Type de Données Abstrait et Structures de Données

Représentation des données nécessaires à la résolution d'un problème

Définition d'un Type de Données Abstrait

Spécification des opérations permettant de manipuler les objets du type.
Exemple : opérations du TDA `Ens` :

appartient ? : Objet x Ens $\longrightarrow$ Booléen
ajouter : Objet x Ens $\longrightarrow$ Ens
...

$\forall o \in Objet, E \in Ens, appartient?(o, ajouter(o, E)) = Vrai$
...

Implantation d'un Type de Données Abstrait

Description de l'organisation des données et algorithmes réalisant les opérations du TDA (vérifiant ses spécifications) : **Structures de Données**
Exemple : plusieurs implantations possibles du TDA *Ens* : tableau, tableau de booléen, liste chaînée, arbre, table de Hachage, ...

Le type de Données Abstrait Pile

Opérations

pileVide ?(**d** *P* :Pile) :Bool ;
Données : *P* une Pile
Résultat : Renvoie un booléen indiquant si le Pile *P* est vide

créerPile() :Pile ;
Données :
Résultat : Renvoie une Pile vide

sommetPile(**d** *P* :pile) :X ;
Données : *P* une Pile non vide
Résultat : Renvoie l'élément sommet de pile

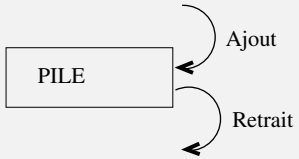
empiler(**dr** *P* :Pile, **d** *e* :X) ;
Données : une pile *P*, *e*
Résultat : ajoute *e* à la pile *P*

dépiler(**dr** *P* :Pile) ;
Données : une pile *P* non vide
Résultat : supprime le sommet de *P*

Le type de Données Abstrait Pile

Propriété

L'élément renvoyé par `sommetPile(P)` est le dernier élément empilé.
Gestion LIFO («Last In First Out»).



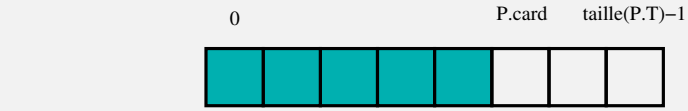
Exemple

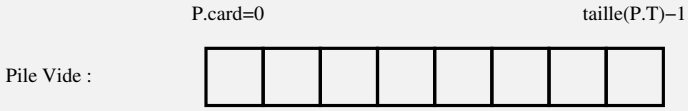
P← créerPile() ; empiler(P,1) ; empiler(P,2) ; empiler(P,3) ;

Implantation du type pile par un Tableau

Une Pile *P* est représentée par un couple :

- un tableau, noté `P.T`
- nombre d'éléments de la pile, noté `P.card`





Implantation du type pile par un Tableau

Algorithme : pileVide ?(**d** P) :Bool ;
début
|
fin algorithme

Algorithme : créerPile() :Pile ;
début
|
fin algorithme

Algorithme : sommetPile(**d** P) :X ;
début
|
fin algorithme

Opérations en

Algorithme :
empiler(**dr** P : Pile, **d** e : X)
début
|
|
|
fin algorithme

Algorithme : dépiler(**dr** P : Pile)
début
|
fin algorithme

Le Type de Données Abstrait File

Opérations

fileVide ?(**d** F : File) : Bool ;
Données : F une File
Résultat : Renvoie un booléen indiquant si la File est vide

créerFile() :File ;
Données :
Résultat : Renvoie une File vide

têteFile(**d** *F* : File) :X ;
Données : *F* une File non vide
Résultat : Renvoie l'élément en tête de File

ajouterFile(**dr** *F* : File,**d** *e* : X) ;
Données : une File *F*, *e*
Résultat : ajoute *e* à la File *F*

retirerFile(**dr** *F* : File) ;
Données : une File *F* non vide
Résultat : supprime la tête de la File *F*

Le Type de Données Abstrait File

Propriété

L'élément renvoyé par `têteFile(F)` est le premier élément ajouté à la file *F*. Gestion FIFO (« First In First Out»).

