

COURS 421-b, COMPOSITION D'INFORMATIQUE

Philippe Jacquet

vendredi 23 janvier 2009

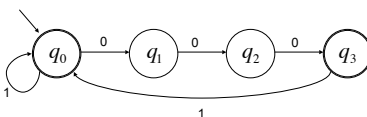
CORRIGÉ

Partie I : Automates et langages

Question 1. Dessiner l'automate déterministe qui reconnaît les séquences binaires constituées de "0" et de "1", et telles que les "0"s apparaissent en groupes disjoints de trois "0"s consécutifs.

Réponse :

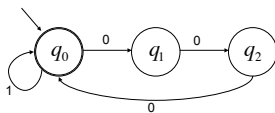
□



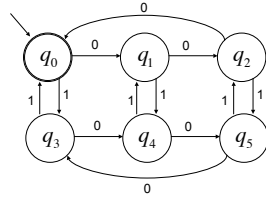
Question 2. Dessiner l'automate déterministe qui reconnaît les séquences binaires constituées de "0" et de "1", et telles que les "0"s apparaissent en groupes de tailles divisibles par trois.

Réponse :

□



Question 3. Dessiner l'automate déterministe qui reconnaît les séquences binaires constituées de "0" et de "1", et telles que les "0"s apparaissent en nombres divisibles par trois, et telles que les "1"s apparaissent en nombres divisibles par deux.



Réponse :

□

Partie II, Les arbres binaires de recherche.

Dans cette partie, nous nous intéressons aux arbres binaires de recherche. Les éléments insérés dans l'arbre sont des entiers. Nous introduisons la classe arbre

```
class arbre{
    int element;
    arbre[2] fils;
    arbre(int a, arbre gauche, arbre droit){ element=a; fils[0]=gauche;fils[1]=droit;}
    arbre(int a){element=a;fils[0]=null; fils[1]=null;}
}
```

Le principe est que les éléments stockés dans `fils[0]` sont inférieurs ou égaux à l'entier `element` stocké dans la racine, et que les éléments stockés dans `fils[1]` sont strictement supérieurs à l'entier `element` stocké dans la racine. Noter que l'arbre supporte la duplication d'éléments, c'est-à-dire peut contenir plusieurs éléments ayant la même valeur entière.

Question 4. Ecrire une méthode `static arbre search(int a, arbre t)` qui renvoie le premier sous-arbre de l'arbre `t` dont la racine contient l'élément `a`. Si l'élément `a` n'est pas stocké dans `t`, alors la méthode renvoie `null`.

Réponse :

```
static arbre search(int a, arbre t){
    if (t==null){return null;}
    if (t.element==a) {return t;}
    if (t.element<a){return search(a,t.fils[0]);}
    else {search(a,t.fils[1]);}
}
```

□

Question 5. Ecrire une méthode persistante `static arbre insert(int a, arbre t)` qui insère dans l'arbre `t` l'élément `a`. On permet la duplication : si l'élément `a` existe déjà dans l'arbre, on en rajoute quand même une nouvelle occurrence.

Réponse :

```
static arbre insert(int a, arbre t){
    if (t==null){return new arbre(a);}
    if (t.element<=a){return new arbre(t.element,insert(a,t.fils[0]),t.fils[1]);}
    if (t.element>a){return new arbre(t.element,t.fils[0],insert(a,t.fils[1]));}
}
```

□

Question 6. Ecrire une méthode **static void** `printinorder(arbre t)` qui imprime la liste des éléments stockés dans l'arbre `t` dans l'ordre croissant. En cas de duplication, un élément doit apparaître dans la liste autant de fois qu'il apparaît dans l'arbre. La méthode devra être de complexité linéaire et n'explorer l'arbre qu'une seule fois.

Réponse :

```
static void printinorder(arbre t){
    if (t!=null){
        printinorder(t.fils[0]);
        System.out.println(t.element);
        printinorder(t.fils[1]);
    }
}
```

□

Question 7. Ecrire une méthode persistante **static** `arbre liftmax(arbre t)` qui renvoie un arbre binaire de recherche qui contient les mêmes éléments (y compris leurs duplicats) de `t` mais avec l'un des plus grands éléments placé à la racine. On cherchera à minimiser la complexité et le nombre d'éléments déplacés.

