Chapitre 21

Structures de données

Sommaire.

Les propositions marquées de ★ sont au programme de colles.

1 Listes chaînées, piles et files.

Définition 1: Liste chaînée.

Une **liste chaînée** est une structure qu'on peut définir inductivement :
- Soit c'est une liste vide.
- Soit c'est un noeud, appelé **cellule**, qui contient une valeur et une référence à une autre liste chaînée.

Définition 2: Pile.

Une **pile** est une structure de données qui permet de stocker des éléments et de les retirer dans l'ordre inverse de leur insertion.
Interface :
- **Empiler** un élément.
- **Dépiler** un élément.
- **Tester** si la pile est vide.

Définition 3: File.

Une **file** est une structure de données qui permet de stocker des éléments et de les retirer dans l'ordre de leur insertion.
Interface :
- **Enfiler** un élément.
- **Défiler** un élément.
- **Tester** si la file est vide.

2 Arbres.

2.1 Arbres binaires.

2.1.1 Définitions.

Définition 4: Arbre binaire.

Un **arbre binaire** est une structure de données hiérarchique où chaque élément est un **noeud**.
Il est défini de manière inductive :
- Soit c'est une arbre vide.
- Soit il est constitué d'un noeud et de deux arbre binaires disjoints, appelés **fils gauche** et **fils droit**

Définition 5: Racine

La **racine** d'un arbre est le seul noeud qui n'a pas de père.

Définition 6: Feuilles, noeuds internes.

Les **feuilles** d'un arbre sont les noeuds qui n'ont pas de fils.
Une **branche** est un chemin de la racine à une feuille.
Les **noeuds internes** sont les noeuds qui ont au moins un fils.
On a donc $|\texttt{feuilles}| + |\texttt{noeuds internes}| = |\texttt{noeuds}|$

Définition 7: Arbre binaire strict.

Un arbre binaire strict est un arbre où tous les noeuds internes ont deux fils.

Proposition 8: Dénombrement. ★

Dans un arbre binaire strict non vide, le nombre de feulles est égal au nombre de noeuds internes plus un.

Preuve :

On a d'abord : $|\texttt{feuilles}| = |\texttt{noeuds}| - |\texttt{noeuds internes}|$.
On dénombre les liaisons père-fils : $N = 2 \cdot |\texttt{noeuds internes}|$ (les noeuds internes ont deux fils).
On dénombre les liaisons fils-père : $N = |\texttt{noeuds}| - 1$ (la racine n'a pas de père).
En combinant les deux : $|\texttt{noeuds}| - 1 = 2|\texttt{noeuds internes}|$.
Donc $|\texttt{feuilles}| = |\texttt{noeuds internes}| + 1$.

Définition 9: Vocabulaire.

La **taille** d'un arbre est son nombre de noeuds.
La **profondeur** d'un noeud est sa distance à la racine.
Un **niveau** d'un arbre est l'ensemble des noeuds de même profondeur.
La **hauteur** d'un arbre est la profondeur maximale de ses noeuds.

Convention: La hauteur d'un arbre vide est -1, la hauteur d'un arbre réduit à sa racine est 0.

Définition 10: Arbre binaire parfait.

Un arbre binaire est **parfait** si toutes ses feuilles sont de même profondeur et que tous ses noeuds internes possèdent deux fils.

Proposition 11: Nombre de noeuds d'un arbre parfait. ★

Un arbre binaire parfait de hauteur h a $2^{h+1} - 1$ noeuds.
De plus, pour tout $k \in [\![0, h]\!]$, il a 2^k noeuds de profondeur k.

Preuve :

Montrons le par récurrence sur la hauteur.
Initialisation. Un arbre de hauteur 0 est réduit à sa racine et de taille $1 = 2^{0+1} - 1$.
Hérédité. Supposons la propriété vraie sur les arbres de hauteur h.
Soit un arbre binaire parfait de hauteur $h+1$.
Un noeud de profondeur $k+1$ est le fils d'un noeud de profondeur k, qui a deux fils.
Ainsi, il y a deux fois plus de noeuds de profondeur $k+1$ que de noeuds de profondeur k.
Alors le nombre de noeuds total vaut $\sum_{k=0}^{h} 2^k = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$.
La propriété est vraie pour $h+1$.
Par récurrence, elle est vraie pour tout $h \in \mathbb{N}$.

