé initialisé)
  - **SUPPRESSION(S, x)** : supprime de l'ensemble S son élément pointé par x

Opérations élémentaires sur les ensemble ordonné

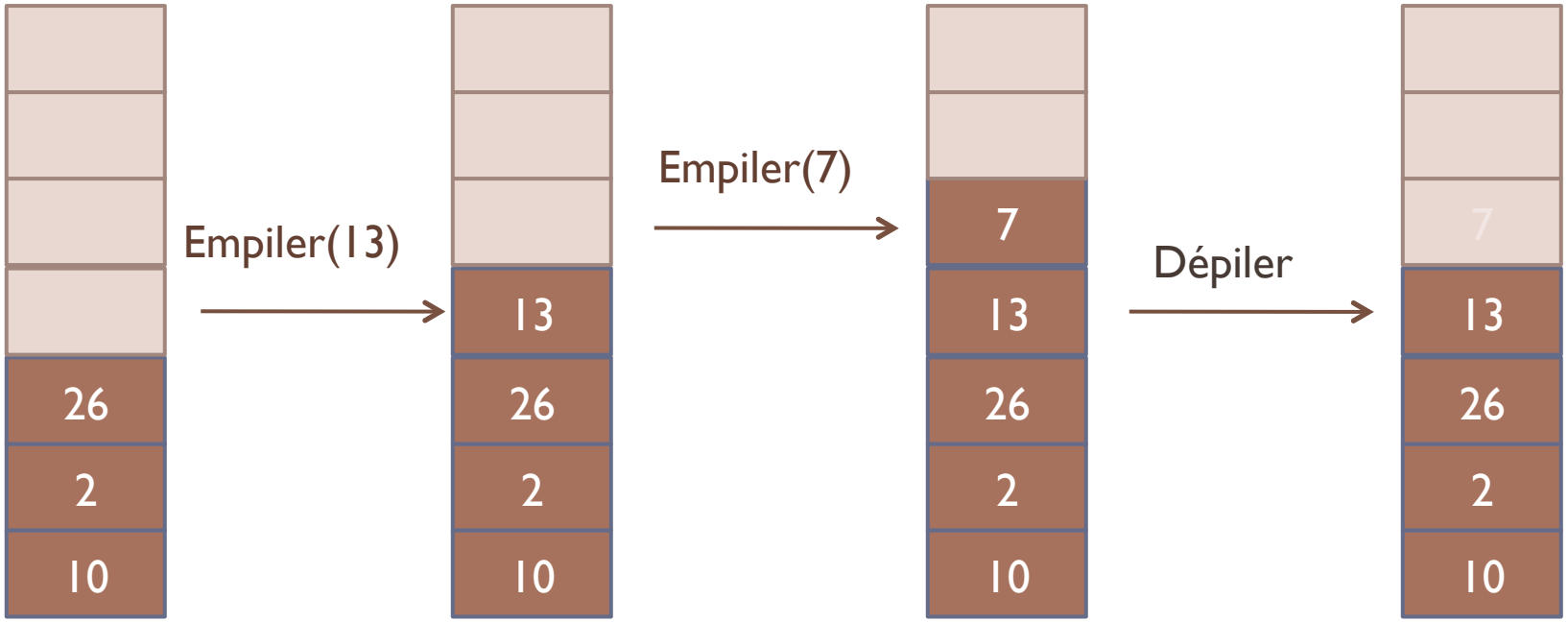
- Si l'ensemble est totalement ordonné, d'autres opérations sont possibles :

  - **MINIMUM(*S*)** : *renvoie l'élément de S de clé minimale.*
  - **MAXIMUM(*S*)** : *renvoie l'élément de S de clé maximale.*
  - **SUCCESSEUR(*S, x*)** : *renvoie, si celui-ci existe, l'élément de S immédiatement plus grand que l'élément de S* pointé par *x, et NULL dans le cas contraire.*
  - **PRÉDÉCESSEUR(*S, x*)** : *renvoie, si celui-ci existe, l'élément de S immédiatement plus petit que l'élément de S* pointé par *x, et NULL dans le cas contraire.*

Les piles

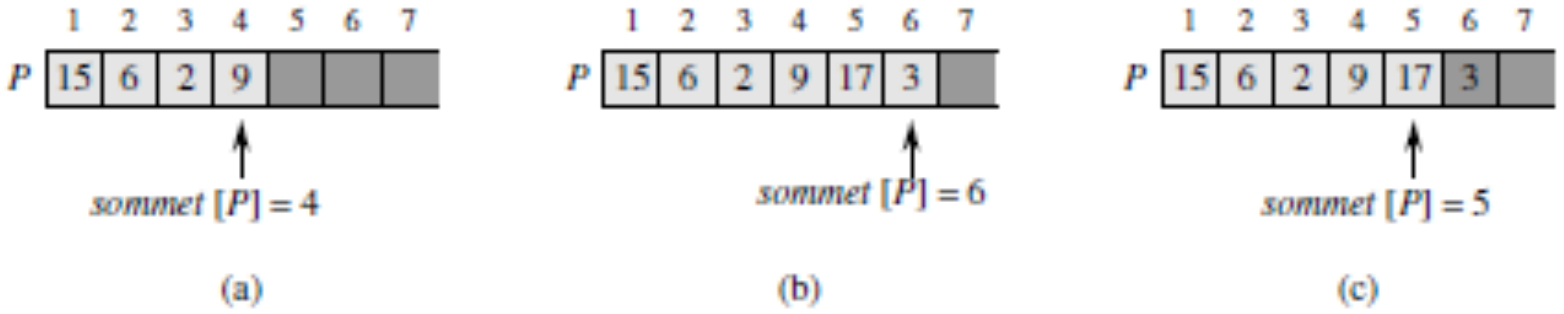
- *Une pile est une structure de données mettant en œuvre le principe*
  - *LIFO : Last-In, First-Out : dernier entré, premier sorti*
  - L'opération **suppression** est spécifiée par définition sur le dernier élément. Cette opération ne prend alors que l'ensemble comme argument.
  - L'opération **insertion** est aussi spécifiée par définition sur le dernier élément. Elle prend comme argument l'élément à insérer
  - Dans une pile, l'opération suppression est communément appelée **dépiler** et l'opération insertion est appelée **empiler**

Exemple


Empiler(13)
Empiler(7)
Dépiler
26
2
10
13
26
2
10
7
13
26
2
10
7
13
26
2
10

Utilisation de tableaux

- Une pile est facilement implémentée par un tableau
- Quelques points à prendre en compte lors de l'implémentation
  - Opération dépiler quand la pile est vide
  - Opération empiler quand la pile est pleine
- Représentation: un tableau plus un pointeur sur la dernière case non vide (appelée sommet)

