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* 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 ?**

Lorsque val==pr->val aucun traitement n'est effectué donc dans ce cas aucun doublon n'est créé.

- **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*.**

[1] Notation équivalente à :

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

```
Node * pRoot;

int main() {
  // à compléter...
  pRoot=0;
  pRoot=insere(1,pRoot);
  pRoot=insere(2,pRoot);
  pRoot=insere(4,pRoot);
}
```

- **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) {




      ...
    }  else if (val < pr->val) { // a gauche




      ...
    }  else if (val > pr->val) { // a droite




      ...
    }  else {
      ...
    }
```

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

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









p->val=val;




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




return p;
```

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 :**

La défintion de *est inclus* s'appuie partiellement sur elle-même : 2 présences de *est inclus* dans la définition de *est inclus*.

- **Ecrire, à partir de cette définition récursive, la fonction *C* implémentant cette relation d'inclusion. On supposera qu'on dispose de la fonction *bool isPresent(int val, Node *pRoot)* permettant de tester la présence d'une valeur donnée dans l'arbre.**

```
bool include(Node *p1, Node *p2) {
   // retourne vrai si p1 est inclus dans p2
   // à compléter.....
  if (p1==0) return true;
  return isPresent(p1->val,p2) &&
      include(p1->pfg,p2) &&
      include(p1->pfd,p2);
}
```

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.

- **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*)

[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.

- 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...
  pRoot1=insere(1, pRoot1);
  pRoot1=insere(2, pRoot1);
  grd(pRoot1);
  pRoot2=insere(2, pRoot2);
  pRoot2=insere(4, pRoot2);
  pRoot2=insere(5, pRoot2);
  grd(pRoot2);
  pRootInter=inter(pRoot1,pRoot2);
  grd(pRootInter);
  pRootUnion=unionSet(pRoot1,pRoot2);
  grd(pRootUnion);
}
```

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* ?**

Parce que *javadoc* nous explique qu'en interne la classe est basé sur une structure d'arbre

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

log(n) log(n) log(n)

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

Arbre binaire de recherche.
Equilibré c'est à dire : ∀ SAG et SAD, abs(lg(SAG)-lg(SAD)<2)

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**

Un ABR est une arbre binaire dont les éléments sont ordonnés. La position d'un élément lors de son

insertion est d'abord déterminée par comparaison entre sa valeur et la valeur des noeuds vers lesquels il est orienté (elle peut varier ensuite si il y a un rééquilibrage, mais l'invariant val(SAG) < val< val (SAD) est maintenu quelque soit le noeud). Cette valeur doit donc être comparable.

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 réduite à la déclaration de la classe et de son constructeur (on ne fait pas figurer les clauses d'*import*) sans chercher à ce stade à placer les méthodes. ***Pour les questions qui suivent vous aurez, éventuellement, à vous référer à l'annexe concernant TreeSet***

- **Compléter la section pointillée**

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


            super();
      }
}
```

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

```
{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 ?**

add et *size* sont héritées de *TreeSet<Integer>* et conviennent tels quels donc aucune redéfinition n'est nécessaire pour ces deux méthodes.
Le constructeur a lui déjà été ajouté.
Union et inter n'existent pas dans *TreeSet* elles devront donc être ajoutées à *IntegerSet*

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**

Pour réaliser l'union de deux ensembles on construit un nouvel ensemble. Pour cela on duplique l'ensemble courant (*this.clone()*) , puis on ajoute à ce duplicata tous les éléments de l'autre ensemble (*set*) . Comme on sait qu'il n'y aura pas de doublons seuls les éléments manquant dans l'ensemble initial seront effectivement ajoutés.

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

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

	for(int i:this)
		if (! set.contains(i)) setInter.remove(i);
	return setInter;

	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.**

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

```
public boolean equal(IntegerSet set) {


        return this.union(set).size()==this.inter(set).size();
//          // ou return union(set).size()==inter(set).size();