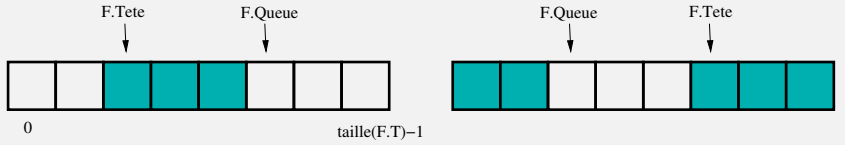
Exemple

F← créerFile() ; ajouterFile(F,1) ; ajouterFile(F,2) ; ajouterFile(F,3) ;

Implantation du type File par un Tableau

Une File *F* est représentée par un triplet :

- un Tableau, noté `F.T`
- l'indice de l'élément Tête de File, noté `F.Tête`
- l'indice suivant le dernier élément ajouté à la File, noté `F.Queue`



File Vide :

File Pleine :

Algorithme : fileVide ?(**d** F : File) :Bool ;
début
|
fin algorithme

Algorithme : créerFile() : File ;
début
|
fin algorithme

Algorithme : FilePleine ?(**d** F :File) :Bool ;
début
|
fin algorithme

Algorithme : ajouterFile(**dr** F : File, **d** e : X)
début
|
fin algorithme

Opérations en

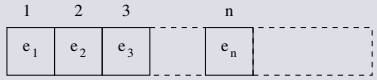
Algorithme : têteFile(**d** F : File) : X ;
début
|
fin algorithme

Algorithme : retirerFile(**dr** F : File)
début
|
fin algorithme

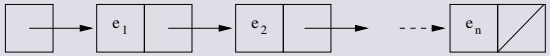
Listes

Une liste est une séquence d'éléments. $(e_1, e_2, \ldots, e_n)$.
Chaque élément de la liste a une place dans la liste. Les éléments de la liste peuvent être rangés de manière

- contigüe : l'élément suivant un élément de place p est à la place $p + 1$
  Tableau, vecteur



- chaînée : la place de l'élément suivant est mémorisée avec l'élément
  Liste chaînée



Remarque

- Le type Liste vu en GLIN101 : représentation chaînée
- Liste en CAML et SCHEME : repésentation chaînée
- Liste en MAPLE, PYTHON : représentation contigüe

Définition récursive des Listes Chaînées

Une liste est :

- soit la liste vide
- soit un couple (`Premier élément`,`Suite de la liste`)
  (tête de liste, queue de liste)

« Le type Liste Chaînée »

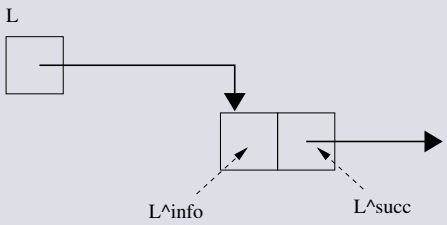
Pour manipuler une liste chaînée, nous avons besoin d'opérations pour :

- tester si une liste est vide
- accéder aux informations d'une liste non vide :
  - valeur du premier élément de la liste *(tête de liste)*
  - suite de la liste *(queue de liste)*
- construire une liste
- **modifier les informations d'une liste** *(nouveau)*

Une structure de Données pour les Listes Chaînées

- Chaque élément d'une liste sera représenté par une cellule double
- Une cellule est connue par son adresse.
- Une Liste est l'adresse de la cellule associée à son premier élément
- La liste vide est représentée par l'adresse `NULL`

Opérations sur les Listes Chaînées



L étant une variable de type liste non vide (pointeur dont la valeur est l'adresse de la cellule associée à son premier élément) :

- `L↑info` désigne la valeur du premier élément de la liste (*tête de liste*)
- `L↑succ` désigne la suite de la liste (*queue de liste*)

La fonction `créerListe` permet de construire une nouvelle liste

Algorithme : créerListe(**d** e : élément, **d** SL : Liste) : Liste
Données : e, SL
Résultat : Renvoie la liste dont le premier élément est `e` et la suite de la liste est `SL`

Exemple de manipulation de liste chaînée

- L1← créerListe(1,NULL) ;
- L2← créerListe(2,NULL) ;
- L3← créerListe(3,L2) ;
- L3↑info ← L2↑info ;
- L3↑succ ← L1 ;
- L2↑succ ← L3↑succ ;

Exemple d'algorithme sur les listes chaînées

Algorithme : longListe(**d** L :Liste) : entier
Données : `L` une liste
Résultat : Renvoie le nombre d'éléments de `L`

Variables P :Liste, nbElem : entier
début
  renvoyer `nbElem`
fin algorithme

Version récursive

Algorithme : longListe(**d** L :Liste) : entier
Données : `L` une liste
Résultat : Renvoie le nombre d'éléments de `L`

début
fin algorithme

Version affreusement fausse et trop souvent rencontrée !

