

#1

Question 1 Montrons par récurrence sur  $n > 0$  que un arbre binomial d'ordre  $n$  a une hauteur égale à  $n$ .

(base) Soit  $n = 1$ . Alors  $a$  admet un unique fils d'ordre 0.  $a$  est donc de la forme  en reprenant la notation de l'énoncé. La hauteur de  $a$  est égale à 1.

(référence) Supposons la propriété vraie pour tout  $k$  :  $0 \leq k \leq n$ .

Soit  $a$  un arbre binomial d'ordre  $n$ .

Alors chacun de ses fils est un autre binomial d'ordre  $k < n$ . Par hypothèse de récurrence chaque fils a une hauteur égale à son ordre. Le fils possédant la plus grande hauteur est celui d'ordre  $n-1$  qui admet une hauteur de  $n-1$ .

L'arbre a donc la hauteur

$$h = 1 + n-1 = n$$

#2

Question 2 : Soit  $a$  un arbre binomial d'ordre  $m$ . Montrons par récurrence sur  $m \in \mathbb{N}^*$  que le cardinal de  $a$  vaut  $2^m$ .

(base) Soit  $a$  l'arbre d'ordre 1.

Alors  $a$  est de la forme !  
et admet donc deux éléments ! : la racine et son unique fil d'ordre 0.  
 $\text{card } a = 2^1 = 2$

(induction) Soit  $k \in \mathbb{N}^*$  tel que  $0 < k < m$   
et supposons que la propriété soit vraie tout  
arbre binomial d'ordre  $k$ .

Alors  $a$  admet  $n$  fils d'ordre  $m-1, m-2, \dots, 0$   
Par hypothèse de récurrence, chaque fils d'ordre  $k$  admet  $2^k$  éléments.

Ainsi  $a$  admet pour cardinal la somme des  
cardinaux de chacun de ses fils  
à laquelle il faut ajouter 1 pour la  
racine de  $a$ . On a donc :

$$\text{Card } a = 1 + \sum_{k=0}^{m-1} 2^k = 1 + \frac{2^m - 1}{2 - 1} = 2^m$$

#3

Question 3 Montrons par récurrence sur  $n \in \mathbb{N}^*$  qu'un arbre binomial d'ordre  $n$  admet  $2^{n-1}$  branches.

(base) Soit  $a$  un arbre binomial d'ordre 1.

Alors  $a$  admet un unique fils d'ordre 0. Ce fils admet 0 branches donc  $a$  n'a que une seule branche.

$$N\text{-branche } a = 1 = 2^{1-1} = 2^0$$

(récurrence) Soit  $a$  un arbre binomial d'ordre  $n > 1$ .

(hypothèse) Supposons la propriété vraie pour tout arbre binomial d'ordre  $0 < k < n$ .

~~Not~~  $\underline{\text{Soit}}$  Alors  $a$  peut se décomposer en deux sous arbres d'ordre  $n-1$ .

$a$  admet autant de branches que la somme des branches de ces deux sous arbres. Or par hypothèse de récurrence chacun de ces sous arbre admet  $2^{n-1-1}$  branches. Donc  $a$  admet  $2^{n-1-1} + 2^{n-1-1} = 2 + 2^{n-2} = 2^{n-1}$  branches.

Question 4 Montons par récurrence sur  $m$  que le nombre de nœuds de  $a$  à la profondeur  $k$  vaut  $\binom{m}{k}$

(base) Soit  $m=1$ . Alors l'arbre  $a$  est :

Il y a 1 nœud à la racine et donc le nombre de nœud à la profondeur 0 vaut bien  $\binom{1}{0} = 1$  et de même, il y a un nœud à la profondeur 1 et on a bien  $\binom{1}{1} = 1$ .

(référence) Soit  $a$  un arbre d'ordre  $m > 0$ . Supposons la propriété vraie pour tout  $1 \leq m' \leq m$ . Soit  $k$  une profondeur  $0 \leq k \leq m$ .

Les nœuds à la profondeur  $k$  sont les descendants de deux nœuds particuliers. En effet, puisque tout arbre d'ordre  $m > 0$  se décompose en deux sous arbres d'ordre  $m-1$ , les nœuds de profondeurs  $k$  sont (1) soit les descendants du fils droit de la racine premier fils de la racine, (2) soit de la racine privée de son premier fil.

En considérant le premier fils comme la racine d'ordre  $m-1$ , il y a exactement autant de nœuds descendant du fils.

Notons (1)  $a_0$  le sous arbre de  $a$  dont la racine est son premier fil

(2)  $a_1$  le sous arbre de  $a$  constitué de la racine sans son premier fil.

Il y a autant de nœuds à la profondeur  $k$  de  $a$  que la sommes des

nœuds à la profondeur  $k-1$  de  $a_0$   
 et des nœuds à la profondeur  $k$  de  $a_1$ .

