DS4 : gestion d'un système de reprographie.

Durée : 4h

L'usage de tout matériel électronique dans le but d'accéder au cours, à internet ou de vérifier votre code est strictement interdit. Si vous remarquez ce qui vous semble être une erreur dans l'énoncé du sujet, vous indiquerez clairement sur votre copie les modifications que vous êtes amenées à considérer pour la suite des questions.

Vous penserez à rendre votre code le plus compréhensible possible, si besoin en y ajoutant des explications en dehors du programme. Vous pouvez avoir recours à des fonctions intermédiaires tant que vous en donnez une implémentation. Sauf demande explicite du sujet, il n'est pas nécessaire du justifier la complexité ou la correction du code que vous écrivez.

Introduction

Dans ce problème, nous nous intéressons à la gestion d'un système de reprographie d'un petit établissement scolaire.

Le problème est divisé en une description du problème, suivie de 4 parties contenant des questions :

- La première partie s'intéresse à une stratégie simple suivant la logique du "premier arrivé, premier servi".
- La seconde propose une autre stratégie plus efficace, intitulée "plus urgent d'abord".
- Dans une 3eme partie, on modifie l'algorithme afin de permettre aux professeurs de modifier leur demandes.
- Enfin, la dernière partie implémente une structure de dictionnaire dans le but d'améliorer le temps de calcul de la partie 3.

La partie 3 dépend de la partie 2. Les autres parties sont indépendantes, sauf pour quelques questions. Il est alors précisé quels résultats des autres parties peuvent servir.

Le langage de programmation des 3 premières parties est **OCaml**. Le langage de programmation de la dernière partie est **C**.

Description du problème

Les professeurs peuvent envoyer des requêtes à l'employé chargé de la reprographie en envoyant un e-mail contenant un fichier, et une date limite à laquelle le fichier doit être imprimé pour la classe. Il y a en tout **7 professeurs** dans cet établissement.

Le système de reprographie est cependant assez ancien, et ne permet de faire qu'**une impression par jour**. De plus, le service de reprographie n'est ouvert qu'en période scolaire, soit **180 jours par an**. Pour simplifier le problème, nous considérerons que les dates sont donc des entiers entre 0 et 179.

L'objectif du problème est de choisir quels fichiers imprimer au fur et à mesure de la réception des e-mails, afin de minimiser le nombre de fichiers non imprimés. A la fin du problème, on donne aussi la possibilité au professeurs de renvoyer un e-mail afin de modifier la date limite d'impression, ou bien d'annuler complètement une impression.

Routine de l'employé et format de données utilisées

Comme il n'est possible de faire qu'une impression par jour, l'employé de reprographie suit une routine assez simple, que l'on décrit ci-dessous :

- Tout d'abord, il lit les mails qu'il a reçu pour la journée. On considère que les professeurs envoient tous leur mail le matin avant cette opération.

- Puis, il note les informations relatives aux nouvelles impressions dans son carnet. Une demande d'impression sera caractérisée par sa date de réception, la date limite d'impression, et un identifiant unique la caractérisant. On ne s'intéressera pas à la manière dont est générée cet identifiant.
- Il choisit ensuite dans son carnet un document à imprimer, et lance l'impression. Bien sûr, la requête choisie doit avoir été demandée pour le jour même au plus tard afin qu'elle soit acceptée par le professeur. Pendant que la machine imprime, il prend sa pause déjeuner.
- Enfin, lorsque l'impression est terminée, il délivre les documents au professeur ayant fait la demande. Sa journée est alors terminée.

