Ficha 11

Árboles Binarios de Búsqueda

1.] Arboles binarios: necesidades de uso.

Para ciertos problemas en los que el conjunto de datos y/o resultados crece de forma impredecible, las listas simples pueden ayudar a plantear soluciones muy eficientes en cuanto al uso de memoria. El típico caso del conteo de frecuencias (determinar cuantas veces aparece cada clave de un conjunto de datos de entrada) es un claro ejemplo de la ventaja que las listas aportan sobre los arreglos cuando se desconoce la cantidad de datos que deberán procesarse. El algoritmo básico para ese problema (usando una lista) podría ser el siguiente:

i.) Cargar una clave.

ii.) Buscar la clave en la lista.

iii.) ¿Existe la clave en la lista?

Sí: entonces incrementar en uno el campo contador de su nodo en la lista.

No: entonces agregar un nodo a la lista, con dos campos: uno para la clave y el otro para contar cuántas veces apareció la misma, inicializando en 1 (uno) el campo contador.

iv.) Si quedan claves por cargar, volver al paso i.)

v.) Si ya no quedan claves, mostrar el contenido de la lista.

El algoritmo basado en una lista es claramente superior al mismo algoritmo basado en un arreglo en cuanto al uso eficiente de la memoria: la lista usará exactamente la memoria que necesite, en cambio el arreglo deberá crearse haciendo una estimación de la cantidad de claves distintas que entrarán (y esa estimación podría quedar corta). Sin embargo, ambos modelos son esencialmente iguales en cuanto al tiempo usado para buscar una clave: en ambas situaciones, el peor caso se da cuando la clave no está en la estructura usada como soporte, o se encuentra muy al final, y deben recorrerse los n componentes comparando uno a uno: ambas soluciones son O(n) en cuanto al tiempo de búsqueda.

Nos preguntamos si es posible mejorar ese rendimiento en el tiempo de búsqueda, pero manteniendo la ventaja del uso eficiente de la memoria. Podríamos pensar en usar un arreglo, pero ordenado por clave, y usar búsqueda binaria cuando entra una clave para ser contada. La búsqueda binaria es O(log(n)) y por lo tanto sería una gran mejora, pero al usar un arreglo volveríamos al problema del uso inadecuado de la memoria.

La solución consiste en introducir una nueva estructura de datos, que permita el mismo manejo eficiente de memoria que las listas, pero reduzca en forma considerable el tiempo de búsqueda y de ser posible, que lo lleve a O(log(n)). Esas estructuras pueden ser los

árboles binarios (en particular, la variante conocida como árbol binario de búsqueda), acerca de los cuales trata esta sección de la Ficha.

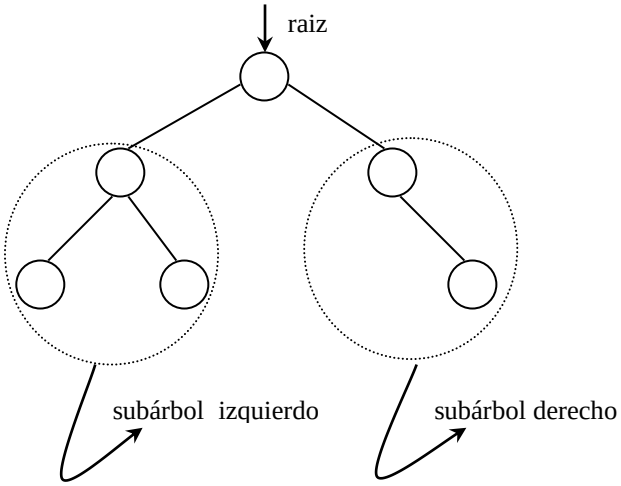
2.] Arboles binarios: definiciones y conceptos.

