Algorithmique (en C)

Julien Alexandre dit Sandretto

Department U2IS
ENSTA ParisTech

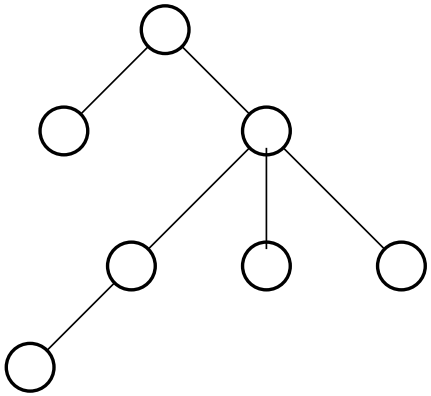
Version 2019-2020

ENSTA

IP PARIS

Les arbres

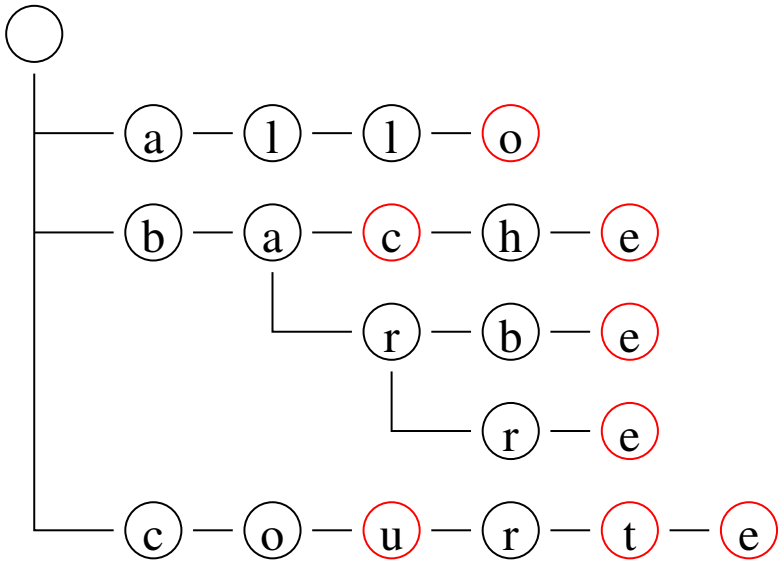
Arbre ?



- Structure **arborescente** : forme d'un . . . arbre !
- Des nœuds, des arcs.
- Pas de « *boucle* ».
- Pas de « *raccourci* ».

Exemple : le dictionnaire

- Permet le partage de préfixes communs.
- Puisque mots préfixes d'autres, nécessité de marquer les fins de mots.
- Exemple : { allo, bac, bache, barbe, barre, cou, court, courte }.



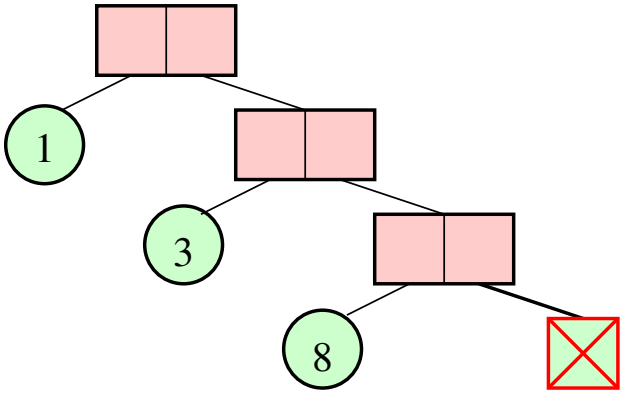
Exemple : le système de fichiers (simple)

```
/
|-- bin
|   |-- alsaunmute
|   |-- arch
|   |-- awk
|   `-- zcat
|-- boot
|   |-- config-2.6.40.6-0.fc15.i686
|   |-- grub
|   |   |-- device.map
|   |   |-- e2fs_stage1_5
|   |   |-- ufs2_stage1_5
|   |   |-- vstafs_stage1_5
|   |   `-- xfs_stage1_5
|   |-- initramfs-2.6.40.6-0.fc15.i686.img
|   `-- vmlinuz-2.6.43.2-6.fc15.i686
|-- cgroup
|-- dev
|   |-- autofs
```

- Racine : /
- Noeuds internes : répertoires.
- Feuilles : fichiers.
- (On ne considère pas les « raccourcis » – ou « liens »).

