Chapitre 8

Structures de données avancées

Une **structure de données** est une organisation logique des données permettant de simplifier ou d'accélérer leur traitement.

8.1. Pile

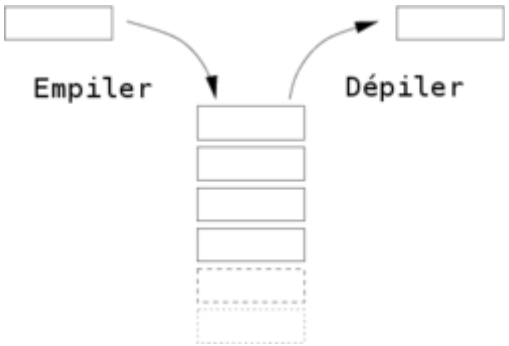
En informatique, une **pile** (en anglais *stack*) est une structure de données fondée sur le principe « dernier arrivé, premier sorti » (ou *LIFO* pour *Last In, First Out*), ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

Le fonctionnement est donc celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

Primitives

Voici les primitives communément utilisées pour manipuler des piles :

- « empiler » : ajoute un élément sur la pile. Terme anglais correspondant : « Push ».
- « dépiler » : enlève un élément de la pile et le renvoie. En anglais : « Pop »
- « vide » : renvoie vrai si la pile est vide, faux sinon
- « remplissage » : renvoie le nombre d'éléments dans la pile.



Applications

- Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente ».
- L'évaluation des expressions mathématiques en notation post-fixée (ou polonaise inverse) utilise une pile.
- La fonction « Annuler la frappe » (en anglais « Undo ») d'un traitement de texte

mémorise les modifications apportées au texte dans une pile.
- Un algorithme de recherche en profondeur utilise une pile pour mémoriser les nœuds visités.
- Les algorithmes récursifs admis par certains langages (LISP, Algol, Pascal, C, etc.) utilisent implicitement une pile d'appel. Dans un langage non récursif (FORTRAN par exemple), on peut donc toujours simuler la récursion en créant les primitives de gestion d'une pile.

Exercice 8.1

Implémentez en Python une classe « pile » avec ces quatre méthodes, ainsi qu'une méthode « afficher » qui liste tous les éléments de la pile, du dernier entré au premier entré.

8.2. File

Une **file** (*queue* en anglais) est une structure de données basée sur le principe « Premier entré, premier sorti », en anglais *FIFO* (*First In, First Out*), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés. Le fonctionnement ressemble à une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file.

Primitives

Voici les primitives communément utilisées pour manipuler des files :
- « ajouter » : ajoute un élément dans la file. Terme anglais correspondant : « enqueue »
- « enlever » : renvoie le prochain élément de la file, et le retire de la file. Terme anglais correspondant : « dequeue »
- « vide » : renvoie « vrai » si la file est vide, « faux » sinon
- « remplissage » : renvoie le nombre d'éléments dans la file.

Applications

- En général, on utilise des files pour mémoriser temporairement des transactions qui doivent attendre pour être traitées.
- Les serveurs d'impression, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent, et les insèrent dans une file d'attente (ou une queue).
- Certains moteurs multitâches, dans un système d'exploitation, qui doivent accorder du temps-machine à chaque tâche, sans en privilégier aucune.
- Un algorithme de parcours en largeur utilise une file pour mémoriser les nœuds visités.

Exercice 8.2

Implémentez en Python une classe « file » avec ces quatre méthodes, ainsi qu'une méthode « afficher » permettant de voir tous les éléments de la file, du premier entré au dernier entré.

Files à priorités

Dans une **file à priorités**, on peut effectuer trois opérations :
- insérer un élément
- supprimer le plus grand élément
- tester si la file à priorités est vide ou pas

Les principales implémentations de ces files à priorités sont le **tas** (voir § 8.9), le **tas binomial** et le **tas de Fibonacci**.

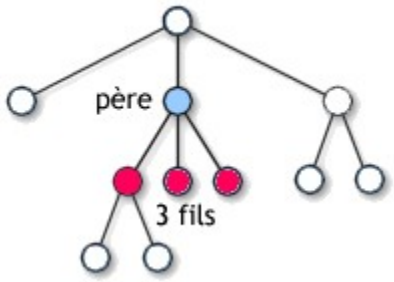
Donald **Knuth**, né en 1938), professeur émérite à Stanford, auteur de l'ouvrage de référence sur l'algorithmique, en plusieurs volumes, intitulé « *The Art of Computer Programming* », considère les arbres comme « la structure la plus fondamentale de l'informatique ».

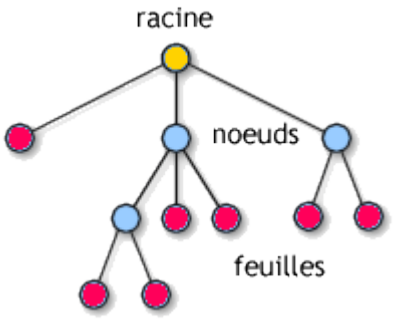
8.3. Arbres

Un arbre est un graphe sans cycle, où des nœuds sont reliés par des arêtes. On distingue trois sortes de nœuds :

- les nœuds internes, qui ont des fils ;
- les feuilles, qui n'ont pas de fils ;
- la racine de l'arbre, qui est l'unique nœud ne possédant pas de père.

Traditionnellement, on dessine toujours la racine en haut et les feuilles en bas.





La profondeur d'un nœud est la distance, *i.e.* le nombre d'arêtes, de la racine au nœud. La hauteur d'un arbre est la plus grande profondeur d'une feuille de l'arbre. La taille d'un arbre est son nombre de nœuds (en comptant les feuilles ou non).

Les arbres peuvent être étiquetés. Dans ce cas, chaque nœud possède une étiquette, qui est en quelque sorte le « contenu » du nœud. L'étiquette peut être très simple (un nombre entier, par exemple) ou plus complexe : un objet, une instance d'une structure de données, etc.

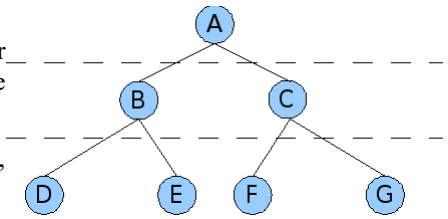
Les arbres sont en fait rarement utilisés en tant que tels, mais de nombreux types d'arbres avec une structure plus restrictive existent et permettent alors des recherches rapides et efficaces. Nous en reparlerons bientôt.

8.3.1. Parcours

Parcours en largeur

Le parcours en largeur correspond à un parcours par *niveau* de nœuds de l'arbre. Un niveau est un ensemble de nœuds ou de feuilles situés à la même profondeur.

Ainsi, si l'arbre ci-contre est utilisé, le parcours sera A, B, C, D, E, F, G.



Parcours en profondeur

Le parcours en profondeur est un parcours récursif sur un arbre. Il existe trois ordres pour cette méthode de parcours. Le parcours en profondeur préfixé est le plus courant.

Parcours en profondeur préfixé

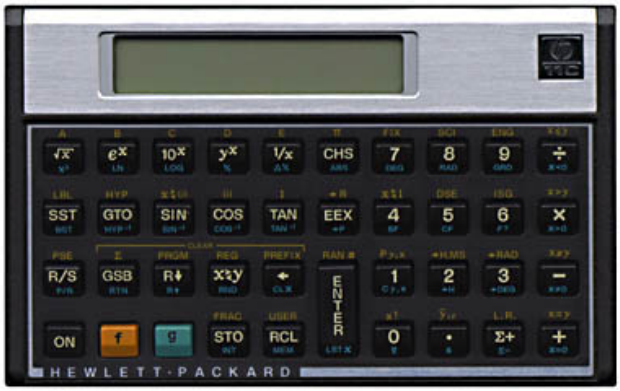
Dans ce mode de parcours, le nœud courant est traité avant le traitement des nœuds gauche et droit. Ainsi, si l'arbre précédent est utilisé, le parcours sera A, B, D, E, C, F, G.

Parcours en profondeur suffixé

Dans ce mode de parcours, le nœud courant est traité après le traitement des nœuds gauche et droit. Ainsi, si l'arbre précédent est utilisé, le parcours sera D, E, B, F, G, C, A.

Jan **Łukasiewicz**
(1878-1956)

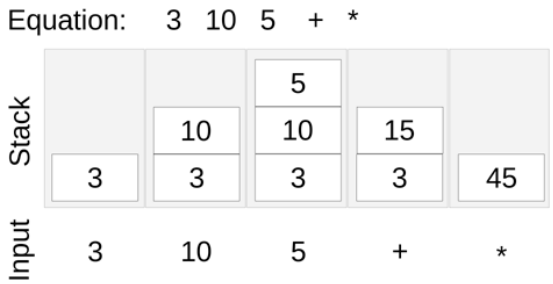
Ce mode de parcours correspond à une *notation polonaise inverse* (**NPI**, en anglais **RPN** pour *R*everse *P*olish *N*otation), également connue sous le nom de **notation post-fixée**, utilisée par certaines calculatrices HP. Dérivée de la notation polonaise présentée en 1924 par le mathématicien polonais Jan **Łukasiewicz**, elle s'en différencie par l'ordre des termes, les opérandes y étant présentés avant les opérateurs et non l'inverse.



HP-11C (1981-1989)

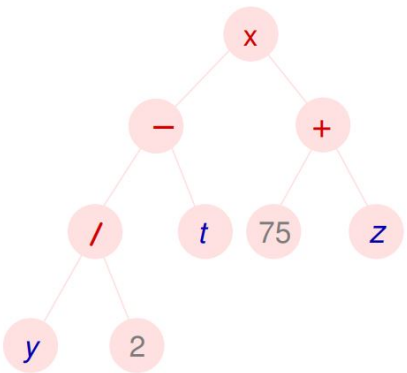
On observant bien le clavier, on s'aperçoit qu'il n'y a ni parenthèses, ni signe =, ce qui peut surprendre… Sur ces modèles, on utilise des piles.

