Exercice 1. Tranche maximum

Soit `t` un tableau d'entiers. Une **somme consécutive** (ou tranche) dans `t` est de la forme $t_{i,j} = \sum_{k=i}^{j} \texttt{t.(k)}$ (où i et j sont des indices de `t`). On note s la valeur maximum d'une somme consécutive.

1. Écrire une fonction `tranche_max` prenant `t` en argument et renvoyant `s`, en complexité quadratique en la taille de `t`.

Si j est un indice de `t`, on note s_j la plus grande somme consécutive finissant en j. Dit autrement :

$$s_j = \max_{0 \leq i \leq j} t_{i,j}$$

2. Calculer tous les s_j, si `t = [|1; -4; 1; 5; -7; 0|]`
3. Si $j > 0$, montrer que :
$$s_j = \max(s_{j-1} + t.(j),\ t.(j))$$
4. Comment peut-on exprimer s en fonction de s_j ?
5. En déduire une fonction `tranche_max` prenant `t` en argument et renvoyant `s`, en complexité linéaire en la taille de `t`.
6. Donner un invariant de boucle permettant de prouver que `tranche_max t` est correct.
7. On se donne maintenant un entier K. Donner un algorithme, de préférence en $O(n)$, pour trouver la valeur maximum de $t_{i,j}$ pour $j - i < k$.
Indice : on pourra utiliser une file à deux bouts, qui permet l'ajout et la suppression en début et en fin de file en O(1).

Exercice 2. Suppression d'arête

1. Rappeler la définition d'une structure de union-find.

Soit T un graphe à n sommets obtenu à partir d'un arbre en ajoutant une arête.

2. Donner un algorithme (en pseudo-code) pour trouver une arête de T que l'on peut enlever de façon à obtenir un arbre. On essaiera d'avoir une complexité aussi faible que possible.
3. Implémenter l'algorithme précédent en OCaml. Le graphe est supposé être donné sous forme de liste d'adjacence.

Une arborescence est un graphe orienté acyclique ayant une racine r et tel que tout sommet autre que la racine possède un degré entrant égal à 1.
Soit $\vec{T}$ un graphe à n sommets obtenu à partir d'une arborescence en ajoutant un arc.

4. Reprendre les questions précédentes pour trouver un arc de $\vec{T}$ que l'on peut enlever de façon à obtenir une arborescence.

Exercice 3. Filtre de Bloom

1. Rappeler la définition d'une structure abstraite d'ensemble.

Un **filtre de Bloom** est une structure de données implémentant un ensemble. Il est composé de :

- Un tableau de bits t de taille p dont chaque bit est initialisé à 0.
- Des fonctions de hachage $h_i : U \longrightarrow \{0, ..., p-1\}$, pour $i \in \{1, ...k\}$, où U est l'ensemble des éléments pouvant être ajouté au filtre de Bloom. Dans toute la suite, on utilisera des entiers positifs pour U, avec le type `int`.

Par soucis d'efficacité, t sera stocké sous forme d'un entier (`int`), représenté en base 2. On pourra utiliser les opérateurs bit-à-bit suivants, où `a` et `b` sont des entiers :

- `a << b` (`a >> b`) : décale la représentation binaire de `a` de `b` bits vers la gauche (droite).
- `a & b` : « et binaire » de `a` et `b`.
- `a | b` : « ou binaire » de `a` et `b`.

2. Écrire une fonction `void add(int t, int e)` ajoutant `e` au tableau de bits `t`.
3. Écrire une fonction `int has(int t, int e)` renvoyant `1` si `e` appartient à `t`, `0` sinon.

Pour ajouter un élément $e \in U$ dans le filtre de Bloom, on met 1 dans t aux positions $h_1(e)$, ..., $h_k(e)$.

4. Écrire une fonction `void bloom_add(int t, int e)` effectuant cet ajout. Pour cette question et la suivante, on utilisera deux fonctions de hachage `int h1(int)` et `int h2(int)`, supposées définies (on a donc $k = 2$).

Pour tester si un élément e appartient au filtre de Bloom, on regarde si t ne contient que des 1 aux positions $h_1(e)$, ..., $h_k(e)$.

5. Écrire la fonction correspondante `int bloom_has(int t, int e)`, renvoyant 0 ou 1.
6. Cette dernière fonction peut avoir des faux-positifs : il est possible que `bloom_has(t, e)` renvoie 1 alors que `e` n'a jamais été ajouté à `t`.
   Expliquer pourquoi.

On veut maintenant estimer la probabilité d'erreur pour `bloom_has(t, e)`. Pour cela, on va supposer que les k fonctions de hachage sont uniformément réparties.

1. On suppose avoir ajouté n éléments dans le filtre de Bloom. Soit b un bit de t. Montrer que :

$$\mathbb{P}(b = 0) = (1 - \frac{1}{p})^{kn}$$

2. En supposant que chaque bit vaut 1 indépendamment des autres, donner la probabilité de faux-positif, puis donner un équivalent plus simple lorsque $\frac{p}{n} \longrightarrow 0$.

Exercice 4. Arbre de segments

Soit `a` un tableau d'entiers.
On souhaite concevoir, à partir de `a`, une structure de donnée `t` permettant de réaliser efficacement les opérations suivantes :

- `min_range i j t` : renvoie le minimum des éléments de `a` entre les indices `i` et `j`.
- `set i e t` : met à jour la structure pour que `a.(i)` soit remplacé par `e`.

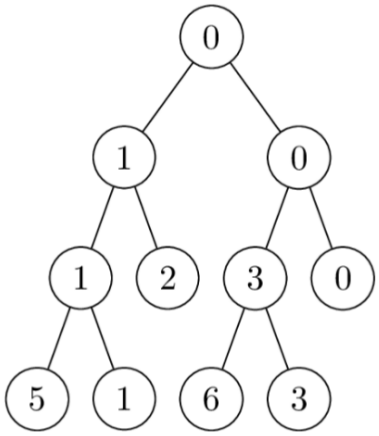
1. Proposer une solution naïve et donner sa complexité.