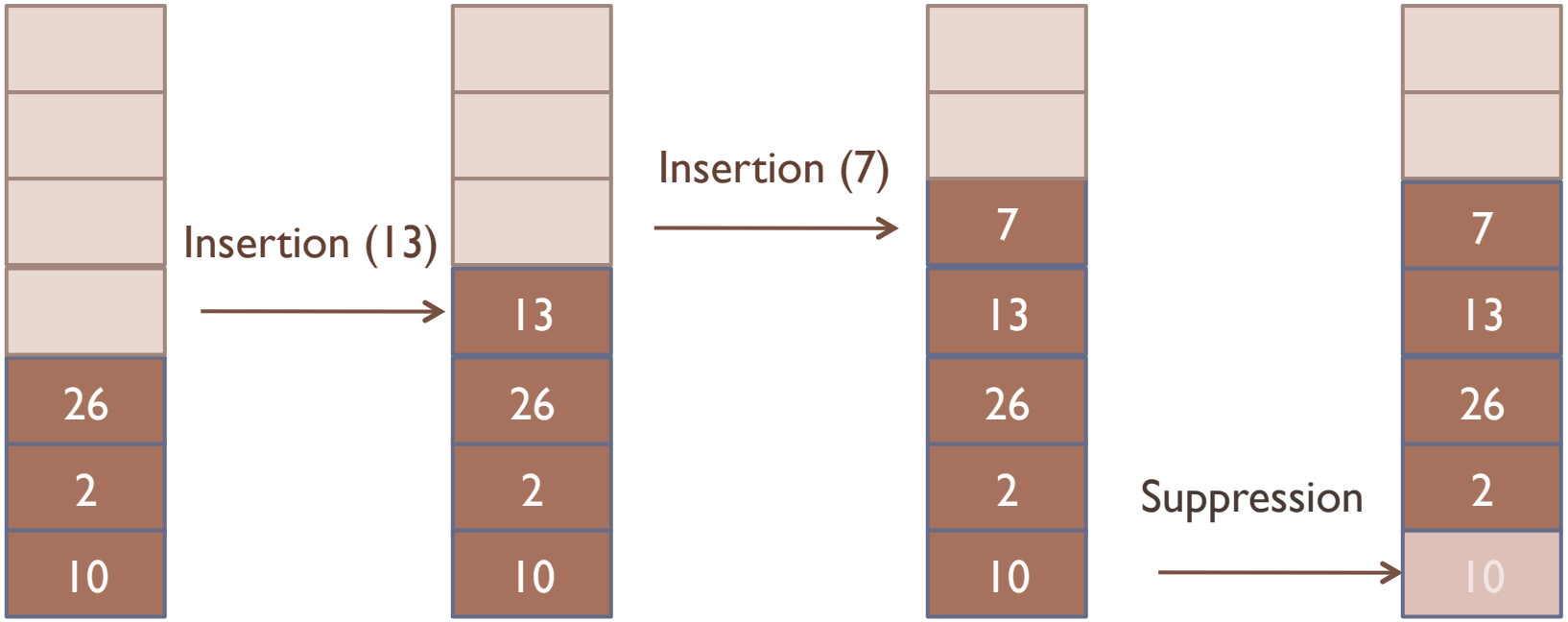


Algorithmes de traitement

- PILE-VIDE(*P*)
  1 **si** *sommet(P)=0* ***alors retourner*** *VRAI*
  2 **sinon retourner** FAUX
- EMPILER(*P, x*)
  1 **si** *sommet(P) = longueur(P)* ***alors erreur*** *« débordement positif »*
  2 **sinon** *sommet(P)* ← *sommet(P)+1 ;*
  3 *P[sommet(P)]* ← *x ;*
- DÉPILER(*P*)
  1 **si** PILE-VIDE(*P*) ***alors erreur*** *« débordement négatif »*
  2 **sinon** *sommet(P)* ← *sommet(P)-1*
  3 **retourner** *P[sommet(P)+1]*

Les files

- *Une file est une structure de données mettant en œuvre le*
  - *FIFO : First-In, First-Out : premier entré, premier sorti*

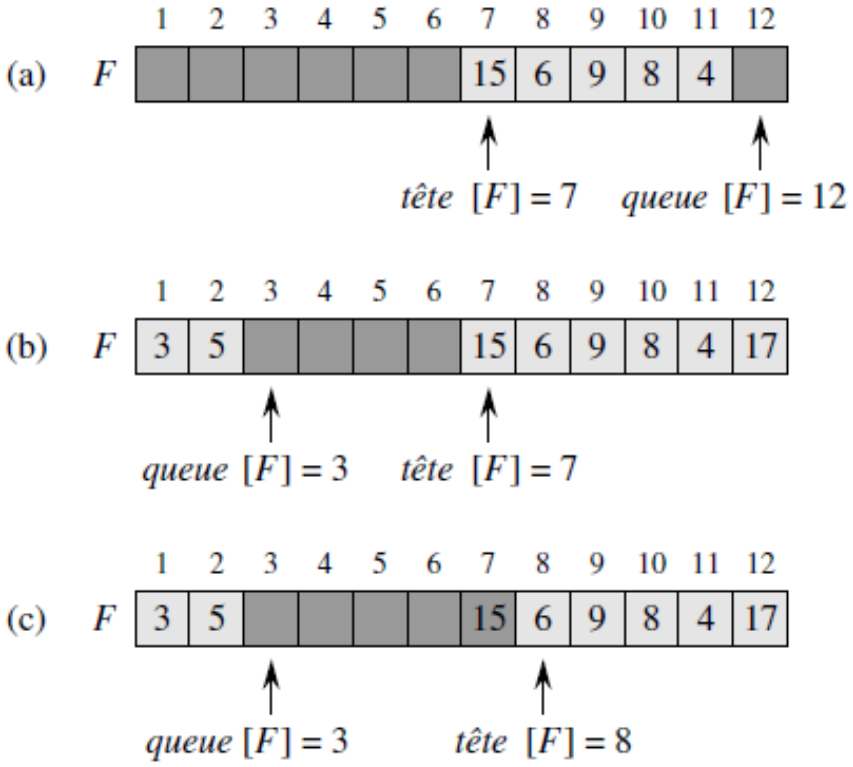


Utilisation de tableaux

- Une file est aussi facilement implémentée par un tableau
- Les mêmes cas de figure que les piles se présentent
  - Opération défiler quand la file est vide
  - Opération enfiler quand la file est pleine
- Représentation: un tableau plus deux pointeurs tête et queue

Les pointeurs tête et queue

- ***tête(F)**: indexe (ou pointe) vers la tête de la file*
- ***queue(F)** qui indexe le dernier élément de la file + 1*
- ***PLEINE** spécifie que la file est pleine (initialisé à faux)*
- ***TAILLE** donne la taille max de la file (fixé) → Longueur*
- Les éléments de la file se trouvent donc aux emplacements *tête(F), tête(F) + 1, ..., queue(F)-1 (modulo n).*



Algorithmes pour les files

- **FILE-VIDE(*F*)**

```
1   si (tête(F)=queue(F) Et non(PLEINE))
alors retourner VRAI
2   sinon retourner FAUX
```

- **ENFILER(*F, x*)**

```
1   si PLEINE alors erreur « débordement
positif »
2   sinon F[queue(F)] ← x
3   si queue(F)=longueur(F) alors
4       queue(F) ← 1
5   sinon
6       queue(F) ← (queue(F)+1)
7   PLEINE ← (queue(F)==tête(F))
```

- **DEFILER(*F*)**

```
1   si FILE-VIDE(F) alors erreur «
débordement négatif »
2   sinon  PLEINE ← FAUX
3   x ← F[tête[F]]
4   Si tête(F)=longueur(F) alors
5       tête(F) ← 1
6   Sinon
7       tête(F) ← (tête(F)+1)
8   Retourner x
```

Listes chainées

- Une liste chaînée est une structure de données dans laquelle les objets sont arrangés linéairement, l'ordre linéaire étant déterminé par des pointeurs sur les éléments

