Preuve par récurrence, arbres d'informaticien

Preuve par récurrence

Démonstration par récurrence : prouver une assertion $A(n)$ dépendant d'un entier naturel n
Montrer que

1. cas de base : $A(0)$ est vraie
2. pas de la récurrence (ou hérédité) : $\forall n \in \mathbb{N}$ on a $A(n) \implies A(n+1)$

On peut alors déduire que $A(n)$ est vraie pour tout n.
Pour démontrer l'assertion seulement à partir d'un certain rang, il suffit d'adapter le cas de base.

Preuve par récurrence

Exemple

Soit l'assertion :

$$A(n) : 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Montrons que $A(n)$ est vraie pour tout entier $n \geq 1$.

Preuve par récurrence

Exemple

Soit l'assertion :

$$A(n) : 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Montrons que $A(n)$ est vraie pour tout entier $n \geq 1$.

- $A(1)$ se traduit en $1 = \frac{1 \times 2}{2}$ qui est vraie.

Preuve par récurrence

Exemple

Soit l'assertion :

$$A(n) : 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Montrons que $A(n)$ est vraie pour tout entier $n \geq 1$.

- $A(1)$ se traduit en $1 = \frac{1\times 2}{2}$ qui est vraie.
- Soit $n \in \mathbb{N}^*$ quelconque et supposons $A(n)$ vraie. Alors on a

  $$1+2+\cdots+n+(n+1) = \frac{n(n+1)}{2}+(n+1) = \frac{(n+1)(n+2)}{2}$$

  ce qui est exactement $A(n+1)$. Donc $A(n) \Longrightarrow A(n+1)$

Preuve par récurrence

Exemple

Soit l'assertion :

$$A(n) : 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Montrons que $A(n)$ est vraie pour tout entier $n \geq 1$.

- $A(1)$ se traduit en $1 = \frac{1 \times 2}{2}$ qui est vraie.
- Soit $n \in \mathbb{N}^*$ quelconque et supposons $A(n)$ vraie. Alors on a

$$1+2+\cdots+n+(n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$$

  ce qui est exactement $A(n+1)$. Donc $A(n) \Longrightarrow A(n+1)$
- Par récurrence, on déduit que $A(n)$ est vraie pour tout $n \geq 1$

Preuve par récurrence forte

Pour l'étape d'hérédité où l'on veut prouver que A(n+1) est vraie, au lieu de supposer seulement que $A(n)$ est vraie, on suppose que $A(1), A(2), \ldots, A(n)$ sont vraies.

Exemple

- Considérons une tablette de chocolat standard : un rectangle avec n carrés.
- On peut casser la tablette en deux morceaux rectangulaires plus petits le long d'une rainure et répéter cette procédure sur les plus petits morceaux jusqu'à ce que vous ayez n morceaux.
- Montrez qu'il faut exactement $n - 1$ cassures, quelle que soit la stratégie.

Démonstration

Soit $A(n)$ l'assertion : Il faut exactement $n-1$ cassures pour casser une tablette de n carrés en n morceaux.
On veut montrer que $A(n)$ est vraie pour tout $n \geq 1$.

- <u>Initialisation</u> : $A(1)$ est évident.

Démonstration

Soit $A(n)$ l'assertion : Il faut exactement $n-1$ cassures pour casser une tablette de n carrés en n morceaux.
On veut montrer que $A(n)$ est vraie pour tout $n \geq 1$.

- Initialisation : $A(1)$ est évident.
- Hérédité : supposons que pour $n \geq 1$ quelconque, $A(1), A(2), \ldots, A(n)$ sont vraies. Il faut prouver que $A(n+1)$ est vraie.

Démonstration

Soit $A(n)$ l'assertion : Il faut exactement $n-1$ cassures pour casser une tablette de n carrés en n morceaux.
On veut montrer que $A(n)$ est vraie pour tout $n \geq 1$.

- Initialisation : $A(1)$ est évident.
- Hérédité : supposons que pour $n \geq 1$ quelconque, $A(1), A(2), \ldots, A(n)$ sont vraies. Il faut prouver que $A(n+1)$ est vraie.
  - On considère une tablette de $n+1$ carrés, il faudra commencer par la couper en deux morceaux.

Démonstration

Soit $A(n)$ l'assertion : Il faut exactement $n-1$ cassures pour casser une tablette de n carrés en n morceaux.
On veut montrer que $A(n)$ est vraie pour tout $n \geq 1$.

