DS4 Informatique - MP2I
Correction

Q1 : Le type `'a set -> 'a` est déjà défini, il n'y a donc qu'à l'utiliser :

```
let retirer_arbitraire (s : 'a set) : 'a =
        let x = choisir s in
        retirer x s ;
        x
```

Q2 :

— On utilisera le type `'a queue` pour représenter un `carnet` cette stratégie. Comme on souhaite enfiler des jobs, on construire une file de type `job queue`.

— Le type `'a queue` représentant une pile ne convient pas : par construction d'une pile, l'élément sortant à un instant donné est le dernier à y être entré.

— Le type `'a set` représentant un ensemble ne convient pas : par construction du type, l'élément sortant à un instant donné est choisi arbitrairement parmi les éléments de l'ensemble.

Q3 :

1. Dans ce cas, la fonction `ecrire` dans le carnet correspond à la fonction `enfiler`.
2.
```
let rec choisir_et_effacer (date : int) (c : carnet) : job option =
        if est_vide_file c then
                None
        else
                let j = defiler c in
                if j.limite < date then
                        choisir_et_effacer date c
                else
                        Some j
```

Q4 : On considère 3 requêtes, qu'on présente dans le tableau suivant :

id	reception	limite
1	1	10
2	1	12
3	2	2

Avec cette configuration, les deux premières impressions seront imprimées les jours 1 et 2, et la troisième sera ignorée.

Q5 : On peut montrer par induction que, si t' est un sous-arbre non-vide de t, alors la racine de t' a une étiquette plus petite que tous ses descendants :

— si t' est une feuille, il n'y a rien à vérifier.

— sinon, t' a une racine de la forme $Noeud(g, x, d)$. Si l'un des deux sous-arbre est vide, il n'y a rien à vérifier pour celui-ci, car il ne contient aucun descendant de x. Si l'un des deux sous-arbres est non-vide, alors par la propriété locale des tas, x est plus petit que les étiquettes des racines de celui-ci. Et par hypothèse d'induction, ces étiquettes sont plus petites que celles de tous leurs descendants.
Par transitivité, on obtient que x est plus petit que les étiquettes de tous ses descendants.

En particulier, sa racine est d'étiquette plus petite que celles de tous les noeuds de t. Donc elle contient l'élément minimal.

Q6 :

1. Il suffit de comparer la valeur du champ `limite` : un job dont le champ `limite` est plus petit qu'un aute job doit être considéré comme prioritaire sur ce dernier.
2.
```
let plus_urgent (j1 : job) (j2 : job) : bool =
        j1.limite < j2. limite
```

Q7 :

```
let propriete_locale (c : carnet) (i : int) : bool =
    let res = ref true in
    if (2*i+1 < taille) && (plus_urgent c.tab.(2*i+1) c.tab.(i)) then
            res := false ;
    if (2*i+2 < taille) && (plus_urgent c.tab.(2*i+2) c.tab.(i)) then
            res := false ;
    !res
```

Q8 :

— dans l'algorithme `retirer_racine`, il est clair que toutes les instructions autres que l'appel à `tasser` termine.

— dans l'appel à tasser, on peut choisir comme variant la quantité i : il s'agit d'un entier qui croît strictement à chaque appel, mais qui est borné par la taille du tableau. Ainsi, il existera une itération de la boucle Tant Que dans `tasser` pour lequel l'algorithme s'arrêtera.

Q9 :

1. Par hypothèse, t forme un arbre complet en entrée de l'algorithme. Ses éléments sont donc représentés dans le tableau `t.tab` dans les cases de `0` à `t.taille-1`. Lorsqu'on diminue la taille du tableau, l'ensemble des éléments présents dans t correspondant maintenant aux cases de `0` à `t.taille-2`, qui forme un ensemble "consécutif" (sans trous) de données. Il s'agit donc bien toujours d'un arbre complet.
   De plus, la seule modification ayant été faite sur les éléments est la permutation de la racine avec l'élément à la position `t.taille-1` ; la diminution de la taille fait sortir du tas l'élément en position `t.taille-1`, c'est-à-dire l'ancienne racine.
2. L'appel à `tasser` n'effectue que des permutations d'éléments dans le tableau `t.tab` entre les cases `0` à `t.taille-1` (pour la nouvelle valeur de `t.taille`). Le tableau obtenu à la fin est donc une permutation du tableau initial sans la racine. La propriété est donc toujours vraie.

Q10 :

— Montrons que cette propriété est vraie avant la boucle de `tasser` : comme t étais un tas-min, la propriété locale était vérifiée pour tous les indices `i` de `0` à `t.taille-1` avant qu'on n'échange la racine avec le dernier élément.
Une fois la diminution de la taille et l'échange effectué, chaque élément du tableau admet au plus les mêmes descendants que précédemment (et vérifie donc la propriété locale), sauf la nouvelle racine.