En effet les nœuds à la profondeur  $k$  de  $a$   
 sont (1) soit descendants de  $a_0$ , auquel cas,  
 puisque  $a_0$  est le premier fils de  $a$ ,  
 ils sont situés à la profondeur  $k-1$  dans  
 $a_0$ .

(2) soit descendants de  $a_1$ , auquel cas  
 ils sont situés à la profondeur  $k$  dans  $a_1$ ,  
 puisque la racine de  $a_1$  est la même que  
 la racine de  $a$ .

Or  $a_0$  et  $a_1$  sont de arbres d'ordre  $m-1$   
 donc par hypothèse de récurrence, ils  
 admettent :

\* pour  $a_0$  :  $\binom{m-1}{k-1}$  nœuds à la profondeur  $k-1$

\* pour  $a_1$  :  $\binom{m-1}{k}$  nœuds à la profondeur  $k$

Ainsi  $a$  admet  $\binom{m-1}{k-1} + \binom{m-1}{k} = \binom{m}{k}$   
 nœuds à la profondeur  $k$ .

Question 5 Montrez par récurrence sur  $m > 1$  que la profondeur moyenne de  $a$  vaut  $\frac{m-1}{2}$

(base) Soit  $a$  un arbre d'ordre 1 alors  $a := \begin{array}{c} 1 \\ | \\ 0 \end{array}$ . La profondeur moyenne de  $a$  vaut :

$$\bar{k} = \frac{0 \times 1 + 1 \times 1}{2} = \frac{1}{2}$$

(référence) Soit  $a$  un arbre d'ordre  $m > 0$ . Supposons la propriété vraie pour tout  $0 \leq m' < m$ .

Décomposez  $a$  en deux sous arbres :

$a_0$  son premier fils d'ordre  $m-1$  et  $a_1$  sa seconde nièce du premier fils, arbre d'ordre  $m-1$  aussi.

Notons  $\bar{k}_0$  la profondeur moyenne ~~des nœuds~~ de  $a_0$  et  $\bar{k}_1$  celle de  $a_1$ .

En remarquant que la profondeur d'un nœud de  $a$  varie de 1 si le nœud est un descendant de  $a_0$  et ne varie pas si il est un descendant de  $a_1$ , il vient que,

$$\bar{k} = \frac{2^{m-1} \times (\bar{k}_0 + 1) + 2^{m-1} (\bar{k}_1)}{2^m}$$

où  $\bar{k}$  est la profondeur moyenne de  $a$ .

En effet,  $a$  possède  $2^m$  nœuds et  $a_0$  et  $a_1$  en possèdent  $2^{m-1}$ .

Grâce à  $a_0$  et  $a_1$  sont des arbres d'ordre  $n-1$   
donc par hypothèse de récurrence,

$$\overline{k}_0 = \frac{n-1}{2} \quad \text{et} \quad \overline{k}_1 = \frac{n-1}{2}$$

Donc

$$\overline{k} = \frac{2^{n-1} \left( \frac{n-1}{2} + 1 \right) + 2^{n-1} \left( \frac{n-1}{2} \right)}{2^n}$$

$$\overline{k} = \frac{2^{n-1} \left( \frac{n-1}{2} + \frac{n-1}{2} \right) + 2^{n-1}}{2^n}$$

$$\overline{k} = \frac{n-1 + 1}{2}$$

$$\overline{k} = \frac{n}{2}$$

Question 6 Montrons par récurrence sur  $n > 0$  que  
le nombre d'octets nécessaires à représenter un arbre d'ordre  $n$  vaut  $2^n M_0 + (2^n - 1) M_1$ .

(base) Soit  $A$  l'arbre d'ordre 1 de la figure.  
Alors  $A$  est un pointeur vers un objet de type ~~arbre-binaire~~. Cet objet  $Nod$  qui  
occupe  $M_0$  octets. Ce nœud racine a  
un seul fils donc son champ fils pointe  
vers un objet de type  $Cons$  qui aura pour  
champ suivant un pointeur vers  $NULL$ .  
Cet unique objet  $Cons$  occupe  $M_1$  octets.

Pour finir, le champ élément de est fixé lors  
que sur un deuxième arbre binomial qui pointe  
vers un deuxième objet de type Es Nœud.

Enfin ce deuxième Nœud n'a pas de fils  
et son champ fils pointe donc sur NULL.

En conclusion, on a bien 2 objets Nœuds  
qui occupent chacun  $M_0$  octets et 1 objet  
Cois qui occupe  $M_1$  octets.

L'espace total de mémoire occupé est bien

$$2^k M_0 + M_1 = 2^k M_0 + (2-1)M_1.$$

(réurrence) Soit à un arbre d'ordre  $n$ .