- Initialisation : $A(1)$ est évident.
- Hérédité : supposons que pour $n \geq 1$ quelconque, $A(1), A(2), \ldots, A(n)$ sont vraies. Il faut prouver que $A(n+1)$ est vraie.
  - On considère une tablette de $n+1$ carrés, il faudra commencer par la couper en deux morceaux.
  - On note n_1 et n_2 les nombres de carrés des deux morceaux (notez que $n_1 + n_2 = n+1$).

Démonstration

Soit $A(n)$ l'assertion : Il faut exactement $n-1$ cassures pour casser une tablette de n carrés en n morceaux.
On veut montrer que $A(n)$ est vraie pour tout $n \geq 1$.

- Initialisation : $A(1)$ est évident.
- Hérédité : supposons que pour $n \geq 1$ quelconque, $A(1), A(2), \ldots, A(n)$ sont vraies. Il faut prouver que $A(n+1)$ est vraie.
  - On considère une tablette de $n+1$ carrés, il faudra commencer par la couper en deux morceaux.
  - On note n_1 et n_2 les nombres de carrés des deux morceaux (notez que $n_1 + n_2 = n+1$).
  - Comme $1 \leq n_1 \leq n$ et $1 \leq n_2 \leq n$, on sait que $A(n_1)$ et $A(n_2)$ sont vraies.

Démonstration

Soit $A(n)$ l'assertion : Il faut exactement $n-1$ cassures pour casser une tablette de n carrés en n morceaux.
On veut montrer que $A(n)$ est vraie pour tout $n \geq 1$.

- Initialisation : $A(1)$ est évident.
- Hérédité : supposons que pour $n \geq 1$ quelconque, $A(1), A(2), \ldots, A(n)$ sont vraies. Il faut prouver que $A(n+1)$ est vraie.
  - On considère une tablette de $n+1$ carrés, il faudra commencer par la couper en deux morceaux.
  - On note n_1 et n_2 les nombres de carrés des deux morceaux (notez que $n_1 + n_2 = n+1$).
  - Comme $1 \leq n_1 \leq n$ et $1 \leq n_2 \leq n$, on sait que $A(n_1)$ et $A(n_2)$ sont vraies.
  - Donc, il faut $(n_1 - 1)$ et $(n_2 - 1)$ cassures pour les deux morceaux, respectivement.

Démonstration - suite

- Hérédité, suite

Démonstration - suite

- Hérédité, suite
  - On conclut qu'il faut $(n_1 - 1) + (n_2 - 1) + 1 = (n_1 + n_2) - 1 = (n + 1) - 1 = n$ cassures pour la tablette originelle.

Démonstration - suite

- Hérédité, suite
  - On conclut qu'il faut $(n_1 - 1) + (n_2 - 1) + 1 = (n_1 + n_2) - 1 = (n + 1) - 1 = n$ cassures pour la tablette originelle.
- Conclusion : $A(n)$ est vraie pour tout $n \geq 1$, c'est-à-dire qu'il faut exactement $n - 1$ cassures pour couper en n morceaux une tablette de n carrés de chocolat.

Méthode pour prouver le pas de récurrence

- On suppose que $A(n)$ est vraie (récurrence faible) ou que $A(1), \dots, A(n)$ sont vraies (récurrence forte) pour un n quelconque.
- On prend une instance (une somme, une tablette de chocolat ...) avec paramètre $n+1$.
- On décompose cette instance en plus petites instances.
- On applique l'hypothèse de récurrence aux petites instances.
- On déduit que $A(n+1)$ est vraie.

Attention

Il ne faut pas commencer par une instance avec paramètre n, et construire une instance avec paramètre $n+1$.

Quizz

Question 1 On suppose qu'une grenouille est devant une suite infinie de nénuphars numérotés avec les entiers naturels 0, 1, 2 ..., elle se trouve sur le nénuphar numéro 0. Elle peut seulement faire des sauts de 3 en 3 (de i à $i+3$) ou de 5 en 5 (de i à $i+5$), et pas de retour en arrière possible. Quels sont les nénuphars accessibles pour elle ?

1. Tous
2. Aucun sauf le nénuphar 0
3. Tous les multiples de 3 ou de 5
4. Tous les nénuphars sauf 1, 2, 4, 7
5. Autre

Quizz

$P(n)$: la position n est accessible pour la grenouille

Montrons par récurrence forte sur n que la propriété $P(n)$ est vraie pour tout $n \geq 8$.
Initialisation : $P(8)$ est vraie car...

Hérédité : soit $n \geq 8$, on suppose que $P(8), \ldots, P(n)$ sont vraies.
Question 2 Quelle est la suite de la preuve ?