Dans la suite, on va utiliser un arbre de segments :

Définition : Arbre de segments

Un arbre de segments (pour un tableau `a`) est un arbre binaire dont :
- Les feuilles sont les éléments de `a`
- Chaque noeud est étiqueté par un triplet `(m, i, j)` tel que son sous-arbre contienne les feuilles `a.(i), ... , a.(j)` et `m` est le minimum de ces valeurs.

Par exemple, voici un arbre de segments obtenu à partir du tableau `[|5; 1; 2; 6; 3; 0|]`, où on a représenté seulement les minimums (premiers éléments de chaque triplet) :



Ainsi, les feuilles sont bien les éléments du tableau `[|5; 1; 2; 6; 3; 0|]` et chaque noeud correspond à un minimum sur une certaine plage du tableau.
Remarque : Il y a d'autres arbres de segments possibles pour le même tableau.
On utilisera le type suivant :

```
type tree = E | N of int * int * int * tree * tree
```

Ainsi, un sous-arbre `N(m, i, j, g, d)` possède `a.(i)`, `a.(i + 1)`, ..., `a.(j)` comme feuilles, de minimum `m` et de sous-arbres `g`, `d`.

2. Écrire une fonction `make : int array -> tree` qui construit un arbre de segments à partir d'un tableau d'entiers. On fera en sorte que l'arbre construit soit de hauteur logarithmique en la taille du tableau.
3. Écrire une fonction `set i e t` qui met à jour `t` en remplaçant `a.(i)` par `e`.
   Quelle est la complexité de cette fonction?
4. Écrire une fonction `min_range i j t` renvoyant le minimum des éléments de `a` entre les indices `i` et `j`.
5. Montrer que la complexité de `min_range i j t` est $O(\log(n))$, où n est la taille de `a`.
6. On s'intéresse à un autre problème : calculer efficacement une somme d'éléments entre les indices `i` et `j`, dans un tableau. Adapter les fonctions précédentes pour y parvenir.

J1 – Arbres évasés

Question 0.

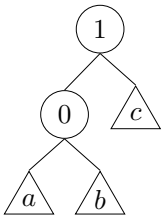
(a) Rappeler ce qu'est un arbre binaire de recherche.

(b) Quelle en est l'utilité ?

(c) Donner le pseudo-code de la fonction d'insertion dans un arbre binaire de recherche.

(d) Discuter de sa complexité en temps.

On se propose d'améliorer la complexité des requêtes sur un arbre binaire de recherche en l'équilibrant. Cet équilibrage s'appuie sur une opération locale, appelée *rotation*, dont il existe deux variantes symétriques. La *rotation à droite* agit sur un arbre de recherche de la forme suivante :



Elle le transforme en un autre arbre de recherche contenant les mêmes clefs, mais dont l'étiquette notée ici 0 est à la racine.

Question 1. Proposer une définition de l'opération de rotation à droite. En donner le pseudo-code.

Question 2.

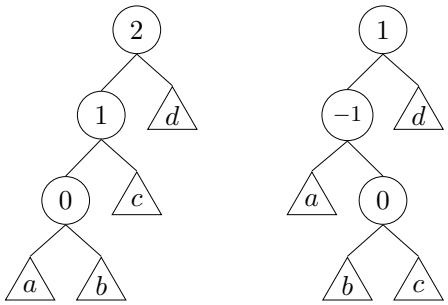
(a) En déduire un algorithme qui prend en paramètre un arbre binaire de recherche et l'étiquette de l'un de ses nœuds et transforme cet arbre binaire de recherche en déplaçant ce nœud à la racine. On supposera que toutes les étiquettes sont distinctes.

(b) Quelle est la complexité en temps de cet algorithme ?

Question 3. Proposer une amélioration de l'algorithme de recherche dans un arbre binaire de recherche, qui répond plus rapidement aux requêtes de recherche qui lui sont posées fréquemment. On ne demande pas ici de preuve de complexité.

Malheureusement, l'algorithme présenté à la question 3 n'a pas de bonnes propriétés de complexité théoriques. Nous allons étudier une autre approche.

Question 4. Considérons les arbres de recherche suivants :



(a) Comment peut-on les transformer afin de placer le nœud 0 à la racine ?

(b) En déduire une variante de l'algorithme donné en Question 3. Est-elle équivalente ?

On définit maintenant la *taille* d'un arbre a, notée taille(a), comme étant le nombre de nœuds qu'il contient. Son *rang* rg(a) est défini par $\text{rg}(a) = \log_2(\text{taille}(a))$. Enfin, le *potentiel* $\Phi(a)$ d'un arbre a est la somme des rangs de tous ses sous arbres.

Question 5.

(a) Soit t un arbre, et soit t' l'arbre résultant d'une rotation dans t (pas nécessairement à sa racine). De plus, soit t'_0 le sous-arbre de t' sur lequel la rotation a été effectuée, et t_0 le sous-arbre de t dont la racine a la même étiquette que celle de t'_0. Montrer que :

$$\Phi(t') - \Phi(t) \leq 3(\text{rg}(t'_0) - \text{rg}(t_0))$$

(b) Dans cette question, considérons les transformations de la question 4.a plutôt qu'une rotation. Montrer que :

$$\Phi(t') - \Phi(t) \leq 3(\text{rg}(t'_0) - \text{rg}(t_0)) - k$$

Où k est une constante entière qui dépend de la transformation et qu'il faudra calculer.

On pourra utiliser, en l'admettant, l'*inégalité arithmético-géométrique* :

$$\sqrt{ab} \leq \frac{a+b}{2}$$

quels que soient a et b deux réels positifs ou nuls.

(c) En déduire la complexité en temps de la variante de l'algorithme de la question 4.b, pour une séquence de requêtes.

Suite des questions

Question 6. Supposons maintenant que chaque étiquette x a un poids $w(x)$. Soit W la somme des poids de toutes les étiquettes de l'arbre. Montrer que la complexité en temps amortie de la question 5 devient $O\left(\log \frac{W}{w(x)}\right)$ lorsque l'on cherche l'étiquette x. Commenter.

Question 7. Proposer des fonctions d'insertion et de suppression similaires à la fonction de recherche de la question 4. Quelles sont leurs complexités amorties ?

