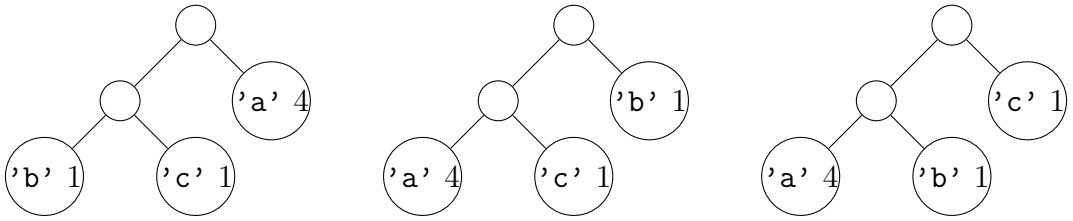
Algorithmique
TP 9 : Tas binaires

Le premier exercice de ce TP n'a pas besoin d'être déposé. Nous vous recommandons tout de même de le terminer chez vous.

1 Encodage de Hufman ✍

Nous vous recommendons de relire la partie du cours sur l'encodage de Hufman avant de commencer cet exercice. On va travailler sur les tas binaires, qui vont avoir la même structure que les arbres binaires.
Le but de cet exercice est de créer un tas binaire dans lequel on va ranger dans les feuilles certaines lettres de l'alphabet avec un poids associé. Plus précisément, à partir d'une chaîne de caractères, on va compter le nombre d'occurences de chaque lettre. Les lettres qui apparaissent dans la chaîne seront celles qui apparaîtrons (exactement une fois) dans le tas binaire. Le poids associé à une lettre correspondra à son nombre d'occurences dans la chaîne de caractère.
Pour chaque lettre dans le tas, on va avoir son poids et la profondeur de sa feuille dans le tas. Le but va alors être d'optimiser la répartition des feuilles dans le graphe, afin de minimiser la somme des poids*profondeur de chaque lettre.
Prenons l'exemple de la chaîne de caractère "abaaca". On va devoir mettre les feuilles `'a'` 4, `'b'` 1 et `'c'` 1. Voici toutes les façons de répartir les feuilles (à symétries près) :



On voit que le poids du premier arbre est de 8, alors que le poids des deux autres arbres est 11.
Le but est donc de créer un arbre de poids minimal à partir d'une chaîne de caractère. Pour faire cela, on va créer une liste d'arbres. Au début, on va mettre les feuilles avec la lettre et son poids. L'approche gloutonne suivante est alors optimale : on retire les deux arbres de poids minimum dans la liste, et on rajoute l'arbre composé d'un noeud ayant pour enfants les deux noeuds retirés. Quand il ne reste plus qu'un arbre dans la liste, on a terminé la construction.

Implémentation de la classe `BinaryNode` Nous avons implémenté pour vous une partie de la classe `BinaryNode` qui codera les arbres de Hufman. Un noeud a 4 attributs :

— `char letter` la lettre stockée par le noeud. Cette valeur contient un espace `'␣'` si on n'est pas une feuille. Si c'est une feuille, ça doit être une lettre de l'alphabet en minuscule.

— `int weight` qui donne le poids du noeud. Sur une feuille, cela correspond à un entier strictement positif associé à la lettre. Sur un noeud interne, cette valeur vaut la somme des poids des noeuds enfants.

— `BinaryNode leftSon` et `BinaryNode rightSon` qui représentent les deux noeuds enfants. Dans le cas des arbres de Hufman, soit on a aucun enfant, soit on en a deux.

Certaines méthodes ont déjà été codées. En particulier, le constructeur de feuille, les getters, setters, et le `toString`.

