

Institut Supérieur du Numérique (SupNum)



Structures de données et complexité algorithmique

Semestre : S3



STRUCTURES DE DONNÉES SÉQUENTIELLES ET LINÉAIRES

- Structures de données séquentielles (Tableaux)
- Structures de données linéaires (listes chaînées, piles, files)

STRUCTURES DE DONNÉES SÉQUENTIELLES ET LINÉAIRES

- Structures de données séquentielles (Tableaux)
- Structures de données linéaires (listes chaînées, piles, files)

LES TABLEAUX

Définition

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, le tableau.

Un tableau en mémoire

Définition

Dans le tableau, tous les éléments ont la même taille mémoire.

Nombre d'élément : n , taille d'un élément t :

$T[0]$	$T[1]$	$T[2]$	$\dots$	$T[n-1]$
--------	--------	--------	---------	----------

On suppose que le tableau commence à l'adresse d

- $T[0]$ occupe les cases d à $d + t - 1$;
- $T[1]$ occupe les cases $d + t$ à $d + 2t - 1$;
- $T[i]$ occupe les cases $d + it$ à $d + (i + 1)t - 1$;
- Le tableau entier occupe les cases d à $d + nt - 1$.

Tableau partiellement remplis

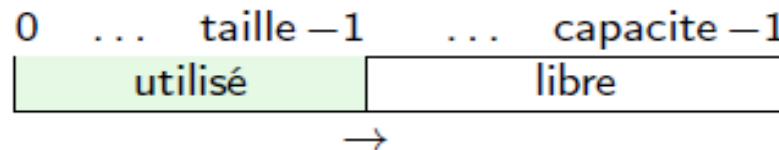
Retenir

Pour simuler un tableau de taille variable, on peut

- réserver une certaine quantité de mémoire appelée **capacité**,
- et ranger les valeurs **au début du tableau**.

Organisation des données (structure, classe) :

- tableau de taille capacité alloué
- éléments d'indice i pour $0 \leq i < \text{taille} \leq \text{capacité}$ initialisés



Opérations de base

Hypothèses :

- tableau de taille capacité alloué
- éléments $0 \leq i < \text{taille} \leq \text{capacité}$ initialisés

Retenir (Opérations de base)

- *accès au premier élément* : $\Theta(1)$
- *accès à l'élément numéro i* : $\Theta(1)$
- *accès au dernier élément* : $\Theta(1)$
- *insertion/suppression d'un élément au début* : $\Theta(\text{taille})$
- *insert./suppr. d'un élt en position i* : $\Theta(\text{taille} - i) \subset O(\text{taille})$
- *insert./suppr. d'un élt à la fin* : $\Theta(1)$

LISTES CHAÎNÉES

Définition

- une liste linéaire = une suite d'éléments de même type repérés par leur emplacement,
- elle comporte n élément où n peut varier au cours de l'exécution,
- chaque élément est à une place donnée. Les places sont ordonnées et repérées par leur rang,
- la première place est appelé tête de la liste et la dernière "queue" de la liste
- on suppose qu'il y a une fonction successeur (succ) qui donne l'élément suivant pour chaque élément de la liste
- on va distinguer deux types de listes : accès direct sur un élément de rang i ou indirect de façon séquentielle à partir de la tête (la fonction succ)

IMPLÉMENTATION DES LISTES CHAINÉES À ACCÈS INDIRECT EN C/C++

```
typedef struct Place Place;  
struct Place  
{  
    int val;  
    struct Place *suiv;  
};  
typedef Place* Liste;
```

TYPES ET OPÉRATIONS

- Type : Liste, Place
- Utilise : Élément

Opérations

- 1 ListeVide : $\longrightarrow$ Liste
- 2 tête : Liste $\longrightarrow$ Place
- 3 Contenu : Place $\longrightarrow$ Élément
- 4 Cons : Liste $\times$ Élément $\longrightarrow$ Liste
- 5 Succ : Place $\longrightarrow$ Place
- 6 Premier : Liste $\longrightarrow$ Élément
- 7 Fin : Liste $\longrightarrow$ Liste

QUELQUES AXIOMES

Axiomes

- ① tête(listevide), fin(listevide) et premier(listevide) ne sont pas définis,
- ② premier(cons(L,e)) = e,
- ③ premier(L) = contenu(tête(L))
- ④ fin(cons(L,e)) = L,
- ⑤ succ(tête(L)) = tête(fin(L))

