Introducción a la Programación

Tema 17. Implementación de tipos

En este capítulo vamos a ver más detalles para implementar tipos. Recordaremos las ideas fundamentales para diseñar la funcionalidad de un tipo e implementarlo. Introduciremos en concepto de requisitos no funcionales. Y como tarea central de este capítulo estudiaremos un conjunto de tipos básicos que nos servirán como elementos de una caja de herramientas adecuada para construir e implementar nuevos tipos. Estos tipos básicos tienen, en general, como finalidad servir para construir tipos nuevos más que ser utilizados directamente. Entre ellos veremos los árboles y sus subtipos, las tablas hash, los arrays de bits, las listas enlazadas, etc.

1. Introducción

Recordemos, en primer lugar, las ideas ya vistas en capítulos anteriores sobre diseño e implementación de tipos. En primer lugar veamos las ideas relacionadas con el diseño de tipos, después las que tienen que ver con la implementación de tipos y por último la forma de comprobar que una implementación cumple las restricciones del tipo.

1.1 Diseño de tipos

Un buen diseño de tipos es básico para que los programas sean comprensibles y fáciles de mantener. Veamos algunas pautas para este diseño y algunos ejemplos que puedan servir de guía.

Al diseñar un tipo nuevo debemos partir de los ya existentes. Es necesario decidir a qué otros tipos extender. El nuevo tipo puede usar los tipos disponibles para declarar variables parámetros formales, etc. Decimos que el nuevo tipo **usa** esos tipos. También puede diseñarse el nuevo tipo extendiendo algunos de los disponibles. En este caso decimos que el nuevo tipo es un **subtipo** de los tipos de los que hereda o también decimos que **refina** esos tipos.

Todo tipo tiene, además de las heredadas de los tipos que refina, unas propiedades y posiblemente unas operaciones nuevas. Cada propiedad tiene un nombre, un tipo, puede ser **consultada** y además **modificada** o sólo consultada, y puede ser una **propiedad simple** o una **propiedad derivada**. Además las propiedades pueden ser **individuales** y **compartidas**. Cada tipo tiene una población. La **población de un tipo** es el conjunto de objetos que podemos crear de ese tipo. Como vimos en el capítulo 1 las propiedades individuales son específicas de un objeto individual. Las propiedades compartidas son comunes a todos los objetos de la población del tipo. Las propiedades derivadas pueden ser calculadas a partir de las otras propiedades. Las simples o básicas no. Las propiedades son usualmente consultables y pueden ser también modificables.

Las propiedades pueden tener parámetros y una precondición. Una precondición es una expresión lógica que indica en qué condiciones es posible obtener el valor de la propiedad.

Según sean modificables o sólo consultables, deduciremos un conjunto de métodos. De las operaciones deduciremos otro conjunto de métodos. Con todos ellos definiremos un nuevo tipo. Este nuevo tipo lo podemos crear mediante una interfaz y posteriormente implementarlo mediante una clase o directamente mediante la parte pública de una clase.

1.2 Contrato de un tipo y casos de prueba relacionados

Como hemos visto en capítulos anteriores para cada tipo es conveniente definir un **contrato**. Un **contrato** es un documento en el que se define la **funcionalidad** ofrecida por **un tipo** y por lo tanto el uso del mismo por parte de sus posibles clientes. Los detalles de cómo definir un contrato ya los vimos en capítulos anteriores.

El conjunto de casos de prueba se presenta como una tabla. Cada caso de prueba da lugar a un test. Es decir, una prueba del funcionamiento adecuado de un método en un caso concreto.

En general entendemos por contrato de un tipo tanto el conjunto de aserciones sobre los métodos del tipo (invariantes, precondiciones, postcondiciones, condiciones de disparo de excepciones, etc.) como el conjunto de casos de prueba adicionales. Estos casos de prueba adicionales pueden ser redundantes con las aserciones o contemplar casuísticas no tenidas en cuenta en las mismas. Un contrato será más completo que otro si (entre las aserciones y los casos de prueba) tiene en cuenta más casuísticas de funcionamiento. De las aserciones de un contrato pueden ser deducidos casos de prueba adicionales.

El conjunto de casos de prueba para un método debe capturar lo mejor posible la casuística de utilización del método. Los casos de prueba incluirán los ejemplos de funcionamiento incluidos en el contrato. Además incluirán otros casos de prueba deducidos de las restricciones del contrato.

1.3 Implementación de un tipo

Para implementar la funcionalidad expresada por un contrato en Java, debemos construir una clase que será la implementación del contrato. Una implementación es una relación entre un **contrato** y una **clase** concreta. La clase deberá cumplir el contrato definido para un tipo dado. Para comprobar que una clase implementa un contrato se generarán un conjunto de casos de prueba.

Como se ha dicho, un tipo se implementará mediante una clase. En esta debemos definir los atributos, constructores y métodos (públicos y privados) y para ello debemos tomar algunas decisiones.

Al implementar el estado y decidir el número de atributos:

- ¿Cuántos atributos y de qué tipos? Una primera idea es poner un atributo por cada propiedad no derivada. Cada atributo tiene como nombre el de la propiedad pero empezando por minúscula y el tipo de la misma. Si la propiedad asociada es

compartida por toda la población del tipo entonces el atributo llevará el modificador *static.*

- El estado de los objetos debe representar de forma mínima las propiedades de los objetos. Esto quiere decir que tenemos que implementar una forma concreta de guardar las propiedades en los atributos de la forma más simple y eficiente posible. Es lo que denominamos **forma normal** de los valores de ese tipo. Elegir adecuadamente esta **forma normal** es muy importante, tal como hemos visto anteriormente, para implementar la igualdad, los métodos *hashCode, toString* y el orden natural. Desde este punto de vista escogemos la forma normal para que guarde los valores del representante canónico de cada una de las clases de equivalencia definidas por la igualdad.

Al implementar los constructores:

- ¿Cuántos constructores? Uno con todos los datos suficientes para dar valor a los atributos, otro sin parámetros y en muchos casos uno que tome un *String* como parámetro. En algunos casos puede ser interesante algún constructor más.
- Los constructores deben disparar excepciones si no pueden construir un objeto en un estado válido con los parámetros dados. Por ejemplo, si un constructor *RacionalImpl* toma un parámetro que obligue a hacer cero el denominador.
- Los constructores deben dar valor a todos los atributos.
- Debemos tener en cuenta el **estado inicial** y el **invariante** del contrato (deben cumplirse al finalizar la ejecución del constructor).
- Si no puede garantizar el cumplimiento de ambas restricciones ante unos parámetros determinados, se debe lanzar una **excepción**.
- Es recomendable que el constructor dé un valor inicial a cada uno de los atributos.

Al implementar los métodos:

- Debemos respetar las signaturas establecidas en la interfaz.
- Debemos tener en cuenta la precondición, postcondición e invariante. Debe asegurarse que si se cumple la precondición, se cumplirán tras la ejecución del método tanto la postcondición como la invariante.
- Si se verifica alguna condición de disparo de excepción, entonces hay que disparar las excepciones definidas en el contrato.
- En la zona no definida en el contrato el método puede ser implementado libremente.

1.4 Requisitos no funcionales de un tipo

Dado un contrato de un tipo que exprese las propiedades funcionales del mismo es posible encontrar varias implementaciones posibles. Normalmente cuando queremos usar un tipo junto a las propiedades funcionales del mismo, dependiendo del contexto, es conveniente exigir unas determinadas propiedades no funcionales para el mismo. Algunas de estas propiedades no funcionales son la complejidad de los métodos del tipo. Es decir una medida del tiempo de respuesta del método en función del tamaño del objeto a considerar.

Para caracterizar una implementación concreta desde un punto de vista externo debemos añadir, junto a la funcionalidad del tipo, la complejidad de sus operaciones.

2. Tipos básicos

Presentamos aquí varios tipos que sirven usualmente para implementar muchos de los tipos complejos vistos previamente. Por ello es necesario conocerlos. Unos vienen proporcionados en el API de Java y otros no. Por motivos didácticos para algunos tipos básicos que ya vienen proporcionados en el entorno Java diseñaremos otra versión básica que sólo contenga los aspectos esenciales. Estas versiones básicas las nombraremos empezando por *Basic*. Por cada tipo presentaremos también un posible constructor o factoría del mismo.

2.1 Tipo BasicLinkedList

Es un tipo de datos básico a partir del cual pueden implementarse otros más complejos. Se suele denominar lista enlazada. El tipo es genérico y puede definirse recursivamente. Una lista enlazada es una lista vacía o un elemento seguido de una lista que llamaremos cola. Una lista enlazada no vacía tiene un primer y un último elemento. Por lo tanto las propiedades esenciales del tipo *BasicLinkedList<E>* son:

- *Empty: boolean*, la lista está vacía, consultable.
- *size(): int,* tamaño de la lista
- *get(int i): E*, elemento en posición i, precondición *i>=0 , i < size(),* y si no se cumple la precondición se dispara la excepción *NoSuchElementException.*
- *set(int index, E e): E,* modifica el elemento en posición i, precondición *i>=0 , i < size(),* y si no se cumple la precondición se dispara la excepción *NoSuchElementException.*

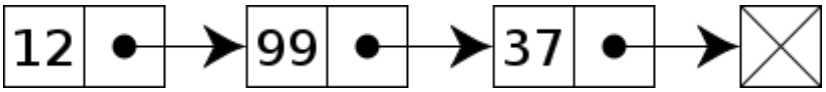
Y las operaciones:

- *add(E e): boolean,* añade *e* al final de la lista y por lo tanto e se convierte en el último elemento de la lista.
- *add(int index, E e): void,* añade el elemento e en la posición index y desplaza a la derecha los elementos en posiciones mayores.
- *remove(int index): E,* elimina el elemento

Constructor:

- *BasicLinkedList():* Construye una lista vacía.

Un ejemplo de lista enlazada de enteros es:



El primer elemento es 12, el último 37, etc. Tal como se ve en el ejemplo anterior una lista enlazada es una secuencia enlazada de entradas. Cada entrada es de un tipo privado que llamaremos *BasicLinkedList.Entry<E>.* Este tipo consta de dos propiedades consultables y modificables: *Element* de tipo *E* para guardar un elemento y *Next* de tipo *BasicLinkedList.Entry<E>* para guardar la entrada siguiente. El tipo *BasicLinkedList* puede implementarse con dos atributos: *first* y *last* de tipo *Entry<E>* que son ambos *null* o guardan el primer y último entrada de la lista. Una lista vacía puede implementarse con ambos atributos a *null*. Podemos añadir, por eficiencia otros atributos redundantes como *size*. Con esa implementación las propiedades *Empty*, *size()* y la operación *add(E)* tienen una complejidad constante. Sin embargo, como puede comprobarse, las operaciones *add(int,E)*, *get(int)* y *remove(int)* tienen complejidades lineales en el tamaño de la lista. Una posible implementación sería:

```
public class BasicLinkedList<E> {
	private Entry<E> first;
	private Entry<E> last;
	private int size;
	//invariant size == 0   => first == null && last == null
	//invariant size > 0    => first != null && last != null


	public BasicLinkedList() {
		super();
		this.first = null;
		this.last = null;
		this.size=0;
	}

	public int size() {
		return size;
	}

	public boolean isEmpty(){
		return size==0;
	}

	public E get(int index){
		return entryInPos(index).getElement();
	}

	public E set(int index, E e){
		Entry<E> e1 = entryInPos(index);
		E r = e1.getElement();
		e1.setElement(e);
		return r;
	}

	public boolean add(E e){
		Entry<E> e1 = new Entry<E>(e);
		if(last==null){
			first = e1;
			last=e1;
```

```
		}else{
			last.setNext(e1);
			last = e1;
		}
		size++;
		return true;
	}

	public void add(int index, E e){
		Preconditions.checkPositionIndex(index, size);
		Entry<E> ne = new Entry<E>(e);
		if(index==size){
			add(e);
		} else if(index==0){
			ne.setNext(first);
			first = ne;
		} else {
			Entry<E> pe = entryInPos(index-1);
			ne.setNext(pe.getNext());
			pe.setNext(ne);
		}
		size++;
	}

	private Entry<E> entryInPos(int index){
		Preconditions.checkElementIndex(index, size);
	    Entry<E> pe = first;
	    for(int p = 0 ; p < index; p++){
		  pe = pe.getNext();
	    }
		  return pe;
	}

	public E remove(int index){
		Preconditions.checkElementIndex(index, size);
		Entry<E> e = null;
		E element;
		if(index==0){
			e = first;
			first = first.getNext();
			element = e.getElement();
		} else {
			Entry<E> pe = entryInPos(index-1);
			element = pe.getNext().getElement();
			if(index == size-1){
				last = pe;
			}else{
				pe.setNext(pe.getNext().getNext());
			}
		}
		size--;
		return element;
	}

	public String toString(){
		String s = "{";
		boolean prim = true;
		for(Entry<E> e = first;e!=null;e = e.getNext()){
			if(prim){
				prim = false;
```

```
                        s = s+e.getElement();
                    }else{
                        s = s+","+e.getElement();
                    }
                }
                s = s+"}";
                return s;
        }

        public class Entry<F> {
                private F element;
                private Entry<F> next;
                public Entry(F element, Entry<F> next) {
                        super();
                        this.element = element;
                        this.next = next;
                }
                public Entry(F element) {
                        super();
                        this.element = element;
                        this.next = null;
                }
                public F getElement() {
                        return element;
                }
                public void setElement(F element) {
                        this.element = element;
                }
                public Entry<F> getNext() {
                        return next;
                }
                public void setNext(Entry<F> next) {
                        this.next = next;
                }
        }
}
```

Como vemos la implementación mantiene un **invariante de la implementación.** Es decir una restricción entre el valor de los atributos elegidos. En este caso si *size ==0* (lista vacía) los atributos *first* y *last* deben estar a *null* y distintos de null en caso contrario. Se añade por eficiencia el atributo size para implementar la propiedad derivada correspondiente.

