Algo

- Numero: 4
- Prof: Alexandre Duret-Lutz
- Date: 24 Octobre 2017

Objectif

On va voir un theoreme qui fait le cafe. Notament pour avoir la complexite d'une fonction recursif.

Theoreme general

Ce theoreme permet de resoudre les équations de complexites de la forme:

Tips: Il sera donne en partiel. Il faut quand meme savoir l'utiliser.

$T(n) = aT(\frac{n}{b} \pm \Theta(1)) + f(n) \implies \Theta(n \log(n))$
ou $a \geq 1, b \geq 1$, et $f(n)$ fonction positive

Il y a 4 cas:

1. si $f(n) = \mathcal{O}(n^{log_b(a)-\varepsilon})$ pour un $\varepsilon \geq 0$ alors $T(n) = \Theta(n^{log_b a})$
2. si $f(n) = \Theta(n^{log_b(a)})$ pour un $\varepsilon \geq 0$ alors $T(n) = \Theta(n^{log_b a} \times \log n) = \Theta(f(n) \log n)$
3. si $f(n) = \Omega(n^{log_b(a)+\varepsilon})$ pour un $\varepsilon \geq 0$ et de plus il existe $c \in [0, 1[$ tq $af(n/b) \leq cf(n)$ alors $T(n) = \Theta(f(n))$
4. sinon le theoreme ne s'applique pas.

Exemples

1^er^ exemple

$T(n) = 3T(\frac{n}{4}) + \sqrt{n}$
Donc $a = 3$ et $b = 4$. De plus $\log_4(3) \simeq 0.792$

$\sqrt{n} = n^{0.5}$ on est donc dans le cas 1.

$\mathcal{O}(n^{\log_4 3-\varepsilon})$ avec $\varepsilon = 0.1$ par exemple.

2^eme^ exemple

$T(n) = 3T(\frac{n}{4}) + n$
Donc $a = 3$ et $b = 4$.

On est dans le cas 3.

$\Omega(n^{\log_4 3+\varepsilon})$ avec $\varepsilon = 1$ et de plus il existe $c = 3/16 \leq 1$ tel que $af(n/b) \leq cf(n)$
$3(\frac{n}{4})^2 \leq cn^2$
$\frac{3}{16} \leq c$

Donc $T(n) = \Theta(n^2)$

La substitution

$T(n) = 2T(\frac{n}{2}) + cn \log_2 n$

1^er^ substitution:

$T(n) = 2T(2T(\frac{n}{4}) + c\frac{n}{2} \log \frac{n}{2}) + cn \log_2 n$
$= 4T(\frac{n}{4}) + cn(log_2(\frac{n}{2}) + \log_2 n)$

2^eme^ substitution:

$T(n) = 4T(2T(\frac{n}{8}) + c\frac{n}{4} \log_2 \frac{n}{4} + cn(log_2(\frac{n}{2}) + \log_2 n)$
$= 8T(\frac{n}{8}) + cn(log_2(\frac{n}{4}) + log_2(\frac{n}{2}) + \log_2 n)$

On voit un pattern apparaitre.

$T(n) = 2T(\frac{n}{2} + \Theta(n \log n)$
$a = 2b = 2f(n) = \Theta(n \log n)$

On est dans le 4^eme^ cas du theoreme general.

$$T(n) = nT(n) + cn\left(\sum_{k=0}^{\log_2(n)-1} \log_2 n - \sum_{k=0}^{log_2(n)-1} k\right)$$

$T(n) = \Theta(n) + \Theta(n \log(n) \log(n))$
$= \Theta(n \log(n) \log(n))$

Arbre parfait

> Arbre binaire (chaque sommet a au plus deux fils), avece tous les niveaux pleins, sauf eventuellement le derniers ou les feuilles sont toutes a gauche.