Un Arbol Binario es una estructura de datos no lineal, que puede estar vacía, o que puede contener un nodo llamado la raiz del árbol. Si el nodo raiz existe, de él se desprenden dos subconjuntos de elementos, con las siguientes características:

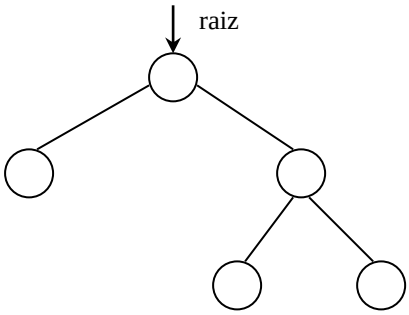
a.) Ambos subconjuntos tienen intersección vacía: es decir, ningún nodo contenido en uno de los subconjuntos, puede estar también en el otro.

b.) Ambos subconjuntos son a su vez árboles binarios, y se designan respectivamente como *subárbol izquierdo* y *subárbol derecho*.

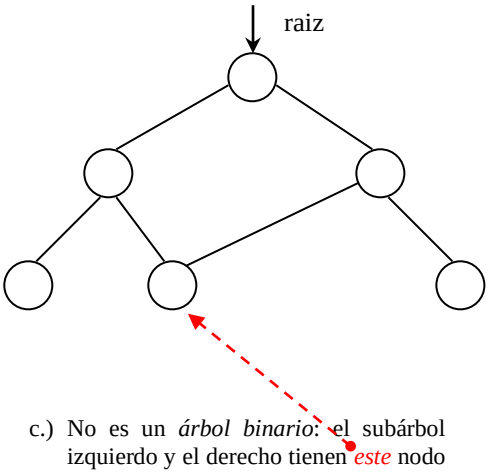
El siguiente gráfico muestra algunos ejemplos de figuras que representan árboles binarios, y figuras que no los representan pues no respetan en algún punto la definición anterior:



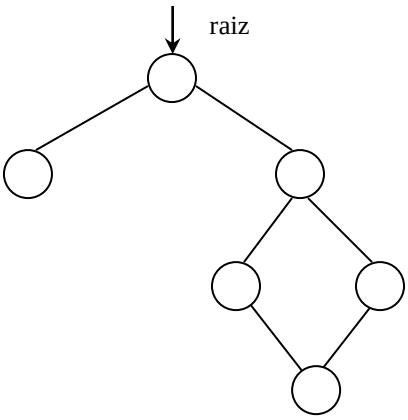
a.) Ejemplo correcto de *árbol binario*, marcando sus dos subárboles



b.) Otro ejemplo correcto



c.) No es un *árbol binario*: el subárbol izquierdo y el derecho tienen *este* nodo en común (no respeta la regla a).

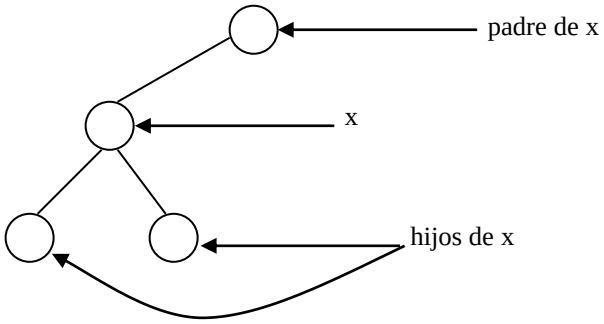


d.) No es *árbol binario*: el subárbol de la derecha no es un árbol binario, y por lo tanto no se cumple la regla b).

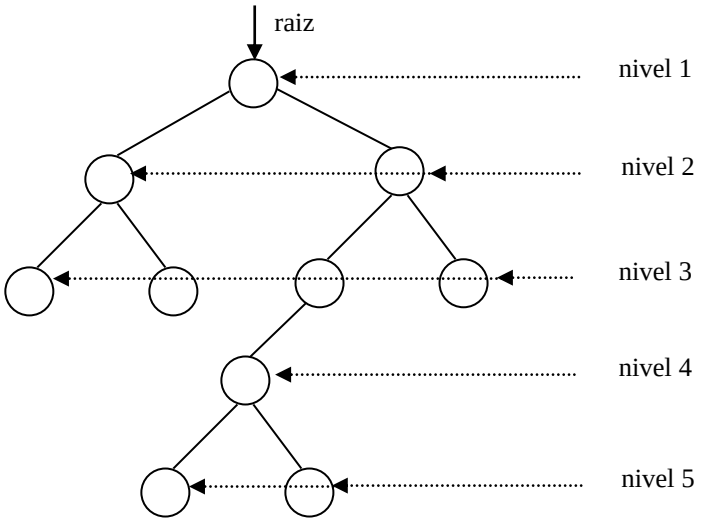
Observemos que la definición de árbol binario es recursiva, pues según la regla b), un árbol binario está compuesto de otros dos árboles binarios... Como vemos, el objeto que se quiere definir forma parte de la definición, y esta es la esencia de una definición recursiva. Esto debe ser tenido en cuenta a la hora de intentar procesar el árbol binario en forma completa, pues los algoritmos para hacerlo se aplican sobre el árbol principal primero, y luego recursivamente sobre cada uno de los subárboles.