Réponse :

```
static arbre liftmax(arbre t){
    if (t==null || t.fils[1]==null){return t;}
    else {
        arbre t1=liftmax(t.fils[1]);
        arbre t2=new arbre(t.element,t.fils[0],t1.fils[0]);
        return new arbre(t1.element,t2,null);}
}
```

□

Question 8. Ecrire une méthode **static** `arbre cutright(int a, arbre t)` qui renvoie l'arbre obtenu en retirant de l'arbre `t` tous les éléments supérieurs ou égaux à l'entier `a`. Ecrire une méthode **static** `arbre cutleft(int b, arbre t)` qui retire tous les éléments de l'arbre `t` inférieurs ou égaux à l'entier `b`. On s'attachera à minimiser la complexité.

Réponse :

```
static arbre cutright(int a, arbre t){
    if (t==null){return t;}
    if (t.element>=a){return cutright(a,t.fils[0]);}
    else {return new arbre(t.element,t.fils[0],cutright(a,t.fils[1]));}
}

static arbre cutleft(int b, arbre t){
    if (t==null){return t;}
    if (t.element<=b){return cutright(b,t.fils[0]);}
    else {return new arbre(t.element,cutleft(b,t.fils[0]),t.fils[1]);}
}
```

□

Question 9. Ecrire une méthode **static** `arbre cutblock(int a, int b, arbre t)` qui renvoie

l'arbre obtenu en retirant de l'arbre t tous les éléments à la fois supérieurs ou égaux à l'entier a et inférieurs ou égaux à l'entier b (on suppose $a \leq b$). On s'efforcera de limiter la complexité.

Réponse :

```
static arbre cutblock(int a, int b, arbre t){
    if (t==null){return null;}
    if (t.element<a){return new arbre(t.element,t.fils[0],cutblock(a,b,t.fils[1]));}
    if (t.element>b){return new arbre(t.element,cutblock(a,b,t.fils[0]),t.fils[1]);}
    arbre t0=cutright(a,t.fils[0]);
    arbre t1=cutleft(b,t.fils[1]);
    if (t0==null){
        if (t1==null){return null;}
        else{return t1;}
    } else{
        arbre t2=liftmax(t0);
        return new arbre(t2.element,t2.fils[0],t1);
    }
}
```

□

Partie III, Les arbres 2-3 de recherche.

Un arbre 2-3 est un arbre où chaque sommet peut contenir soit un élément, alors le sommet est de type 2, soit deux éléments, alors il est de type 3. Un sommet de type 2, qui contient l'élément a , a deux sous-arbres : le sous-arbre de gauche qui contient les éléments inférieurs ou égaux à l'entier a , et le sous-arbre de droite qui contient les éléments strictement supérieurs ou égaux à a .

Un sommet de type 3, qui contient deux éléments a et b , avec $a \leq b$, a trois sous-arbres : un sous-arbre de gauche qui contient les éléments inférieurs ou égaux à a , un sous-arbre du milieu qui contient les éléments strictement supérieurs à a et inférieurs ou égaux à b , et un sous-arbre de droite qui contient les éléments strictement supérieurs à b .

Ces arbres sont des alternatives aux arbres AVL et permettent d'améliorer les temps de recherche par rapport à l'arbre binaire de recherche.

Nous donnons la classe arbre23

```
class arbre23{
    int type;
    int[] element;
    arbre23[] fils;
    arbre23(int a, int b, arbre23 gauche, arbre23 milieu, arbre23 droit){
        type=3;
        element=new int [2];
        if (a<b) {element[0]=a; element[1]=b;}
        else{element[0]=b;element[1]=a;}
        fils=new arbre23[3];
        fils[0]=gauche;
        fils[1]=milieu;
        fils[2]=droit;
    }
    arbre23(int a, arbre23 gauche, arbre23 droit){
        type=2;
        element=new int [1];
        element[0]=a;
    }
}
```

```

        fils=new arbre23[2];
        fils[0]=gauche;fils[1]=droit;
    }
    arbre23(int a){
        type=2;
        element=new int [1];
        element[0]=a;
        fils=new arbre23[2];
        fils[0]=null;fils[1]=null;
    }
}

```

Noter que l'on autorise toujours la duplication d'éléments dans les arbres 2-3, comme elle était déjà autorisée avec les arbres binaires de recherche.

