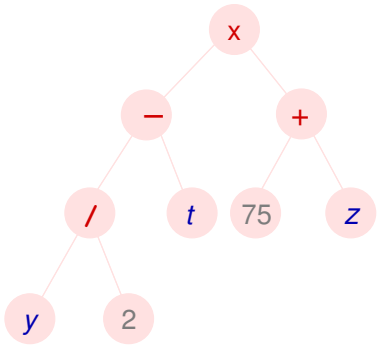
Chapitre 1

Arbres binaires de recherche

[1]Les arbre sont très utilisés en informatique, d'une part parce que les informations sont souvent hiérarchisées, et peuvent être représentées naturellement sous une forme arborescente, et d'autre part, parce que les structures de données arborescentes permettent de stocker des données volumineuses de façon que leur accès soit efficace.

1.1 Quelques exemples de données arborescentes

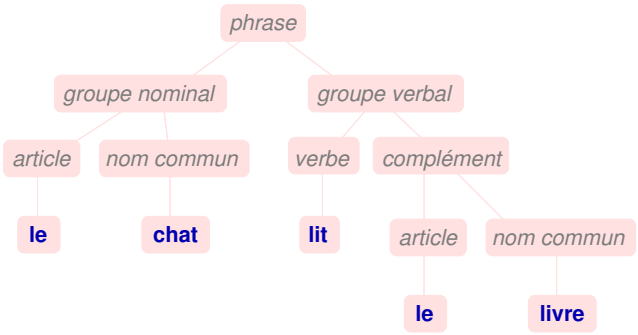
Expressions arithmétiques. On peut représenter les *expressions arithmétiques* par des arbres étiquetés par des *opérateurs*, des *constantes* et des *variables*. La structure de l'arbre rend compte de la priorité des opérateurs et rend inutile tout parenthésage.



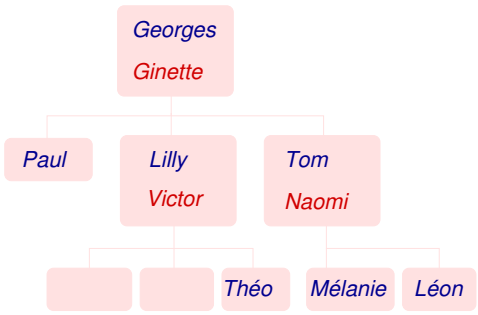
1. Ce chapitre reprend en quasi totalité le chapitre de même nom figurant dans le polycopié du cours d'Algorithmique des L2 d'informatique et de mathématiques de l'université d'Aix Marseille, rédigé par F. Denis, S. Grandcolas et Y. Vaxès.

Exercice : Dessinez l'arbre correspondant à l'expression $3(2x-1)+1$.

Arbres syntaxiques. Un *arbre syntaxique* représente l'analyse d'une phrase à partir d'un ensemble de règles qui constitue la *grammaire* : une *phrase* est composée d'un *groupe nominal* suivi d'un *groupe verba*l, un groupe nominal peut-être constitué d'un *article* et d'un *nom commun*,...

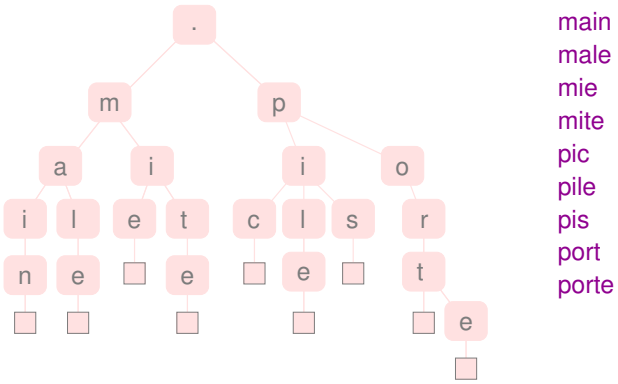


Arbres généalogiques. Un *arbre généalogique* (descendant dans le cas présent) représente la descendance d'une personne ou d'un couple. Les noeuds de l'arbre sont étiquetés par les membres de la famille et leurs conjoints. L'arborescence est construite à partir des liens de parenté (les enfants du couple).



Arbre lexicographique. Un *arbre lexicographique*, ou arbre en parties communes, ou dictionnaire, représente un ensemble de mots. Les préfixes communs à plusieurs mots apparaissent une seule fois dans l'arbre, ce qui se traduit par un gain d'espace mémoire. De plus la recherche d'un mot est assez efficace, puisqu'il suffit de parcourir une branche de l'arbre en partant

de la racine, en cherchant à chaque niveau parmi les fils du noeud courant la lettre du mot de rang correspondant.

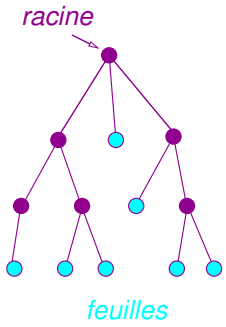


Exercice : Rajoutez le mot `mal`.