En el trabajo con árboles binarios se maneja una terminología muy específica, aunque sencilla de definir y comprender. Haremos una breve referencia a los principales términos y conceptos usados:

- ✓ Los enlaces o uniones entre los distintos nodos, se llaman *arcos*. Dependiendo de la forma en que se implemente el árbol esos *arcos* podrán ser punteros, a la manera de los punteros *next* en los nodos de una lista.
- ✓ Dado un nodo *x*, los dos nodos inmediatamente debajo de *x* (y unidos a él con sendos arcos) se llaman *hijos de x*. El nodo de cual depende *x* hacia arriba, se dice el *padre de x*. De las reglas dadas en la definición, se deduce que en un árbol binario ningún nodo puede tener más de un padre, y todo nodo puede tener dos hijos, un hijo, o ningún hijo. Tómese un tiempo el lector para comprender el porqué de estas afirmaciones.



- ✓ Se llama *nivel de un nodo*, a la cantidad de arcos que deben seguirse para llegar hasta ese nodo comenzando desde el arco para entrar a la *raiz*. El concepto de *nivel* da una idea formal de la división del árbol en "franjas horizontales" o "pisos". Así, el nodo *raiz* es alcanzable con un sólo arco, y por lo tanto se dice que está en el *nivel 1 del árbol*. Los dos hijos del nodo raiz (si existen) son alcanzables con dos arcos cada uno, y por lo tanto se dice que ambos están en el *nivel 2 del árbol*, y así con el resto (ver figura siguiente).



- ✓ Se llama *altura de un árbol* al número de nivel máximo del mismo. En el árbol de la figura anterior la altura es *h = 5*, pues el número máximo de nivel de ese árbol es *5*. Veremos más adelante, en ocasión del tema *árboles binarios de búsqueda*, que el control de la altura es un elemento muy importante a tener en cuenta en el rendimiento del tiempo de búsqueda en un árbol.
- ✓ La *implementación* de un *árbol binario* puede hacerse en forma similar a lo realizado con las listas simples: los nodos pueden representarse como objetos de una clase *Nodo*, de forma que cada nodo tendrá *tres atributos*: en uno de ellos (el atributo *info*) se guarda la información propia del nodo (y a los efectos del polimorfismo en nuestra implementación será una referencia a *Comparable*), y los otros dos son referencias con las direcciones de los nodos hijos. Esas referencias a los hijos pueden valer *null* si el hijo correspondiente no existe.
- ✓ Luego, una clase *Arbol* se puede usar para representar al árbol mismo, conteniendo una referencia *raiz* que se usará para contener la dirección del primero nodo del árbol (de forma análoga al puntero *frente* en las listas). De hecho, lo único que se necesita declarar de antemano para trabajar con el árbol es la referencia *raiz*: los nodos serán creados con el operador *new* cada vez que sea requerido, dentro de algún método de inserción de la clase que representa al árbol. Veremos la forma de llevar a la práctica esta implementación en el capitulo siguiente.