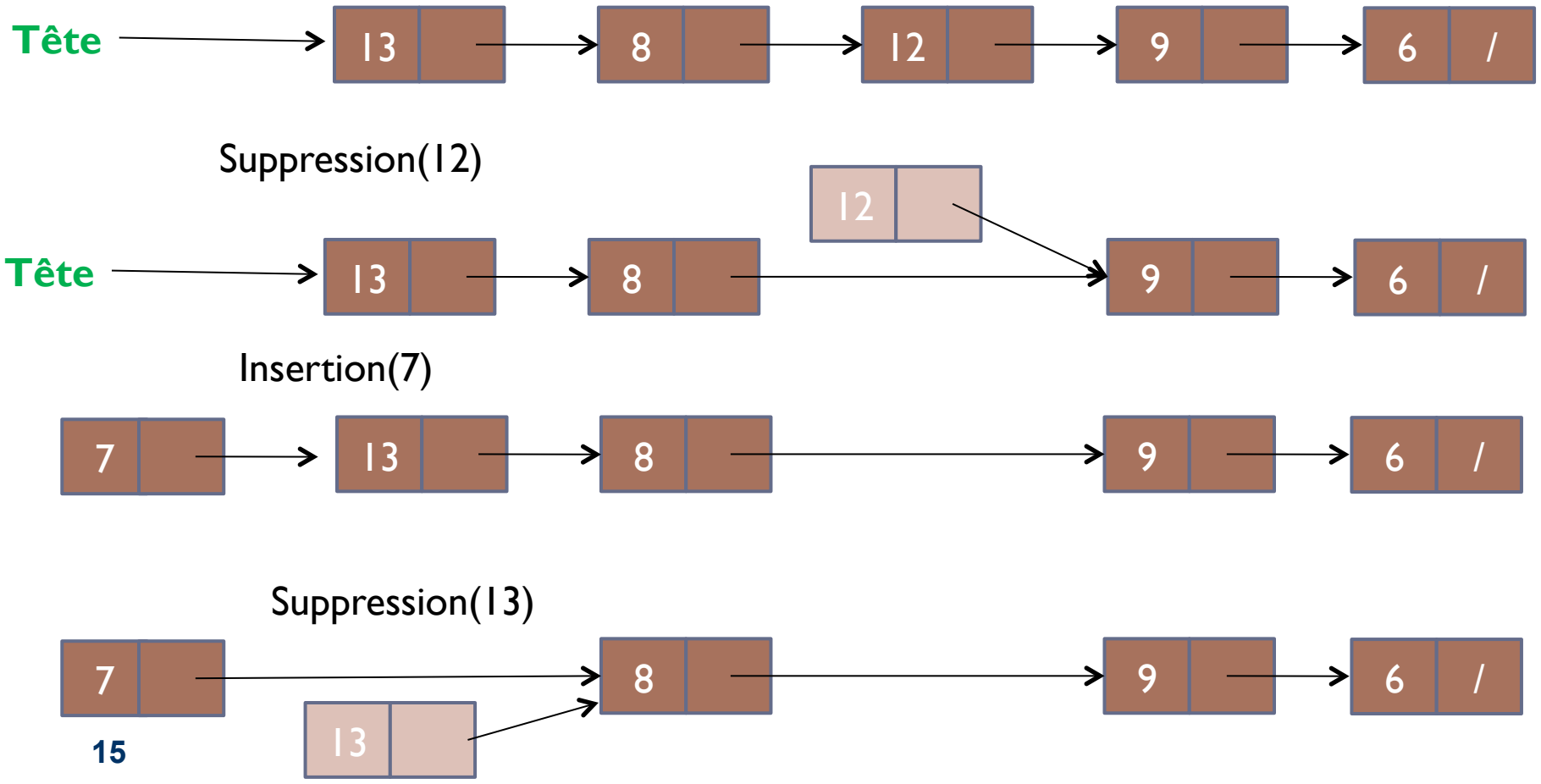
- Chaque élément de la liste, outre le **champ clé**, contient un **champ successeur** qui est pointeur sur l'élément suivant dans la liste chaînée.

- Si le champ successeur d'un élément vaut **NULL,** cet élément n'a pas de successeur et est donc le dernier élément de la liste chaînée

Manipulation de listes chaînées

- Le premier élément de la liste est appelé la **tête** de la liste
- Le dernier élément est appelé la **queue** de la liste
- Une liste *L est* manipulée via un pointeur vers son premier élément, que l'on notera **TÊTE(*L*)**
- *Si la liste est vide,* ***TÊTE(L)*** *vaut NULL*

Exemple


Tête
13
8
12
9
6
/
Suppression(12)
12
Tête
13
8
9
6
/
Insertion(7)
7
13
8
9
6
/
Suppression(13)
7
8
9
6
/
13

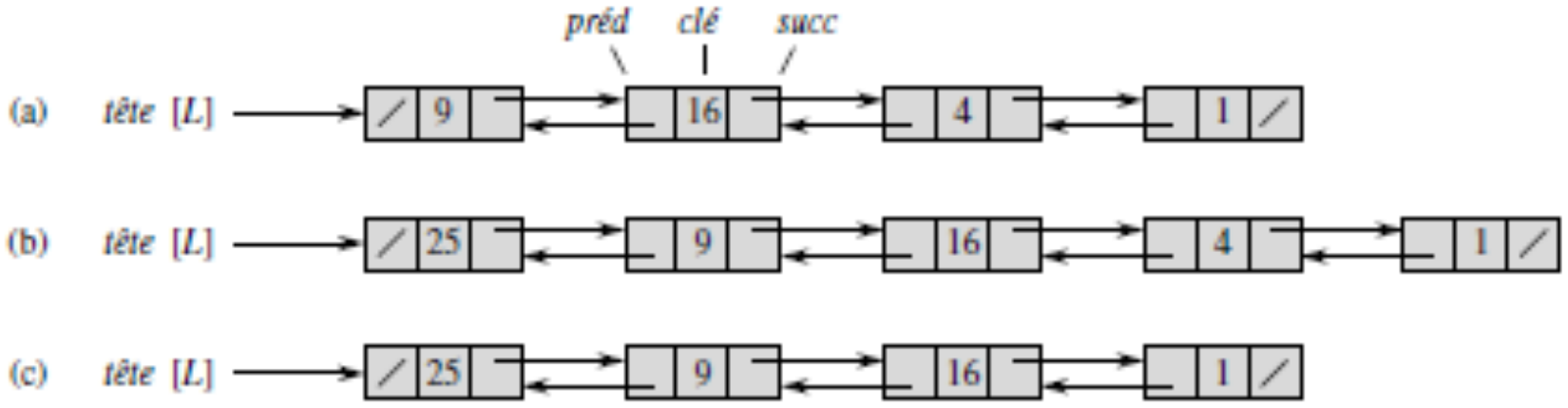
Types des listes chaînées

- Simplement chaînée: liste comprenant le seul pointeur successeur

- doublement chaînée :
  - en plus du champ ***successeur***, *chaque élément contient un champ* ***prédécesseur*** qui est un pointeur sur l'élément précédant dans la liste.

  - Si le champ ***prédécesseur*** *d'un élément vaut NULL, cet* élément n'a pas de prédécesseur et est donc la tête de la liste.

Exemple


prédécesseur
clé
successeur
(a) tête [L]
9
16
4
1
(b) tête [L]
25
9
16
4
1
(c) tête [L]
25
9
16
1

Types de listes chaînées

- **Triée ou non triée** : suivant que l'ordre linéaire des éléments dans la liste correspond ou non à l'ordre linéaire du contenu de ces éléments.

- **Circulaire** : si le champ prédécesseur de la tête de la liste pointe sur la queue, et si le champ successeur de la queue pointe sur la tête. La liste est alors vue comme un anneau.

Algorithme de **recherche**

- L'algorithme RECHERCHE-LISTE(L, k) :
  - trouve le premier élément de clé k dans la liste L par une simple recherche linéaire, et retourne un pointeur sur cet élément.
  - Si la liste ne contient aucun objet de clé k, l'algorithme renvoie NULL.

- RECHERCHE-LISTE(L, k)

```
1    x←TÊTE(L)
2    tant que x<> NULL et clé[x]<> k faire
3            x ← succ[x]
4    retourner x
```

- L'algorithme marche pour les listes simplement et doublement chaînées

Algorithme **d'insertion**

- Étant donné un élément x et une liste L, l'algorithme INSERTION-LISTE insère x en tête de L.

- INSERTION-LISTE(L, x)

```
1     succ[x]← TÊTE(L)
2     si TÊTE(L) <> NULL alors
3             préd[TÊTE(L)]← x
4     TÊTE(L)← x
5     préd[x]← NULL
```

- L'algorithme traite les listes doublement chaînées
- Dans le cas des listes simplement chaînées, on supprime les instructions avec 'prédécesseurs'

Algorithme de **suppression** (doublement chaînées)

- L'algorithme SUPPRESSION-LISTE:
  - élimine un élément *x d'une liste chaînée L.*
  - *Cet algorithme a besoin d'un pointeur* sur l'élément *x à supprimer. Si on ne possède que la clé de cet élément, il faut préalablement utiliser l'algorithme* RECHERCHE-LISTE pour obtenir le pointeur nécessaire.