1.2 Définitions et terminologie

Définition. Un arbre est un ensemble organisé de noeuds dans lequel chaque noeud a un *père* et un seul, sauf un noeud que l'on appelle la *racine*.

Si le noeud p est le père du noeud f, nous dirons que f est un *fils* de p, et si le noeud p n'a pas de fils nous dirons que c'est une *feuille*. Chaque noeud porte une *étiquette* ou *valeur* ou *clé*. On a l'habitude, lorsqu'on dessine un arbre, de le représenter avec la tête en bas, c'est-à-dire que la racine est tout en haut, et les noeuds fils sont représentés en-dessous du noeud père.



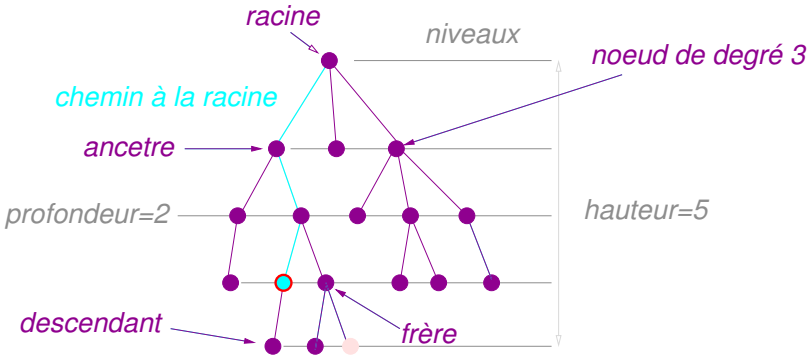
Un noeud est défini par son *étiquette* et ses *sous-arbres*. On peut donc représenter un arbre par un n-uplet $\langle e, a_1, \ldots, a_k \rangle$ dans lesquel e est l'étiquette portée par le noeud, et $a_1, \ldots, a_k$ sont ses sous-arbres. Par exemple l'arbre correspondant à l'expression arithmétique $(y/2-t)\times(75+z)$ sera représenté par

$$\langle \times, \langle -, \langle /, \langle y \rangle, \langle 2 \rangle \rangle, \langle t \rangle \rangle, \langle +, \langle 75 \rangle, \langle z \rangle \rangle \rangle$$

On distingue les *arbres binaires* des arbres généraux. Leur particularité est que les fils sont singularisés : chaque noeud a un fils gauche et un fils droit. L'un comme l'autre peut être un arbre vide, que l'on notera $\langle\rangle$. L'écriture d'un arbre s'en trouve modifiée, puisqu'un noeud a toujours deux fils. En reprenant l'expression arithmétique précédente, l'arbre binaire qui la représente s'écrit

$$\langle \times, \langle -, \langle /, \langle y, \langle\rangle, \langle\rangle \rangle, \langle 2, \langle\rangle, \langle\rangle \rangle \rangle, \langle t, \langle\rangle, \langle\rangle \rangle \rangle, \langle +, \langle 75, \langle\rangle, \langle\rangle \rangle, \langle z, \langle\rangle, \langle\rangle \rangle \rangle \rangle$$

On utilise pour les arbres une terminologie inspirée des liens de parenté :
- les *descendants* d'un noeud p sont les noeuds qui apparaissent dans ses sous-arbres,
- un *ancêtre* d'un noeud p est soit son père, soit un ancêtre de son père,
- le chemin qui relie un noeud à la racine est constitué de tous ses ancêtres (c'est-à-dire de son père et des noeuds du chemin qui relie son père à la racine),
- un *frère* d'un noeud p est un fils du père de p, et qui n'est pas p.

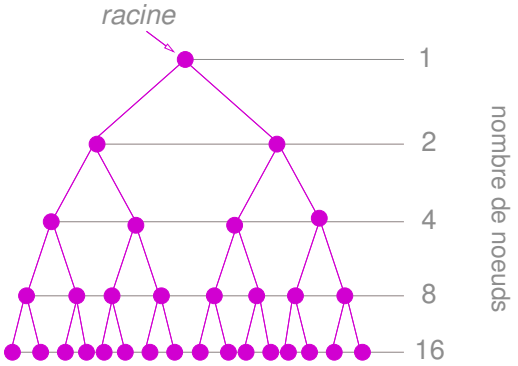


Les noeuds d'un arbre se répartissent par *niveaux* : le premier niveau (par convention ce sera le niveau 0) contient la racine seulement, le deuxième

niveau contient les deux fils de la racine,..., les noeuds du niveau k sont les fils des noeuds du niveau $k-1$,.... La *hauteur* d'un arbre est le nombre de niveaux de ses noeuds. C'est donc aussi le nombre de noeuds qui jalonnent la branche la plus longue. Attention, la définition de la hauteur varie en fonction des auteurs. Pour certains la hauteur d'un arbre contenant un seul noeud est 0.

Arbres binaires : hauteur, nombre de noeuds et nombre de feuilles.

