TP : Optimisation en base de données

Ce sujet est composé de trois parties qui doivent être toutes traitées. Il sera attendu du candidat qu'il puisse démontrer la correction de son code lors de son oral ou des questions.

1 Arbre de recherche

Cette partie devra être implémentée en OCaml.

Les arbres B+ sont des arbres de recherche qui permettent de chercher, ajouter et supprimer des données rapidement. Ces structures sont en particulier utilisées dans les bases de données pour optimiser la recherche de faits ayant une valeur précise à l'un de leurs attributs. Une telle structure permet de rechercher si une valeur est présente dans l'arbre, d'ajouter une valeur et de supprimer une valeur. Les valeurs sont stockées dans les feuilles de l'arbre et des informations pour faciliter les opérations sont stockées dans les nœuds internes. Il n'y a pas de doublons dans les valeurs stockées dans les feuilles.

Le but de cet exercice est d'implémenter un arbre B+ et il est séparé en trois parties : l'implémentation de la structure ainsi que deux des 3 opérations : recherche d'une valeur, ajout d'une valeur.

1.1 Présentation de la structure

Soit V un ensemble de valeurs ordonnées par la relation $\leq_V$ comme, par exemple, les entiers pour la comparaison usuelle. En pratique, pour l'implémentation en OCaml, on utilisera la comparaison polymorphe.

Étant donné un paramètre entier $k \geq 2$, un k-arbre B+ est composé d'un ensemble de nœuds, chaque nœuds comportant entre $\lceil k/2 \rceil$ et k valeurs. Une exception est la racine qui peut stocker au plus k informations mais n'a pas de limite inférieure au nombre de valeurs stockées dans la racine.

Chaque nœud n qui n'est pas la racine comporte, en notant $m(n)$ le nombre de valeurs de ce nœud, soit $m(n)+1$ sous-arbres (c'est un nœud interne), soit 0 sous-arbres (c'est une feuille).

Tout arbre B+ doit vérifier la propriété suivante : Toutes les valeurs d'un nœud sont classées par ordre croissant, et pour un nœud interne, les sous-arbres et les valeurs stockées dans le nœuds sont ordonnées ainsi que les valeurs en suivant l'ordre $\leq_V$: en notant $v(n)$ les valeurs stockées dans un nœud et $vt(t)$ les valeurs stockées dans le sous-arbre t, pour tous sous-arbres t_1 et t_2 de n,

- Soit pour toute valeur v_1 dans $vt(t_1)$ et toute valeur v_2 dans $vt(t_2)$, $v_1 < v_2$ et il existe une valeur dans $v(n)$ telle que $v_1 < v \leq v_2$;
- Soit pour toute valeur v_1 dans $vt(t_1)$ et toute valeur v_2 dans $vt(t_2)$, $v_2 < v_1$ et il existe une valeur dans $v(n)$ telle que $v_2 < v \leq v_1$.

Dans la suite de ce TP, **nous considérons les arbres B+ ayant un k égal à 2**. Cela signifie que tout nœud possède une ou deux valeurs et zéro, deux ou trois sous-arbres, et pour tout nœud interne n, s'il ne possède qu'une seule valeur v, alors il possède deux sous arbres t_1 et t_2 tels que toute valeur v_1 dans $vt(t_1)$ est strictement inférieure à v et toute valeur v_2 dans $vt(t_2)$ est supérieure ou égale à v, et si n possède deux valeurs v_1 et v_2 avec $v_1 < v_2$ alors il possède trois sous-arbres t_1, t_2 et t_3 tels que

- pour toute valeur v dans $vt(t_1)$, $v < v_1$
- pour toute valeur u dans $vt(t_2)$, $v_1 \leq u < v_2$
- pour toute valeur w dans $vt(t_3)$, $v_2 \leq w$

Un arbre B+ a également la propriété suivante : la distance d'une feuille à la racine est la même pour toute feuille de l'arbre. Cette propriété est assurée par les algorithmes d'ajout et de suppression.

Notons que dans un arbre B+ classique, chaque feuille comporte un pointeur vers la feuille suivante. Cette partie de la structure n'est pas demandée dans ce sujet et ne doit pas être implémentée.

