Arbres binaires complets et quasi-complets

Michel Billaud michel.billaud@laposte.net

27 septembre 2022

Table des matières

1 Motivation

Cette note vient d'un exercice proposé sur Twitter, à propos de la construction d'un arbre binaire de recherche quasi-complet contenant les valeurs d'une liste déjà ordonnée.

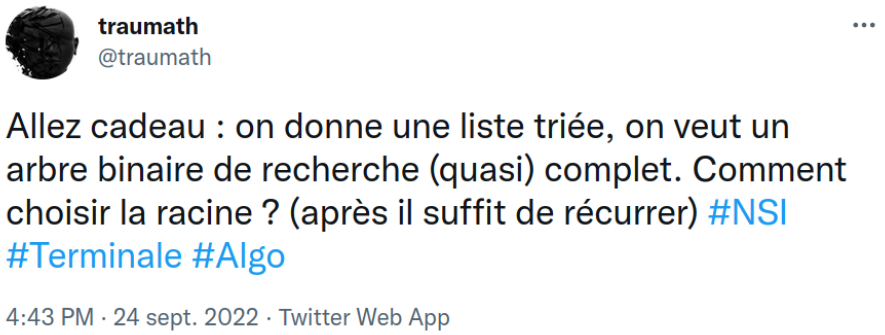


FIG. 1 : https://twitter.com/traumath/status/1573684526559141890

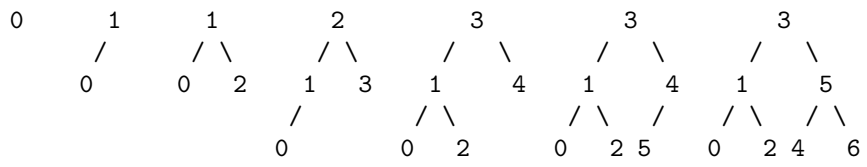
Notions intuitives :

- voir définitions standard pour arbre binaire de recherche
- un arbre binaire est **complet** si toutes ses feuilles sont à la même hauteur. Visuellement, ça fait un triangle.
- dans un arbre **quasi-complet**, la dernière rangée (les feuilles) peut être incomplète à droite à partir d'une certaine position. Ca donne un triangle grignoté à droite.

Une solution possible est de

- calculer, à partir de la taille de la liste, l'indice de valeur qui sera à la racine,
- récursivement, construire le sous-arbre de gauche (resp. droite) à partir de ce qui est dans la liste à gauche (resp. droite) de la racine
- et combiner la racine et les sous-arbres.

La difficulté est de calculer l'indice de la racine à partir de la taille. Ce n'est pas simplement la moitié : pour une liste de taille 5 à 7, cet indice est 3



`arbres binaires quasi-complets pour les listes de 0 à N, pour N=0..6`

2 Définitions

Dans la mesure où s'intéresse seulement au calcul de l'indice de la racine, on se contente de considérer les "formes" d'arbres binaires, ne contenant pas de valeurs.

2.1 Arbres binaires, définitions inductives

Un arbre binaire est

- soit vide,
- soit une combinaison de deux sous-arbres (appelés gauche et droite)

```
Tree :=
   | empty
   | cons(left:Tree, right:Tree)
```

La taille d'un arbre et sa hauteur se définissent aisément par induction

- size(empty) = 0
- size(cons(left, right)) = 1 + size(left) + size(right)

Pour la hauteur, on choisit la convention d'une représentation du nombre de niveaux de sommets

- height(empty) = 0
- height(cons(left, right) = 1 + max(height(left), height(right))

Remarque : il y a un lien entre taille et hauteur

- si height(t) = h, alors size(t) est compris entre h et $2^h - 1$

2.2 Arbres complets

Un arbre t est dit complet de hauteur h (notation : ctree(t,h)) :

- si t = empty alors h = 0
- si t = cons(left, right), alorsc ctree(left, h-1) et ctree(right, h-1)

Il est évident que

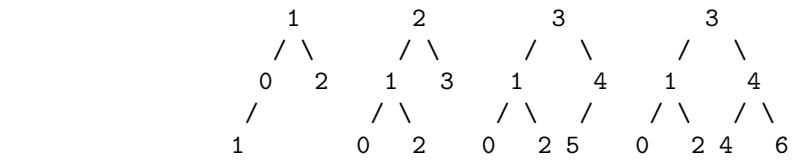
- pour h >= 0 donné, il existe un *unique* arbre t qui satisfasse ctree(t,h)
- que height(t) = h,
- que size(t) = $2^h - 1$

2.3 Arbres quasi-complets

Intuitivement, la dernière rangée d'un arbre quasi-complet de hauteur h

- est *éventuellement* incomplète à droite (les arbres complets sont aussi quasi-complets)n
- contient au moins une feuille (sinon l'arbre serait de hauteur < h)

Si on regarde les exemples d'arbres quasi-complets de hauteur h=3 déjà présentés plus haut



on constate - et c'est une propriété générale -

- qu'il y en a exactement 2^{h-1} ;
- qu'ils vont de l'arbre complet de hauteur h-1 avec une feuille supplémentaire à gauche pour atteindre la hauteur h, jusqu'à l'arbre complet de hauteur h ;
- dont les tailles sont comprises entre de 2^{h-1} à $2^h - 1$