Un arbre binaire est *complet* si toutes ses branches ont la même longueur et tous ses noeuds qui ne sont pas des feuilles ont deux fils. Soit A un arbre binaire complet. Le nombre de noeuds de A au niveau 0 est 1, le nombre de noeuds au niveau 1 est 2,..., et le nombre de noeuds au niveau p est donc 2^p. En particulier le nombre de feuilles est donc 2^{h-1} si h est le nombre de niveaux.



Le nombre total de noeuds d'un arbre binaire complet de h niveaux est

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

On en déduit que la hauteur $ht(A)$ (le nombre de niveaux) d'un arbre binaire A contenant n noeuds est au moins égale à

$$\lfloor \log_2 n \rfloor + 1$$

Preuve. Si A est arbre de hauteur h et comportant n noeuds, pour tout entier m, on a : $h \leq m-1 \Rightarrow n < 2^{m-1}$. Par contraposée, on a : $n \geq 2^{m-1} \Rightarrow h \geq m$. Le plus grand entier m vérifiant $n \geq 2^{m-1}$ est $\lfloor \log_2 n \rfloor + 1$. On a donc $h \geq \lfloor \log_2 n \rfloor + 1$.

Exercice : Montrez que pour tout entier $n \geq 1$, $\lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$.

La hauteur minimale $\lfloor \log_2 n \rfloor + 1$ est atteinte lorsque l'arbre est complet. Pour être efficaces, les algorithmes qui utilisent des arbres binaires font en sorte que ceux ci soient équilibrés (voir les tas ou les AVL-arbres par exemple).

Les arbres en théorie des graphes. En théorie des graphes un arbre est un graphe *connexe* et *sans cycles*, c'est-à-dire qu'entre deux sommets quelconques du graphe il existe un chemin et un seul. On parle dans ce cas d'arbre non planté, puisqu'il n'y a pas de sommet particulier qui joue le rôle de racine. Les sommets sont voisins les uns des autres, il n'y a pas de hiérarchie qui permet de dire qu'un sommet est le père (ou le fils) d'un autre sommet. On démontre qu'un graphe de n sommets qui a cette propriété contient exactement $n-1$ arêtes, et que l'ajout d'une nouvelle arête crée un cycle, tandis que le retrait d'une arête le rend non connexe.

1.3 Arbres binaires

Les *arbres binaires* (AB) forment une structure de données qui peut être définie récursivement de la manière suivante : un arbre binaire est

- soit vide,
- soit composé d'une racine portant une étiquette (clé) et d'une paire d'arbres binaires, appelés fils gauche et droit.

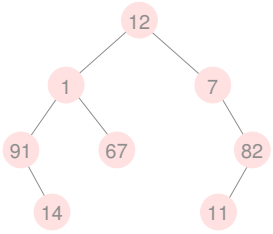
Pour pouvoir manipuler des arbres de recherche, on doit donc disposer des fonctionnalités suivantes :

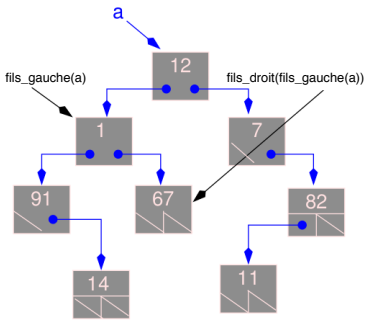
- `booléen : est_vide(AB a)` : une fonction qui teste si un arbre est vide ou non,
- `AB : fils_gauche(AB a)` et `AB : fils_droit(AB a)`, des fonctions qui retournent les fils gauche et droit d'un arbre non vide,
- `CLE : val(AB a)`, une fonction qui retourne l'étiquette de la racine d'un arbre non vide,
- `AB cree_arbre_vide()`, une fonction qui crée un arbre vide,
- `AB cree_arbre(CLE v, AB fg, fd)`, une fonction qui crée un arbre dont les fils gauche et droit sont `fg` et `fd` et dont la racine est étiquetée par `v`
- `change_val(CLE v, AB a)`, une fonction qui remplace l'étiquette de la racine de `a` par `v`, si `a` est non vide, et ne fait rien sinon
- `change_fg(AB fg, AB a)` et `change_fd(AB fd, AB a)`, des fonctions qui remplacent le fils gauche de `a` (resp. le fils droit de `a`) par `fg` (resp.

par `fd`), si `a` est non vide, et ne font rien sinon.

En C on utilise en général des pointeurs pour représenter les liens entre un noeud et ses fils dans un arbre binaire.

En Python, on peut représenter un arbre vide par une liste vide `[]` et un arbre non vide par une liste comprenant 3 éléments `[clé,fils_gauche,fils_droit]`.