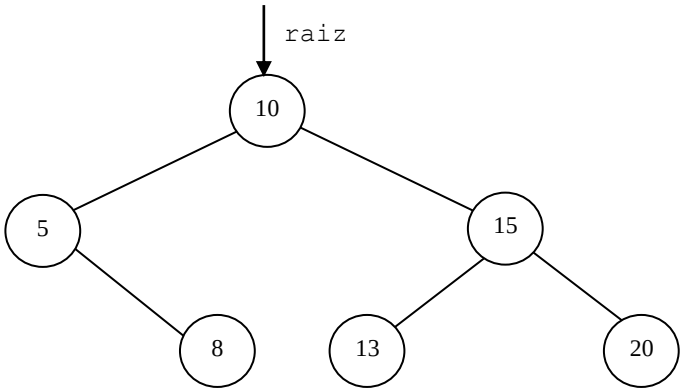
3.] Arboles binarios de búsqueda.

Los *árboles binarios* son estructuras de datos aplicables en numerosas situaciones y problemas. Quizás la aplicación más obvia se da en el campo de la *búsqueda de una clave en un conjunto*. Como vimos, si un conjunto de *n* claves está almacenado en una lista, una búsqueda llevará podríallevar en el peor caso a *n* comparaciones (*O(n)*), pues en ese peor caso se deben comparar todas las claves. Sin embargo, usando un *árbol binario* (y un poco de ingenio), se pueden lograr tiempos de búsqueda sustancialmente mejores: si el árbol se organiza de la manera adecuada, el tiempo de búsqueda puede bajarse a *O(log(n))*.

El secreto para lograr esto, es tratar de organizar las claves de forma que al hacer una comparación, puedan desecharse otras claves sin tener que comparar contra ellas (esto es lo que hace, por ejemplo, la *búsqueda binaria* en un arreglo ordenado: si al comparar contra un elemento del arreglo no se encuentra la clave, se sigue buscando a la derecha o a la izquierda según la clave sea menor o mayor, pero no se busca en todo el arreglo). Con un árbol binario puede lograrse una organización de claves que favorezca una búsqueda rápida, siguiendo una regla sencilla: para insertar una clave *x*, se compara primero contra la clave del nodo *raiz*. Si *x* es mayor, se intenta insertar la clave en el *subárbol derecho*, pero si *x* es menor, se intenta la inserción en el *subárbol izquierdo*. En cada subárbol se repite la misma estrategia contra la raiz del subárbol, hasta que en algún momento se llegue a un camino nulo, y en ese lugar se inserta la clave. Un árbol binario creado con esta regla de *menores a la izquierda, mayores a la derecha*, se suele llamar *árbol binario de búsqueda*, o también *árbol ordenado*. El gráfico de la página siguiente muestra un árbol binario de búsqueda, en el cual las claves se insertaron en esta secuencia: {10, 15, 20, 5, 13, 8}.

Como puede verse, para cada nodo del árbol se cumple que todos sus descendientes por la izquierda son menores que él, y todos sus descendientes por la derecha son mayores. De esta forma, buscar una clave en el árbol resulta una tarea sencilla y rápida: por ejemplo, si nos dieran a buscar la clave *x = 13*, deberíamos comenzar por la *raiz* y comparar contra el *info* de ella. Como el valor almacenado en la raiz es *10*, y *13* es mayor a *10*, la búsqueda sigue

por el *hijo derecho* del nodo *raiz*, que es el *15*. Dado que ese nodo tampoco contiene al *13*, y *13* es menor que *15*, la búsqueda sigue por la izquierda del *15*, donde encontramos al *13*, y allí termina la búsqueda con éxito.



Si observamos, para buscar la clave no fue necesario mirar todos los nodos, sino sólo uno en cada nivel del árbol. Esto reduce de manera dramática la cantidad de comparaciones a realizar, comparado contra una búsqueda secuencial en la que se revisan todos los nodos en el peor caso. Puede probarse que si el árbol cumple con ciertas condiciones de *balance* o *equilibrio* (que discutiremos más adelante), el tiempo para buscar una clave es *O(log(n))*.

La secuencia de nodos que efectivamente fueron *vistos* al buscar la clave *x*, se llama *camino de búsqueda de la clave x*. En nuestro ejemplo, el camino de búsqueda de *x = 13* es el conjunto de nodos *{10, 15, 13}*.