Définition 12: Arbre binaire complet.

Un arbre binaire de hauteur h est **complet** si tous ses niveaux sont remplis, sauf peut-être le dernier, qui est rempli de gauche à droite.

Corrolaire 13: Nombre de noeuds d'un arbre complet.

La taille d'un arbre binaire complet de hauteur h est comprise entre 2^h et $2^{h+1} - 1$.
La hauteur d'un arbre complet de taille n est $\lfloor \log n \rfloor$.

Définition 14: Étiquette.

Une **étiquette** est une valeur associée à un noeud d'un arbre.

2.1.2 Parcours.

Définition 15: Parcours en profondeur. ★

Un **parcours en profondeur** est un parcours où on termine d'explorer une branche avant d'en visiter une autre.
Il existe trois types de parcours en profondeur :
- **Préfixe** : on visite le noeud, puis le fils gauche, puis le fils droit.
- **Infixe** : on visite le fils gauche, puis le noeud, puis le fils droit.
- **Postfixe** : on visite le fils gauche, puis le fils droit, puis le noeud.

Définition 16: Parcours en largeur. ★

Un **parcours en largeur** est un parcours où on visite les noeuds de même profondeur avant de passer à la profondeur suivante.
On utilise une file pour implémenter ce parcours.

2.2 Tas.

2.2.1 Définitions.

Définition 17: File de priorité.

Une **file de priorité** est une structure de données qui permet de stocker des éléments d'un ensemble totalement ordonné.
Interface :
- Insérer un nouvel élément.
- Extraire l'élément le plus grand.
- Modifier la valeur d'un élément.
- Tester si la file est vide.

Définition 18: Tas.

Un **tas** est un arbre binaire complet tel que tout noeud porte une étiquette supérieure à celle de ses fils.

2.2.2 Implémentation.

Définition 19: Percoler vers le bas. ★

Algorithme 1 : Percoler vers le bas
Entrées : Un tas A, un indice i et une valeur v
Sorties : Un tas similaire à A, tel que $A[i] = v$
1 $A[i] \leftarrow v$.
2 max $\leftarrow$ indice du noeud d'étiquette maximale entre $A[i]$ et ses fils.
3 **si** max $\neq i$ **alors**
4 $A[i] \leftarrow A[\max]$.
5 `percoler_vers_le_bas`(A, max, v).
6 **fin**

Terminaison.
Les lignes $1, 2, 3, 4$ se terminent. Le variant d'appel est la hauteur du noeud d'indice i.
Complexité.
Notons $T(h)$ le nombre d'opérations élémentaires pour une certaine hauteur h.
On a $T(0) = \alpha$ et $T(h) = \alpha + T(h-1)$ donc $T(h) = \alpha h = O(h)$.
Or $2^h \leq n \leq 2^{h+1} - 1$ avec n le nombre de noeuds de l'arbre.
Donc $h \leq \log_2(n) \leq h+1$: complexité dans le pire des cas en $O(\log n)$.
Correction.
Soit h la hauteur d'un noeud d'indice i.
Si $h = 0$, alors c'est une feuille et $max = i$: il n'y a pas d'appel.
Supposons que l'appel est correct pour une certaine hauteur $h-1$. Montrons que l'appel sur h fonctionne.
On prend un indice i d'un noeud de hauteur h.
Si $max = i$ alors par hypothèse de récurrence, les sous-arbres de i sont des tas et l'appel est correct.
Si $max \neq i$ alors on remplace $A[i]$ par le max et la condition est donc vérifiée entre i et ses fils.
Par hypothèse de récurrence, on sait que l'appel récursif est correct.

Définition 20: Percoler vers le haut.