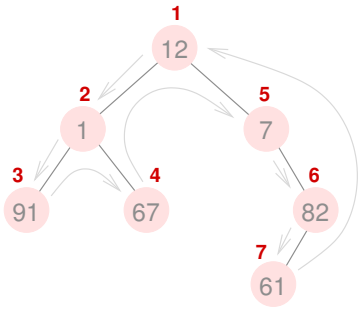


Parcours d'un arbre binaire. Le parcours le plus simple à programmer est le parcours dit *en profondeur d'abord*. Son principe est simple : pour parcourir un arbre non vide a, on parcourt récursivement son sous-arbre gauche, puis son sous-arbre droit, la racine de l'arbre pouvant être traitée au début, entre les deux parcours ou à la fin. Dans le premier cas, on dit que les nœuds sont traités dans un ordre *préfixe*, dans le second cas, dans un ordre *infixe* et dans le troisième cas, selon un ordre *postfixe*.

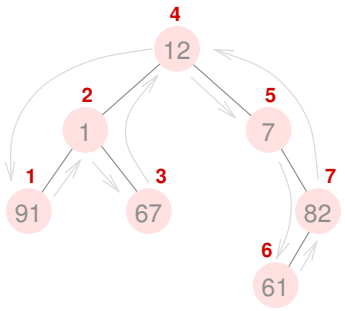
```
Parcours(AB a)
   si NON est_vide(a)
      Traitement_prefixe(val(a))
      Parcours(fils_gauche(a))
      Traitement_infixe(val(a))
      Parcours(fils_droit(a))
      Traitement_postfixe(val(a))
```

L'affichage des étiquettes des nœuds d'un arbre binaire peut se faire dans un ordre préfixe, infixe ou postfixe. Pour l'arbre ci-dessus, un affichage préfixe donnerait : 12 1 91 67 7 82 61.

```
Affichage_préfixe(AB a)
   si NON est_vide(a)
      Afficher val(a)
      Affichage_préfixe(fils_gauche(a))
      Affichage_préfixe(fils_droit(a))
```

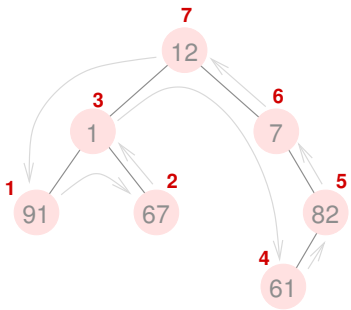



```
Affichage_infixe(AB a)
   si NON est_vide(a)
      Affichage_infixe(fils_gauche(a))
      Afficher val(a)
      Affichage_infixe(fils_droit(a))
```



91 1 67 12 7 61 82

```
Affichage_postfixe(AB a)
   si NON est_vide(a)
      Affichage_postfixe(fils_gauche(a))
      Affichage_postfixe(fils_droit(a))
      Afficher val(a)
```



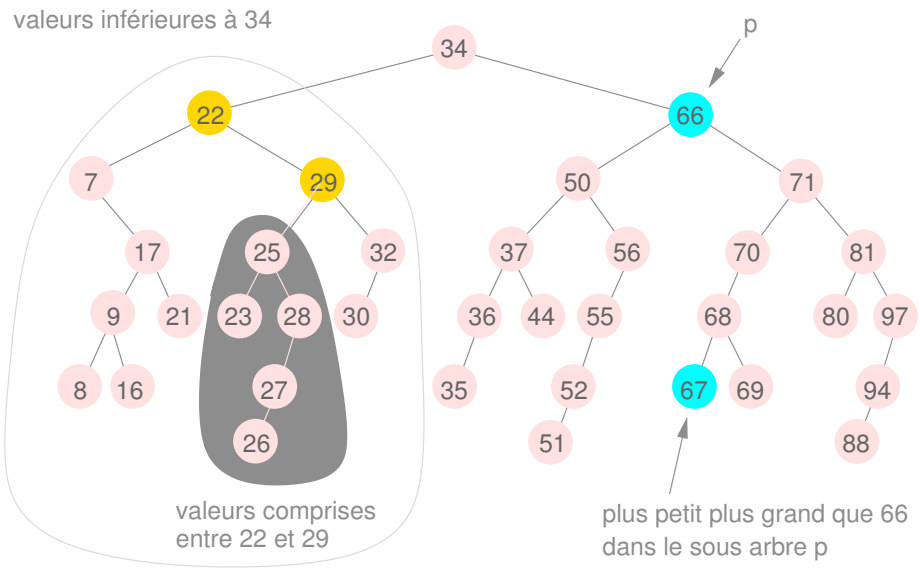
91 67 1 61 82 7 12

1.4 Arbres binaires de recherche.

Un arbre binaire de recherche (ou ABR) est une structure de donnée qui permet de représenter un ensemble de valeurs si l'on dispose d'une relation d'ordre sur ces valeurs. Les opérations caractéristiques sur les arbres binaires de recherche sont l'**insertion**, la **suppression**, et la **recherche** d'une valeur. Ces opérations sont peu coûteuses si l'arbre n'est pas trop