- SUPPRESSION-LISTE(*L, x)*

```
1     si préd[x] <> NULL alors
2           succ[préd[x]] ←succ[x]
3     sinon TÊTE(L) ← succ[x]
4     si succ[x] <> NULL alors
5           préd[succ[x]] ← préd[x]
```

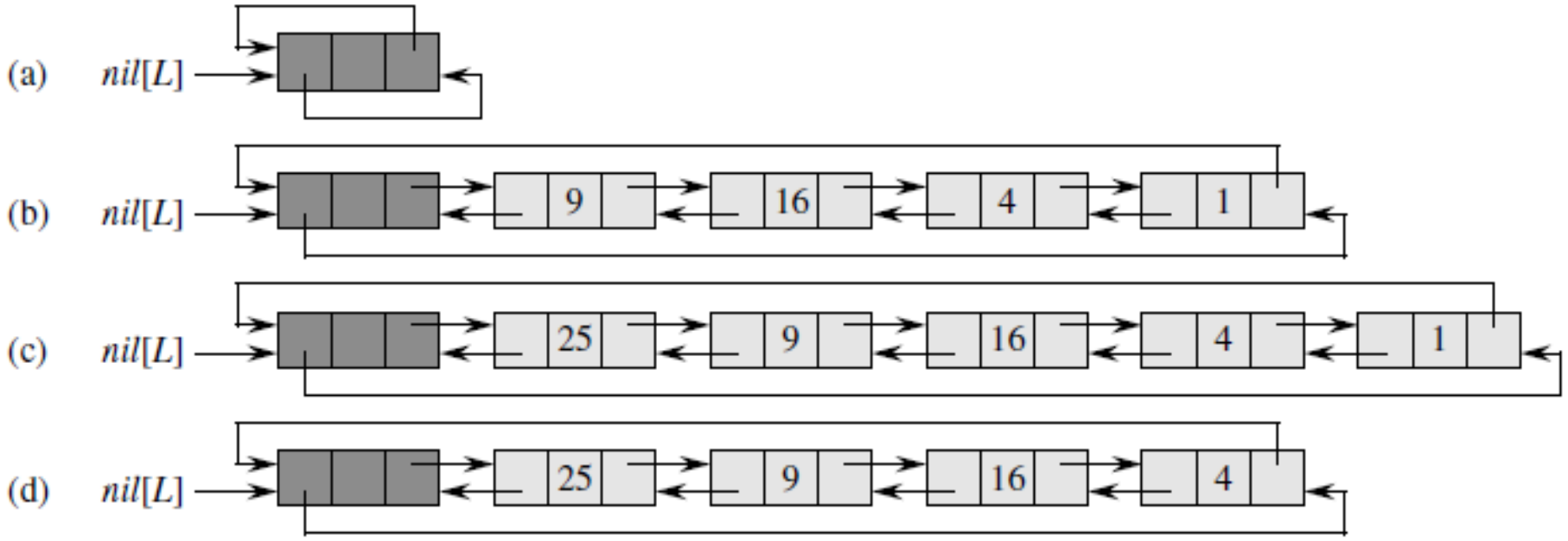
Algorithme de **suppression** (simplement chaînées)

- L'algorithme pour les listes simplement chaînées est plus complexe
  - Nous n'avons pas de moyen simple de récupérer un pointeur sur l'élément qui précède celui à supprimer

- SUPPRESSION-LISTE(L, x)

```
1    si x = TÊTE(L)  alors
2            TÊTE(L) ← succ[x]
3    sinon y ← TÊTE(L)
4    tant que succ[y]<>x faire
5            y← succ[y]
6    succ[y]← succ[x]
```

Utilisation de sentinelle



Liste circulaire doublement chaînée, avec sentinelle. La sentinelle *nil*[*L*] apparaît entre la tête et la queue. L' attribut *tête*[*L*] n'est plus nécessaire, car on peut accéder à la tête de la liste *via succ*[*nil*[*L*]]. (a) Une liste vide. (b) La liste chaînée de la figure (a), avec la clé 9 en tête et la clé 1 en queue. (c) La liste après exécution de INSÉRER-LISTE′(*L*, *x*), où *clé*[*x*] = 25. Le nouvel objet devient la tête de la liste. (d) La liste après suppression de l'objet ayant la clé 1. La nouvelle queue est l'objet ayant la clé 4.

Liste chainées avec sentinelles

- RECHERCHE-LISTE(L, k)
- 1 $x \leftarrow succ[NIL[L]]$
- 2 **tant que** x **<>** $NIL[L]$ et $clé[x]$ **<>** k
- 3 **faire** $x \leftarrow succ[x]$
- 4 **retourner** x

- INSÉRER-LISTE(L, x)
- 1 $succ[x] \leftarrow succ[NIL[L]]$
- 2 $préd[succ[NIL[L]]] \leftarrow x$
- 3 $succ[NIL[L]] \leftarrow x$
- 4 $préd[x] \leftarrow NIL[L]$

- SUPPRIMER-LISTE(L, x)
- 1 $succ[préd[x]] \leftarrow succ[x]$
- 2 $préd[succ[x]] \leftarrow préd[x]$

Comparaison de complexité: l'opération accéder/rechercher

- Rechercher (L,K), dans le pire cas
  - Pour les listes:
    - O(n): il faut parcourir toute la liste des successeurs pour récupérer l'élément
  - Pour les tableaux
    - O(1): l'élément est directement accessible par son indice

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau trié	Tableau non trié
Acceder(L,k)	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)
Rechercher(L,k)	O(n)	O(n)	O(n)	O(n)	O(log(n))	O(n)

Comparaison de complexité: l'opération insertion

- Insertion (L,x), dans le pire cas
  - Pour les listes non triées:
    - O(1): il suffit de l'insérer au début
  - Pour les listes triées
    - O(n), pire cas quand il faut l'insérer en dernier
  - Pour les tableaux non triés
    - O(n): erreur ou créer un nouveau tableau de taille supérieure
  - Pour les tableaux triés
    - O(n): en plus il faut décaler les suivants (pire cas, un élément k plus petit que tous les éléments du tableau)

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau trié	Tableau non trié
Insertion(L,x)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)

Comparaison de complexité: l'opération suppression

- Suppression (L,x), dans le pire cas
  - Pour les listes simples:
    - O(n): il faut chercher le prédécesseur
  - Pour les listes doubles
    - O(1)
  - Pour les tableaux
    - O(n): il faut décaler les suivants (pire cas, supprimer le premier élément)

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau trié	Tableau non trié
Suppression(L,x)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)

Comparaison de complexité: l'opération successeur

- Successeur (L,x), dans le pire cas
  - Pour les listes ou les tableaux non triés:
    - O(n): il faut parcourir toute la liste si c'est le dernier
  - Pour les listes ou les tableaux triés
    - O(1): c'est l'élément d'indice i+1 pour les tableaux et c'est le successeur pour les listes

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Successeur(L,x)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)

Comparaison de complexité: l'opération prédécesseur

- Prédécesseur (L,x), dans le pire cas
  - Pour les listes non triées
    - O(n): il faut parcourir toute la liste
  - Pour les listes simples triée
    - O(n): il faut parcourir la liste pour trouver le prédécesseur
  - Pour les listes doubles triées
    - O(1)
  - Pour les tableaux non triés
    - O(n): parcourir tout le tableau
  - Pour les tableaux triés
    - O(1), c'est l'indice i-1

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Prédécesseur(L,x)	O(n)	O(n)	O(n)	O(1)	O(n)	O(1)

Comparaison de complexité: l'opération minimum

- Minimum (L), dans le pire cas
  - Pour les listes et les tableaux non triés
    - O(n): il faut parcourir toute la liste
  - Pour les listes et les tableaux triés
    - O(1): C'est la tête pour les listes et l'indice 0 (ou 1) pour les tableaux

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Minimum(L)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)

Comparaison de complexité: l'opération maximum

- Maximum (L), dans le pire cas
  - Pour les listes et les tableaux non triés:
    - O(n): il faut parcourir toute la liste si c'est le dernier
  - Pour les listes triés
    - O(n): aller jusqu'à la fin de la liste
  - Pour les tableaux triés
    - O(1): c'est l'élément à l'indice n-1 (ou n)

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Maximum(L)	O(n)	O(n)	O(n)	O(n)	O(n)	O(1)