El tipo *LinkedList*, ofrecido por *Java*, es una implementación del tipo *List* basándose en el tipo *BasicLinkedList* anterior. El tipo *LinkedList* implementa, además de *List*, otros tipos que no comentaremos ahora aquí.

Dejamos como ejercicio los detalles del tipo *LinkedList* y el cálculo de las complejidades de cada uno de los métodos asumiendo que la implementación se basa en *BasicLinkedList*.

2.2 Tipo DynamicArray

Un **dynamic array** de tipo *E* es un tipo de datos similar a un array de tipo *E* pero dónde podemos cambiar el tamaño en tiempo de ejecución. Este cambio del tamaño en tiempo de ejecución es una operación costosa pero el tipo ofrece otros otras propiedades interesantes.

Las propiedades y operaciones básicas del tipo son:

- *Empty: boolean*, la lista está vacía, consultable.
- *size(): int,* tamaño de la lista
- *get(int i): E*, elemento en posición i, precondición *i>=0 , i < size(),* y si no se cumple la precondición se dispara la excepción *NoSuchElementException.*
- *set(int index, E e): E,* modifica el elemento en posición i, si i es mayor que la capacidad la aumenta y rellena los huecos a *null*. El nuevo tamaño es *index + 1*.

Y las operaciones:

- *add(E e): void,* añade *e* al final de la lista y por lo tanto e se convierte en el último elemento de la lista.
- *add(int index, E e): void,* añade el elemento e en la posición index y desplaza a la derecha los elementos en posiciones mayores.
- *remove(int index): E,* elimina el elemento

Y los constructores:

- *DynamicArray(int capacity):* Construye un nuevo capacidad *capacity* y todas las casillas a *null*.
- *DynamicArray(int capacity, DynamicArray<E> d):* Construye un nuevo array dinámico con capacidad *capacity*, copia los elementos de *d* en las primeras casillas y pone el resto a *null*. Tiene como precondición que capacity sea mayor que la propiedad *Capacity* de *d*. Si no se cumple la precondición se dispara la excepción *IllegalArgumentException.*

Una implementación posible para este tipo básico puede ser:

```
public class DynamicArray<E> {

	private int capacity;
	private int size;
	private E[] elements;
	private final int INITIAL_CAPACITY = 10;
	private final int GROWING_FACTOR = 2;

	public DynamicArray() {
		super();
		this.capacity = INITIAL_CAPACITY;
		this.size = 0;
		this.elements = null;
	}

	public DynamicArray(int capacity) {
		super();
		Preconditions.checkArgument(capacity>0);
		this.capacity = capacity;
		this.size = 0;
		this.elements = null;
```

```
	}

	public DynamicArray(DynamicArray<E> a) {
		super();
		this.capacity = a.capacity;
		this.size = a.size();
		this.elements = Arrays.copyOf(a.elements,a.capacity);
	}

	public DynamicArray(E[] a) {
		super();
		this.capacity = a.length;
		this.size = capacity;
		this.elements = Arrays.copyOf(a, capacity);
	}

	private void grow(int newCapacity){
		E[] oldElements = elements;
		capacity = newCapacity;
		elements = Arrays.copyOf(oldElements, capacity);
	}

	public int size() {
		return size;
	}

	public boolean isEmpty(){
		return size == 0;
	}

	public E get(int index) {
		Preconditions.checkElementIndex(index, size);
		return elements[index];
	}

	@SuppressWarnings("unchecked")
	public E set(int index, E e){
		if(size == 0){
			if(index >= capacity){
				capacity = index+1;
			}
			elements = (E[]) Array.newInstance(e.getClass(), capacity);
		}
		if(index >=  capacity){
			grow(index+1);
		}
		if(index >= size){
			size = index+1;
		}
		E r = get(index);
		elements[index]= e;
		return r;
	}

	@SuppressWarnings("unchecked")
	public boolean add(E e) {
		if(size == 0){
			elements = (E[]) Array.newInstance(e.getClass(), capacity);
		}
		if(size==capacity){
```

```
                grow(capacity*GROWING_FACTOR);
            }
            elements[size] = e;
            size++;
            return true;
        }

        public void add(int index, E e) {
            Preconditions.checkPositionIndex(index, size);
            add(e);
            // size ya ha quedado aumentado
            for(int i = size-1; i > index; i--){
                elements[i]= elements[i-1];
            }
            elements[index]=e;
        }

        public E remove(int index) {
            Preconditions.checkElementIndex(index, size);
            E e = elements[index];
            for(int i = index; i < size-1; i++){
                elements[i]= elements[i+1];
            }
            size --;
            return e;
        }

        public E[] toArray(){
            E[] r = Arrays.copyOf(elements, size);
            return r;
        }

        public String toString(){
            String s = "{";
            boolean prim = true;
            for(int i=0; i< size; i++){
                if(prim){
                    prim = false;
                    s = s+elements[i];
                }else{
                    s = s+","+elements[i];
                }
            }
            s = s+"}";
            return s;
        }
}
```

Como vemos las principales dificultades están en la creación de un array genérico de forma dinámica. Esto lo hacemos al conocer la clase del primer elemento a añadir a la lista y usando la clase *Array* de Java y las posibilidades reflexivas del lenguaje. La otra dificultad es la sustitución del array que contiene los elementos por otro de mayor capacidad cuando el tamaño es igual a la capacidad y queremos añadir un elemento nuevo. Esto lo resolvemos con el método adecuado de la clase Arrays

El tipo *ArrayList*, ofrecido por *Java*, es una implementación del tipo *List* basándose en el tipo *BasicArrayList* anterior. El tipo *ArrayList* implementa, además de *List*, otros tipos que no comentaremos ahora aquí.

Dejamos como ejercicio los detalles del tipo *ArrayList* y el cálculo de las complejidades de cada uno de los métodos asumiendo que la implementación se basa en *BasicArrayList*.

2.3 Tipo Arrays de Bits (BitSet)

El tipo **Arrays de Bits** (Bitset) sirve para gestionar un vector de bits que puede crecer cuando se le necesite. Es un tipo básico ofrecido por el entorno de Java. Esencialmente los valores del tipo son secuencias de bits. Cada bit representa un valor booleano y puede ser indexado por un entero no negativo *{0,1,2,...,n-1}*. El valor de cada bit puede ser consultado, modificado o combinado con otro bit mediante las operaciones lógicas *NOT, AND, OR* inclusivo y OR exclusivo. Desde otro punto de vista un BitSet puede considerarse como la definición de un conjunto de enteros en el rango *[0, n)* donde n es el tamaño (*size*). Es decir el número de bits que se usan en cada momento. El conjunto definido por cada valor viene dado por los bits puestos a true. El conjunto vació viene representado por todos los bits a *false*. La cardinalidad por el número de bits a *true*.

Cada *BitSet* tiene un tamaño que es el número de bits que usa en cada momento. La implementación del tipo hace que la mayoría de las operaciones puedan ser llevadas a cabo en tiempo constante. La implementación puede hacerse de forma similar a un array dinámico pero ahora accediendo a cada uno de los bits con las operaciones disponibles para ello. El array que almacena los bits crece de tamaño cunado sea necesario.

La funcionalidad que ofrece es:

Métodos	**Descripción**
`and(BitSet set)`	Realiza la operación lógica *and* entre cada bit de *this* y los bits de set
`andNot(BitSet set)`	Pone a false en *this* todos los bits que están a *true* en set
`cardinality()`	Número de bits a true
`clear()`	Coloca todos los bits a *false*
`flip(int bitIndex)`	Hace el complemento sobre el bit indicado (operación lógica *not* sobre el bit)
`flip(int fromIndex, int toIndex)`	Hace el complemento sobre los bits indicados
`get(int bitIndex)`	Valor del bit indexado por bitIndex
`get(int fromIndex, int toIndex)`	Bitset compuesto por los bits indicados
`intersect(BitSet set)`	Verdadero si existen bits en *this* y en *set* indexados por el mismo entero y con valor *true*
`isEmpty()`	Verdadero si todos los bit tienen valor false
`length()`	Devuelve el índice más alto (mas uno) de los bits con valor *true*
`or(BitSet set)`	Realiza la operación lógica *or* entre cada bit de this y los bits de

	set
`set(int bitIndex)`	Hace true el bit indexado
`set(int fromIndex, int toIndex)`	Hace true desde *fromIndex* hasta *toIndex*
`set(int bitIndex, boolean booleanValue)`	Coloca el bit especificado al valor indicado
`set(int fromIndex, int toIndex, boolean booleanValue)`	Coloca el rango de bits especificado al valor indicado
`size()`	Devuelve el espacio en bits ocupado por el array
`xor(BitIndex bitIndex)`	Realiza la operación lógica *xor* entre cada bit de this y los bits de set. Es decir lleva a cabo la operación lógica *or* exclusivo

Propiedades específicas de BitSet.

Los constructores disponibles son:

- *BitSet(),* Crea un array de bit con todos los bits a false.
- *BitSet(int nbits),* Crea un array de *bits* suficientemente grande para almacenar *nbits* que pueden ser indexado de *0* a *nbits-1* y con todos los bits a false.

2.4 Tipo BasicHashTable

Una **tabla hash** es un tipo de datos que asocia claves con valores. Desde un punto de vista abstracto podemos considerarlo con un conjunto de pares clave-valor. La clave de tipo *K* y los valores de tipo *V*.

Esencialmente consiste en transformar la clave en un número que la tabla *hash* utiliza para localizar el valor deseado.

El tipo usa un tipo interno *BasicHashTable.Entry<K,V>*. Los valores de este tipo representan pares clave-valor. Sus propiedades consultables y modificables son: *Key: K* y *Value: V*. Para diseñar el tipo partimos de dos elementos: una función *hash* y un array dinámico cuyos elementos son listas enlazadas de pares clave-valor. Usando lo visto anteriormente este tipo podría ser *DynamicArray<LinkedList<Entry<K,V>>>.*

La función *hash* tiene el prototipo int *hash(int a)* y tiene como objetivo convertir el entero *a* en otro del rango *[0,m)* con la mayor dispersión posible. Es decir que si se toman dos enteros distintos la función *hash* los transforme, con alta probabilidad, en otros dos también distintos. Veremos algunos ejemplos de esta función.

Asumiendo un número *n* de entradas (pares clave-valor) el mecanismo de la *tabla hash* consiste en agruparlas en *m* grupos identificados por un entero de *0* a *m-1*. Todas las entradas que están en el grupo *i* cumplen la condición: *i == hash(e.getKey().hashCode()).* Donde *hashCode()* es un método disponible para todos los objetos en Java y *hash()* es un método de la tabla hash que transforma un entero cualquiera en otro en el rango *0..m-1.* Cada grupo *i* es un agregado de entradas que modelamos mediante una lista enlazada. Una propiedad

importante del estado de la tabla hash es el factor de carga *n/m (load factor)*. Es el número medio de entradas en cada grupo. Llamaremos capacidad (*capacity*) al valor *m* anterior, tamaño (*size*) al *n*. Asumimos un factor de carga de referencia (*reference load factor*).

El mecanismo es el siguiente: a partir de una tabla vacía se van añadiendo entradas. Cada entrada se coloca en el grupo correspondiente. Cuando se supera el factor de carga de referencia se amplia la capacidad y se sitúan, de nuevo, cada una de las entradas en los nuevos grupos. Este mecanismo interno lo llamaremos *rehash*.

Las principales propiedades del tipo son:

- *size(): int*, el tamaño, es decir el número de claves que es igual al número de pares clave-valor.
- *Empty: boolean,*
- *get(K key): V*, el valor asociado a la clave o null si no existe un par con la clave key.

Y las operaciones:

- *put(K k, V, v):V*, Añade el par *(k,v)* y devuelve el antiguo valor asociado a *k* o *null* si no existe.
- *remove(Object k): V,* Elimina la clave *k* y su valor asociado si existe. Devuelve el valor asociado a la clave o null si no existe.