Nous représenterons une demande d'impression par le type enregistrement suivant :

```
type job {
  reception : int; (* date de réception, entre 0 et 179 *)
  id : int; (* nombre identifiant de maniere unique chaque document a imprimer *)
  limite : int; (* dernier jour auquel le document peut etre imprimé, entre 0 et 179 *)
}
```

On rappelle par ailleurs que le type `'a option` est défini en OCaml par

```
type 'a option =
  |None
  |Some of 'a
;;
```

Afin de simuler la réception des emails au jour le jour, on utilisera une liste de variables de type `job`, triées par date de réception croissante. On peut alors écrire un code à trous en OCaml simulant la routine de l'employé, où le type carnet représente une structure de données définie plus tard dans le problème :

```
let rec lire_emails (date : int) (emails : job list) (c : carnet) : job list =
  (*Lit les emails du jour, et renvoie les emails restant, pour les prochains jours *)
  match emails with
  |a::q when a.reception <= date ->
    ecrire a c;
    lire_emails q
  |_ -> emails (* tous les emails du jour ont été lus *)
;;

let rec routine (date : int) (emails : job list) (c : carnet) : unit =
  (* Effectue le travail d'une journée, et renvoie les emails a lire dans le futur *)

  if date < 180 then
    (* Lecture des emails, et inscription des données dans le carnet *)
    let prochains_emails : job list = lire_emails date emails c in

    (* Choix de l'impression du jour *)
    let impression_du_jour : job option = choisir_et_effacer date c in

    (* Impression du document *)
    imprimer impression_du_jour;

    (* Jour suivant*)
    routine (date+1) prochains_emails c
;;
```

La fonction `imprimer` n'a pas d'implémentation donnée, et il n'est pas demandée d'en inventer une. Elle sert uniquement à retracer la situation que l'on cherche à simuler dans le programme.

En ce qui concerne les fonction `ecrire` et `choisir_et_effacer`, on en proposera plusieurs implémentations dans la suite du problème, dépendant du type choisi pour `carnet`.

I Stratégie premier arrivé, premier servi

Dans un premier temps, l'employé de reprographie choisit d'imprimer les documents dans l'ordre où ils ont été reçus.

On se donne trois types *impératifs* pouvant représenter le carnet (dont le type sera noté `carnet`) :

- Une **pile** de type `'a stack` sur laquelle on dispose des opérations :
  - `empiler : 'a -> 'a stack -> unit` ajoutant un élément à la pile.

  - `depiler : 'a stack -> 'a` retirant un élément de la pile.
  - `est_vide_pile : 'a stack -> bool` indiquant si une pile est vide.
- Une **file** de type `'a queue` avec les opérations :
  - `enfiler : 'a -> 'a queue -> unit` ajoutant un élément à la file.
  - `defiler : 'a queue -> 'a` retirant un élément de la file.
  - `est_vide_file : 'a queue -> bool` indiquant si une file est vide.
- Un **ensemble** de type `'a set` muni des opérations suivantes :
  - `ajouter : 'a -> 'a set -> unit` ajoutant un élément à l'ensemble.
  - `retirer : 'a -> 'a set -> unit` retirant un élément de l'ensemble. Si l'élément n'est pas présent, l'ensemble n'est pas modifié.
  - `choisir : 'a set -> 'a` renvoyant un élément arbitraire de l'ensemble, sans le supprimer.
  - `est_vide_ensemble : 'a set -> bool` indiquant si un ensemble est vide.

Question 1 :

Ecrire une fonction `retirer_arbitraire : 'a set -> 'a` retirant un élément arbitraire d'un ensemble et le renvoyant.

Question 2 :

Un seul type permet de représenter la stratégie "premier arrivé, premier servi". Indiquer lequel, en précisant quel type remplacera le `'a` dans notre application. Pour les deux autres, expliquer pourquoi ils ne peuvent pas convenir.

Question 3 :

1. Quelle fonction doit remplacer la fonction `ecrire` de la routine si le carnet est représenté par le type abstrait choisi à la question précédente ?
2. Donner une implémentation de la fonction `choisir_et_effacer : int -> carnet -> job option` renvoyant l'impression de la journée à effectuer. On prendra bien garde à ne pas renvoyer une impression invalide car la date limite est dépassée.
   Le premier argument est la date du jour. Un document peut être imprimé le jour de sa date limite. On renverra None si aucun document ne peut être imprimé.

Question 4 :

Montrer que cette stratégie n'est pas optimale : on exhibera un exemple de requêtes pouvant toutes être satisfaites, mais ne l'étant pas en pratique avec la stratégie "premier arrivé, premier servi".
Indice : 3 demandes d'impressions reçues sur 2 jours différents suffisent à créer un contre-exemple.