Exemple 1 *La figure 1 présente un exemple d'arbre B+ contenant les valeurs* 3, 10, 13, 21 *et* 34.

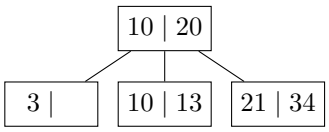

FIGURE 1 – Exemple d'un arbre B+

Question 1 *Coder en* OCAML *une structure pour représenter les arbres B+ polymorphes, c'est-à-dire qui peut prendre des valeurs dans un* `'a` *quelconque. Cette structure doit fonctionner exactement pour k égal à* 2*. Le type de cette structure est nommé* `'a bptree`*.*

1.2 Recherche de valeurs

Une des opérations importantes des arbres B+ est de rechercher si une valeur est stockée dans **l'une des feuilles** de l'arbre B+.

Question 2 *En utilisant les propriétés des arbres B+ et de la structure implémentée précédemment, vous devez implémenter en* OCAML *une fonction* `search` *qui est de type* `'a` $\longrightarrow$ `'a bptree` $\longrightarrow$ `bool`*. Cette fonction prend en entrée une valeur de type* `'a` *et un arbre B+ de type* `'a bptree` *et renvoie* `true` *si et seulement si la valeur est présente dans l'une des feuilles de l'arbre.*

L'implémentation proposée devra être de complexité linéaire en la hauteur de l'arbre.

1.3 Insertion d'une valeur

Un arbre B+ possède des propriétés importantes qui doivent être préservées lorsqu'une valeur est ajoutée ou supprimée. Dans ce sujet, nous nous focalisons sur l'ajout d'une valeur.

Pour insérer une valeur v, il faut tout d'abord rechercher la feuille telle que les valeurs de la précédente feuille soient inférieures à v et les valeurs de la feuille suivante soient supérieures à la valeur v.

Question 3 *Implémenter la fonction* `searchnode`*, de type* `'a` $\longrightarrow$ `'a bptree` $\longrightarrow$ `'a bptree`*, qui renvoie la feuille de l'arbre dans laquelle l'algorithme de recherche précédent s'arrête en cherchant valeur* v*. La complexité de la fonction doit être linéaire dans la hauteur de l'arbre.*

Exemple 2 *Dans le cadre de la figure 1, si l'on souhaite rajouter la valeur* 15*, la fonction* `searchnode` *va renvoyer la feuille ayant pour valeur* 10 *et* 13*.*

Une fois cette feuille obtenue, il existe alors plusieurs cas :

- Soit il existe assez de place pour insérer cette valeur dans la feuille et la valeur est simplement insérée dans la feuille adéquate ;
- Soit il n'existe pas assez de place pour insérer cette valeur dans la feuille. Dans ce cas, la feuille est divisée en deux feuilles, l'une contient la valeur la plus petite de $v(f) \cup \{v\}$ et l'autre les 2 plus grandes valeurs. La valeur médiane est alors ajoutée au nœud parent et les deux nouvelles feuilles sont ajoutées au nœud parent.

L'ajout d'une valeur à un nœud interne provient toujours de la séparation d'un nœud-fils en deux : un sous-arbre doit être remplacé par une valeur et deux sous-arbres. Nous expliquons ici comment cet ajout est fait, l'on ajoute la valeur v et les sous-arbres t_1 et t_2 à un nœud n. Nous considérons les différents cas :

- n a une valeur w et deux sous-arbres s_1 et s_2 et l'ajout de v provient de s_2. Le nœud n est transformé en ajoutant v à n et t_1 et t_2 remplacent s_2.
- n a une valeur w et deux sous-arbres s_1 et s_2 et l 'ajout de v provient de s_1. Le nœud n est transformé en ajoutant v à n et t_1 et t_2 remplacent s_1.
- n a deux valeurs w_1 et w_2 et trois sous-arbres s_1, s_2 et s_3 et l'ajout de v provient de s_1. Le nœud n est transformé en deux sous-arbres, le premier sous-arbre ayant pour racine un nœud contenant v et pour sous arbres t_1 et t_2, le deuxième sous-arbre ayant pour racine un nœud ayant pour valeur w_2 et les sous-arbres s_2 et s_3, la valeur w_1 est ajoutée au nœud parent.