Supposons la propriété vraie pour tout  
arbre d'ordre  $k$  inférieur à  $n$ .  $0 \leq k \leq n$ .

a est une racine se décompose en un très arbre  $a_0$   
(fils d'ordre 1) d'ordre  $n-1$  relié à un  
autre arbre  $a_1$  (racine privée de son premier fils)  
d'ordre  $n-1$ .

L'occupation mémoire totale de a vaut donc  
la somme de celle de  $a_0$  avec celle de  $a_1$   
à quoi il faut ajouter l'espace mémoire  
nécessaire à relier  $a_0$  et  $a_1$ .

$$\text{Espace Mémoire}_a = \text{Espace Mémoire}_{a_0} + \text{Espace Mémoire}_{a_1}$$

$$+ \text{Espace Mémoire Liaison}$$

62 par hypothèse de récurrence on a

$$\text{Espace Mem}_{a_0} = \bar{\text{Espace Mem}}_{a_1} = 2^{m-1} M_0 + (2^{m-1} - 1) M_1$$

Déterminons  $\bar{\text{Espace Mem}}_{\text{liaison}}$ .

Pour cela, on remarque que la liste chaîne de la racine de  $a$  possède un maillon de tête supplémentaire ~~différent~~ que celle de la racine de  $a_1$ .

Ce maillon supplémentaire pointe vers la racine de  $a_0$  et admet pour chaîne suivant un pointeur vers le ~~racines~~ premier fils de  $a_1$ .

Le maillon supplémentaire est le seul changement nécessaire pour passer de  $a_0$  et  $a_1$  à  $a$ .

Ainsi :  $\bar{\text{Espace Mem}}_{\text{liaison}} = 1 \times M_1$ .

$$\begin{aligned}\text{DHC : } \bar{\text{Espace Mem}}_a &= 2 \times \left( 2^{m-1} M_0 + (2^{m-1} - 1) M_1 \right) + M_1 \\ &= 2^m M_0 + 2^m M_1 - 2 M_1 + M_1 \\ &= 2^m M_0 + M_1 (2^m - 1)\end{aligned}$$

Ainsi, en factorisant, on a obtenu

$$\text{Espace Mem}_a = 2^m (M_0 + M_1) - M_1$$

$$\begin{aligned}\text{d'où} \quad \left. \begin{array}{l} A = -M_1 \\ B = M_0 + M_1 \end{array} \right.\end{aligned}$$

## Question 7

```
bool est-binomial (arbre-binomial a, int n) {  
    // cas de base  
    if ((n == 0) || (n == 1)) { return a->fils == NULL; }  
    if (a->fils == NULL) return false;  
    return a->fils->suivant == NULL;  
}  
  
// cas récursif  
int ordre = n - 1;  
arbre-binomial fils-courant = a->fils;  
while (! (fils-courant == NULL || ordre < 0)) {  
    if (! est-binomial (fils-courant, ordre)) {  
        return false; }  
    fils-courant = fils-courant->fils;  
    ordre -= 1;  
}  
return (fils-courant == NULL && ordre == -1);  
}
```

### Question 8

Pour déterminer la complexité temporelle, on s'intéresse au pire des cas qui survient lorsque l'arbre est binomial. M'intervient, c'est-à-dire lorsque l'arbre est un arbre binomial.

On remarque que la fonction effectue un appel récursif par fils du nœud courant.

Ainsi, pour un arbre binomial, un appel sur le nœud racine engendrera  $2^m - 1$  appels récursifs puisqu'il y a  $2^m - 1$  fils dans l'arbre.

En comptant l'appel initial, la fonction sera exécutée  $2^m$  fois. La complexité temporelle est

puisque les appels récursifs, la fonction ne possède pas d'autre boucle et les instructions exécutées se font en temps constant : tests binaire, renvoi, affectation, construction.

Ainsi la fonction admet une complexité temporelle en  $\Theta(2^n)$

Question 9 Afin d'effectuer un parcours en largeur sans récursivité, il faut utiliser une structure de données de type FIFO.

Or la liste simplement chaînée, en ne donnant accès qu'à la tête et au maillon suivant, est une structure de type LIFO.

Il est donc nécessaire de utiliser une structure FIFO comme la file.

Si on s'était contenté de la structure existante de liste chaînée, on aurait alors mis en place un parcours en profondeur de l'arbre.

Question 10 Montrons cette propriété par induction sur la file de nœuds.

(base) initialement, la file est vide. La propriété de cohérence et de différence est donc trivialement vraie.

(induction) supposons qu'en début de boucle la file est constituée de nœuds situés à profondeur  $i_{\text{max}}$  et que la différence entre la profondeur de tête et de queue vaut 1 au plus 1.

(1) supposons que cette différence vaut 0. La file est constituée uniquement de frères de même profondeur. Le traitement se fait sur le premier frère en tête de file qui a été défilé. La seconde boucle while ajoute tous les

enfant du premier père au queue de file  
Chacun à une profondeur spécifique à 1  
par rapport à elle initialement présente dans  
la file -

Les profondeurs sont donc classées de façon  
croissantes et leur différence est  
évidemment égale à 1.