II Stratégie optimale d'impression : le plus urgent d'abord

Le but de cette section est de développer une stratégie optimale d'impression, qui minimise le nombre d'impressions non effectuées. Nous allons pour cela imprimer en priorité les documents urgents, c'est-à-dire ayant une date limite d'impression minimale.

Nous représenterons donc le carnet par une file de priorité dans laquelle l'élément retiré est l'élément de priorité minimale.

Définition :

Un tas-min t est un arbre binaire non strict respectant :

- t est complet
- **(Propriété locale)** Si $u = Nœud(g, x, d)$ est un nœud de t, x est inférieur ou égal aux étiquettes des racines des sous-arbres g et d, si elles existent.

Un tas-min est analogue à la structure de tas-max vu en cours, à ceci près que la condition locale a été renversée pour avoir les plus petites étiquettes en haut à la place des plus grandes.

Question 5 :

Montrer que si t est un tas-min, sa racine contient l'étiquette minimale.

II.1 Type OCaml utilisé pour le carnet

Nous utiliserons la représentation usuelle des arbres complets par un tableau pour représenter un tas min. On rappelle qu'elle suit la convention suivante :

- Les nœuds de t sont représentés par un indice entre 0 et $n(t) - 1$ inclus.
- Dans le tableau, l'étiquette d'un nœud est stockée à l'indice lui correspondant.
- La racine est représentée par l'indice 0.
- Les enfants d'un nœud stocké à l'indice k sont stockés aux indices $2k + 1$ (enfant gauche) et $2k + 2$ (enfant droit).

Nous utiliserons donc le type suivant pour représenter le carnet. Il s'agit d'un tas-min dont les étiquettes sont de type `job` :

```
type carnet = {
  tab : job array;
  taille : int;
}
```

On met à disposition une fonction permettant d'échanger deux éléments du tableau d'un tas-min :

```
let swap (t : carnet) (i : int) (j : int) : unit =
  let temp = t.tab.(i) in
  t.tab.(i) <- t.tab.(i);
  t.tab.(j) <- temp
;;
```

Question 6 :

1. Comment comparer deux objets de type `job` afin que le plus urgent soit placé à la racine ?
2. Ecrire une fonction `plus_urgent : job -> job -> bool` indiquant si le premier argument est plus urgent que le second (i.e. si sa priorité est plus faible).

Question 7 :

Ecrire une fonction `propriete_locale : carnet -> int -> bool` vérifiant si le nœud représenté à l'indice i d'un carnet respecte la propriété locale de tas-min.

II.2 Retrait de l'élément le plus urgent du carnet

On propose les algorithme suivant permettant de retirer l'élément le plus urgent du carnet supposé non vide :

Algorithm 1: retirer_racine(t)

```
1 x ← t.tab.(0);
2 Echanger les nœuds stockés à l'indice 0 (racine) et t.tab.(t.taille -1) (feuille la plus à droite du dernier
    niveau);
3 Diminuer la taille de t de 1;
4 tasser(t, 0);
5 Renvoyer x;
```

Algorithm 2: tasser(t, i)

```
1 Tant que le nœud stocké à l'indice i ne respecte pas la propriété locale faire
2 |  j ← indice de l'enfant de i le plus urgent (i.e. ayant la priorité la plus faible);
3 |  Echanger les nœuds stockés aux indice i et j;
4 |  i ← j;
5 FinTantQue
```

Question 8 :

Montrer que l'algorithme `retirer_racine` termine.

Question 9 :

1. Montrer qu'après l'instruction de la ligne 3 de l'algorithme `retirer_racine`, t représente un arbre complet, et contient tous les éléments présents initialement dans le tas-min, sauf sa racine.
2. En déduire que cette propriété est aussi vrai à la fin de l'algorithme.

Question 10 :

Montrer que lors de l'appel à `tasser` utilisé par l'algorithme `retirer_racine`, l'invariant I suivant est vérifié : "Seul le nœud stocké à l'indice i peut ne pas respecter la propriété locale de tas-min."