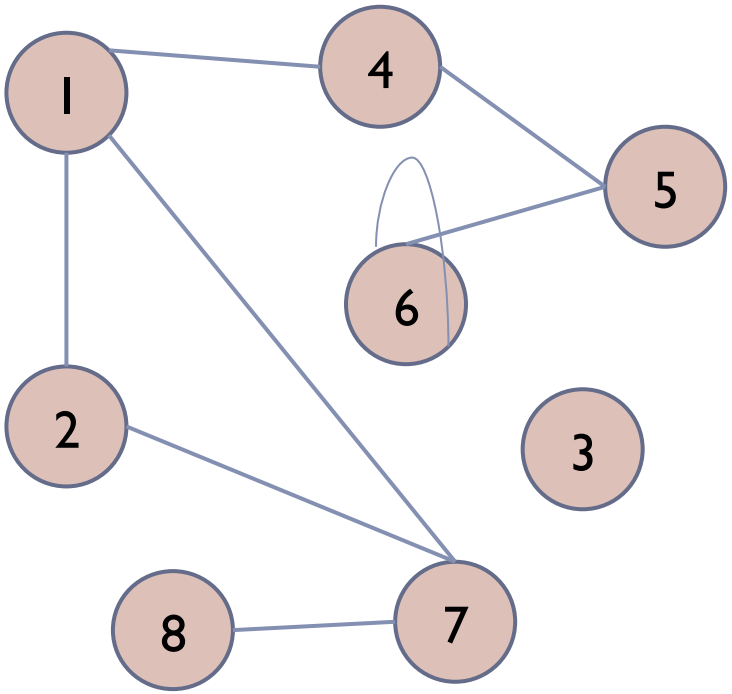
Tableaux de comparaison global

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Rechercher(L,k)	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)
Insertion(L,x)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)
Suppression(L,x)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Successeur(L,x)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)
Prédécesseur(L,x)	O(n)	O(n)	O(n)	O(1)	O(n)	O(1)
Minimum(L)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)
Maximum(L)	O(n)	O(n)	O(n)	O(n)	O(n)	O(1)

Graphes et Arbres
Graphes non orientés

- Un **graphe non orienté G** est représenté par un couple (S, A).
  - S est un ensemble fini. Il correspond à l'ensemble des **sommets** (ou **nœuds**) de G
  - A une relation binaire sur S. Elle définit l'ensemble des **arrêtes** de G
- Graphiquement, les nœuds sont représentés par des cercles et les arcs par des traits
- Les boucles sont les arcs qui relient un nœud à lui-même

Exemple



G=(A,S)

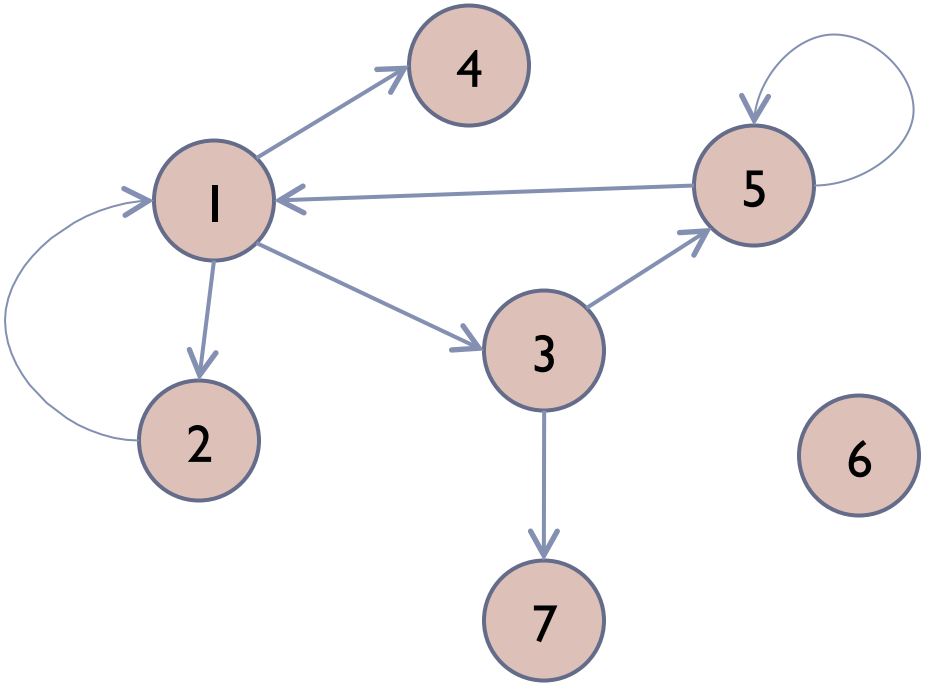
S={1;2;3;4;5;6;7;8}

A={
{1,2};{1,4};{1,7};
{2,7};
{4,5};
{5,6};
{6,6};
{7,8}
}

Graphes orientés

- Un **graphe orienté** *G* *est représenté par un couple (S, A).*
  - *S est un ensemble fini. Il correspond à l'ensemble des sommets (ou nœuds) de G*
  - *A une relation binaire* sur *S. Elle définit l'ensemble des* ***arcs*** *de G*
- *Graphiquement, les nœuds sont représentés par des cercles et les arcs par des flèches*
- *Les arcs sont définis par des couples (ordonnés) au lieu d'ensembles de deux éléments.*

Exemple



G=(S,A)
S={1;2;3;4;5;6;7}
A={(1,2);(1,3);(1,4);(2,1);(3,5);(3,7);(5,1);(5,5)}

Lexique (graphes non orientés)

- Si (u;v) est une arête d'un graphe non orienté G = (S;A), on dit que l'arête {u,v} est **incidente** aux sommets u et v.
- Dans un graphe non orienté, le **degré d'un sommet** est le nombre d'arêtes qui lui sont incidentes.
- Si un sommet est de degré 0, il est dit **isolé**.

Lexique (graphes orientés)

- Si (u,v) est un arc d'un graphe orienté G = (S;A), on dit que (u,v) **part** du sommet u et **arrive** au sommet v.
- Dans un graphe orienté,
  - **le degré sortant** d'un sommet est le nombre d'arcs qui en partent,
  - **le degré entrant** est le nombre d'arcs qui y arrivent
  - **le degré** est la somme du degré entrant et du degré sortant.

Les chemins et les chaînes

- Dans un *graphe orienté G = (S;A), un* **chemin de longueur k** *d'un sommet u à un sommet v est une séquence* $(u_0;u_1;\ldots;u_k)$ *de sommets telle que*
  - $u = u_0$,
  - $v = u_k$
  - $(u_{i-1},u_i) \in A$ *pour tout i dans* $\{1;\ldots;k\}$.
- *Un chemin est* **élémentaire** si ces sommets sont tous distincts
- Pour un graphe non orienté, on parle de **chaîne** à la place de chemins

Sous chemins

- Un sous-chemin p' d'un chemin $p = (u_0;u_1; \ldots;u_k)$ est une sous-séquence contiguë de ses sommets:
  - il existe i et j, avec $0 \leq i \leq j \leq k$, tels que
  - $p' = (u_i; u_{i+1}; \ldots; u_j)$.
- Pour les graphes non orientés, on parle de la notion de sous chaîne.

Les circuits

- Dans un graphe G=(S;A), un chemin $(u_0;u_1;\ldots;u_k)$ forme **un circuit** si $u_0 = u_k$ et si le chemin contient au moins un arc.
- **Un circuit est élémentaire** si les sommets $u_1, \ldots, u_k$ sont distincts.
- Une boucle est un circuit de longueur 1
- Un graphe sans cycle est dit **acyclique**.

Connexité

- Un graphe non orienté est **connexe** si chaque paire de sommets est reliée par une chaîne.
- **Les composantes connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « est accessible à partir de »
- Un graphe orienté est **fortement connexe** si chaque sommet est accessible à partir de n'importe quel autre.
- **Les composantes fortement connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation **«**sont accessibles l'un à partir de l'autre ».

