

I. Implémentation d'une structure de dictionnaire par tables de hachage

Une table de hachage est un tableau de w listes de couples (clé, valeur) muni d'une fonction dite de hachage qui associe à une clé un indice entre 0 et $w - 1$. On pourra noter K l'ensemble des clés, E l'ensemble des valeurs et $h : K \rightarrow \llbracket 0, w - 1 \rrbracket$ la fonction de hachage. La fonction de hachage renvoie le "haché" d'une clé qui est l'indice de la liste dans laquelle devra figurer un couple associé à cette clé.

On utilisera le type suivant

```
type ('a, 'b) table_hachage = {
  hache: 'a -> int;
  donnees: ('a * 'b) list array;
  largeur: int};;
```

Q1. Ecrire une fonction `creer_table h w` de signature `('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer_table h w` renvoie une nouvelle table de hachage vide de largeur w munie de la fonction de hachage h .

Q2. Ecrire une fonction `recherche t k` de signature `('a, 'b) table_hachage -> 'a -> bool` indiquant si la clé k est présente dans la table t .

Q3. Ecrire une fonction `element t k` de signature `('a, 'b) table_hachage -> 'a -> 'b` renvoyant l'élément associé à la clé k dans la table t lorsque celle-là est présente.

Q4. Ecrire une fonction `ajout t k e` de signature `('a, 'b) table_hachage -> 'a -> 'b -> unit` ajoutant le couple (k, e) dans la table t . On n'effectuera aucun changement si la clé est déjà présente.

Q5. Ecrire une fonction `suppression t k` de signature `('a, 'b) table_hachage -> 'a -> unit` supprimant le couple associé à la clé k dans la table t . On n'effectuera aucun changement en cas d'absence de cette clé.

On appelle alvéole chacune des listes de stockage des couples (clé, valeur). Une bonne fonction de hachage devra envoyer de façon uniforme les clés dans les différentes alvéoles. On fera cette hypothèse dans la suite. On note n le nombre de clés stockées dans la table. On appelle $\alpha = n/w$ le facteur de remplissage de la table. On suppose que la fonction de hachage renvoie un résultat en temps constant.

Q6. Montrer que la complexité de la recherche d'une clé k dans la table t est en $O(1 + \alpha)$. Etudier la complexité de l'ajout d'une clé. Comparer ces complexités avec celles obtenue pour la structure de dictionnaire via des arbres binaires de recherche. Donner les qualités et défauts de ces structures.

II. Tables de hachage dynamiques

L'analyse de la complexité précédente montre qu'il faut essayer de limiter le facteur de remplissage. Lorsque le nombre de clés à insérer est inconnu, il est utile de pouvoir ajuster la largeur de la table afin de maintenir un facteur de remplissage borné. Pour chaque largeur w , on devra avoir une fonction de hachage h_w ; on

utilisera donc une famille $(h_w)_{w \in \mathbb{N}^*}$ de fonctions de hachage. Des tables de hachage à largeur variable sont appelées tables de hachage dynamiques.

On utilisera l'implémentation suivante :

```
type ('a, 'b) table_dyn = {  
  hache: int -> 'a -> int;  
  mutable taille: int;  
  mutable donnees : ('a * 'b) list array;  
  mutable largeur: int};;
```

Le champ `taille` contiendra le nombre de clés présentes. Il permettra de savoir quand modifier largeur de la table de hachage.

Q7. Ecrire une fonction `creer_table_dyn` h de signature $('a \rightarrow \text{int}) \rightarrow ('a, 'b) \text{ table_dyn}$ permettant de créer une table de hachage dynamique vide dont la largeur initiale est 1.

Q8. Ecrire une fonction `recherche_dyn` $t \ k$ de signature $('a, 'b) \text{ table_dyn} \rightarrow 'a \rightarrow \text{bool}$ vérifiant si une clé est présente dans une table de hachage dynamique.

Q9. Ecrire une fonction `element_dyn` $t \ k$ de signature $('a, 'b) \text{ table_dyn} \rightarrow 'a \rightarrow 'b$ renvoyant l'élément associé à une clé dans une table de hachage dynamique lorsque celle-là est présente.

Q10. Ecrire une fonction `rearrange_dyn` $t \ w_2$ prenant en argument une table de hachage dynamique et une largeur qui recrée la table de hachage sur la largeur w_2 . La complexité doit être un $O(n + w + w_2)$ où n est le nombre de clés présentes, w l'ancienne largeur et w_2 la nouvelle.