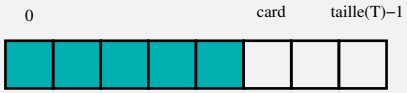
```
Variable cpt : entier
début
    cpt ← 0
    si L=NULL alors
        renvoyer cpt
    sinon
        cpt ← cpt+1
        longListe(L↑succ)
    fin si
fin algorithme
```

Opérations sur les Listes Chaînées

Opérations sur les listes

1. Recherche de la place d'un élément dans une liste
2. Insertion d'un élément dans une liste à une place déterminée
3. Suppression d'un élément d'une place donnée d'une liste

Si liste implantée par tableau,



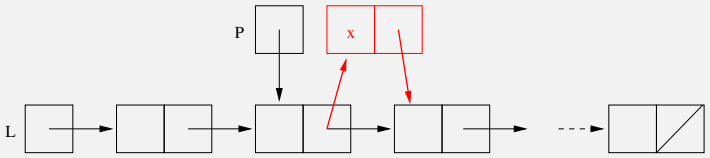
Les complexités de ces opérations sont, en fonction du nombre n d'éléments :

Recherche

Algorithme : recherche(**d** L : Liste, **d** x : X) :Liste
Données : L est une Liste Chaînée ; x
Résultat : renvoie NULL si $x \notin L$; sinon renvoie l'adresse de la première cellule de L contenant x

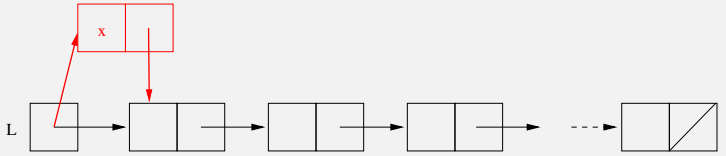
Variables : P : Liste
début
|
fin algorithme

début
|
fin algorithme



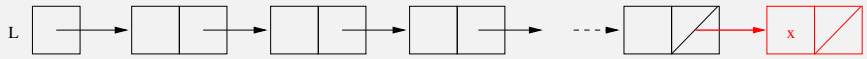
Insertion

Algorithme : insérerAprès(**dr** L :Liste,**d** P :Liste, **d** x : X)
Données : L est une Liste non vide, P un pointeur vers une cellule de L, x
Résultat : Insère dans la liste L une cellule contenant x après celle pointée par P
Variable Q : Liste
début
|
fin algorithme



Insertion

Algorithme : insérerDébut(**dr** L :Liste, **d** x : X)
Données : L est une Liste chaînée
Résultat : Insère au début de L une cellule contenant x
début
|
fin algorithme

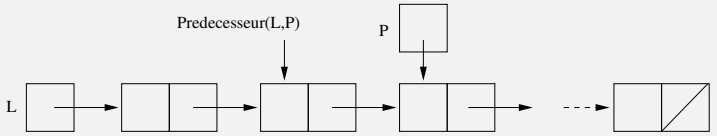


Insertion

Algorithme : insérerFin(**dr** L :Liste, **d** x : X)
Données : *L* est une Liste chaînée
Résultat : Insère en fin de la Liste *L* une cellule contenant *x*

Variable P : Liste
début
  si `L=NULL` **alors**
    L← créerListe(x,NULL)
  sinon
  fin si
fin algorithme

début
  si `L=NULL` **alors**
    L← créerListe(x,NULL)
  sinon
  fin si
fin algorithme

