Thème info 1 – Arbres binaires et tri par tas

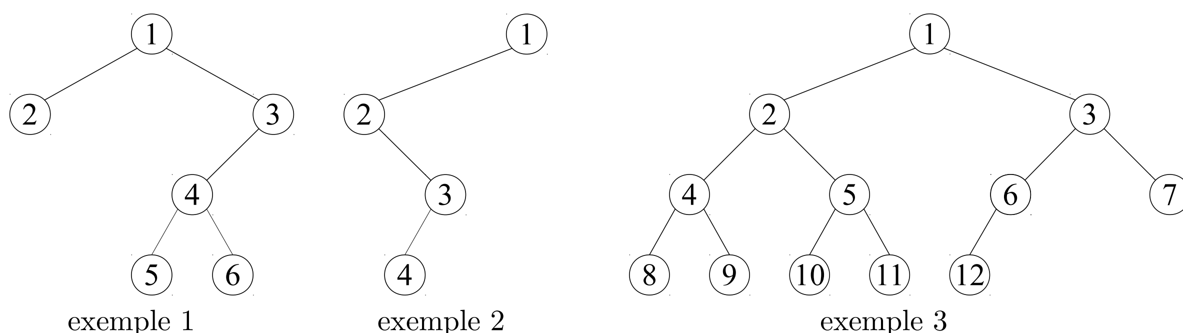
I - Structure d'arbre binaire - représentations en Python

1) Le type de données *arbre binaire*

Par définition (récursive !) un *arbre binaire étiqueté* (par des éléments d'un ensemble E) est soit l'arbre vide [ ], soit de la forme $a = [e, fils_g, fils_d]$ où e est un élément de E *(l'étiquette)* et $fils_g$, $fils_d$ sont deux arbres binaires étiquetés disjoints, le *fils gauche* et le *fils droit* de l'arbre a. On dit aussi que a est le *père* de ses deux fils. Cette terminologie est évidemment empruntée au contexte des arbres généalogiques ; on parle aussi de *sous-arbre gauche* et de *sous-arbre droit.*
L'essentiel du vocabulaire est en effet issu de la botanique :

- les divers triplets de la forme $[e, fils_g, fils_d]$ constituant l'arbre sont les *nœuds*
- le nœud initiant l'arbre entier (le seul qui n'ait pas de père) est la *racine*, les autres nœuds ont un unique père
- les nœuds "terminaux" dont les deux fils sont vides sont les *feuilles* !

Voici quelques exemples, où l'on voit qu'un arbre est parfois incomplet ou déséquilibré... Le nœud 1 est la racine, souvent représentée en haut de l'arbre... Par convention on ne représente pas les sous-arbres vides.



exemple 1 exemple 2 exemple 3

Les étiquettes sont indépendantes de la structure de l'arbre, mais ce sont elles qui servent à stocker des données.

La *hauteur* (ou la *profondeur*, ou encore le *niveau*) d'un nœud est définie récursivement par :

$$h(x) = 0 \quad \text{si } x \text{ est la racine, } h(x) = 1 + h(y) \quad \text{si } y \text{ est le père de } x.$$

La *hauteur d'un arbre* non vide est la plus grande des hauteurs de ses nœuds.

Question 1 : montrer que la hauteur h d'un arbre comprenant n nœuds vérifie :

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1.$$

Définition : un arbre binaire est dit *parfait* si tous ses niveaux sont remplis, sauf éventuellement le dernier niveau, auquel cas les nœuds (feuilles !) du dernier niveau sont groupés le plus à gauche possible (cf. exemple 3 ci-dessus).

Propriété : la hauteur d'un arbre parfait à n nœuds est exactement $\lfloor \log_2 n \rfloor$.

NB : on peut définir plus généralement des arbres admettant un nombre arbitraire (mais fini !) de fils, de la forme $[e, liste_de_fils]$ où $liste_de_fils$ est une liste d'arbres...

2) Champs d'intervention

Les arbres sont utilisés dans de nombreux contextes : la généalogie bien sûr, les tournois sportifs, les probabilités, le stockage d'expressions algébriques (les feuilles contenant les opérandes et les autres nœuds les opérateurs, les fils d'un opérateur représentent les arguments de l'opération correspondante), etc. Voir aussi la section **II** !