Un arbre 2-3 qui a tous ses sommets de type 2 est un arbre binaire de recherche.

On dit qu'un arbre 2-3 est *saturé* quand ses sommets autres que les feuilles sont tous de type 3. Les feuilles peuvent être de type 2 ou 3.

Question 10. Quelle propriété a le sous-arbre du milieu quand la racine est de type 3 et ses deux éléments sont identiques?

Réponse : Le sous-arbre du milieu est vide. □

Question 11. Ecrire une méthode persistante **static** `arbre convert(arbre23 t)` qui convertit un arbre 2-3 en un arbre binaire de recherche. On s'attachera à minimiser la complexité et le nombre d'éléments déplacés.

Réponse :

```

static arbre convert(arbre23 t){
    if (t==null){return null;}
    if (t.type==2){return new arbre(t.element[0],convert(t.fils[0]),convert(t.fils[1]));}
    arbre t0=new arbre(t.element[0],convert(t.fils[0]),convert(t.fils[1]));
    return new arbre(t.element[1],t0.convert(t.fils[2]));
}

```

□

Question 12. Ecrire une méthode persistante **static** `arbre23 convert23(arbre t)` qui convertit un arbre binaire de recherche en un arbre 2-3 saturé.

Réponse :

```

static arbre23 convert23(arbre t){
    if (t==null){return null;}
    if (t.fils[0]==null){
        if (t.fils[1]==null) {return new arbre23(t.element);}
        else {
            arbre23 t10=convert23(t.fils[1].fils[0]);
            arbre23 t11=convert23(t.fils[1].fils[1]);
            return new arbre23(t.element,t.fils[1].element,null,t01,t11);
        }
    }
    arbre 23 t00=convert23(t.fils[0].fils[0]);
    arbre 23 t01=convert23(t.fils[0].fils[1]);
    arbre 23 t1=convert23(t.fils[1]);
    return new arbre23(t.fils[0].element,t.element,t00,t01,t1);
}

```

}

□

Question 13. Ecrire une méthode `static void printinorder23(arbre23 t)` qui imprime la liste des éléments stockés dans l'arbre `t` dans l'ordre croissant en respectant les duplications et en n'explorant l'arbre qu'une seule fois.

Réponse :

```
static void printinorder23(arbre23 t){
    if (t!=null){
        printinorder23(t.fils[0]);
        System.out.println(t.element[0]);
        printinorder23(t.fils[1]);
        if(t.type==3){
            System.out.println(t.element[1]);
            printinorder23(t.fils[2]);
        }
    }
}
```

□

Question 14. Soit t l'arbre 2-3 dessiné dans la figure 1. L'arbre t comporte une racine de type 2 avec l'élément c . La racine a comme fils gauche un sommet de type 3, et comme fils droit un arbre t_4 . La racine de type 3 du sous-arbre gauche de t a pour éléments a et b , $a \leq b \leq c$, et t_1 comme fils gauche, t_2 comme fils du milieu, et t_3 comme fils de droite. Dessiner une modification de l'arbre t qui ait une racine de type 3, sans que la modification affecte les sous-arbres t_1 , t_2 , t_3 et t_4 , et ne change pas le nombre des occurrences de chacun des éléments.

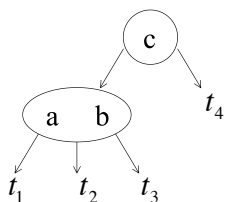


Figure 1: L'arbre 2-3 t

Réponse :

Attention, la variante où le sommet de type 3 contenant a et b , serait simplement déplacé à la racine n'est pas valable car il aurait le sommet de type 2 contenant c en fils droit. Cette configuration interdirait les cas d'égalité $b = c$.

□

Question 15. Ecrire une méthode persistante `static arbre23 sature(arbre23 t)` qui renvoie une version saturée d'un arbre 2-3 t quelconque sans changer le nombre d'éléments et le nombre de leurs occurrences. On s'attachera à minimiser la complexité et le nombre d'éléments déplacés.

