UV Informatique

Seuls les documents distribués en cours et les notes personnelles sont autorisés.

NOM :.....

Prénom:.....*Filière:*.....

Ensemble en C

On souhaite dans la suite utiliser un arbre binaire de recherche pour implémenter la notion d'ensemble au sens mathématique du terme. On se restreint à la modélisation d'un ensemble d'entiers. On souhaite notamment capturer le fait qu'un ensemble, en mathématique, ne comporte pas de doublon. La description explicite d'un ensemble s'effectue au moyen de la notation { } : ainsi *{ 1 , 2 , 4}* représente l'ensemble contenant les éléments *1*, *2* et *4*. Par ailleurs un certain nombre de fonctionnalités sont définies : *intersection* d'ensembles, *union*, *appartenance* d'un élément à un ensemble, *inclusion*, *produit cartésien* de 2 ensembles etc...

On fait donc le choix d'une arbre binaire de recherche (ABR) comme structure de données d'implémentation de cette notion d'ensemble. Celui-ci est construit classiquement à l'aide de la structure de noeud ci-dessous[1]:

```
struct Node {
    int val;
    Node *pfg,*pfd;
};
```

Pour construire un ensemble il est nécessaire de pouvoir lui ajouter des éléments. cela se traduit ici par la mise à disposition d'une fonction d'insertion :

```
Node * insere (int val,Node *pr) {
    if (pr==0) {
        Node *p=(Node *)malloc(sizeof(Node));
        p->val=val;
        p->pfg=p->pfd=0;
        return p;
    }
    else if (val < pr->val) { // a gauche
         pr->pfg=insere(val,pr->pfg);
         return pr;
    }
    else if (val > pr->val) { // a droite
        pr->pfd=insere(val,pr->pfd);
        return pr;
    }
    else return pr;
}
```

- **Comment cette procédure d'insertion exprime-t-elle le fait qu'un ensemble ne contient pas de doublon ?**

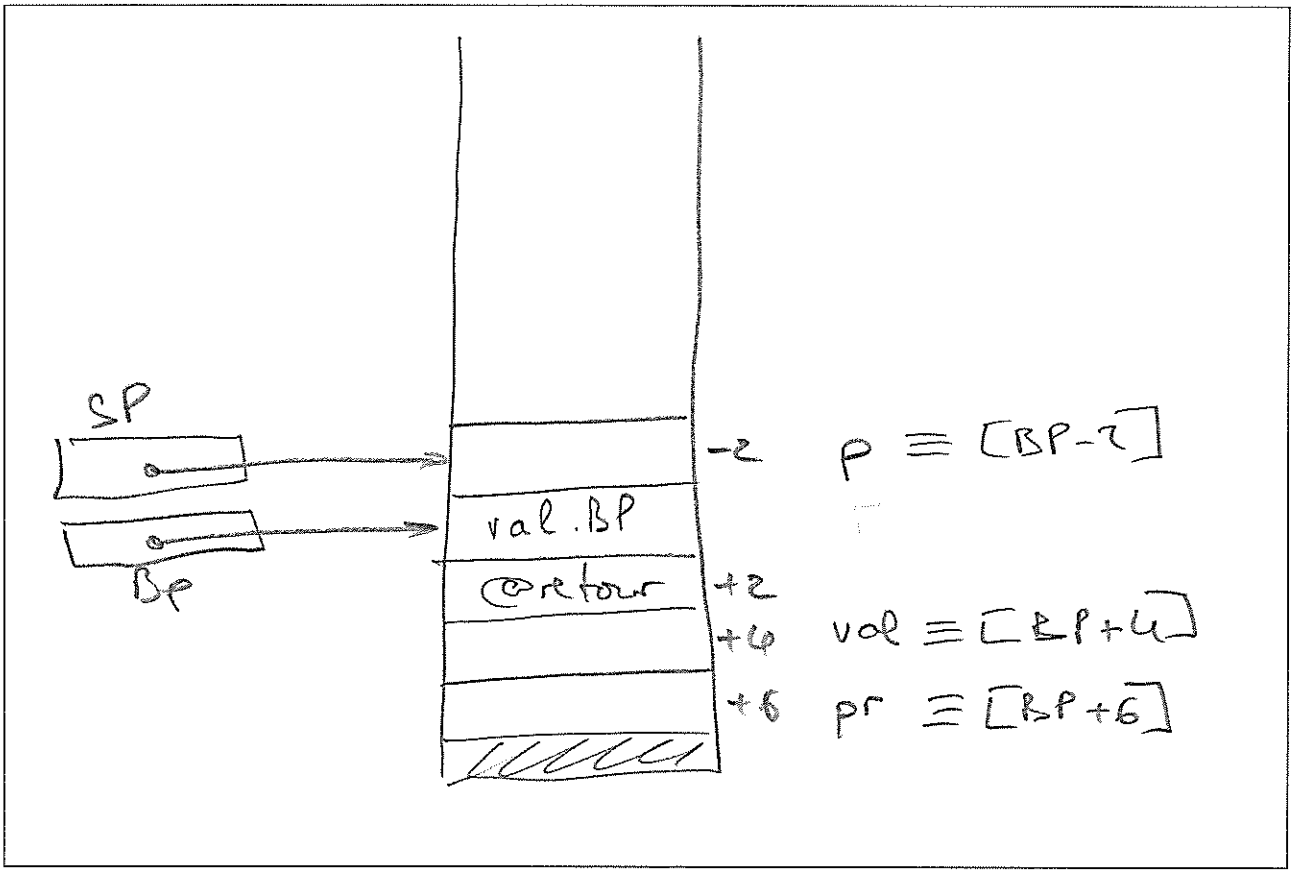
[1] Notation équivalente à :

```
typedef struct {
    int val;
    Node *pfg,*pfd;
} Node;
```