Corrigé

Question 0. Un arbre binaire de recherche est un arbre binaire étiqueté par des éléments d'un ensemble totalement ordonné. Il vérifie la propriété suivante : si l'arbre contient un nœud $\texttt{Nœud}(g, x, d)$ (de sous-arbres gauche et droit g et d, et d'étiquette x), alors toutes les étiquettes des nœuds de g sont plus petites que x, et toutes les étiquettes des nœuds de d sont plus grandes que x. Dans le contexte de cette épreuve, on ignore le cas où plusieurs nœuds pourraient avoir la même étiquette (c'est impossible pour les dictionnaires).

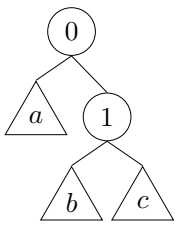
Les arbres binaires de recherche peuvent être utilisés pour implémenter une structure de donnée de dictionnaire, lorsque l'ensemble des clefs est totalement ordonné : chaque nœud de l'arbre est étiqueté par une entrée du dictionnaire, en ordonnant les entrées par l'ordre des clefs. Lorsque l'arbre n'est pas déséquilibré, les requêtes d'insertion, de suppression, et de lecture d'un nœud sont rapides, et permettent donc d'implémenter efficacement les requêtes correspondantes sur le dictionnaire.

La fonction d'insertion dans un arbre binaire de recherche peut se pseudo-coder comme suit :

```
Insérer(x, a) :=
  Si a = Vide
    Renvoyer Nœud(Vide, x, Vide)
  Si a = Nœud(g, y, d)
    Si x < y
      Renvoyer Nœud(Insérer(x, g), y, d)
    Si x > y
      Renvoyer Nœud(g, y, Insérer(x, d))
    Si x == y
        (* Dans le cas d'un dictionnaire, on peut imaginer que la fonction
           Insérer est utilisée pour modifier une entrée déjà existante. *)
      Renvoyer Nœud(g, x, d)
```

Sa complexité est asymptotiquement la profondeur du nœud inséré. Dans le pire cas, si l'arbre est complètement déséquilibré (c'est un "peigne"), la complexité est linéaire en fonction du nombre de nœuds de l'arbre. Par contre, si l'arbre est bien équilibré, la complexité est logarithmique, puisque la profondeur de l'arbre est logarithmique par rapport au nombre de nœuds.

Question 1. Après la rotation à droite, l'arbre est modifié comme suit :



Il s'agit, en fait, de la seule transformation possible qui ne modifie pas les sous-arbres a, b et c.

Cette opération est implémentée par le pseudo-code suivant :

```
Rotation_droite(n)
  Posons Nœud(Nœud(a, e0, b), e1, c) = n
  Renvoyer Nœud(a, e0, Nœud(b, e1, c))
```

On pourrait aussi donner un pseudo-code pour la rotation gauche, de manière symétrique.

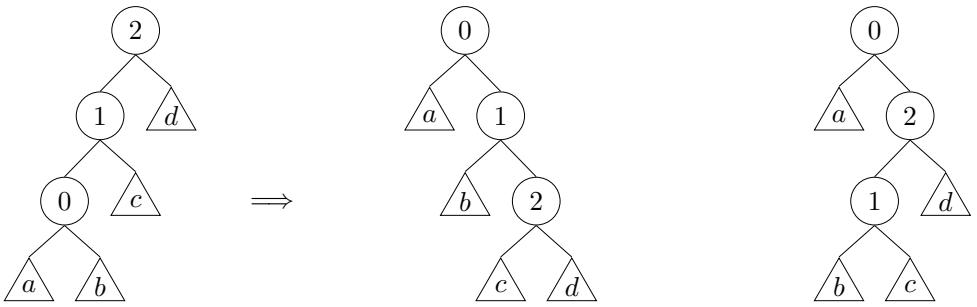
Question 2. L'algorithme procède comme pour une recherche dans un arbre binaire de recherche, mais effectue une rotation (droite ou gauche) après chaque appel récursif afin de placer à la racine le nœud demandé :

```
Recherche_et_évase(n, e)
  (* Puisqu'on suppose que l'étiquette e est dans l'arbre, n ne peut être vide *)
  Posons Nœud(g, e', d) = n
  Si e = e'
    Renvoyer n
  Si e < e'
    Renvoyer Rotation_droite(Nœud(Recherche_et_évase(g, e), e', d))
  Si e > e'
    Renvoyer Rotation_gauche(Nœud(g, e', Recherche_et_évase(d, e)))
```

Sa complexité est asymptotiquement la hauteur du nœud recherché. Dans le pire cas, si l'arbre est déséquilibré et que le nœud recherché est une feuille, la complexité est linéaire en la taille de l'arbre.

Question 3. L'idée est d'utiliser l'algorithme de la question 2 afin de répondre à la requête de recherche, mais aussi de modifier l'arbre binaire de recherche pour que le nœud qui vient d'être cherché se trouve à la racine. Ainsi, les nœuds fréquemment cherchés se trouvent proches de la racine et sont donc trouvés plus rapidement. Il s'agit là de l'idée proposée par Brian Allen et Ian Munro en 1978 [AM78].

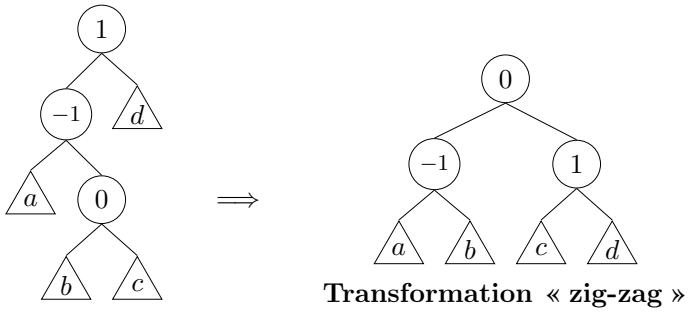
Question 4. Le premier arbre peut se transformer de deux façons différentes :



Transformation « zig-zig »

La deuxième transformation ci-dessus est celle qu'effectuerait l'algorithme de la Question 3. Par contre, la première est nouvelle : elle peut être obtenue en effectuant les deux rotations en procédant du haut vers le bas. Par la suite, on appellera cette transformation cette transformation *« zig-zig »*.