Réponse : Il y a plusieurs solutions : celles qui favorisent la fusion à gauche et celles qui favorisent la fusion à droite. Nous décrivons une solution qui favorise la fusion à gauche.

```
static arbre23 sature(arbre23 t){
```

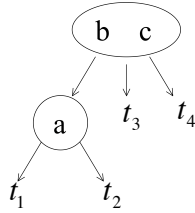


Figure 2: L'arbre 2-3 t

```

if (t==null){return null;}
if(t.type==3){
    arbre23 t0=sature(t.fils[0]);
    arbre23 t1=sature(t.fils[1]);
    arbre23 t2=sature(t.fils[2]);
    return new arbre23(t.element[0],t.element[1],t0,t1,t2);
}
arbre23 t0=t.fils[0];
arbre23 t1=t.fils[1];
if(t0!=null){
    arbre23 t00=t0.fils[0]);
    arbre23 t01=t0.fils[1]);
    if (t0.type==2){
        return new arbre23(t0.element[0],t.element[0],sature(t00),sature(t01),sature(t1));
    }
    arbre23 tt=new arbre23(t0.element[0],t00,t01)
    return new arbre23(t0.element[1],t.element[0],sature(tt),sature(t0.fils[2]),sature(t1));
}
if(t1!=null){
    arbre23 t10=t1.fils[0]);
    arbre23 t11=t1.fils[1]);
    if (t1.type==2){
        return new arbre23(t.element[0],t1.element[0],sature(t0),sature(t10),sature(t11));
    }
    arbre23 tt=new arbre23(t0.element[0],t00,t01)
    return new arbre23(t.element[0],t1.element[0],sature(t0),sature(t1.fils[0]),sature(tt));
}
return t;
}

```

□

Partie IV insertion et suppression dans les arbres 2-3

L'insertion d'un élément a , ou d'une nouvelle occurrence de l'élément a , dans un arbre 2-3 t , consiste à explorer l'arbre t à partir de la racine et à ajouter a dans le premier sommet de type 2 qui peut le contenir pour en faire un sommet de type 3. L'ajout de a ne doit pas modifier les sous-arbres de ce sommet. Si un tel sommet n'existe pas on crée une feuille de type 2 qui contiendra l'élément a (ou sa nouvelle occurrence).

Question 16. Ecrire une méthode persistante `static arbre23 insert23(int a, arbre23 t)` qui renvoie l'arbre 2-3 obtenu en insérant l'entier `a` dans l'arbre 2-3 `t`. On s'attachera à minimiser la complexité.

Réponse :

```
static arbre23 insert23(int a, arbre23 t){
    if (t==null){return new arbre23(a);}
    if (t.type==3){
        if (a<t.element[0]){
            arbre23 tt=insert23(a,t.fils[0]);
            return new arbre23(t.element[0],t.element[1],tt,t.fils[1],t.fils[2]);}
        if (a<=t.element[1]){
            arbre23 tt=insert23(a,t.fils[1]);
            return new arbre23(t.element[0],t.element[1],t.fils[0],tt,t.fils[2]);}
        arbre23 tt=insert23(a,t.fils[2]);
        return new arbre23(t.element[0],t.element[1],t.fils[0],t.fils[1],tt);
    }
    if (a<=t.element[0]){
        if (t.fils[0]==null) {return new arbre23(a,t.element[0],null,null,t.fils[1]);}
        return new arbre23(t.element[0],insert23(a,t.fils[0]),t.fils[1]);
    }
    if (t.fils[1]==null){return new arbre23(t.element[0],a,t.fils[0],null,null);}
    return new arbre23(t.element[0],t.fils[0],insert23(a,t.fils[1]));
}
```

Il existe un raffinement de la méthode précédente où le sommet est de type 2 et contient déjà une occurrence de l'entier `a`. Dans ce cas on rajoute `a` à ce sommet en en faisant un sommet de type 3 et on ajoute un sous-arbre du milieu mis à `null`. □

Question 17. Ecrire une méthode persistante `static arbre23 liftmax23(arbre23 t)` qui renvoie un arbre 2-3 qui contient les mêmes éléments (y compris leurs duplicats) de `t` mais avec l'un des plus grands éléments placé à la racine.