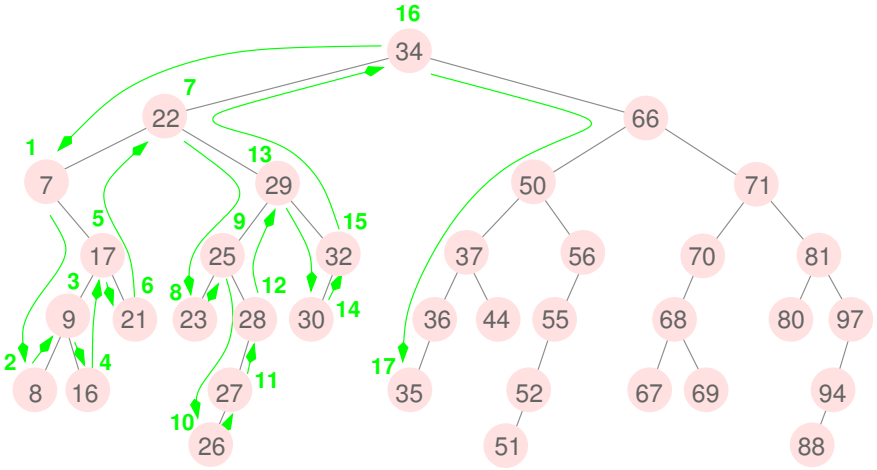
déséquilibré. En pratique, les valeurs sont des clés permettant d'accéder à des enregistrements.

Soit E un ensemble muni d'une relation d'ordre, et soit A un arbre binaire portant des valeurs de E. L'arbre A est un *arbre binaire de recherche* si pour tout noeud p de A, la valeur de p est strictement plus grande que les valeurs figurant dans son sous-arbre gauche et strictement plus petite que les valeurs figurant dans son sous-arbre droit. Cette définition suppose donc qu'une valeur n'apparaît au plus qu'une seule fois dans un arbre de recherche.



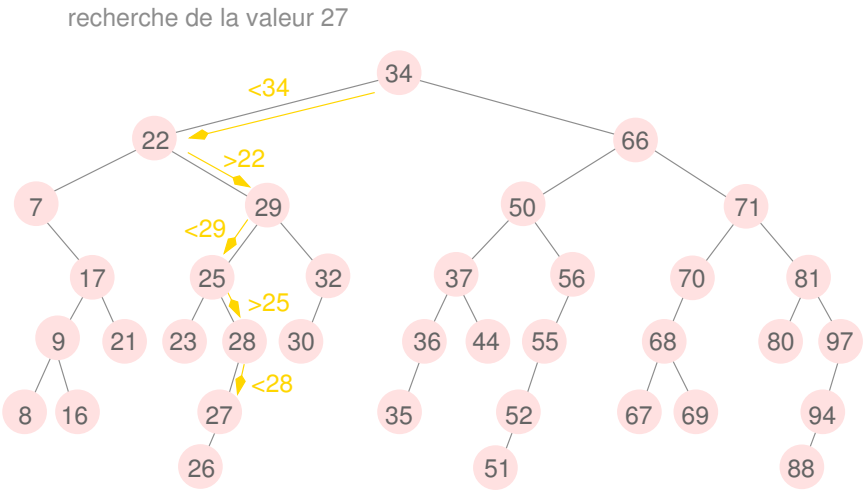
Pour accéder à la clé la plus petite (resp. la plus grande) dans un ABR il suffit de descendre sur le fils gauche (resp. sur le fils droit) autant que possible. Le dernier noeud visité, qui n'a pas de fils gauche (resp. droit), porte la valeur la plus petite (resp. la plus grande) de l'arbre.

Le parcours infixé de l'arbre produit la suite ordonnée des clés



7 8 9 16 17 21 22 23 25 26 27 28 29 30 32 34 35 36 37 ...

Recherche d'une valeur. La recherche d'une valeur dans un ABR consiste à parcourir une branche en partant de la racine, en descendant chaque fois sur le fils gauche ou sur le fils droit suivant que la clé portée par le noeud est plus grande ou plus petite que la valeur cherchée. La recherche s'arrête dès que la valeur est rencontrée ou que l'on a atteint l'extrémité d'une branche (le fils sur lequel il aurait fallu descendre n'existe pas).



Fonction recherche(a, v) : version itérative

```
entrée : a est un ABR, v est une clé.
sortie : Vrai si v figure dans a et Faux sinon.
début
    tant que NON est_vide(a) ET v ≠ val(a) faire
        si v < val(a) alors
            a = fils_gauche(a)
        sinon
            a = fils_droit(a)
        finsi
    fintq
    si est_vide(a) alors
        retourner Faux
    sinon
        retourner Vrai
    finsi
fin
```

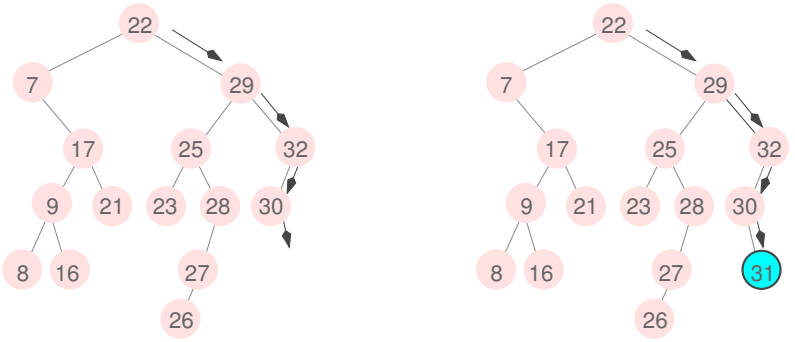
Exercice : montrez que "si v figure dans l'arbre d'origine, alors v figure dans a" est un invariant de l'algorithme. En déduire qu'il est correct.