- **Ecrire la portion de code qui crée l'ensemble { 1 , 2 , 4 } , en s'appuyant sur la fonction insere ci-dessus, dans l'arbre repéré par la variable *pRoot*.**

```
Node * pRoot;

int main() {

  // à compléter...









}
```

- **En supposant que ce code est compilé pour la plate-forme *Intel 8086*, dessinez le contexte de pile de la fonction *insere*. Faites figurer notamment, les déplacements relatifs au registre *BP* caractérisant les variables locales et paramètres. On supposera un modèle mémoire où tous les pointeurs et adresses sont codés sur 2 octets.**



- **Imaginez le code assembleur que le compilateur pourrait générer pour les structures de contrôle de la fonction *insere*. Pour *p*, *pr* et *val*, on utilisera les déplacements relatifs à *BP* définis à la question précédente.**

```
    if (pr==0) {
            CMP  [BP+6], 0
            JNE  els0
            ...
            JMP  fsi0
    ...
} else if (val < pr->val) { // a gauche
els0:       MOV  BX, [BP+6]
            MOV  AX, [BP+4]
            CMP  AX, [BX]
            JGE  els1
            ...
            JMP  fsi0
    ...
} else if (val > pr->val) { // a droite
els1:       CMP  AX, [BX]
            JLE  els2
            ...
            JMP  fsi0
    ...
} else {
els2:       ...
}
fsi0:
```

- **De la même façon, imaginez le code généré pour la portion de code suivante :**

```
p=(Node *)malloc(sizeof(Node));
```

```
MOV  AX, 6
PUSH AX
CALL :malloc
POP  CX
MOV  [BP-2], AX
```

```
p->val=val;
```

```
MOV  AX, [BP+4]
MOV  BX, [BP-2]
MOV  [BX], AX
```

```
p->pfg=p->pfd=0;
```

```
MOV  AX, 0
MOV  [BX+2], AX
MOV  [BX+4], AX
```

ou bien :

```
MOV  [BX+2], 0
MOV  [BX+4], 0
```

```
return p;
```

```
MOV  AX, [BP-2]
JMP  finInsere
```

→ fin de la fonction avec l'épilogue habituel

On souhaite maintenant écrire la fonction *bool include(Node *p1, Node *p2)* qui correspond à la fonction ensembliste permettant de tester l'inclusion d'un ensemble (ici *p1*) dans un autre (ici *p2*).

Par exemple l'ensemble *{ 1 , 2 }* est inclus dans l'ensemble *{ 1 , 2 , 3 }*. On considère qu'un ensemble est également inclus dans lui-même (inclusion au sens large).