En el proyecto *TSBSearchTree* que acompaña a esta ficha, se implementa de forma básica la idea del *árbol binario de búsqueda*. La clase *TreeNode* representa al nodo, y no ofrece mayor dificultad (está definida como clase interna de la clase *TSBSearchTree* que representa al árbol):

```
protected class TreeNode<E>
{
   private E info;
   private TreeNode<E> left;
   private TreeNode<E> right;

   public TreeNode(E info, TreeNode<E> left, TreeNode<E> right)
   {
      if(info == null)
      {
         throw new IllegalArgumentException("Node(): parámetro null...");
      }
      this.info = info;
      this.left = left;
      this.right = right;
   }

   public E getInfo()
   {
      return this.info;
   }
```

```
// use con cuidado... y a su propio riesgo...
private void setInfo(E info)
{
   this.info = info;
}

public TreeNode<E> getLeft()
{
   return this.left;
}

public TreeNode<E> getRight()
{
   return this.right;
}

public void setLeft(TreeNode<E> newLeft)
{
   this.left = newLeft;
}

public void setRight(TreeNode<E> newRight)
{
   this.left = newRight;
}

@Override
public int hashCode()
{
   int hash = 7;
   hash = 61 * hash + this.info.hashCode();
   return hash;
}

@Override
public boolean equals(Object obj)
{
   if (this == obj) { return true; }
   if (obj == null) { return false; }
   if (this.getClass() != obj.getClass()) { return false; }

   final TreeNode<E> other = (TreeNode<E>) obj;
   if (!Objects.equals(this.info, other.info)) { return false; }
   return true;
}

@Override
public String toString()
{
   return this.info.toString();
```

```
        }
    }
```

Note que el método *setInfo()* de la clase *TreeNode* está definido como *privado*: la idea es que el objeto o valor contenido en un nodo no debería ser cambiado sin estar seguros de que ese cambio no afectará al orden interno del árbol, y por lo tanto el método que lo hace no está disponible en forma pública: sólo otros métodos de la clase pueden accederlo, y con ello, sólo los programadores que saben perfectamente cuándo sí y cuándo no esa operación debe permitirse.

La clase *TSBSearchTree* representa al árbol de búsqueda, y se la define de forma de emular en forma básica la funcionalidad de la clase *java.util.TreeSet* (que en ese sentido es la clase nativa que puede usarse en Java si se necesita un árbol binario de búsqueda). La definición de la clase y sus atributos más esenciales se muestra a continuación:

```
public class TSBSearchTree<E> extends AbstractSet<E>
                                    implements NavigableSet<E>, Cloneable, Serializable
{
    // referencia al nodo raíz del árbol...
    private TreeNode<E> root;

    // la cantidad de elementos que contiene el árbol...
    private int count;

    // conteo de operaciones de cambio de tamaño (fail-fast iterator).
    protected transient int modCount;

    // el comparador usado para establecer el orden en el árbol, o null si el
    // árbol usa el ordenamiento natural (definido por compareTo())...
    private final Comparator<? super E> comparator;

    // resto de la clase aquí…
}
```

El atributo *root* es la referencia a la raíz o nodo de entrada del árbol. El atributo *count* se usa para llevar la cuenta de la cantidad de nodos/elementos que el árbol tiene. El atributo *modCount* es el ya conocido contador de operaciones que modifican el tamaño del árbol y se usa para implementar la estrategia *fail-fast iterator* ya explicada en fichas anteriores.