3) Les primitives du type abstrait de données "arbre binaire"

Les primitives permettant de créer et de manipuler (souvent récursivement) les arbres binaires sont :

- `est_vide(a)` : renvoie `Vrai` si l'arbre `a` est vide, `Faux` sinon ;
- `arbre_vide()` : renvoie l'arbre vide ;
- `cons(e,g,d)` : renvoie l'arbre d'étiquette `e`, de fils gauche (*resp.* droit) `g` (*resp.* `d`) ;
- `contenu(a)` : renvoie l'étiquette de l'arbre **non vide** `a` ;
- `f_g(a)` (*resp.* `f_d(a)`) : renvoie le fils gauche (*resp.* droit) de l'arbre **non vide** `a`.

4) Implémentation en Python

a) Représentation par une liste

La souplesse de la structure de liste en Python permet de coller à la définition ci-dessus.
L'arbre de l'exemple 2 peut ainsi être représenté par la liste

$$[1, [2, [\,], [3, [4, [\,], [\,]], [\,]]], [\,]]$$

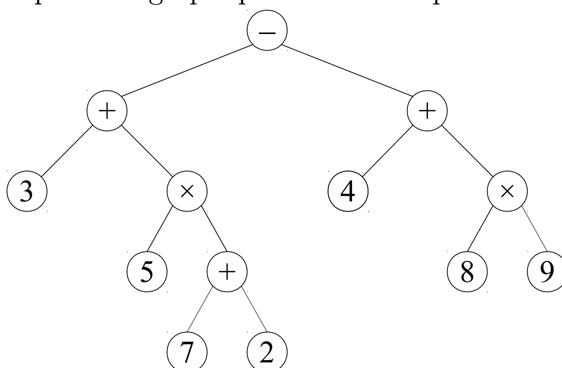
(où l'on voit l'intérêt des représentations graphiques, pour l'œil humain...).

Question 2 : donner la liste associée à l'exemple 1 du § **1)**.

Question 3 : écrire en Python les primitives du § **3)** dans cette représentation.

Question 4 : programmer (récursivement !) dans ce contexte le calcul de la hauteur d'un arbre.

Question 5 : l'arbre suivant représente graphiquement une expression algébrique.



Programmer le *parcours* (récursif !) d'un tel arbre pour écrire dans une chaîne de caractères l'expression algébrique associée. Il sera sage d'ajouter des parenthèses (cf. le chapitre 3...). Tester le programme avec l'exemple ci-dessus.

NB : l'opération inverse consistant à reconstruire l'arbre à partir de la chaîne de caractères est plus délicate (cf. chapitre 3).

b) Représentation par un tableau

On peut convenir de stocker les étiquettes dans un tableau t, en numérotant (**à partir de 1**) les nœuds de haut en bas et de gauche à droite pour chaque niveau, $t\,[k]$ contenant l'étiquette du nœud n° k s'il existe, une valeur "artificielle" sinon (valeur absente de l'ensemble des étiquettes possibles, par exemple -1 si l'on stocke des valeurs positives...).

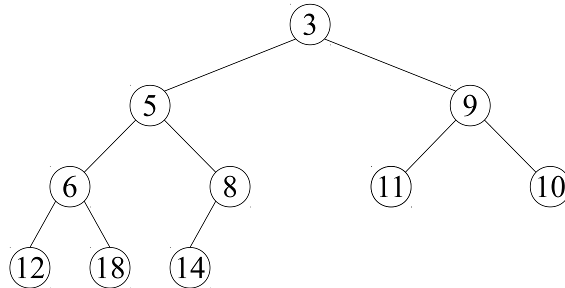
Dans ce cas, le niveau i correspond aux indices k de l'intervalle $[\![2^i, 2^{i+1}-1]\!]$ (la racine est au niveau zéro, rappelons que l'on n'utilise pas $t\,[0]$...). Les fils gauche et droit (s'ils existent) du nœud n° k ont pour indices $2k$ et $2k+1$; pour $k > 1$, le père du nœud n° k a pour indice $\lfloor k/2 \rfloor$ (utiliser `k//2` en Python). Cf. l'exemple 3 du § **1**, où les étiquettes sont les indices des nœuds selon ce principe).