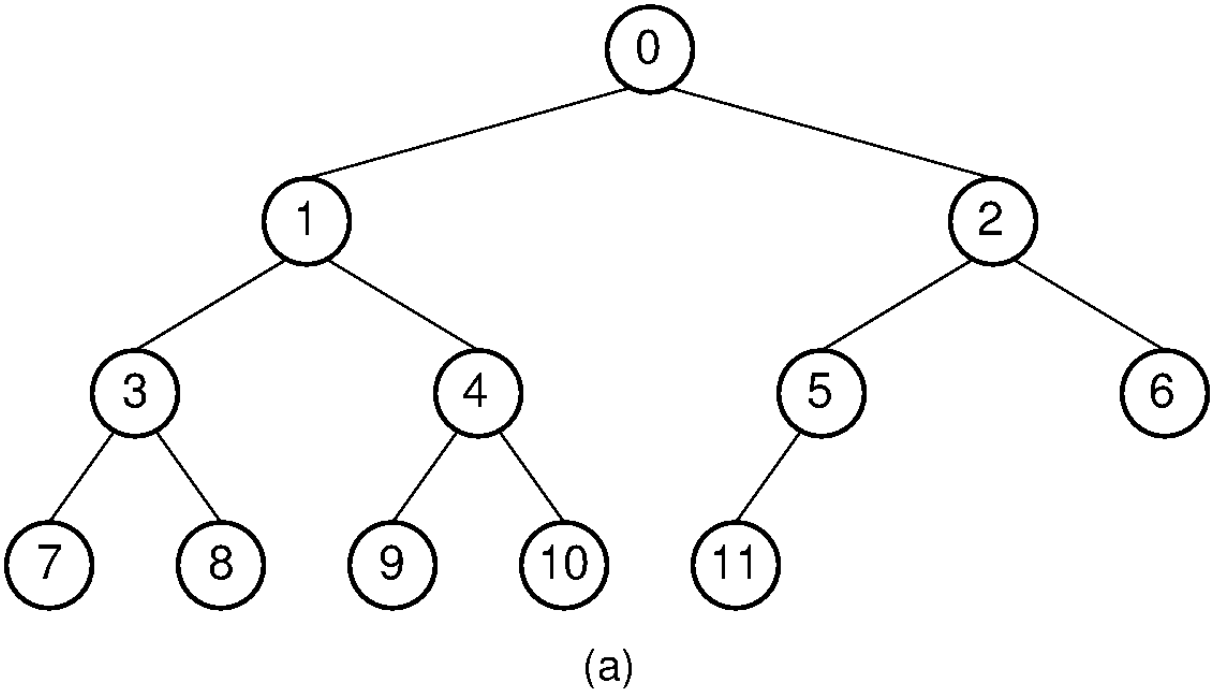

Figure 1: Arbre parfait

Chaque numero de l'arbre va correspondre sa place dans une liste. Cela est une representation simple:

0	1	2	3	4	5	6	7	8	9	10	11

- `leftchild(i)= 2 * i + 1` (s'il existe)
- `rightchild(i)= 2 * i + 2` (s'il existe)

Interet:

> Un arbre parfait se represente avec un tableau en lisant les valeurs ligne par ligne.

Tas

Structures

> Arbre parfait avec la contrainte de tas (chaque noeud est plus grand que ces fils). Pour un tas “max”.

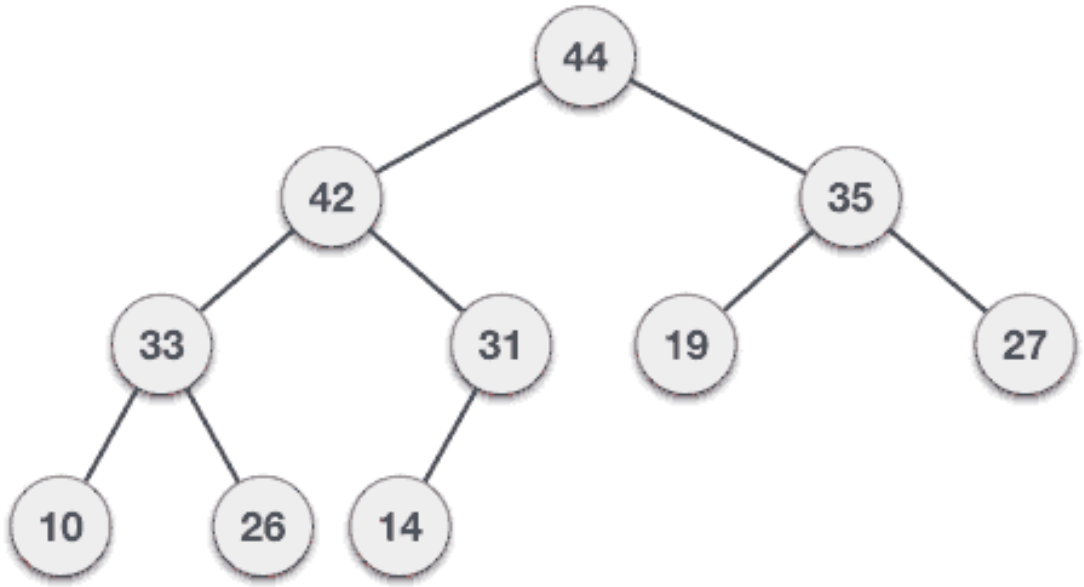

Figure 2: Tas max

Pour voir l’animation des algos d’ajouts et suppression d’elements: Clique ICI

Algos

```
def Heapify(A, m, i): //m la taille du tas
  // On suppose fg(i) et fd(i) sont des tas
  // mais i pas forcément.
  l = leftChild(i)
  r = rightChild(i)
  if l < m and A[l] > A[i]:
    largest = l
  else:
    largest = i
  if r < m and A[r] > A[largest]:
    larget = r
  if largest != i:
    swap(A, largest, i)
    Heapify(A, m, largest)
```

```
def BuildHeap(A, m):
  for i in range(n // 2 - 1, -1, -1):
    Heapify(A, m, i)
```

```
def HeapSort(A, m):
  BuildHeap(A, m)
  for i in range(m - 1, 0, -1): // Le cas ou i = 0 est inutile a
     traiter
    swap(A, 0, i)
    Heapify(A, i, 0)
```

Complexite

$T_H(n) \leq T_H(\frac{2}{3}n) + \Theta(1)$

On est dans le cas 2 du theoreme general. Donc:

$T(n) \leq \Theta(\log n)$