Arbres - Définitions

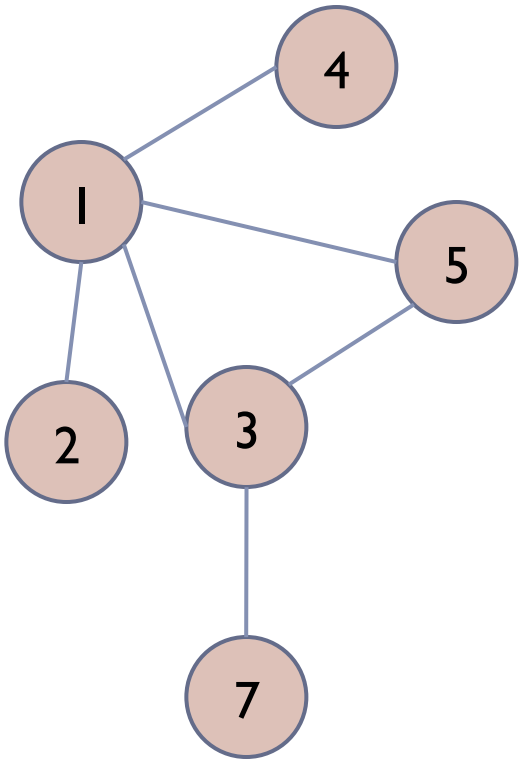
- un graphe non orienté connexe acyclique est **un arbre**
- Un graphe non orienté acyclique est **une forêt**
- *Les composantes connexes d'un graphe non orienté acyclique constituent les arbres de la forêt*

Arbres - Définitions

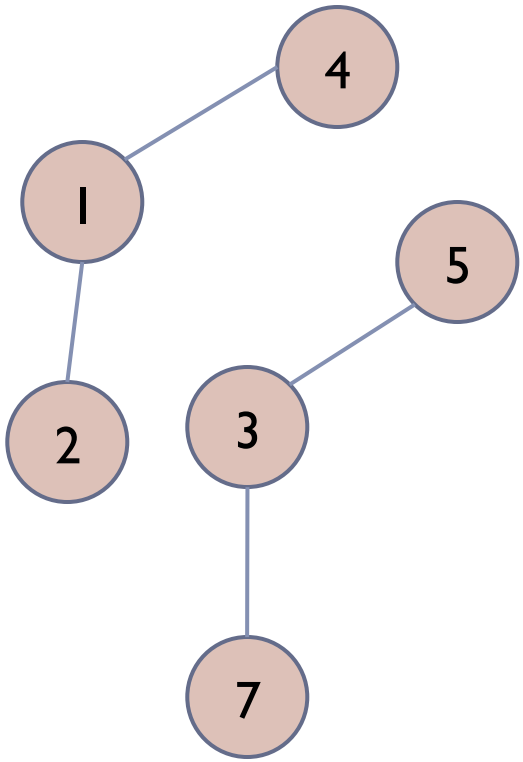
Définitions équivalentes

- *Soit G = (S;A) un graphe non orienté. Les affirmations suivantes sont équivalentes.*
  - *G est un arbre.*
  - *Deux sommets quelconques de G sont reliés par un unique chemin élémentaire.*
  - *G est connexe, mais si une arête quelconque est ôtée de A, le graphe résultant n'est plus connexe.*
  - *G est connexe et |A| = |S|-1.*
  - *G est acyclique et |A| = |S|-1.*
  - *G est acyclique, mais si une arête quelconque est ajoutée à A, le graphe résultant contient un cycle.*

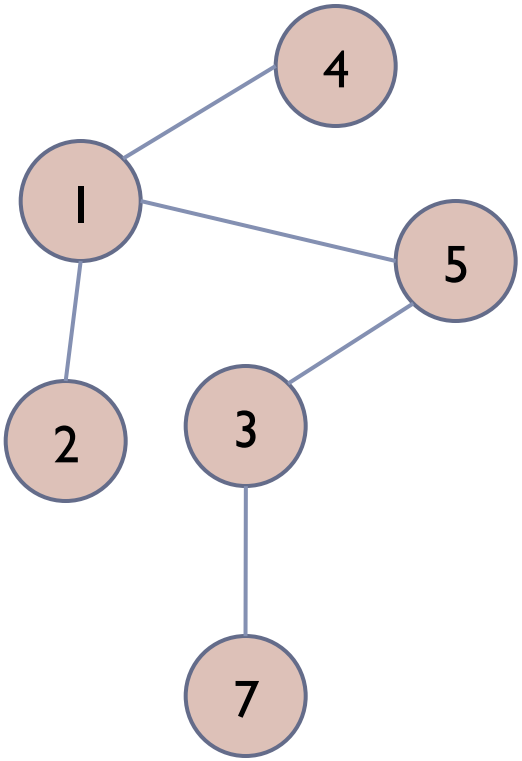
Exemples



Graphe contenant un cycle



Forêt



Arbre

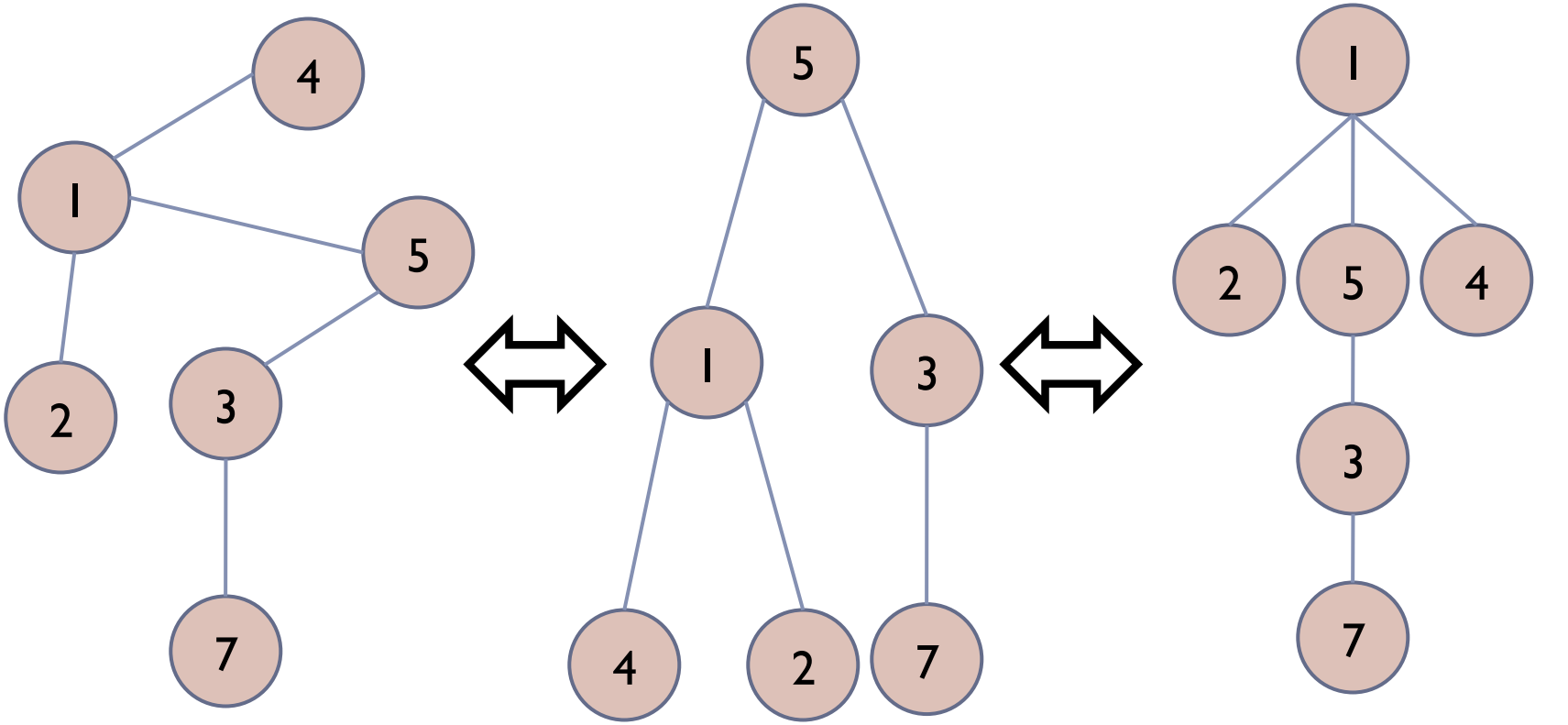
Arbre enraciné

- **Un arbre enraciné** est un arbre dans lequel l'un des sommets se distingue des autres.
- On appelle ce sommet **la racine**.
- Ce sommet particulier impose en réalité un sens de parcours de l'arbre et l'arbre se retrouve orienté par l'utilisation qui en est faite

Types de nœuds

- Soit x un nœud d'un arbre T de racine r. Un nœud quelconque y sur l'unique chemin allant de **r** à x est appelé **ancêtre** de x.
- Si T contient l'arête {y,x} alors y est **le père** de x, et x est **le fils** de y.
- La racine est le seul nœud qui ne possède pas de père
- Un nœud sans fils est **un nœud externe** ou **une feuille**
- Un nœud qui n'est pas une feuille est **un nœud interne**
- Le **sous-arbre de racine x** est l'arbre composé des descendants de x, enraciné en x.