L'expression « 3 × (5 + 10) » peut s'écrire en NPI sous la forme « 10 ENTER 5 + 3 × », ou encore sous la forme « 3 ENTER 10 ENTER 5 + × », illustrée sur le schéma ci-dessous :



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.

L'arbre binaire ci-dessous représente l'expression arithmétique ($y/2 - t$) x ($75 + z$).

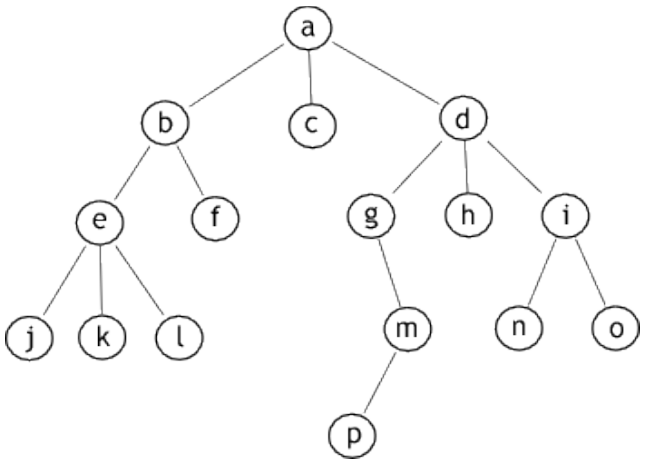


Parcours en profondeur infixé

Dans ce mode de parcours (qui n'est applicable qu'à des arbres binaires), le nœud courant est traité entre le traitement des fils gauche et droit. Ainsi, si l'arbre précédent est utilisé, le parcours sera D, B, E, A, F, C puis G.

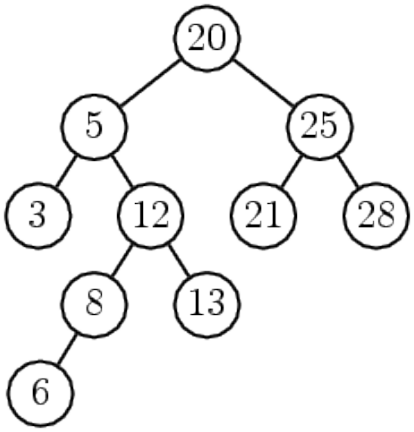
Exercice 8.3

Parcourez l'arbre ci-dessous selon les quatre méthodes vues précédemment (si possible).



8.4. Arbres binaires

Dans un **arbre binaire**, chaque nœud possède au plus deux fils, habituellement appelés « gauche » et « droit ». Du point de vue des fils, l'élément dont ils sont issus au niveau supérieur est logiquement appelé père.



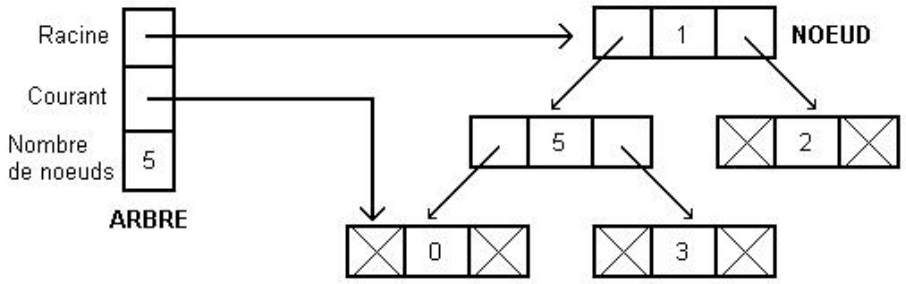
8.4.1. Types d'arbres binaires

- Un arbre binaire entier est un arbre dont tous les nœuds possèdent zéro ou deux fils.
- Un arbre binaire parfait est un arbre binaire entier dans lequel toutes les feuilles sont à la même hauteur.
- L'arbre binaire parfait est parfois nommé **arbre binaire complet**. Cependant certains définissent un arbre binaire complet comme étant un arbre binaire entier dans lequel les feuilles ont pour profondeur n ou n-1 pour un n donné.

8.4.2. Méthodes pour stocker des arbres binaires

Structure à 3 nœuds

Les arbres binaires peuvent être construits de différentes manières. Dans un langage avec structures et pointeurs (ou références), les arbres binaires peuvent être conçus en ayant une structure à trois nœuds qui contiennent quelques données et des pointeurs vers son fils droit et son fils gauche. Parfois, il contient également un pointeur vers son unique parent. Si un nœud possède moins de deux fils, l'un des deux pointeurs peut être affecté de la valeur spéciale nulle.



L'arbre conservera un pointeur vers la racine, un pointeur vers le nœud courant (qui permet de simplifier l'ajout et la consultation) et éventuellement le nombre de nœuds du graphe.

Avantages

- Conçu pour contenir un nombre variable de nœuds.
- Pas de gaspillage de mémoire.

Inconvénients

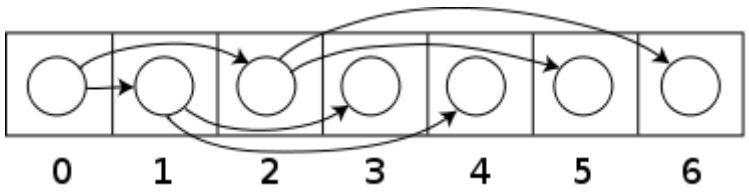
- Implémentation délicate à réaliser quand on est débutant en programmation.
- Pas d'accès direct à un nœud de l'arbre.

Tableau

Les arbres binaires peuvent aussi être rangés dans des tableaux, et si l'arbre est un arbre binaire complet, cette méthode ne gaspille pas de place, et la donnée structurée résultante est appelée un tas.

Dans cet arrangement compact, un nœud a un indice i, et ses fils se trouvent aux indices $2i+1$ et $2i+2$, tandis que son père se trouve à l'indice floor($(i-1)/2$), s'il existe.

La fonction floor(x) retourne l'entier inférieur ou égal à x.



La racine (nœud 0) a pour fils les nœuds 1 et 2. Le nœud 1 a pour fils 3 et 4, etc.

Avantages

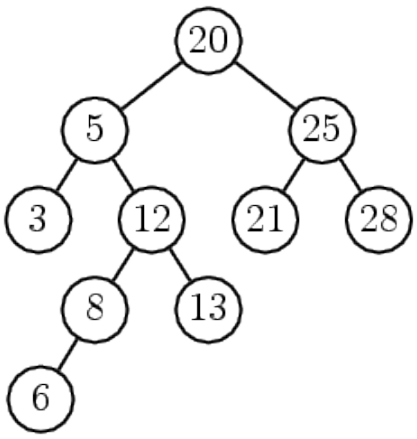
- Implémentation facile à réaliser.
- Possibilité d'accès direct à un nœud de l'arbre (un seul accès en mémoire).

Inconvénients

- Conçu pour contenir un nombre fixe de nœuds.
- Si l'arbre est profond mais contient peu de nœuds, il se produit un gaspillage important de mémoire.

Exercice 8.4

Programmez en Python les trois algorithmes de parcours en profondeur vus au § 8.3.1 puis exécutez-les sur un tableau représentant l'arbre binaire ci-dessous :



8.5. La notation *O(f (n))*

Dans les années 1960 et au début des années 1970, alors qu'on en était à découvrir des algorithmes fondamentaux, on ne mesurait pas leur efficacité. On se contentait de dire, par exemple : « Cet algorithme se déroule en 6 secondes avec un tableau de 50'000 entiers choisis au hasard en entrée, sur un ordinateur IBM 360/91. Le langage de programmation PL/I a été utilisé avec les optimisations standards. »

Une telle démarche rendait difficile la comparaison des algorithmes entre eux. La mesure publiée était dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation et du compilateur utilisé, etc.

Comme il est très difficile de comparer les performances des ordinateurs, une approche indépendante des facteurs matériels était nécessaire pour évaluer l'efficacité des algorithmes. Donald Ervin **Knuth** (1938-) fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa fameuse série *The Art of Computer Programming*.

La notation grand *O* (avec la lettre majuscule *O*, et non pas le chiffre zéro) est un symbole utilisé en mathématiques, notamment en théorie de la complexité, pour décrire le comportement asymptotique des fonctions.

> Une fonction $g(n)$ est dite $O(f(n))$ s'il existe un entier N et une constante réelle positive c tels que pour $n > N$ on ait $g(n) \leq c \cdot f(n)$.

En d'autres termes, $O(f(n))$ est l'ensemble de toutes les fonctions $g(n)$ bornées supérieurement par un multiple réel positif de $f(n)$, pour autant que n soit suffisamment grand.

Attention, $g(n)$ est $O(f(n))$ ne veut pas dire que $g(n)$ s'approche asymptotiquement de $f(n)$, mais seulement qu'au-delà de certaines valeurs de son argument, $g(n)$ est dominé par un certain multiple de $f(n)$.