Le deuxième arbre, par contre, ne peut se transformer que de la façon suivante :



Transformation « zig-zag »

On appellera cette transformation *« zig-zag »*.

Bien sûr, 3 autres transformation symétriques existent pour des arbres dont la forme est symétrique aux deux arbres donnés dans l'énoncé. De manière similaire, on obtient alors les transformations *« zag-zig »* et *« zag-zag »*. On peut se alors se convaincre facilement qu'on a traité tous les cas possibles pour remonter un nœud de deux étages.

Il est à noter que, pour simplifier l'implémentation, toutes ces transformations peuvent s'obtenir en composant des rotations.

Pour obtenir des variantes de l'algorithme de la question 3, on peut donc utiliser une stratégie différente pour remonter le nœud en question à la racine. Plutôt que d'effectuer les rotations une par une de bas en haut, on peut les effectuer deux par deux, toujours de bas en haut, en appliquant les transformations *« zig-zig »*, *« zig-zag »*, *« zag-zig »* ou *« zag-zag »*, selon le cas pertinent. S'il y a un nombre impair de niveaux à remonter, il faut compléter ces transformations par une rotation gauche ou droite, à la racine par exemple.

On obtient alors l'opération d'évasement décrite par Sleator et al. [ST85]. Elle n'est pas équivalente à l'algorithme présenté dans la question 3, puisque, comme nous l'avons déjà évoqué, les transformations « zig-zig » et « zag-zag » n'effectuent pas les rotations de bas en haut comme dans la question 3. Comme nous allons le voir, cette opération d'évasement permet d'obtenir des garanties fortes de complexité amortie.

Question 5. Dans cette question, on utilisera les étiquettes de nœuds choisies dans les schémas des questions précédentes. Le sous-arbre t_0 correspond au sous-arbre dont la racine est le nœud 0, t_1 a le nœud 1 à la racine, etc...

Considérons d'abord le cas d'une rotation. La variation de potentiel n'est due qu'à la variation des rangs des sous-arbres enracinés en 0 et 1, puisque tous les autres sous-arbres gardent la même taille, et donc le même rang. On a :

$$\begin{aligned}
\Phi(t') - \Phi(t) &= \mathrm{rg}(t'_1) + \mathrm{rg}(t'_0) - \mathrm{rg}(t_1) - \mathrm{rg}(t_0) \\
&= \mathrm{rg}(t'_1) - \mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_0) = \mathrm{rg}(t_1) \\
&\leq \mathrm{rg}(t'_0) - \mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_1) \leq \mathrm{rg}(t'_0) \\
&\leq 3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0)) && \text{car } \mathrm{rg}(t'_0) = \mathrm{rg}(t_1) \geq \mathrm{rg}(t_0)
\end{aligned}$$

Considérons maintenant la transformation « zig-zig » de la question 4.

Indication : utiliser l'inégalité arithmético-géométrique pour majorer $(\mathrm{rg}(t'_2) + \mathrm{rg}(t_0))/2$.
On a :

$$\begin{aligned}
\Phi(t') - \Phi(t) &= \mathrm{rg}(t'_2) + \mathrm{rg}(t'_1) + \mathrm{rg}(t'_0) - \mathrm{rg}(t_2) - \mathrm{rg}(t_1) - \mathrm{rg}(t_0) \\
&= \mathrm{rg}(t'_2) + \mathrm{rg}(t'_1) - \mathrm{rg}(t_1) - \mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_0) = \mathrm{rg}(t_2)
\end{aligned}$$

Que faire de ces 4 termes ?

— On peut utiliser l'inégalité arithmético-géométrique pour $\mathrm{rg}(t'_2)$:

$$\begin{aligned}
\frac{\mathrm{rg}(t'_2)+\mathrm{rg}(t_0)}{2} &= \log_2\left(\sqrt{\mathrm{taille}(t'_2)\cdot\mathrm{taille}(t_0)}\right) && \text{par définition}\\
&\leq \log_2\left(\frac{\mathrm{taille}(t'_2)+\mathrm{taille}(t_0)}{2}\right) && \text{inégalité arithmético-géométrique}\\
&\leq \log_2(\mathrm{taille}(t'_0))-1 && \text{en remarquant que } \mathrm{taille}(t'_2)+\mathrm{taille}(t_0)\leq\mathrm{taille}(t'_0)
\end{aligned}$$

En réordonnant les termes, on obtient $\mathrm{rg}(t'_2) \leq 2\mathrm{rg}(t'_0) - \mathrm{rg}(t_0) - 2$.

— On a facilement $\mathrm{rg}(t'_1) \leq \mathrm{rg}(t'_0)$.

— De manière similaire, $-\mathrm{rg}(t_1) \leq -\mathrm{rg}(t_0)$.

— On conserve $-\mathrm{rg}(t_0)$.

En combinant toutes ces inégalités, on obtient $\Phi(t') - \Phi(t) \leq 3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0)) - 2$.

On considère maintenant le cas de la transformation « zig-zag ».

Indication : utiliser l'inégalité arithmético-géométrique pour majorer $(\mathrm{rg}(t'_{-1}) + \mathrm{rg}(t'_1))/2$.

De manière similaire à la transformation « zig-zig », on a :

$$\begin{aligned}
\Phi(t')-\Phi(t) &= \mathrm{rg}(t'_1)+\mathrm{rg}(t'_{-1})+\mathrm{rg}(t'_0)-\mathrm{rg}(t_1)-\mathrm{rg}(t_{-1})-\mathrm{rg}(t_0)\\
&= \mathrm{rg}(t'_1)+\mathrm{rg}(t'_{-1})-\mathrm{rg}(t_{-1})-\mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_0)=\mathrm{rg}(t_1)\\
&\leq 2\mathrm{rg}(t'_0)-2-\mathrm{rg}(t_{-1})-\mathrm{rg}(t_0) && \text{par inégalité arithmético-géométrique}\\
&\leq 2(\mathrm{rg}(t'_0)-\mathrm{rg}(t_0))-2 && \text{car } \mathrm{rg}(t_{-1})\geq\mathrm{rg}(t_0)\\
&\leq 3(\mathrm{rg}(t'_0)-\mathrm{rg}(t_0))-2 && \text{car } \mathrm{rg}(t'_0)=\mathrm{rg}(t_1)\geq\mathrm{rg}(t_0)
\end{aligned}$$

Pour la dernière transformation évoquée à la question 4 (à laquelle nous n'avons pas donné de nom), on peut simplement se rappeler qu'elle s'obtient comme composition de deux rotations sur le nœud 0.