(2) Supposons maintenant qu'en sortant dans la  
boucle while les profondeurs sont  
maisées avec une différence de 1.

Alors le nœud en tête de file est défilé.  
Il admet une profondeur minimale.  
La seconde boucle while enfile tous ses  
fils qui ont une profondeur augmentée  
de 1 par rapport à leur père.

Ainsi les profondeurs des nœuds ajoutés en  
queue de file sont égales à la  
valeur maximale qui était celle de la  
du dernier nœud de la file.

L'ordre des profondeurs n'est pas changé et  
donc il est toujours croissant.

La différence est toujours au plus, égale  
à 1 entre la nouvelle  
profondeur de tête et celle de la queue.

Question 11 Montrez par récurrence sur  $m \geq 0$  la taille maximale de la file lors d'un parcours en largeur vaut  $2Bm$ .

(base) Soit à l'arbre d'ordre 1. alors  $\frac{a}{1} = 1$   
La file est initialement vide puis contient la racine qui est défilée aussitôt.  
La deuxième boucle while ajoute le fils unique dans cette file vide.

U La 1<sup>re</sup> boucle while principale s'exécute sur la file contenant uniquement la fils et le défile. N'ayant aucun fils, la boucle s'arrête ce qui termine le parcours.

La file a contenu au maximum 1 élément et on a trouvée  $\binom{1}{L_{2-1}} = \binom{1}{0} = 1$ .

(réurrence) Soit à l'arbre d'ordre  $n$ . La propriété est vraie pour tout parcours d'arbre d'ordre  $m \leq n$ .

On se décompose en  $a_0$  : fils premier, racine du sous arbre d'ordre  $n-1$  et en  $a_1$ , sous arbre constitué de la racine privée du premier fils.

Les fils du parcours en largeur de  $a$ , la file ne peut pas avoir plus de nœud que la file qui a parcouru  $a_0$  ajouté.

La somme du nombre de nœud des fils parcourant en largeur les sous arbres  $a_0$  et  $a_1$ .

On a donc

$$\text{TailleMax}_a \leq \text{TailleMax}_{a_0} + \text{TailleMax}_{a_1}$$

or Par hypothèse de récurrence sur  $a$

$$\text{TailleMax}_{a_0} = 2B_{m-1} = 2\left(\left\lfloor \frac{m-1}{2} \right\rfloor\right)$$

$$\text{TailleMax}_{a_1} = 2B_{m-1} = 2\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right)$$

De plus  $a_0$  est un arbre donc la racine est située à la profondeur 1 de  $a$  donc en prenant comme référence  $k=0$  la racine de  $a$ , on a :

$$\text{TailleMax}_{a_0} = 2\left(\left\lfloor \frac{m-1}{2} \right\rfloor + 1\right)$$

Ce qui donne en prenant  $k=0$  pour la racine de  $a$  :

$$\text{Taille Max}_a \leq 2 \binom{m-1}{\lfloor \frac{m-1}{2} \rfloor + 1} + 2 \binom{m-1}{\lfloor \frac{m-1}{2} \rfloor}$$

donc par propriété de l'opérateur

$$\binom{m}{k} = \binom{m-1}{k-1} + \binom{m-1}{k}$$

on obtient :

$$\begin{aligned} \text{Taille Max}_a &\leq 2 \left( \binom{m-1}{\lfloor \frac{m-1}{2} \rfloor + 1} + \binom{m-1}{\lfloor \frac{m-1}{2} \rfloor} \right) \\ &\leq 2 \binom{m}{\lfloor \frac{m-1}{2} \rfloor + 1} \end{aligned}$$

or, puisque  $\max_{0 \leq k \leq m} \binom{m}{k} = B_m$

on obtient à bien  $\binom{m}{\lfloor \frac{m-1}{2} \rfloor + 1} \leq B_m$

Ainsi

$$\text{Taille Max}_a \leq 2 B_m.$$

Question 12 Pour déterminer l'ordre de l'arbre binomial passé en paramètre, je vais renouveler son nombre de fils.

```
int ordre ( arbre_binomial a ) {  
    int m = 0 ; // ordre courant  
    arbre_binomial tête = a->fils ;  
    while ( tête != NULL ) {  
        m += 1 ;  
        tête = tête->fils ;  
    }  
    return m ;  
}
```

Question 13

```
arbre_binomial singleton ( int val ) {  
    arbre_binomial a = malloc ( sizeof ( Node ) ) ;  
    a->fils = malloc ( sizeof ( Node ) ) ;  
    a->fils->val = val ;  
    a->fils->fils = NULL ;  
    a->val = val ;  
    return a ; }
```