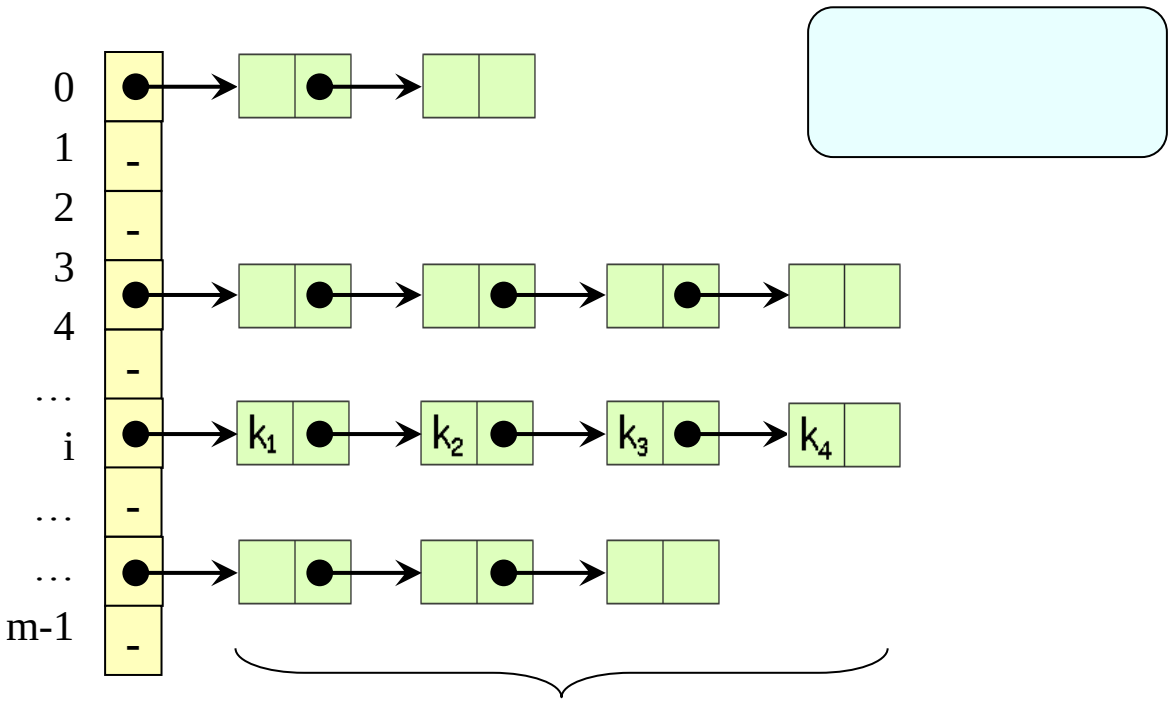
Los constructores

- *BasicHashTable<K,V> ():* Construye un atabla hash vacía con una capacidad por defecto
- *BasicHashTable<K,V)(int capacity):* Construye una tabla hash vacía con capacidad capacity.

Los invariantes que mantiene son:

- No existen dos pares clave-valor con la misma clave
- Todos los pares en el mismo grupo *i* cumplen *i == hash(e.getKey().hashCode()).*

Las sucesivas llamadas a los métodos *put* irán cambiando el factor de carga. Para mantener el invariante la implementación deberá invocar al método *rehash()* cuando se supere un factor de carga de referencia. También se podría llamar al método rehash(), aunque no se ha hecho en la implementación de abajo, cuando el factor de carga baja por debajo de un valor.



Una implementación posible es:

```
public class BasicHashTable<K, V> {
	private int capacity;
	private final int GROWING_FACTOR = 2;
	private int initialCapacityOfGroups;
	private int size;
	private double loadFactorReference;
	private DynamicArray<DynamicArray<Entry<K,V>>> elements;

	public BasicHashTable(int capacity, int initialCapacityOfGroups,
			double loadFactorReference) {
		super();
		this.capacity = capacity;
		this.initialCapacityOfGroups = initialCapacityOfGroups;
		this.loadFactorReference = loadFactorReference;
		initial();
	}

	public BasicHashTable() {
		super();
		this.capacity = 10;
		this.initialCapacityOfGroups = 2;
		this.loadFactorReference = 0.75;
		initial();
	}

	private void initial(){
		elements =
		  new DynamicArray<DynamicArray<Entry<K,V>>>(capacity);
		for(int i = 0; i < capacity; i++){
			elements.add(new
			DynamicArray<Entry<K,V>>(initialCapacityOfGroups));
		}
```

```
	}

	private int hash(int a){
		return a%capacity;
	}

	private void rehash(int newCapacity){
		DynamicArray<DynamicArray<Entry<K,V>>>
		     oldElements = elements;
		int oldCapacity = capacity;
		capacity = newCapacity;
		initial();
		Entry<K,V> e;
		for(int i = 0; i < oldCapacity; i++){
			for(int j=0; j < oldElements.get(i).size(); j++){
				e = oldElements.get(i).get(j);
				put(e);
			}
		}
	}

	private double getLoadFactor(){
		double sd = size;
		return sd/capacity;
	}

	public int size() {
		return size;
	}

	public boolean isEmpty(){
		return size == 0;
	}

	private Entry<K,V> getEntry(K key){
		int c = hash(key.hashCode());
		Entry<K,V> r = null;
		for(int i = 0; i < elements.get(c).size();i++){
			if(key.equals(elements.get(c).get(i).getKey())){
				r = elements.get(c).get(i);
			}
		}
		return r;
	}

	public V get(K key){
		Entry<K,V> e = getEntry(key);
		V r = null;
		if(e != null){
			r = e.getValue();
		}
		return r;
	}

	private void put(Entry<K,V> e){
		int c = hash(e.getKey().hashCode());
		elements.get(c).add(e);
		if(getLoadFactor() > loadFactorReference ){
			rehash(capacity*GROWING_FACTOR);
		}
```

```
	}

	public V put(K key, V value){
		Entry<K,V> e = getEntry(key);
		V r = null;
		if(e == null){
			put(new Entry<K,V>(key,value));
			size++;
		} else {
			r = e.getValue();
			e.setValue(value);
		}
		return r;
	}

	public V remove(K key){
		int c = hash(key.hashCode());
		V r = null;
		int  p = -1;
		for(int i = 0; i < elements.get(c).size();i++){
			if(key.equals(elements.get(c).get(i).getKey())){
				r = elements.get(c).get(i).getValue();
				p = i;
			}
		}
		if(p >= 0){
			elements.get(c).remove(p);
			size--;
		}
		return r;
	}

	public String toString(){
		String s = "{";
		boolean prim = true;
		for(int i=0; i < capacity; i++){
			for(int j=0;j< elements.get(i).size(); j++){

				if(prim){
					prim = false;
					s = s+elements.get(i).get(j);
				}else{
					s = s+","+elements.get(i).get(j);
				}
			}
		}
		s = s+"}";
		return s;
	}

	public class Entry<K1,V1> {
		private K1 key;
		private V1 value;
		public Entry(K1 key, V1 value) {
			super();
			this.key = key;
			this.value = value;
		}
		public K1 getKey() {
			return key;
```

```
		}
		public void setKey(K1 key) {
			this.key = key;
		}
		public V1 getValue() {
			return value;
		}
		public void setValue(V1 value) {
			this.value = value;
		}
		public String toString(){
			return "("+key+","+value+")";
		}
	}
}
```

3. Árboles

3.1 Tipo Tree

Un árbol (**Tree<E>)** es un tipo de datos que puede definirse recursivamente. Desde ese punto de vista un árbol es: un árbol vacío o un árbol con una etiqueta de tipo *E* y una secuencia de hijos que son también árboles. Los árboles son tipos útiles para implementar muchos otros tipos. Ofrecen varias propiedades y operaciones que podemos concretar en los siguientes métodos:

- *int getNumChildren():* Número de hijos
- *int size():* Número de etiquetas del árbol
- *boolean isEmpty():* Si el árbol está vacío. Es decir no tiene ninguna etiqueta.
- *boolean isRoot():*
- *boolean isLeaf():* Es un árbol vacío o con cero hijos.
- *E getLabel():* La etiqueta del árbol. Tiene como precondición que el árbol no esté vacío.
- *void setLabel(E label):* Cambia el valor de la etiqueta. Tiene como precondición que la etiqueta no sea *null*.
- *Tree<E> getParent():* El árbol padre.
- *int getDepth():* La profundidad del árbol. Es decir la longitud del camino hasta la raíz. La raíz tiene profundidad cero y si es vacía -1.
- *int getHeight():* La altura de un árbol. Es decir la longitud del camino más largo hasta sus hojas. La altura de un árbol con una etiqueta y sin hijos es cero. Si es vacío -1.
- *Tree<E> getElement(int index):* El árbol hijo que ocupa la posición *index*. Los hijos tienen los índices de 0 a *getNumChildren()-1*. Tiene como precondición que index sea mayor o igual que cero y menor que *getNumChildren()* y que el árbol no esté vacío.
- *Tree<E> setElement(int index, Tree<E> element):* Cambia el valor del árbol ubicado en posición *index* y devuelve el antiguo árbol en esa posición. Si *index* es mayor o igual a *numChildren()* entonces numChildren queda actualizado a *index+1* y los árboles en

posiciones no definidas previamente se hacen vacíos. Tiene como precondición que el árbol no esté vacío.

- *boolean add(Tree<E> element):* Añade un hijo más a la derecha de los demás. Tiene como precondición que el árbol no esté vacío.
- *void add(int index, Tree<E> element):* Añade un hijo en la posición *index* y desplaza a la derecha los de índice mayor a *index*. Tiene como precondición que index sea mayor o igual a cero y menor o igual a *getNumChildren()* y que el árbol no esté vacío.
- *Tree<E> remove(int index):* Elimina el hijo en posición index desplazando a la izquierda los de índice mayor. Devuelve el árbol en posición *index*. Tiene como precondición que index sea mayor o igual que cero y menor que *getNumChildren()* y que el árbol no esté vacío.

Una posible implementación es:

```
public class Tree<E> {
	private E label;
	private final int INITIAL_CAPACITY = 2;
	private DynamicArray<Tree<E>> elements;
	private Tree<E> parent;

	public Tree() {
		super();
		this.label = null;
		this.elements = new
			DynamicArray<Tree<E>>(INITIAL_CAPACITY);
		this.parent = null;
	}

	public Tree(E label){
		super();
		this.label = label;
		this.elements = new
			DynamicArray<Tree<E>>(INITIAL_CAPACITY);
		this.parent = null;
	}

	public Tree(E label, Tree<E>[] elements) {
		super();
		this.label = label;
		this.elements = new DynamicArray<Tree<E>>(elements);
		this.parent = null;
	}

	public Tree(E label, DynamicArray<Tree<E>> elements) {
		super();
		this.label = label;
		this.elements = elements;
		this.parent = null;
	}

	public int getNumChildren(){
		int r = 0;
		if(!isEmpty()){
			r = elements.size();
		}
```

```
		return r;
	}

	private int size(Tree<E> t){
		int r;
		if(t== null || t.isEmpty()){
			r = 0;
		}else if(t.getNumChildren()==0){
			r = 1;
		}else {
			r = 1;
			for(int i=0; i<t.getNumChildren(); i++){
				r = r+size(t.getElement(i));
			}
		}
		return r;
	}

	public int size(){
		return size(this);
	}

	public boolean isEmpty() {
		return label == null;
	}

	public boolean isRoot() {
		return parent == null;
	}

	public boolean isLeaf(){
		return isEmpty() || getNumChildren() == 0;
	}

	public E getLabel() {
		Preconditions.checkState(!isEmpty());
		return label;
	}

	public void setLabel(E label) {
		Preconditions.checkNotNull(label);
		this.label = label;
	}

	public Tree<E> getParent() {
		return parent;
	}

	private void setParent(Tree<E> parent) {
		this.parent = parent;
	}

	public Tree<E> getElement(int index) {
		Preconditions.checkState(!isEmpty());
		Tree<E> r = elements.get(index);
		if(r==null){
			r = new Tree<E>();
		}
		return r;
	}
```

```
public Tree<E> setElement(int index, Tree<E> element) {
	Preconditions.checkState(!isEmpty());
	element.setParent(this);
	Tree<E> r =this.elements.set(index, element);
	return r;
}

public int getDepth() {
	int r = -1;
	if(!isEmpty()){
		r =0;
		Tree<E> p = getParent();
		while(p!=null){
			p = p.getParent();
			r++;
		}
	}
	return r;
}

private int getHeight(Tree<E> t){
	int r;
	if(t== null || t.isEmpty()){
		r = -1;
	}else if(t.getNumChildren()==0){
		r = 0;
	}else {
		r = -1;
		for(int i=0; i<t.getNumChildren(); i++){
			r = Math.max(r,getHeight(t.getElement(i)));

		}
		r++;
	}
	return r;
}

public int getHeight() {
	return getHeight(this);
}

public boolean add(Tree<E> element){
	Preconditions.checkState(!isEmpty());
	element.setParent(this);
	boolean r = elements.add(element);
	return r;
}

public  void add(int index, Tree<E> element){
	Preconditions.checkState(!isEmpty());
	element.setParent(this);
	elements.add(index,element);
}

public Tree<E> remove(int index){
	Preconditions.checkState(!isEmpty());
	Tree<E> r = elements.remove(index);
	r.setParent(null);
	return r;
```

```
        }

        …

        public String toString(){
                return toString(this);
        }

        public E[] toArray(){
                E[] r = Arrays.copyOf(this.elements, size);
                return r;
        }

        private String toString(Tree<E> t){
                String r;
                boolean prim = true;
                if(t == null || t.isEmpty()){
                        r =" ";
                }else if(t.getNumChildren()==0){
                        r = t.getLabel().toString();
                }else {
                        r= t.getLabel().toString()+";";
                        for(int i=0; i<t.getNumChildren(); i++){
                                if(prim){
                                        r = r+toString(t.getElement(i));
                                        prim = false;
                                }else{
                                        r = r+","+toString(t.getElement(i));
                                }
                        }
                }
                return "("+r+")";
        }
}
```

3.2 Funciones sobre árboles (Trees)

En la clase *Trees* agrupamos un conjunto de métodos adecuados para trabajar con árboles. La funcionalidad de los mismos es:

- *Iterable<Tree<E>> breadth(Tree<E> t):* Devuelve un iterador en anchura que va recorriendo todos los subárboles de t, incluido él mismo
- *Iterable<Tree<E>> depth(Tree<E> t, int pos):* Devuelve un iterable general en profundidad que va recorriendo todos los subárboles de *t*, incluido él mismo. Toma un segundo parámetro que indica la posición en la que se visitará el árbol actual: 0 delante del primer hijo, 1 delante del segundo hijo, etc.
- *boolean equals(Tree<E> t1, Tree<E> t2):* Devuelve verdadero si t1 es igual a t2. Dos árboles son iguales si ambos son vacíos o si ambos no son vacíos tienen iguales sus etiquetas y cada uno de sus hijos
- *boolean isOrdered(Tree<E> t, Comparator<? super E> cmp):* Devuelve verdadero si el árbol t está ordenado según el orden del comparator.