El atributo *comparator* se usa para especificar qué critero será aplicado en el árbol para comparar los objetos y determinar cuándo un objeto es menor, mayor o igual que otro (esto es, para establecer una *relación de orden* entre esos objetos). Si el atributo *comparator* es *null*, entonces el árbol usará el método *compareTo()* que los objetos que se insertan deben tener implementado (esto es, la clase de esos objetos debe implementar la interface *Comparable*). Se dice que el método *compareTo()* establece un *orden natural* entre los objetos de una clase (debido a que normalmente es el método de comparación que se usa por *default*). A modo de ejemplo, supongamos la siguiente clase para representar estudiantes (que está incluida en el modelo *TSBSearchTree* que viene con esta ficha):

```
import java.util.Comparator;
```

```
public class Student implements Comparable<Student>
{
    private int id;
    private String name;

    public Student(int id, String name)
    {
       this.id = id;
       this.name = name;
    }

    public int getId()
    {
       return id;
    }

    public void setId(int id)
    {
       this.id = id;
    }

    public String getName()
    {
       return name;
    }

    public void setName(String name)
    {
       this.name = name;
    }

    public int hashCode()
    {
       int hash = 7;
       hash = 67 * hash + this.id;
       return hash;
    }

    public boolean equals(Object obj)
    {
       if (this == obj) { return true; }
       if (obj == null) { return false; }
       if (getClass() != obj.getClass()) { return false; }

       final Student other = (Student) obj;
       if (this.id != other.id) { return false; }
       return true;
    }

    public String toString()
    {
```

```
        return "Student{" + "id=" + id + ", name=" + name + '}';
    }

    public int compareTo(Student o)
    {
        return this.id - o.id;
    }

    public static Comparator<Student> comparator()
    {
        return new NameComparator();
    }

    private static class NameComparator implements Comparator<Student>
    {
        public int compare(Student o1, Student o2)
        {
            return o1.name.compareTo(o2.name);
        }
    }
}
```

Como se ve, la clase *Student* implementa la interface *Comparable*, lo cual implica que debe implementar el método *compareTo()*. Ese método (resaltado en rojo) compara los números de id de dos objetos Student para determinar si el primero es el mayor (retorna un número positivo en ese caso), o lo es el segundo (retorna un negativo en ese caso), o bien si son iguales (retorna 0 en ese caso). Si se usa este método *compareTo()* para comparar dos objetos *Student*, la relación de orden estará basada en los valores del atributo *id*: un árbol de búsqueda que use ese método para agregar objetos *Student*, quedará ordenado de acuerdo a los valores del atributo *id*.

Pero ¿qué pasaría si además de un árbol de estudiantes ordenado por el valor id, se necesitase al mismo tiempo *otro* arbol pero ordenado por el atributo *name*? Si la clase *TSBSearchTree* sólo usa el método el método *compareTo()* no habría una manera simple de hacerlo. Para casos como este, sirve la interface *Comparator* que define un método *compare()*. La idea es que una clase que implementa *Comparator* debe definir el método *compare()*, y a través de él indicar una forma de comparación alternativa. Hay muchas formas de hacerlo: nuestra clase *Student* incluye una clase interna muy sencilla (resaltada en azul en el modelo anterior) llamada *NameComparator* que implementa la interface *Comparator*. Lo único que contiene esta clase es la implementación del método *compare()*, el cual simplemente compara los nombres de los dos estudiantes tomados como parámetro y retorna el resultado de esa comparación.

Cuando en cualquier contexto se necesite usar ese método para comparar objetos *Student* (en lugar de *compareTo()*), la clase *Student* prove un método estático *comparator()*, que crea y retorna un objeto de la clase *NameComparator*.

Todo lo anterior se cierra con los constructores de la clase *TSBSearchTree*, que son los siguientes (el método auxiliar privado *init()* también se muestra aquí):

```
    public TSBSearchTree()
```

```
    {
       this.init(null, 0, 0);
       this.comparator = null;
    }

    public TSBSearchTree(Collection<? extends E> c)
    {
       this();
       this.addAll(c);
    }

    public TSBSearchTree(SortedSet<E> s)
    {
       this(s.comparator());
       addAll(s);
    }

    public TSBSearchTree(Comparator<? super E> comparator)
    {
       this.init(null, 0, 0);
       this.comparator = comparator;
    }

    private void init(TreeNode<E> r, int c, int m)
    {
       this.root = r;
       this.count = c;
       this.modCount = m;
    }
```

El primero crea un árbol inicialmente vacío, dejando el atributo *comparator* en *null*. El segundo crea un árbol copiando en él los elementos de la colección tomada como parámetro y también deja el atributo *comparator* en *null*. El tercero hace algo similar al segundo, pero inicializando el atributo *comparator* con el *comparator* de la estructura ordenada tomada como parámetro. Y el cuarto, crea un árbol vacío pero recibe como parámetro un objeto *Comparator* con el cual se inicializa el atributo *comparator* del árbol.