### Question 14

arbre\_binaire coller (arbre\_binomial a<sub>1</sub>, arbre\_binomial a<sub>2</sub>) {

arbre\_binomial racine = a<sub>1</sub> ;

liste\_chainee tete = malloc (n) ↗ of (cons) ;

tete → element = a<sub>2</sub> ;

tete → suivant = racine → fils ;

racine → fils = tete ;

return racine ;

}

### Question 15

Par récurrence une fonction récursive :

bool est\_en\_tas (arbre\_binomial a) {

int n = ordre a ;

// cas de base

if (n == 0) { return true ; }

if (n == 1) { return a . valeur ≤  
a → fils → element . valeur ; }

// récursion

arbre binomial  $a0 = a \rightarrow \text{fils} \rightarrow \text{élément}$  ;

arbre binomial  $a1 = \text{singleton}(a.\text{valeur})$  ;

$a1 \rightarrow \text{fils} = a \rightarrow \text{fils} \rightarrow \text{suivant}$  ;

// cas  $a0 > a1$  :

```
if ( $a0.\text{valeur} > a1.\text{valeur}$ ) {  
    return false; }
```

// récursion

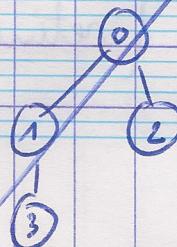
return ( $\text{est-en-tas}(a0) \wedge \text{est-en-tas}(a1)$ );

Question 16 Montrons par récurrence sur  $n \geq 1$  que pour un arbre binomial d'ordre  $n$  la fonction  $\text{est-en-tas}$  effectue au plus  $2^{n-1} + 1$  comparaisons.

(base)  $m=1$ . Soit  $a$  l'arbre d'ordre 1. Alors c'est un cas de base de la fonction et une seule comparaison est effectuée.

$m=2$  - Soit  $a$  l'arbre d'ordre 2

$a :=$



Question 16 Soit à un abse d'ordre  $n$ . et  $N(n)$  le nombre maximal de comparaisons faites par la fonction est-entre.

Puisque la fonction décompose à un deux sous autres d'ordre  $n-1$  pour ensuite comparer les deux racines on a la relation de récurrence suivante :

$$\begin{cases} N(1) = 1 \\ N(n) = 2 N(n-1) + 1 \quad n > 1 \end{cases}$$

En sommant, on obtient

$$\begin{aligned} \sum_{k=2}^n 2^{n-k} N(k) &= \sum_{k=2}^n 2^{n-k} (2N(k-1) + 1) \\ &= \sum_{k=2}^n 2^{n-k} \cdot 2N(k-1) + \sum_{k=2}^n 2^{n-k} \end{aligned}$$

En posant  $j = k-1$  on a :

$$\sum_{k=2}^n 2^{n-k} N(k) = \sum_{j=1}^{n-1} 2^{n-j} N(j) + \sum_{k=2}^n 2^{n-k}$$

Ainsi

$$2^{n-n} N(n) = 2^{n-1} N(1) + \sum_{k=2}^n 2^{n-k}$$

$$N(n) = 2^{n-1} + \sum_{k=0}^{n-2} 2^k$$

$$\text{Enfin : } N(n) = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$$

On a donc, pour  $n > 0$ ,

$$N(n) = 2^n - 1$$

### Question 17

arbre binomial collerontas (arbre binomial  $a_1$ ,  
arbre binomial  $a_2$ ) {

if ( $a_1 \cdot \text{valen} \leq a_2 \cdot \text{valen}$ ) {

return coller ( $a_1, a_2$ ); }

else {

return coller ( $a_2, a_1$ ); }

}

### Question 18 Étudions tous les cas possible.

Cas sans retenue

$$r_i = 0$$

(1)  $a_i = 0$  et  $b_i = 0$

$$\text{Alors } R_{i+1} = 0 \cdot (\dots) + 0 \cdot 0 = 0$$

$$\text{et } S_i = 0 \oplus 0 \oplus 0 = 0$$

(2)  $a_i = 1$  et  $b_i = 0$

$$r_{i+1} = 0 + 0 \cdot 1 = 0$$

$$s_i = 0 \oplus 1 \oplus 0 = 1$$

(3)  $a_i = 0$  et  $b_i = 1$

cas symétrique de (2)

(4)  $a_i = 1$  et  $b_i = 1$

$$r_{i+1} = 0 + 1 \cdot 1 = 1$$

$$s_i = 0 \oplus 1 \oplus 1 = 0$$

Dans les cas (1), (2), (3) et (4)  
on a bien conservé la paire avec

$$a_i 2^i + b_i 2^i + r_i 2^i = s_i 2^i + r_{i+1} 2^{i+1}$$

Cas avec retenue :

$$r_i = 1$$

(5)  $a_i = 0$  et  $b_i = 0$

$$r_{i+1} = 1 (0+0) + 0 \cdot 0 = 0$$

$$s_i = 1 \oplus 0 \oplus 0 = 1$$

(6)  $a_i = 1$  et  $b_i = 0$

$$r_{i+1} = \overbrace{1 (1+0)}^1 + \dots = 1$$

$$s_i = 1 \oplus 1 \oplus 0 = 0$$

(7) cas où  $a_i = 0$  et  $b_i = 1$

cas symétrique de (6)