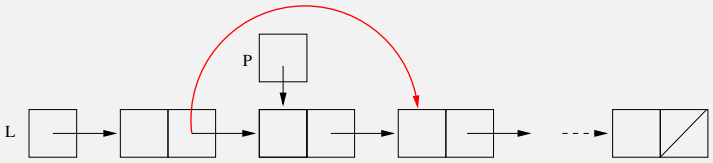


Prédécesseur

Algorithme : predecesseur(**d** L : Liste,**d** P : Liste) :Liste
Données : *L* est une Liste non vide ;*P* pointeur vers un élément de *L*, $L \neq P$
Résultat : renvoie l'adresse de la cellule précédant dans *L* celle repérée par *P*

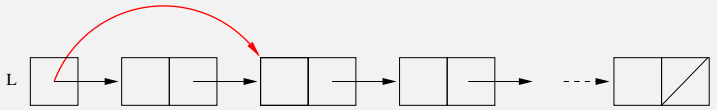
Variables : Q :Liste
début
  $Q \leftarrow L$
fin algorithme

début
  si `(L↑succ)=P` **alors**
  sinon
  fin si
fin algorithme



Suppression

Algorithme : supprimer(**dr** L : Liste, **d** P : Liste)
Données : *L* est une Liste non vide ;*P* un pointeur vers une cellule de *L*
Résultat : Modifie L en supprimant de *L* la cellule pointée par *P*
début
  si `L=P` **alors**
    L← L↑succ
  sinon
  fin si
fin algorithme



Suppression

Algorithme : supprimerDébut(**dr** L : Liste)
Données : *L* est une Liste non vide
Résultat : Supprime le premier élément de *L*
début
fin algorithme

Suppression

Algorithme : supprimerFin(**dr** L : Liste)
Données : *L* est une Liste non vide
Résultat : Supprime le dernier élément de *L*
Variable P

début
  si `L↑succ=NULL` **alors**
    L←NULL
  sinon
    fin si
fin algorithme

début
  si `L↑succ=NULL` **alors**
    L←NULL
fin algorithme

Exemple de représentation des listes en C

Définition du type liste et de la fonction `créerListe`

```
typedef   struct cellule {
          int info ;
          struct cellule *succ ;} CelluleSC ;
typedef   CelluleSC *ListeSC ;
```

```
ListeSC   creerLSC(int val, ListeSC succ){
          ListeSC li = (ListeSC) malloc(sizeof(CelluleSC)) ;
          li -> info = val ;
          li -> succ = succ ;
          return li ;}
```

version récursive de la fonction `insérerFin`

```
void  insererFinLSC( ListeSC *p, int val ){
      if ((*p)==NULL) (*p)=creerLSC(val,NULL) ;
      else insererFinLSC(&((*p)->succ),val) ;
      return ; }
```


Implantation de Pile et File par Liste Chaînée

Pile

On obtient les propriétés d'une Pile :

- empiler :
- dépiler :

File

On obtient les propriétés d'une File :

- ajouterFile :
- retirerFile :

Autres Listes

Défaut des Listes Simplement Chaînées

Les opérations coûteuses sur les listes chaînées :

- Calul de la cellule précédant une cellule (opération de suppression)
- Calcul de la dernière cellule (opérations d'insertion et de suppression en fin de liste)

Pour améliorer les complexités on peut mémoriser ces informations.

Listes Doublement Chaînées

Mémorisation du prédécesseur

Chaque élément d'une Liste Doublement Chaînée est représenté par une cellule triple. Chaque cellule d'adresse P contient contient 3 informations :

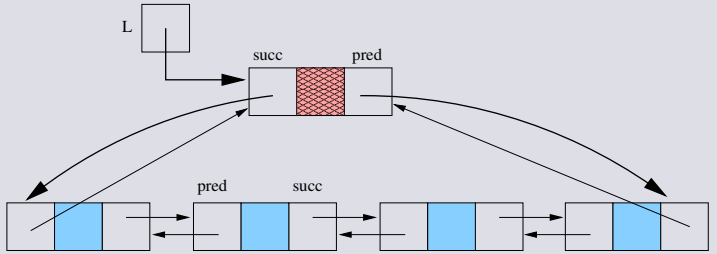
- la valeur de l'élément : $P\uparrow$info
- l'adresse de la cellule suivante : $P\uparrow$succ
- l'adresse de la cellule précédente : $P\uparrow$pred