On choisit le critère suivant pour réarranger une table de hachage en modifiant sa largeur : lorsque sa taille n dépasse $3w$, on la réarrangera en prenant $w_2 = 3w$.

Q11. Ecrire une fonction `ajout_dyn` $t \ k \ e$ ajoutant le couple (k,e) à la table de hachage dynamique t en réarrangeant si nécessaire la table selon le principe précédent.

Pour ne pas maintenir un nombre d'alvéoles inutilement élevé, lorsque le nombre de clé deviendra inférieur à sa largeur w , on réarrangera la table avec une nouvelle largeur $w_2 = w/3$.

Q12. Ecrire une fonction `suppression_dyn` $t \ k$ supprimant le couple associé à une clé. On réarrangera éventuellement la table selon le principe précédant.

III. Une structure de file à l'aide de listes

Une file est une structure permettant de conserver des valeurs. On doit pouvoir ajouter une valeur (en fin de liste) et retirer, en la renvoyant, la valeur insérée le plus tôt dans la file parmi les valeurs y figurant. On parle de structure LIFO ("last in, first out").

On peut implémenter une telle file par deux listes, l'une représentant le début de la liste, l'autre la fin. Les éléments du début de la liste seront énumérés dans l'ordre inverse de leur ajout dans la liste.

Par exemple les couples "debut=[1;2], fin=[4;0;3]", "debut=[1], fin=[4;0;3;2]", "debut=[], fin=[4;0;3;2;1]"

ou encore "debut=[1;2;3;0;4], fin=[]" représentent tous la même file, à savoir la file qui a reçu dans l'ordre 1, puis 2, puis 3, puis 0 et enfin 4. Si l'on ajoute un nouvel élément, il le sera à la suite de 4; si on retire un élément, ce sera 1.

On utilisera l'implémentation suivant :

```
type 'a file1 = { mutable debut : 'a list ; mutable fin : 'a list }
```

Q13. Ecrire des fonctions `creer_file_vide : unit -> 'a file1` et `est_file_vide : 'a file1 -> bool` permettant de créer la file vide et de vérifier si une file est vide.

Q14. Ecrire une fonction `insere_file : 'a -> 'a file1 -> unit` permettant d'insérer un élément à la fin d'une liste.

Q15. Ecrire une fonction `supprime_file : 'a file1 -> 'a` supprimant l'élément en tête de file en le renvoyant. Lorsque la tête de file est vide, on videra la fin de la liste dans le début en inversant l'ordre des éléments.

Q16. Analyser la complexité des fonctions précédentes. On pourra évoquer la complexité "amortie" (ou encore en moyenne) pour certaines d'entre elles.

IV. Une structures de files à l'aide de tableaux

On pourra aussi représenter une file par un tableau de taille n , un indice de début d et un indice de fin f compris entre 0 et $n-1$. Lorsque $d < f$, l'élément de tête de la liste est celui en position d , le suivant en position $d+1$, etc. Le dernier élément sera en position $f-1$. Lorsque $d = f$, la file sera vide. Lorsque $d > f$, le premier élément sera en position d , suivi de l'élément en position $d+1$, etc. jusqu'à l'élément en position $n-1$, suivi de l'élément en position 0, suivi de l'élément en position 1, etc. jusqu'à l'élément en position $f-1$.

On utilisera l'implémentation :

```
type 'a file 2 = { contenu : 'a array ; mutable d : int ; mutable f : int }
```

Q17. Ecrire une fonction `creer_file_vide2 : int -> 'a file2` créant une file vide dont le tableau "contenu" a taille celle de l'entier en argument.

Q18. Ecrire une fonction `est_file_vide2 : 'a file2 -> bool` vérifiant si une file est vide.

Q19. Ecrire une fonction `supprime_file2 : 'a file2 -> 'a` supprimant l'élément en tête de file en le renvoyant.

Q20. Ecrire une fonction `insere_file2 : 'a -> 'a file2 -> unit` ajoutant un élément en fin de liste. Si la liste est déjà pleine, on doublera la taille du tableau en recopiant le tableau initial.

Q21. Donnant la complexité des fonctions précédentes. Comparer les avantages et les inconvénients de cette structure par rapport à celle de la partie précédente.