Cette représentation est avantageuse dans le cas des arbres parfaits (on stocke les n étiquettes dans $t\,[1]$,..., $t\,[n]$). Réciproquement on stocke n valeurs dans un arbre de hauteur $\lfloor \log_2 n \rfloor$, cf. section **II**. Toutefois elle a de gros défauts dans le cas général : dans le pire des cas, on peut avoir besoin d'un tableau de taille 2^n pour stocker n nœuds ! De plus, la structure récursive de l'arbre binaire n'apparaît pas clairement.

II - Tri par tas (*heapsort* en anglais)

1) Notion de tas

Un *tas* est un arbre binaire parfait vérifiant la condition suivante : l'étiquette de chaque nœud (autre que la racine !) est supérieure ou égale à celle de son père. Dans ce cas l'étiquette de la racine est la plus petite de toutes les étiquettes. Par exemple, l'arbre parfait représenté par le tableau $[0, 3, 5, 9, 6, 8, 11, 10, 12, 18, 14]$ est un tas (la valeur d'indice 0 n'est pas utilisée) ; cela se voit mieux ainsi :



Cette notion a été introduite par J.W.J. WILLIAMS en 1964, pour expliciter son algorithme de *tri par tas*, dont l'idée consiste, pour trier un tableau, à créer un tas avec les valeurs contenues dans le tableau, puis à extraire les valeurs dudit tas de sorte qu'elles se retrouvent triées. Les questions suivantes montrent que l'on peut effectuer ces deux phases avec une complexité en nombre de comparaisons qui est un $O(n \log_2 n)$, ce qui donne un algorithme de tri globalement en $O(n \log_2 n)$, cela dans tous les cas.

Nous allons voir que l'implémentation se fait très bien avec les tableaux (et de façon **itérative**…), mais la vision arborescente est fondamentale pour imaginer et visualiser les algorithmes. Pour la suite, on suppose donné un tableau initial ti contenant les n valeurs à trier. Pour stocker n valeurs dans un tas, on utilisera un tableau de taille $n+1$, les cases "utiles" étant indexées de 1 à n comme expliqué précédemment.

2) Tri par tas, première version

La version la plus naturelle consiste à utiliser un tableau auxilaire *tas* que l'on remplira avec les données initiales, avant de le vider.

Question 6 : écrire une procédure *entasser*, recevant pour paramètres un tas t et une valeur x et ajoutant x dans t en préservant la structure de tas. On pourra ajouter x à la fin du tas et le faire "remonter vers la racine" par échanges successifs $fils \leftrightarrow père$ autant que nécessaire (cf. l'idée du tri par insertion…).

Question 7 : écrire une procédure *suppr_min*, recevant pour paramètres un tas t et supprimant de t sa racine en préservant la structure de tas. On pourra remplacer la racine par la dernière valeur de t, supprimer cette dernière valeur et faire "redescendre" cette nouvelle racine dans le tas à l'aide d'échanges $père \leftrightarrow fils$ (attention à choisir le bon fils lorsqu'il y en a deux…).

Question 8 : déduire des deux questions précédentes une fonction *tri_tas1*, recevant pour paramètre le tableau initial ti et le renvoyant trié. Seules les valeurs indexées à partir de 1 seront triées.

3) Tri par tas en place

Question 9 : modifier les programmes des questions **6**, **7**, **8** afin d'obtenir une fonction *tri_tas2* triant "en place" le tableau reçu en paramètre (*i.e.* sans utiliser de tableau auxiliaire). Là encore, seules les valeurs indexées à partir de 1 seront triées.