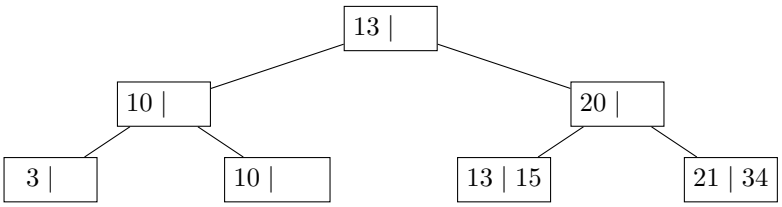

FIGURE 2 – Exemple d'un arbre B+ après l'ajout de la valeur 15

— n a deux valeurs w_1 et w_2 et trois sous-arbres s_1, s_2 et s_3 et l'ajout de v provient de s_2. Le nœud n est transformé en deux sous-arbres, le premier sous-arbre ayant pour racine un nœud contenant w_1 et pour sous arbres s_1 et t_1, le deuxième sous-arbre a pour racine un nœud ayant pour valeur w_2 et les sous-arbres t_2 et s_3, la valeur v est ajoutée au nœud parent.
— n a deux valeurs w_1 et w_2 et trois sous-arbres s_1, s_2 et s_3 et l'ajout de v provient de s_3. Le nœud n est transformé en deux sous-arbres, le premier sous-arbre ayant pour racine un nœud contenant w_1 et pour sous arbres s_1 et s_2, le deuxième sous-arbre a pour racine un nœud ayant pour valeur v et les sous-arbres t_1 et t_2, la valeur w_2 est ajoutée au nœud parent.

La procédure d'ajout continue récursivement jusqu'à la création possible d'une nouvelle racine si cette dernière avait déjà 2 valeurs.

Exemple 3 *Dans le cadre de la figure 1, si l'on souhaite rajouter la valeur* 15 *à la feuille ayant pour valeur* 10 *et* 13*, cela n'est pas possible car cette feuille a déjà deux valeurs. D'après notre algorithme, cette feuille est alors divisée en deux feuilles l'une contenant la valeur* 10 *et l'autre contenant les valeurs* 13 *et* 15*. La valeur médiane* 13 *est rajoutée au nœud parent. Hélas, la racine contient déjà deux valeurs. Il faut donc diviser ce nœud en deux en utilisant la valeur* 13 *comme médiane. Un nouveau nœud* n_1 *ayant pour valeur* 10 *est créé et a pour sous-arbres les deux feuilles contenant des valeurs strictement inférieures à* 13 *et un autre nœud* n_2 *est créé ayant pour valeur* 20 *et ayant pour sous-arbres les deux feuilles contenant des valeurs supérieures à* 13*. Finalement, une nouvelle racine est créée avec pour valeur* 13 *et possède deux sous arbres ayant pour racine* n_1 *et* n_2*. La figure 2 représente l'arbre résultant.*

Question 4 *Vous implémenterez la fonction* `insert` *de type* `'a` $\longrightarrow$ `'a bptree` $\longrightarrow$ `'a bptree` *qui prend en entrée une valeur et un arbre B+ et qui renvoie l'arbre obtenu en insérant la valeur à l'arbre. Cette fonction devra suivre l'idée proposée ci-dessus. Vous devrez également justifier que votre code préserve les propriétés de l'arbre B+, soit 1. l'ordre croissant des valeurs stockées dans l'arbre et 2. toutes les feuilles sont à la même profondeur.*

Cette question est indépendante de la question précédente.

2 Optimisation pour produire l'intersection d'ensemble d'entiers

Cette partie devra être implémentée en C.

Dans le cadre de la gestion des bases de données, l'approche classique pour optimiser une requête de base de données est de limiter le nombre d'accès au disque dur. Pour rappel, un ordinateur stocke ses données pérennes dans son disque dur, possède une mémoire vive et un cache pour faire des calculs sur des données éphémères. Historiquement, la mémoire vive et le cache étaient insuffisants pour stocker l'ensemble des données d'une base. De plus, l'accès au disque dur est bien plus long que l'accès à la mémoire vive et au cache. Les accès disque étaient donc les parties à optimiser. Par exemple pour implémenter l'opérateur de jointure, il existe plusieurs algorithmes pour limiter le nombre d'accès au disque.