(8)  $a_i = 1$  et  $b_i = 1$

$$r_{i+1} = \underbrace{1 \cdot (1+1)}_{1} + \dots = 1$$

$$r_i = 1 \oplus 1 \oplus 1 = 1$$

Dans les cas (5)(6)(7)(8) on a encore la conservation de la forme.

Question 19 Avec les variables :

tas-binomial  $a, b, r_2, \Delta$  ;

la propriété (2) devient :

$$\begin{aligned} & \text{occ}\left(\text{ou}\left(\text{ou}\left(a.\text{tableau}[i], b.\text{tableau}[i]\right), r_2.\text{tableau}[i]\right)\right) \\ &= \text{occ}\left(\text{ou}\left(r_2.\text{tableau}[i], r_2.\text{tableau}[i+1]\right)\right); \end{aligned}$$

Question 20

Puisque la fonction  $\text{ou}$  n'est pas commutative, je fais attention à utiliser  $\text{ou}(a, b)$  plutôt que  $\text{ou}(b, a)$ . ainsi que  $a_i \oplus b_i \oplus r_i$  plutôt que  $r_i \oplus a_i \oplus b_i$ .

En rappelant que tous les espaces mémoires sont réservés, on a :

$$r_2.\text{tableau}[0] = \text{NULL};$$

r. tableau [ $i+1$ ] =  
ou ( et ( r.tableau [ $i$ ] , ou ( a.tableau [ $i$ ],  
b.tableau [ $i$ ] ) ) ,  
et ( a.tableau [ $i$ ] , s.tableau [ $i$ ] ) ) ;

s.tableau [ $i$ ] =  
ou\_excluif ( ou\_excluif ( a.tableau [ $i$ ] ,  
b.tableau [ $i$ ] ) ,  
r.tableau [ $i$ ] ) ;

### Question 21 :

```
tas-binomial initialise_tas (int max) {  
    tas-binomial * t = malloc ( sizeof (tas-binomial) ) ;  
    t.taille = max ;  
    t.tableau = malloc ( max * sizeof (arbre-binomial) ) ;  
    for (int i=0 ; i < max ; i++) {  
        t.tableau [i] = NULL ; }  
    return t ;  
}
```

### Question 22

```
double cardinal_tas (tas_binomial t) {  
    double card = 0 ;  
  
    for (int i=0 ; i<t.taille ; i++) {  
        if (t.tableau [i] != NULL) {  
            card += (1<<i) ;  
        }  
    }  
  
    return card ;  
}
```

### Question 23

```
int position_minimum_tas (tas_binomial t) {  
    int imin = -1 ;  
    for (int i=0 ; i<t.taille ; i++) {  
        if (t.tableau [i] != NULL) {  
            // initialise  
            if (imin == -1) {imin = i;}  
            else {  
                if (t.tableau [i] < t.tableau [imin]) {  
                    imin = i; }  
            }  
        }  
    }  
    return imin ;  
}
```

### question 24

Dans le pire des cas, lorsque le tableau a toutes ces cases contenant un arbre binomial, il y a autant de comparaisons que de cases, soit  $t \cdot \text{taille}$  comparaisons.

Or, en notant  $\text{card}(t)$  le cardinal du tas, on a :  $\text{card}(t) \leq 2^{t \cdot \text{taille}} - 1$

$$\text{donc } t \cdot \text{taille} \leq \log_2 (\text{card}(t) + 1)$$

Ainsi, le nombre de comparaisons est borné par  $\log_2 (\text{card}(t) + 1)$ .

### Question 25



## question 24

arbre-binomial insertion\_tas (int valeur, tas-binomial tas){

    arbre-binomial r0 = singleton (valeur) ;

    arbre-binomial r1 = singleton (valeur) ;

    int i = -1 ;

    while (! (r1 == NULL || i >= tas.taille)){

        i += 1 ;

        r0 = r1 ;

        r1 = et (r0, tas.tableau [i]) ;

        // mixe à jour du tas  
        tas.tableau [i] = ou-exclusif  
            (r0, tas.tableau [i]) ;

}

    if (r1 != NULL) {return r1 ;}

    else {return NULL ;}

}

### Question 25

Les comparaisons ont lieu dans le prédicat de l'instruction `while`, la fonction et et la fonction `on-exclusif`.

Il y a donc un nombre de comparaisons proportionnel au nombre de boucles.