On propose la définition récursive d'une telle fonction appliquée à un *ABR* est la suivante :

- *l'arbre vide est inclus dans tout ensemble.*
- *un ABR1 noté <SAG1,RAC1,SAD1> est inclus dans une ABR2 si :*
  RAC1 appartient à ABR2
  ***et** SAG1 est inclus dans ABR2*
  ***et** SAD1 est inclus dans ABR2*

- **Expliquez en quoi cette définition est récursive :**

- **Ecrire, à partir de cette définition récursive, la fonction *C* implémentant cette relation d'inclusion**

```
bool include(Node *p1, Node *p2) {
   // retourne vrai si p1 est inclus dans p2
   // à compléter.....









}
```

On suppose maintenant que toutes les autres fonctions sont écrites; le fichier *Node.h* est donc le suivant :

```
struct Node {
    int val;
    Node *pfg,*pfd;
};
Node * insere (int val,Node *pr);
void GRD(Node *pr);     // parcours GRD (affichage)
bool isPresent(int val, Node *pr);
Node *inter(Node *p1,Node *p2); // construit en retour un nouvel ABR (p1 et p2
reste inchangés)
Node *unionSet[2](Node *p1,Node *p2); // idem ci-dessus
bool include(Node *p1, Node *p2);
```

Les foncions *inter*, *unionSet* et *include* correspondent respectivement à l'intersection, l'union et l'inclusion de 2 ensembles.

[2] le nom *union* qui aurait été naturel ici ne peut pas être employé car *union* est un mot réservé du C. On lui a substitué le mot *unionSet* pour désigner cette opération ensembliste.

- **Ecrire la portion de code de test qui réalise le travail suivant :**
  - **création de l'ensemble { 1 , 2 } (noté *pRoot1*)**
  - **parcours *GRD* de cet ensemble**
  - **création de l'ensemble { 2 , 4 , 5 } (noté *pRoot2*)**
  - **parcours *GRD* de cet ensemble**
  - **construction par invocation des fonctions ci-dessus de leur intersection (noté *pRootInter*)**
  - **parcours *GRD* de cette intersection**
  - **construction par invocation des fonctions ci-dessus de leur union (noté *pRootunion*)**
  - **parcours *GRD* de cette union**

  Le résultat à l'écran de ce travail est donc :
  1 2
  2 4 5
  2
  1 2 4 5

```
Node *pRoot1=0,*pRoot2=0,*pRootInter=0,*pRootUnion=0;

main () {
  // à compléter...









}
```

La classe TreeSet

On choisit maintenant d'exprimer au moyen d'une classe Java la notion d'ensemble, basé sur une structure d'arbre binaire, et exploré précédemment au moyen du langage C . La documentation au format javadoc de la classe *TreeSet*, du package *java.util*, est décrite dans l'annexe de ce sujet qui

servira de base documentaire pour les réponses aux premières questions. Comme toutes les classes conteneurs depuis Java 1.5 *TreeSet* est une classe générique. La généricité signifie qu'on peut spécifier le type des éléments contenus dans la structure. Ainsi *TreeSet<Integer>* est un conteneur d'entier, *TreeSet<String>* est un conteneur de *String*. Le nom de la classe générique, pourvu du type passé en paramètre, constitue le nom complet de la classe. La création d'une instance de *TreeSet* contenant des entiers s'effectue par exemple par:

TreeSet<Integer> ts=new TreeSet<Integer> ();

- **Pourquoi le nom de cette classe contient-elle le mot *Tree* ?**

- **Quelles sont les complexités des opérations de recherche d'un élément, d'ajout d'un élément, et de suppression d'un élément ?**

- **Quelle structure et quelle invariant de structure permettent d'atteindre une telle complexité ?**

La structure *TreeSet* présente une certain nombre de caractéristiques héritées d'interface ou de classes. Celle concernant *Set* implique qu'un *TreeSet* ne peut pas contenir de doublon (comme dans la notion mathématique d'ensemble) Celle concernant *SortedSet* implique que chaque élément soit pourvu d'une méthode (*compareTo)* permettant de le comparer à un autre élément.

- **Rappeler la raison fondamentale pour laquelle il est nécessaire que les éléments de cette structure soient comparables**

On souhaite étendre par héritage la classe *TreeSet<Integer>* pour implémenter une nouvelle classe (*IntegerSet*) destinée à la fourniture d'un certain nombre de services concernant les opérations ensemblistes (union, intersection etc...). La toute première étape consiste à écrire le code source qui suit (on ne fait pas figurer les clauses d'*import*)

- **Compléter la section pointillée**

```
public class IntegerSet extends TreeSet<Integer> {
```

```
        public IntegerSet() {


                ….....
        }
}
```