8.5.1. Exemples

$g(n) = n^3 + 2n^2 + 4n + 2 = O(n^3)$ $\qquad$ $g(n) = n^3 + 10^{100}n^2 = O(n^3)$

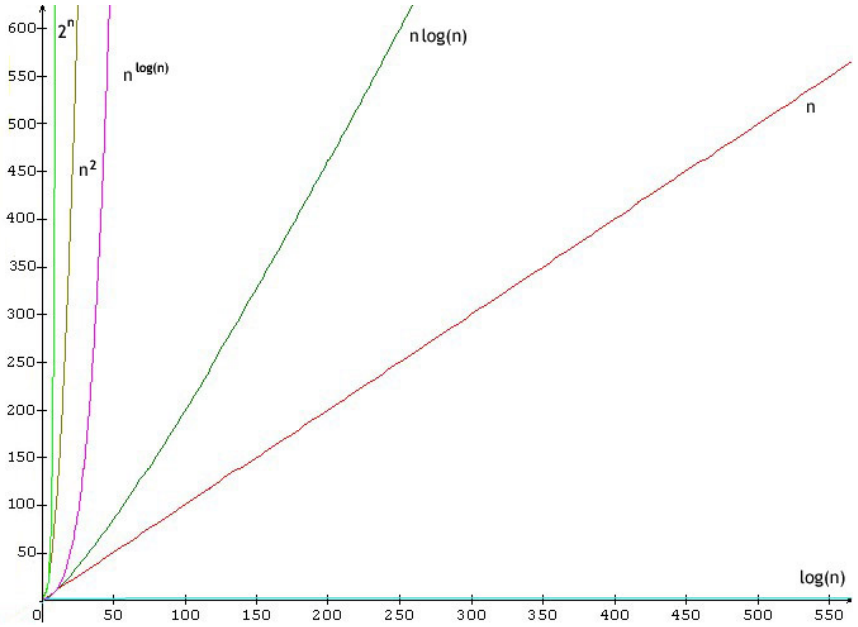
$g(n) = n \cdot log(n) + 12n + 888 = O(n \cdot log(n))$

$g(n) = 1000n^{10} - n^7 + 12n^4 + 2^n = O(2^n)$

8.5.2. Quelques classes de complexité

Notation	Type de complexité
$O(1)$	complexité constante (indépendante de la taille de la donnée)
$O(log(n))$	complexité logarithmique
$O(n)$	complexité linéaire
$O(n \cdot log(n))$	complexité quasi-linéaire
$O(n^2)$	complexité quadratique
$O(n^3)$	complexité cubique
$O(n^p)$	complexité polynomiale
$O(2^n)$	complexité exponentielle
$O(n!)$	complexité factorielle

Les complexités sont données de la meilleure, $O(1)$, à la pire, $O(n!)$.



Quelques courbes de complexité

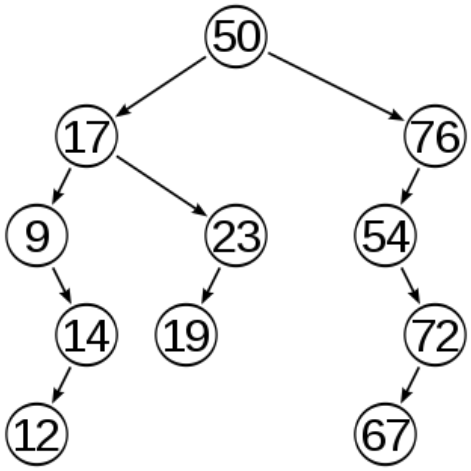
Exercice 8.5

Un programme résout un problème de taille n en $n!$ secondes. Combien de temps est nécessaire pour résoudre un problème de taille $n = 12$?

8.6. Arbres binaires de recherche

Un arbre binaire de recherche (ABR) est un arbre binaire dans lequel chaque nœud possède une étiquette, telle que chaque nœud du sous-arbre gauche ait une étiquette inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une étiquette supérieure ou égale à celle-ci. Selon la mise en œuvre de l'ABR, on pourra interdire ou non des étiquettes de valeur égale. Dans ce cours, nous supposerons que chaque étiquette est unique.

On prendra ici des nombres comme étiquettes, mais on peut imaginer des mots, des personnes, des animaux, etc.



Si vous explorez cet arbre selon un parcours en profondeur infixé, vous obtiendrez les nombres par ordre croissant. Essayez !

8.6.1. Recherche

La recherche dans un arbre binaire d'un nœud ayant une étiquette particulière est un procédé récursif. On commence par examiner la racine. Si l'étiquette de la racine est l'étiquette recherchée, l'algorithme se termine et renvoie la racine. Si l'étiquette cherchée est inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même, si l'étiquette recherchée est strictement supérieure à l'étiquette de la racine, la recherche continue sur le sous-arbre droit. Si on atteint une feuille dont l'étiquette n'est pas celle recherchée, on sait alors que cette étiquette n'est pas dans l'arbre.

Cette opération requiert un temps en $O(\log(n))$ dans le cas moyen, mais $O(n)$ dans le pire des cas où l'arbre est complètement déséquilibré où chaque père a un seul fils.

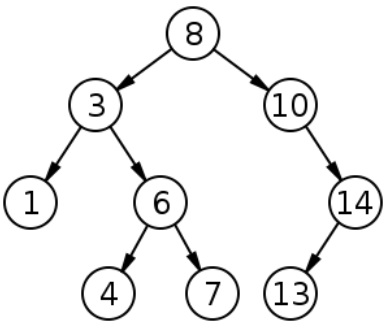
8.6.2. Insertion

L'insertion d'un nœud commence par une recherche : on cherche l'étiquette du nœud à insérer. Si on la trouve, on ne fait rien. Sinon, lorsqu'on ne peut plus descendre dans l'arbre, cela signifie qu'on a trouvé le père. On insère le nœud en comparant son étiquette à celle de son père : si elle est inférieure, le nouveau nœud sera son fils gauche ; sinon il sera son fils droit. Ainsi, chaque nœud ajouté sera une feuille.

La complexité de l'insertion est $O(\log(n))$ dans le cas moyen et $O(n)$ dans le pire des cas.

Exercice 8.6

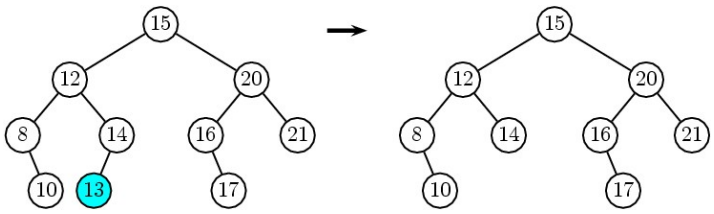
Insérez dans l'arbre binaire de recherche ci-dessous les valeurs 11 et 5.



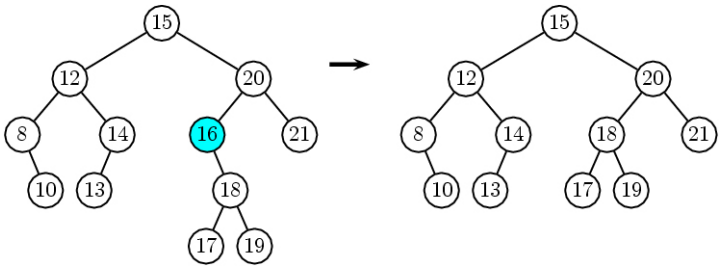
8.6.3. Suppression

Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé :

- **Suppression d'une feuille**
  Il suffit de l'enlever de l'arbre étant donné qu'elle n'a pas de fils.

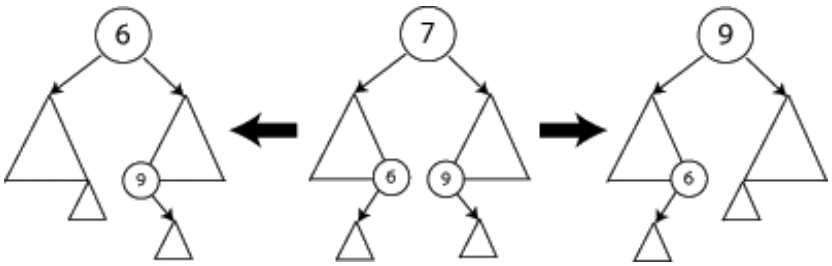


- **Suppression d'un nœud avec un seul fils**
  On l'enlève de l'arbre et on le remplace par son fils.

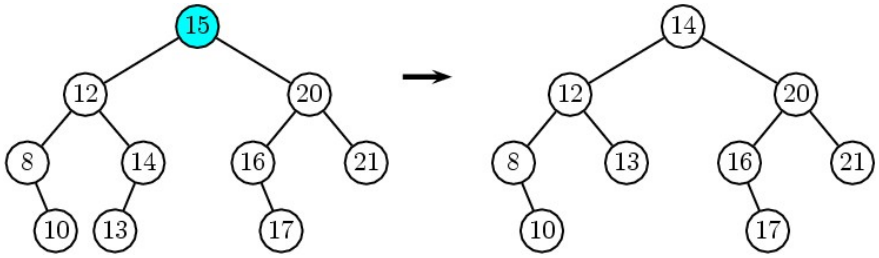


Pour aller le plus à gauche dans le sous-arbre droit, on part de la racine, on se déplace une fois à droite, puis toujours à gauche, tant qu'on le peut.

- **Suppression d'un nœud avec deux fils**
  Supposons que le nœud à supprimer soit appelé *N* (le nœud de valeur 7 dans le schéma ci-dessous). On le remplace alors par son successeur le plus proche, donc le nœud le plus à gauche du sous-arbre droit (ci-après, le nœud de valeur 9) ou son plus proche prédécesseur, donc le nœud le plus à droite du sous-arbre gauche (ci-dessous, le nœud de valeur 6).



Cela permet de garder une structure d'arbre binaire de recherche. Puis on applique à nouveau la procédure de suppression à *N*, qui est maintenant une feuille ou un nœud avec un seul fils.



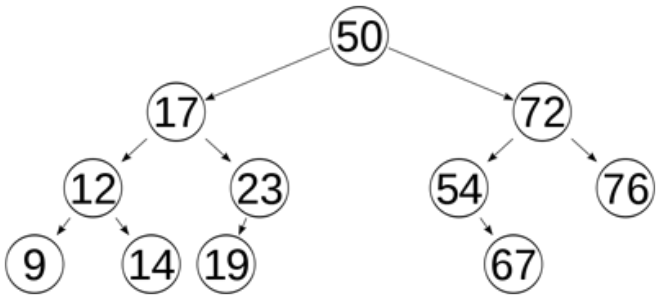
Pour une implémentation efficace, il est déconseillé d'utiliser uniquement le successeur ou le prédécesseur, car cela contribue à déséquilibrer l'arbre.