Fonction recherche(a, v) : version récursive

```
entrée : a est un ABR, v est une clé.
sortie : Vrai si v figure dans a et Faux sinon.
début
    si est_vide(a) alors
        retourner Faux
    sinon
        si v == val(a) alors
            retourner Vrai
        sinon
            si v < val(a) alors
                retourner recherche(v,fils_gauche(a))
            sinon
                retourner recherche(v,fils_droit(a))
            finsi
        finsi
    finsi
fin
```

Exercice : prouvez la correction de l'algorithme précédent.

Insertion d'une nouvelle valeur. Le principe est le même que pour la recherche. Un nouveau noeud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arêtée.

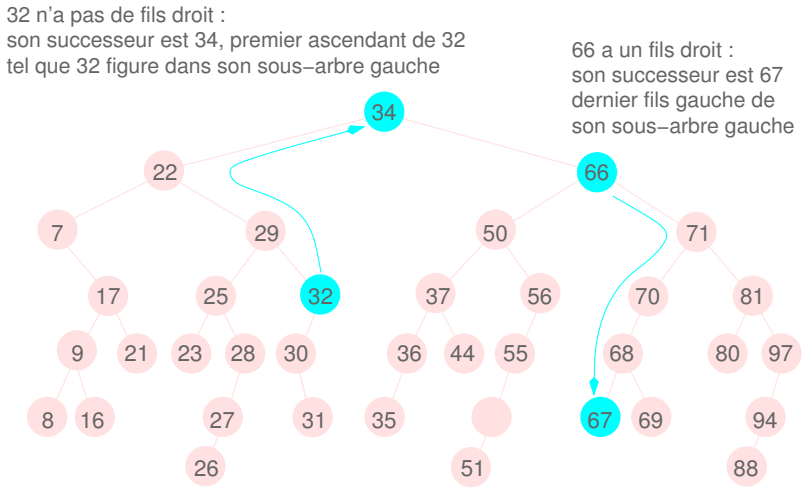


Algorithme Insertion: Insertion d'une nouvelle clé dans un ABR

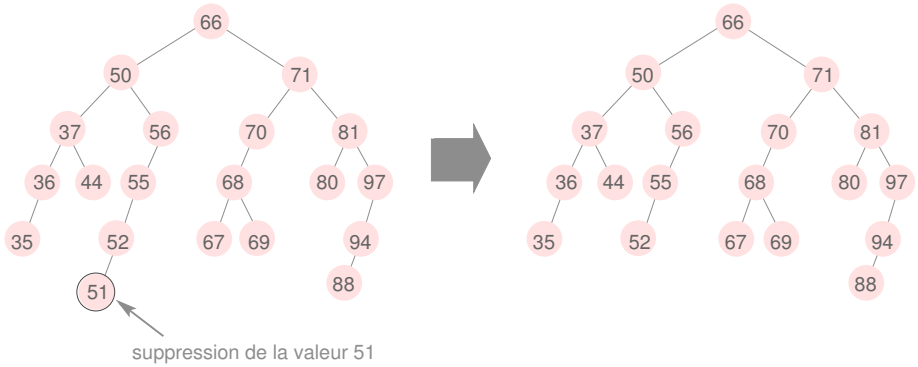
```
entrée : a est un ABR, v est une clé.
résultat : v est insérée dans a
début
    si est_vide(a) alors
        a=cree_arbre(v,cree_arbre_vide(),cree_arbre_vide())
    sinon
        si v < val(a) alors
            Insertion(v,fils_gauche(a))
        sinon
            si v > val(a) alors
                Insertion(v,fils_droit(a))
            finsi
        finsi
    finsi
fin
```

Recherche du successeur d'un noeud. Étant donné un noeud p d'un arbre A, le successeur de p si il existe, est le noeud de A qui porte comme valeur la plus petite des valeurs qui figurent dans A et qui sont plus grandes que la valeur de p. Si p possède un fils droit, son successeur est le noeud le plus à gauche dans son sous-arbre droit (on y accède en descendant sur le fils

gauche autant que possible). Si p n'a pas de fils droit alors sont successeur est le premier de ses ascendants tel que p apparaît dans son sous-arbre gauche. Si cet ascendant n'existe pas c'est que p portait la valeur la plus grande dans l'arbre. Remarquez qu'il est nécessaire d'avoir pour chaque noeud un lien vers son père pour mener à bien cette opération.