IMPLÉMENTATION DE QUELQUES OPÉRATIONS

ListeVide

- 1: ListeVide($L : \text{Liste}$)
- 2: $L \leftarrow \text{NULL}$ (ou retourner NULL)

IMPLÉMENTATION DE QUELQUES OPÉRATIONS

Contenu

- 1: contenu(E : Élément, P : Place)
- 2: Entrée : P
- 3: Sortie : E
- 4: $E = P.val$ (ou retourner $P.val$)

IMPLÉMENTATION DE QUELQUES OPÉRATIONS

SUCC

- 1: succ(P, Q : Place)
- 2: Entrée : P
- 3: Sortie : Q
- 4: Si P.suiv = NULL Alors erreur
- 5: Sinon Q $\leftarrow$ *(P.suiv)

IMPLÉMENTATION DE QUELQUES OPÉRATIONS

SUCC

- 1: succ(P, Q : Place)
- 2: Entrée : P
- 3: Sortie : Q
- 4: Si P.suiv = NULL Alors erreur
- 5: Sinon Q $\leftarrow$ *(P.suiv)

IMPLÉMENTATION DE QUELQUES OPÉRATIONS

Fin

- 1: fin(L 1 :Liste, L2 : Liste)
- 2: Entrée : L1
- 3: Sortie : L2
- 4: Si L1 = NULL Alors erreur
- 5: Sinon L2 $\leftarrow$ L1->suiv

PILES

Définition

- ① une pile = une liste dans laquelle les éléments sont ajoutés et supprimés à une seule extrémité, le sommet de la pile (analogie avec une pile d'assiettes),
- ② elles portent donc le nom de Last In First Out (LIFO) : dernier entré, premier sorti

Applications

- ① appel des fonctions récursives (sont utiles pour le passage d'une fonction récursive à une itérative),
- ② analyses syntaxique
- ③ calcul arithmétique

PILES

Opérations

- ① $\text{pileVide} \longrightarrow \text{Pile}$
- ② $\text{empiler Pile} \times \text{Élément} \longrightarrow \text{Pile}$
- ③ $\text{dépiler Pile} \longrightarrow \text{Pile}$
- ④ $\text{sommet Pile} \longrightarrow \text{Élément}$
- ⑤ $\text{estVide Pile} \longrightarrow \text{Booléen}$

Axiomes

- ① $\text{sommet}(\text{pilevide})$ et $\text{dépiler}(\text{pileVide})$ ne sont pas définis,
- ② $\text{sommet}(\text{empiler}(P,E)) = E,$
- ③ $\text{dépiler}(\text{empiler}(P,E)) = P,$
- ④ $\text{estvide}(\text{pileVide}) = \text{vrai},$
- ⑤ $\text{estvide}(\text{empiler}(P,E)) = \text{faux}$

EXEMPLE

Exemple

- ① $P \rightarrow \text{pileVide}$
- ② `empiler(P,100)`
- ③ `empiler(P,17)`
- ④ `afficher(sommet(P))`
- ⑤ `dépiler(P)`
- ⑥ `afficher(sommet(P))`
- ⑦ `dépiler(P)`
- ⑧ `afficher(estVide(P))`

IMPLÉMENTATION

```
typedef struct Place Place;  
struct Place  
{  
    int val;  
    Place *prec;  
};  
typedef Place* Pile;
```

OPERATIONS

pileVide

- 1: pileVide(P : Pile)
- 2: Sortie : P
- 3: $P \leftarrow NULL$ (ou retourner NULL)

Empiler

- 1: empiler(P : pile, E : Élément)
- 2: Entrée : P, E
- 3: Sortie : P
- 4: $Q \rightarrow \text{Allouer}(Q)$ (Q de type *Place)
- 5: $Q \rightarrow \text{val} \leftarrow E$
- 6: $Q \rightarrow \text{prec} \leftarrow P$
- 7: $P \leftarrow Q$ (ou retourner Q)

Dépiler

- 1: dépiler(P : pile)
- 2: Entrée : P
- 3: Sortie : P
- 4: SI $P = NULL$ Alors erreur
- 5: Sinon
- 6: $Q \longrightarrow P \rightarrow \text{prec}$
- 7: $P \leftarrow Q$ (ou retourner Q)

Sommet

- 1: sommet(P : pile, E : Élément)
- 2: Entrée : P
- 3: Sortie : E
- 4: SI $P = NULL$ Alors erreur
- 5: Sinon
- 6: $E \longrightarrow P \rightarrow \text{val}$ (retourner $P \rightarrow \text{val}$)

EstVide

- 1: estVide(P : pile, B : Booléen)
- 2: Entrée : P
- 3: Sortie : B
- 4: $B \leftarrow (P == \text{NULL})$ (retourner $(P == \text{NULL})$)

FILES

Définition

- une file est une liste pour laquelle les éléments sont ajoutés à une extrémité et retiré d'une autre,
- elles portent le nom de First In First Out (FIFO) : premier entré, premier sorti,
- les opérations définies sur les files sont : accéder au premier élément, ajouter un élément, retirer le premier élément, tester si la file est vide.