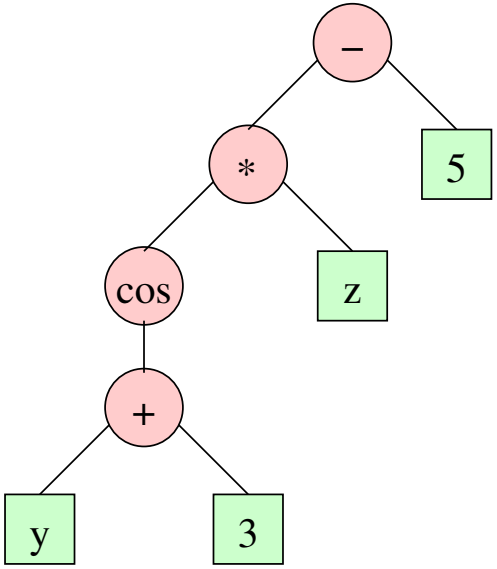
Exemple : les listes

- [1;3;8]
- Type inductif :
  - Liste vide.
  - Élément suivi d'une liste.
- Longueur de la liste : hauteur de l'arbre.



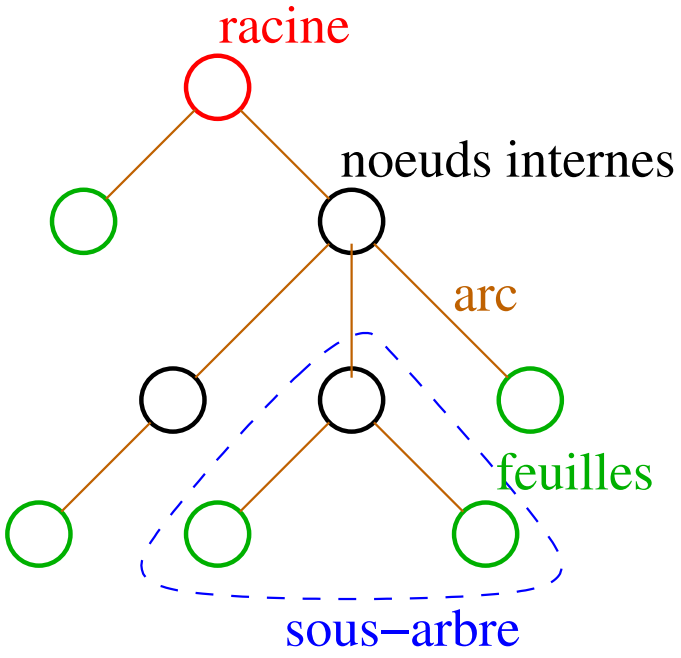
Exemple : les arbres de syntaxe

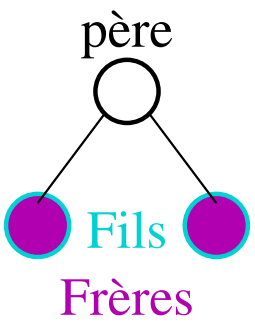
- Expressions arithmétiques :
  - Constantes.
  - Opérateurs binaires entre **expressions arithmétiques**.
  - Appel de fonction avec des **expressions arithmétiques** en arguments.
  - Variables.
- Exemple : $cos(y+3) \times z - 5$
- Permet de calculer sur les expressions :
  - évaluation,
  - dérivation,
  - simplification,
  - compilation...



Vocabulaire (1)

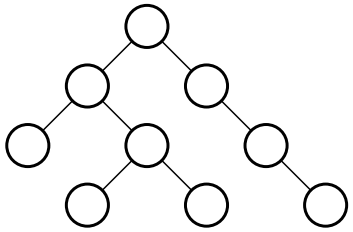
- Nœud (« *node* ») :
  - Racine (« *root* »)
  - Nœud interne
  - Feuille (« *leaf* »)
- Arc (« *edge* »)
- Sous-arbre (« *subtree* »)
- Parenté :
  - Père (« *parent* »)
  - Fils (« *child* »)
  - Frère (« *sibling* »)



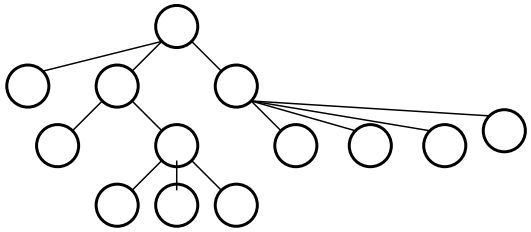


Vocabulaire (2)

- **Degré d'un nœud** : nombre de fils.
- **Arité d'un arbre** : nombre **max** de fils.
  - Certains nœuds peuvent en avoir moins.
- Arbre **binaire** : arité 2.