}
```

De même on ajoute la méthode *isIncluded* qui permet de tester l'inclusion d'un ensemble dans un autre. L'expression mathématique $A \subseteq B$ est représentée par l'expression Java *a.isIncluded(b)*. La signature de la méthode *isIncluded* dans *IntegerSet* 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 absent de *set* on retourne *true*.

- **Écrire le code de *isIncluded*.**

```
public boolean isIncluded(IntegerSet set) {
     boolean isIncluded=true;
     for(int i:this)
           if (!set.contains(i)) return false;
      return true;

}
```

- **Proposer une définition de *equal* alternative à la précédente et basée sur *isIncluded***

```
public boolean equal(IntegerSet set) {
        return this.isIncluded(set) && set.isIncluded(this);

}
```

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 }
{3 4 5 6 7 }
{0 1 2 3 4 5 6 7 }
{3 4 }
```

- **Pourquoi parle-t-on de redéfinition de *toString* et pas simplement de définition? Où *toString* est-il déjà définie?**

La méthode *toString* est définie dans la classe concrète *Object,* ancêtre de toutes les classes. De ce fait toutes les classes bénéficie de cette méthode qu'elles peuvent redéfinir au besoin.

- **Écrire la méthode *toString* de la classe *IntegerSet* qui produit le résultat ci-dessus**

```
@Override
public String toString() {
      String s="{";
      for(Object o:this) s+=o+" ";
      s+="}";
      return s;
}
```

On décide de tenir compte maintenant de l'universalité des opérations ensemblistes impliquées : elles ne sont pas spécifiques aux ensembles d'entiers mais ont du sens pour n'importe quel ensemble. On décide d'extraire dans une interface les opérations principales par *refactoring*. Cette interface est nommée *ISet.* Au format *JavaDoc* cette interface se présente ainsi :

Interface ISet

`public interface` **`ISet`**

Method Summary	
boolean	**equal**(ISet set)
java.lang.Object[]	**getList**()
ISet	**inter**(ISet set)
boolean	**isIncluded**(ISet set)
ISet	**union**(ISet set)

La classe *IntegerSet* après avoir été modifié pour supporter cet héritage (notamment par l'adjonction de *getList* qui permet de disposer d'un moyen de récupérer les éléments contenus dans un ensemble sous la forme d'un tableau) se présente ainsi :