IMPLÉMENTATION

Implémentation avec pointeurs

```
typedef struct Place Place;
struct Place
{
    int val;
    Place *suiv;
};
typedef struct File File;
struct File
{
    Place *debut
    Place *fin;
};
```

OPERATIONS

Enfiler (ajout d'un élément à la fin)

```
1: enfiler(F : File, E : Élément)
2: Entrée : F, E
3: Sortie : F
4: Allouer(Q)
5: Q->val ← E
6: Q->suiv ← NULL
7: Si F.debut = NULL Alors
8:   F.debut ← Q
9:   F.fin ← Q
10: Sinon
11:   (F.fin)->suiv ← Q
12:   F.fin ← Q
```

OPÉRATIONS

Premier (récupération de la valeur du premier élément)

```
1: Premier(F : File, E : Élément)
2:  Entrée : F
3:  Sortie : E
4:  Si F.debut = NULL Alors
5:    Erreur
6:  Sinon
7:    E ← (F.debut)->val
```

Défiler (suppression du premier élément)

```
1: defiler(F : File)
2:  Entrée : F
3:  Sortie : F
4:  Si F.debut = NULL Alors Erreur
5:  Sinon
6:    F.debut ← (F.debut)->suiv
7:  Si F.debut = NULL Alors
8:    F.fin ← NULL
```

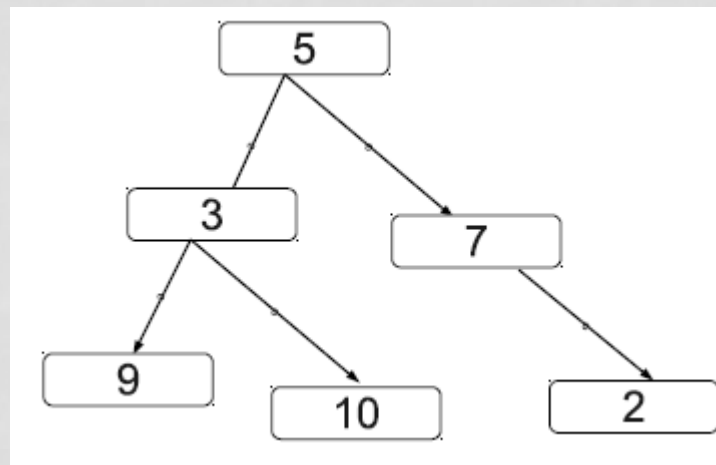
ARBRES

- souvent utiles pour représenter des données dont l'organisation est de nature hiérarchique
- par exemple, une expression arithmétique
- Les arbres binaires sont tout particulièrement intéressants car ils permettent de représenter généralement n'importe quel type d'arbre planaire

ARBRES BINAIRES

définition

un arbre binaire B est soit un arbre vide, soit un élément de la forme $\langle o, B1, B2 \rangle$ où o est un nœud appelé racine et où $B1$ et $B2$ sont des arbres binaires disjoints (pas de nœud en commun)



ARBRES BINAIRES - REPRÉSENTATION

Implémentation

```
typedef struct Noeud Noeud;  
struct Noeud  
{  
    int val;  
    Noeud * g;  
    Noeud * d;  
};  
typedef Noeud * Arbre;
```

DES OPÉRATIONS

- `ArbreVide` : retourne un arbre vide
- `Cons(Nœud, arbre, arbre)` : retourne un arbre
- `Racine (arbre)` : retourne un nœud
- `Gauche(arbre)` : retourne un arbre
- `Droit( arbre)` : retourne un arbre

ArbreVide

- 1: ArbreVide($B : \text{Arbre}$)
- 2: Entrée : Rien
- 3: Sortie : B
- 4: $B \leftarrow \text{NULL}$

Racine

- 1: Racine($B : \text{Arbre}; N : \text{Noeud}$)
- 2: Entrée : B
- 3: Sortie : E
- 4: Si ($B = \text{NULL}$) Alors erreur
- 5: Sinon
- 6: $E \leftarrow *B$