Algorithme 2 : Percoler vers le haut
Entrées : Un tas A, un indice i et une valeur v
Sorties : Un tas similaire à A, tel que $A[i] = v$
1 $A[i] \leftarrow v$
2 min $\leftarrow$ indice du noeud d'étiquette minimale entre $A[i]$ et son père.
3 **si** max $\neq i$ **alors**
4 $A[i] \leftarrow A[\max]$
5 `percoler_vers_le_haut`(A, min, v)
6 **fin**

Mêmes propriétés que pour `percoler_vers_le_bas`.

Définition 21: Modification.

Algorithme 3 : Modification
Entrées : Un tas A, un indice i et une valeur v
Sorties : Un tas similaire à A, tel que $A[i] = v$
1 **si** $v > A[i]$ **alors**
2 `percoler_vers_le_bas`(A, i, v)
3 **fin**
4 **sinon**
5 `percoler_vers_le_haut`(A, i, v)
6 **fin**

Sa correction / complexité découle de celles de `percoler_vers_le_bas` et `percoler_vers_le_haut`.

Définition 22: Insertion.

Algorithme 4 : Insertion
Entrées : Un tas A et une valeur v
Sorties : La valeur v a été insérée dans le tas A.
1 Ajouter v à la fin du tas.
2 `percoler_vers_le_haut`(A, n, v).

Définition 23: Extraction.

Algorithme 5 : Extraction
Entrées : Un tas A
Sorties : La valeur maximale a été extraite du tas A.
1 **si** le tas est de taille 1 **alors**
2 supprimer son élément et le renvoyer.
3 **fin**
4 Garder en mémoire le plus grand élément du tas (le premier).
5 Remplacer le premier élément par le dernier.
6 Le percoler vers le bas.
7 **retourner** le plus grand élément.

Définition 24: Construire un tas. ★

Algorithme 6 : Construire un tas
Entrées : Un tableau T
Sorties : T tel qu'il vérifie la condition de tas.
1 **pour** i de $\lfloor n/2 \rfloor$ à 1 **faire**
2 `percoler_vers_le_bas`(T, i, $T[i]$).
3 **fin**

Compléxité.
Soit H la hauteur du tas. On sait que percoler_vers_le_bas est en $O(H)$.
À une certaine profondeur p, il y a au plus 2^p noeuds.
De plus, ces noeuds sont à hauteur soit $h = H - p$, soit $h = H - p - 1$, donc $p = H - h$ ou $p = H - h - 1$.
La complexité de percoler_vers_le_bas sur un noeud de hauteur h est donc de αh, $\alpha \in \mathbb{R}$.
Ainsi, il y a au plus 2^{H-h} noeuds de hauteur h.
Alors, dans le pire des cas :

$$\sum_{h=0}^{H} \alpha h \cdot 2^{H-h} = \alpha 2^H \sum_{h=0}^{H} \frac{h}{2^h}$$

On pose $f : x \mapsto \sum_{h=0}^{H} x^h$, alors $f(x) = \sum_{h=0}^{H} x^h = \frac{x^{H+1}-1}{x-1}$.
De plus, on pose $g : x \mapsto xf'(x)$, et pour $x < 1$: $f'(x) = \frac{Hx^{H+1} - Hx^H + x^H + 1}{(x-1)^2} = \frac{1}{(x-1)^2} + Hx^H \frac{x - 1 - \frac{1}{H}}{(x-1)^2} \to \frac{1}{(x-1)^2}$.
Alors $g(x) \leq \frac{x}{(x-1)^2} + \beta$ en particulier pour $\frac{1}{2}$.

$$\sum_{h=0}^{H} \alpha h 2^{H-h} \leq \alpha 2^H (2 + \beta)$$

Alors $\alpha 2^H \sum_{h=0}^{H} \frac{h}{2^h} = O(2^H = n)$.

2.3 Arbres binaires de recherche.

Définition 25: Arbre binaire de recherche.

Un **arbre binaire de recherche (ABR)** a des noeuds étiquettés dans un ensemble totalement ordonné et tel que l'étiquette de chaque noeud interne est supérieur aux étiquettes de son sours-arbre gauche, et inférieure aux étiquettes de son sous-arbre droit.
Ce type d'arbres n'est pas nécéssairement strict.