1. Supposons que la grenouille est sur la position n et montrons qu'elle peut aller sur la position $n+1$.
2. Supposons que la grenouille est sur la position $n+1$.
3. Montrons que la grenouille peut aller sur la position $n+1$.
4. Autre

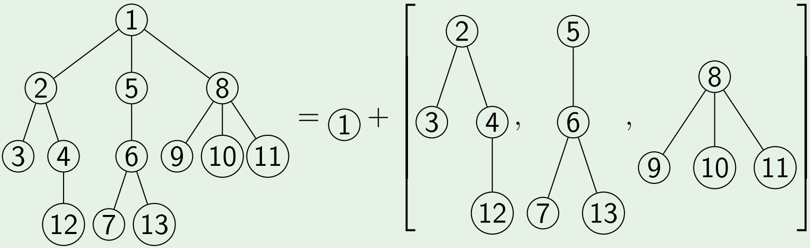
Arbre d'informaticien

Définition

Un arbre est composé d'une racine et d'une liste[1](éventuellement vide) d'arbres.

Un arbre trivial est un arbre qui consiste d'une racine (liste vide)

Exemple



1. Pour l'instant : liste = ensemble ordonné d'éléments

Arbre d'informaticien

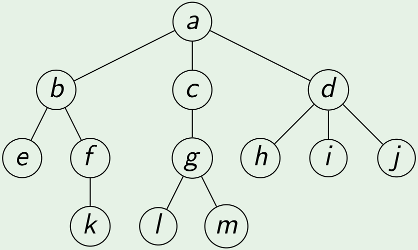
La liste définit un ordre sur les enfants.

Racine : nœud qui n'est enfant de personne (qui n'a pas de parent)
Feuille : nœud qui n'a pas d'enfants.

La *profondeur* d'un noeud est la distance du noeud à la racine (nombre de traits).
La *profondeur (ou hauteur)* d'un arbre est la plus grande distance de la racine à un nœud.
La *largeur* d'un arbre est le nombre maximal de noeuds qu'il y a sur un niveau.

Arbre enraciné

Exemple



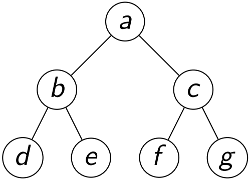
- Si la racine est a,
  - quelle est la hauteur de l'arbre ?
  - quels sommets sont les fils de a ?
  - quel sommet est le père de c ?
- Cas général :
  - Quelle racine permet d'avoir la plus grande hauteur ?
  - Est-ce que les feuilles dépendent de la racine ?

Arbre enraciné

Exemples d'arbres enracinés

- Structure hiérarchique
- Ascendants d'une personne (arbre binaire...)
- Descendants d'une personne
- Expression arithmétique
- Système de fichiers
- Configurations du jeu du morpion, puissance 4, dames, échecs... (arbre de décision)

Arbres k-aires

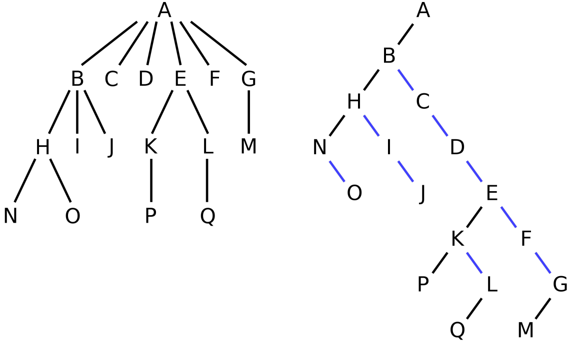


Définition

- Un arbre est k-**aire** si tout sommet a au plus k fils.
- Si $k = 2$: **arbre binaire**.
  - 2 emplacements pour les fils (droit et gauche) qui peuvent contenir un arbre ou être vides.

Encoder un arbre quelconque en un arbre binaire

- On peut encoder tout arbre T en un arbre binaire T_B.
- Pour chaque noeud N de T, on crée un noeud N_B pour T_B, qui doit avoir deux fils :
  - le fils gauche correspond au premier fils de N
  - le fils droit au frère droit de N.
- On continue de manière récursive.