Dans tous les cas, une suppression requiert de parcourir l'arbre de la racine jusqu'à une feuille : le temps d'exécution est donc proportionnel à la profondeur de l'arbre qui vaut n dans le pire des cas, d'où une complexité maximale en $O(n)$.

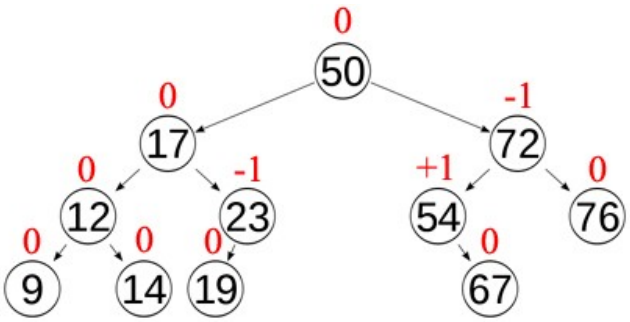
8.7. Arbres AVL

La dénomination « arbre AVL » provient des noms de ses deux inventeurs russes, Georgy Maximovich **Adelson-Velsky** et Evgenii Mikhailovich **Landis**, qui l'ont publié en 1962 sous le titre *An algorithm for the organization of information*.

Les arbres AVL ont été historiquement les premiers arbres binaires de recherche automatiquement équilibrés. Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un. La recherche, l'insertion et la suppression sont toutes en $O(\log(n))$ dans le pire des cas.

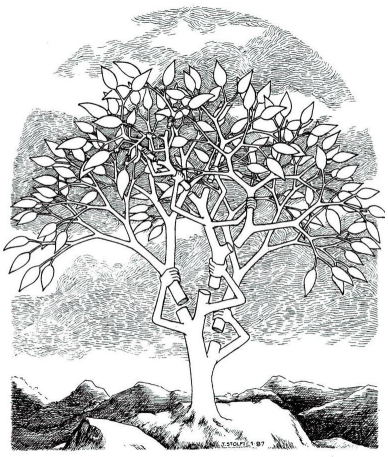


Le facteur d'équilibrage d'un nœud est la différence entre la hauteur de son sous-arbre droit et celle de son sous-arbre gauche. Un nœud dont le facteur d'équilibrage est +1, 0, ou -1 est considéré comme équilibré.



Un nœud avec tout autre facteur est considéré comme déséquilibré et requiert un rééquilibrage. Chaque fois qu'un nœud est inséré ou supprimé d'un arbre AVL, le facteur d'équilibrage de chaque nœud le long du chemin depuis la racine jusqu'au nœud inséré (ou supprimé) doit être recalculé.

Si l'arbre est resté équilibré, il n'y a rien à faire. Si ce n'est pas le cas, on effectuera des rotations d'équilibrage de manière à obtenir à nouveau un arbre AVL.



8.7.1. Rotations d'équilibrage

Une rotation est une modification locale d'un arbre binaire. Elle consiste à échanger un nœud avec l'un de ses fils.

Dans la rotation droite, un nœud devient le fils droit du nœud qui était son fils gauche.

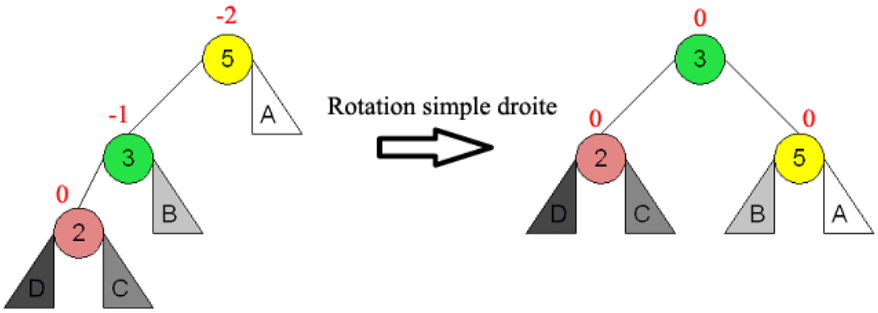
Dans la rotation gauche, un nœud devient le fils gauche du nœud qui était son fils droit.

Les rotations gauches et droites sont inverses l'une de l'autre.

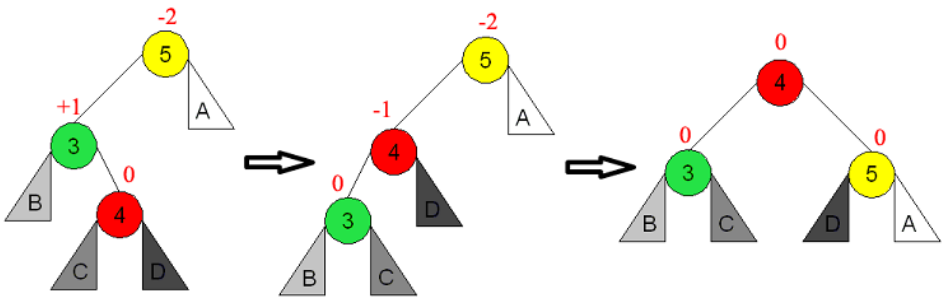
Les rotations ont la propriété de pouvoir être implémentées

en temps constant, et de préserver l'ordre infixe. En d'autres termes, si A est un arbre binaire de recherche, tout arbre obtenu à partir de A par une suite de rotations gauche ou droite d'un sous-arbre de A reste un arbre binaire de recherche.

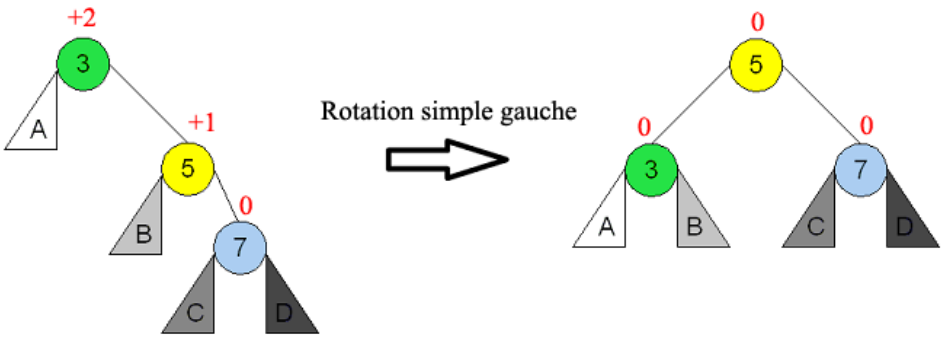
Une rotation simple droite est utilisée quand un nœud a un facteur d'équilibrage inférieur à -1 et que son fils gauche a un facteur d'équilibrage de -1.



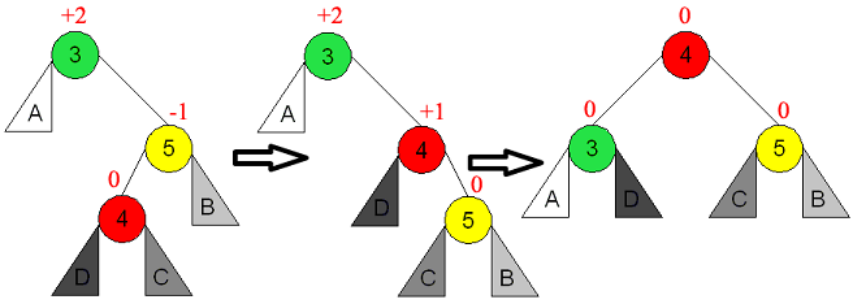
Une rotation double droite est utilisée quand un nœud a un facteur d'équilibrage inférieur à -1 et que son fils gauche a un facteur d'équilibrage de +1. La double rotation droite est une rotation simple gauche du sous-arbre gauche, suivi d'une rotation simple droite du nœud déséquilibré. Cette opération est aussi appelée parfois une rotation gauche-droite.



Une rotation simple gauche est utilisée quand un nœud a un facteur d'équilibrage supérieur à +1 et que son fils droit a un facteur d'équilibrage de +1.



Une rotation double gauche est utilisée quand un nœud a un facteur d'équilibrage supérieur à +1 et que son fils droit a un facteur d'équilibrage de -1. La double rotation gauche est une rotation simple droite du sous-arbre droit, suivi d'une rotation simple gauche du nœud déséquilibré. Cette opération est aussi appelée parfois une rotation droite-gauche.



8.7.2. Recherche

La recherche dans un arbre AVL se déroule exactement comme pour un arbre binaire de recherche, et comme la hauteur d'un arbre AVL est en $O(\log(n))$, elle se fait donc en $O(\log(n))$.

8.7.3. Insertion dans un arbre AVL

L'insertion dans un arbre AVL se déroule en deux étapes :

1. tout d'abord, on insère le nœud exactement de la même manière que dans un arbre binaire de recherche ;
2. puis on remonte depuis le nœud inséré vers la racine en effectuant une rotation sur chaque sous-arbre déséquilibré.

On peut montrer qu'il suffit d'une rotation simple ou d'une double rotation pour rééquilibrer un arbre AVL après une insertion.

La hauteur de l'arbre étant en $O(\log(n))$, et les rotations étant à temps constant, l'insertion se fait finalement en $O(\log(n))$.

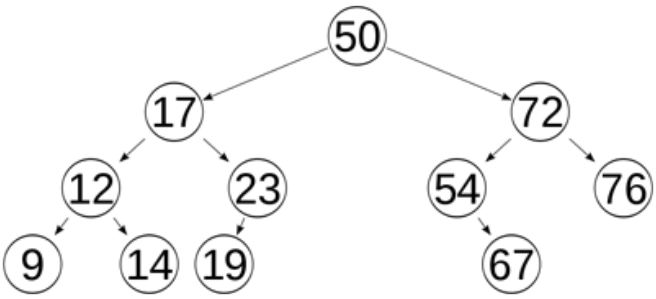
8.7.4. Suppression

Pour supprimer un nœud dans un arbre AVL, on procède comme avec un arbre binaire de recherche (voir § 8.6.3). On remonte ensuite le chemin depuis le parent du nœud enlevé jusqu'à la racine pour rééquilibrer les nœuds qui en ont besoin.