Proposition 26: Élément minimal.

Le minimum se trouve à l'extrémité gauche de l'arbre.

Preuve :

Par récurrence sur la hauteur h de l'arbre.
Initialisation: Pour $h = 0$, un seul élément, c'est l'extrémité gauche.
Hérédité: Supposons que la propriété est vraie pour les arbres de hauteur $h - 1$.
Soit un arbre binaire de recherche de hauteur h.
1er cas: Pas de sous-arbre gauche, la racine est le minimum.
2ème cas: Il y a un sous-arbre gauche, le minimum y appartient, or ce sous-arbre est de hauteur inférieure.
Par hyothèse, le minimum est à l'extrémité gauche de ce sous-arbre.
Donc le minimum est à l'extrémité gauche de l'arbre.
Par récurrence, on a bien la propriété.

Proposition 27: Parcours infixe. ★

Soit $\mathscr{A}$ un ABR de hauteur h et P_h : «Le parcours infixe de $\mathscr{A}$ donne une liste triée».

Preuve :

Initialisation. Pour $h = 0$ c'est trivial car l'arbre est réduit à sa racine.
Hérédité. Supposons P_h vrai pour toute hauteur **strictement** inférieure à h. Montrons P_h.
On note $\mathscr{A}_g$ le fils gauche de $\mathscr{A}$, $\mathscr{A}_d$ son fils droit et r sa racine.
Par supposition, le parcours est correct sur $\mathscr{A}_g$ et $\mathscr{A}_d$ car ils sont de hauteurs strictement inférieures à h.
Le parcours infixe parcourt d'abord $\mathscr{A}_g$, puis r, puis $\mathscr{A}_g$.
On a que tout élément de $\mathscr{A}_g$ est inférieur à r et que tout élément de $\mathscr{A}_d$ est supérieur à r par propriété des ABR.
Donc le parcours de $\mathscr{A}_g$ puis r puis $\mathscr{A}_d$ est dans l'ordre croissant.

2.4 Arbres rouge-noir.

2.4.1 Définitions.

Définition 28: Arbre rouge-noir.

Un **arbre rouge-noir (ARN)** est un ABR strict dont toutes les feuilles sont vides, dont chaque noeud est coloré en rouge, ou en noir, tel que :
- Chaque feuille est noire.
- La racine est noire.
- Les fils rouges d'un noeud noir sont noirs.
- Tout chemin de la racine à une feuille contient le même nombre de noeuds noirs.

Définition 29: Hauteur noire.

La **hauteur noire** d'un arbre rouge-noir est le nombre de noeuds noirs sur un chemin de la racine à une feuille.

Proposition 30

Soit un arbre rouge-noir de hauteur h et de taille n, on a $h \leq 2\log_2(n+1)$.

Preuve :

Montrons qu'un ARN de hauteur noire k a au moins $2^k - 1$ noeuds.
Initialisation. Pour $k = 0$, un arbre de hauteur noire 0 est réduit à sa racine, donc un seul noeud.
Hérédité. Supposons la propriété vraie pour tout arbre de hauteur noire inférieure à k. Montrons le pour $k+1$.
Les deux fils de la racine sont donc de hauteur noire k : ils ont au moins $2^k - 1$ noeuds.
Donc A a au moins $2(2^k - 1) + 1 = 2^{k+1} - 1$ noeuds.
Alors $k \leq \log_2(n+1)$ et $h \leq 2k$ donc $h \leq 2\log_2(n+1)$

2.4.2 Implémentation.

Théorème 31: Implémentation. ★

Implémentation des arbres rouge-noir.

2.5 Applications des arbres binaires.

Théorème 32: Complexité de tris par comparaisons. ★

Soit un algorithme de tri utilisant uniquement des comparaisons. Sa complexité temporelle dans le pire des cas est en $\Omega(n \log n)$ où n est la taille du tableau à trier.