— Supposons l'invariant vrai au début d'une itération. A la fin de l'itération concernée, chaque élément possède au plus les mêmes descendants qu'au début de l'itération (et vérifie donc la propriét élocale), sauf les éléments aux positions `i` et `j` qui ont été échangés.
`j` possède nécessairement une étiquette plus petite que l'autre "ancien fils" de `i` (s'il existe), puisqu'on l'a choisi comme étant le minimum des deux ; de même, `j` possède une étiquette plus petite que celle de `i`, puisqu'on les a échangés. Comme ces deux noeuds sont les seuls descendants possibles de `j`, alors `j` vérifie également la propriété locale.
Donc le noeud `i` est le seul qui peut ne pas la vérifier.

Q11 : L'invariant précédént stipule que le noeud `i` est le seul qui peut ne pas respecter la propriété locale. Or, lorsque l'appel à `tasser` s'arrête, cela signifie par construction que `i` la respecte. Donc tous les noeuds respectent la propriété locale après l'appel à tasser.
De plus, on a toujours un arbre binaire complet d'après ce qui précède, et qui contient les mêmes étiquettes que celles de t sauf son ancienne racine.
L'algorihme est donc correct.

Q12 : La complexité de `retirer_racine` est majorée par celle de `tasser` asymptotiquement. Or, ce dernier effectue au plus h itérations, où h est la hauteur de l'arbre t, et chaque itération s'effectue en temps constant.
Or, on peut montrer que h est de l'ordre de $\log(n(t))$, étant donné qu'il s'agit d'un arbre complet. Ceci montre le résultat.

Q13 : Si on veut garder un arbre binaire complet, il faut ajouter un noeud à la première position disponible dans le tableau : $k =$ `t.taille`. Si on l'ajoute à une position ultérieure, l'ensemble des données de l'arbre binaire ne forme plus un ensemble "consécutif" dans le tableau (il y a des trous), et l'arbre n'est donc plus complet : un niveau peut ne pas être rempli, ou bien le dernier niveau peut être rempli sans que tous les noeuds ne soient tassés à gauche.

Q14 :

```
let tamiser (c : carnet) (k : int) : unit =
        let i = ref k in
        while i > 0 && not (propriete_locale ((i-1)/2) )do
                swap c i ((i-1)/2) ;
                i := ((i-1)/2)
        done
```

Q15 :

```
let ajouter_impression (c : carnet) (j : job) : unit =
        c.tab.(c.taille) <- j ;
        c.taille <- c.taille+1 ;
        tamiser c (c.taille - 1)
```

Q16 : Si on reçoit n e-mails dans l'année, dans le pire des cas, l'algorithme effectue n insertions et n retraits.

Q17 : On remplacera `enfiler` par `ajouter_impression` et `defiler` par `retirer_racine`.

On effectue au plus n appels à chacune de ces deux fonctions, et les autres opérations s'effectuent en temps constant. Or, chacune de ces opérations a une complexité en $O(\log(|c|)$, où $|c|$ est la taille du carnet.

Celle-ci étant majorée par n, la complexité est en $O(\log(n))$.

Q18 : Sur l'exemple donné, la demande 3 sera traitée avant. On n'a donc plus de problème, et la stratégie est optimale sur cet exemple.

La formulation de la question suivante est ambiguë, on mélange priorité et date limite ; mais augmenter la priorité d'une impression revient à diminuer sa date limite, et réciproquement. Nous interpréterons ici "diminuer la priorité" comme "diminuer la date", et réciproquement.

Q19 :

— Si on diminue la priorité d'un noeud, alors seul son père peut ne plus respecter la propriété locale. Il suffit donc de tamiser comme précédemment.

— Si on augmente la priorité du noeud, il faut tasser à partir de ce noeud, qui est le seul à potentiellement ne plus respecter la propriété locale.

—
```
let changer_priorite (c : carnet) (i : int) (date : int) : unit =
        let ancien_job = c.tab.(i) in
        let nouveau_job = {
                reception = ancien_job.reception ;
                id = ancien_job.id ;
                limite = date
        } in
        if plus_urgent nouveau_job ancien_job then
                tamiser c i
        else
                tasser c i
```

Q20 : Pour annuler une impression en envoyant un e-mail au jour j, il suffit de changer sa date limite en $j-1$.

Q21 :

```
let ecrire (j : job) (c : carnet) : unit =
        let ecrit = ref false in
        for i = 0 to c.taille-1 do
                if c.tab.(i).id = j.id then
                        begin
                        changer_priorite c i j.limite
                        ecrit := true
                        end
        done ;
        if not (!ecrit) then
                ajouter_impression c j
```

Q22 : Comme on parcourt tout le tas à chaque écriture pour vérifier si le travail concerné n'existe pas déjà, l'ajout coûte désormais $O(n)$. Comme on l'effectue dans le pire des cas n fois, la complexité totale est en $O(n^2)$, ce qui n'est pas satisfaisant comparé à la complexité précédente.