La suppression se fait aussi en $O(\log(n))$.

Exercice 8.7

Insérez dans l'arbre AVL ci-dessous les valeurs 20 et 70.



Exercice 8.8

Observez sur l'applet du site compagnon comment insérer, supprimer et rechercher une valeur dans un arbre AVL.

8.8. Tas

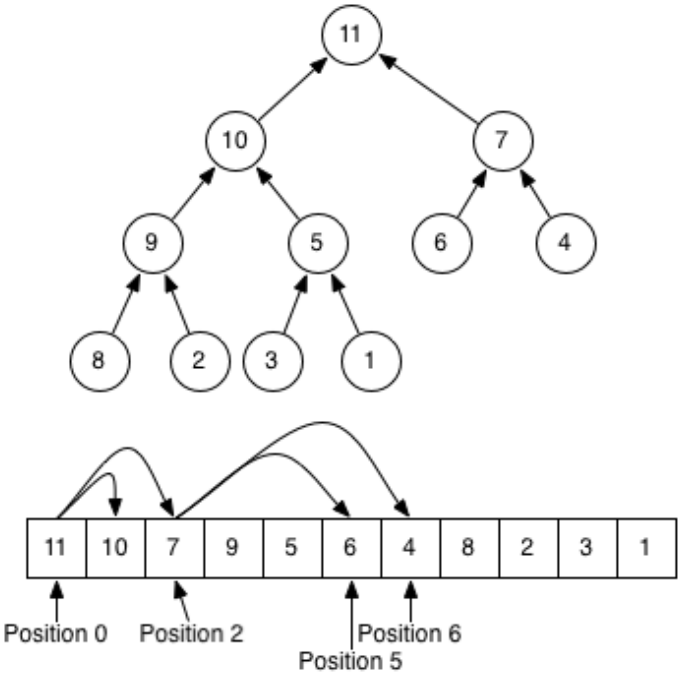
On dit qu'un arbre binaire complet est ordonné en tas[1] (on dit aussi parfois « monceau ») lorsque la propriété suivante est vérifiée :

Pour tous les nœuds de l'arbre, étiquette(*père*) ≥ étiquette(*fils*).

Cette propriété implique que la plus grande étiquette est située à la racine du tas. Ils sont ainsi très utilisés pour implémenter les files à priorités[2] car ils permettent des insertions en temps logarithmique et un accès direct au plus grand élément.

Le fait qu'un tas soit un arbre binaire complet permet de le représenter d'une manière efficace par un tableau unidimensionnel.

En effet, dans un tableau indicé à partir de 0, le père d'un nœud en position i a pour position l'entier inférieur ou égal à $\frac{i-1}{2}$; les enfants d'un nœud en position i sont situés à $2i+1$ pour le fils gauche et $2i+2$ pour le fils droit (voir § 8.4.2).



8.8.1. Primitives

Une caractéristique fondamentale de cette structure de données est que la propriété du tas peut être restaurée efficacement après la modification d'un nœud. Si la valeur du nœud est augmentée et qu'elle devient de ce fait supérieure à celle de son père, il suffit de l'échanger avec celle-ci, puis de continuer ce processus vers le haut jusqu'au rétablissement de la propriété du tas. Nous dirons que la valeur modifiée a été percolée jusqu'à sa nouvelle position.

Inversement, si la valeur du nœud est diminuée et qu'elle devient de ce fait inférieure à celle d'au moins un de ses fils, il suffit de l'échanger avec la plus grande des valeurs de ses fils, puis de continuer ce processus vers le bas jusqu'au rétablissement de la propriété du tas. Nous dirons que la valeur modifiée a été tamisée jusqu'à sa nouvelle position.

Plus formellement, la modification d'un nœud se fait par les algorithmes suivants :

1 *Heap* en anglais
2 Voir § 8.2. Le mot « file » est mal choisi, car une file de priorité ne ressemble pas à une file, ni par son aspect, ni par son comportement.

```
PROCEDURE percoler(T[0..n-1], i)

Données : tableau T, noeud i
Résultat : T est de nouveau un tas

k := i
REPETER
    j := k
    SI j > 0 ET T[(j-1) div 2] < T[k] ALORS
        k := (j-1) div 2
    echanger T[j] et T[k]
JUSQU'A j = k
```

```
PROCEDURE tamiser(T[0..n-1], i)

Données : tableau T, noeud i
Résultat : T est de nouveau un tas

k := i
REPETER
    j := k
    (* recherche du plus grand fils du noeud j *)
    SI 2j+1 <= n-1 ET T[2j+1] > T[k] ALORS
        k := 2j+1
    SI 2j+1 < n-1 ET T[2j+2] > T[k] ALORS
        k := 2j+2
    echanger T[j] et T[k]
JUSQU'A j = k
```

```
PROCEDURE modifier_tas(T[0..n-1], i, v)
(* on veut changer la valeur v du noeud i en préservant la propriété
du tas *)

Données : tableau T, noeud i, valeur v
Résultat : T est de nouveau un tas

x := T[i]
T[i] := v
SI v < x ALORS
    tamiser(T, i)
SINON
    percoler(T, i)
```

Cette propriété du tas en fait une structure de données idéale pour trouver le maximum, éliminer la racine, ajouter un nœud et modifier un nœud. Ce sont précisément les opérations voulues pour l'implémentation efficace d'une liste de priorité dynamique : la valeur d'un nœud indique la priorité de l'événement correspondant. L'événement le plus prioritaire se trouve toujours à la racine.

```
PROCEDURE eliminer_racine(T[0..n-1])

Données : tableau T
Résultat : T[0..n-2] est de nouveau un tas

T[0] := T[n-1]
tamiser(T,0)
supprimer(T[n-1])
```

```
PROCEDURE ajouter_noeud(T[0..n-1], v)

Données : tableau T, valeur v
Résultat : T[0..n] est de nouveau un tas

T[n] := v
percoler(T,n)
```

Exercice 8.9

Écrivez un programme Python qui gère un tas : représentez le tas par un tableau. puis programmez les procédures ou fonctions suivantes : `percoler`, `tamiser`, `modifier_tas`, `maximum`, `eliminer_racine`, `ajouter_noeud`.

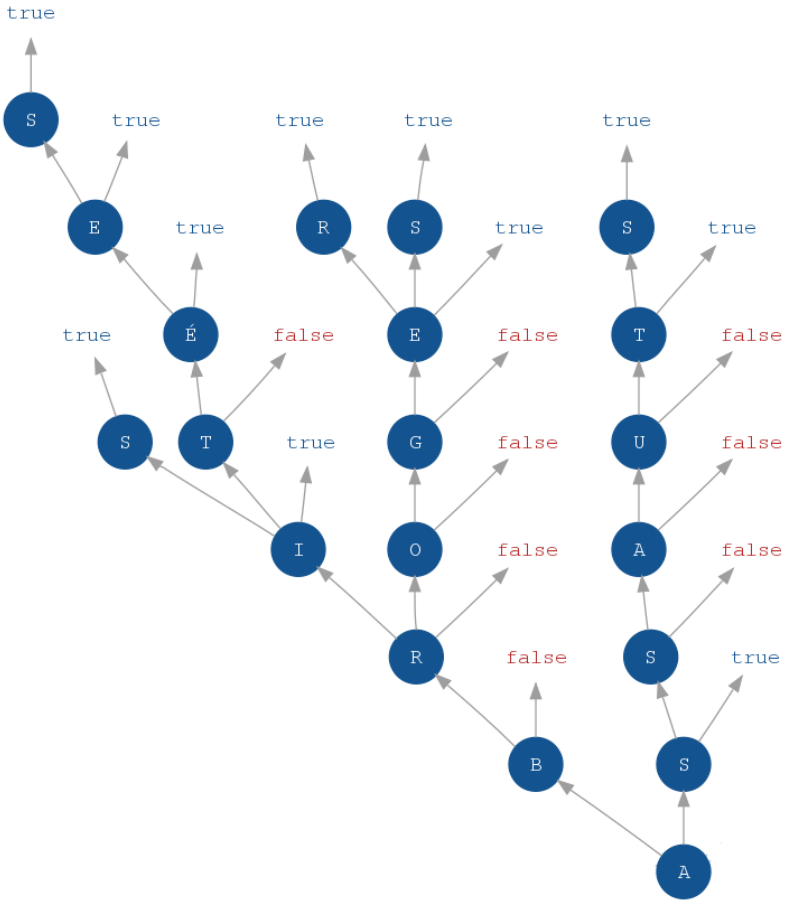
Écrivez enfin une procédure qui crée un tas de *n* valeurs tirées aléatoirement.

8.9. Trie

Le terme vient de *re*trie*vable memory*, mais l'usage dans la littérature francophone est d'utiliser le masculin.

Un trie ou arbre préfixe est un arbre numérique ordonné qui est utilisé pour stocker une table associative où les clés sont généralement des chaînes de caractères. Contrairement à un arbre binaire de recherche, aucun nœud dans le *trie* ne stocke la chaîne à laquelle il est associé. C'est la position du nœud dans l'arbre qui détermine la chaîne correspondante.

Pour tout nœud, ses descendants ont en commun le même préfixe. La racine est associée à la chaîne vide. Des valeurs ne sont pas attribuées à chaque nœud, mais uniquement aux feuilles et à certains nœuds internes se trouvant à une position qui désigne l'intégralité d'une chaîne correspondant à une clé.



Trie construit avec les mots :
abri, abris, abrité, abritée, abritées, abroge, abroger, abroges, as, assaut et assauts.