Indication : réutiliser le résultat sur les rotations. Ne pas passer beaucoup de temps sur cette transformation inutile.

En utilisant deux fois l'inégalité obtenue pour les rotations, on remarque qu'elles se télescopent, et on obtient :

$$\Phi(t') - \Phi(t) \leq \mathrm{rg}(t'_0) - \mathrm{rg}(t_0) \leq 3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0))$$

La complexité d'une séquence de requêtes à l'algorithme de la question 4.a peut être évaluée en majorant la variation de potentiel d'une requête.

Pendant une telle requête, on fait des transformations dont on a déjà majoré la variation de potentiel avec les inégalités ci-dessus. Ces transformations sont de deux types :

— Les transformation « zig-zig », « zig-zag », « zag-zig » et « zag-zag » permettent de remonter le nœud d'intérêt de deux niveaux. La variation de potentiel associée est majorée par $3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0)) - 2$ (on a établi le résultat ci-dessus pour deux de ces transformations ; les deux autres sont symétriques). Si le nœud d'intérêt a initialement une profondeur de p, alors on utilise $\left\lfloor \frac{p}{2} \right\rfloor$ telles transformations.

— Éventuellement, si la profondeur du nœud est impaire, au plus une rotation (à gauche ou à droite) permet de finir le travail. Dans ce dernier cas, la variation de potentiel due à cette transformation est majorée par $3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0))$.

On peut alors combiner les majorations obtenues, qui se télescopent, et obtenir :

$$\Phi(t') - \Phi(t) \leq 3(\mathrm{rg}(t') - \mathrm{rg}(t_0)) - 2\left\lfloor \frac{p}{2} \right\rfloor \leq 3\log_2 N - p + 1$$

en notant :

— t et t' les arbres avant et après une requête, respectivement ;

— t_0 le sous-arbre de t dont la racine est le nœud recherché ;

— p la profondeur du nœud recherché dans t ;

— N la taille de t (ou de t').

Ainsi, pour une séquence de M requêtes de profondeurs $p_1, \ldots, p_M$, on peut utiliser l'inégalité ci-dessus et télescoper les valeurs de Φ intermédiaires. On obtient :

$$\Phi(t') - \Phi(t) \leq M(3\log_2 N + 1) - \sum_{i=1}^{M} p_i$$

D'où :

$$\sum_{i=1}^{M} p_i \leq M(3\log_2 N + 1) - \Phi(t') + \Phi(t) = O(M\log N) + O(N\log N)$$

Or, $\sum_{i=1}^{M} p_i$ est asymptotiquement le temps d'exécution des M requêtes. En particulier, si on considère un grand nombre de requêtes, le terme $O(N\log N)$ devient négligeable. Donc, en complexité amortie, le coût d'une requête est $O(\log N)$.

Question 6. On modifie les notions de rang et de potentiel pour utiliser les poids donnés aux étiquettes. En particulier, on définit le rang d'un sous-arbre comme étant le logarithme de la somme des poids des étiquettes qu'il contient :

$$\mathrm{rg}(t) = \sum_{x \in t} w(x)$$

On note W la somme de tous les poids de l'arbre.

De la même façon qu'à la question 5, on obtient l'inégalité suivante pour la variation du potentiel due à une requête de recherche de l'étiquette x :

$$\Phi(t') - \Phi(t) \leq 3(\mathrm{rg}(t') - \mathrm{rg}(t_0)) - p + 1 \leq 3(\log_2 W - \log_2 w(x)) - p + 1$$

Puis, pour une séquence de requêtes des étiquettes $x_1, \ldots, x_M$ aux profondeurs $p_1, \ldots, p_m$:

$$\Phi(t') - \Phi(t) \leq M + 3\sum_{i=1}^{M} \log_2 \frac{W}{w(x_i)} - \sum_{i=1}^{M} p_i$$

Donc le coût d'une séquence de M requêtes est asymptotiquement :

$$O\left(M + \sum_{i=1}^{M} \log_2 \frac{W}{w(x_i)}\right) + O\left(\sum_{x \in t} \log_2 \frac{W}{w(x)}\right)$$

Pour un grand nombre de requêtes, le second terme devient négligeable. Ainsi, le coût amorti d'une requête pour l'étiquette x est $O\left(\log_2 \frac{W}{w(x)}\right)$.

On peut voir ainsi voir que l'arbre s'adapte de manière automatique à la distribution des requêtes : si une requête est plus fréquente, alors son coût sera moins important. En pratique, si un nœud est accédé plus fréquemment, les transformations de l'arbre le déplaceront régulièrement en haut de l'arbre, ce qui rendra son accès plus rapide.

Question 7. Il y a plusieurs solutions pour l'insertion et la suppression. Voici une possibilité.

Pour l'insertion, on peut commencer par remonter à la racine le nœud immédiatement inférieur au nœud à insérer en coût amorti $O(\log N)$, puis il est facile d'insérer le nouveau nœud à la racine en temps constant. La variation de potentiel due à cette dernière opération est de $O(\log N)$, donc le coût total de l'opération est $O(\log N)$.

Évidemment, cette opération ne peut se faire si aucun nœud n'est inférieur au nœud à insérer. Mais dans ce cas, le problème est trivial : on peut simplement insérer le nouveau nœud à la racine.

Pour la suppression, on peut commencer par remonter à la racine le nœud à supprimer, puis le supprimer. La première opération peut s'effectuer en temps amorti $O(\log N)$. La suppression seule peut se faire en temps constant. La variation de potentiel étant négative, on peut ne pas la comptabiliser.