Preuve :

L'action de l'algorithme est d'appliquer une permutation sur l'entrée, dépendant uniquement de l'ordre relatif de ses éléments. On considère l'arbre de flot de contrôle de l'algorithme sur une entrée donnée. On arrive à une feuille si le tri est terminé, correspondant à une unique permutation de l'entrée. On a donc $n!$ feuilles.
On note h la hauteur de cet arbre, et on a alors que $2^h \geq n!$ donc $h \geq \log(n!)$. Alors :

$$h \geq \log(n!) = \sum_{k=1}^{n} \log(k) \geq \sum_{k=n/2}^{n} \log(k) \geq \frac{n}{2} \log\left(\frac{n}{2}\right).$$

On en conclut que $h = \Omega(\frac{n}{2})$.

Définition 33: Tri par tas. ★

Algorithme 7 : Tri par tas
Entrées : Un tableau T
Sorties : T trié
construire_tas(T)
pour i allant de n à 1 **faire**
échanger les cases 1 et i
percoler_vers_le_bas(T, 1, T[i])
fin

Complexité:
Soit n la taille de T. On sait que construire_tas s'effectue en $O(n)$ et percoler_vers_le_bas en $O(\log n)$.
L'échange des cases et la décrémentation s'effectuent en $O(1)$.
La boucle for se termine, le variant est $n - 1$, chaque itération se termine aussi et il y a n itérations.
On en déduit que cet algorithme est en $O(n \log n)$, donc en $\Theta(n \log n)$ d'après 32.

Définition 34: Tri rapide (sans doublons). ★

Algorithme 8 : Partitionner
Entrées : Tableau $\mathcal{T}$, entier debut, entier fin
Sorties : $\mathcal{T}$ partitionné et indice du pivot
pivot $\leftarrow \mathcal{T}$[debut]
inf $\leftarrow$ debut+1
sup $\leftarrow$ fin
tant que true **faire**
tant que $\mathcal{T}$[sup] $\geq$ pivot et sup > debut **faire**
sup $\leftarrow$ sup - 1
fin
tant que $\mathcal{T}$[inf] < pivot et inf < fin **faire**
inf $\leftarrow$ inf +1
fin
si inf $\geq$ sup **alors**
break
fin
echanger($\mathcal{T}$, inf, sup)
fin
echanger($\mathcal{T}$, debut, sup)
renvoyer sup

Algorithme 9 : Tri rapide
Entrées : Tableau $\mathcal{T}$, entier debut, entier fin
Sorties : $\mathcal{T}$ trié
si debut < fin **alors**
pivot $\leftarrow$ partitionner($\mathcal{T}$, debut, fin)
tri_rapide($\mathcal{T}$, debut, pivot-1)
tri_rapide($\mathcal{T}$, pivot+1, fin)
fin

Terminaison.
Les boucles while internes de `partitionner` se terminent car $O(1)$ et sup et inf sont variants. La boucle while externe se termine car $sup - inf$ est un variant. Les autres instructions sont en $O(1)$.
Alors `partitionner` se termine.
Les appels à `partitionner` se terminent donc `tri_rapide` aussi.
Correction.
Montrons la correction de `partitionner`.
Prédicat : «les cases d'indices debut+1 à inf-1 ont des valeurs strictement inférieurs au pivot, les cases d'indices sup+1 à fin ont des valeurs strictement supérieures au pivot».
Avant la boucle, l'ensemble des cases est vide, donc le prédicat est vrai.
Supposons que le prédicat est vrai au début d'une itération.
Les deux boucles internes gardent le prédicat, tout comme le reste des instructions.
C'est bien un invariant, il est vérifié en fin de boucle.
L'échange final permet de mettre le pivot au bon endroit.
`Partitionner` fonctionne donc correctement, et par récurrence sur fin-debut, `tri_rapide` aussi.
Complexité.
On prend la comparaison comme opération élémentaire.
On note S le nombre de noeuds de l'arbre des appels.
Alors $h = \Omega(\log S)$ donc $h = \Omega(\log n)$.
et $h = O(S)$ donc $h = O(n)$.
Le meilleur des cas correspond à la plus petite valeur de h : $\beta \log n$ alors on a une complexité en $O(n \log n)$.
Le pire des cas correspond à la plus grande valeur de h : βn, alors on a une complexité en $O(n^2)$.
Complexité en moyenne.
Si les données de départ dans un ordre aléatoire uniforme, alors c'est le cas de tout sous-tableau de l'entrée.
Soit $T(n)$ le nombre d'opérations élémentaires sur une entrée de taille n et $\overline{T}(n)$ le nombre moyen.