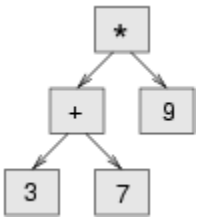
- *Iterable<Tree<E>> inOrder(Tree<E> t):* Devuelve un iterable en inorden que va recorriendo todos los subárboles de t, incluido él mismo
- *Iterable<Tree<E>> postOrder(Tree<E> t):* Devuelve un iterador en postorden que va recorriendo todos los subárboles de t, incluido él mismo
- *Iterable<Tree<E>> preOrder(Tree<E> t):* Devuelve un iterador en preorden que va recorriendo todos los subárboles de t, incluido él mismo
- *Tree<E> clone(Tree<E> t):* Construye una copia del árbol. El tipo de la etiqueta deber ser copiable.
- *String toString(Tree<E> t):* Crea una representación del árbol como cadena de caracteres.

Los diferentes tipos de recorridos sobre árboles binarios podemos definirlos recursivamente de la siguiente forma:

- *Recorrido en preorden*: Se visita primero el árbol actual luego el primer hijo, posteriormente el segundo hijo, etc.
- *Recorrido en postorden*: Se visita primero el primer hijo, posteriormente el segundo hijo derecho, etc. y por último el árbol actual.
- *Recorrido en inorden*: Se visita primero el primer hijo, luego el árbol actual y finalmente el segundo hijo, etc.
- *Recorrido general en profundidad*: Es una generalización de los anteriores donde se indica la posición el momento en el que se visitará el árbol actual.
- *Recorrido en anchura o por niveles*: Se visita primero el árbol actual (nivel 1), luego los árboles de nivel 2 de izquierda a derecha, etc.

4. Árboles de Expresiones y Árboles Sintácticos

Un caso particular de los árboles son los **árboles de expresiones**. Son árboles donde las etiquetas asociadas a nodos internos son operadores y las asociadas a nodos hoja operados. De forma general los operandos vamos a considerarlos como operadores con ningún operando. Como por ejemplo:



Los árboles de expresión suelen tener, además de los métodos de los árboles, métodos del tipo: *getValue()* (para obtener el valor), *getType()* (para obtener el tipo), check() para comprobar la consistencia de la expresión, etc. Una posible implementación es:

```
public class Expression extends Tree<Operator> {

	public Expression(Operator label) {
		super(label);
	}

	public Expression(Operator label, Expression... elements) {
		super(label, elements);
	}

	private Object getValue(Expression e){
		Object r;
		Preconditions.checkArgument(e!=null && !e.isEmpty());
		if(e.isLeaf()){
			r = e.getLabel().getValue(null);
		}else {
			DynamicArray<Object> t =
			   new DynamicArray<Object>();
			Expression e1;
			for(int i=0; i < e.getNumChildren(); i++){
				e1 = (Expression) e.getElement(i);
				t.add(e1.getValue(e1));
			}
			Object[] ta = t.toArray();
			r = e.getLabel().getValue(ta);
		}
		return r;
	}

	public Object getValue(){
		return getValue(this);
	}

	Class<?> getResultType(){
		return getLabel().getResultType();
	}

	private boolean check(Expression e){
		Preconditions.checkArgument(e!=null);
		boolean r;
		if(e.isEmpty()){
			r = false;
		} else {
			DynamicArray<Class<?>> t =
			    new DynamicArray<Class<?>>();
			for(int i=0; i < e.getNumChildren(); i++){
				t.add(e.getElement(i).getLabel().getResultType());

			}
			Class<?>[] ta = t.toArray();
			r = e.getLabel().check(ta);
		}
		return r;
	}

	public boolean check(){
		return check(this);
	}

	public String toString(){
```

```
            return toString((Expression)this);
      }

      private String toString(Expression t){
            String r;
            boolean prim = true;
            if(t.isLeaf()){
                  r = t.getLabel().toString();
            }else {
                  r= t.getLabel().toString()+"(";
                  for(int i=0; i<t.getNumChildren(); i++){
                        if(prim){
                              r = r+toString((Expression)t.getElement(i));
                              prim = false;
                        }else{
                              r = r+","+
                                    toString((Expression)t.getElement(i));
                        }
                  }
                  r=r+")";
            }
            return r;
      }
}
```

El tipo operador es tiene métodos para calcular el valor a partir del de sus operandos, devolver el tipo del resultado de la expresión y comprobar la consistencia de la expresión. Es decir cada operador se aplica a los operandos adecuados en número y tipo.

```
public interface Operator {
      Object getValue(Object[] values);
      Class<?> getResultType();
      boolean check(Class<?>[] typesOfValues);
}
```

Como hemos dicho arriba un operando vamos a representarlo por un operador sin operandos. La clase Id implementa un operando en general.

```
public class Id implements Operator {
      private Object value;
      private Class<?> type;
      private String name;

      public Id(String name, Object value) {
            super();
            this.name = name;
            this.value = value;
            this.type = value.getClass();
      }

      @Override
      public Object getValue(Object[] values) {
            return value;
      }

      public void setValue(Object value) {
```

```
			this.value = value;
			this.type = value.getClass();
		}

		@Override
		public Class<?> getResultType() {
			return type;
		}

		@Override
		public boolean check(Class<?>[] typesOfValues) {
			boolean r = true;
			if(typesOfValues != null){
				r = false;
			}
			return r;
		}

		public String toString(){
			return name;
		}
}
```

El operador + binario sobre número reales se puede implementar como:

```
public class MasBinario implements Operator {

		@Override
		public Object getValue(Object[] values) {
			Double r = ((Double)values[0])+((Double)values[1]);
			return r;
		}

		@Override
		public Class<?> getResultType() {
			return Double.class;
		}

		@Override
		public boolean check(Class<?>[] typesOfValues) {
			return typesOfValues.length==2 &&
				typesOfValues[0].equals(Double.class) &&
				typesOfValues[1].equals(Double.class);
		}
}
```

El operador unario para convertir un dato númerico de un tipo en otro:

```
public class ConvTypeNumber implements Operator {
		Class<?> inType;
		Class<?> outType;

		public ConvTypeNumber(Class<?> inType, Class<?> outType) {
			super();
			this.inType = inType;
			this.outType = outType;
		}
```

```
	@Override
	public Object getValue(Object[] values) {
		// TODO Auto-generated method stub
		Object r = null;
		Number n = (Number) values[0];
		if(outType.equals(Double.class)){
			r = n.doubleValue();
		}else if(outType.equals(Integer.class)){
			r = n.intValue();
		}else if(outType.equals(Long.class)){
			r = n.longValue();
		}else if(outType.equals(Float.class)){
			r = n.floatValue();
		}else {
			Preconditions.checkState(r!=null);
		}
		return r;
	}

	@Override
	public Class<?> getResultType() {
		return outType;
	}

	@Override
	public boolean check(Class<?>[] typesOfValues) {
		return typesOfValues.length==1 &&
			inType.equals(typesOfValues[0]);
	}
}
```

Un programa principal para comprobar el funcionamiento:

```
public static void main(String[] args) {
		Expression e1 = new Expression(new Id("a",3.));
		Expression e2 = new Expression(new Id("b",7));
		Expression e4 = new Expression(new
			ConvNumberType(Integer.class,Double.class),e2);
		Expression e3 = new Expression(new MasBinario(),e1,e4);
		System.out.println(e3.check());
		System.out.println(e3.getResultType());
		System.out.println(e3.getValue());
		System.out.println(e3);
}
```

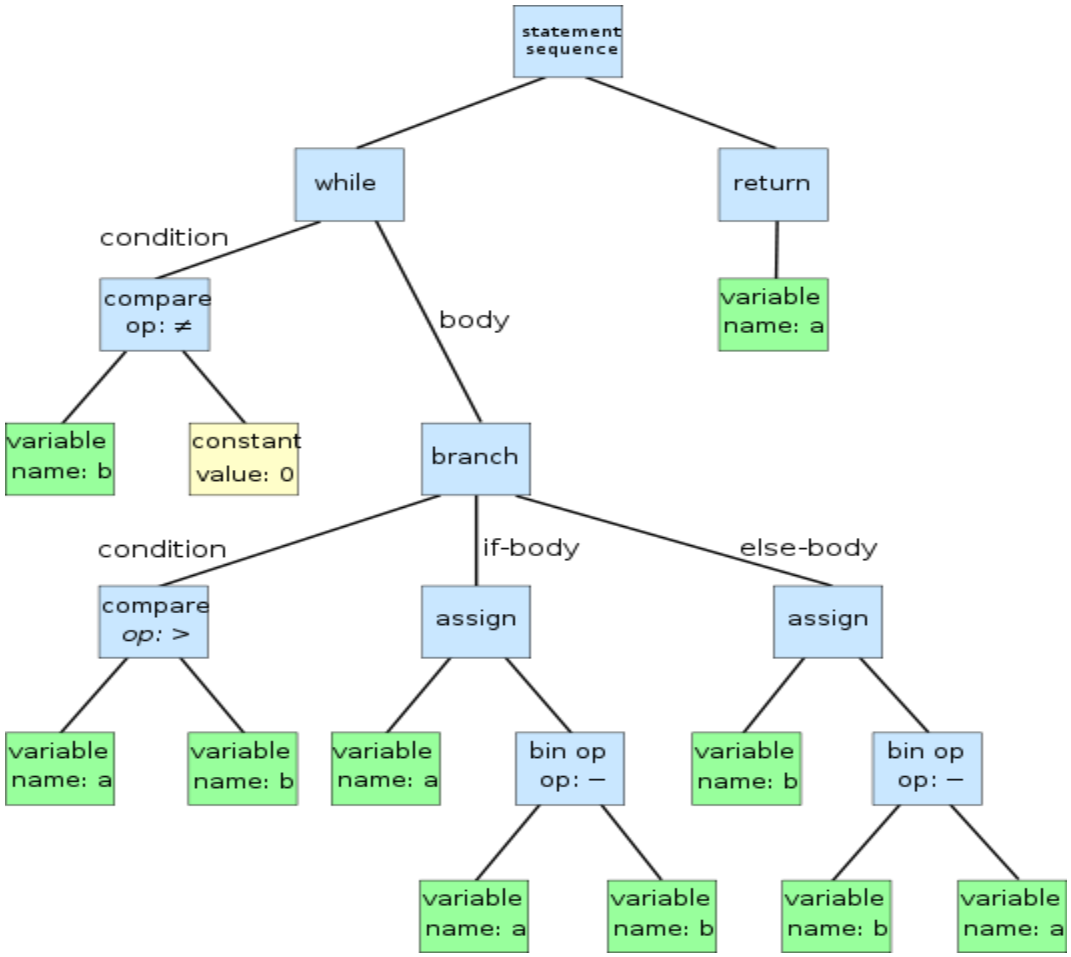
4.1 Arboles sintácticos

Los árboles sintácticos (los árboles de sintaxis abstracta en general) son una generalización de los árboles de expresiones donde los operadores pueden ser del tipo*: while, ifthenelse, etc.* Pueden servir, entre otras cosas para representar la estructura abstracta de un programa. Por ejemplo el siguiente segmento de código:

```
while(b != 0){
```

```
        if (a > b) {
                a := a - b
        } else {
                b := b - a
        }
return a
```

Puede ser representado por el árbol de sintaxis abstracta:



En estos árboles no se suele estar interesado en el valor del árbol, pero sí en la consistencia del mismo, o en el tipo resultado, etc.

5. Árboles de búsqueda (SearchTree) y otros usos de los árboles

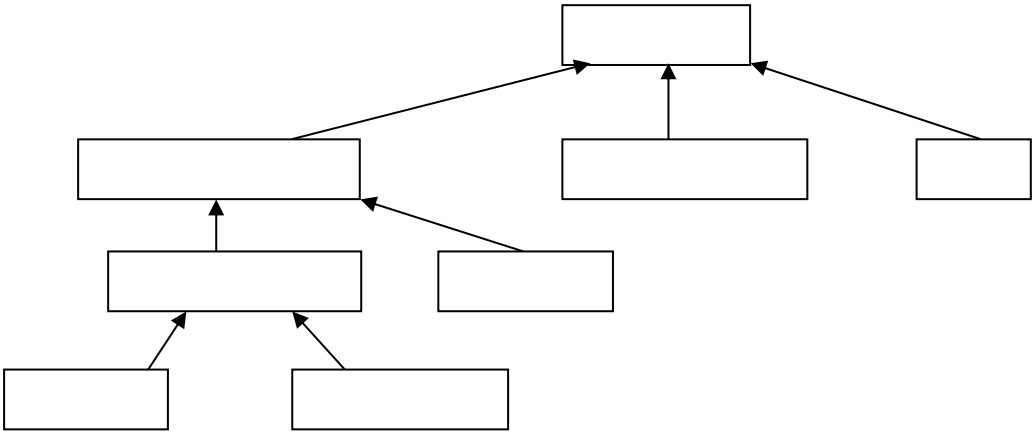
Los árboles de búsqueda son estructuras de datos adecuadas para implementar agregados de datos con operaciones de búsqueda, añadir, eliminar elementos. Son árboles que mantienen ordenadas sus etiquetas. Suponiendo un orden definido sobre las etiquetas y siendo *a* la etiqueta de la raíz, *b* una etiqueta cualquiera del subárbol izquierdo y *c* una cualquiera del subárbol derecho entonces se cumple $b<a<c$. Los árboles de búsqueda tienen operaciones para añadir, eliminar, buscar una etiqueta y contar cuantas hay. Un árbol de búsqueda dispone, por tanto de los métodos:

```
interface SearchTree<E> {
	int size();
	boolean isEmpty();
	E element();
	boolean add(E e);
	E remove(E e);
	E remove();
	boolean contains(E e);
	Comparator<E> comparator();
}
```

La semántica de los métodos será:

- *int size():* Número de etiquetas del agregado
- *boolean isEmpty():* Verdadero si el número de etiquetas es cero.
- *E element():* El primer elemento según el orden del agregado o null si estaba vacío
- *boolean add(E e):* Añade la etiqueta *e* al agregado. Si ya estaba devuelve *false* y si no estaba *true.*
- *E remove(E e):* Elimina la etiqueta e del agregado. Devuelve la etiqueta eliminada o null si no estaba en el árbol.
- *E remove(E e):* Elimina la primera etiqueta del agregado. Devuelve la etiqueta eliminada o null si no estaba en el árbol.
- *boolean contains(E e):* Verdadero si el agregado contiene la etiqueta.
- *Comparator<E> comparator():* El orden asociado al agregado.