Q23 : Procédons par récurrence forte :

— si $n = 0$, on utilise l'arbre vide.

— sinon, soit x un élément dont le champ `prio` a la valeur maximale. Soit $E_g = \{x_j \mid x_j.clef < x_i.clef\}$ et $E_d = \{x_j \mid x_j.clef > x_i.clef\}$. Comme toutes les clés sont deux-à-deux distinctes, on a $E = E_g \uplus E_d \uplus \{x\}$.

De plus, $|E_g| < |E|$ et $|E_d| < |E|$. Donc par hypothèse de récurrence, il existe des arbres tas g et d représentant E_g et E_d. Il suffit ensuite de construire l'arbre-tas donc la racine est $Noeud(g, x_i, d)$.

Il respecte bien la structure de tas par construction de x_i, et d'arbre binaire de recherche par construction des ensembles E_g et E_d.

Q24 :

```
int recherche (treap t, int k){
        if(t == NULL){
                return -1;
        }
        if(t->clef == k){
                return t->valeur;
        }
        if(t->clef > k){
                return recherche(t->gauche, k);
        }
        return recherche(t->droit, k);
}
```

Q25 :

```
if ((*t)->clef == k){
        (*t)->valeur = v;
        prio_insertion = (*t)->prio;
}
if ((*t)->clef < k){
        prio_insertion = ajouter(&((*t)->droit),k,v);
}
if ((*t)->clef > k){
        prio_insertion = ajouter(&((*t)->gauche),k,v);
}
```

Q26 : Si on échange un noeud avec son père, on ne respecte plus la propriété d'arbre binaire de recherche sur les clefs.

Q27 : Avant et après rotation, les deux arbres obtenus fournissent le même parcours infixe. Or, on sait qu'un arbre est un arbre binaire de recherche si et seulement si son parcours infixe parcourt les noeuds dans l'ordre croissant des étiquettes. Effectuer une rotation conserve donc cet ordre, et le résultat est toujours un arbre binaire de recherche.

Q28 : Plutôt que d'échanger i avec son père, on peut effectuer une rotation si le père ne respecte pas la propriété locale : d'après ce qui précède, on conserve la propriété d'arbre binaire de recherche. Il suffit de continuer à effectuer des rotations pour faire remonter l'élément, jusqu'à ce que la propriété locale de tas soit respectée par ce noeud.

Q29 : On remarque que lors d'un appel à `ajouter`, on a effectué des appels récursifs qui ont déjà pu effectuer des rotations. Concrètement, on n'effectue donc qu'une rotation dans cette partie, et on l'effectue si et seulement si les deux conditions suivantes sont vraies :

— le noeud inséré est remonté jusqu'à l'enfant du noeud actuel.

— sa priorité est plus grande que celle du noeud actuel.

La deuxième condition est suffisante, puisqu'elle implique en fait la première si elle est vraie... Nous ne vérifierons donc que celle-ci. Pour conserver un côté pratique, nous présenterons en fait deux blocs de codes, et seul l'un d'entre eux sera effectué selon le cas, selon qu'on ait placé le noeud à droite ou à gauche du noeud courant.

Si on a inséré le noeud à droite, on fait alors une rotation gauche :

```
if(prio_insertion > (*t)->prio){
        treap t1 = (*t)->gauche ;
        treap t2 = (*t)->droit->gauche ;
        treap t3 = (*t)->droit->droit
        treap x = (*t)-> droit;
        treap y = *t;
        *t = x;
        x->gauche = y;
        x->droit = t3;
        y->gauche = t1;
        y->droit = t2;
}
```

Si on a inséré le noeud à gauche, on fait alors une rotation droite :

```
if(prio_insertion > (*t)->prio){
        treap t3 = (*t)->droit ;
        treap t2 = (*t)->gauche->droit ;
        treap t1 = (*t)->gauche->gauche
        treap y = (*t)-> gauche;
        treap x = *t;
        *t = y;
        x->gauche = t2;
        x->droit = t3;
        y->gauche = t1;
        y->droit = x;
}
```

Q30 :

1. Il faut faire des appels à `ajouter` pour garder à jour le dictionnaire. Lorsqu'on échange deux éléments, il suffit donc d'ajouter à nouveau ces deux éléments dans le dictionnaire, ce qui aura pour effet de modifier leur valeur (position dans le tas) tout en conservant la structure d'arbre-tas.
2. Un appel à ajouter s'effectue en $O(\log(k))$ pour les mêmes raisons que précédememnt. Ici, on en effectue 2 dans la fonction `swap`, la complexité est donc en $O(\log(k))$.

Q31 : D'après ce qui précède, une modification du carnet (ajout, retrait, ou modification de priorité) s'effectue maintenant en $O(\log(n)^2)$, car on y effectue $\log(n)$ appels à `swap`.

L'algorithme effectue toujours au plus n ajouts et n retraits, chacun ayant le coût précédemment indiqué. Au total, la complexité est donc bien en $O(n\log(n)^2)$.