El siguiente ejemplo (tomado de la clase *Test* incluida en el modelo que acompaña a esta ficha) muestra una forma de usar todo lo anterior:

```
public class Test
{
      public static void main(String args[])
      {
            // algunos estudiantes para hacer pruebas...
            Student s1 = new Student(50, "Ana");
            Student s2 = new Student(10, "Luis");
            Student s3 = new Student(30, "Carla");
            Student s4 = new Student(20, "Juan");

            // un árbol de estudiantes - orden natural (por id - con compareTo())...
```

```
            TSBSearchTree<Student> t1 = new TSBSearchTree<>();
            t1.add(s1);
            t1.add(s2);
            t1.add(s3);
            t1.add(s4);
            System.out.println("Listado de estudiantes (orden: por id):");
            System.out.println(t1);

            // un árbol de estudiantes - orden no natural (por name - con compare())...
            TSBSearchTree<Student> t2 = new TSBSearchTree<>(Student.comparator());
            t2.add(s1);
            t2.add(s2);
            t2.add(s3);
            t2.add(s4);
            System.out.println("Listado de estudiantes (orden: por name):");
            System.out.println(t2);
        }
    }
```

El árbol *t1* se crea con el primer constructor, con lo que su atributo *comparator* queda en *null*. Cuando luego se usa el método *add()* (que veremos a continuación) para insertar en él objetos *Student*, ese método usará el método *compareTo()* para determinar en qué orden irán los objetos en el árbol (y como vimos, eso implica que el árbol será ordenado por el *id* de cada *Student*). Al mostrar la conversión a *String* de ese árbol, efectivamente los elementos salen ordenados por *id*:

```
Listado de estudiantes (orden: por id):
Student{id=10, name=Luis} Student{id=20, name=Juan} Student{id=30, name=Carla} Student{id=50, name=Ana}
```

Pero el árbol *t2* se crea con el cuarto constructor, tomando como parámetro el objeto retornado por *Student.comparator()* (o sea, en nuestro caso, un objeto de la clase *NameComparator*), con lo que el atributo *comparator* de este segundo árbol queda apuntando a ese *NameComparator*. Cuando luego se usa el método *add()* para insertar objetos *Student* en el árbol t2, ese método usará el método *compare()* de la clase *NameComparator* para determinar en qué orden irán los objetos en el árbol (y eso implica que el árbol será ordenado por el *name* de cada *Student*). Al mostrar la conversión a *String* de ese árbol t2, efectivamente los elementos salen ordenados por *name*:

```
Listado de estudiantes (orden: por name):
Student{id=50, name=Ana} Student{id=30, name=Carla} Student{id=20, name=Juan} Student{id=10, name=Luis}
```

El método *add()* intenta insertar un objeto *e* en el árbol. Si el objeto ya existía no lo agregará y retornará false. Sólo si no existía será agregado y retornará *true*: comienza realizando una búsqueda del objeto *e* en el árbol. La búsqueda se hace avanzando en el camino de búsqueda de *e* usando una referencia *p*, que se ajusta a la izquierda o la derecha del nodo visitado según *e* sea menor o mayor:

```
public boolean add(E e)
{
    if(e == null) { throw new NullPointerException("add(): parámetro null..."); }

    TreeNode<E> p = this.root, q = null;
```

```
        while(p != null)
        {
                E y = p.getInfo();
                if(this.compare(e, y) == 0) { break; }

                q = p;
                if(this.compare(e, y) < 0) { p = p.getLeft(); }
                else { p = p.getRight(); }
        }

        // si ya existia... retorne false.
        if(p != null) { return false; }

        // si no existia... hay que insertarlo.
        TreeNode<E> nuevo = new TreeNode<>(e, null, null);
        if(q == null) { this.root = nuevo; }
        else
        {
          if (this.compare(e, q.getInfo()) < 0) { q.setLeft(nuevo); }
          else { q.setRight(nuevo); }
        }

        this.count++;
        this.modCount++;

        return true;
    }
```