2.4 Caractérisation inductive des arbres quasi-complets

En observant davantage les exemples, on remarque aussi que

- le sous-arbre de gauche d'un arbre t quasi-complet de hauteur h > 1 est un arbre quasi-complet de hauteur h-1 ;
- le sous-arbre de droite est
  - soit complet de hauteur h-2,
  - soit quasi-complet de hauteur h-1, et dans ce cas celui de gauche est complet (de hauteur h-1)

Ceci conduit à une caractérisation inductive : un arbre t est quasi-complet de hauteur h - notation qctree(t, h) :

- quand t = empty : si h = 0

- quand t = cons(left, right) :
  - si qctree(left, h-1) et ctree(right, h-2)
  - ou si ctree(left, h-1) et qctree(right, h-1)

Intuitivement, il y a deux cas parce que les feuilles de la dernière rangée peuvent "commencer à manquer" dans le sous-arbre de gauche (dans ce cas il n'y en n'a pas dans le sous-arbre de droite) ou dans le sous-arbre de droite (et donc le sous-arbre de gauche est complet).

3 Indice de la racine

Rappel du problème : il s'agit, à partir d'une liste (qu'on peut supposer non vide) ordonnée de valeurs, de determiner l'indice de la valeur qui sera la racine de l'arbre binaire de recherche quasi-complet contenant les valeurs de la liste.

Ce problème se ramène à déterminer la taille du sous-arbre de gauche de cet arbre, en fonction de la taille de la liste.

3.1 Taille du sous-arbre de gauche d'un arbre quasi-complet

Nous avons vu que les arbres quasi-complets de hauteur h étaient de taille comprise entre 2^{h-1} et $2^h - 1$.

Inversement, la hauteur d'un arbre quasi-complet de taille n sera donc $h = \lfloor log_2 n \rfloor + 1$

Considérons maintenant la taille du sous-arbre gauche d'un arbre binaire quasi-complet de taille n, avec n évidemment strictement supérieur à 1, faute de quoi il n'a pas de sous-arbre gauche.

Premier cas

La première moitié des arbres quasi-complets de hauteur h, sont tels que $2^{h-1} \leq n < 2^{h-1} + 2^{h-2}$, et sont composés

- à gauche d'un arbre quasi-complet de hauteur h-1
- en partie droite d'un arbre complet de taille h-2

La taille de l'arbre de gauche s'obtient en soustrayant de n la taille du sous-arbre de droite et 1 (la racine), soit $n - (2^{h-2} - 1) - 1 = n - 2^{h-2}$.

Second cas

Dans le cas où $n \geq 2^{h-1} + 2^{h-2}$

le sous-arbre de gauche est complet, et est donc de taille $2^{h-1} - 1$

3.2 En résumé

Soit n la taille de la liste (supposée non vide)

- Si n = 1, la racine est à l'indice 0.
- Sinon, on pose $h = log_2 n + 1$
  - si $n < 2^{h-1} + 2^{h-2}$, la racine est à l'indice $n - 2^{h-2}$.

– sinon elle est à l'indice $2^{h-1} - 1$

3.3 Vérification par programme

Le programme Python ci-dessous produit une table (au format Markdown) des indices des racines en fonction de la taille de la liste

```
import math

def index_racine(n):
    if n == 1:
        return 0
    h = math.floor(math.log(n, 2)) + 1
    # print ( "n = %s, h = %s\n" % (n, h))
    if n < 2**(h-1) + 2**(h-2):
        return n - 2**(h-2)
    else:
        return 2**(h-1) - 1

def show_markdown_table(taille_max):
    print ("|taille", end = "|")
    for i in range(1, taille_max+1):
        print ("%s" % i, end = "|" )
    print ()
    print ("|-", end="|")
    for i in range(1, taille_max+1):
        print ("-", end = "|" )
    print ()
    print("|indice", end="|")
    for i in range(1, taille_max+1):
        print ("%s" % index_racine(i), end="|" )
    print ()

show_markdown_table(16)
```

Résultat affiché

taille	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
indice	0	1	1	2	3	3	3	4	5	6	7	7	7	7	7	8

4 Programmation

Le code suivant

```
print(toString(mkTree([0, 11, 22, 33, 44, 55])))
```

affiche une représentation textuelle de l'arbre construit à partir de la liste à 6 éléments `[0, 11, 22, 33, 44, 55]`

Résultat

```
node(node(node(., 0, .), 11, node(., 22, .)), 33, node(node(., 44, .), 55, .))
```

Structure de données “arbre”

Cette structure récursive est déclarée comme une classe “Plain Old Data”, sans méthodes, pour ne pas effaroucher les débutants

```
class Tree:
    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right
```

Le calcul de la représentation d'un arbre sous forme de chaîne est récursif :

```
def toString(t):
    if t == None:
        return "."
    else:
        return "node(%s, %s, %s)" % (toString(t.left),
                                     t.value,
                                     toString(t.right))
```

Construction d'un arbre

Pour éviter d'extraire des sous-listes, la fonction `mkTree` est un “lanceur” pour une autre (récursive) à qui on indique les indices de début et fin (non comprise) de la portion de liste à considérer

```
def mkTree(l):
    return mkTree_r(l, 0, len(l))
```

Enfin, la fonction récursive fait le job, en partageant au besoin la sous-liste à traiter

```
# construit l'ABR quasi-complet de la sous-liste
# de l, d'indices start (inclus) à end (exclus)

def mkTree_r(l, start, end):
    if start == end:
        return None
    root = start + index_racine(end - start)
    return Tree(l[root],
                mkTree_r(l, start, root),
                mkTree_r(l, root+1, end))
```