Dans cet exercice, nous introduisons ces problèmes dans une version simplifiée. Les tables sont des ensembles d'entiers **non nuls**. Les jointures deviennent dans ce cadre des intersections d'ensembles. De

plus, le stockage sur disque des ensembles est représenté par des tableaux. Ainsi chaque ensemble est représenté par un tableau dont chaque entrée représente un élément de l'ensemble.

Le but de cet exercice est donc de limiter le nombre d'accès aux tableaux dans les algorithmes proposés. La complexité utilisée dans cet exercice est définie comme le nombre d'accès aux différents tableaux.

Représentation des ensembles par des tableaux Un ensemble d'entiers positifs non nuls est représenté dans notre exercice par un tableau. Pour plus de facilité, nous créons une structure qui contient la taille du tableau et le tableau lui-même. Plus précisément, cette structure est appelée `Table`. Cette structure possède deux champs : `tab` qui contient le tableau et `size` qui correspond à la taille du tableau. **Attention, dans notre modélisation, les tableaux peuvent être plus grands que les ensembles qu'ils représentent. La valeur 0 est utilisée pour remplir les cases superflues du tableau et doit ainsi être ignorée lors de l'intersection. C'est la seule valeur qui peut être apparaître plusieurs fois dans un même tableau.** Si un 0 apparaît à l'indice i du tableau alors toutes les valeurs apparaissant à des indices supérieurs à i sont égales à 0. Comme dans les ensembles, nous supposons qu'il n'y a pas de valeurs dupliquées dans les tableaux.

```
struct Table
{
  int *tab; /* tableau codant le set */
  int size ; /* taille du tableau */
};
```

Cet exercice est séparé en deux parties : une première sur l'optimisation de l'intersection de deux ensembles d'entiers et une deuxième sur l'optimisation de l'intersection de plusieurs ensembles d'entiers.

2.1 Optimisation de l'intersection de tableaux

Question 5 *Écrire une fonction qui prend trois structures de type `Table` en entrée. On suppose, pour cette question, que les tableaux ne sont pas triés. La fonction fait l'intersection des ensembles représentés par les deux premières structures et le résultat est stocké dans la troisième. Nous supposons que le troisième tableau a une taille suffisante pour stocker le résultat.* ***Nous imposons que les seules variables locales utilisées dans le programme sont des variables de type entier****.*

Intersection de tableaux triés

Question 6 *Écrire une fonction qui prend trois structures de type `Table`. Les deux premiers tableaux dans les structures sont triés par ordre croissant. La fonction fait l'intersection des ensembles représentés par les deux premières structures et le résultat est stocké dans la troisième structure. À nouveau, nous supposons que la troisième structure a une taille suffisante pour stocker le résultat. Nous imposons que* ***les seules variables qui sont utilisées dans le programme sont des variables de type entier****.*

2.1.1 Intersection de tableaux avec Index

Dans une base de données, un index est une structure qui permet de rapidement des enregistrements, comme par exemple ceux possédant une valeur précise.

Dans le cadre simplifié de notre modélisation, nous utiliserons comme index une structure qui permet de dire efficacement si une valeur donnée en entrée appartient à l'ensemble. En pratique, un index pour un ensemble E sera représenté dans ce sujet par un tableau t de booléens tel que $t[i]$ est la valeur `true` si et seulement si la valeur i apparaît dans E.

Question 7 *Construire une fonction qui prend en entrée une structure de type `Table` représentant un ensemble et qui renvoie un index défini comme plus haut ainsi que sa taille. L'index et sa taille peuvent être contenus dans une structure similaire au type `Table`.*

Question 8 *Écrire une fonction qui prend en entrée deux structures de type `Table` représentant deux ensembles, deux tableaux représentant les index de ces ensembles, deux entiers représentant les tailles de ces deux index et une dernière structure de type `Table`. La fonction doit calculer l'intersection des*

ensembles représentés par les deux premières structures et stocker le résultat dans la dernière structure. Nous supposons que la dernière structure a une taille suffisante pour stocker le résultat. Nous imposons que les seules variables utilisées dans le programme sont des variables de type entier. Vous vous aiderez des index pour faire l'intersection des deux premiers tableaux. La complexité de l'implémentation devra être en $\min(|t_1|, |t_2|)$ *où* t_1 *et* t_2 *sont les ensembles donnés en entrée, et* $|t_1|$ *et* $|t_2|$ *leurs nombres d'éléments respectifs.*

2.1.2 Comparaison des algorithmes

Nous cherchons à présent à comprendre comment se comportent les algorithmes implémentés dans cette section.