Les logiciels de traitement de texte présentent souvent une fonctionnalité d'**auto-complétion** des mots au cours de la frappe. Certains éditeurs de texte l'étendent aux commandes d'un langage de programmation. Dans tous les cas, l'auto-complétion est grandement facilitée par la représentation des mots sous forme de trie : il suffit en effet d'extraire le sous-arbre correspondant à un préfixe donné pour connaître toutes les suggestions en rapport avec le texte saisi !

8.9.1. Un trie en Python

```
class Trie:
    def __init__(self, letter=""): # au début, l'arbre est vide
        self.branches = {}
        self.letter = letter
        self.is_word_end = False

    def __getitem__(self, letter): # permet d'accéder à une valeur
        return self.branches.get(letter, None)

    def isWordEnd(self): # est-ce la fin d'un mot ?
        return self.is_word_end

    def setWordEnd(self, value=True):
        # marquer que c'est la fin d'un mot
        self.is_word_end = value

    def getLetter(self): # lire une lettre
        return self.letter

    def addBranch(self, branch): # ajouter une branche à l'arbre
        self.branches[branch.getLetter()] = branch

    def makeTrie(self, liste_mots):
        # crée un trie à partir d'une liste de mots
        for un_mot in liste_mots:
            noeud = self         # on se place au début de l'arbre
            mot = un_mot.rstrip()
            if len(mot)>1:
                for lettre in mot:    # lit les lettres une à une
                    if not noeud[lettre]:
                        # on crée une nouvelle branche
                        noeud.addBranch(Trie(lettre))
                    noeud = noeud[lettre]   # on avance dans l'arbre
                noeud.setWordEnd()          # marque de fin de mot
        return self

    def inTrie(self, mot):
        # mot est-il dans le Trie ?
        noeud = self
        for lettre in mot:
            if noeud[lettre] != None :
                noeud = noeud[lettre]
            else:
                return False
        else:
            if noeud.isWordEnd() :
                return True
            else:
                return False
```

8.9.2. Utilisation d'un trie comme dictionnaire

Le programme ci-dessous, utilise le trie défini au §8.10.1 et un fichier « dictionnaire.txt ». Ce dictionnaire contient 328'465 mots français non accentués entre 2 et 16 lettres, sans tiret et sans apostrophe. On entre simplement un mot français (sans accent) et le programme dit s'il est français.

```
from trie_fr import Trie

# ----------------------------------------------------

def creer_trie(dico):
```

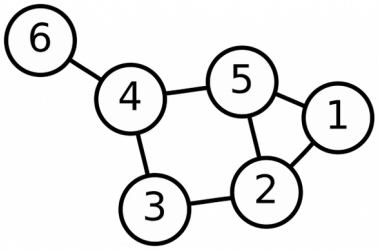
```
    print("Lecture du dictionnaire")
    fichier = open(dico, 'r')
    liste_mots = fichier.readlines()
    fichier.close()
    print("Création du Trie")
    trie = Trie()
    trie.makeTrie(liste_mots)
    print("Trie terminé")
    return trie

# -------------- programme principal --------------------

trie = creer_trie('dictionnaire.txt')
while True:
    donnees = input("Entrez un mot français : ")
    if donnees=="":
        break
    if trie.inTrie(donnees):
        print(donnees,"est dans le dictionnaire")
    else:
        print(donnees,"n'est pas dans le dictionnaire")
```

Comme vous pourrez le constater en testant ce programme, la création du trie prend un peu de temps (quelques secondes). Par contre, une fois le trie créé, la réponse à la question est instantanée.

8.10. Représentation des graphes

Un graphe fini $G = (V, E)$ est défini par l'ensemble fini $V = \{v_1, v_2, ..., v_n\}$ dont les éléments sont appelés sommets, et par l'ensemble fini $E = \{e_1, e_2, ..., e_m\}$ dont les éléments sont appelés arêtes. Une arête e de l'ensemble E est définie par une paire non-ordonnée de sommets, appelés les extrémités de e. Si l'arête e relie les sommets a et b, on dira que ces sommets sont adjacents.

Représentons le graphe ci-contre.



Ensemble des sommets :
$V = \{1, 2, 3, 4, 5, 6\}$

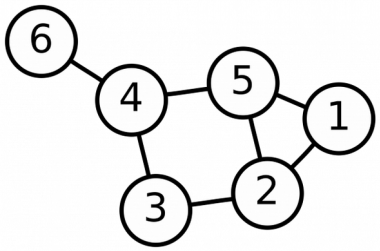
Ensemble des arêtes :
$E = \{(1, 2), (1, 5), (2, 5), (2, 3), (3, 4), (4, 5), (4, 6)\}$

Ce graphe peut se représenter graphiquement comme ci-contre (il y a une infinité de représentations possibles).

8.10.1. Listes d'adjacences

On peut aussi représenter un graphe en donnant pour chacun de ses sommets la liste des sommets auxquels il est adjacent. C'est la méthode qui est utilisée pour implémenter un graphe sur ordinateur.

Les listes d'adjacences du graphe ci-contre sont :



1 : 2, 5
2 : 1, 3, 5
3 : 2, 4
4 : 3, 5, 6
5 : 1, 2, 4
6 : 4

8.10.2. Trouver tous les mots d'une grille de Ruzzle

Ruzzle est un jeu de lettres aux règles simples : une grille de quatre fois quatre lettres, et deux minutes pour trouver le plus de mots possibles, avec pour seul impératif que les cases se touchent. On ne peut pas utiliser deux fois la même case, ce qui fait que la longueur maximale d'un mot sera de 16.

Pour pimenter le jeu, *Ruzzle* reprend également les cases bonus « lettre compte double », « lettre compte triple », « mot compte double » et « mot compte triple » du Scrabble. Pour gagner, il faudra obtenir le plus gros total de points !

Dans la version que nous allons voir, on ne tiendra pas compte de ces bonus, ni de la valeur des lettres ; on se contentera de trouver tous les mots possibles, du plus long au plus court.

Voici le programme complet que nos allons décortiquer :

```
from trie_fr import Trie

# --------- case de la grille de Ruzzle -------------

class Case:
    def __init__(self, value):
        self.value = value
        self.neighbors = []

    def addNeighbor(self, neighbor):
        # ajoute un voisin à la liste
        self.neighbors.append(neighbor)

    def getNeighbors(self):
        # renvoie la liste des voisins d'une case
        return self.neighbors

    def getValue(self):
        # renvoie la lettre d'une case
        return self.value

# -----------------------------------------------------

def creer_trie(dico):
    print("Lecture du dictionnaire")
    fichier = open(dico, 'r')
    liste_mots = fichier.readlines()
    fichier.close()
    print("Création du Trie")
    trie = Trie()
    trie.makeTrie(liste_mots)
    print("Trie terminé")
```

```
    return trie

def creer_connexions(grille):
    # crée les listes d'adjacences du graphe
    for m in range(16):
        case = grille[m]
        i, j = m//4, m%4
        for n in range(16):
            x, y = n//4, n%4
            if abs(i-x)<=1 and abs(j-y)<=1 and m != n:
                case.addNeighbor(grille[n])

def chercher(noeud, case, vus=[]):
    # cherche les mots français dans le Trie
    trouves = []
    if noeud == None:
        return trouves
    if noeud.isWordEnd():
        trouves.append(noeud.getLetter())
    vus.append(case)
    for voisin in case.getNeighbors():
        if voisin not in vus:
            results = chercher(noeud[voisin.getValue()], voisin, vus)
            trouves.extend([noeud.getLetter()+ending for ending in results])
    vus.remove(case)
    return trouves

# -------------- programme principal --------------------

trie = creer_trie('ruzzle_dictionnaire.txt')
while True:
    donnees = input("Entrez la grille ligne par ligne : ")
    while len(donnees)!=16 and len(donnees)!=0:
        donnees = input("Entrez la grille ligne par ligne : ")
    if donnees=="": # taper 'return' pour sortir de la boucle infinie
        break
    donnees = donnees.lower()
    grille = [Case(lettre) for lettre in donnees]
    creer_connexions(grille)
    resultats = []
    for case in grille: # première lettre du mot à chercher
        noeud = trie    # on se place au début du Trie
        mots_trouves = chercher(noeud[case.getValue()], case)
        if len(mots_trouves) > 0:
            resultats.extend(mots_trouves)
    resultats = list(set(resultats))  # élimine les doublons
    output = sorted(resultats, key=lambda mot: [len(mot)], reverse=True)
    print(len(output),"mots trouvés")
    print(output)
    print()
```

On voit sur la première ligne que le programme importe la classe `Trie` du §8.9.1. La procédure `creer_trie` a déjà été utilisée au §8.9.2

On a créé une classe `case`. Une case contient une lettre et la liste des lettres qui sont sur des cases voisines (pas voisin, on entend qui jouxte horizontalement, verticalement ou en diagonale).

```
class Case:
    def __init__(self, value):
        self.value = value
        self.neighbors = []

    def addNeighbor(self, neighbor):
        # ajoute un voisin à la liste
        self.neighbors.append(neighbor)
```

```
    def getNeighbors(self):
        # renvoie la liste des voisins d'une case
        return self.neighbors

    def getValue(self):
        # renvoie la lettre d'une case
        return self.value
```

La liste des lettres voisines d'une case est créée par la procédure `creer_connexions(grille)`. La grille est implémentée par une liste de 16 éléments.