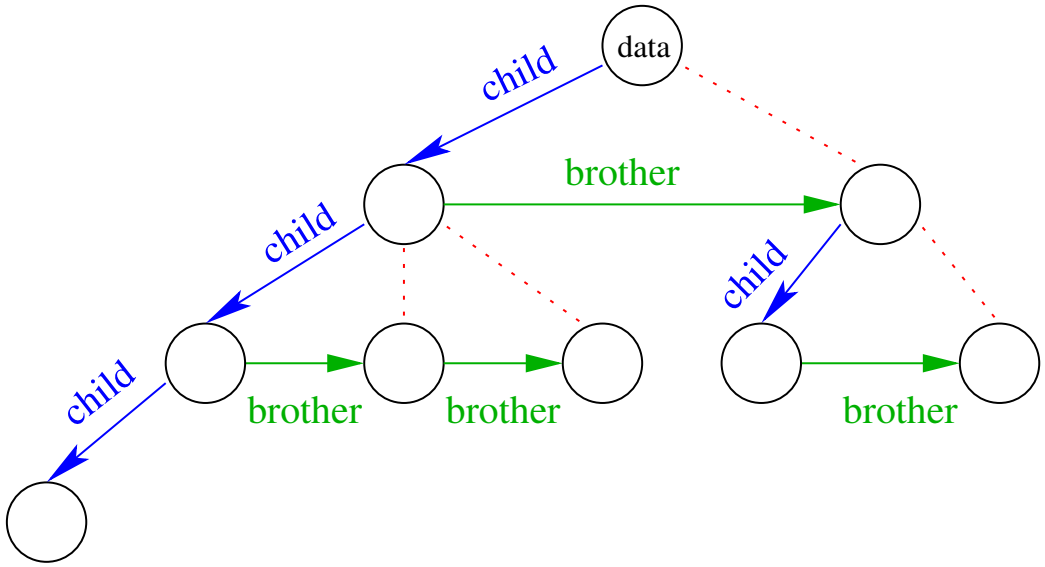


- Sinon, arbre **n-aire**.



Implémentation par pointeurs

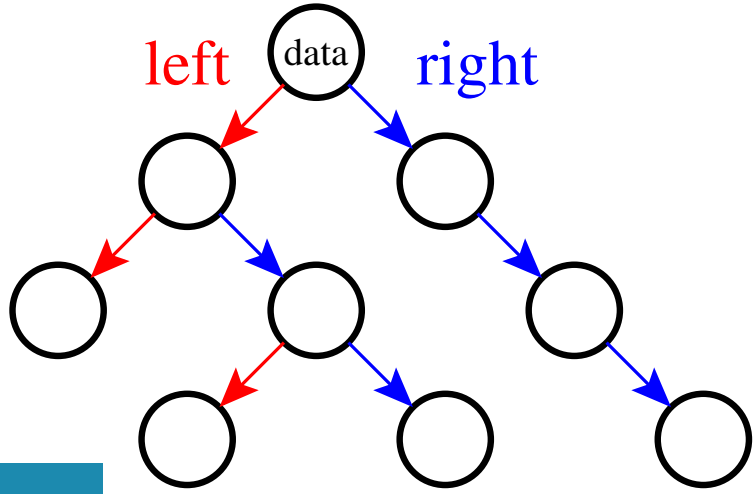
- Cas général : nombre quelconque de fils :
  - **Liste chaînée** de tous les fils (frères).
  - Le parent a un **pointeur** sur son 1er fils.




```
struct node {
  int data ;
  struct node *child ;
  struct node *brother ;
};
```

Arbres binaires

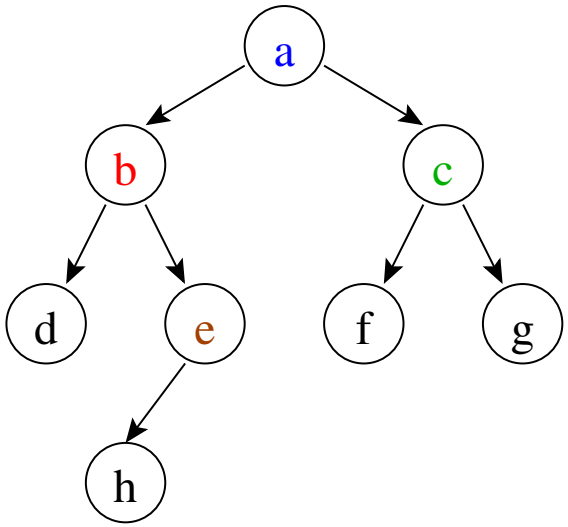
- Un nœud a **au plus 2** fils :
  - 1 fils « *droit* »
  - 1 fils « *gauche* »
- Simples mais nombreuses applications :
  - Codage de Huffman.
  - Calcul de la complexité optimale du tri.
  - Recherche sur un ensemble ordonné.
  - Satisfiabilité de formules booléennes (Binary Decision Diagram) . . .