Or dans le pire des cas, il y a  $(\text{tas.taille} + 1)$  boucles lorsqu'une retenue engendre un déplacement ~~et~~ débordement.

Dans le pire des cas, le tas est complet et il admet  $2^{\text{t.taille}} - 1$  élément

$$\text{donc } \log_2 (\text{card}(\text{tas}) + 1) = \text{t.taille}$$

Ainsi le nombre de comparaisons est proportionnel à  $\log_2 (\text{card}(\text{tas}) + 1) + 1$

On a bien montré que le nombre de comparaison est logarithmique en fonction du cardinal du tas.

Question 26 Si l'inversion vérifie  $P_k$  alors  $R_{k+1}$  et la première retenue non nulle.

Il y a donc  $k+1$  comparaisons : une par retenue de  $r_0$  à  $r_k$ ,

Question 27 Soit  $0 \leq k < N$ .

Lorsque l'on fait des insertions successive, la propriété  $P_k$  n'est vérifiée que une seule fois lorsque les  $k+1$  premières retenues  $r_0 ; r_1 ; r_2 ; \dots ; r_k$  sont non nulles.

Ceci arrive quand le tas possède  $2^{k+1} - 1$  éléments.

Pour tout autre cardinal,  $P_k$  n'est pas vérifié.

Ainsi, lorsque  $M$  tend vers l'infini,  $P_{k \in [0, N]}$  est vérifié en moyenne 0 fois.

Pour  $k = N$ ,  $P_N$  est vérifié lorsqu'il y a défondement de retenue.

Ceci arrive dès que le tas possède  $2^N - 1$  éléments.

Pour toute insertion supplémentaire,  $P_N$  reste vraie.

Donc en moyenne, lorsque  $M$  tend vers l'infini,  $2^N - 1$  devient négligeable et  $P_N$  sera vérifié en moyenne 1 fois.

Question 28

### Question 29

arbre\_binomial union-tas ( tas\_binomial tas1, tas\_binomial tas2 ) {

    tasre\_binomial r0, r1 ;

    r0 = NULL ;

    for ( int i=0 ; i < tas2.taille ; i++ ) {

        r1 = ou ( et ( r0, ou ( tas1.tableau[i], tas2.tableau[i] ) ),  
                 et ( tas1.tableau[i], tas2.tableau[i] ) );

        tas1.tableau[i] = ou-exclusif ( r0,  
                                       ou-exclusif ( tas1.tableau[i], tas2.tableau[i] ) );

        r0 = r1     }

    if ( r0 != NULL && tas1.taille == tas2.taille )  
        { return r0; }

    else { return NULL; }

}

Question 30 Le pire des cas survient lorsque  
il y a débordement de ramme.

Ce qui arrive si les deux tas ont la même  
capacité  $\frac{1}{2}$  et sont complets car ils possèdent  
 $2^k - 1$  éléments chacun.

Les comparaisons apparaissent dans l'instruction for,  
les appels ou à la fonction ou (2 fois),  
et (2 fois) et ou-exclusif (2 fois).

Ainsi que 2 comparaisons en dehors de la boucle.

Le nombre de comparaisons est donc proportionnel au nombre de tour de boucle.

Dans le pire des cas, il y a  $N$  tours de boucle avec  $N$  la capacité tas1.taille.

Comme  $\text{card}(N) = 2^N - 1$

On a une complexité en  $O(\log_2(N))$  dans le pire des cas avec  $N = \max(N_1, N_2)$  où  $N_1$  est la capacité du tas 1 et  $N_2$  celle du tas 2.

Question 31

```
int position_minimale_tas(tas t, int i){  
    int imin = -1;  
  
    for (int i=0; i < t.taille; i++) {  
        if (t.tableau[i] < t.tableau[imin])  
            { imin = i; }  
    }  
    return imin; }
```

### Question 32

```

tas-binomial decremente (abre-binomial a, int n) {
    tas-binomial t = initialise_tas (n);
    liste-chainee tete = a → fils;
    for (int i = n-1; i > -1; i--) {
        t.tableau[i] = tete → element;
        tete = tete → suivant;
    }
    return t;
}

```

3

### Question 33

```

int retrait_minimum_tas (tas-binomial t) {
    int imin = position_minimum_tas (t); // O(N)
    abre-binomial amin = t.tableau[imin];
    t.tableau[imin] = NULL; // O(1)
    tas-binomial tmin = deacrementi (amin, imin); // O(imin) C O(N)
    union-tas (t, tmin) // renvoie forcément NULL
    return amin.valeur;
}

```

Question 34 Soit  $N$  la capacité du tas  $t$   
(t. tas) :

Alors chacune des fonctions appelée est en  $\Theta(N)$ .