1. Coder le constructeur qui prend deux enfants et fait le noeud correspondant, en respectant les règles expliquées pour `letter` et `weight`.
2. Recoder la fonction `compareTo` qui compare le poids des noeuds (deux noeuds de même poids seront égaux selon cette méthode).
3. Coder les fonctions `isLeaf` qui vérifie si le noeud est une feuille et `isHufman` qui vérifie récursivement si le noeud est bien un arbre de Hufman.
4. Coder la fonction `mergeTrees` qui prend une liste d'arbres de Hufman, et utilise l'algorithme glouton expliqué plus haut pour tout fusionner en un seul arbre.
5. Coder la fonction `letterCount` qui prend une chaîne de caractère et renvoie un tableau de 26 cases, comptant les occurrences de chaque lettre de l'alphabet. On doit ignorer les autres caractères (ponctuation, chiffres...). On doit prendre en compte les majuscules et les minuscules. La case 0 va par exemple compter combien de `'a'` et `'A'` se trouvent dans la chaîne de caractères.
6. Coder la fonction `buildTree` qui prend le tableau de la question précédente et crée l'arbre de Hufman correspondant. Coder ensuite le constructeur qui crée l'arbre de Hufman à partir d'une chaîne de caractère.
7. Coder la fonction `weightTree` qui calcule le poids de l'arbre. Pour cela, il faut additionner, pour chaque lettre dans l'arbre, son poids multiplié par la profondeur où se trouve la feuille correspondante.

2 Min-tas

Dans cet exercice, vous allez implémenter un min-tas [1] (se référer au cours pour plus de détails) pour des valeurs entières.

Implémentation concrète On utilisera la représentation en tableau vue en cours, pour stocker les étiquettes des nœuds de l'arbre.
À la différence du cours : comme nous programmons en Java, le premier élément (la racine) a pour indice 0 et non 1.
Ainsi, dans cette représentation, les fils d'un nœud d'indice i ont pour indices $2i+1$ et $2i+2$. De même, l'indice du père d'un nœud d'indice $i > 0$ est la partie entière de $(i-1)/2$.

1. En passant : la classe `PriorityQueue` utilisée au TP précédent est un tas binaire, tel que celui que vous allez implémenter ; en l'occurence, c'est un min-tas quand il est utilisé sur les entiers munis de leur « ordre naturel ».

Le « tableau » (`ArrayList`) Pour éviter de gérer explicitement la taille du tableau et son niveau de remplissage, la classe `MinTas` fournie pour ce TP, n'utilise en fait pas directement un tableau, mais plutôt un `ArrayList` (attribut `elements`). Cette structure contient un tableau mais se charge de l'agrandir et de le rétrécir automatiquement quand c'est nécessaire. La complexité des opérations courantes est la même que si on utilisait un tableau [2]. Ainsi :

- `elements.size()` donne directement le nombre d'éléments actuellement dans le tas (et non la taille du tableau sous-jacent).
  Complexité : $O(1)$.
- `elements.add(x)` ajoute `x` à la fin de la liste (dont la taille est automatiquement incrémentée)
  Complexité : $O(1)$ [3].
- `elements.get(idx)` retourne l'élément en position `idx`
  Complexité : $O(1)$.
- `elements.set(idx, x)` écrase l'élément en position `idx` par `x` (si la liste a au moins `idx+1` éléments, sinon erreur)
  Complexité : $O(1)$.
- `elements.remove(idx)` supprime l'élément en position `idx` (déplace les éléments qui suivent vers la gauche, le cas échéant ; la taille de la liste est automatiquement décrémentée) et retourne la valeur supprimée
  Complexité : $O(1)$ [4] si `idx` est le dernier élément (ce qui sera le cas ici), sinon $O(N)$.
- `elements.indexOf(x)` cherche `x` dans le tableau et retourne son index si `x` est dans le tableau, `-1` sinon.
  Complexité : $O(N)$.