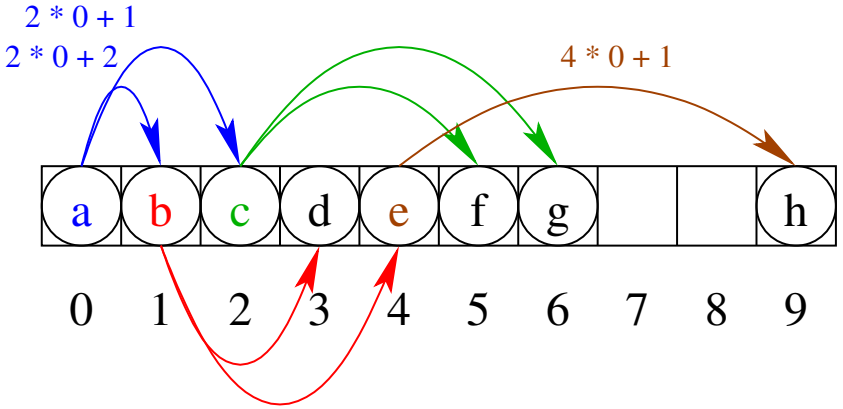



```
struct bnode {
  int data ;
  struct bnode *left ;
  struct bnode *right ;
};
```

Implémentation par tableau

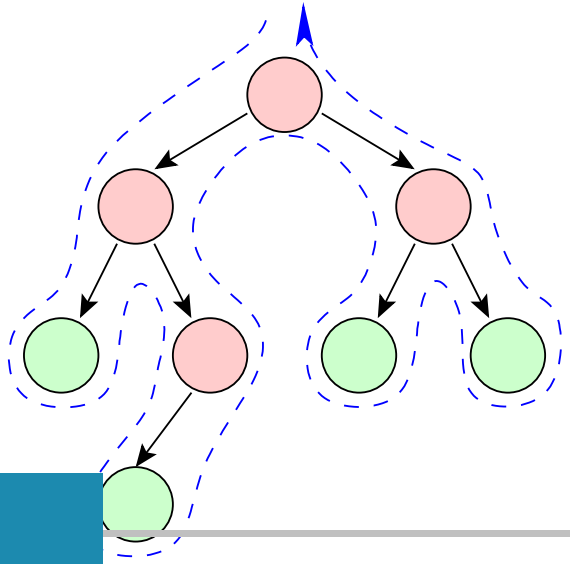
- Un nœud a un indice i.
- Ses **fils** sont aux indices $2i+1$ et $2i+2$.
- Son **père** est à l'indice $floor((i-1)/2)$.
- $\Rightarrow$ Pas besoin de pointeurs.
- $\Rightarrow$ Si arbre binaire **complet**, pas de perte de place.





Arbres binaires : parcours en profondeur

- En Anglais : « *Depth First Search* » (DFS).
- Descente **au plus profond** en commençant par le fils gauche (ou droit).
- Parcours récursif :
  - À chaque nœud, on applique la descente sur le fils gauche.
  - Une fois le sous-arbre gauche exploré, on explore le fils droite.
  - Une fois le sous-arbre droite exploré, on remonte et applique le parcours.
  - Fin : fin de l'exploration du sous-arbre droite.



```
void dfs (struct b_node *n)
{
  if (n != NULL) {
    dsf (n->left) ;
    dsf (n->right) ;
}
```

Ici, on a fait que parcourir, aucun traitement...

Arbres binaires : parcours en profondeur préfixe

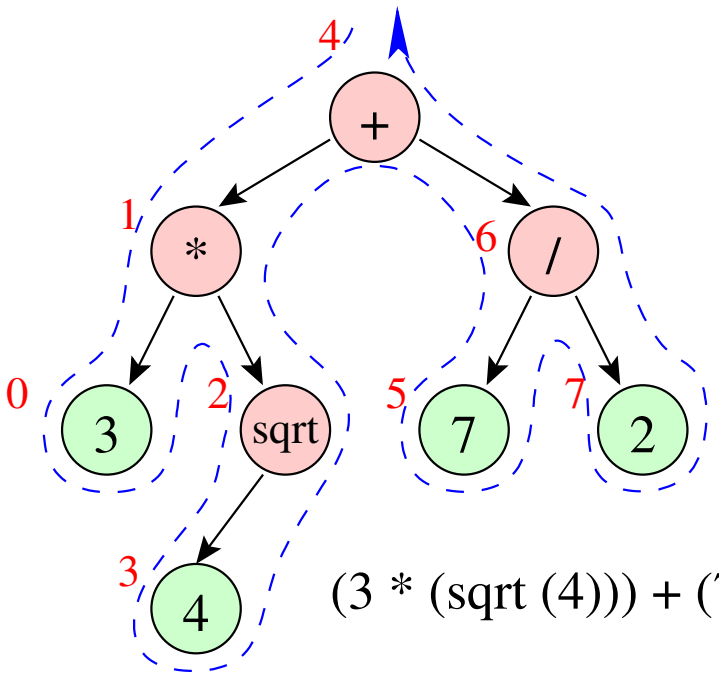
- 3 endroits pour insérer du traitement du nœud courant.
  - Au début.

```
void prefix (b_node *n) {
  if (n != NULL) {
    do_something (n->data) ;
    prefix (n->left) ;
    prefix (n->right) ;
  }
}
```