Il reste alors à fusionner les deux sous-arbres restants, dont on sait que les clefs de l'un sont toutes inférieures aux clefs de l'autre. Si l'un des deux arbres est vide, c'est trivial et immédiat. Sinon, un moyen simple pour effectuer cette fusion est de monter à la racine du sous-arbre inférieur son nœud minimal, en temps amorti $O(\log N)$. Après cet opération, la racine de l'arbre inférieur ne peut pas avoir de fils droit. On peut donc y placer l'autre sous-arbre. La variation de potentiel due à cette dernière opération (ajouter le sous-arbre supérieur comme fils droit de la racine de l'autre) peut se faire en temps constant, et la variation de potentiel associée n'est due qu'à la variation de rang de la racine, qui est $O(\log N)$. Au total, la suppression peut se faire en temps amorti $O(\log N)$.

Références

[AM78] Brian Allen and Ian Munro. Self-organizing binary search trees. *Journal of the ACM (JACM)*, 25(4) :526–535, 1978. Non disponible en libre accès.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3) :652–686, 1985. `https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`.

J2 – Paresse et file persistante

Question 0.

(a) Qu'est-ce qu'une file ?

(b) Rappeler la distinction entre structure de donnée persistante et impérative.

(c) Donner une implémentation *persistante* d'une file.

(d) En déduire une implémentation *impérative* d'une file.

(e) Dans le pire cas, quelle est la complexité de chacune des opérations des ces deux implémentations ?

Lorsque l'on analyse la *complexité amortie* d'une bibliothèque, on s'intéresse à la complexité d'une séquence d'opérations dans son ensemble plutôt qu'à la complexité de chaque opération fournie par la bibliothèque. Ainsi, même si une opération A est très coûteuse, son coût peut être compensé par l'exécution préalable d'un grand nombre d'opérations B, de façon à ce que la complexité globale de la séquence d'opérations soit asymptotiquement le même que si A était peu coûteuse.

Question 1.

(a) Faire l'analyse de complexité amortie de l'implémentation impérative de la question 0.

(b) Ce raisonnement peut-il s'appliquer pour l'implémentation persistante ?

On se propose d'implémenter, en OCaml, une bibliothèque de *calcul paresseux*. Celle-ci expose un type paramétré de *suspensions* `'a susp` et des fonctions de types suivants :

```
susp : (unit -> 'a) -> 'a susp
force : 'a susp -> 'a
```

Une suspension (de type `'a susp`) contient une fonction permettant de calculer une valeur de type `'a`. Elle peut être construite facilement grâce à la fonction `susp`. Le calcul n'est effectué que lorsque l'utilisateur de la bibliothèque le demande via la fonction `force`. Cette dernière fonction vérifie si le calcul a déjà été effectué : si tel est le cas, elle en renvoie le résultat pré-calculé. Sinon, elle lance le calcul, stocke le résultat pour de futurs appels, et elle le renvoie.

Question 2. Donner une implémentation possible de cette bibliothèque, dans le langage OCaml.

On définit le type des *listes paresseuses* en OCaml :

```
type 'a slist_cell =
| SNil
| SCons of 'a * 'a slist
and 'a slist = 'a slist_cell susp
```

Question 3.

(a) Comparer la notion de liste paresseuse avec la notion habituelle de liste.

(b) Écrire une fonction `scons : 'a -> 'a slist -> 'a slist` qui ajoute un élément en tête d'une liste paresseuse, ainsi qu'une valeur `snil` de liste paresseuse vide.

(c) Écrire deux fonctions `shd : 'a slist -> 'a` et `stl : 'a slist -> 'a slist` qui prennent une liste paresseuse non vide en paramètre, et qui renvoient respectivement son premier élément et la liste paresseuse des autres éléments.

(d) Écrire une fonction `sappend : 'a slist -> 'a slist -> 'a slist` qui concatène *de manière paresseuse* deux listes paresseuses. Cette fonction devra s'exécuter en temps constant.

(e) Écrire une fonction `srev : 'a slist -> 'a slist` qui renverse une liste paresseuse de manière efficace. Quelle est la complexité des accès aux différents éléments de la liste renversée ?

Grâce aux listes paresseuses, on propose ici une variante de la file proposée dans la question 0.c, qui évite le problème de complexité expliqué dans la question 0.e. Dans cette nouvelle implémentation, une file sera représenté en OCaml par le type enregistrement suivant :

```
type 'a queue = {
  rear : 'a slist;
  len_rear : int;
  front : 'a slist;
  len_front : int;
}
```

Le plus souvent, on enfilera les éléments au début de la liste `rear` et on les défilera au début de la liste `front`. De plus, on maintiendra les invariants suivants :

— les champs `len_front` et `len_rear` contiennent les longueurs des listes `front` et `rear` ;

— on a toujours `len_rear` $\leq$ `len_front`.

Question 4.

(a) Définir les fonctions d'enfilage et de défilage pour cette variante d'implémentation de file.

(b) Prouver que le temps d'exécution *amorti* de chacune de ces deux opérations est $O(1)$.

Suite des questions

Question 5. Une liste paresseuse est-elle toujours de longueur finie ? Définir en OCaml une liste paresseuse qui énumère les carrés parfaits.

Corrigé

Question 0. (a) Une file est une structure de donnée représentant une séquence d'éléments, et qui permet d'ajouter un élément au début de la séquence, et de récupérer en supprimant un élément à la fin de la séquence.

(b) Dans une structure de donnée impérative, les fonctions modifient la structure de donnée en mémoire, alors qu'une structure de donnée persistante renvoie une nouvelle version de la structure de donnée sans modifier (de manière observable) celle qui a été donnée en paramètre.

