UNIVERSITÉ HENRI POINCARÉ NANCY 1
ESIAL

ESIAL 1ère année
Date : 29 mai 2007
Horaire : 8h à 10h

Epreuve d'Informatique de base (IB3)
Documents non autorisés
Calculatrices non autorisées

Avertissement : la clarté de la rédaction et la justification des réponses sont des éléments essentiels de l'appréciation. Le barême est donné à titre indicatif.

Exercice 1 Elément médian (7 points)

L'élément médian d'un ensemble ordonné contenant n éléments (avec $n > 0$) est le $k$ième plus petit élément de l'ensemble, avec $k = (n+1)/2$ (/ dénote la division entière). Par exemple, l'élément médian de {16, 12, 99, 95, 18, 87, 10} est 18, l'élément médian de {26, 33, 55, 14, 87, 10} est 26.

On travaille ici avec des ensembles d'entiers, représentés par des tableaux. On peut remarquer que dans un tableau a : $tab[1..n]$ trié par ordre croissant, l'élément médian est l'élément d'indice $(n+1)/2$. Le calcul de l'élément médian peut évidemment se faire en triant le tableau, mais on se propose ici d'écrire un algorithme plus efficace.

La méthode proposée pour calculer l'élément médian utilise une fonction *partitionne* (comme dans le tri rapide). Rappelons la spécification de la fonction *partitionne* :

partitionne(t, p, r) : q

Précondition : t : $tab[1..n]$*d'entiers* $\wedge$ $n \geq 1$ $\wedge$ p, r : *entier* $\wedge$ $1 \leq p \leq r \leq n$

Postcondition : q : *entier* $\wedge$ $p \leq q \leq r$ $\wedge$ $\forall u \in [p, q[\ t[u] < t[q]$ $\wedge$ $\forall v \in [q, r]\ t[v] \geq t[q]$

Le résultat de *partitionne* est un entier q, indice de l'élément pivot qui sert à partitionner le sous-tableau $t[p..r]$. Remarquons que l'ensemble des éléments du sous-tableau $t[p..r]$ n'est pas modifié par l'exécution de la fonction *partitionne* (cette condition n'a pas été exprimée dans la postcondition pour ne pas alourdir la spécification).

Soit t : $tab[1..n]$ un tableau d'entiers, posons $milieu = (n+1)/2$. Le calcul de l'élément médian peut se modéliser comme suit. On applique la fonction *partitionne* au tableau t, avec les indices $p = 1$ et $r = n$. Soit q le résultat de cet appel ; on distingue alors les trois cas suivants :

- si $q = milieu$ alors l'élément médian de t est $t[q]$
- si $q < milieu$ alors l'élément médian appartient au sous-tableau $t[q+1..r]$, on continue la recherche dans le sous-tableau $t[q+1..r]$
- si $q > milieu$ alors l'élément médian appartient au sous-tableau $t[p..q-1]$, on continue la recherche dans le sous-tableau $t[p..q-1]$

Question a) Ecrire une fonction *median*(t; p, r : *entier*) : *res* : *entier* telle qu'un appel de la forme *median*(t, 1, n) calcule l'élément médian du tableau t à l'aide de la méthode décrite ci-dessus.

Question b) Evaluer la complexité dans le pire des cas de la fonction *median* précédemment écrite en fonction du nombre de comparaisons d'éléments du tableau, sachant que le nombre de comparaisons de la fonction `partitionne` pour un tableau de taille n est $n-1$.

Question c) Evaluer la complexité en moyenne de la fonction *median* en fonction du nombre de comparaisons, en supposant qu'à chaque étape la fonction *partitionne* partage le tableau en deux sous-tableaux de tailles approximativement égales.

Question d) Généraliser l'algorithme précédent au calcul du $i$ème plus petit élément du tableau t.

Exercice 2 Arbres binaires de recherche (7 points)

Voici un extrait de la spécification algébrique du type `BinarySearchTree[T]` qui décrit les arbres binaires de recherche (tous les axiomes ne sont pas cités). Les fonction `empty` et `makeRoot` sont les deux constructeurs du type.

