Partiel de POO & AA

le 25/11/2009 10h00 – 12h00
Aucun document autorisé

1. Constantes

Considérons la classe Toto et la fonction f suivantes :

```
class Toto {
   public:
      void g(const Toto&);
      void h(Toto&) const;
};

void f(const Toto& t1, Toto& t2) {
   t1.g(t2);
   t1.h(t2);
   t2.g(t1);
   t2.h(t1);
}
```

Lesquels parmi les quatre appels dans f sont corrects et lesquels sont erronés ? Pourquoi ?

2. Listes triées

Nous allons considérer des listes chaînées qui sont maintenues triées en ordre croissant des valeurs. Les nouvelles valeurs seront ajoutées à la position correcte afin de maintenir l'ordre. Nous utiliserons les classes suivantes en C++ :

```
template<typename T> class ListeTriee {
   public:
      ListeTriee() { premier = dernier = NULL; }

      // ajoute t dans la liste en préservant l'ordre
      void ajouter(const T& t);

      // ajoute toutes les valeurs de l dans la liste en préservant l'ordre
      void ajouter_tous(const ListeTriee<T>& l);
      ...
   private:
      Element<T> *premier, *dernier;
      ...
};

template<typename T> class Element {
   public:
      Element(const T& t) : valeur(t), precedent(NULL), suivant(NULL) {}
   private:
      T valeur;
      Element<T> *precedent, *suivant;
   friend class ListeTriee<T>;
};
```

1. Implémenter la fonction membre ajouter (on suppose que le type-paramètre T définit les opérations de comparaison).
2. Transformer la classe ListeTriee en forme canonique en ajoutant un constructeur par recopie, un destructeur et un opérateur d'affectation.
3. Considérons l'implémentation suivante de la fonction membre ajouter_tous :

```
template<typename T> void ListeTriee<T>::ajouter_tous(const ListeTriee<T>& l) {
    for (Element<T> *p = l.premier; p != NULL; p = p->suivant)
        ajouter(p->valeur);
}
```

Cette implémentation n'est pas très efficace car si nous avons deux listes l1 et l2 de tailles respectivement n_1 et n_2, le temps d'exécution de l'appel l1.ajouter_tous(l2) est $O((n_1 + n_2)n_2)$. Donner une implémentation plus efficace de complexité $O(n_1 + n_2)$.

Examen de POO & Algorithmique Avancée

le 19/01/2009
Durée 3 h
Aucun document autorisé

1 Parcours d'arbres

Un arbre binaire est déterminé de manière unique par les séquences de ses noeuds en ordre préfixe et en ordre infixe.

1.1. Soit la séquence en ordre préfixe des étiquettes d'un arbre binaire :

A B C D E F G H I

et soit la séquence des étiquettes du même arbre en ordre infixe :

B C A E D G H F I

Construire l'arbre binaire correspondant en expliquant les différentes étapes de cette construction.

1.2. Si le parcours en ordre préfixe de trois noeuds donne la suite 1 2 3, combien y a-t-il d'arbres binaires possibles correspondant à ce parcours ? Dessiner chacun de ces arbres et donner leur parcours en ordre infixe.

2 Arbres d'intervalles

Les intervalles sont pratiques pour représenter des événements qui occupent chacun une période de temps continue. On pourrait, par exemple, avoir envie d'interroger une base de données d'intervalles de temps pour retrouver les événements qui se produisent pendant un intervalle donné. Les *arbres d'intervalles* fournissent un moyen efficace pour gérer une telle base d'intervalles.

Nous allons représenter les intervalles fermés par la classe suivante :

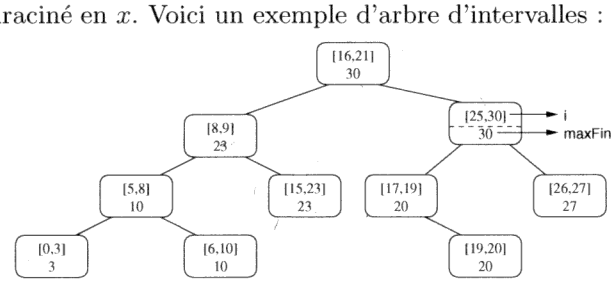
```
public class Intervalle
{  private double debut, fin;

   public Intervalle(double a, double b)
   {  debut = Math.min(a, b);
      fin = Math.max(a, b);
   }

   public double getDebut() { return debut; }
   public double getFin() { return fin; }

   // vrai si this recoupe i (this et i ont une intersection non vide)
   public boolean recoupe(Intervalle i)
   {  return debut <= i.fin && i.debut <= fin;
   }
}
```