Question 11 :

Déduire des question précédentes que l'algorithme `retirer_racine` est totalement correct.

Question 12 :

Montrer que la complexité en temps de l'algorithme `retirer_racine` est en $\mathcal{O}(\log(n(t)))$, où $n(t)$ est la taille de l'arbre t passé en argument.

II.3 Ajout d'une impression au carnet

On souhaite maintenant élaborer un algorithme permettant d'ajouter des éléments au carnet. Les fonctions `tamiser` et `ajouter_impression` devront s'exécuter en $\mathcal{O}(\log(n(t)))$ (avec $n(t)$ la taille de l'arbre t passé en argument) dans le pire des cas, mais il n'est pas demandé de justifier leur complexité.

Question 13 :

Montrer que l'ajout d'un nœud dans un tas-min t ne peut se faire qu'à un seul endroit si l'on veut garder un arbre complet.
On décrira cet endroit en indiquant l'indice k du tableau auquel l'ajout doit s'effectuer. On justifiera que l'ajout d'un nœud représenté par un autre indice ne conserve pas le caractère complet de l'arbre.

Question 14 :

En ajoutant un nœud représenté par l'indice k, la propriété locale n'est plus forcément vérifier pour le père de k. Ecrire une fonction `tamiser : carnet -> int -> unit` permettant de rétablir le tas en faisant remonter le nœud inséré.
L'entier passé en argument est l'indice k dont le père peut ne pas respecter la propriété locale. On rappelle que le père de ce nœud est stocké à l'indice `(k-1)/2`.

Question 15 :

En déduire une fonction `ajouter_impression : carnet -> job -> unit` ajoutant une impression au carnet.

II.4 Complexité de l'algorithme

Question 16 :

En utilisant un tas-min pour représenter le carnet, et en supposant que l'on reçoit n e-mails dans l'année, combien d'insertions dans le carnet et de retraits d'éléments du carnet sont effectués par l'algorithme dans le pire des cas ?

Question 17 :

En déduire le temps d'exécution dans le pire des cas de l'algorithme si l'on reçoit n e-mails dans l'année.
On pourra considérer que la fonction `imprimer` *s'exécute en temps constant, et qu'il y a plus de demandes que de jours dans l'année. On ne demande pas de réécrire les fonctions* `ecrire` *et* `choisir_et_effacer` *pour prendre en compte le nouveau type du carnet, mais on pourra indiquer succinctement comment les obtenir à partir de la question 3 afin de justifier le raisonnement.*

II.5 Optimalité de la solution

Question 18 :

Montrer que la stratégie "plus urgent d'abord" résout le problème exposé en question 4 avec la stratégie "premier arrivé, premier servi".

Remarque :

> On peut même montrer que la solution est optimale, c'est-à-dire qu'elle minimise le nombre d'impressions non réalisées. Cependant, le preuve de ce résultat est trop fastidieuse pour ce DS.

III Modifications de la date limite d'impression

On reprend la structure de tas-min permettant d'implémenter la stratégie optimale développée dans la partie II. On souhaite maintenant changer la priorité (i.e. la date limite pour imprimer le document) d'une demande d'impression.

III.1 Modification de la priorité en connaissant l'indice auquel est stocké la demande d'impression

Dans cette partie, on suppose que l'on sait à quel indice `i` une demande d'impression est stockée dans le tas-min de type `carnet`. On veut changer sa date limite d'impression.

On pourra utiliser dans cette sous-partie la fonction `tasser : carnet -> int -> unit` effectuant l'algorithme `tasser (t,i)`, ainsi que la fonction `tamiser : carnet -> int -> unit` développés en partie 2.2 et à la question 14 respectivement.

Question 19 :

1. Si on diminue la priorité, quel(s) nœud(s) ne respecte(nt) plus la propriété locale de tas ? En déduire une opération permettant de restaurer la structure après mise à jour de la priorité.
2. Même question si on augmente la priorité du nœud.
3. En déduire une fonction `changer_priorite : carnet -> int -> int -> unit` changeant la date limite d'impression d'un document. Le second argument correspond à `i` et le troisième à la nouvelle date limite.