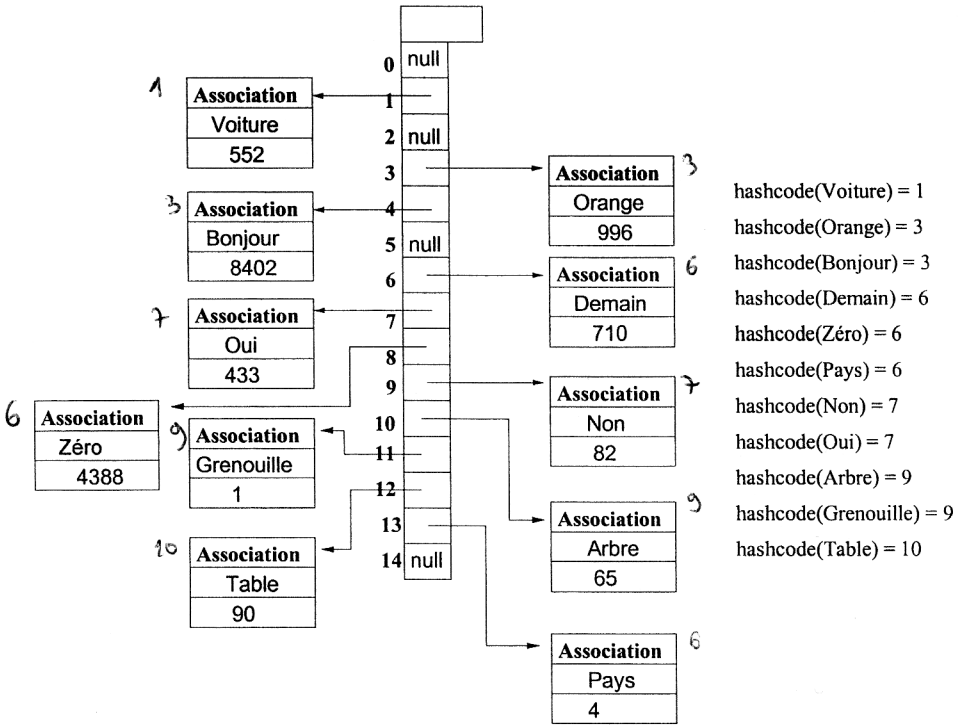
```
Type BinarySearchTree [T]
Opérations
 empty : ⟶ BinarySearchTree[T]                                   -- créer un arbre vide
 makeRoot : BinarySearchTree [T] X T X BinarySearchTree [T] ⟶ BinarySearchTree [T]
                                                                 -- enraciner 2 ss-arbres

 isEmpty : BinarySearchTree [T] ⟶ booléen                        -- arbre vide ?
 hasLeft : BinarySearchTree [T] ⟶ booléen                        -- a un sous-arbre gauche non vide ?
 hasRight : BinarySearchTree [T] ⟶ booléen                       -- a un sous-arbre droit non vide ?
 has : BinarySearchTree [T] X T ⟶ booléen                        -- existence de la valeur ?
 value : BinarySearchTree [T] ⟶ T                                -- valeur attachée à la racine
 left : BinarySearchTree [T] ⟶ BinarySearchTree [T]              -- sous-arbre gauche
 right : BinarySearchTree [T] ⟶ BinarySearchTree [T]             -- sous-arbre droit
 height : BinarySearchTree [T] ⟶ entier                          -- hauteur
 count : BinarySearchTree [T] ⟶ entier                           -- nombre de nœuds
 smallest : BinarySearchTree[T] ⟶ T                              -- plus petite valeur
 greatest : BinarySearchTree[T] ⟶ T                              -- plus grande valeur
 predecessor : BinarySearchTree[T] ⟶ T                           -- valeur précédente de la racine
 successor : BinarySearchTree[T] ⟶ T                             -- valeur suivante de la racine
 add : BinarySearchTree[T] X T ⟶ BinarySearchTree[T]             -- ajouter une valeur
 deleteGreatest : BinarySearchTree[T] ⟶ BinarySearchTree[T]      -- supprimer la plus grande valeur
 deleteSmallest : BinarySearchTree[T] ⟶ BinarySearchTree[T]      -- supprimer la plus petite valeur
 delete : BinarySearchTree[T] X T ⟶ BinarySearchTree[T]          -- supprimer une valeur
Préconditions
 makeRoot(l, v, r) défini ssi
              isEmpty(l) ou isEmpty(r) ou (greatest(l) < v < smallest(r))
 smallest(b) défini ssi non isEmpty(b)
 greatest(b) défini ssi non isEmpty(b)
 predecessor(b) défini ssi hasLeft(b)
 successor(b) défini ssi hasRight(b)
Axiomes
 deleteSmallest(empty()) = empty()
 deleteSmallest(makeRoot(empty(), v, r)) = r
 deleteSmallest(makeRoot(g, v, r)) = makeRoot(deleteSmallest(g), v, r) avec g ≠ empty()
 .....
```

Question a) La classe `BinarySearchTree` est implantée par trois champs : le champ `value` de type `Object` contient la valeur du nœud, les champs `left` et `right` de type `BinarySearchTree` contiennent les fils gauche et droit. L'arbre vide est toujours représenté par une instance de `BinarySearchTree` avec trois champs contenant `null`. Ecrire la définition de la fonction `deleteSmallest` en respectant la spécification.