Exemples: deux arbres avec deux racines différentes



Graphe G

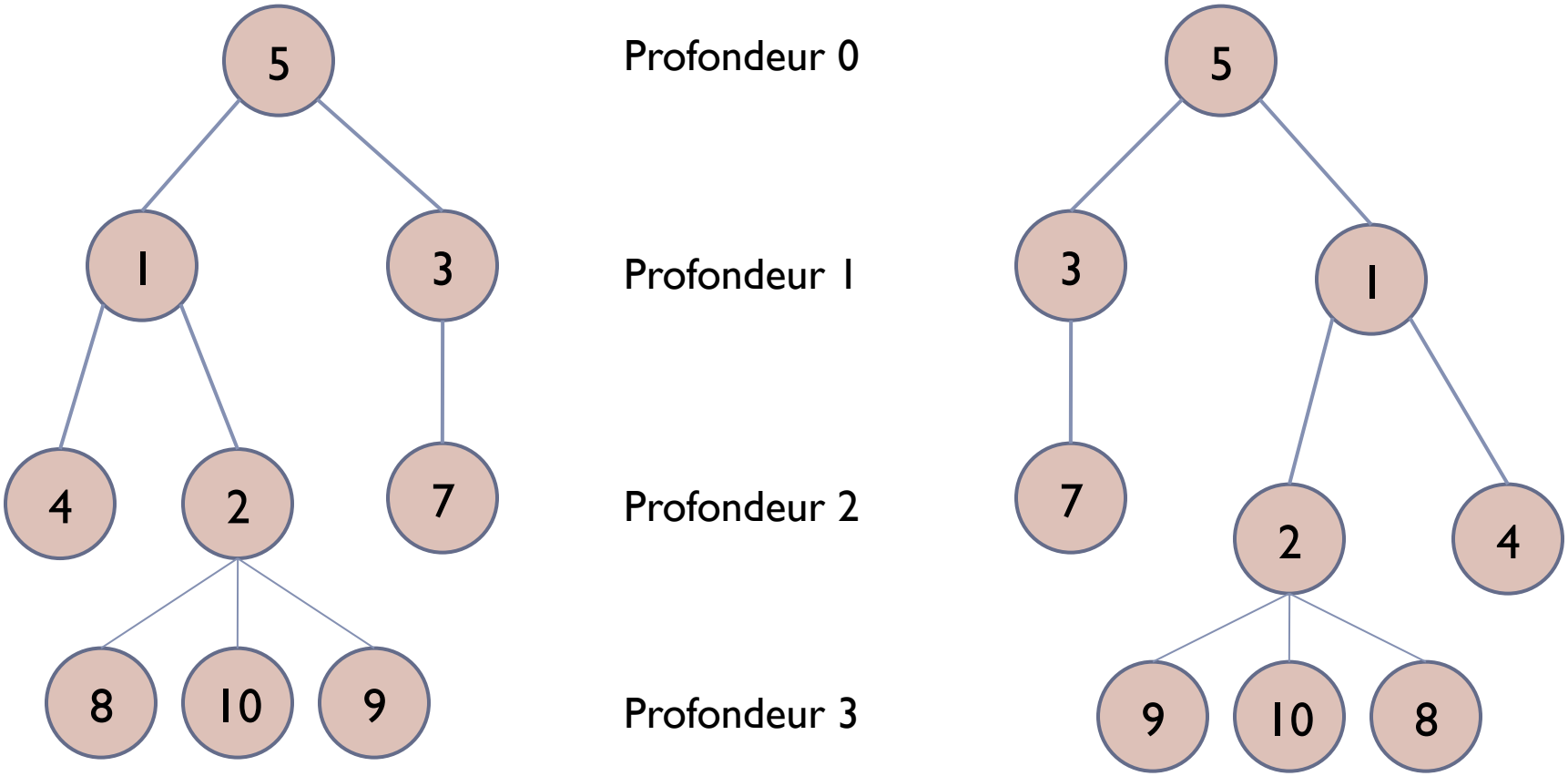
Arbre avec G enraciné sur le nœud 5

Arbre avec G enraciné sur le nœud 1

Degré et profondeur

- Le nombre de fils du nœud *x* *est appelé le* ***degré de x***
- La longueur du chemin entre la racine ***r*** *et* le nœud *x* *est la* ***profondeur de x***
- Un **arbre ordonné** est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés entre eux.

Exemples profondeur et ordre



Deux arbres qui ne diffèrent que par l'ordre

Arbres binaires

- Un **arbre binaire** est un arbre ordonné dont chaque nœud serait de degré au plus deux
- Un arbre binaire T est une structure définie récursivement sur un ensemble fini de nœuds et qui :
  - ne contient aucun nœud, ou
  - est formé de trois ensembles disjoints de nœuds:
    - une racine
    - un arbre binaire appelé son sous-arbre gauche
    - un arbre binaire appelé son sous-arbre droit
- Dans un arbre binaire, si un nœud n'a qu'un seul fils, la position de ce fils (fils gauche ou fils droit) est importante

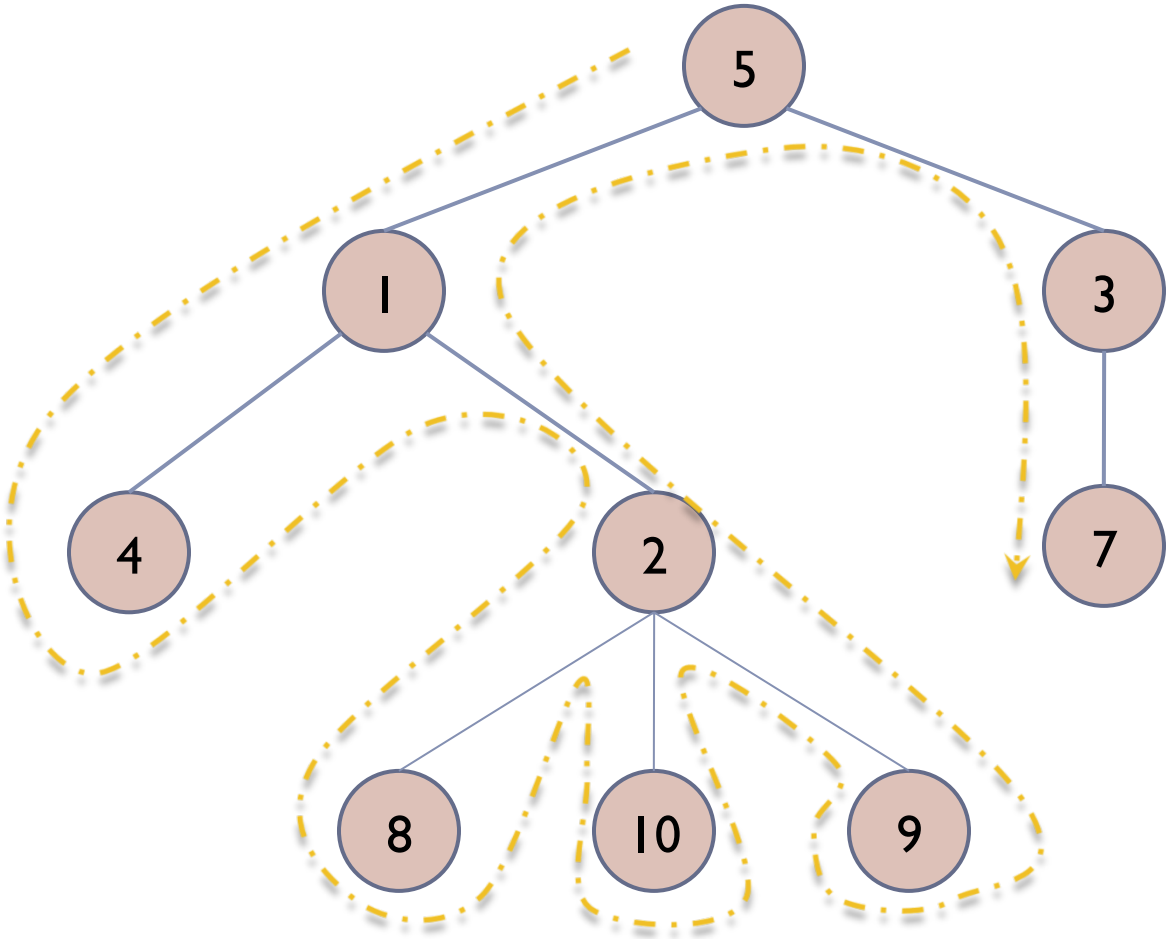
Arbres complets

- Dans **un arbre binaire complet** chaque nœud est soit une feuille, soit de degré deux, aucun nœud n'est donc de degré un.
- **Un arbre k-aire** est une généralisation de la notion d'arbre binaire où chaque nœud est de degré au plus k et non plus simplement de degré au plus 2
- **Un arbre k-aire complet** est un arbre où chaque nœud est soit de degré k ou soit une feuille