Arbres binaires : parcours en profondeur infixe

- 3 endroits pour insérer du traitement du nœud courant.
  - Au milieu.
  - Si `do_something` est l'impression : correspond à l'écriture *« normale »* d'une expression arithmétique.

```
void infix (struct b_node *n)
{
  if (n != NULL) {
    infix (n->left) ;
    do_something (n->data) ;
    infix (n->right) ;
  }
}
```

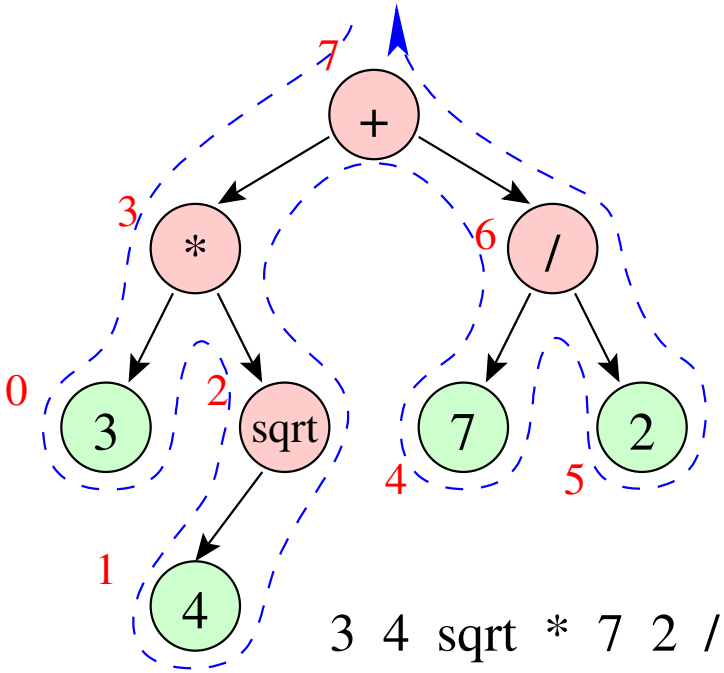


(3 * (sqrt (4))) + (7 / 2) (= 9.5)

Arbres binaires : parcours en profondeur postfixe

- 3 endroits pour insérer du traitement du nœud courant.
  - À la fin.
  - Si `do_something` est l'impression : correspond à l'écriture RPN (polonaise inverse).

```
void postfix (struct b_node *n)
{
  if (n != NULL) {
    postfix (n->left) ;
    postfix (n->right) ;
    do_something (n->val) ;
  }
}
```



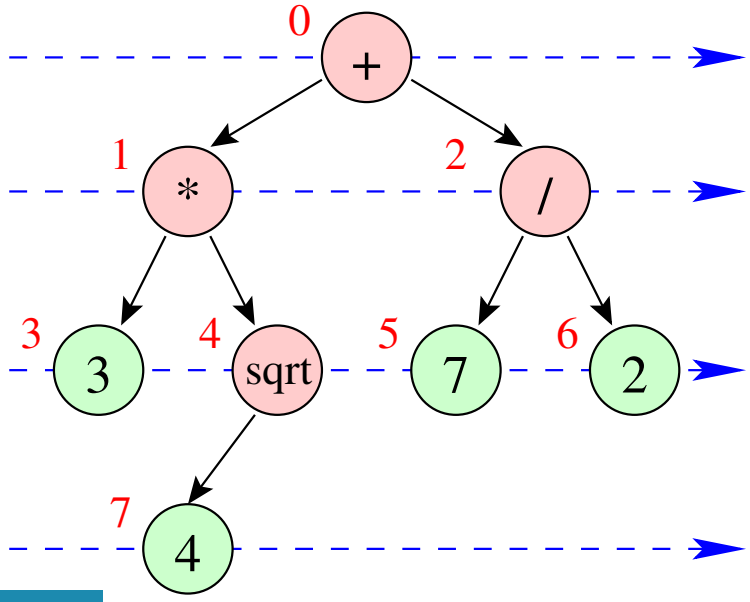
3 4 sqrt * 7 2 / + (= 9.5)

Arbres binaires : parcours en profondeur, applications

- Recherche de composantes connexes.
- Tri topologique.
- Recherche de points dans un graphe.
- Recherche de composantes fortement connexes.
- Test de « planarité » d'un graphe.
- Recherche de solution de jeux (labyrinthe par exemple).
- Génération de labyrinthe (DFS aléatoire).
- Etc. . .