On souhaite mettre en place une méthode union dans cette classe pour implémenter la notion mathématique d'union[3] de deux ensembles.
Le test de cette fonction peut être le suivant :

```
public class Launcher {
        public static void main(String[] args) {
                // construction de l'ensemble set1
                IntegerSet  set1=new IntegerSet();
                for(int i =0;i<5;i++) set1.add(i);
                System.out.println("set1 : "+set1.size()+" items");
                System.out.print("{");
                for(Integer i:set1) System.out.print(i+" ");
                System.out.println("}");

                // construction de l'ensemble set2
                IntegerSet  set2=new IntegerSet();
                for(int i =3;i<8;i++) set2.add(i);
                System.out.println("set2 : "+set2.size()+" items");
                System.out.print("{");
                for(Integer i:set2) System.out.print(i+" ");
                System.out.println("}");

                // construction de l'ensemble (set1 union set2 )
                IntegerSet set3=set1.union(set2);
                System.out.println("set1 union set2 : "+set3.size()+" items");
                System.out.print("{");
                for(Integer i:set3) System.out.print(i+" ");
                System.out.println("}");

                // construction de l'ensemble (set1 inter set2 )
                IntegerSet set4=set1.inter(set2);
                System.out.println("set1 inter set2 : "+set4.size()+" items");
                System.out.print("{");
                for(Integer i:set4) System.out.print(i+" ");
                System.out.println("}");
        }
}
```

L'affichage produit par ce code est le suivant :

```
set1 : 5 items
{0 1 2 3 4 }
set2 : 5 items
{3 4 5 6 7 }
set1 union set2 : 8 items
{0 1 2 3 4 5 6 7 }
set1 inter set2 : 2 items
{3 4 }
```

- **Dans ce test la classe *IntegerSet* est sollicitée sur son constructeur, ainsi que sur les méthodes *add*, *size*, *union* et *inter*. Pourtant seules les méthodes *union* et *inter* devront être ajoutées à la classe *IntegerSet*. Pourquoi ?**

[3] On peut maintenant nommer *union* cette méthode car *union* n'est, contrairement au C, pas un mot réservé de Java.

On ajoute ci-dessous la méthode *union* à la classe *IntegerSet*. La méthode *union* y est explicitée

```
public class IntegerSet extends TreeSet<Integer> {
        public IntegerSet() {
                <... question précédente...>
        }
        public IntegerSet union(IntegerSet set) {
                IntegerSet setUnion = (IntegerSet) this.clone();
                setUnion.addAll(set);
                return setUnion;
        }
        public IntegerSet inter(IntegerSet set) {
                IntegerSet setInter = (IntegerSet) this.clone();


                ….....................................................

                return setInter;
        }
}
```

- **Expliquer le code de *union* sans le paraphraser**

- **Compléter la partie manquante de la méthode *inter***

```
        public IntegerSet inter(IntegerSet set) {
                IntegerSet setInter = (IntegerSet) this.clone();




                ….....................................................
```

```
            return setInter;
        }
```

- **On souhaite ajouter une méthode permettant de tester l'égalité de deux ensembles à la classe *IntegerSet*. Sa signature est : *public boolean equal(IntegerSet set)*[4] . Une implémentation paresseuse nous conduit à exploiter le résultat mathématique suivant: $A=B \Leftrightarrow card(A \cup B)=card(A \cap B)$. Implémenter la méthode *equal* en utilisant ce résultat.**

```
public boolean equal(IntegerSet set) {

        ….....................................

        return ....................

}
```

De même on ajoute la méthode *isIncluded* qui permet de tester l'inclusion d'un ensemble dans un autre. L'expression mathématique $A \cup B$ est représentée par l'expression Java *a.isIncluded(b)*. La signature de *isIncluded* est :
public boolean isIncluded(IntegerSet set)
Pour l'implémentation on utilise la méthode suivante : on itère à travers les entiers de *this* et dès qu'on trouve un entier non contenu dans *set* on retourne *false*. Si aucun élément de *this* n'a été trouvé comme non contenu dans *set* on retourne *true*.

- **Écrire le code de *isIncluded*.**

```
public boolean isIncluded(IntegerSet set) {

            ….....................................

            return ....................

}
```

On souhaite ajouter une redéfinition de *toString* dans la classe *IntegerSet* de façon à faciliter les tests.

```
public class Launcher {
        public static void main(String[] args) {
                ViewerSet vs;
                // construction de l'ensemble set1
                IntegerSet  set1=new IntegerSet();
                for(int i =0;i<5;i++) set1.add(i);
                // construction de l'ensemble set2
                IntegerSet  set2=new IntegerSet();
                for(int i =3;i<8;i++) set2.add(i);
                // isualisation
                System.out.println(set1);
                System.out.println(set2);
                System.out.println(set1.union(set2));
                System.out.println(set1.inter(set2));
        }
}
Ce code produit l'affichage suivant :
{0 1 2 3 4 }
```

[4]Il ne s'agit pas d'une redéfinition de *equals*.