À faire

1. Complétez les méthodes `idxPere`, `idxFilsGauche` et `idxFilsDroite` donnant l'indice, respectivement, du père, du fils gauche, du fils droite du nœud dont l'indice est passé en paramètre.
2. Complétez la méthode `echangeSiNecessaire`, qui échange les éléments aux deux indices passés en paramètre s'ils n'étaient pas dans le bon ordre (c'est-à-dire si l'élément d'indice plus grand est plus petit que celui d'indice plus petit).
   Elle retourne `true` si un échange a effectivement eu lieu, `false` sinon.
   (Remarque : cette méthode ne doit être appelée que sur deux nœuds dont l'un est ancêtre de l'autre. Le résultat dans les autres cas est sans importance.)
3. Complétez la méthode `percoleVersLeHaut`.
   Cette méthode répare le tas en s'assurant que l'élément à la position passée en paramètre, son père, son grand-père, etc. jusqu'à la racine sont bien en ordre décroissant.

2. Complexité « amortie », pour être exact : en effet, quand un redimensionnement doit être réalisé, les ajouts et suppressions peuvent, occasionnellement mais rarement, prendre plus de temps.
3. Complexité « amortie » de $O(1)$ pour être exact.
4. Idem.

Pour ce faire, elle compare la valeur à l'indice courant à celle de son père et les échange si elles ne sont pas dans l'ordre, puis, s'il y a eu un échange, elle recommence avec le père et le grand-père, et ainsi de suite (elle s'arrête à la première position où un échange n'est pas nécessaire).

4. Complétez la méthode `insere`, qui ajoute un élément dans le tas en s'assurant que le tas reste un tas.

   Pour ce faire : l'élément est ajouté après la dernière position remplie du tableau, puis le tas est réparé en « percolant vers le haut » depuis cette position.

   Quelle est la complexité de l'insertion ?

5. Complétez la méthode `percoleVersLeBas`.

   Cette méthode répare le tas en s'assurant que le sous-arbre dont la racine est l'indice passé en paramètre est bien un tas.

   Pour ce faire elle procède à des échanges successifs entre l'indice courant et son plus petit fils, puis entre ce fils et son fils, et ainsi de suite, jusqu'à ce qu'on ne puisse plus échanger.

   — Initialement, l'indice courant est le paramètre.
   — Un échange est effectué si le nœud courant possède un fils de valeur plus petite que la sienne. Dans ce cas, l'échange est effectué avec le plus petit des fils. L'indice de ce dernier devient le nouvel indice courant.
   — On recommence tant que le nœud a un fils de valeur plus petite.

6. Complétez la méthode `elimine` qui retire du tas son élément minimal (situé à sa racine/l'index 0), s'assure que le tableau contienne toujours un tas correct, puis retourne cet élément.

   Pour ce faire, le dernier élément du tas est déplacé vers la première position (ce qui écrase la racine de l'arbre et diminue de 1 la taille de la liste), puis le tableau est réparé en « percolant vers le bas » depuis la nouvelle racine.

   Quelle est la complexité de l'élimination ?

7. (bonus) Programmez la méthode `elimineIdx` qui retire du tas l'élément d'indice passé en paramètre.

   Cette méthode fonctionne de la mème manière que `elimine`, sauf que :

   — c'est l'indice i à retirer qui est écrasé par le dernier élément (et non la racine/l'élément d'indice 0).
   — si l'élément que vous avez placé en i est plus petit que celui que vous avez écrasé, il est aussi nécessaire réparer le haut du tas en « percolant vers le haut » depuis cette position.

   Remarquez que la méthode `elimine` aurait pu être programmée par un appel à `elimineIdx` (lequel ?).

8. (bonus) Enfin, une question pour réfléchir : serait-il possible de programmer efficacement une méthode `elimineValeur` qui prend en paramètre, non pas un indice dans le tas, mais un élément à retirer du tas ?

   Quelle serait la complexité d'une implémentation triviale basée sur `indexOf` (d'`ArrayList`) et `elimineIdx` ?

   Que pourrait-on imaginer pour faire descendre la complexité à $O(\log N)$? (Ne le programmez-pas !)