$$\begin{aligned}
& T(n) = n + 1 + T(k) + T(n-k-1) \\
\implies & \overline{T}(n) = n + 1 + \frac{1}{n}\sum_{k=0}^{n-1} \overline{T}(k) + \overline{T}(n-k-1) = n + 1 + \frac{2}{n}\sum_{k=0}^{n-1} \overline{T}(k) \\
\implies & n\overline{T}(n) = n(n+1) + 2\sum_{k=0}^{n-1} \overline{T}(k) \\
\implies & (n-1)\overline{T}(n-1) = n(n-1) + 2\sum_{k=0}^{n-2} \overline{T}(k) \qquad (n \leftarrow n-1) \\
\implies & n\overline{T}(n) - (n-1)\overline{T}(n-1) = n(n+1) - n(n-1) + 2\overline{T}(n-1) \\
\implies & n\overline{T}(n) = 2n + (n+1)\overline{T}(n-1) \\
\implies & \frac{1}{n+1}\overline{T}(n) = \frac{2}{n+1} + \frac{1}{n}\overline{T}(n-1) \\
\implies & \frac{\overline{T}(n)}{n+1} = \overline{T}(0) + 2\sum_{k=0}^{n+1} \frac{1}{k} = O(\log n) \qquad \text{(téléscopage)} \\
\implies & \overline{T}(n) = O(n \log n)
\end{aligned}$$

3 Dictionnaires.

Définition 35

Les **dictionnaires** permettent de manipuler des ensembles d'associations.
On appelle **clé** un élément de l'ensemble de départ et **valeur** son élément associé.
Interface:
- **Insertion** (clé, valeur).
- **Recherche** de clé.
- **Suppression** de clé.

Contraintes:
- Les clés sont toutes de mêmes types.
- Les valeurs sont toutes de mêmes types.
- Les clés sont uniques.

Si les clés sont des entiers, on peut utiliser un tableau.
On parle de table à adressage direct, on appelle chaque case du tableau une **alvéole**.

4 Tables de hachage.

Définition 36: Fonction de hachage.

Une **fonction de hachage** h est une fonction qui associe à chaque clé un entier, appelé **haché**.

Définition 37: Table de hachage.

Une **table de hachage** est un tableau associatif, où chaque case est une alvéole.
On utilise une fonction de hachage pour associer une clé à une alvéole.

4.1 Résolution des collisions par sondage.

Définition 38

Principe : S'il n'y a pas de place, on en cherche ailleurs.
Comment choisir le «ailleurs» ?
On définit un cycle σ, qu'on applique successivement au hachage d'une clé, jusqu'à trouver une place libre.

4.2 Résolution des collisions par chaînage.

Définition 39

Principe : On chaîne les clés avec le même hachage.
Dans le pire des cas : insertion en $O(1)$, recherche et suppression en $O(n)$.

Définition 40: Facteur de remplissage.

Le **facteur de remplissage** α est le rapport entre le nombre d'éléments stockés et le nombre d'alvéoles.

Théorème 41: Complexité. ★

Dans une table de hachage où les collisions sont résolues par chaînage, la recherche d'un élément a une complexité temporelle moyenne en $O(1 + \alpha)$.

Preuve :

On se place dans le cas où les hachés sont uniformément distribués.
1er cas: La recherche échoue.
Le calcul de la valeur hachée est en $O(1)$, le parcours moyen d'un liste chaînée est en $O(\alpha)$.
Ainsi, pour toute la recherche, on obtient une complexité moyenne en $O(1 + \alpha)$.

2ème cas: La recherche réussit.
Comme on est susceptible de s'arrêter avant la fin du parcours, la complexité est meilleure que dans le premier cas, donc en $O(1 + \alpha)$.