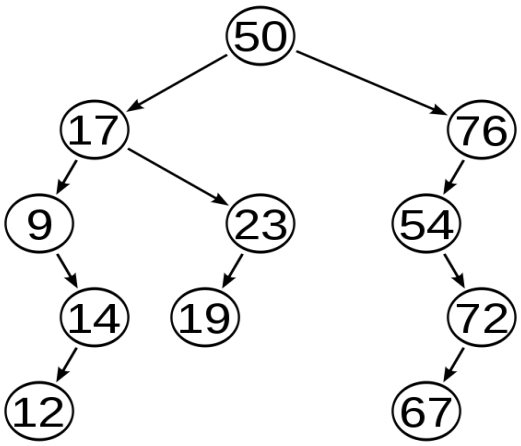
Por simplicidad asumimos que en un árbol de búsqueda no hay dos etiquetas iguales aunque esta restricción podría eliminarse. Añadiendo diversos invariantes obtenemos diversos tipos de árboles de búsqueda.



Los árboles de búsqueda binarios pueden ser vacíos o tener cero o dos hijos.

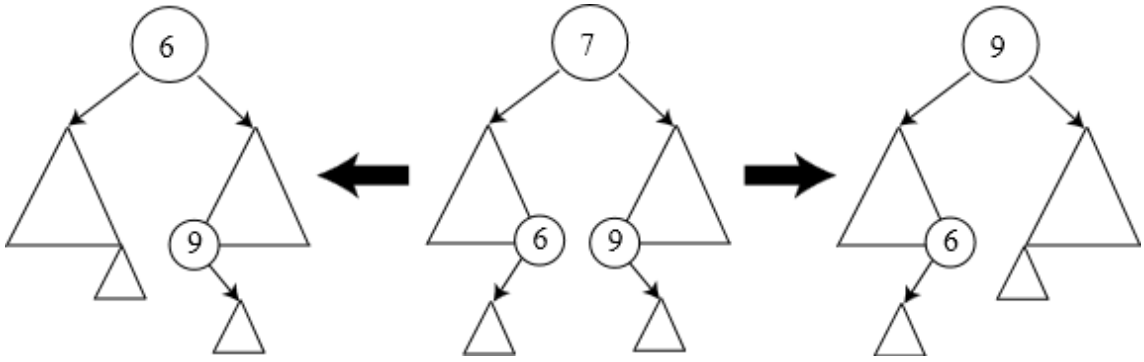
5.1 Tipo BinarySortedTree

Un **árbol binario ordenado** es un subtipo de árbol binario de búsqueda donde las etiquetas del subárbol izquierdo son menores que la etiqueta de la raíz y esta, a su vez, menor que las etiquetas en el árbol derecho. Cada subárbol es un árbol binario ordenado.



En este tipo de árboles la búsqueda de una etiqueta (método *contains(E e))* se implementa siguiendo un camino que parte de la raíz y va decidiendo en cada etiqueta del subárbol correspondiente si parar (si ha encontrado la etiqueta) o continuar por el subárbol izquierdo o derecho según que la etiqueta a buscar sea menor o mayor que la del subárbol. Si se encuentra un árbol vacío es que el agregado no contiene la etiqueta. La complejidad del caso peor para este método es $\Theta(n)$. Donde *n* es el número de etiquetas en el agregado y el caso peor es cuando el árbol degenera en una lista.

La operación de añadir *(add(E e))* sigue la secuencia anterior y convierte el árbol vacío encontrado en un árbol con la etiqueta a añadir. El primer elemento del agregado (*element())* se encuentra siguiendo el hijo izquierdo hasta que no haya otro hijo izquierdo. La operación de eliminar *(remove(E e))* elimina la etiqueta *e*. Primero la busca, si la encuentra y está en una hoja lo elimina, si el árbol tiene un hijo lo sustituye por su subárbol. En otro caso el árbol tiene dos hijos. Sea R el valor de la etiqueta a borrar y S la etiqueta siguiente o anterior a la misma en inorden. Entonces se trata de sustituir R por S y luego eliminar S. Abajo se muestran las dos posibilidades.



Como podemos ver los árboles binarios de búsqueda tienen sus operaciones con una complejidad en el caso pero de $\Theta(n)$. Si el árbol estuviera equilibrado la complejidad sería menor. En efecto en un árbol binario equilibrado (cada árbol tiene dos hijos exactamente) el número total de etiquetas en un árbol de altura *k* es:

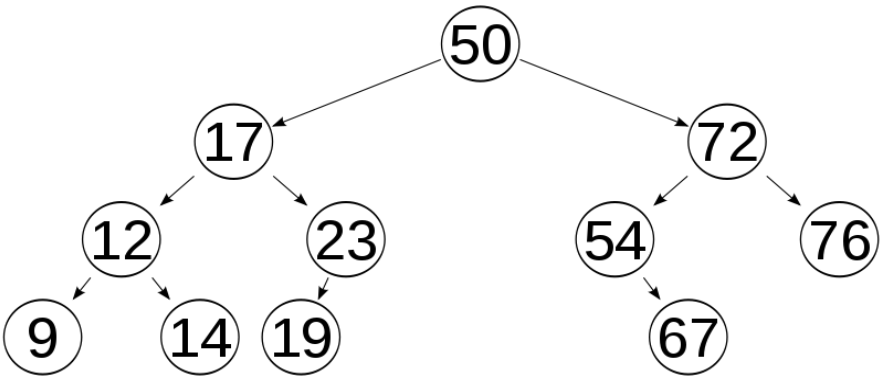
$$n = \sum_{i=0}^{k} 2^i = \frac{2^{k+1} - 1}{1} \approx 2^{k+1}, \quad k \approx \Theta(\log n)$$

Por lo tanto si el árbol esta equilibrado las operaciones anteriores tienen una complejidad en el caso peor de $\Theta(\log n)$ debido a que el camino más largo es de $\log n$ si hay *n* etiquetas. Pero las operaciones de inserción y eliminación desequilibran un árbol previamente equilibrado. Son necesarios árboles con invariantes más fuertes para conseguir esa complejidad.

5.2 Tipo AVLTree

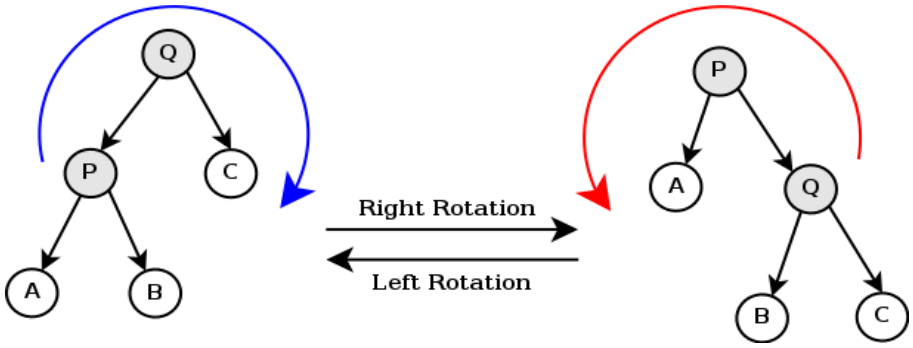
Los árboles **AVL** (Adelson-Velskii y Landis) son árboles binarios ordenados casi equilibrados. Es decir además de las restricciones de los árboles binarios de búsqueda se cumple la restricción:

- En cada subárbol la altura del hijo izquierdo no difiere en más de una unidad de la altura del hijo derecho.

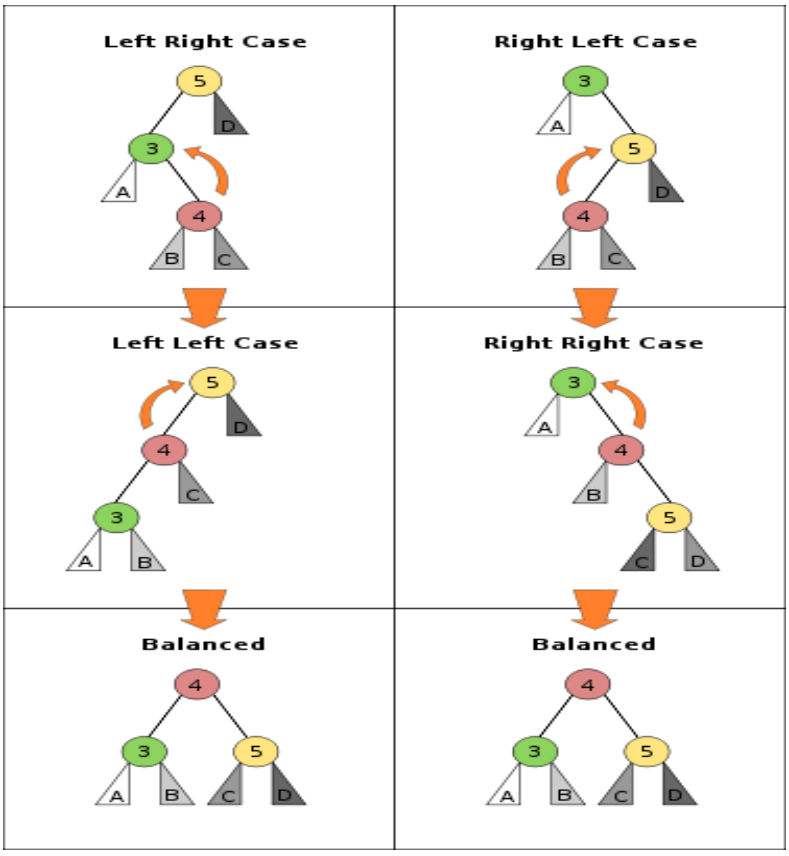


Como vemos los árboles AVL son casi equilibrados. Como hemos comentado arriba las operaciones de inserción y eliminación desequilibran el árbol. Para mantenerlo casi equilibrado (según el invariante establecido para los árboles **AVL**) debemos llevar a cabo, después de la inserción o la eliminación operación de requilibrado.

Para implementar las operaciones de reequilibrado se diseñan dos transformaciones básicas sobre árboles binarios: *leftRotation, rightRotation.* Cada una de ellas reordena un árbol dado tal como se muestra en la figura siguiente:



Cuando un árbol AVL se desequilibra podemos nos encontraremos con alguno de los cuatro casos de la figura de abajo. Para conseguir equilibrar el árbol debemos transformar el árbol con las operaciones anteriores según se indica en la figura de abajo.

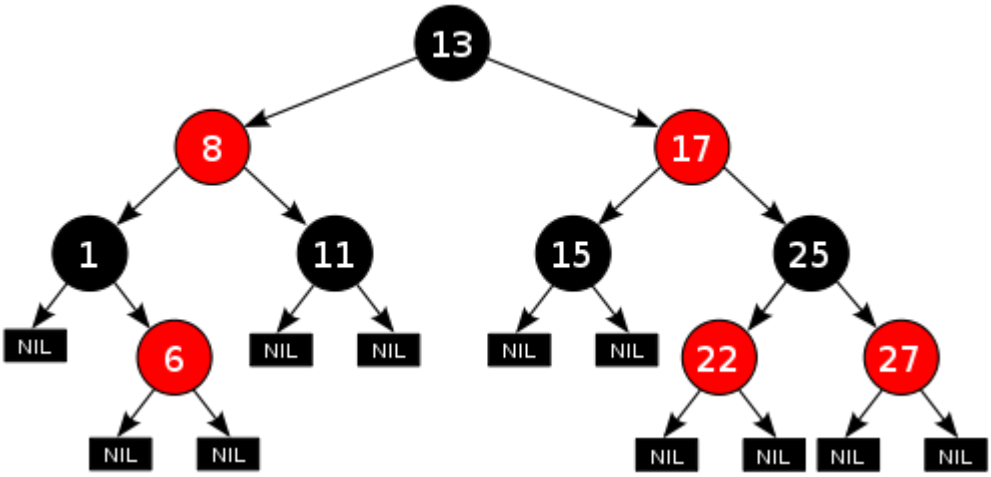


5.3 Tipo RedBlackTree

Los **árboles rojo-negro** son árboles binarios ordenados que están aproximadamente equilibrados en el sentido que se explica abajo. Además de las propiedades de los árboles binarios de búsqueda cumplen las siguientes restricciones:

- Todo nodo es o bien rojo o bien negro.
- La raíz es negra.
- Las hojas son árboles vacíos y son negras.
- Los hijos de todo nodo rojo son negros

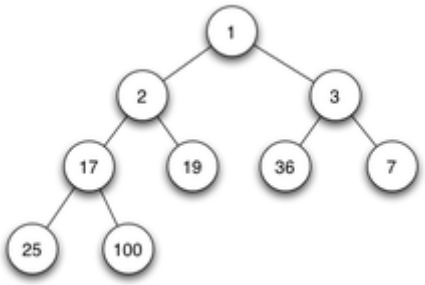
- Cada camino desde un nodo a una hoja descendiente contiene el mismo número de nodos negros. Para un árbol dado el número de nodos negros desde la raíz a las hojas es constante para todos los caminos y se denomina *Altura negra del árbol.*
- El camino más largo desde la raíz hasta una hoja no es más largo que 2 veces el camino más corto desde la raíz del árbol a una hoja en dicho árbol.