(c) On peut implémenter de manière persistante une file avec deux listes. On enfile toujours au début de la première liste, et on défile au début de la seconde. Si la seconde file est vide lorsque l'on veut défiler, alors on renverse la première liste pour lui faire prendre la place de la seconde (dans ce code, on a mis toutes les annotations de type, mais elles sont bien sûr optionnelles) :

```
type 'a queue = 'a list * 'a list

let vide : 'a queue = ([], [])

let push (q : 'a queue) (x : 'a) : 'a queue =
  (x::fst q, snd q)

let pop (q : 'a queue) : 'a * 'a queue =
  match snd q with
  | x::q2 -> (x, (fst q2, q2))
  | [] ->
    match List.rev (fst q) with
    | x::q2 -> (x, ([], q2))
    | [] -> assert false  (* La file est vide, erreur *)
```

(d) On peut simplement mettre la file persistante dans une référence. On obtient alors une file impérative :

```
type 'a queue' = 'a queue ref

let vide' () : 'a queue' =
  ref vide

let push' (q : 'a queue') (x : 'a) : unit =
  q := push !q x

let pop' (q : 'a queue') : 'a =
  let (x, qp) = pop !q in
  q := qp;
  x
```

(e) La création d'une nouvelle file est toujours en temps constant. L'opération `push` est aussi en temps constant. Par contre, l'opération `pop` est, au pire cas, en temps $O(N)$, où N est le nombre d'éléments dans la file. En effet, s'il faut renverser la première file, cela prend un temps $O(N)$.

Question 1. (a) Dans une séquence d'opérations `push` et `pop` sur une file, le retournement de liste ne traitera un élément donné qu'une seule fois. Ainsi, le coût global des retournements ne dépassera pas asymptotiquement le nombre d'éléments enfilés. Si on attribue le coût des retournements à l'enfilage plutôt qu'au défilage, la complexité amortie des opérations `push` et `pop` est donc $O(1)$, *dans l'implémentation impérative.*

(b) Dans le cas de l'implémentation persistante, il n'est plus vrai que chaque élément enfilé n'est traité qu'une seule fois par le retournement. En effet, il est très facile de copier une file persistante. Ainsi, si on commence par enfiler N éléments, puis qu'on effectue M copies, et que pour chacune de ces copies, on effectue un défilage, alors on va renverser une liste de taille N une fois pour chaque copie (soit M fois au total). La complexité globale serait alors $O(NM)$ alors qu'on a effectué que $O(N)$ enfilages et $O(M)$ défilages.

Question 2. Une suspension peut avoir deux états : en attente de calcul ou calculé. Cet état est modifié de manière impérative lors du premier appel à `force` la concernant. Il est donc naturel d'utiliser un type somme à deux alternatives pour représenter une suspension. De plus, pour permettre le changement d'état, on va utiliser une référence :

```
type 'a susp_state =
| Computed of 'a
| Suspended of (unit -> 'a)

type 'a susp = 'a susp_state ref
```

Pour la fonction `susp_state`, on utilise simplement le constructeur `Suspended` dans une référence fraîche :

```
let susp f = ref (Suspended f)
```

Enfin, la fonction `force` peut être implémentée facilement en distinguant les deux cas :

```
let force p =
  match !p with
  | Computed x -> x
  | Suspended f ->
    let x = f () in
    p := Computed x;
    x
```

Une autre possibilité est d'utiliser simplement une référence sur une fonction : après avoir évalué la suspension, on modifie la référence pour qu'elle contienne la fonction qui renvoie immédiatement la valeur déjà calculée. On obtient alors le code suivant :

```
type 'a susp = (unit -> 'a) ref

let susp f = ref f

let force p =
  let x = !p () in
  p := (fun () -> x);
  x
```

Question 3. (a) Dans la notion de liste habituelle, les éléments de la liste sont présents en mémoire dès que la liste existe. Au contraire, pour une liste paresseuse, ils sont calculés à la demande, lorsque l'on accède aux éléments de la liste. De manière intéressante, la structure de liste paresseuse peut être utilisée pour représenter des séquences infinies d'éléments, en ne stockant pas explicitement les éléments qui ne sont pas encore accédés : on stocke plutôt une fonction qui permettra, le moment venu, de calculer plus d'éléments de cette liste.

(b)

```
let snil = susp (fun () -> SNil)
let scons x l = susp (fun () -> SCons (x, l))
```

(c)

```
let shd x =
  match force x with
  | SCons (hd, _) -> hd
  | SNil -> assert false

let stl x =
  match force x with
  | SCons (_, tl) -> tl
  | SNil -> assert false
```

(d) Une possibilité serait d'utiliser une fonction récursive qui itère les éléments de la première liste afin de l'ajouter à la liste paresseuse un à un avec la fonction `scons` ci-dessus. Malheureusement, cela requiert d'accéder à tous les éléments de la première liste pendant l'appel à `sappend`. Ceci ne peut s'effectuer en temps constant.

On utilise donc une stratégie différente, qui accède à la première liste à la demande, lorsque l'accès aux éléments de la liste renvoyée est demandé :

```
let rec sappend l1 l2 =
  susp (fun () ->
    match force l1 with
    | SCons (t, q) -> SCons (t, sappend q l2)
    | SNil -> force l2)
```

Cette fonction termine bien en $O(1)$, puisque la fonction `susp` termine en $O(1)$.

(e) Comme pour une liste non paresseuse, on peut utiliser un algorithme inefficace qui ajoute, un par un, les éléments de la liste d'entrée à la fin de la liste résultat. Cependant, comme pour une liste non paresseuse, on peut faire plus efficace, en utilisant un accumulateur qui contient la liste partiellement renversée :

```
let srev l =
  let rec aux acc l =
    match force l with
    | SCons (t, q) -> aux (scons t acc) q
    | SNil -> acc
  in
  susp (fun () -> force (aux snil l))
```

Au contraire de la fonction `sappend`, la fonction `srev` a besoin de parcourir en entier la liste passée en paramètre, dès que le premier élément de la liste donnée en entrée est demandé. Donc :

— lorsque la fonction `srev` est appelée, son coût "direct" est $O(1)$, puisqu'il ne s'agit que d'un appel à `susp` ;

— lorsque le premier élément est demandé, le coût est le coût de l'évaluation de la liste d'entrée plus $O(N)$, où N est la longueur de la liste ;

— lorsque les éléments suivants sont demandés, leur coût est de $O(1)$, puisque la suspension correspondante dans le code est `scons t acc`.

Question 4. Il s'agit de la *file du banquier* proposée par Okasaki [Oka99, §3.4.2].