Un arbre d'intervalles est un arbre binaire dans lequel chaque noeud contient un intervalle. C'est un arbre binaire de recherche par rapport à l'ordre des débuts des intervalles. Ainsi, par un parcours infixe, on peut recenser les intervalles triés selon leur début. En plus, chaque noeud x contient une valeur maxFin qui représente la valeur maximale parmi les fins d'intervalles stockés dans le sous-arbre enraciné en x. Voici un exemple d'arbre d'intervalles :



Les arbres d'intervalles seront représentés par la classe

```
public class ArbreIntervalle
{  private static class Noeud
   {  Intervalle i;
      double maxFin;
      Noeud gauche, droit, pere;
   }

   private Noeud racine;

   public Intervalle rechercher(Intervalle i)
   {  Noeud x = racine;
      while (x != null && !i.recoupe(x.i))
      {  if (x.gauche != null && x.gauche.maxFin >= i.getDebut()) x = x.gauche;
         else x = x.droit;
      }
      if (x == null) return null;
      else return x.i;
   }
   ...
}
```

2.1. Que renvoie la méthode rechercher sur l'arbre de l'exemple

(a) si i est l'intervalle $[22, 25]$;

(b) si i est l'intervalle $[11, 14]$.

2.2. Démontrer que la méthode rechercher soit renvoie un intervalle qui appartient à l'arbre et qui recoupe i, soit renvoie **null** et dans ce cas l'arbre ne contient aucun intervalle qui recoupe i.

2.3. Déterminer la complexité de l'algorithme implémenté par la méthode rechercher

(a) dans le pire des cas;

(b) dans le cas d'un arbre équilibré.

2.4. Ecrire une méthode **public** Intervalle minContientTous() de la classe ArbreIntervalle qui renvoie l'intervalle minimal contenant tous les intervalles appartenant à cet arbre. Sur l'arbre de l'exemple cette méthode doit renvoyer l'intervalle $[0, 30]$. Analyser la complexité de l'algorithme proposé.

2.5. Donner le résultat de l'insertion de l'intervalle $[14, 28]$ dans l'arbre de l'exemple.

2.6. Ecrire une méthode **public void** inserer(Intervalle i) de la classe ArbreIntervalle qui insère l'intervalle i dans cet arbre. La méthode doit maintenir à jour les champs maxFin des noeuds de l'arbre sans augmenter la complexité de l'algorithme d'insertion dans un arbre binaire de recherche.

2.7. Pour maintenir un arbre d'intervalles équilibré, nous avons besoin d'opérations de rotation efficaces. Donner le résultat d'une rotation gauche autour de la racine de l'arbre de l'exemple.

2.8. Ecrire une méthode **private void** rotationGauche(Noeud x) de la classe ArbreIntervalle qui effectue une rotation gauche autour de x et qui met à jour les champs maxFin en temps $O(1)$.

Examen de POO & Algorithmique Avancée

le 13/01/2010
Durée 3 h
Aucun document autorisé

1. Arbres binaires de recherche

Le professeur Arboretum pense avoir découvert une propriété remarquable des arbres binaires de recherche. Prenons un chemin de la racine à une feuille quelconque. Partageons les clés dans l'arbre en trois ensembles : A, les clés à gauche de ce chemin ; B, celles situées sur le chemin ; et C, les clés à droite du chemin. Le professeur Arboretum affirme que si l'on prend trois clés quelconques $a \in A$, $b \in B$ et $c \in C$, elles doivent satisfaire à $a \leq b \leq c$. Donner un contre-exemple qui démontre que le professeur Arboretum a tort.

2. Arbres 2-3-4

Les arbres 2-3-4 sont une généralisation des arbres binaires de recherche. Nous les représenterons par les classes suivantes en Java :

```
public class Arbre234 {
   private class Noeud {
      int n;            // nombre de clés stockées dans le noeud
                        // 0 (uniquement pour la racine d'un arbre vide), 1, 2 ou 3
      int[] cle;        // tableau contenant les n clés en ordre non décroissant
      Noeud[] fils;     // tableau contenant les n+1 fils du noeud
      boolean feuille; // vrai si et seulement si le noeud est une feuille

      Noeud() {
         n = 0;
         cle = new int[3];
         fils = new Noeud[4];
         feuille = true;
      }

      // on suppose que feuille vaut false, n < 3 et fils[i].n = 3
      void partagerFils(int i) { ... }
   }

   private Noeud racine;

   public Arbre234() {
      racine = new Noeud();
   }

   // renvoie true si la clé k est présente dans l'arbre
   public boolean rechercher(int k) { ... }

   // ajoute la clé k dans l'arbre
   public void ajouter(int k) { ... }
}
```

Un arbre 2-3-4 a les propriétés suivantes :

1. Chaque noeud x contient x.n clés rangées en ordre croissant non strict :

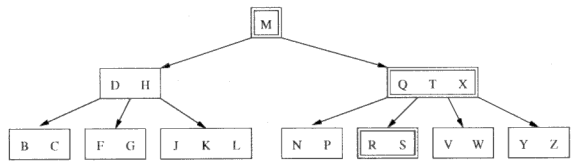
   x.cle[0] <= ... <= x.cle[x.n−1].