Réponse :

```
static arbre23 liftmax23(arbre23 t){
    if (t==null){return null;}
    if(t.fils[t.type-1]==null){return t;}
    arbre23 tt=liftmax(t.fils[t.type-1]);
    if (t.type==2){
        arbre 23 t2=new arbre23(t.element[0],t.fils[0],tt.fils[0]);}
    else{
        arbre 23 t2=new arbre23(t.element[0],t.element[1],t.fils[0],t.fils[1],tt.fils[0]);}
    if (tt.type==2){
        return new arbre23(tt.element[0],t2,null);}
    else{return new arbre23(tt.element[0],tt.element[1],t2,tt.fils[1],null);}
}
```

□

Question 18. Ecrire une méthode `static arbre23 cutright23(int a, arbre23 t)` qui renvoie l'arbre 2-3 obtenu en retirant de l'arbre `t` tous les éléments supérieurs ou égaux à l'entier `a`. On s'attachera à minimiser la complexité et le nombre d'éléments déplacés.

Réponse :

```
static arbre23 cutright23(int a, arbre23 t){
```

```

if (t==null){return t;}
if(t.element[0]>=a) return cutright23(a,t.fils[0]);
if(element[t.type-2]>a){
    arbre23 t2=cutright(a,t.fils[t.type-1]);
    if (t.type==2){return new arbre23(t.element[0],t.fils[0],t2);}
    else{return new arbre23(t.element[0],t.element[1],t.fils[0],t.fils[1],t2);}
}
return new arbre23(t.element[0],t.fils[0],cutright23(a,t.fils[1])
}

```

□

Partie V Complexité sur les arbres de recherche

On appelle R_n le nombre d'arbres binaires de recherche différents contenant n éléments différents qui sont les n premiers entiers.

Question 19. En posant la convention $R_0 = 1$, justifier la récurrence pour $n > 1$: $R_n = \sum_{k=1}^{k=n} R_{k-1}R_{n-k}$

Réponse : Pour un arbre binaire de recherche il y a un élément dans la racine et $n - 1$ dans les sous-arbres. Si l'élément à la racine est k ($1 \leq k \leq n$), alors, le sous-arbres de gauche contient les $k - 1$ premiers entiers et le sous-arbre de droite, les $n - k$ derniers entiers inférieur ou égaux à n . Alors le nombre d'arbres possible est $R_{k-1} \times R_{n-k}$ en fonction de comment s'arrangent les éléments dans le sous-arbre de gauche, ce qui se fait indépendamment de l'arrangement dans le sous-arbre de droite. On conclut en sommant sur toutes les valeurs de k □

Question 20. Calculer les valeurs de R_n pour les valeurs de n de 0 à 4. On constatera qu'elles sont égales à $\frac{(2n)!}{n!(n+1)!}$.

Réponse :

n	R_n
0	1
1	1
2	2
3	5
4	15

□

Question 21. En admettant l'identité $R_n = \frac{(2n)!}{n!(n+1)!}$, utiliser la formule de Stirling $n! \sim \sqrt{2\pi n}n^{n+\frac{1}{2}}e^{-n}$, pour donner un équivalent asymptotique de R_n quand n tend vers l'infini.

Réponse : Avec $(2n)! \sim \sqrt{4\pi n}2^{2n}n^{2n}e^{-n}$ on a

$$\begin{aligned}
 R_n &= \frac{(2n)!}{(n!)^2} \frac{1}{n+1} \\
 &\sim \frac{2^n}{\sqrt{\pi n}} \frac{1}{n+1} = O\left(\frac{2^n}{n^{\frac{3}{2}}}\right)
 \end{aligned}$$

□

Soit T_n le nombre d'arbres 2-3 différents contenant n éléments différents qui sont les n premiers entiers.

Question 22. Avec la convention $T_0 = 1$, trouver la récurrence pour $n > 2$ qui détermine T_n .

Réponse : La formule est

$$T_n = \sum_{k=1}^n T_{k-1}T_{n-k} + \sum_{1 \leq k < \ell \leq n} T_{k-1}T_{\ell-k-1}T_{n-\ell}$$

Le premier terme du membre de gauche compte les arbres 2-3 dont la racine est de type 2, et qui se décline comme dans la question précédente. Le deuxième terme compte les arbres 2-3 dont la racine est de type 3, les indices k et ℓ ($k < \ell$) donnent la valeur des éléments insérés dans la racine.

□