Pour chacune des 16 lettres de la grille, on calcule sa ligne `i` et sa colonne `j` en fonction de la position `m` dans `grille`. On parcourt ensuite les 15 autres cases de la grille et on calcule leur ligne `x` et leur colonne `y`. Pour être voisin de cette case, il faut que `|i-x|` soit inférieur ou égal à 1 et de même pour `|j-y|`.

```
def creer_connexions(grille):
    # crée les listes d'adjacences du graphe
    for m in range(16):
        case = grille[m]
        i, j = m//4, m%4
        for n in range(16):
            x, y = n//4, n%4
            if abs(i-x)<=1 and abs(j-y)<=1 and m!=n:
                case.addNeighbor(grille[n])
```

Voici les listes que l'on obtient pour la grille

```
u a e e
s l r u
i i e a
o t s y
```

```
u : a s l
a : u e s l r
e : a e l r u
e : e r u
s : u a l i i
l : u a e s r i i e
r : a e e l u i e a
u : e e r e a
i : s l i o t
i : s l r i e o t s
e : l r u i a t s y
a : r u e s y
o : i i t
t : i i e o s
s : i e a t y
y : e a s
```

La fonction la plus délicate à analyser est évidemment `chercher`, car elle est récursive.

On a vu comment chercher un mot dans le trie (§8.9.2). Ici, le problème est un peu différent. En effet, il ne serait pas très efficace de chercher toutes les suites de lettres possibles (il y en a 12'029'624 selon [7]) et vérifier ensuite lesquels sont des mots français dans le trie. Par expérience, il y a généralement entre 250 et 400 mots valides dans une grille.

On va faire l'inverse : on va parcourir le trie en utilisant les voisins.

On part du sommet du trie avec la première lettre de la grille (celle en haut à gauche).

On choisit son premier voisin dans la grille et on regarde si la suite de ces deux lettres existe dans le trie.

Si la suite n'existe pas, on abandonne cette branche du trie, on marque ce voisin comme vu et on essaie un autre voisin.

Si la suite existe, on regarde si c'est un mot complet. Si oui, on le mémorise. Si non, on prend un voisin de la deuxième lettre et on analyse la suite des trois lettres.

Et ainsi de suite…

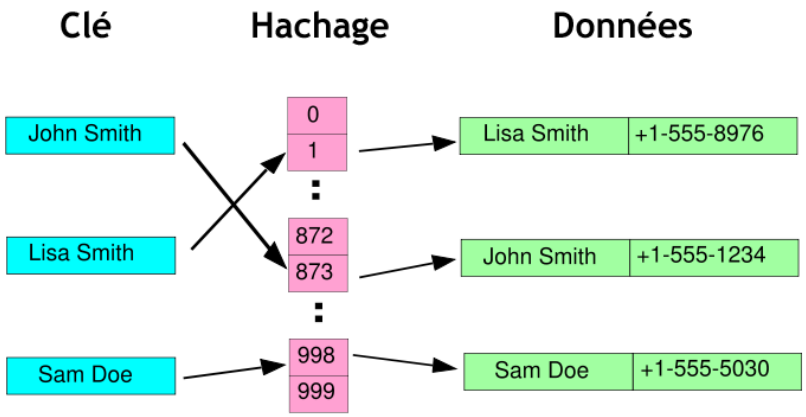
```
def chercher(noeud, case, vus=[]):
    # cherche les mots français dans le Trie
    trouves = []
    if noeud == None:
        return trouves
    if noeud.isWordEnd():
        trouves.append(noeud.getLetter())
    vus.append(case)
    for voisin in case.getNeighbors():
        if voisin not in vus:
            results = chercher(noeud[voisin.getValue()], voisin, vus)
            trouves.extend([noeud.getLetter()+ending for ending in results])
    vus.remove(case)
    return trouves
```

On parcourt selon ce principe le trie 16 fois, car il y a 16 cases possibles pour la première lettre. On élimine finalement les doublons dans la liste des mots trouvés et on affiche tous les mots du plus long au plus court.

```
    for case in grille: # première lettre du mot à chercher
        noeud = trie    # on se place au début du Trie
        mots_trouves = chercher(noeud[case.getValue()], case)
        if len(mots_trouves) > 0:
            resultats.extend(mots_trouves)
    resultats = list(set(resultats))  # élimine les doublons
    output = sorted(resultats, key=lambda mot: [len(mot)], reverse=True)
    print(len(output),"mots trouvés")
    print(output)
```

8.11. Table de hachage

Une table de hachage (*hash table* en anglais) est une structure de donnée permettant d'associer une valeur à une clé. Il s'agit d'un tableau ne comportant pas d'ordre (un tableau est indexé par des entiers). L'accès à un élément se fait en transformant la clé en une valeur de hachage (ou simplement hachage) par l'intermédiaire d'une fonction de hachage. Le hachage est un nombre qui permet la localisation des éléments dans le tableau, typiquement le hachage est l'indice de l'élément dans le tableau. Une case dans le tableau est appelée alvéole.



Une table de hachage implémentant un annuaire téléphonique

Différentes opérations peuvent être effectuées sur une table de hachage :

- création d'une table de hachage ;
- insertion d'un nouveau couple (clé, valeur) ;

- suppression d'un élément ;
- recherche de la valeur associée à une clé (dans l'exemple de l'annuaire, retrouver le numéro de téléphone d'une personne) ;
- destruction d'une table de hachage (pour libérer la mémoire occupée).

Tout comme les tableaux, les tables de hachage permettent un accès en $O(1)$ en moyenne, quel que soit le nombre d'éléments dans la table. Toutefois, le temps d'accès dans le pire des cas peut être de $O(n)$. Comparées aux autres tableaux associatifs, les tables de hachage sont surtout utiles lorsque le nombre d'entrées est très important.

La position des éléments dans une table de hachage est aléatoire. Cette structure n'est donc pas adaptée pour accéder à des données triées.

8.11.1. Choix d'une bonne fonction de hachage

Le fait de créer un hachage à partir d'une clé peut engendrer un problème de collision, c'est-à-dire qu'à partir de deux clés différentes, la fonction de hachage pourrait renvoyer la même valeur de hachage, et donc par conséquent donner accès à la même position dans le tableau. Pour minimiser les risques de collisions, il faut donc choisir soigneusement sa fonction de hachage.

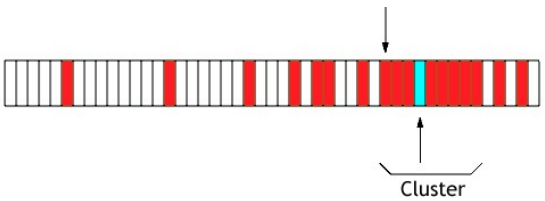
Les collisions étant en général résolues par des méthodes de recherche linéaire, une mauvaise fonction de hachage (produisant beaucoup de collisions) va fortement dégrader la rapidité de la recherche.

Le calcul du hachage se fait parfois en deux temps :

1. une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la donnée d'origine
2. ce nombre entier est converti en une position possible de la table, en général en calculant le reste modulo la taille de la table.

Les tailles des tables de hachage sont souvent des nombres premiers, afin d'éviter les problèmes de diviseurs communs, qui créeraient un nombre important de collisions. Une alternative est d'utiliser une puissance de deux, ce qui permet de réaliser l'opération modulo par de simples décalages, et donc de gagner en rapidité.

Un problème fréquent et surprenant est le phénomène de « clustering » (*grumelage*) qui désigne le fait que des valeurs de hachage se retrouvent côte à côte dans la table, formant des « clusters » (« grappes » en français). Ceci est très pénalisant pour les techniques de résolution des collisions par adressage ouvert (voir ci-après). Les fonctions de hachage réalisant une distribution uniforme des hachages sont donc les meilleures, mais sont en pratique difficile à trouver.



La fonction de hash primaire (la flèche en haut) calcule une adresse et génère une collision. La première case libre en ordre croissant, ici en bleu, est trouvée et utilisée, consolidant ainsi deux clusters, provoquant une congestion supplémentaire.

Exercice 8.10

Programmez en Python la fonction `hashCode(s)` de la chaîne s de longueur n, qui renvoie le hachage ord(s[0])*32^(n–1) + ord(s[1])*32^(n–2) + ... + ord(s[n–1]) mod N, où **ord** est la fonction qui renvoie le code ASCII d'un caractère et N la taille de la table de hachage.

Testez cette fonction avec les noms « Chico », « Groucho », « Gummo », « Harpo », « Zeppo » et $N = 5$, puis $N = 11$, puis $N = 12$.

8.11.2. Résolution des collisions

Lorsque deux clés ont la même valeur de hachage (collision), ces clés ne peuvent être stockées à la même position, on doit alors employer une stratégie de résolution des collisions.

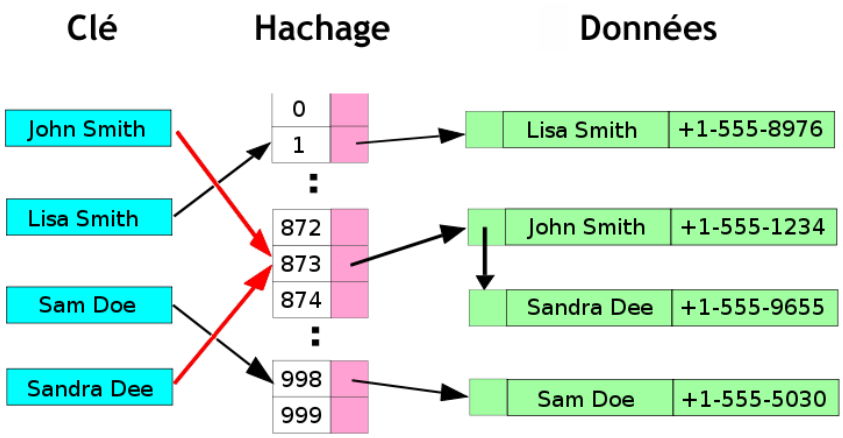
Pour donner une idée de l'importance d'une bonne méthode de résolution des collisions, considérons ce résultat issu du paradoxe des anniversaires[3]. Même si nous sommes dans le cas le plus favorable où la fonction de hachage a une distribution uniforme, il y a 95 % de chance d'avoir une collision dans une table de taille 1 million avant qu'elle ne contienne 2500 éléments.

De nombreuses stratégies de résolution des collisions existent mais les plus connues et utilisées sont le chaînage et l'adressage ouvert.