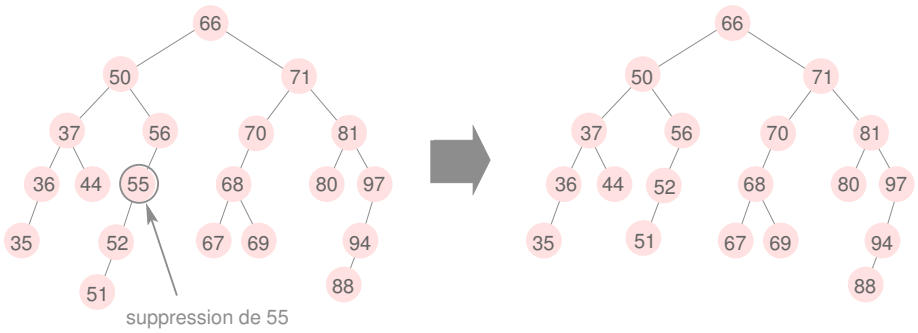


Suppression d'un noeud. L'opération dépend du nombre de fils du noeud à supprimer.

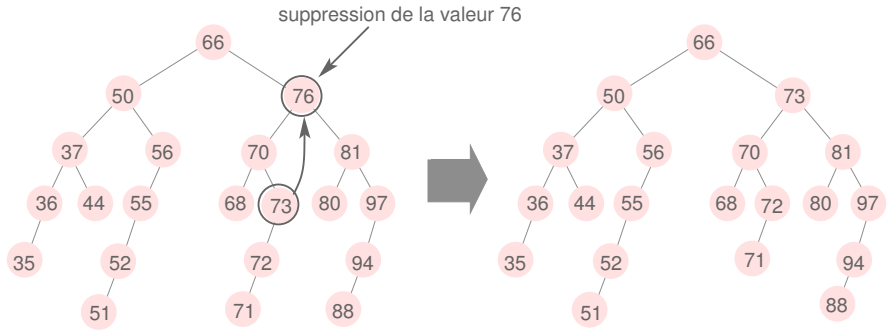


Cas 1 : le noeud à supprimer n'a pas de fils, c'est une feuille. Il suffit de *décrocher* le noeud de l'arbre, c'est-à-dire de l'enlever en modifiant le lien

du père, si il existe, vers ce fils. Si le père n'existe pas l'arbre devient l'arbre vide.



Cas 2 : le noeud à supprimer a un fils et un seul. Le noeud est *décroché* de l'arbre comme dans le cas 1. Il est remplacé par son fils unique dans le noeud père, si ce père existe. Sinon l'arbre est réduit au fils unique du noeud supprimé.



Cas 3 : le noeud à supprimer p a deux fils. Soit q le noeud de son sous-arbre gauche qui a la valeur la plus grande (on peut prendre indifféremment le noeud de son sous-arbre droit de valeur la plus petite). Il suffit de recopier la valeur de q dans le noeud p et de décrocher le noeud q. Puisque le noeud q a la valeur la plus grande dans le fils gauche, il n'a donc pas de fils droit, et peut être décroché comme on l'a fait dans les cas 1 et 2.

1.5 Exercices

1.5.1 Implémentation en Python par des listes

On implémente les arbres binaires non vides par des listes de trois éléments : `[valeur,[fils_gauche],[fils_droit]]`. Écrivez les fonctions suivantes en Python :

1. `est_vide(a)` : retourne `True` si `a` est vide et `False` sinon.
2. `fils_gauche(a)` : retourne le fils gauche de `a` si `a` est non vide et rien sinon.
3. `fils_droit(a)` : retourne le fils droit de `a` si `a` est non vide et rien sinon.
4. `insertion(v, a)` : insère la valeur `v` dans `a` si elle n'y figure pas et ne fait rien sinon ; l'arbre passé en paramètre doit être modifié.
5. `insertion_liste(lv, a)` : insère les valeurs de la liste `lv` dans `a` qui n'y figurent pas déjà.
6. `afficher(a,mode)` : affiche les valeurs de `a` selon le mode précisé par le second argument, où `mode` est une chaîne de caractères qui peut prendre les valeurs `prefixe`, `infixe`, `postfixe`.
7. `recherche_Iter(v, a)` et `recherche_Rec(v, a)` : retourne `True` si `v` figure `a` et `False` sinon - versions itérative et récursive.
8. `max_val(a)` : retourne la valeur maximale d'un ABR non vide.
9. `hauteur(a)` : retourne la hauteur d'un ABR.
10. `nb_noeuds(a)` : retourne le nombre de noeuds d'un ABR.
11. `successeur(v,a)` : retourne la plus petite valeur plus grande que v figurant dans a s'il y en a une, et v sinon.
12. `extrait_max(a)` : retourne la valeur maximale d'un ABR non vide et supprime le noeud correspondant.
13. `supprime(v, a)` : supprime le noeud de `a` portant la valeur `v`, s'il en existe un.