Parcours des arbres ordonnés
Parcours en profondeur

- Dans un *parcours en profondeur d'abord:*
  - *on descend le plus profondément possible dans l'arbre*
  - *une fois* qu'une feuille a été atteinte, on remonte pour explorer les autres branches en commençant par la branche la plus basse parmi celles non encore parcourues
  - *les fils d'un nœud sont bien évidemment parcourus suivant l'ordre sur l'arbre*

Sens du parcours



Algorithme de parcours en profondeur d'abord

ParcoursProfondeur(***A***)

si *A n'est pas réduit à une feuille*

alors

pour tous les fils *u* de *racine(A) (dans l'ordre)*

ParcoursProfondeur(u)

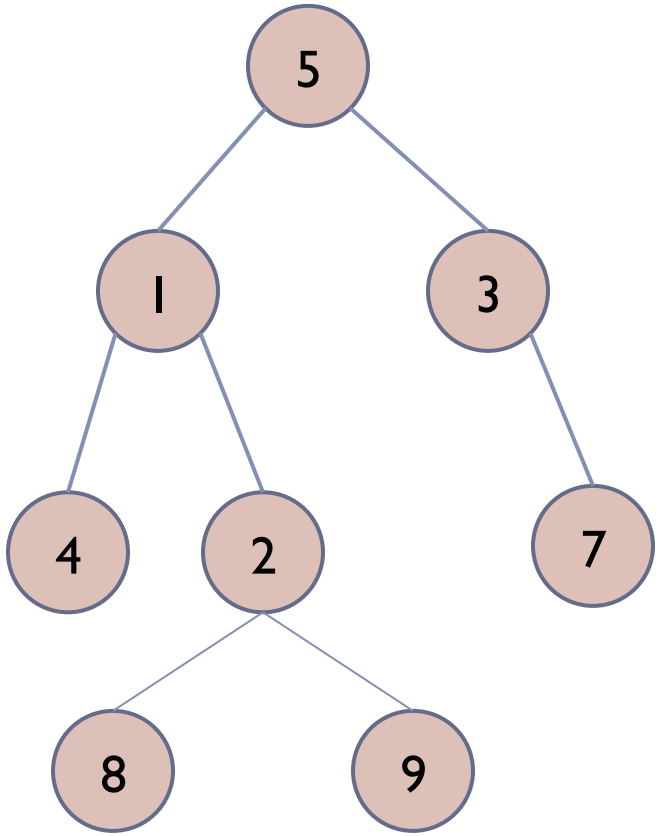
FinPour

Finsi

Exemples d'utilisation de la recherche en profondeur

Algo 1 : Parcourt préfixe

```
PRÉFIXE(A)
    si A <> NULL alors
    affiche racine(A)
    PRÉFIXE(FILS-GAUCHE(A))
    PRÉFIXE(FILS-DROIT(A))
```

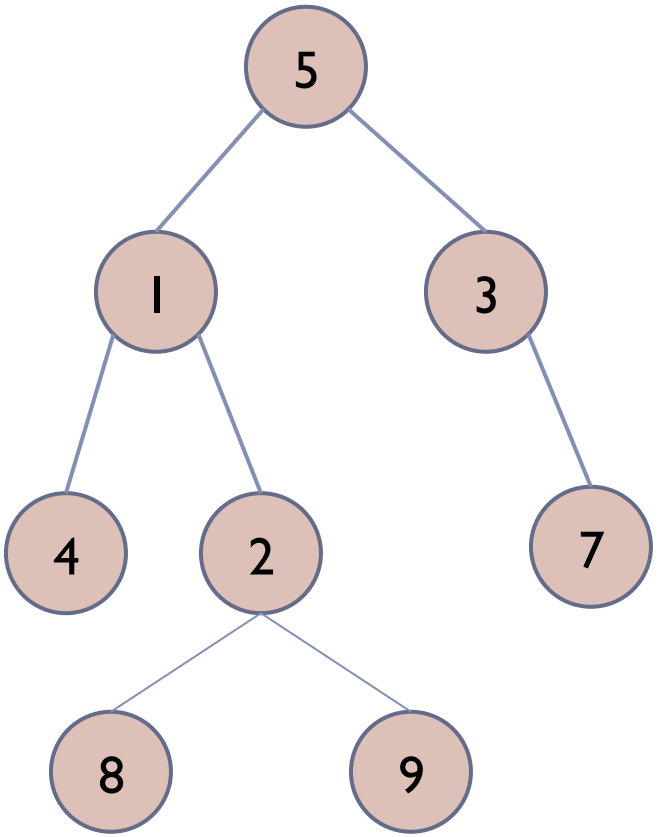


Résultat: 5;1;4;2;8;9;3;7

Exemples d'utilisation de la recherche en profondeur

Algo2: Parcourt Infixe

```
INFIXE(A)
    si A <> NULL faire
     INFIXE(FILS-GAUCHE(A))
     affiche racine(A)
     INFIXE(FILS-DROIT(A))
```



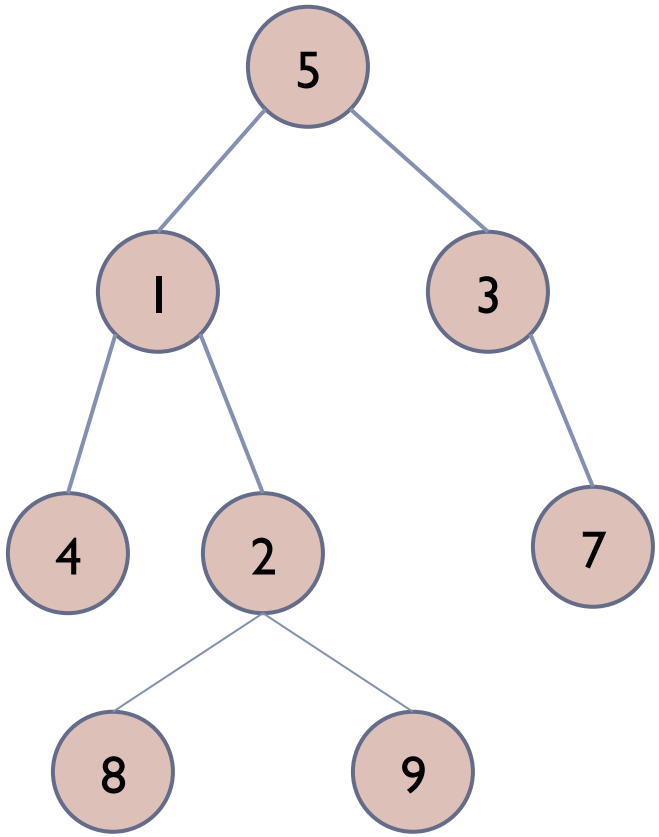
Résultat: 4;1;8;2;9;5;3;7

Exemples d'utilisation de la recherche en profondeur

Algo3: Parcourt Postfixe

```
POSTFIXE(A)
    si A <> NULL faire
     POSTFIXE(FILS-GAUCHE(A))
     POSTFIXE(FILS-DROIT(A))
     affiche racine(A)
```

Résultat: 4;8;9;2;1;7;3;5



Parcours en largeur

- Dans un *parcours en largeur d'abord:*
  - *On commence par la racine*
  - *On parcourt tous les nœuds à la profondeur i dans l'ordre*
  - *On passe ensuite au nœuds de profondeur i+1*
  - *Jusqu'à atteindre la profondeur de l'arbre*

Algorithme de parcours

- Pour établir l'algorithme de parcours en largeur d'abord, il faut sauvegarder les branches non encore visitées
- La structure la plus appropriée pour cela est la file

- **ParcoursLargeur(A)**

```
F ← {racine(A)}
tant que F <> { } faire
   u ←SUPPRESSION(F)
   pour tous les fils v de u faire dans l'ordre
      INSERTION(F,v)
   FinPour
FinTantQue
```

Sens du parcours