Parcours des arbres

Parcours d'un arbre : façon d'en ordonner les nœuds
Un algo de parcours des arbres :

- pré-traitement de la racine
- parcourir récursivement chacun des enfants
- post-traitement de la racine

Le parcours est appelé sur l'arbre complet
On supposera implicitement que les fils d'un nœud v_i sont ordonnés $v_{i,1}, \ldots, v_{i,k_i}$ et que cet ordre est connu et fixé une fois pour toutes

Parcours des arbres

Parcours d'un arbre : façon d'en ordonner les nœuds
Un algo de parcours des arbres :

- pré-traitement de la racine
- parcourir récursivement chacun des enfants
- post-traitement de la racine

Le parcours est appelé sur l'arbre complet
On supposera implicitement que les fils d'un nœud v_i sont ordonnés $v_{i,1}, \dots, v_{i,k_i}$ et que cet ordre est connu et fixé une fois pour toutes

- Parcours en profondeur (DFS : *depth first search*)
- Parcours en largeur (BFS : *breadth first search*)

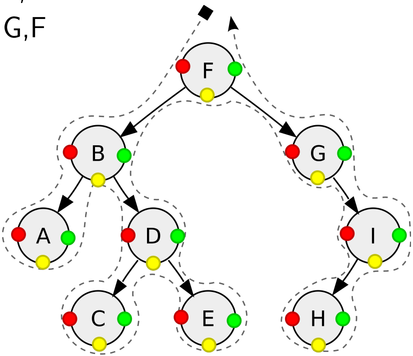
Parcours en profondeur (DFS)

Trois types de **parcours en profondeur** : les parcours préfixe, infixe et suffixe

- Pour le parcours **préfixe**, on parcourt d'abord la racine v_0 puis on parcourt, de manière récursive et dans l'ordre, les sous-arbres de racine respective $v_{0,1}, \ldots, v_{0,k_0}$.
- Pour le parcours **suffixe** (ou **postfixe**), on parcourt d'abord, de manière récursive et dans l'ordre, les sous-arbres de racine $v_{0,1}, \ldots, v_{0,k_0}$, puis la racine v_0.
- Pour le parcours **infixe**, nous supposerons disposer d'un arbre binaire (chaque nœud a au plus deux fils). (On peut donc parler du sous-arbre de gauche et du sous-arbre de droite.) On parcourt d'abord, de manière récursive, le sous-arbre de gauche, puis la racine, et enfin le sous-arbre de droite.

Illustration des trois parcours en profondeur

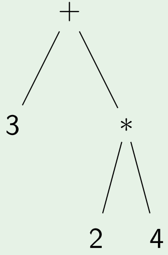
- préfixe : F,B,A,D,C,E,G,I,H
- infixe : A,B,C,D,E,F,G,H,I
- suffixe : A,C,E,D,B,H,I,G,F



Expressions arithmétiques

Exemple

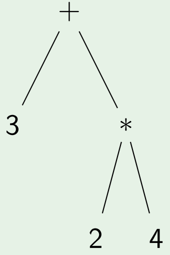
Donnez les expressions arithmétiques correspondant aux parcours infixe, préfixe et postfixe de l'arbre suivant.



Expressions arithmétiques

Exemple

Donnez les expressions arithmétiques correspondant aux parcours infixe, préfixe et postfixe de l'arbre suivant.

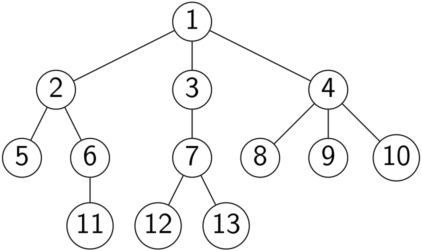


- infixe : $3 + 2 * 4$
- préfixe : $(+\ 3\ (*\ 2\ 4))$ *(notation polonaise)*
- postfixe : $3\ 2\ 4\ *\ +$ *(reverse polish, lisp)*

Parcours en largeur (BFS)

Parcours en largeur

- Respect des générations.
- On parcourt les nœuds par profondeur croissante.



Quand utiliser le parcours en largeur ?

- Imaginez que l'arbre décrit le plan d'un labyrinthe : l'entrée est à la racine, et quand vous êtes dans la salle correspondant à un nœud, vous pouvez vous rendre dans les salles enfant (ou remonter dans la salle parent).
- Certains nœuds contiennent des trésors.

Question

Quel algorithme allez-vous utiliser pour trouver le trésor le plus proche de l'entrée (qui est aussi la sortie) ?

Quand utiliser le parcours en largeur ?

- Il faut utiliser un parcours en largeur : il va visiter les cases les plus proches de la racine en premier.
- Dès que vous aurez trouvé un trésor, vous savez que c'est le trésor le plus proche de l'entrée, ou en tout cas un des trésors les plus proches : il y a peut-être d'autres trésors dans la même couche.
- Un parcours en profondeur ne permet pas cela : le premier trésor qu'il trouve peut être très profond dans l'arbre, très loin dans la racine.