4) Optimisation de la création du tas

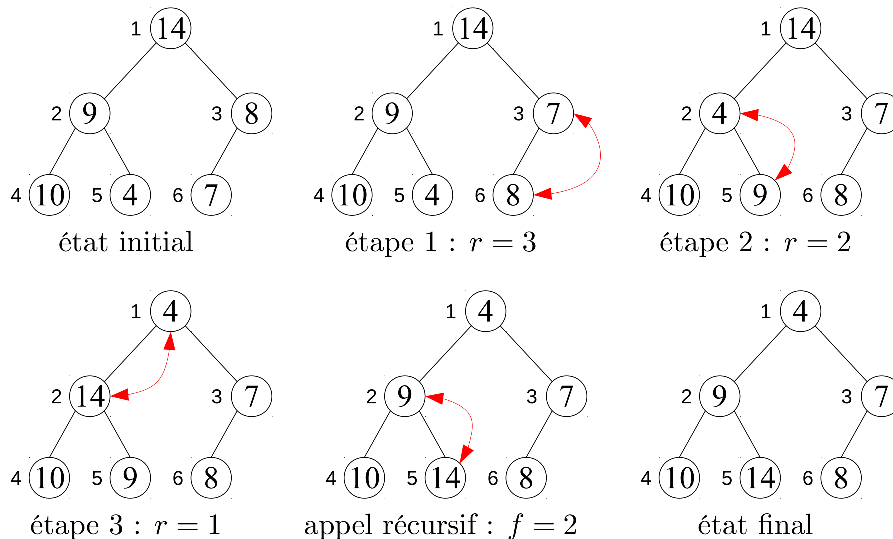
Si l'on s'y prend bien, la complexité de la fonction *entasser* de la question **6** est majorée par $\log_2 i$, où i est le nombre de valeurs dans le tas initial. Il en résulte une création progressive du tas contenant les n valeurs avec une complexité majorée par $\log_2(n!)$, soit un $O(n \ln n)$ grâce à la formule de Stirling. En fait cette création du tas peut se faire avec une complexité linéaire.

Pour cela, on part du tableau t contenant les valeurs à trier (initialement dans un ordre quelconque) dans les cases indexées de 1 à n, on pose $d = \lfloor n/2 \rfloor$, indice du nœud le plus à droite de l'avant-dernier niveau du tas. L'idée est d'ordonner successivement les sous-arbres dont les racines sont les nœuds d'indices $d, d-1, \ldots, 1$.

Pour ce faire on écrit une fonction $ordonner(t, r)$ qui traite le sous-arbre dont la racine a pour indice r, en supposant que les sous-arbres de ce dernier ont déjà été traités (ce qui justifiera une preuve par récurrence descendante !). Le traitement se déroule ainsi :

- on recherche l'indice f du plus petit fils de $t\,[r]$
- si besoin on échange $t\,[r]$ et $t\,[f]$ et dans ce cas, si $t\,[f]$ n'est pas une feuille, on appelle (récursivement) $ordonner\,(t, f)$ puisque le sous-arbre de racine $t\,[f]$ n'est peut-être plus un tas...

Voici un exemple à partir du tableau $t = [0, 14, 9, 8, 4, 10, 7]$ où l'on appelle successivement *ordonner* avec $r = 3, 2, 1$ (on a fait figurer son numéro à gauche de chaque nœud) :



état initial — étape 1 : $r = 3$ — étape 2 : $r = 2$

étape 3 : $r = 1$ — appel récursif : $f = 2$ — état final

Question 10 : programmer la fonction *ordonner* et l'utiliser pour écrire une fonction tri_tas3.

Question 11 (facultative) : montrer que la complexité de la création du tas à l'aide de la fonction *ordonner* est un $O\,(n)$.

III - Affichage d'un arbre parfait en mode texte

Question 12 (facultative) : programmer l'affichage (en mode texte) de l'arbre parfait représenté comme au **II** par un tableau. Par exemple, le tableau $[0, 3, 5, 9, 6, 8, 11, 10, 12, 18, 14]$ s'affichera de la façon suivante :

```
                     3
           5                   9
      6          8         11        10
  12   18   14
```

<u>*Indication*</u> : on peut calculer pour chaque niveau le nombre d'espaces à insérer, d'une part en début de ligne, d'autre part entre deux étiquettes, afin de construire la chaîne de caractères correspondant audit niveau. On pourra utiliser les instructions Python suivantes :

- `' '*n` construit une chaîne de `n` espaces ;
- `'{:^3}'.format(e)` crée une chaîne de trois caractères où la valeur de l'entier `e` (supposé compris entre 0 et 999) est centrée.