Arbres binaires : parcours en largeur

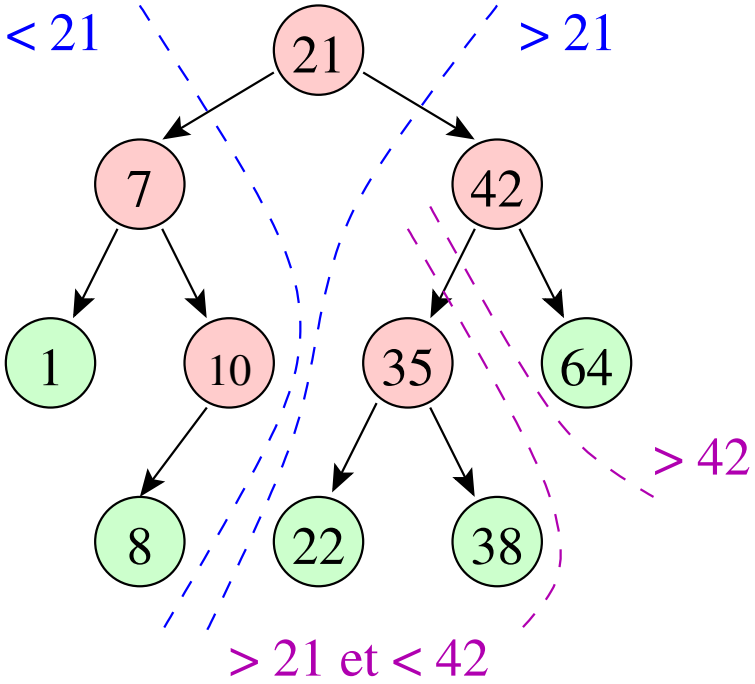
- En Anglais : « *Breadth First Search* » (BFS).
- Parcours « *par niveaux* » (croissants).
- On visite d'abord tous les nœuds de **même profondeur**.
- Utilisation d'une **file** :
  - On extrait un nœud, on le traite, on insère tous ses fils.




```
void largeur (struct b_node *n)
{
  enqueue (n) ;
  while ((n = take ()) != NULL){
    do_something (n->data);
    if (n->left != NULL)
      enqueue (n->left) ;
    if (n->right != NULL)
      enqueue (n->right) ;
   }
}
```

Arbres binaires de recherche

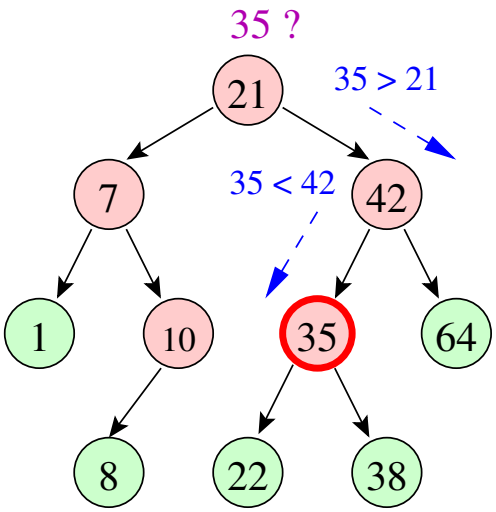
- Structure d'arbre binaire telle que :
  - Chaque nœud contient une **clef**.
  - La clef d'un nœud est > à celle de son fils gauche.
    - Inductivement > à celles du sous-arbre gauche.
  - La clef d'un nœud est < à celle de son fils droite.
    - Inductivement < à celles du sous-arbre droite.

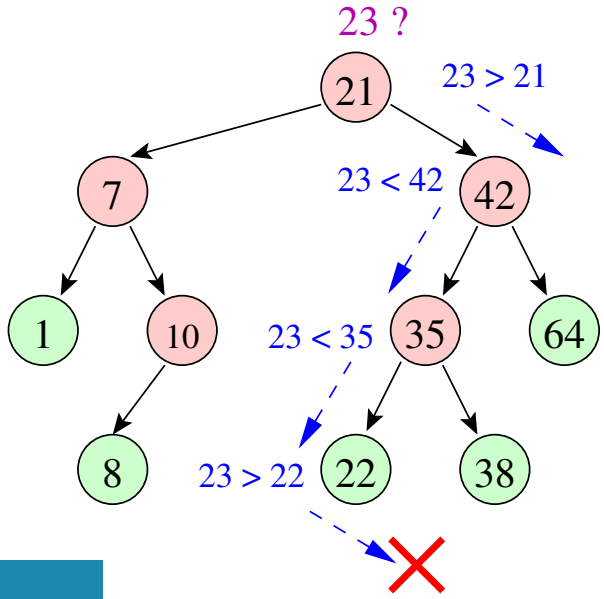


Arbres binaires de recherche : intérêts

- Stockage en $\Theta(n)$.
- Recherche en $\Theta(h)$ (h hauteur de l'arbre).
- Insertion en $\Theta(h)$.
- Suppression en $\Theta(h)$.
- En moyenne $h = \Theta(log(n))$: $\rightarrow$ bonne solution au problème de recherche en table.
- Dans le pire cas $h = n$: $\rightarrow$ intérêt d'avoir des arbres **équilibrés**.
- Par rapport aux tables de hachage :
  - Pas besoin de connaître n à l'avance.
  - Pas de mémoire gaspillée.
  - Les ABR équilibrés sont une solution efficace dans le **pire** cas.