1.5.2 Arbres fortement équilibrés

Un arbre binaire est *fortement équilibré* s'il est vide ou si, pour chacun de ses noeuds n, les fils gauche et droit de n ont le même nombre de noeuds, à une unité près :

$$|nb\_noeuds(fils\_gauche(n)) - nb\_noeuds(fils\_droit(n))| \leq 1.$$

1. Exprimez la hauteur d'un arbre fortement équilibré `a` en fonction du nombre de noeuds de `a`.
2. Quelle est la complexité de la recherche d'un élément dans un arbre fortement équilibré `a` en fonction du nombre de noeuds de `a` ?
3. Si un ABR n'est pas fortement équilibré, quelle peut-être la complexité de ces opérations dans le pire des cas ?
4. Écrivez une fonction récursive `int : est_fortement_equilibré(a)` qui renvoie -1 si a n'est pas fortement équilibré, et le nombre de noeuds de a sinon.
5. Étant donnée une liste $L = [x_1, \ldots, x_n]$ de n nombres entiers distincts, quel est ou quels sont les éléments de L qui peuvent étiqueter la racine d'un ABR fortement équilibré composé des éléments de L ? En déduire un algorithme récursif construisant un ABR fortement équilibré à partir d'une liste **triée** L de nombres entiers distincts.
6. Écrivez des fonctions
   (a) insérant une nouvelle valeur et
   (b) supprimant une valeur,

   dans un arbre `a` fortement équilibré, et préservant cette propriété.
7. Quelle est la complexité de ces opérations en fonction de la hauteur de `a`, du nombre de noeuds de `a` ?
8. La propriété d'être fortement équilibré est inutilement forte - et cela pénalise les algorithmes d'insertion et de suppression. Quelle autre propriété d'équilibrage pourrait-on envisager ?

1.5.3 Successeur d'un noeud

Dans un arbre binaire de recherche `a`, on considère les nœuds ou ensembles de nœuds suivants (on identifie un nœud au sous-arbre correspondant) :

- `x` est un nœud de `a`, γ est le chemin qui relie la racine de `x` à la racine de `a`,
- `y` est le nœud de γ, le plus proche de `x` tel que $x \in fg(y)$,
- $z = fg(y)$,
- $A = \{t \in a : t \notin y \text{ et } val(t) < val(y)\}$,
- $B = \{t \in a : t \notin y \text{ et } val(t) > val(y)\}$,
- $C = fd(x)$,
- $D = fg(z)$,
- $E = fd(y)$,

– F est composé des nœuds restants.

1. Représentez ces éléments sur un dessin.
2. Classez x, y, z et les éléments de A, B, C, D, E et F par ordre croissant d'étiquette.
3. En déduire le successeur de x ; envisagez tous les cas particuliers.

1.5.4 Arbres binaires de recherche et rangs

On enrichit la structure des arbres binaires de recherche en ajoutant un champ numérique *nbg* à chaque noeud, qui contiendra le nombre de noeuds de son sous-arbre gauche. Un ABR non vide est donc représenté par une liste à 4 éléments `[val, nbg, fg, fd]`.

On supposera que les valeurs d'un arbre de recherche sont toutes distinctes.

1. Dessinez l'arbre de recherche obtenu en insérant dans un arbre vide les valeurs suivantes : 12, 5, 24, 3, 37, 49, 25, 17, 8 et 29, dans cet ordre et en faisant apparaître le champ *nbg* pour chaque noeud.
2. L'algorithme permettant d'insérer une nouvelle valeur dans un arbre de recherche doit être modifié de façon à mettre à jour le champ *nbg*. Écrivez ce nouvel algorithme d'insertion. On supposera que l'on dispose
   – d'une fonction `nbg(a)` qui retourne le nombre de noeuds du sous-arbre gauche de l'arbre non vide `a`,
   – d'une fonction `cree_arbre(val,nbg,fg,fd)` qui retourne l'arbre `[v,nbg,fg,fd]`,
   – et d'une fonction `change_nbg(n,a)` qui remplace la valeur du champ `nbg` de `a` par `n`.
3. Le rang $r(x)$ d'un élément `x` figurant dans l'arbre de recherche `a` est égal au nombre d'éléments de `a` strictement plus petit que `x` : le rang du plus petit élément de `a` est donc égal à 0.

   Soit $r \geq 0$. On souhaite trouver l'élément `x` de `a` de rang r. On suppose que `a` contient au moins $r + 1$ éléments. Montrez que
   – si $r = nbg(a)$, alors $val(a) = x$,
   – si $r < nbg(a)$ alors `x` est aussi l'élément de rang r du sous-arbre gauche de `a`,
   – si $r > nbg(a)$ alors `x` est l'élément de rang $r-1-nbg(a)$ du sous-arbre droit de `a`.

En déduire un algorithme qui prend en entrée un arbre de recherche `a` et un rang r et retourne l'élément de `a` de rang r. On suppose que `a` contient au moins $r + 1$ éléments.

4. Écrivez un algorithme qui prend en entrée un arbre de recherche `a` et un élément `x` et retourne le rang de `x` dans `a` (on supposera que `x` figure bien dans `a`).