Mémorisation du dernier

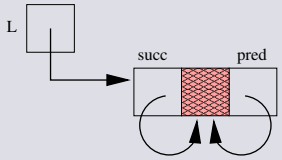
Une liste est représentée par 2 pointeurs contenant les adresses de la première et dernière cellule de la liste. Ces 2 pointeurs sont mémorisés dans une cellule de la liste. Une Liste Doublement Chaînée (LDC) L est un pointeur vers une cellule :

- $L\uparrow$info n'est pas utilisé
- $L\uparrow$succ est l'adresse de la cellule associée au premier élément
- $L\uparrow$pred est l'adresse de la cellule associée au dernier élément

Liste Doublement Chaînée de 4 éléments



Liste Doublement Chaînée Vide



Recherche

Algorithme : recherche(**d** L :LDC,**d** x : X) :adrCellule
Données : L est une Liste Doublement Chaînée x
Résultat : renvoie NULL si $x \notin L$; sinon renvoie l'adresse de la première cellule de L contenant x
Variables : P : adrCellule ; **début**

fin algorithme

Algorithme : créerTriplet(**d** LPred : adrCellule, **d** x : X, **d** LSucc : adrCellule) : adrCellule
Données : *LPred* et *LSucc* 2 adresses de cellule ; x une valeur
Résultat : Renvoie l'adresse d'une nouvelle cellule triple, dont l'information est x, l'adresse de la cellule précèdente est *LPred*, l'adresse de la cellule suivante est *LSucc*.

Insertion

Algorithme : insérerAprès(**d** L :LDC,**d** P :adrCellule, **d** x : X)
Données : L est une Liste Doublement Chaînée non vide, P un pointeur vers une cellule de L, x
Résultat : Insère dans la liste L une cellule contenant x après celle pointée par P
Variable Q : adrCellule ; **début**

fin algorithme

Insertion en début de liste

Algorithme : insérerDébut(**d** L :LDC, **d** x : X)
Données : *L* est une Liste Doublement Chaînée
Résultat : Insère au début de la liste *L* une cellule contenant *x*
début
|
fin algorithme

Insertion en fin de liste

Algorithme : insérerFin(**d** L :LDC, **d** x : X)
Données : *L* est une Liste Doublement Chaînée
Résultat : Insère en fin de la liste *L* une cellule contenant *x*
début
|
fin algorithme

Suppression

Algorithme : supprimer(**d** L : LDC, **d** P : adrCellule)
Données : *L* est une Liste Doublement Chaînée non vide ;*P* un pointeur vers une cellule de *L* ($P \neq L$)
Résultat : Supprime de *L* la cellule pointée par *P*
début
|
fin algorithme

Suppression en début de liste

Algorithme : supprimerDébut(**d** L :LDC)
Données : *L* est une Liste Doublement Chaînée non vide
Résultat : Supprime le premier élément de *L*
début
|
fin algorithme

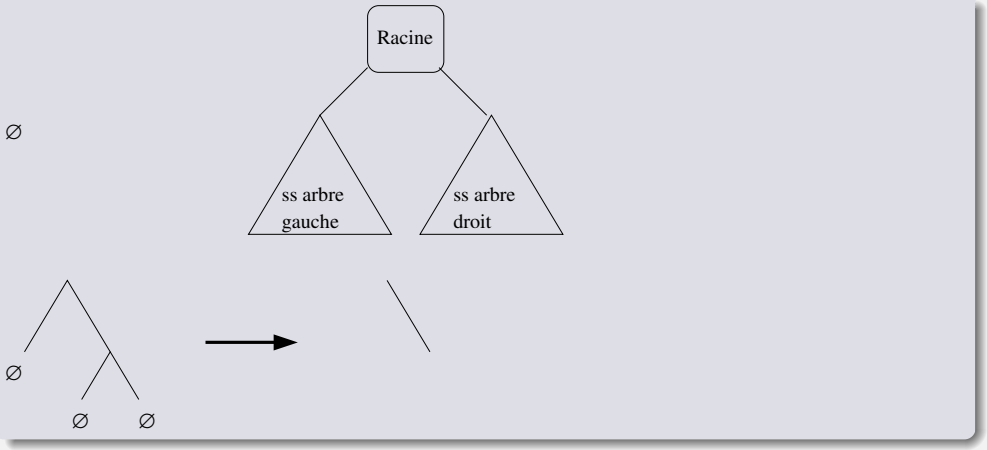
Suppression en fin de liste

Algorithme : supprimerFin(**d** L :LDC)
Données : *L* est une Liste Doublement Chaînée non vide
Résultat : Supprime le dernier élément de la liste *L*
début
|
fin algorithme

Définition

Un arbre binaire est

- soit l'arbre vide
- soit un triplet composé d'une racine et de 2 arbres binaires, appelés sous-arbre gauche et sous-arbre droit.