Question 20 :

Comment un professeur peut-il annuler une impression en envoyant un e-mail au jour `j`, uniquement en modifiant la date limite d'impression ?

III.2 Modification de la routine pour autoriser les modifications de date limite

Question 21 :

Réécrire la fonction `ecrire : job -> carnet -> unit` permettant de noter les informations d'un e-mail dans le carnet, en utilisant la structure de tas-min pour le carnet, et en permettant des modifications de date limite aux professeurs.
On prendra bien garde à vérifier que l'e-mail ne concerne pas une demande existante avant de l'ajouter au carnet. Il faudra utiliser les fonctions `ajouter_impression` (question 15), `changer_priorite` (question 19).

Question 22 :

Quelle est la complexité en temps de la simulation dans le pire des cas si n e-mails sont reçus ? Est-ce satisfaisant par rapport à la complexité établie en partie II à la question 17 ?

IV Recherche efficace de la position de la demande dans le carnet

Afin d'avoir une recherche efficace de l'endroit où est stocké une demande d'impression dans le tas (dans le but de la modifier), nous allons utiliser un dictionnaire associant à un identifiant unique d'impression l'indice du tableau de la structure de tas auquel la demande a est stockée. Le dictionnaire sera représenté par un arbre-tas (ou *treap* en anglais), dont nous détaillons l'implémentation et la complexité dans les différentes sous-parties ci-dessous.

IV.1 Type associé à un arbre-tas et recherche d'une impression à effectuer

Nous utiliserons le type `treap` suivant pour représenter un arbre-tas :

```
type struct noeud {
  int clef; //identifiant unique d'impression
  int valeur; //valeur associée à la clef : position dans le tas-min du noeud correspondant.
  int prio; //nécessaire a la structure de treap
  struct noeud* gauche; //enfant gauche
  struct noeud* droit; //enfant droit
};

typedef struct noeud* treap;
```

Ce type correspond à la représentation d'un arbre binaire non strict en C. L'étiquette x d'un nœud interne comporte la donné des champs `clef`, `valeur`, et `prio`.

Un arbre-tas respecte les propriétés suivantes, sur chacun de ses nœuds de la forme $Noeud(g, x, d)$:

- (Propriété d'arbre binaire de recherche sur le champ `clef`) Soit k la clef contenue dans l'étiquette x. Alors k est strictement supérieure à toutes les clefs du sous arbre g, et strictement inférieure à toutes les clefs du sous arbre d.

- (Propriété locale de tas-max sur le champ `prio`) Soit p la valeur du champ `prio` contenue dans l'étiquette x. Alors p est supérieure ou égale aux valeurs des champs `prio` des nœuds g et d (si ce ne sont pas des arbres vides).

Question 23 :

Montrer que pour tout ensemble d'étiquettes $E = \{x_1, ...x_n\}$ dont les clefs sont deux à deux distinctes, il existe un arbre-tas à n nœuds dont les étiquettes sont $x_1, ..., x_n$.

Question 24 :

Ecrire une fonction `int recherche(treap t, int k)` recherchant une clef k dans le dictionnaire t.
Si k est associé à v dans le dictionnaire, on renverra v. Sinon, on renverra `-1`, qui est une valeur invalide dans notre application.
Votre fonction devra s'exécuter en $\mathcal{O}(h(t))$, où $h(t)$ est la hauteur de l'arbre-tas passé en argument. Il n'est pas demandé de justifier cette complexité.

IV.2 Ajout d'un élément dans l'arbre-tas

Afin d'ajouter un élément à l'arbre-tas, on va procéder en deux temps :

α. Tout d'abord, on insère l'élément de manière à respecter la propriété globale d'arbre binaire de recherche. Le champ `prio` est initialisé avec une valeur aléatoire.

β. Le champ `prio` du nouveau nœud peut être trop élevé pour que la propriété locale de tas-max soit respectée sur tous les nœuds. On va donc le faire remonter dans l'arbre jusqu'à satisfaire cette propriété.