5.4 Tipo HeapTree

Los **montículos binarios** (**HeapTree**) son árboles binarios ordenados con las siguientes restricciones:

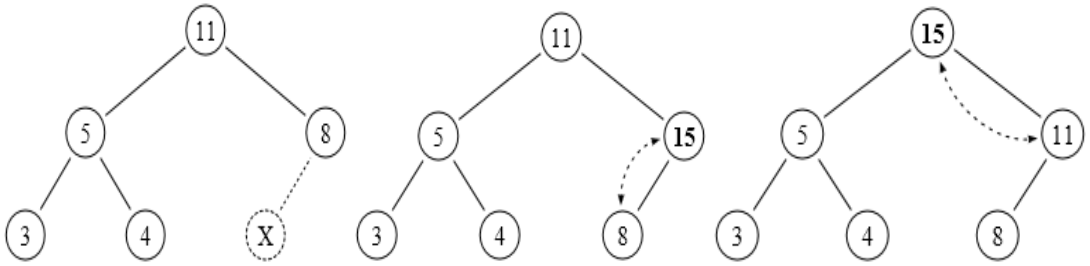
- Los subárboles hojas no son nunca árboles vacíos y todos los niveles están completos excepto posiblemente el último que está lleno de izquierda a derecha.
- La etiqueta asociada a la raíz de un árbol es siempre menor que las etiquetas contenidas en los árboles hijos.
- Cada subárbol es un montículo binario



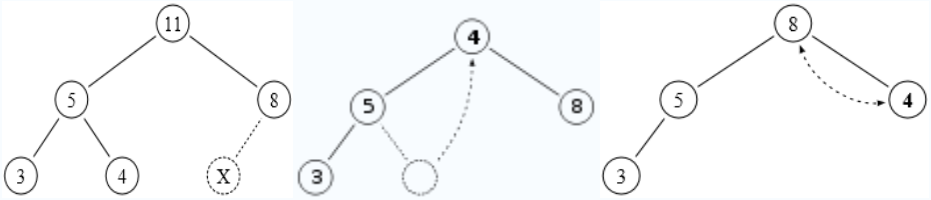
Ejemplo de HeapTree.

Los montículos suelen denominarse montículos de mínimos como el de arriba (cuando la raíz es menor que cada uno de los hijos) o de máximos (como abajo) cuando la raíz es mayor que cada uno de los hijos. Inserción de la etiqueta 15 se hace incluyéndola en la siguiente posición

libre del último nivel del árbol (posición señalada con X). Posteriormente para mantener el invariante se va intercambiando la etiqueta por la raíz del árbol correspondiente.



Eliminación de la etiqueta en la raíz se hace intercambiándola por la etiqueta en la posición más a la derecha del último nivel del árbol. Posteriormente, para mantener el invariante, se intercambia la raíz por la etiqueta de alguno de sus hijos.



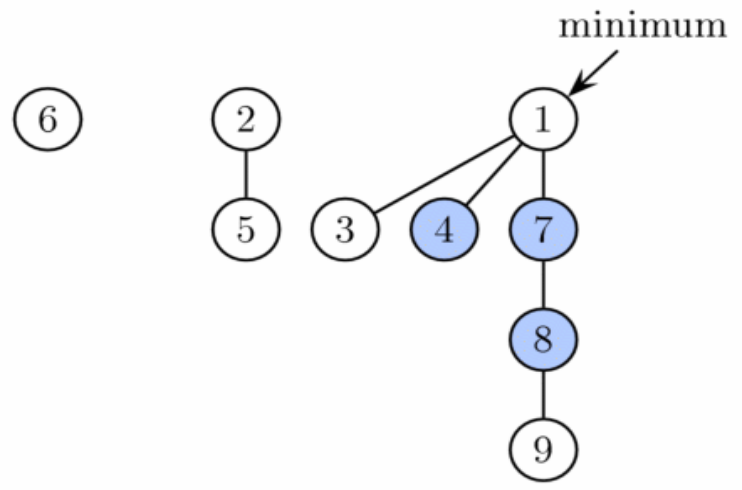
5.5 Montículo de Fibonacci

Esta es una estructura de datos compleja pero muy eficiente. Un montículo de Fibonacci es una lista de árboles (no necesariamente binarios) en cada uno de los cuales la etiqueta de la raíz es menor que la de los hijos. Esto implica que la clave mínima está siempre en la raíz de uno de los árboles. Los árboles no tienen una forma predefinida y en un caso extremo el montón puede tener cada elemento en un árbol separado o en un único árbol de profundidad *n*. Además los nodos pueden estar marcados o no. Un nodo está marcado si al menos uno de sus hijos se cortó desde que el nodo fue hecho hijo de otro nodo (todas las raíces están desmarcadas). La estructura puede estar consolidada o no. La operación de consolidación, que explicaremos abajo, normaliza la lista de árboles para que no haya dos con el mismo número de hijos.

Para describir algunas propiedades de esta estructura usaremos la siguiente notación:

- *n* = número de nodos (etiquetas o claves) en el montón.
- *r(x)* = número de hijos del nodo x.
- *r(H)* = rango máximo de los nodos en el montón H.
- *t(H)* = número de árboles en el montón H.
- *m(H)* = número de nodos marcados en el montón H.
- *s(t)* = tamaño del árbol (número nodos)
- t_k = árbol cuya raíz tiene *k* hijos

Un ejemplo de montículo de Fibonacci se muestra en la figura siguiente:



Ejemplo de un montículo de Fibonacci. Tiene tres árboles de rangos 0, 1 y 3. Tres vértices están marcados (mostrados en azul).

La estructura mantiene un invariante sobre el grado de los nodos de los árboles (el número de hijos). El grado de los nodos se tiene que mantener bajo: cada nodo tiene un grado máximo de *O(log n),* donde n el número de etiquetas del montículo. En concreto:

$$r(H) \leq \log_\theta n\,, \quad \theta = \frac{1+\sqrt{5}}{2}$$

El tamaño (número de claves) de un subárbol cuya raíz tiene grado *k* es por lo menos F_{k+2}, donde F_k es un número de Fibonacci (pero considerando $F_0 = 1$, $F_1 = 2$).

$$s(t_k) \geq F_{k+2}$$

Cuando la estructura está consolidada se cumple:

$$t(H) \leq r(H) + 1$$

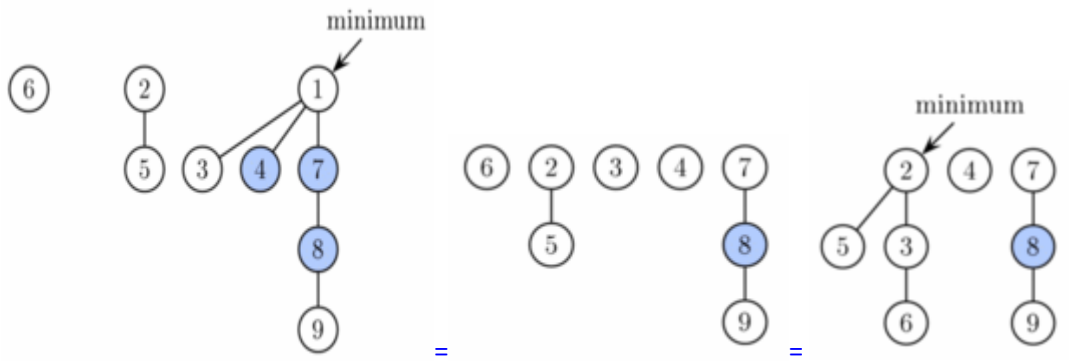
La estructura es adecuada para implementar eficientemente las operaciones del tipo *SearchTree* y dos operaciones más: *Unión* y *Decrementar el valor de la clave*.

La operación *Encontrar Mínimo* es trivial. La *Unión* se implementa simplemente concatenando las listas de raíces de árboles de los dos montones. La operación *Insertar* trabaja creando un nuevo montículo con un elemento y haciendo la *Unión*. En ambos casos la estructura queda sin consolidar. La operación de *Consolidación,* explicada más abajo, se pospone hasta la siguiente llamada a la operación de *Eliminar el Mínimo*.

La operación *Eliminar Mínimo* opera en tres fases. Primero cogemos la raíz con el elemento mínimo, la borramos y convertimos sus hijos en raíces de nuevos árboles. En la segunda fase decrementamos el número de raíces agrupando sucesivamente las raíces con el mismo grado.

Cuando dos raíces *u* y *v* tienen el mismo grado, hacemos que la que tenga la clave mayor sea hija de la otra. Por lo tanto colocamos la más pequeña como raíz. Esto se repite hasta que todas las raíces tienen un grado diferente. Esta segunda fase se denomina *Consolidación* (*consolidate*). Para implementarla eficientemente se mantienen varias listas cada una de las cuales enlaza los árboles cuyas raíces tienen el mismo número de hijos. Por último, comprobamos cada una de las raíces restantes y encontramos el mínimo.

La secuencia de eliminación se muestra en la figura:

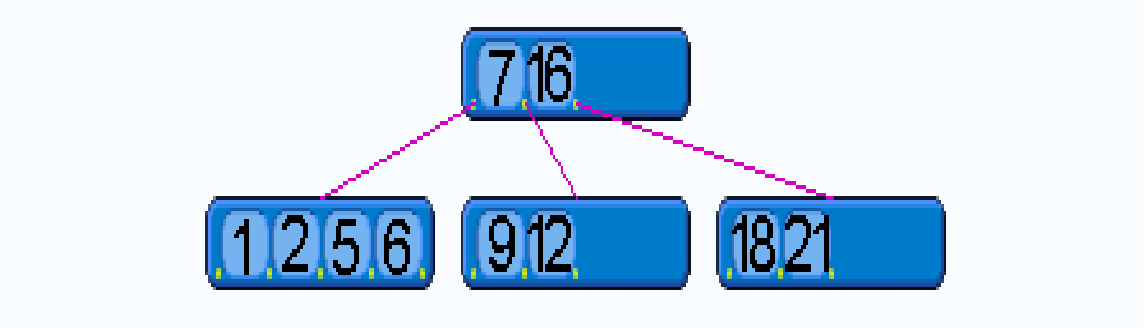


La operación *Decrementar Clave* decrementará la clave del nodo y si viola la propiedad del montículo (la nueva clave es más pequeña que la clave del padre), el nodo se corta de su padre. Si el padre no es una raíz, se marca. Si ya estaba marcado, se corta también y su padre se marca. Continuamos subiendo hasta que, o bien alcanzamos la raíz o un vértice no marcado. En el proceso creamos un número *k* de nuevos árboles. Los nodos marcados en cada momento son aquellos de los que se ha eliminado un hijo.

Por último, la operación *Borrar* puede ser implementada simplemente decrementando la clave del elemento a borrar a menos infinito, convirtiéndolo en el mínimo de todo el montículo, entonces llamamos a *Extraer Mínimo* para borrarlo.

5.6 Tipo BTree

Un árbol B **(BTree)** es una generalización de los árboles AVL. Es un árbol donde cada etiqueta es de tipo *List<E>.* Es decir tenemos asociada a la raíz de cada árbol una lista de etiquetas.



Ejemplo de BTree

Un árbol-B de orden M (el máximo número de hijos que puede tener cada nodo) es un árbol que satisface las siguientes propiedades:

- Cada subárbol tiene como máximo M hijos, como mínimo (excepto posiblemente raíz y hojas) tiene como mínimo M/2 hijos.
- La raíz tiene al menos 2 hijos si no es un nodo hoja.
- Todos los nodos hoja aparecen al mismo nivel.
- En cada subárbol con m hijos el número de etiquetas asociadas a la raíz es m-1.
- Sean e_0, e_1, e_2, …, e_{m-2} las etiquetas asociadas a la raíz y h_0, h_1, h_2, …, h_{m-1} los hijos entonces se cumple
    - Para cada *i* en *[0,m-2]* la etiqueta e_i es mayor que las contenidas en le subárbol h_i y menor que las contenidas en h_{i+1}.

5.7 Otros usos de los árboles: la estructura UnionFind.

Esta es una estructura muy adecuada para mantener y gestionar un conjunto de subconjuntos disjuntos. Esta dotada de las operaciones:

- *void addElement(T e)*: Crea un conjunto con el element e y lo añade a la estructura.
- *T find(T e)*: Devuelve el representante del conjunto al que pertenece e. Todos los elementos que están en el mismo conjunto tienen el mismo representante.
- *void union(T e1, T e2)*: Une en uno los dos conjuntos a los que pertenecen e1 y e2.

El constructor

- *UnionFind(Set<T> elements):* Construye las estructura con un conjunto por cada elemento en elements.