(a) Aussi bien lors du défilage que de l'enfilage, l'invariant `len_rear` $\leq$ `len_front` peut être violé par la modification de la seule liste concernée (`rear` pour l'enfilage, `front` pour le défilage). Pour rétablir cet invariant, il faut, comme dans le cas de la file présentée dans la question 0.c, renverser la liste `rear` pour la placer à la fin de la liste `front`. On commence par écrire une fonction qui effectue cette opération de rétablissement d'invariant, si nécessaire :

```
let refresh q =
  if q.len_rear <= q.len_front then q
  else
    { rear = snil;
      len_rear = 0;
      front = sappend q.front (srev q.rear);
      len_front = q.len_front + q.len_rear }
```

On peut alors facilement écrire les fonctions d'enfilage et de défilage :

```
let push q x =
  let q = { q with rear = scons x q.rear; len_rear = q.len_rear + 1 } in
  refresh q

let pop q =
  (shd q.front, refresh { q with front = stl q.front; len_front = q.len_front - 1 })
```

(b) Le piège dans l'analyse de complexité de cette implémentation serait d'oublier de compter le coût d'évaluation des suspensions. En effet, la fonction `refresh` termine toujours en temps $O(1)$ puisqu'elle ne fait que reporter le renversement et la concaténation des listes. Du coup, la fonction `push` termine aussi en temps constant, et, si on oubliait de compter l'évaluation de la suspension qui a lieu lors de l'appel à `shd`, la fonction `pop` terminerait aussi en temps $O(1)$ (rappelons-nous que l'appel à `stl` réutilise l'évaluation de la suspension de `shd`).

Donc, pour faire l'analyse de complexité amortie de cette implémentation, il faut garder trace des coûts des différentes suspensions présentes dans la structure de données (ou qu'il est prévu de calculer). Par exemple, le coût d'une suspension créée par `susp` est $O(1)$. Le coût de la suspension correspondant à la première cellule d'une liste de longueur N renvoyée par `srev` est $O(N)$ plus le coût de l'évaluation de toutes les cellules de la liste initiale. Le coût des suspensions suivantes est $O(1)$.

Enfin, si on concatène deux listes de tailles N et M, le coût des suspensions renvoyées par `sappend` sont :

— une constante plus le coût d'évaluation des suspensions correspondantes de la première liste pour les N premières suspensions,

— le coût d'évaluation des suspensions correspondantes de la seconde liste pour les M suivantes.

Indication 1 : Une remarque cruciale est qu'on peut payer le coût d'une suspension *à l'avance*, avant que celle-ci ne soit évaluée. Ceci repose sur le fait qu'une suspension n'est évaluée qu'une seule fois : si une suspension est copiée puis évaluée deux fois, son coût n'a besoin d'être payé qu'une seule fois.

Indication 2 : On peut aller plus loin que la remarque précédente : lorsqu'une suspension est une partie du résultat du calcul d'une autre suspension (typiquement dans le cas d'une liste paresseuse), on peut déplacer le coût de la suspension emboîtée à la suspension qui va la calculer. Plus précisément, s'il est prévu qu'une suspension A renvoie à la suite de son calcul une suspension B, alors on peut décider de payer (une partie) du coût de la suspension B en tant que coût de la suspension A. Ainsi, si A a un coût c_A et B un coût c_B, alors on peut considérer que A a un coût $c_A + x$ et B un coût $c_B - x$ pour tout $x \geq 0$. Bien sûr, cela repose aussi sur le fait qu'une suspension n'est évaluée qu'une seule fois : si A était évaluée deux fois, alors chaque calcul renverrait une suspension B différente, et on ne pourrait pas payer leur coût en avance dans A.

Ainsi, quitte à payer le coût $O(1)$ à l'avance au moment de l'appel des fonctions, on peut considérer que le coût de la suspension renvoyée par `scons` ou par `snil` est nul. Puisque `rear` n'est alimenté qu'avec `scons` et `snil`, on peut donc considérer que le coût des suspensions de la liste `rear` est nul.

Comment payer le coût des suspensions de la liste `front` ? On va prouver ici, que, quitte à déplacer les coûts des suspensions comme expliqué dans les remarques ci-dessus, le coût des `len_front - len_rear` premières suspensions de la liste `front` est $O(1)$, alors que les `len_rear` dernières suspensions sont gratuites. Il sera alors facile de conclure que `push` et `pop` sont en temps constant.

Pour prouver que cet invariant tient, on commence par remarquer qu'il est vrai trivialement lors de l'initialisation de la queue.

Lorsque l'on retire un élément avec `pop` sans rééquilibrer les listes, on paie simplement le coût $O(1)$ de la suspension que l'on évalue au début de la liste `front`. L'invariant est donc maintenu.

Lorsque l'on insère un élément avec `push` sans rééquilibrer les listes, il faut payer le coût $O(1)$ de la dernière suspension non gratuite (à la position `len_front - len_rear - 1`). Grâce aux deux remarques ci-dessus, on peut le faire par anticipation, même si cette suspension est au beau milieu de la liste `front`. On peut donc aussi maintenir l'invariant sur les coûts.

Finalement, lorsqu'un appel à `push` ou `pop` rééquilibre les deux listes, on a `len_rear = len_front` au début de l'opération. Du coup, au moment de l'appel à `refresh`, toutes les suspensions des deux listes sont gratuites. Après l'appel à `refresh`, les suspensions de la première moitié de la liste coûtent $O(1)$ (à cause de l'appel à `sappend`), la suspension suivante coûte $O($`len_front`$)$ (à cause de l'appel à `srev`), et les suspensions du reste de la liste coûtent toutes $O(1)$ (encore à cause de l'appel à `srev`). En répartissant sur les suspensions de la première moitié de la liste le coût $O($`len_front`$)$ de la suspension centrale (avec la deuxième remarque ci-dessus), on peut effectivement aboutir à un coût de $O(1)$ pour toutes les suspensions de la liste, comme le demande l'invariant.

Question 5. Puisqu'elle n'est jamais représentée directement en mémoire, une liste paresseuse n'est pas nécessairement de longueur finie. Elle peut représenter une séquence infinie d'éléments.

On peut définir la séquence des carrés parfaits de la façon suivante :

```
let carres =
  let rec aux i =
    susp (fun () -> SCons (i*i, aux (i + 1)))
  in
  aux 0
```

Références

[Oka99] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.