Chaînage

Cette méthode est la plus simple. Chaque case de la table est en fait une liste chaînée des clés qui ont le même hachage. Une fois la case trouvée, la recherche est alors linéaire en la taille de la liste chaînée. Dans le pire des cas où la fonction de hachage renvoie toujours la même valeur de hachage quelle que soit la clé, la table de hachage devient alors une liste chaînée, et le temps de recherche est en $O(n)$. L'avantage du chaînage est que la suppression d'une clé est facile ainsi que la recherche.

D'autres structures de données que les listes chaînées peuvent être utilisées. En utilisant un arbre équilibré, le coût théorique de recherche dans le pire des cas est en $O(\log(n))$. Cependant, la liste étant supposée être courte, cette approche est en général peu efficace à moins d'utiliser la table à sa pleine capacité, ou d'avoir un fort taux de collisions.



Résolution des collisions par chaînage

Adressage ouvert

L'adressage ouvert consiste, dans le cas d'une collision, à stocker les valeurs de hachage dans d'autres alvéoles vides.

Soit $h(k,i)$ une fonction de sondage. Une méthode de sondage consiste à essayer les alvéoles d'indice $h(k,0)$, $h(k,1)$, ..., jusqu'à ce qu'on trouve une alvéole vide.

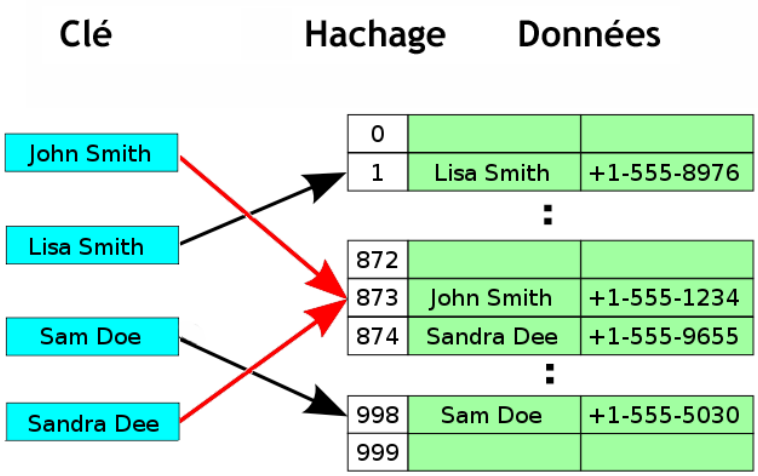
Il y a trois méthodes de sondage :

Sondage linéaire

Soit $H : U \rightarrow \{0, \ldots N-1\}$ une fonction de hachage.

La fonction de sondage linéaire sera : $h(k,i) = (H(k) + i) \bmod N$, avec $i = 0, 1, \ldots, N-1$.

3 Le **paradoxe des anniversaires** est à l'origine une estimation probabiliste du nombre de personnes que l'on doit réunir pour avoir une chance sur deux que deux personnes de ce groupe aient leur anniversaire le même jour de l'année. Il se trouve que ce nombre est 23, ce qui choque l'intuition. À partir d'un groupe de 57 personnes, la probabilité est supérieure à 99 %.



Résolution des collisions par adressage ouvert et sondage linéaire

Sondage quadratique
Soit $H : U \rightarrow \{0, \ldots, N-1\}$ une fonction de hachage.
La fonction de sondage quadratique sera :
$h(k,i) = (H(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod N$, avec $i = 0, 1, ..., N-1$ et $c_2 \neq 0$

Double hachage
Soient H_1 et $H_2 : U \rightarrow \{0, \ldots, N-1\}$ deux fonctions de hachage. Il faudra faire en sorte que H_2 ne soit jamais égal à 0.
La fonction de sondage par double hachage sera :
$h(k,i) = (H_1(k) + i \cdot H_2(k)) \bmod N$, avec $i = 0, 1, ..., N-1$.

Recherche

Lors d'une recherche, si l'alvéole obtenue par hachage direct ne permet pas d'obtenir la bonne clé, une recherche sur les cases obtenues par une méthode de sondage est effectuée jusqu'à trouver la clé, ou tomber sur une alvéole vide, ce qui indique qu'aucune clé de ce type n'appartient à la table.

Suppression

Cette manière de faire s'appelle en anglais « lazy deletion », en français « suppression paresseuse ».

Dans un adressage ouvert, la suppression d'un élément de la table de hachage est délicate. Dans le schéma ci-dessus, si l'on supprime « John Smith » sans précaution, on ne retrouvera plus « Sandra Dee ». La manière la plus simple de s'en sortir est de ne pas vider l'alvéole où se trouvait « John Smith », mais d'y placer le mot « Supprimé ». On distinguera ainsi les alvéoles vides des alvéoles où un nom a été effacé : une alvéole contenant « Supprimé » sera considérée comme occupée lors d'une suppression, mais vide lors d'une insertion.

Facteur de charge

Une indication critique des performances d'une table de hachage est le facteur de charge qui est la proportion de cases utilisées dans la table. Plus le facteur de charge est proche de 100 %, plus le nombre de sondages à effectuer devient important. Lorsque la table est presque pleine, les algorithmes de sondage peuvent même échouer : ils peuvent ne plus trouver d'alvéole vide, alors qu'il y en a encore.

Le facteur de charge est en général limité à 80 %, même en disposant d'une bonne fonction de hachage. Des facteurs de charge faibles ne sont pas pour autant significatifs de bonne performance, en particulier si la fonction de hachage est mauvaise et génère du clustering.

Avantages et inconvénients

Un des avantages du chaînage est que la suppression d'une clé est facile. Contrairement aux sondage et double hachage, vous devrez utiliser des structures externes au tableau, en l'occurrence des files. Autant ce désavantage en espace mémoire est comblé dans des situations où les deux autres méthodes dégénèrent (avec des tableaux bien remplis), autant ces dernières sont à préférer pour un remplissage du tableau modéré.

Exercice 8.11

On souhaite stocker des nombres entiers positifs dans une table de hachage. La table possédant 13 emplacements (numérotés de 0 à 12), on utilise la clef suivante : $H_1(x) = x \bmod 13$.

1. Calculez la valeur de la clé pour chacun des 10 éléments de la liste suivante :
   15, 22, 127, 4, 56, 89, 17, 26, 5, 78.
2. Insérez ces éléments par ordre d'apparition dans la table de hachage. Résolvez les collisions par chaînage. Dessinez la table obtenue.
3. Idem en résolvant les collisions par un sondage linéaire $h(k,i) = (H_1(k) + i) \bmod 13$.
4. Idem en utilisant un sondage quadratique : $h(k,i) = (H_1(k) + i^2) \bmod 13$.
5. Idem en utilisant un double hachage :
   $h(k,i) = (H_1(k) + i \cdot H_2(k)) \bmod 13$, avec $H_2(x) = (2x+1) \bmod 11 + 1$.

Exercice 8.12

Utilisez la fonction de hachage programmée dans l'exercice 8.10 dans une table de taille $N = 12$.

- Implémentez une classe « table_de_hachage ».
- Écrivez une méthode « inserer », qui insère un nouvel abonné (nom et numéro de téléphone) dans l'annuaire.
- Écrivez une méthode « rechercher », qui affiche le numéro de téléphone d'un abonné. Si plusieurs abonnés portent le même nom, la méthode affichera les différents numéros.
- Écrivez une méthode « supprimer », qui retire un abonné de l'annuaire.

Testez ensuite votre programme avec les données suivantes :

```
Chico     0324661193          Groucho  0324667543
Gummo     0324664578          Harpo    0324668501
Zeppo     0324660031          Zeppo    0324660032
```

Faites cet exercice deux fois : une fois en résolvant les collisions par chaînage et l'autre fois en les résolvant par adressage ouvert (avec un sondage linéaire).

Exercice 8.13

Reliez chaque tâche à effectuer à la structure de données la plus adéquate.

stocker les dernières pages web visitées dans un navigateur	**1**	**A**	file
écrire tous les mots dont le début est connu	**2**	**B**	tas
retrouver instantanément un nom dans un annuaire	**3**	**C**	arbre AVL
connaître à tout moment le prochain événement à traiter dans une simulation (p. ex. la collision entre 2 boules de billard)	**4**	**D**	trie
trier des nombres donnés au fur et à mesure	**5**	**E**	table de hachage
traiter les demandes sur une imprimante branchée en réseau	**6**	**F**	pile

Sources

[1] Wikipédia, « Structures de données », <http://fr.wikipedia.org/wiki/Catégorie:Structure_de_données>

[2] Wikipédia, « Arbre (Structures de données) », <http://fr.wikipedia.org/wiki/Catégorie:Arbre_(structure_de_données)>

[3] Wikipédia, « Arbre binaire », <http://fr.wikipedia.org/wiki/Arbre_binaire>

[4] Wikipédia, « Arbre binaire de recherche », <http://fr.wikipedia.org/wiki/Arbre_binaire_de_recherche>

[5] Wikipédia, « Arbre AVL », <http://fr.wikipedia.org/wiki/Arbre_AVL>

[6] Ferraro Tyler, « Ruzzle-Solver », <https://github.com/TylerFerraro/Ruzzle-Solver>

[7] Mannino Miro, « Ruzzle Solver Algorithm », <http://miromannino.com/ruzzle-solver-algorithm/>

[8] Wikipédia, « Table de hachage », <http://fr.wikipedia.org/wiki/Table_de_hachage>

[9] Wikipédia, « Fonction de hachage », <http://fr.wikipedia.org/wiki/Fonction_de_hachage>

[10] Nicod Jean-Marc, « Les tables de hachage », <http://ifc.univ-fcomte.fr/~nicod/slidesHashTableL3.pdf>

[11] Wikipédia, « Théorie de la complexité des algorithmes », <http://fr.wikipedia.org/wiki/Théorie_de_la_complexité_des_algorithmes>