Class IntegerSet

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.TreeSet<java.lang.Integer>
              └ IntegerSet
```

All Implemented Interfaces:

ISet, java.io.Serializable, java.lang.Cloneable, java.lang.Iterable<java.lang.Integer>, java.util.Collection<java.lang.Integer>, java.util.NavigableSet<java.lang.Integer>, java.util.Set<java.lang.Integer>, java.util.SortedSet<java.lang.Integer>

`public class` **`IntegerSet`**`extends java.util.TreeSet<java.lang.Integer>implements ISet`

Constructor Summary	
IntegerSet()	

Method Summary	
boolean	**equal**(ISet set)
java.lang.Object[]	**getList**()

IntegerSet	**inter**(ISet set)
boolean	**isIncluded**(ISet set)
IntegerSet	**union**(ISet set)

Methods inherited from class java.util.TreeSet
add, addAll, ceiling, clear, clone, comparator, contains, descendingIterator, descendingSet, first, floor, headSet, headSet, higher, isEmpty, iterator, last, lower, pollFirst, pollLast, remove, size, subSet, subSet, tailSet, tailSet

Methods inherited from class java.util.AbstractSet
equals, hashCode, removeAll

Methods inherited from class java.util.AbstractCollection
containsAll, retainAll, toArray, toArray, toString

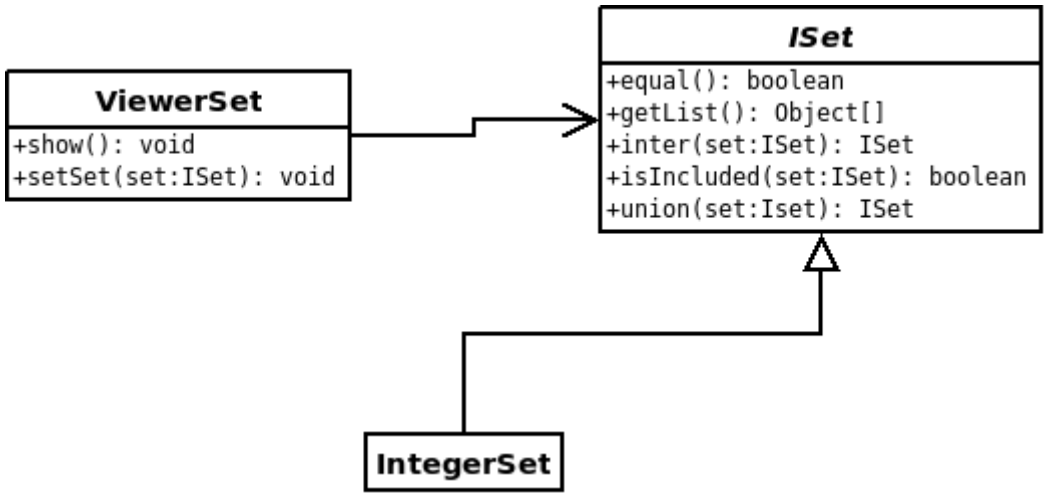
Methods inherited from class java.lang.Object
getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Set
containsAll, equals, hashCode, removeAll, retainAll, toArray, toArray

- **En quoi cette opération de *refactoring* peut-elle être intéressante pour la suite des développements?**

On a intérêt chaque fois que possible d'abstraire dans une interface les caractéristiques essentielles d'une classe. Cela permettra de rendre moins contraignant le couplage entre cette classe et les architectures d'objets dans lesquelles elle est impliquée. Le couplage faible des composants entre eux est une préoccupation permanente en orienté objet.

Dans le cadre d'une application pédagogique destinée à permettre à l'utilisateur de mettre en oeuvre graphiquement des opérations ensemblistes simples, on appelle *ViewerSet* l'entité de l'application chargée de permettre une visualisation graphique de l'ensemble. On souhaite valider ce choix d'architecture en implémentant dans un premier temps un *viewer* d'ensemble en mode console. L'architecture visée est la suivante :



Dans cette phase de test d'architecture la méthode *show* se contente à ce stade d'afficher la liste des éléments encadrée par un jeu d'accolades. Un exemple de mis en oeuvre de cette classe :

```
public static void main(String[] args) {
```

```
            ViewerSet vs=new ViewerSet();
            // 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);
            // Visualisation
            vs.setSet(set1);
            vs.show();
            vs.setSet(set2);
            vs.show();
            vs.setSet(set1.union(set2));
            vs.show();
            vs.setSet(set1.inter(set2));
            vs.show();
      }
}
```

Le résultat sur la console est le suivant :

```
{0 1 2 3 4 }
{3 4 5 6 7 }
{0 1 2 3 4 5 6 7 }
{3 4 }
```

- **Ecrire complètement la classe ViewerSet (son constructeur, ses deux méthodes)**

```
public class ViewerSet {
      private ISet set;
      public ViewerSet() {

      }
      public void show() {
            System.out.print("{");
            for(Object o:set.getList()) System.out.print(o+" ");
            System.out.println("}");
      }
      public void setSet(ISet set) {
            this.set=set;
      }
}
```

- **Dans l'exemple *ViewerSet* travaille avec la classe *IntegerSet*. *ViewerSet* peut-elle travailler avec des ensembles contenant autres choses que des entiers ? A quelle(s) condition(s)?**

Oui à condition que la classe en question implémente (au sens de *hérite de*) *ISet*

- ***De même ViewerSet* peut-elle travailler avec des ensembles non fondés sur *TreeSet<E>* ?**

Oui et à la même condition. Il y a aucun couplage entre TreeSet et ViewerSet

- **Dans l'affirmative proposer une classe minimale *StringSet* compatible avec *ViewerSet* et non basée sur *TreeSet* et permettant de construire un ensemble de chaînes de caractères (vous pouvez utiliser un tableau en interne par exemple)**