Parcours

Quel parcours utiliser ?

- la liste des ...
- le plus court chemin qui ...
- le nombre total de ...
- le nœud le plus proche qui ...
- la liste des ... en ordre de distance croissante
- le plus long chemin qui ...

Parcours

Quel parcours utiliser ?

parcours en largeur

- le plus court chemin qui ...
- le nœud le plus proche qui ...
- la liste des ... en ordre de distance croissante

parcours en profondeur

- le plus long chemin qui ...

les deux...

- la liste des ...
- le nombre total de ...

Liste, pile, file

Principe :

- On rajoute un élément au début de la liste

Liste, pile, file

Principe :

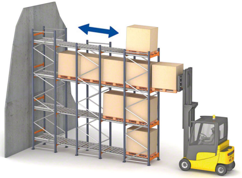
- On rajoute un élément au début de la liste

- On retire l'élément en fin de liste $\rightarrow$ **file**

(ex : queue au supermarché, file d'attente)
premier entré premier sorti,
FIFO, first in first out

- On retire l'élément en début de la liste $\rightarrow$ **pile**

(ex : pile de copies à corriger, pile d'assiettes)
dernier entré premier sorti,
LIFO, last in first out, *stack*



Listes

Interface file

- Structure de données : liste
- Fonctions de manipulation :
  - `isempty` : file $\rightarrow$ booléen
    renvoie si la file est vide ou non
  - `push` : (file, element) $\rightarrow$ vide
    met un element en queue de la file (ne renvoie rien)
  - `pop` : (file) $\rightarrow$ element
    vide la file de l'élément au sommet et renvoie cet élément

Listes

Interface pile

- Structure de données : liste
- Fonctions de manipulation :
  - `isempty` : pile $\rightarrow$ booléen
    renvoie si la pile est vide ou non
  - `push` : (pile, element) $\rightarrow$ vide
    met un element sur la tête de la pile (ne renvoie rien)
  - `pop` : (pile) $\rightarrow$ element
    vide la pile de l'élément au sommet et renvoie cet élément

Parcours

Algorithme générique itératif

```
parcourir(Arbre T){
    créer une liste vide s
    s.add(racine de T)
    tant que non s.isEmpty() {
        t = s.remove();
        traiter t
        pour chaque fils f de t
            s.add(f)
    }
}
```

Parcours

Algorithme générique itératif

```
parcourir(Arbre T){
    créer une liste vide s
    s.add(racine de T)
    tant que non s.isEmpty() {
        t = s.remove();
        traiter t
        pour chaque fils f de t
            s.add(f)
    }
}
```

- Si s est une pile : parcours en profondeur (préfixe) (attention à l'ordre de traitement des fils)
- Si s est une file : parcours en largeur.

Parcours

Algorithme générique itératif

```
parcourir(Arbre T){
    créer une liste vide s
    s.add(racine de T)
    tant que non s.isEmpty() {
        t = s.remove();
        traiter t
        pour chaque fils f de t
            s.add(f)
    }
}
```

- Si s est une pile : parcours en profondeur (préfixe) (attention à l'ordre de traitement des fils)
- Si s est une file : parcours en largeur.

Pour un parcours en profondeur, on privilégie une méthode récursive !

ParcoursLargeur(T, r **)**

Données : un arbre T enraciné au sommet r

Résultat : affiche les sommets de l'arbre dans l'ordre d'un parcours en largeur

```
F ← File()
F.enfile(r)
tant que F n'est pas vide faire
    v ← F.pop()
    Afficher v
    pour tout fils u de v faire
        F.enfile(u)
```

Parcours sur les arbres enracinés

ParcoursProfondeur(T, r**)**

Données : un arbre T enraciné au sommet r
Résultat : affiche les sommets de l'arbre dans l'ordre d'un parcours en profondeur (préfixe)

$P \leftarrow Pile()$
P.empile(r)
tant que *P n'est pas vide* **faire**
    $v \leftarrow P.pop()$
    Afficher v
    # attention à l'ordre pour l'itération ci-dessous
    pour *tout fils u de v* **faire**
        P.empile(u)

Parcours sur les arbres enracinés

ParcoursProfondeurRec(T**,** v**)**

Données : un arbre T, un noeud v
Résultat : affiche les sommets du sous-arbre de T enraciné en v dans l'ordre d'un parcours en profondeur (préfixe)

Afficher v
pour *tout fils* u *de* v **faire**
  ParcoursProfondeurRec(T, u)