

Cours 11 - correction annale

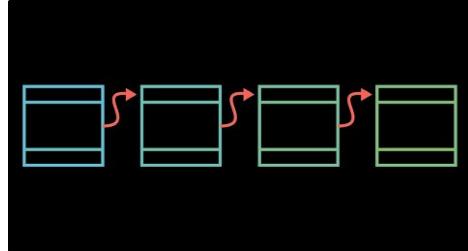
2021

But how does bitcoin actually work?

The math behind cryptocurrencies. Help fund future projects:

<https://www.patreon.com/3blue1brown> An equally valuable form of support is to simply share some of...

▶ <https://www.youtube.com/watch?v=bBC-nXj3Ng4&feature=share>



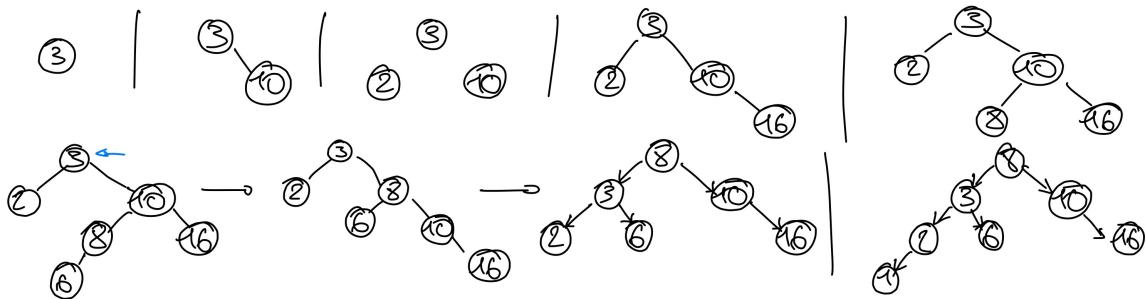
Correction annale

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e301ac20-a235-45a7-bf16-525d3d29ef90/exam2021.pdf>

Exos d'application du cours (questions sur les efficacités des différentes opérations sur différentes structures) puis exo d'application où on fait du code

Exo 1

- ▼ 1) Insertion 3, 10, 2, 16, 8, 6, 1 dans un AVL(= Arbre binaire équilibré)



▼ 2) Représentation table de hachage

0	16 ← 8
1	1 ← 1
2	10 ← 2 ← 6
3	3

▼ 3) Structure la plus adaptée pour des entiers de 1 à 8? à 32?

Pour 1 à 8 on préfère la table de hachage car au plus 2 tests (au lieu de 3 pour l'AVL)

Pour 1 à 32 on préfère l'AVL car au plus 6 tests (au lieu de 8 pour la table de hachage)

Pour un AVL et des entiers de 1 à n on fait $\log_2(n) + 1$ tests

▼ 4) Structure la plus adaptée pour des entiers de 1 à m (m très grand)

Un tableau de booléen suffit → test en $O(1)$

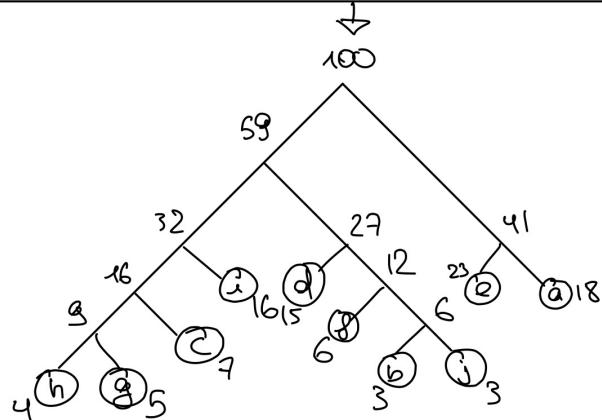
Exo 2

Le code de Huffman crée autant d'arbres que de caractères différents, puis fusionne à chaque étape les arbres dont les clés ont les plus petites fréquences

▼ 1) Construction de l'arbre du code de Huffman

a	b	c	d	e	f	g	h	i	j
18	3	7	15	23	6	5	4	16	3

e:23 a:18 i:16 d:15 c:7 f:6 g:5 h:4 b:3 j:3



▼ 2) Étape la plus coûteuse? Structure adaptée

La **recherche du minimum** est la plus coûteuse, la meilleure structure pour l'implémenter est un **tas** (suppression du min en $O(\log 2(n))$ avec n la taille; un tas a son min à la racine, et le 2ème plus petit élément est tout en bas à gauche)

▼ 3) Quelle structure de données permet de coder un message par le code d'Huffman le plus rapidement?

Table de hachage:

"a"	11
"b"	0100
"c"	0001

Exo 3

$G = (S, A)$, $S \in [1, m - 1]$

A ensemble d'arêtes, chaque arête a une valeur associée $val(u, v)$

Un **arbre couvrant** de G est un sous-graphe de G qui ne contient pas de cycle et qui relie toute paire de noeuds par une chaîne.

```
typedef struct arete{
    int vois; /* voisin du sommet */
    float val; /* valeur de l'arete */ struct arete* suiv;
} Arete;

typedef Arete* Liste_arete;

typedef struct{
    int nbsom; /* Nombre de sommets dans le graphe */
    Liste_arete* TabSom; /* vecteur des listes d'adjacences des sommets */
} Graphe;
```

▼ 1) Fonction `float val_arete(Graphe *G, int u, int v);` qui renvoie la valeur $val(u, v)$ de l'arête (u, v) si l'arête existe et -1 si elle n'existe pas

```
float val_arete(Graphe* G, int u, int v){
    Arete* cour = G->TabSom[u];
    while((cour!=NULL)&&(cour->vois!=v)){
        cour=cour->suv;
    }
    if(cour!=NULL){
        return cour->val;
    }
    return -1;
}
```

On rappelle qu'une **arborescence enracinée en r** est un arbre orienté de manière à ce que les arcs forment des chemins allant de r à tous les autres sommets.

▼ 2) Donner la fonction `float valeurACM(Graphe *G, int *ACM);` qui calcule et renvoie la valeur d'un arbre couvrant stocké dans le tableau ACM

```
float valeurACM(Graph* G, int* ACM){
    float val_acm=0;
    int i;
    for(i=0; i<G->nbsom; i++){
        if(ACM[i]!=-1){
            val_acm=val_acm+val_arete(G,i,ACM[i]);
        }
    }
    return val_acm;
}
```