```
class StringSet implements Iset {
    …. doivent figurer :
    constructeur
    méthode pour ajouter des éléments
    l'implémentation des 5 méthodes héritées de ISet
```

```
}
```

java.util
Class TreeSet<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ **java.util.TreeSet<E>**

Type Parameters:
E - the type of elements maintained by this set

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, NavigableSet<E>, Set<E>, SortedSet<E>

```
public class TreeSet<E>extends AbstractSet<E>implements NavigableSet<E>, Cloneable,
Serializable
```

A `NavigableSet` implementation based on a `TreeMap`. The elements are ordered using their natural ordering, or by a `Comparator` provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (`add`, `remove` and `contains`).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be *consistent with equals* if it is to correctly implement the `Set` interface. (See `Comparable` or `Comparator` for a precise definition of *consistent with equals*.) This is so because the `Set` interface is defined in terms of the `equals` operation, but a `TreeSet` instance performs all element comparisons using its `compareTo` (or `compare`) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Set` interface.

Note that this implementation is not synchronized. If multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the `Collections.synchronizedSortedSet` method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));
```

The iterators returned by this class's `iterator` method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:
1.2
See Also:
`Collection`, `Set`, `HashSet`, `Comparable`, `Comparator`, `TreeMap`, Serialized Form

Constructor Summary
`TreeSet()` Constructs a new, empty tree set, sorted according to the natural ordering of its elements.

TreeSet(Collection<? extends E> c) Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.	
TreeSet(Comparator<? super E> comparator) Constructs a new, empty tree set, sorted according to the specified comparator.	
TreeSet(SortedSet<E> s) Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.	

Method Summary	
boolean	**add**(E e) Adds the specified element to this set if it is not already present.
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set.
E	**ceiling**(E e) Returns the least element in this set greater than or equal to the given element, or `null` if there is no such element.
void	**clear**() Removes all of the elements from this set.
Object	**clone**() Returns a shallow copy of this `TreeSet` instance.
Comparator<? super E>	**comparator**() Returns the comparator used to order the elements in this set, or `null` if this set uses the natural ordering of its elements.
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
Iterator<E>	**descendingIterator**() Returns an iterator over the elements in this set in descending order.
NavigableSet<E>	**descendingSet**() Returns a reverse order view of the elements contained in this set.
E	**first**() Returns the first (lowest) element currently in this set.
E	**floor**(E e) Returns the greatest element in this set less than or equal to the given element, or `null` if there is no such element.
SortedSet<E>	**headSet**(E toElement) Returns a view of the portion of this set whose elements are strictly less than `toElement`.
NavigableSet<E>	**headSet**(E toElement, boolean inclusive) Returns a view of the portion of this set whose elements are less than (or equal to, if `inclusive` is true) `toElement`.
E	**higher**(E e) Returns the least element in this set strictly greater than the given element, or `null` if there is no such element.
boolean	**isEmpty**() Returns `true` if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set in ascending order.
E	**last**() Returns the last (highest) element currently in this set.
E	**lower**(E e) Returns the greatest element in this set strictly less than the given element, or `null` if there is no such element.
E	**pollFirst**() Retrieves and removes the first (lowest) element, or returns `null` if this set is empty.
E	**pollLast**()

	Retrieves and removes the last (highest) element, or returns `null` if this set is empty.
boolean	**remove**(Object o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).
NavigableSet<E>	**subSet**(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) Returns a view of the portion of this set whose elements range from `fromElement` to `toElement`.
SortedSet<E>	**subSet**(E fromElement, E toElement) Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.
SortedSet<E>	**tailSet**(E fromElement) Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.
NavigableSet<E>	**tailSet**(E fromElement, boolean inclusive) Returns a view of the portion of this set whose elements are greater than (or equal to, if `inclusive` is true) `fromElement`.

Methods inherited from class java.util.AbstractSet
equals, hashCode, removeAll

Methods inherited from class java.util.AbstractCollection
containsAll, retainAll, toArray, toArray, toString

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Set
containsAll, equals, hashCode, removeAll, retainAll, toArray, toArray