Question 9 *Présenter une analyse qui permet de comparer les différentes implémentations en fonction du nombre d'accès aux tableaux.*

Nous rappelons que cet exercice a pour but d'étudier les algorithmes de jointure de tables dans une base de données relationnelle. Dans le cadre de l'optimisation des jointures, le but est d'optimiser le nombre d'accès au disque dur où sont stockées les données. Dans cet exercice, nous simulons ces problèmes en représentant les données stockées sur le disque dur par des tableaux et le but est d'optimiser le nombre d'accès à ces tableaux.

Question 10 *Discuter la modélisation du problème étudié dans cet exercice au regard de votre implémentation. Vous pourrez décrire la complexité en nombre d'appels disques de vos algorithmes de façon précise si possible ainsi que l'impact de ceux-ci sur la mémoire vive.*

2.2 Intersection de multiples ensembles

Nous étudions maintenant le problème de l'intersection d'un grand nombre d'ensembles. Ces ensembles sont stockés dans des tableaux.

Question 11 *Construire une fonction qui prend en entrée un tableau de structures de type* `Table`*, un entier indiquant la taille de ce tableau et une structure de type* `Table` *et stocke dans le tableau de cette dernière structure l'intersection des ensembles représentés par les structures de type* `Table` *du premier tableau. Le tableau de la dernière structure est supposé suffisamment grand pour stocker les intersections intermédiaires et le résultat. On s'efforcera malgré tout de limiter sa taille.* ***Les seules variables possibles dans votre programme sont des variables de type entier.***

3 Audit de code

Cet exercice est composé de deux parties chacune composée d'un énoncé et d'une proposition de solution. Vous commenterez et corrigerez la solution proposée pour chaque énoncé. Les codes ci-dessous sont disponibles en lecture seule dans votre compte, à la racine.

3.1 Parcours de graphes

Énoncé 1 *Le but de cet exercice est de parcourir un graphe orienté acyclique dont les feuilles (les nœuds sans successeur) sont annotées par des entiers et dont chaque nœud a deux successeurs ou aucun. Présenter une structure qui représente ces graphes. Implémenter une fonction qui prend en entrée votre structure représentant le graphe et renvoie la somme des valeurs associées aux feuilles de ce graphe.*

Solution 1

```
type graph = Leaf of int | Node of graph * graph ;;

let rec sumgraph graph =
  match graph with
  | Leaf v -> v
  | Node (g,h) -> (sumgraph g) + (sumgraph h) ;;
```

3.2 Parcours de listes

Énoncé 2 *Le but de cet exercice est de parcourir une liste d'entiers et de filtrer cette liste pour ne conserver que les valeurs supérieures à une certaine valeur. Implémenter une fonction qui prend en entrée une liste d'entiers et un entier et qui modifie la liste pour ne conserver que les éléments strictement supérieurs à cet entier. La fonction renvoie l'adresse de la tête de la liste.*

Solution 2

```
#include <stdio.h>
#include <stdlib.h>
typedef struct s_List List;
struct s_List {
  List *next; /* pointeur sur le reste de la liste */
  int data;   /* pointeur sur une donnée generique */
};
int size_list(List *list) {
  int size = 0;
  List *current = list;
  while (current != NULL) {
    size += 1;
    current = current->next;
  }
  return size;
}

List *filter(List *list, int a) {
  int k = size_list(list);
  List *prev = NULL;
  List *head = NULL;
  List *current = list;
  for (int i = 0; i < k; i++) {
    int value = current->data;
    if (value <= a) {
      if (prev != NULL) {
        prev->next = current->next;
      }
      List *tmp = current;
      current = current->next;
      free(tmp);
    } else {
      if (prev == NULL) {
        prev = current;
        head = current;
      }
      current = current->next;
    }
  }
  return head;
}
```