On se basera sur le squelette suivant en C :

```c
int ajouter(treap* t, int k, int v){
  //insere un nouveau noeud, et renvoie la valeur aléatoire du champ prio qui lui a été associée.
  if(*t == NULL){
    //creation du noeud
    int p = rand()%1024; //valeur aléatoire du champ prio
    treap nouveau_noeud = (treap)malloc(sizeof(struct noeud));
    nouveau_noeud -> clef = k;
    nouveau_noeud -> valeur = v;
    nouveau_noeud -> prio = p;
    nouveau_noeud -> gauche = NULL;
    nouveau_noeud -> droite = NULL;
    *t = nouveau_noeud;
    return p;
  }

  int prio_insertion;

  /* Compléter ici : Q25 */

  /* Compléter ici : Q29 */

  return prio_insertion;
}
```

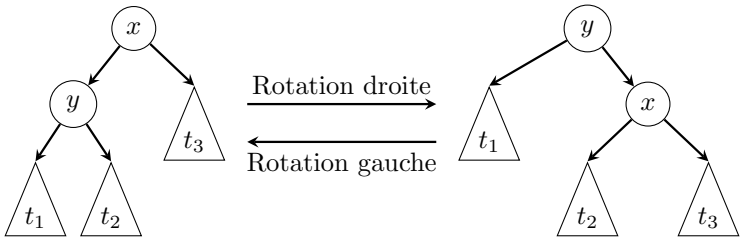
Question 25 :

Compléter la première partie du squelette pour effectuer l'étape α de l'insertion d'un nœud à l'aide d'un appel récursif à `ajouter`. On mettra la variable `prio_insertion` à jour avec la valeur du champ `prio` du nœud inséré.

Question 26 :

Pourquoi ne peut-on pas (de manière similaire aux tas) échanger le nœud que l'on vient d'insérer avec son père jusqu'à retrouver la propriété locale de tas-max ?

On appelle rotations les opérations décrites par le schéma suivant, dans lequel x et y sont des nœuds, et t_1, t_2, t_3 des sous arbres, potentiellement vides. On notera que x et y ne sont pas forcément une racine de l'arbre.



Question 27 :

Montrer que si t est un arbre respectant la propriété d'arbre binaire, et que l'on effectue une rotation (gauche ou droite) sur un nœud de t, le résultat de l'opération respecte aussi la propriété d'arbre binaire.

Question 28 :

Proposer un algorithme permettant de restaurer la structure d'arbre-tas à l'aide de rotations une fois que l'on a inséré une nouvelle feuille.

Question 29 :

Compléter la seconde partie du squelette pour permettre de restaurer la structure d'arbre tas.
Si cela n'a pas déjà été fait dans la question Q25, il sera utile de garder en mémoire dans quel sous-arbre t_1 le nouveau nœud a été créé. Puisque l'appel récursif effectue l'insertion et le rééquilibrage dans t_1, une partie des rotations nécessaires y a déjà été effectué.

Remarque :

> Les arbres tas sont (en moyenne) équilibrés, ce qui garantit que les opérations implémentées ci dessus s'exécutent en $\mathcal{O}(\log(n(t)))$ en moyenne. On peut de même définir une opération de suppression d'élément dans un arbre-tas ayant la même complexité.

IV.3 Complexité de l'algorithme final

On modifie l'algorithme afin de maintenir le dictionnaire en plus du carnet. On considère de plus que l'on ne retire jamais d'élément du dictionnaire (bien que ce soit techniquement faisable avec une bonne complexité).

Question 30 :

1. Comment peut-on modifier la fonction `swap` donnée dans l'ennoncé de la première partie afin de maintenir le dictionnaire ?
2. En déduire la complexité dans le pire des cas de la nouvelle fonction `swap` en fonction du nombre k d'e-mails lues avant son appel.

Question 31 :

Montrer que la complexité temporelle de l'algorithme dans le pire des cas est alors en $\mathcal{O}(n \times \log(n)^2)$, où n désigne le nombre d'e-mails reçus dans l'année.
On justifiera les complexités calculées en indiquant succinctement comment les différentes fonctions doivent être modifiées pour maintenir le dictionnaire à jour.