Por las operaciones disponibles podemos representar todos los objetos que pertenecen a un mismo conjunto como un árbol. La raíz será la representante de todos ellos. Para buscar el representante de un elemento se trata de seguir el camino de cada nodo a su padre hasta uno, la raíz, que la hacemos padre de si mismo. Para conseguir que el árbol que representa cada conjunto tenga la mínima altura posible, en al operación *find*, se hace cada nodo hijo del antecesor de su padre hasta conseguir que todos los nodos de esa rama sean hijos de la raíz.

La unión se consigue haciendo que el árbol de menor altura sea hijo del de mayor altura. En este caso concreto sólo nos interesan de los árboles la propiedad de padre de cada nodo y altura del mismo. Por esta razón todos los árboles se pueden representar por dos funciones: *parentMap y rankMap*. El primero nos da la etiqueta padre de una dada y el segundo una aproximación a la altura del árbol con una etiqueta dada.

El código, tal como se ha implementado en *jGrapht* es:

```
public class UnionFind<T> {
    private Map<T, T> parentMap;
    private Map<T, Integer> rankMap;
```

```
    public UnionFind(Set<T> elements){
        parentMap = new HashMap<T, T>();
        rankMap = new HashMap<T, Integer>();
        for (T element : elements) {
            parentMap.put(element, element);
            rankMap.put(element, 0);
        }
    }

    public void addElement(T element){
        parentMap.put(element, element);
        rankMap.put(element, 0);
    }

    public T find(T element){
        if (!parentMap.containsKey(element)) {
            throw new IllegalArgumentException(
                "elements must be contained in given set");
        }

        T parent = parentMap.get(element);
        if (parent.equals(element)) {
            return element;
        }

        T newParent = find(parent);
        parentMap.put(element, newParent);
        return newParent;
    }

    public void union(T element1, T element2){
        if (!parentMap.containsKey(element1)
            || !parentMap.containsKey(element2))
        {
            throw new IllegalArgumentException(
                "elements must be contained in given set");
        }

        T parent1 = find(element1);
        T parent2 = find(element2);

        if (parent1.equals(parent2)) {
            return;
        }

        int rank1 = rankMap.get(parent1);
        int rank2 = rankMap.get(parent2);
        if (rank1 > rank2) {
            parentMap.put(parent2, parent1);
        } else if (rank1 < rank2) {
            parentMap.put(parent1, parent2);
        } else {
            parentMap.put(parent2, parent1);
            rankMap.put(parent1, rank1 + 1);
        }
    }
}
```

6. Vistas y su Implementación

De una manera muy general podemos decir que las **vistas** son objetos para los que los valores de sus propiedades están ligados a los valores que en cada momento tienen los valores de las propiedades de otro objeto (llamado objeto ligado). Las vistas de un objeto nos permiten restringir las operaciones a realizar sobre otro, limitar los elementos a los que se pueden acceder en un agregado o definir restricciones adicionales a las operaciones del contrato. El objeto que es vista de otro objeto al que está ligado puede ser del mismo tipo o de otro tipo diferente.

Podemos considerar diferentes tipos de vistas:

- Un primer tipo es la **vista con invariante**. En este caso la vista y objeto ligado son agregados de objetos y donde los objetos de la vista son los del objeto ligado pero filtrados por un invariante. Desde este punto de vista una variable *s* es una vista de otra *r* cuando ambas son agregados de objetos y están ligadas en el sentido de que en cada momento los objetos en *s* (la vista) son los de *r* que cumplan un invariante dado. Es decir si modificamos la variable *r* los valores de la *s* quedan actualizados a los de *r* que cumplan el invariante. Igualmente ocurre si modificamos *s* pero a esta no podemos añadir objetos que no cumplan el invariante. Esta idea de vista con invariante puede ser extendida a tipos que no sean agregados de datos.
- Un segundo tipo donde los valores de la vista y el original son los mismos pero la vista sólo ofrece un subconjunto de los métodos disponibles. Son las vistas restringidas. Ejemplos de vistas del segundo tipo son las **vistas no modificables** de un agregado dado.
- Un tercer tipo donde los valores y los métodos son los mismos en la vista y en el objeto ligado pero los métodos de la vista tienen alguna semántica adicional. Ejemplos del tercer tipo son las **vistas concurrentes**.

6.1 Tipo inmutable

Junto a los conceptos anteriores de vistas recordaremos el concepto de **tipo inmutable**. Como hemos visto en temas anteriores un tipo es inmutable cuando sólo tiene métodos observadores y sus valores no pueden cambiar. Las diferencias entre los valores de un tipo inmutable y los de una vista no modificable deben estar claras. En una vista no modificable los valores pueden cambiar si cambian los de la variable ligada aunque no podamos usar los métodos modificadores de la vista. Los valores de un tipo inmutable no pueden cambiar una vez que se crean.

6.2 Vistas en el API de Java

La *API* de Java ofrece muchos tipos de vistas. El contrato *Collection* define una única vista, que es un iterador. El contrato *List* define dos nuevas vistas. Ambas se describen en la siguiente tabla

Vistas propias del contrato List< E>	
ListIterator<E> listIterator()	Iterador de lista con operaciones adicionales.
List<E> subList(int from, int to)	Sublista que solo permite trabajar con los elementos de [from, to)

El contrato *Set* no define ninguna nueva vista. Sin embargo, los conjuntos ordenados sí definen un conjunto de vistas adicionales.

Vistas del contrato SortedSet< E>	
SortedSet<E> headSet(E toElement)	Vista que incluye los elementos comprendidos desde el primero hasta el indicado
SortedSet<E> tailSet(E toElement)	Vista que incluye los elementos comprendidos desde el indicado hasta el último.
SortedSet<E> subSet(E fromElement, E toElement)	Vista que incluye los elementos comprendidos en [from, to).

Igualmente ofrecen vistas *NavigableSet, Map, SortedMap, NavigableMap,*

Igualmente con los métodos de la clase *Collections (unmodifiableList, unmodifiableSortedSet, unmodifiableSet, synchronizedList, synchronizedSet, synchronizedSortedSet, etc.)* es posible conseguir vistas no modificables o sincronizadas.

Versiones inmutables para los tipos más usuales se pueden conseguir en la librería en *Guava.* Ejemplos son: *ImmutableList, ImmutableSet*, etc.

6.3 Implementación de vistas

Según la forma de la implementación podemos clasificar las vistas en acopladas y desacopladas. En las primeras, vista y objeto ligado están estrechamente ligados, mientras en el segundo caso la vista y el objeto ligado se comunican mediante eventos.

Veamos, en primer lugar, ideas para implementar vistas acopladas. Este tipo de vistas se implementan mediante el *Patrón Delegado (Delegate)* mas una factoría para crear las vistas. Esencialmente para las vistas acopladas se trata de diseñar una clase interna, que implemente el tipo adecuado con las precondiciones e invariantes requeridos y delegue muchos de sus métodos en los de la clase original. La factoría creará y devolverá un objeto de esa clase interna.

Veamos un ejemplo. Vamos a implementar la vista devuelta por el método *subList* del tipo *List*. El primer paso es tener claro cuáles son las restricciones que va introducir la vista. A continuación, se muestran dichas restricciones expresándolas como ejemplos. A partir de estos ejemplos sería automático generar casos de prueba que validaran la implementación de la vista. Por simplicidad no se muestra el tipo de los objetos en la lista aunque se puede deducir del contexto.

Código	Resultado esperado
Integer[] i = {1, 2, 3, 4, 5, 6}; List l = new ArrayList(Arrays.asList(i)); List vista = l.subList(2, 5);	Vacío
System.out.println(vista);	[3, 4, 5]
System.out.println(vista.get(0) + ", " + vista.get(1));	3, 3
System.out.println(vista.size());	3
vista.add(0, 0); System.out.println(vista); System.out.println(l);	[0, 3, 4, 5] [1, 2, 0, 3, 4, 5, 6]
vista.remove(2); vista.remove(2); System.out.println(vista); System.out.println(l);	[0, 3] [1, 2, 0, 3, 6]

Después, creamos una nueva clase que implemente la interfaz *List*. Ahora es el momento de implementar cada uno de los métodos del contrato (en este caso List), utilizando el patrón decorador. Para emplear este patrón, todos los métodos de la implementación de la vista deben llamar a métodos de la clase original añadiendo el código necesario. Una posible implementación, basada en el propio código fuente de Java, para la clase interna que implementa la sublista se muestra a continuación.

```
import static com.google.common.base.Preconditions.*;

class SubList<E> {
    private List<E> l;
    private int offset;
    private int size;
    private int expectedModCount;

    SubList(List<E> list, int fromIndex, int toIndex) {
        checkElementIndex(fromIndex, list.size(),
             "Index: "+fromIndex+",Size: "+list.size());
        checkPositionIndex(toIndex, list.size(),
             "Index: "+toIndex+",Size: "+list.size());
       checkArgument(fromIndex > toIndex, "fromIndex(" + fromIndex + ") >
             toIndex(" + toIndex + ")");
        l = list;
        offset = fromIndex;
        size = toIndex - fromIndex;
    }

    public E get(int index) {
        checkElementIndex(index, size, "Index: "+index+",Size: "+size);
        return l.get(index+offset);
    }

    public int size() {
        return size;
    }

    public void add(int index, E element) {
        checkPositionIndex(index, size);;
```

```
        l.add(index+offset, element);
        size++;
    }

    public E remove(int index) {
        checkElementIndex(index, size, "Index: "+index+",Size: "+size);
        E result = l.remove(index+offset);
        size--;
        return result;
    }

    // Continúa ……
}
```

La implementación anterior usa la clase *Preconditions* de *Guava.* El método *subList* en una clase que implemente *List<E>* sería:

```
public List<E> subList(int fromIndex, int toIndex){
       return new SubList(this,fromIndex,toIndex);
}
```

6.4 Clases Forward de Guava

Se ha visto en la sección anterior que para implementar vistas se utilizan delegados. Para facilitar el desarrollo de estos delegados a la hora de trabajar con tipos de datos, el paquete *Guava* incluye un conjunto de clases llamadas forward. *Guava* incluye unas **clases forward** para cada tipo de dato definido en el API de Java y para cada tipo de dato definido en *Guava*.

Todas las clases forward de Guava siguen un mismo patrón. Todas implementan un contrato y delegados de ese contrato. Estas clases son útiles cuando se crean vistas con restricciones que afectan sólo a algunos métodos. Esencialmente se trata de heredar de esas clases y sobrescribir el método correspondiente.

Como ejemplo, vamos a implementar una vista del contrato *List<E>* que imponga como restricción que no se puede decrementar el tamaño de la lista. Eso significa que los métodos *remove*, *removeAll* y *clear* deben lanzar una excepción mientras que el resto de los métodos se comportará normalmente. El código resultante utilizando la clase *ForwardingList<E>* de *Guava* se muestra a continuación.

```
class NoRemoveList<E> extends ForwardingList<E> {
       List<E> l;
       public NoRemoveList(List<E> l) {
              this.l = l;
       }
       @Override protected List<E> delegate() {
              return l;
       }
       public E remove(int index) {
              throw new UnsupportedOperationException();
              return null;
```

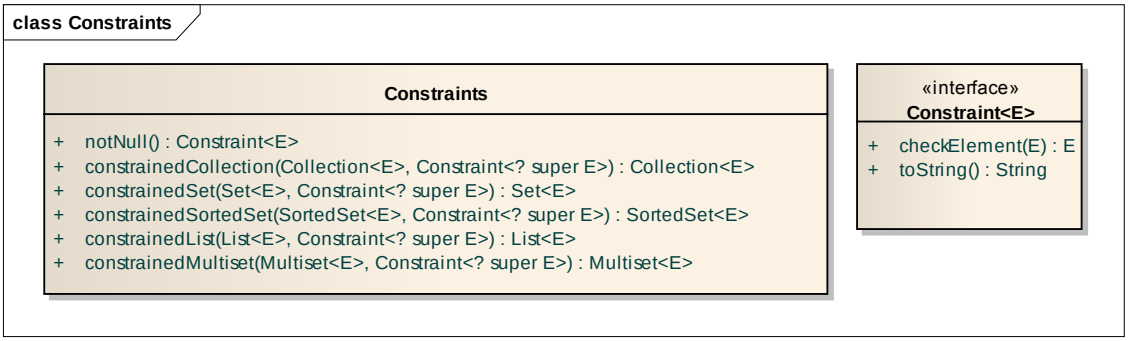
```
        }
        public boolean removeAll() {
                throw new UnsupportedOperationException();
                return false;
        }
        public void clear() {
                throw new UnsupportedOperationException();
        }
}
```

Como puede verse, gracias a ForwardList sólo es necesario volver a implementar los métodos que modifican su comportamiento, en vez de todos los métodos del contrato. Esta solución no está completa, ya que es necesario modificar todas las vistas para no poder eliminar elementos a partir de ellas, pero esa modificación se deja propuesta como ejercicio.

6.5 Colecciones con restricciones

Guava ya ofrece un mecanismo para crear colecciones cuyos elementos cumplen una restricción. Para definir una restricción se usa la interfaz *Constraint* y la clase *Constrains* que incluye métodos factorías para crear colecciones con restricciones.

El tipo *Constraint<E>* tiene el método *E checkElement(E e)* verifica si el elemento recibido como parámetro cumple la restricción. Si no la cumple, el método debe lanzar una excepción y, si la cumple, el método debe devolver el mismo elemento que recibe por parámetro. Las diversas restricciones se obtienen implementando esa interfaz. La clase *Constraints* incluye métodos para crear colecciones con restricciones pasándole como parámetro un objeto que implemente *Constraint<E>.* Los métodos de la clase *Constraints* se implementan siguiendo ideas similares al caso de las vistas.