La fonction `retrait_minimum_tas` est donc en  $\Theta(N)$ . Dans le pire des cas le tas est complet et :

Puisque  $\text{Card}(t) = 2^N - 1$

$$m \geq \log_2(\text{Card}(t)+1) = N$$

La complexité dans le pire des cas est donc en  $O(\log_2(m))$  avec  $m$  le nombre d'éléments du tas.

Question 35 Le type abstrait file à priorité admet comme interface :

- \* création d'une file vide  
connaissée par `tas_binomial initialise_tas(int max);`
- \* ajout d'une nouvelle valeur dans la file :  
`arbre_binomial insertion_tas(int valeur, tas_binomial t);`
- \* renvoie de la valeur minimale de la file  
et mise à jour de la file en supprimant cette valeur minimale :  
`int retrait_minimum_tas(tas_binomial t);`

Pour compléter le type, il manque les fonctionnalités suivantes :

- \* est-file-vide : teste si la file est vide
- \* mise-a-jour (int ancienne Valeur, int nouvelle Valeur) ; qui modifie la valeur d'un élément dans la file de priorité.

Question 36 : En notant  $n$  le nombre d'éléments à trier :

void tri-binomial (int taille, int tab [ ]) {

    int capacité = ceil ( log2 (taille) + 1 ) ; // O(s)

    // ajoute les valeurs du tableau dans un tas

    tas-binomial t = initialise-tas (capacité) ; // O(log n)

    for (int i = 0 ; i < taille ; i++) {

        insertion-tas (tab[i], t) } // O(log<sub>2</sub> n)

    // mises à jour du tableau

    for (int i = 0 ; i < taille ; i++) {

        tab[i] = retrait-minimal-tas (t) ; } // O(log<sub>2</sub>(n))

}

Question 37 : En notant  $n$  le nombre d'éléments à trier, on a : chaque boucle effectue  $n$  itérations et chaque itération appelle une fonction qui est d'une complexité temporelle de  $\log_2(n)$  dans le pire des cas.

Cette fonction de tri binomial est d'une complexité temporelle  $\Theta(n \log_2(n))$ .

Question 38 Soit  $n$  le nombre d'éléments à trier. L'occupation mémoire du tableau est d'une complexité  $\Theta(n)$ .

Déterminons la complexité spatiale d'un tas de  $n$  éléments.

Un tas est constitué d'un nombre entier (taille) et d'un tableau de taille + 1 (cases).

Puisqu'il y a  $n$  élément, la longueur du tableau sera égale à  $\lfloor \log_2(n+1) \rfloor + 1$  cases.

Ceci est une conséquence de la relation  $n = 2^m - 1$ .

Le tableau de pointeurs pointe vers  $\lfloor \log_2(n+1) \rfloor + 1$  arbres binomiaux d'ordres  $0, 1, \dots, \lfloor \log_2(n+1) \rfloor$ .

Et nous avons vu que un arbre binomial d'ordre  $k$  occupe  $A + B2^k$  octets.

Ainsi, l'occupation mémoire générée par le tableau est au plus égale à :

$$M = \sum_{k=0}^{\lfloor \log_2(n+1) \rfloor} (A + B2^k) = (\lfloor \log_2(n+1) \rfloor + 1) A + B \sum_{k=0}^{\lfloor \log_2(n+1) \rfloor} 2^k$$

$$M = A + A \lfloor \log_2(n+1) \rfloor + B(2$$

ou pour  $n > 1$  on a

$$\lfloor \log_2(n+1) \rfloor \leq \log_2(n+1) \leq \log_2(2n) = \log_2(2 \cdot n)$$

$$\leq 2 \log_2 n \in \Theta(\log_2 n)$$

et

$$2^{\lfloor \log_2(n+1) \rfloor} \leq 2^{\log_2(n+1)+1} = 2 \times 2^{\log_2(n+1)}$$

$$\leq 2 \times 2^{\log_2(2n)} = 2 \cdot 2n = 4n$$

$$\in \Theta(n)$$

Donc on a

$$M \in \Theta(\log_2 n) + \Theta(n) \in \Theta(n)$$

La complexité spatiale est donc en  $\Theta(n)$   
pour  $n$  le nombre d'éléments à trier. pour  
les binomials

Finalement, l'occupation mémoire générée par  
la fonction de tri est en  $\Theta(n)$ .

### Question 3g)

Le tri par tas binomial ~~est~~ apparaît comme un tri de point de vue de ses complexités.

C'est un tri par comparaison qui a une complexité de  $O(n \log_2(n))$  ce qui est optimal pour cette classe de tri par comparaison.

Concernant sa complexité réelle, ~~elle~~ est en  $O(n)$  ce qui est aussi optimal puisque de fait, les valeurs à trier occupent nécessairement un espace en  $O(n)$  (ce qui est le cas du tableau en entrée).

Donc, aux constantes d'implémentations près, c'est un excellent algorithme de tri que nous avons décortiqué durant ces ~~nombreuses~~ nombreuses heures d'étude.

Merci.