Construction d'un ARB à partir de ses fils et sa racine

```
1: Cons(B, B1, B2 : Arbre; E : Elément)
2: Entrée : B1, B2, E
3: Sortie : B
4: Allouer(B)
5: B->val ← E
6: B->g ← B1
7: B->d ← B2
```

Récupération du fils gauche d'un arbre

```
1: gauche(B, B1 : Arbre)
2: Entrée : B
3: Sortie : B1
4: Si B=NUL Alors erreur
5: Sinon B1 ← B->g
```

QUELQUES NOTATIONS ET PROPRIÉTÉS

On considère un arbre $B = \langle o, B1, B2 \rangle$. On définit alors les termes suivants :

- la racine = o ,
- le sous-arbre gauche = $B1$,
- le sous-arbre droit = $B2$,
- un sous-arbre de $B = B$ lui même, $B1$, $B2$, ou un sous-arbre de $B1$ ou $B2$,
- fils gauche de o = racine $o1$ de $B1$ (lien gauche entre o et $o1$),
- fils droit de o = racine $o2$ de $B2$ (lien droit entre o et $o2$),
- o est le père de o' si o' est un fils (droite ou gauche) de o ,

- o et o' sont frères s'ils ont le même père,
- o est un ascendant de o' si o est le père de o' , ou si o est un ascendant du père de o' ,
- définition analogue pour descendant,
- o est un nœud externe (ou feuille) si o n'a aucun fils,
- o est un nœud interne (on dit aussi "interne au sens large") si o a au moins un fils,

- o est un point simple à gauche si o a seulement un fils à gauche,
- définition analogue pour point simple à droite (seulement un fils à droite),
- o est un nœud double si o a seulement exactement deux fils,
- une branche est un chemin menant de la racine à une feuille (B a autant de branches que de feuilles),

- le bord gauche de B est le chemin obtenu en partant de la racine o et en ne suivant que des liens gauches,
- définition analogue pour bord droit,
- la hauteur h (ou encore profondeur, niveau) d'un nœud o' dans un arbre B est définie récursivement par :
 - $h(o') = 1$ si o' est la racine de B
 - $h(o') = 1 + h(o'')$ où o'' est le père de o' ,

HAUTEUR D'UN ARBRE BINAIRE

- la hauteur h ou profondeur de B est la hauteur maximale de ses nœuds. Dans la suite on utilisera la définition de $h(\text{arbrevide}) = 0$,

Calcul de la hauteur

- 1: hauteur(B : Arbre, h : Entier)
- 2: Entrée : B
- 3: Sortie : h
- 4: Si $B = \text{NUL}$ Alors $h \leftarrow 0$
- 5: Sinon $h \leftarrow 1 + \max(\text{hauteur}(\text{gauche}(B)), \text{hauteur}(\text{droite}(B)))$

PARCOURS DES ARBRES

- parcours en profondeur
- parcours en largeur

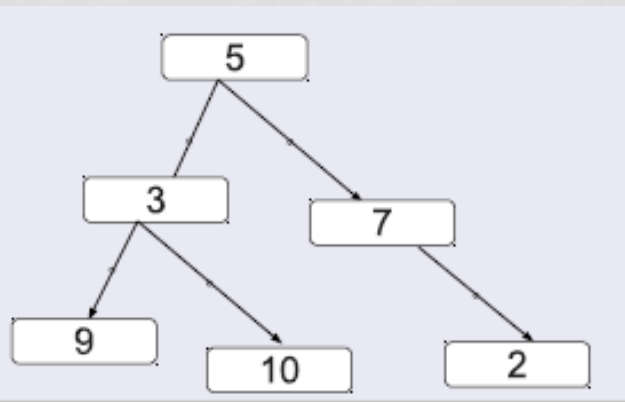
PARCOURS EN PROFONDEUR

Parcours en profondeur

On cherche à s'enfoncer dans l'arbre le plus profondément possible. Ainsi, On parcourt une branche jusqu'(au bout pour passer à la branche suivante (on commencer par la **gauche** ou la droite de l'arbre).