Arbres binaires de recherche : recherche



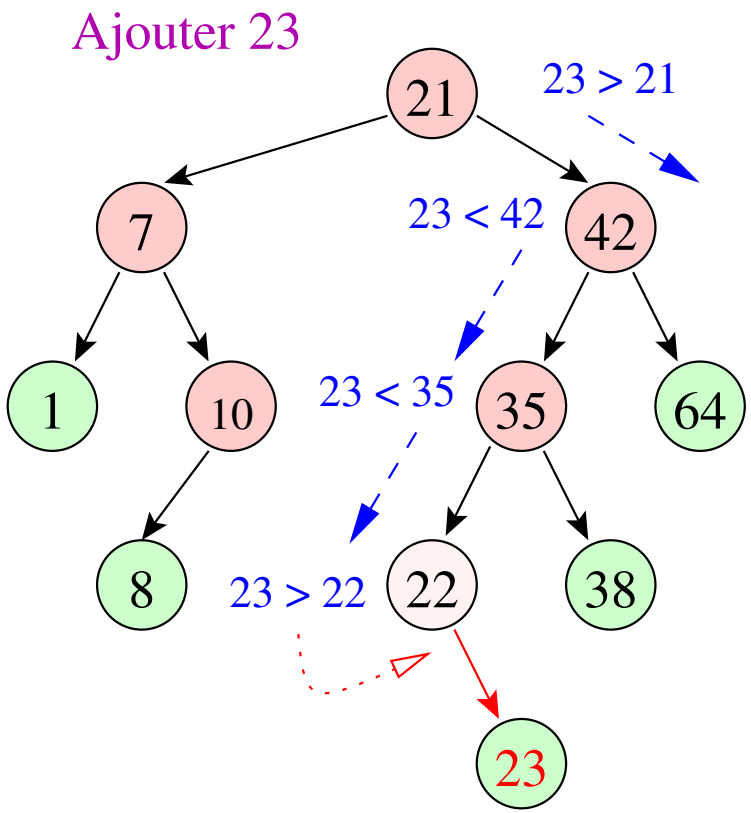



```
#include <stdbool.h>

bool search (int v, struct b_node *n)
{
  if (n == NULL) return (false) ;
  if (v == n->data) return (true) ;
  if (v < n->data)
    return (search (v, n->left)) ;
  else
    return (search (v, n->right)) ;
}
```

Arbres binaires de recherche : insertion (version naïve)

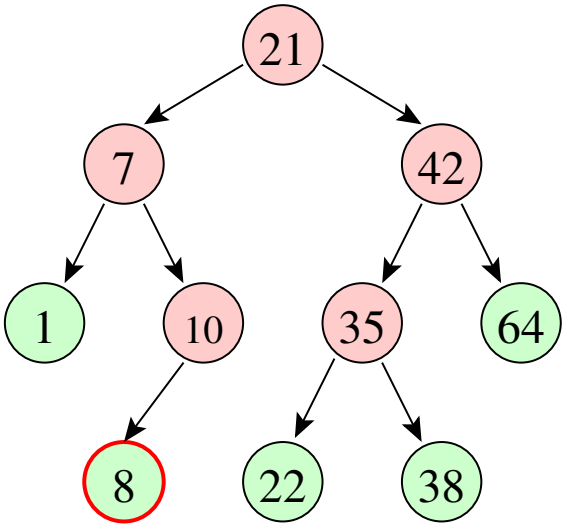
- Rechercher la position du nœud dans l'arbre.
- L'ajouter comme fil du dernier nœud rencontré.

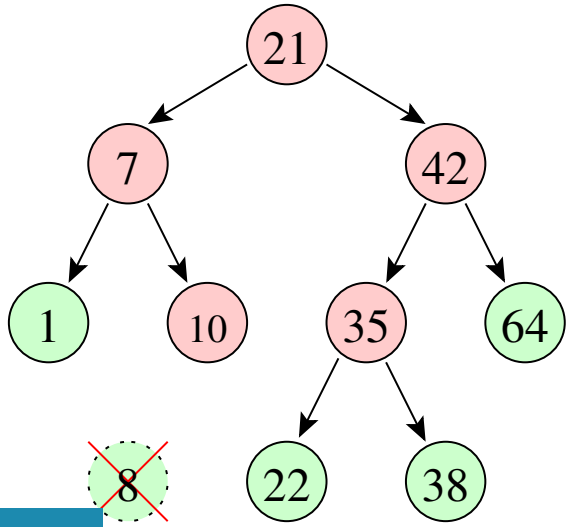


Arbres binaires de recherche : suppression

- On commence par rechercher le nœud à supprimer dans l'arbre.
- 3 cas dont 2 simples :
  - Suppression d'une feuille (→ facile).
  - Suppression d'un nœud avec 1 seul fils (→ facile).
  - Suppression d'un nœud avec 2 fils (→ plus compliqué).