2. Chaque noeud interne x possède x.n + 1 fils.
3. Les clés déterminent les intervalles de clés stockées dans chaque sous-arbre. Pour chaque noeud x :
   - Toutes les clés stockées dans le sous-arbre de racine x.fils[0] sont inférieures ou égales à x.cle[0] ;

- Toutes les clés stockées dans le sous-arbre de racine x. fils [i] (i = 1 ... x.n−1) sont entre x.cle[i−1] et x.cle[i] ;
- Toutes les clés stockées dans le sous-arbre de racine x. fils [x.n] sont supérieures ou égales à x.cle[x.n − 1].

4. Toutes les feuilles ont la même profondeur (la hauteur h de l'arbre)
5. Le nombre de clés dans chaque noeud vérifie les contraintes suivantes :
   - Tout noeud contient au moins une clé. La seule exception est la racine d'un arbre vide qui ne contient pas de clé. Ainsi tout noeud interne possède au moins deux fils.
   - Tout noeud peut contenir au plus trois clés. Un noeud interne possède donc au plus quatre fils. On dit qu'un noeud est complet s'il contient exactement trois clés.

Ces propriétés impliquent que chaque noeud interne possède 2, 3 ou 4 fils d'où le nom arbres 2-3-4. Voici un exemple d'arbre 2-3-4. Les noeuds parcourus à la recherche de la clé R sont dessinés en doubles lignes.



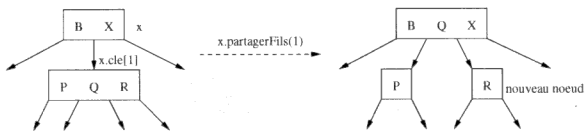
2.1. Démontrer que pour tout arbre à N clés de hauteur h

$$h \leq \log_2 \frac{N+1}{2}$$

Justifier l'intérêt d'utiliser les arbres 2-3-4 plutôt que les ABR classiques.

2.2. Implémenter la méthode rechercher de la classe Arbre234. Déterminer sa complexité asymptotique en fonction du nombre de clés N dans l'arbre.

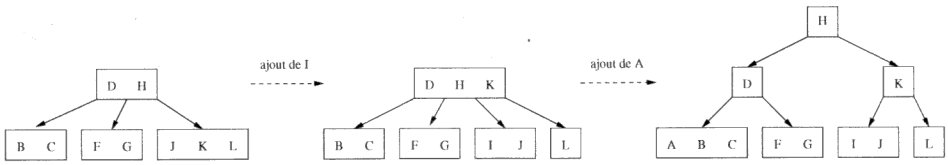
2.3. Pour effectuer une insertion, nous avons besoin d'une opération qu'on appellera partage de fils. Soit x un noeud interne qui n'est pas complet et x. fils [i] un fils complet de x. L'opération consiste à remonter dans x (à la place de x.cle[i]) la clé qui se trouve au milieu de x. fils [i] en décalant le reste à droite et à partager en deux le fils de x comme le montre le schéma suivant :



Ainsi l'ordre des clés est préservé et tous les noeuds ont entre 1 et 3 clés.

Implémenter la méthode partagerFils de la classe Noeud.

2.4. L'ajout d'une nouvelle clé se fait toujours au niveau d'une feuille. Cela peut poser un problème si la feuille est déjà complète. Pour éviter que cela arrive, on partage tous les noeuds complets qu'on rencontre sur le chemin. Le schéma suivant est un exemple d'ajout des clés I et A dans un arbre.



Pour ajouter la clé I on part de la racine et on voit qu'il faut aller à droite de H. Avant de descendre, on partage le noeud complet JKL en remontant K à la racine. Maintenant il faut descendre entre H et K et inserer I dans la feuille qui contient J. Pour effectuer l'insertion de A, on part à nouveau de la racine. Cette fois la racine est un noeud complet et on la partage (ceci permettra d'y remonter une clé si on rencontre un noeud complet plus bas). On crée un nouveau noeud qui accueille la clé H et devient la nouvelle racine. On descend à gauche de H dans D et ensuite à gauche de D dans BC. C'est une feuille et on y insère A.

Implémenter la méthode ajouter de la classe Arbre234. Déterminer sa complexité asymptotique en fonction du nombre de clés N dans l'arbre.