Trois stratégies sont possible

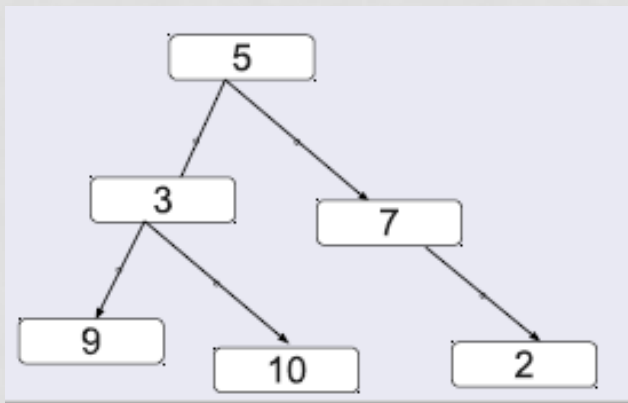
- parcours en profondeur d'ordre infixe
- parcours en profondeur d'ordre préfixe
- parcours en profondeur d'ordre postfixe(suffixe)



PARCOURS EN LARGEUR

Parcours en largeur

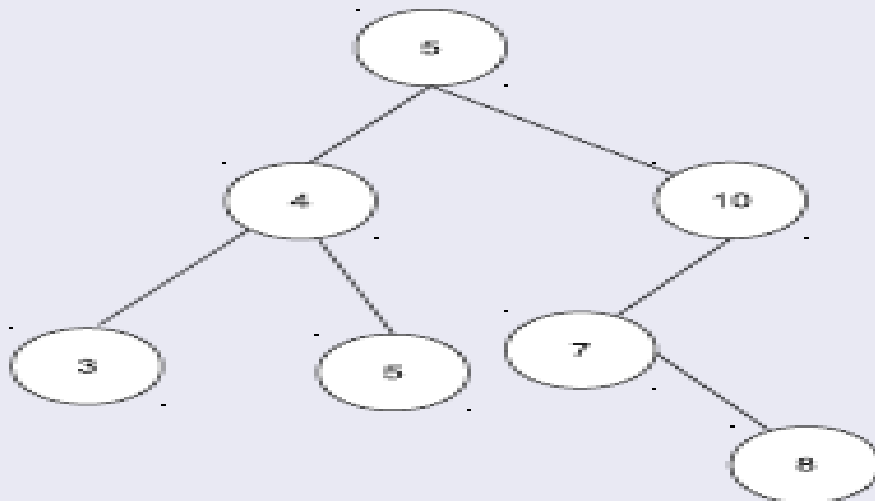
Consiste à parcourir l'arbre niveau par niveau en commençant par le niveau 1. Ainsi on traite la racine puis tous les fils de la racine avant de traiter leur fils.



ARBRES BINAIRES DE RECHERCHE

Un arbre binaire B est un arbre binaire de recherche s'il possède les propriétés suivantes, pour tout nœud x de B :

- $\forall y \in \text{sous arbre droit de } x, x < y$,
- $\forall y \in \text{sous arbre gauche de } x, y \leq x$,



OPERATIONS DE BASE

- adjonction d'un élément E aux feuilles de B ,
- recherche d'un élément E ,
- plus petit/grand élément de B ,
- affichage des éléments dans l'ordre (parcours infixe de l'arbre),
- suppression d'un élément E

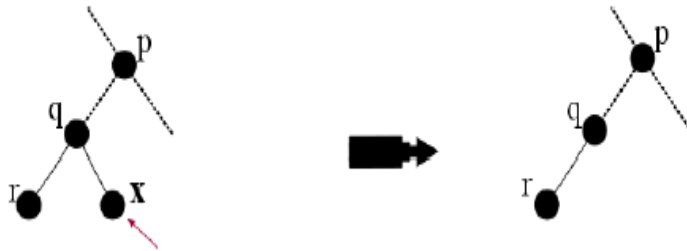
SUPPRESSION D'UN ÉLÉMENT

On distingue les cas suivants :

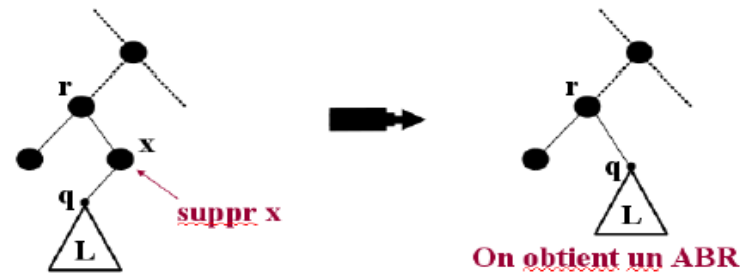
- B est vide,
- E n'est pas dans B,
- E est une feuille,
- E est un nœud simple,
- E est un nœud double : on remplace E soit par son successeur dans B soit par son prédécesseur.

SUPPRESSION D'UN ÉLÉMENT

- E est une feuille,



- E est un nœud simple,



- E est un nœud double,