A modo de ejemplo, el siguiente código crea una lista que no permite valores nulos:

```
List<String> l =
  Constraints.constrainedList(new ArrayList<String>(), Constraints.notNull());
```

6.6 Objetos Escuchables y Vistas Desacopladas.

Existe una segunda forma de crear vistas. Es haciendo que el objeto original produzca un conjunto de eventos, la vista los reciba y a partir de ellos actualice su estado. Para implementar este mecanismo debemos explicar previamente el *Patrón Observador-Observable.* En este patrón existen dos roles: *Listenable y Listener*. El rol *Listenable* ofrece fundamentalmente dos métodos:

- *void addListener(Listener e):* Añade un objeto interesado en el evento correspondiente
- *void removeListener(Listener e):* Elimina el objeto como interesado en el evento correspondiente.

El rol *Listener* ofrece un método:

- *void update(Event e):* Actualiza su estado al recibir el evento e.

La idea central es que el objeto observado (*Listenable*) envía un mensaje a todos los objetos que se han suscrito cuando un evento previamente diseñado ocurre. Por ese mecanismo el objeto observador (*Listener*) puede actualizar su estado para mantener una vista del objeto observado o simplemente tener en cuenta los eventos ocurridos el objeto observado. Un ejemplo lo tenemos en el tipo *ListenableGraph<V,E>* que ofrece el API de *jgrapht.*

El primer paso para que un objeto ofrezca una vista *listenable* es decidir el evento o eventos que puede ser de interés para posibles *listeners*. Posteriormente hacer que el objeto original ofrezca una vista *listenable* y los posibles objetos interesados una vista como *listeners*. Veamos como convertir un conjunto en observable de los eventos que ocurre cuando se añade o se elimina con éxito un elemento.

En primer lugar diseñamos las interfaces adecuadas:

```
public interface ListenableSet<E> extends Set<E> {
	void addSetAddListener(SetAddListener<E> s);
	void addSetRemoveListener(SetRemoveListener<E> s);
	void removeSetAddListener(SetAddListener<E> s);
	void removeSetRemoveListener(SetRemoveListener<E> s);
}

public interface SetAddListener<E> {
	void elementAdded(SetAddEvent<E> e);
}

public interface SetRemoveListener<E> {
	void elementRemoved(SetRemoveEvent<E> e);
}

public class SetAddEvent<E> extends EventObject {
	private E element;
	public SetAddEvent(Object source, E e) {
		super(source);
		element=e;
```

```
	}
	public E getElementAdded() {
		return element;
	}
}

public class SetRemoveEvent<E> extends EventObject {
	private E element;
	public SetRemoveEvent(Object source, E e) {
		super(source);
		element=e;
	}
	public E getElementRemoved() {
		return element;
	}
}
```

Luego las clases que implementan el objeto *listenable y listener.* Una implementación para un *ListenableSet* set puede ser:

```
public class ListenableHashSet<E> extends HashSet<E> implements
	ListenableSet<E> {
	private List<SetAddListener<E>> addListeners;
	private List<SetRemoveListener<E>> removeListeners;

	public ListenableHashSet() {
		super();
		addListeners = Lists.newArrayList();
		removeListeners = Lists.newArrayList();
	}

	public ListenableHashSet(Collection<? extends E> arg0) {
		super(arg0);
		addListeners = Lists.newArrayList();
		removeListeners = Lists.newArrayList();
	}

	public ListenableHashSet(int arg0, float arg1) {
		super(arg0, arg1);
		addListeners = Lists.newArrayList();
		removeListeners = Lists.newArrayList();
	}

	public ListenableHashSet(int arg0) {
		super(arg0);
		addListeners = Lists.newArrayList();
		removeListeners = Lists.newArrayList();
	}

	@Override
	public void addSetAddListener(SetAddListener<E> s) {
		addListeners.add(s);
	}

	@Override
	public void addSetRemoveListener(SetRemoveListener<E> s) {
		removeListeners.add(s);
	}
```

```
	@Override
	public void removeSetAddListener(SetAddListener<E> s) {
		addListeners.remove(s);
	}

	@Override
	public void removeSetRemoveListener(SetRemoveListener<E> s) {
		removeListeners.remove(s);
	}

	private void notifySetAddedListeners(E element){
		SetAddEvent<E> e = new SetAddEvent<E>(this, element);
		for(SetAddListener<E> ls : addListeners){
			ls.elementAdded(e);
		}
	}

	private void notifySetRemoveListeners(E element){
		SetRemoveEvent<E> e = new SetRemoveEvent<E>(this, element);
		for(SetRemoveListener<E> ls : removeListeners){
			ls.elementRemoved(e);
		}
	}

	public boolean add(E element){
		boolean r = super.add(element);
		if(r){
			notifySetAddedListeners(element);
		}
		return r;
	}

	@SuppressWarnings("unchecked")
	public boolean remove(Object element){
		boolean r = super.remove(element);
		if(r){
			notifySetRemoveListeners((E)element);
		}
		return r;
	}
}
```

Y la clase que implementa el listener:

```
public class Counter<E> implements SetAddListener<E>, SetRemoveListener<E> {

	List<E> elementsAdded;
	List<E> elementsRemoved;

	public Counter() {
		super();
		this.elementsAdded = Lists.newArrayList();
		this.elementsRemoved = Lists.newArrayList();
	}

	@Override
	public void elementRemoved(SetRemoveEvent<E> e) {
		// TODO Auto-generated method stub
```

```
            elementsRemoved.add(e.getElementRemoved());
        }

        @Override
        public void elementAdded(SetAddEvent<E> e) {
            // TODO Auto-generated method stub
            elementsAdded.add(e.getElementAdded());
        }

        public String toString(){
            String s = "";
            s = elementsAdded+"\n"+elementsRemoved;
            return s;
        }
}
```

Y el programa principal:

```
public static void main(String[] args) {
            ListenableSet<Double> s = new ListenableHashSet<Double>();
            Counter<Double> c = new Counter<Double>();
            s.addSetAddListener(c);
            s.addSetRemoveListener(c);
            for(double r = 0. ; r<5.; r=r+0.5){
                s.add(r);
            }
            for(double r = 1. ; r < 5.; r=r+0.5){
                s.remove(r);
            }
            System.out.println(s);
            System.out.println(c);
        }
```

7. Ejercicios propuestos

1. Implemente el tipo *Set<E>* a partir del tipo *BasicHashTable<K,V>.* Calcule las complejidades de las diferentes operaciones en función del tamaño del conjunto.
2. Realice una segunda implementación del tipo *Set<E>* a partir del tipo *BasicHashTable<K,V>* con la restricción adicional de que la iteración sobre los elementos mantenga el orden de inserción. Calcule las complejidades de las diferentes operaciones en función del tamaño del conjunto.
3. Implemente el tipo *List<E>* a partir del tipo *DynamicArray<E>.* Calcular las complejidades de las diferentes operaciones en función del tamaño del conjunto. Considere el concepto de coste amortizado como el coste promedio de ejecutar una operación un gran número de veces.
4. Implemente el tipo *List*<E> a partir del tipo *BasicLinkedList*<E>. Calcule las complejidades de las diferentes operaciones en función del tamaño del conjunto.
5. Implemente el tipo *Set<Integer>*, donde los enteros que forman el conjunto están en el rango *[a1,a2)*, a partir del tipo *Bitset.* Calcule las complejidades de las diferentes operaciones en función del tamaño del conjunto.

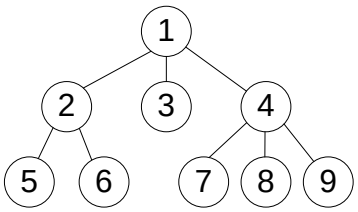
6. Implemente el tipo *Set<E>* a partir del tipo *BasicHashTable<K,V>.* Calcular las complejidades de las diferentes operaciones en función del tamaño del conjunto.
7. Implemente el tipo *Map<K,V>* a partir del tipo *BasicHashTable<K,V>.* Calcular las complejidades de las diferentes operaciones en función del tamaño del conjunto de las claves.
8. Implemente los tipos *Queue<E>, Stack<E>, PriorityQueue<E>.* Sumimos que implementan la interfaz *BasicCollection<E>* siguiente con la semántica correspondiente. Eleja en cada caso cuál es mejor tipo para llevar a cabo la implementación.

```
interface BasicCollection<E> {
	int size();
	boolean isEmpty();
	E element();
	boolean add(E e);
	E remove();
	boolean contains(E e);
}
```

9. Implemente los métodos de la clase *Trees* comentados anteriormente.
10. Implemente los métodos para realizar rotaciones a la izquierda o a la derecha en la clase AVLTree. Esta clase se implementará usando el tipo *Tree<E>* anterior.
11. Implemente los métodos para realizar inserciones, eliminar elementos o encontrarlos en la clase AVLTree.
12. Implemente el tipo SortedSet<E> usando la clase AVLTree anterior.
13. Implemente los métodos *containsValue, get, put, remove, size* y *values* del tipo *Multimap<K,V>* basándose en el tipo *Map<K,V>*.
14. Dada las siguientes secuencias de nodos obtenidos en preorden, inorden y postorden de un árbol binario cuyas etiquetas son caracteres:
    - preorden: A-D-K-E-B-G-L-M-O-F-U-Y
    - inorden: K-D-E-G-B-A-M-O-L-U-F-Y
    - postorden: K-G-B-E-D-O-M-U-Y-F-L-A

    Se pide dibujar el correspondiente árbol binario. Diseñe un método que recibiendo la secuencia de nodos en preorden o en inorden, devuelva el *Tree* correspondiente a dichas secuencias.

15. Escriba un algoritmo que tomando como argumento un árbol (Tree<T>), imprima la información contenida en cada nodo por niveles. Es decir, primero se debe imprimir la información de la raíz (nivel 0), luego la de sus hijos (nivel 1), después la información de los hijos de estos (nivel 2) y así sucesivamente. Por ejemplo, para el siguiente árbol, el resultado sería: 1 2 3 4 5 6 7 8 9.

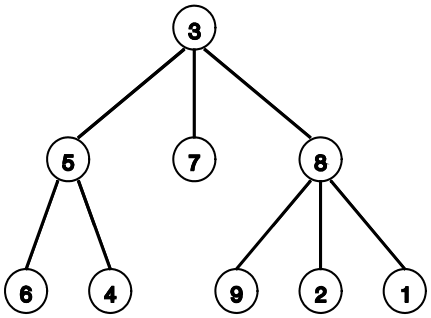


16. Escriba un algoritmo que dados dos árboles, compruebe si dichos árboles (*Tree<T>)* son exactamente iguales (tanto en estructura como en contenido), dando el correspondiente mensaje. Sólo se pueden usar todas las operaciones del tipo *Tree<T>* salvo el método *equals*.

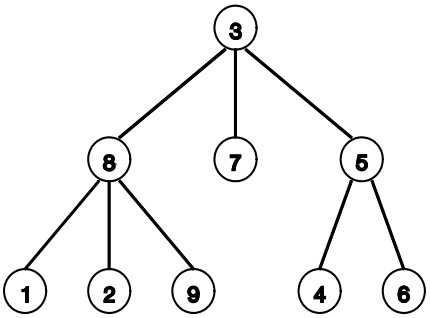
17. ¿Para qué pueden ser útiles los árboles B? Ponga un ejemplo.

18. Realice una función recursiva que aplicada a un árbol, devuelva un nuevo árbol sea la imagen especular del primero.

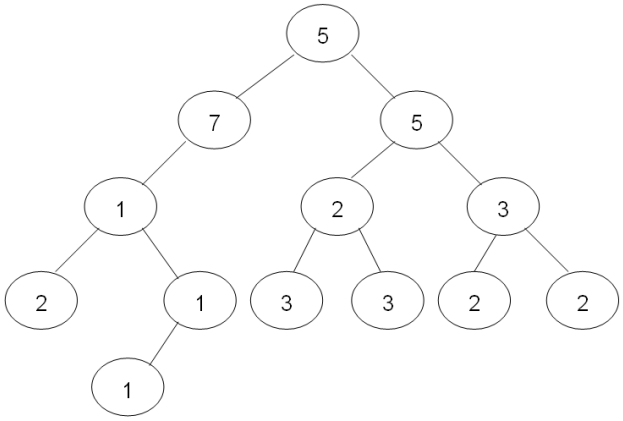
Árbol de entrada



Árbol de salida

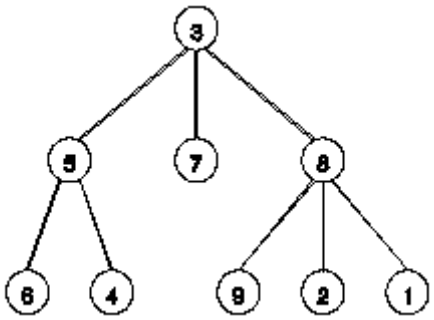


19. Un árbol monodistante de orden N es un árbol que almacena números enteros en el que la suma de los valores de los nodos de cada camino que va desde la raíz a un nodo hoja es igual a N. En el siguiente ejemplo se presenta un árbol monodistante de orden 15. Implemente un método que compruebe si un árbol es monodistante de un determinado orden.



20. Realice un método que recibiendo un Tree<T> y un elemento de tipo T, devuelve una lista con los hermanos de dicho elemento. Suponga que en el árbol no hay elementos repetidos (cada elemento sólo aparece una vez en el árbol completo).

    Ejemplo:



    La salida del método para este árbol y el elemento 2 será:{9,1}

21. Proponga una implementación posible para el tipo *Graph<V,E>.* Discuta los casos posibles según el subtipo de grafo considerado.