ABR, suppression : 1 feuille

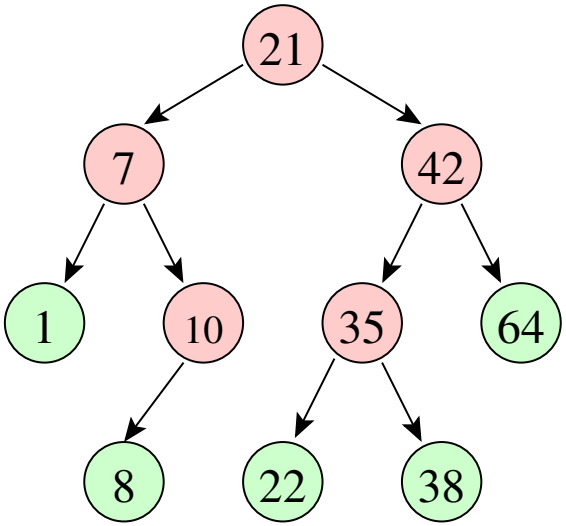


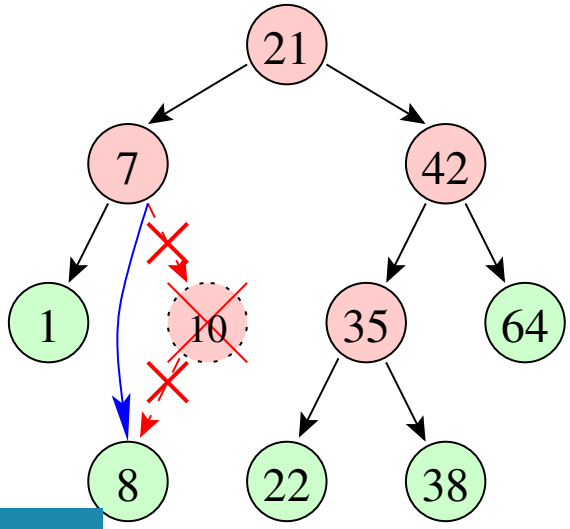


Suppression du nœud 8 . . .

- Mettre le pointeur droit du nœud 10 à `NULL`.
- Libérer la mémoire occupée par le nœud 8.

ABR, suppression : 1 seul fils

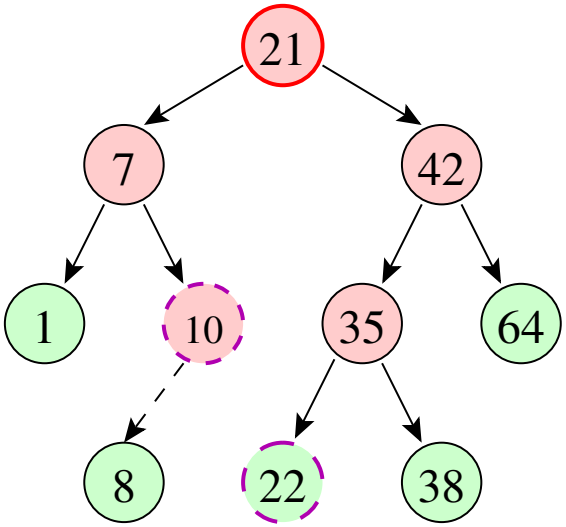


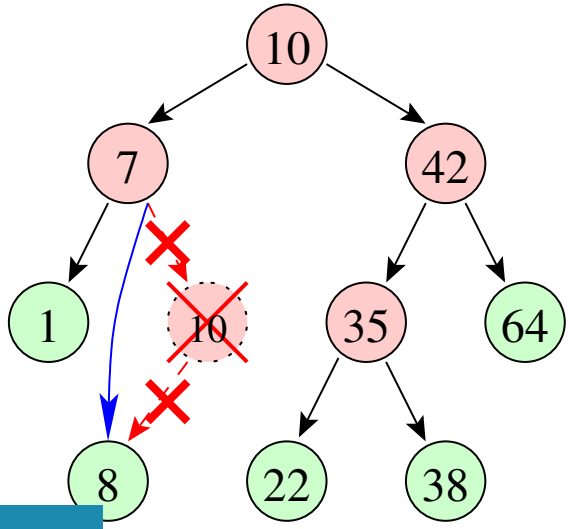


Suppression du nœud 10 . . .

- Faire pointer son père (7) vers son fils (8).
- Libérer la mémoire occupée par le nœud 10.

ABR, suppression : 2 fils





Suppression du nœud 21 ...

- Remplacement par le min du fils droit ou le max du fils gauche.
  - Successeur ou prédécesseur le plus proche.
- Choisir un candidat.
- On recopie sa clef et ses données pour remplacer le nœud 21.
- Ensuite, comme précédemment avec 1 fils.

Pour la culture . . . d'arbres. . .

- Nombreuses autres formes d'arbres de recherche :
  - AVL trees.
  - Red-Black trees.
  - Patricia trees (radix trees).
  - Splay trees.
  - Etc. . .