Question b) Ecrire les axiomes de la fonction `delete`, en cherchant à équilibrer au mieux les hauteurs des fils gauche et droit de l'arbre produit. *Idée : appuyez-vous sur l'exemple des axiomes de `deleteSmallest` qui définissent l'effet de cette fonction sur le résultat des deux constructeurs.*

Exercice 3 Implantation de table par hachage ouvert (6 points)

Le hachage ouvert est un cas particulier d'implantation par hashcode. Les associations <key, value> sont stockées dans une structure contigüe ; aucune structure supplémentaire n'est utilisée. Les éléments d'une sous-table (donc de même hashcode) ne sont pas chaînés entre eux, leur rangement ne respecte aucun ordre particulier dans la structure contigüe. La première association d'une sous-table commence à un indice égal ou supérieur au hashcode des éléments de la sous-table (modulo la capacité de la structure). Toutes les autres associations de la sous-table sont rangées à un indice supérieur à celui de la première association (modulo la capacité de la structure). La fin d'une sous-table est identifiée par la présence de `null`.



Dans l'exemple ci-dessus, la sous-table de hashcode 0 est vide ; la sous-table de hashcode 1 commence en 1 ; la sous-table de hashcode 10 commence en 12. La recherche (`rechercheCle`) et l'adjonction (`add`) sont simplement réalisées par des algorithmes de recherche associative, qui débutent à l'indice donné par la fonction de hashcode. Une clé est présente dans la structure si on peut l'atteindre à partir de l'indice égal à son hashcode sans rencontrer `null`.

Lors de la suppression (`delete`), il faut garantir la continuité dans la succession des associations et éventuellement remplacer l'association supprimée par une autre. En effet, dans certains cas, placer simplement `null` dans la case de l'association supprimée invalide la recherche effectuée par la fonction `rechercherCle`.

Dans l'exemple, la suppression de la clé *Voiture* est immédiate ; par contre la suppression de la clé *Table* exige un déplacement d'une autre association, de sorte que la prochaine recherche de la clé *Pays* puisse aboutir.

Un extrait de la classe `HashContiguousDictionary` est fourni en annexe.

Question a) Ecrire le corps de la fonction `delete`.

Question b) Décrire la situation la plus défavorable lors de la recherche d'une clé. Estimer la complexité en nombre de comparaisons d'associations. Comparer cette estimation avec la complexité de cette recherche dans l'implantation `HashLinkedDictionary`, étudiée en Td/Tp.

Annexe

```
1   package tables ;
    public class HashContiguousDictionary extends HashDictionary {

        /** Dictionnaire vide
            @param capacity capacité maximale de la table
         */
        //@ requires capacity > 0
        public HashContiguousDictionary(int capacity) {
            super(2*capacity) ;
10      } // HashContiguousDictionary(int)

        // ------------------------------------------------------
        /** Vrai si la table est pleine
         */
        public boolean isFull() {
            return size*2 >= capacity ;
        } // isFull()

        // ------------------------------------------------------
20      /** Ajouter un couple < clé, valeur >
         */
        public void add (Object key, Object value) {
            int hash = hashCode(key) ;
            // recherche d'une place libre
            while (getKeyValue(hash) != null)
                  hash = (hash+1)%capacity ;
            // ranger le nouveau couple
            setKeyValue(hash, new Association(key, value)) ;
        } // add(Object, Object)
30
        // ------------------------------------------------------
        /** Rechercher la place d'une clé
         *  @return indice ou -1 si pas trouvé
         */
        protected int rechercherCle (Object key) {
            int hash = hashCode(key) ;
            int libre = -1 ;
            while (getKeyValue(hash) != null && libre == -1) {
                 if (getKey(hash).equals(key)
40                  libre = hash ;
                 else
                    hash = (hash+1)%capacity ;
        } // rechercherCle(Object)

        // ------------------------------------------------------
        /** Supprimer une clé
         */
        public void delete (Object key)  { ....  }

50      // ------------------------------------------------------
        // Fonctions de consultation/mise à jour de la structure contigüe

        // Consulter la clé de rang k
        protected Object getKey(int k) { ....  }

        //Consulter l'association de rang k
        protected Association getKeyValue(int k) { ....  }

        //Modifier l'association de rang k
60      protected void setKeyValue(int k, Association a) { ....  }

    } // class HashContigousDictionary
```