# INF 101 : Les trucs utiles d’informatique théorique

Pour ce qui concerne les structures de données basiques et leurs implémentations, c’est relativement facile. A noter simplement que lorsqu’on implémente une file par une liste/tableau, on décale la « partie active » du tableau qui permet d’implémenter la file au fur et à mesure des défilements et des enfilements.

[00000000011111111111111110000000000] 🡨 Voici ici un tableau exemple

*Partie déjà Partie en cours La partie restante du tableau*

*Défilée d’utilisation (la file) pour des enfilements futurs (espace libre)*

En ce qui concerne les généralités sur les arbres, il faut connaître le vocabulaire et les parcours basiques d’un arbre : préfixe (1ère rencontre du nœud), infixe (2e rencontre du nœud) et suffixe/postfixe (3e rencontre du nœud, ou dernière rencontre). Il faut faire attention dans le cas INFIXE : lorsqu’un nœud est placé en descendant DROIT d’un autre nœud, son père sera marqué AVANT lui nécessairement. Car la branche gauche (même si elle est vide, dans ce cas la visite est artificielle) aura été visitée au préalable forcément et donc le père est visité 2 fois avant qu’on arrive au descendant. C’est le seul piège.

On retiendra arbre complet (chaque nœud interne a exactement 2 fils), arbre parfait (progression de haut en bas et de gauche à droite) et équilibré (pas de différence de plus de 1 de hauteur entre deux sous-arbres). Parfois un arbre complet doit avoir toutes ses feuilles à la même profondeur, ce qui signifie qu’il est parfait-complet.

## Recherche et Tri

Recherche dichotomique : pour une liste de taille n, la complexité est en O(ln(n)). Cela se montre simplement en disant que la longueur de la liste est majorée par n/2p au bout de p passages dans la boucle ‘Tant que’ et comme l’algorithme tourne tant que cette longueur de liste est supérieure à 1, on obtient l’inégalité n/2p ≥1 et donc p≤log2(n) équivalent à ln(n).

Pour les algorithmes dits comparatifs, on peut donner une majoration en O(nln(n)) en prenant comme pire cas qu’à chaque permutation considérée, la comparaison élimine au plus la moitié des permutations encore envisageables. (Pour a,b,c : a<b ? Non. a<c ? Non. Et donc pour 3 valeurs on est obligés de faire 3 tests alors que deux auraient pu suffire.) La complexité se calcule en supposant qu’il reste n!/2k permutations envisageable (sur le raisonnement précédent) et que l’algorithme ne se termine que quand le nombre de permutations restantes est <2. On obtient alors log(n!) comparaisons en pire cas, équivalent à nln(n).

On citera juste un exemple de tri non comparatif (qui se base sur la comparaison déjà faite de l’ordinateur entre les indices d’une liste) : A chaque entier qu’on parcourt dans la liste à trier (disons p) on ajoute 1 dans une nouvelle liste T dans la case T[p]. Ainsi, une liste finie peut donner [0,1,0,3,2,1,1,2,0,4…] et commencera par 1,3,3,3,4,4,5,6,7,7,9,9,9,9…

### Exemples de tri à connaître :

**Sélection** : on cherche la plus petite valeur et on **l’échange** avec la valeur à la place à laquelle elle devrait être. La première plus petite est échangée avec la valeur en case 0, puis la deuxième plus petite (i.e. dans le sous-tableau de 1 à n) est échangée avec la valeur en case 1…

*Complexité : le pire cas et le meilleur cas ont la même en O(n²)*

**Insertion :** On suppose que la sous-liste de 1 à i est déjà triée et on y ajoute à la bonne place l’élément suivant (i+1). Pour « ajouter à la bonne place », on décale au fur et à mesure les données plus grandes que celle se trouvant à l’origine en i+1. (Un peu comme un tri à bulles sur la sous-liste).

*Complexité : le pire cas et la moyenne ont une complexité en O(n²), mais dans le cas où la liste est presque triée, on voit que la complexité est en O(n).*

**Rapide :** On choisit un pivot on met tout ce qui est plus grand à droite et tout ce qui est plus petit à gauche, et on recommence sur chaque côté (défini récursivement très facilement).

*Complexité : Pire cas en O(n²) s’il faut aller chercher le pivot à une extrémité à chaque fois par exemple. Le meilleur des cas est quand la liste est toujours partagée en deux sous-listes de même longueur, i.e. au milieu. Dans ce cas on fait un appel partition sur la liste de longueur n, pour deux appels sur des longueurs n/2, pour 4 sur des longueurs n/4… On a en tout une complexité de l’ordre de nln(n). La complexité moyenne est également en nln(n).*

### Arbres binaires de recherche

Le parcours infixe (ou symétrique) d’une ABR renvoie la liste triée des éléments de l’arbre. Pour insérer un nouveau nœud dans un ABR sans en perdre la qualité, on le compare d’abord à la racine, puis on descend récursivement dans les bons sous-arbres au fur et à mesure, le cas d’initialisation étant le sous-arbre vide. On crée alors deux adresses de descendant vides.

Complexité des ABR : (page 37 à relire)

### Structure de Tas

Un tas est un AB parfait, dans lequel chaque père est plus grand que ses fils. C’est la seule condition pour avoir un tas, il n’y a aucune relation impliquant des oncles. L’intérêt de la structure d’arbre parfait est de connaître à l’avance les numéros des nœuds et de leur père. (Partie entière / 2)

Pour passer d’une structure de liste à un tas, le processus est simple : on respecte la structure d’arbre binaire parfait, PUIS on respecte la structure de tas en faisant remonter au besoin l’élément que l’on vient d’insérer. Pour insérer une valeur dans un tas déjà constitué, le processus est le même.

Le pseudo-code d’insertion est intéressant par rapport aux indices des différents nœuds (prévisibles) dans un ABP :

* i prend la valeur p
* clé prend la valeur T[p]
* Tant que i≥2 et clé>T[partEnt(i/2)]
* - T[i] prend la valeur T[partEnt(i/2)]
* - i prend la valeur partEnt(i/2)
* T[i] prend la valeur clé

La complexité de cet algorithme est en h=ln(p) avec h la hauteur et p le nombre de données dans le tas. (On fait au plus h comparaisons / échanges d’éléments après insertion de la donnée à remonter)

## Le Hachage (compression)

Ça c’est important mais ce n’est finalement que du codage par des valeurs basses dans un tableau réduit (« lou » = 31) dans lequel on peut implémenter plusieurs façons de gérer les collisions : hachage linéaire (cyclique vers la droite, i.e. vers la droite et on recommence à gauche lorsqu’on arrive au bout), hachage avec chaînage interne (il y a une zone de débordement à droite, et on rajoute une ligne de plus au tableau pour stocker l’adresse du suivant en ordre de collision. Lorsqu’il n’y en a pas, on met -1. On remplit la zone de débordement en partant de la droite, ce qui fait que quand elle est pleine, on vient déborder sur la zone « normale » en partant de la droite. Et enfin le hachage avec chaînage externe, où on utilise l’allocation dynamique : on utilise chaque case du tableau comme début d’une liste chaînée, et les listes chaînées sont les stockeurs des collisions.

## Algorithme de Huffman – codage entropique

PeepoWide

## Graphes et autres trucs vus non au programme du quizz (notamment calcul de plus court chemin, arbres couvrants de poids minimum, BFS en O(m), Dijkstra en O(n²+m), Bellman en O(n²+m), Bellman-Ford en O(mn) , Ford-Dantzig O(m²), Floyd – ah non)