Définition

- Noeuds d'un arbre non vide = sa racine + les noeuds de ses 2 sous-arbres.
- Arbre étiqueté : à chaque noeud est associée une information (étiquette).
- Feuille : Arbre non vide dont les 2 sous-arbres sont vides.
- Profondeur d'un noeud d'un arbre :
  - profondeur de la racine = 0
  - profondeur d'un noeud = profondeur de son père + 1
- Hauteur d'un arbre : profondeur maximum de ses noeuds

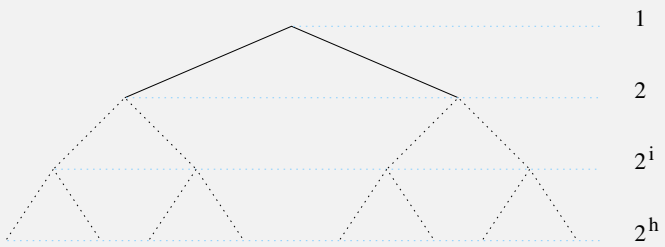
Relation entre hauteur et nombre de noeuds d'un arbre binaire

La hauteur h d'un arbre binaire possédant n noeuds vérifie :

$$\lceil log(n+1) \rceil - 1 \leq h \leq n - 1$$

- Arbre filiforme $h = n - 1$
- Arbre Complet

$n =$ $=$



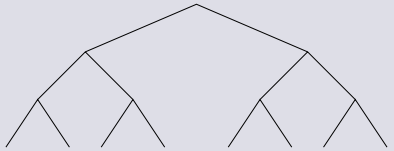
Arbres particuliers

Arbres Filiformes

Arbres n'ayant qu'une feuille.

Arbres Complets

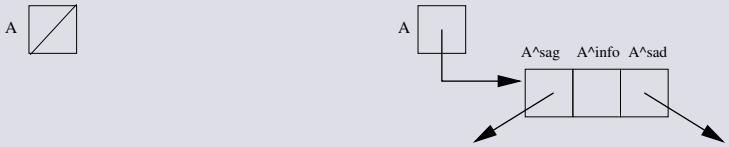
Toutes les feuilles ont même profondeur et seules les feuilles ont un sous-arbre vide.



Structure de Données Arbre Binaire

Un arbre A est :

- soit l'arbre vide représenté par le pointeur `NULL`
- soit un arbre non vide (e, A_1, A_2) représenté par un pointeur vers une cellule contenant :
  - $A \uparrow info$: étiquette de la racine de l'arbre (e)
  - $A \uparrow sag$: sous-arbre gauche (A_1)
  - $A \uparrow sad$: sous-arbre droit (A_2)



Constructeur d'arbre

Algorithme : créerArbre(**d** x : X, **d** A1 : ArbreBin, **d** A2 : ArbreBin) : ArbreBin
Résultat : Renvoie l'arbre dont la racine a pour étiquette x, le sous–arbre gauche est $A1$ et le sous–arbre droit est $A2$

Exemple d'algorithme sur les arbres

Exemple

Algorithme : nbNoeudsArbre(**d** A : ArbreBin) : Entier
Données : *A* est un arbre binaire
Résultat : Renvoie le nombre de noeuds de l'arbre *A*
début

fin algorithme

Parcours d'un Arbre Binaire

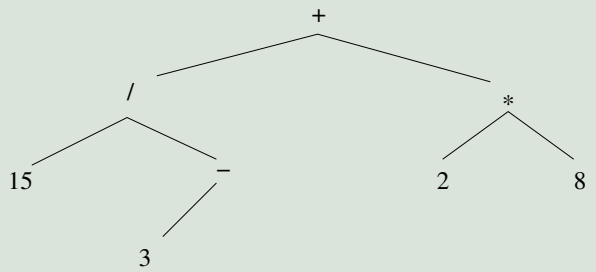
Ordres d'exploration des noeuds d'un Arbre Binaire

Algorithme : Prefixe(A)
début
  si $A \neq NULL$ **alors**
    Traiter(A↑info) ;
    Prefixe(A↑sag) ;
    Prefixe(A↑sad) ;
  fin si
fin algorithme

Algorithme : Infixe(A)
début
  si $A \neq NULL$ **alors**
    Infixe(A↑sag) ;
    Traiter(A↑info) ;
    Infixe(A↑sad) ;
  fin si
fin algorithme

Algorithme : Suffixe(A)
début
  si $A \neq NULL$ **alors**
    Suffixe(A↑sag) ;
    Suffixe(A↑sad) ;
    Traiter(A↑info) ;
  fin si
fin algorithme

Exemple

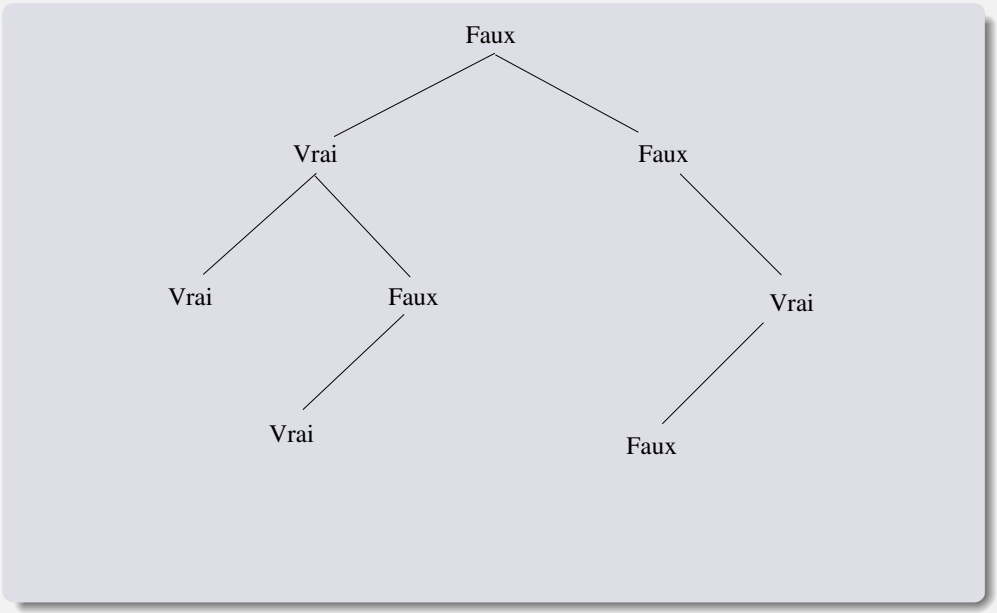


- Le parcours dans l'ordre préfixe est :
- Le parcours dans l'ordre infixe est :
- Le parcours dans l'ordre suffixe est :
- Parcours en largeur (par profondeur croissante) :

Ex d'arbres binaires :arbre préfixe, dictionnaire, tries

Soit un alphabet à 2 lettres *a* et *b*.
On peut représenter un ensemble fini de mots construits avec ces lettres par un arbre binaire étiqueté par des booléens.
L'ensemble de mots représenté par un arbre *A*, noté $Dico(A)$, est défini par :

- $Dico(NULL) = \emptyset$
- si $A \neq NULL$ et $A\uparrow info = Faux$,
  $Dico(A) = \{a.m | m \in Dico(A\uparrow sag)\} \cup \{b.m | m \in Dico(A\uparrow sad)\}$
- si $A \neq NULL$ et $A\uparrow info = Vrai$,
  $Dico(A) = \{\epsilon\} \cup \{a.m | m \in Dico(A\uparrow sag)\} \cup \{b.m | m \in Dico(A\uparrow sad)\}$



Arbres Généraux

Généralisation des arbres binaires : le nombre de sous-arbres peut être supérieur à 2.

Représentation des arbres N-aires

- Extension de la Structure de Données
- Utilisation des arbres binaires