Para hacer la comparación y determinar qué objeto es menor, el método *add()* usa a su vez el método privado *compare()*. Ese método es el que "hace la magia" de aplicar *Comparable.compareTo()* (si *comparator* es *null*) o bien *Comparator.compare()* (si *comparator* es diferente de *null* y apunta a un objeto cuya clase haya implementado *Comparator*):

```
private int compare(Object k1, Object k2)
{
    return (comparator == null)? ((Comparable<? super E>)k1).compareTo((E)k2)
      : comparator.compare((E)k1, (E)k2);
}
```

El bloque de acciones de este método es completamente equivalente a:

```
if (comparator == null)
{
        return ((Comparable<? super E>)k1).compareTo((E)k2);
}
else
{
        return   comparator.compare((E)k1, (E)k2);
}
```

Si *comparator* es *null*, se aplica el método *compareTo()* entre los objetos *k1* y *k2*, pero forzando una conversión al tipo *Comparable<? super E>* por lo menos para el objeto *k1*. Esto implica que se intentará que el tipo de *k1* se convierta al tipo *Comparable*, parametrizado para cualquier clase (*signo ?*) que tenga como superclase a *E*. Con esto se asegura que *k1* pueda invocar a *compareTo()*… pero si la clase de *k1* no implementó *Comparable* ese intento de casting provocará una excepción *ClassCastException*. El segundo objeto *k2* se convierte sólo a *E*, ya que este no es el objeto que invoca a *compareTo()*, pero esa conversión sólo se ejecuta si la conversión de *k1* no lanzó una excepción. Conclusión: los objetos *k1* y *k2* deben ser instancias de una clase (la misma para ambos) que haya implementado *Comparable*, o el programa se interrumpirá con una excepción de conversión de tipos.

Si *comparator* no es *null*, entonces se invoca al método *compare()* que viene provisto por la clase del propio objeto referido por *comparator* (que fue tomado como parámetro por el constructor del árbol). También aquí los objetos *k1* y *k2* se convierten al tipo *E*.

Resuelto lo anterior, notar que el método add() además de la referencia *p* para avanzar en el árbol, usa una referencia *q* a modo de persecutor, para no perder la dirección del último nodo visitado por *p* cuando este se anule si no existe *e*. Al salir del ciclo, se verifica si *p* salió *null*, lo cual sólo puede ocurrir si *e* no existía en el árbol. En ese caso, se crea un nuevo nodo con *new*, se asigna *e* en él, y se "engancha" el mismo en el árbol de la forma siguiente: si el árbol estaba vacío (lo cual se deduce del valor del puntero persecutor *q* al terminar la búsqueda), el nuevo nodo se inserta como *root*. Y si el árbol no estaba vacío, se inserta a la izquierda o a la derecha de *q*, según sea el valor de la clave *e*. En ambos casos, suma uno a *count*, suma también uno a *modCount* y retorna *true*. Notar que si *e* estaba repetido, el algoritmo presentado simplemente termina y retorna *false*.

El método *contains()* toma un objeto *o* como parámetro y simplemente recorre su camino de búsqueda para determinar si está o no en el árbol. El método de inserción también hacía una búsqueda antes de insertar el nodo, pero en esa función la búsqueda se hacía con un puntero de persecución para no perder la dirección del nodo *padre* del que será insertado. En el método *contains()*, sólo se pretende verificar si *o* existe, y por ello el persecutor no es necesario. Las comparaciones entre objetos, obviamente, se hacen con el mismo método privado *compare()* que acabamos de describir. El método *contains()* devuelve *true* si es que *o* existe en el árbol, o devuelve la *false* en caso contrario:

```
public boolean contains(Object o)
{
    if(o == null) { return false; }

    E x = (E)o;
    TreeNode<E> p = this.root;
    while(p != null)
    {
        E y = p.getInfo();
        if(this.compare(x, y) == 0) { return true; }
        if(this.compare(x, y) < 0) { p = p.getLeft(); }
        else { p = p.getRight(); }
    }
    return false;
```

```
}
```

El método *toString()* recorre el árbol y arma una cadena ordenada de menor a mayor (según el criterio de comparación que se haya aplicado para crear el árbol), retornando al final esa cadena. El método es recursivo y aprovecha alguna característica del árbol de búsqueda, que vermos, y se incluye aquí a modo de adelanto:

4.] Recorrido de Árboles Binarios.

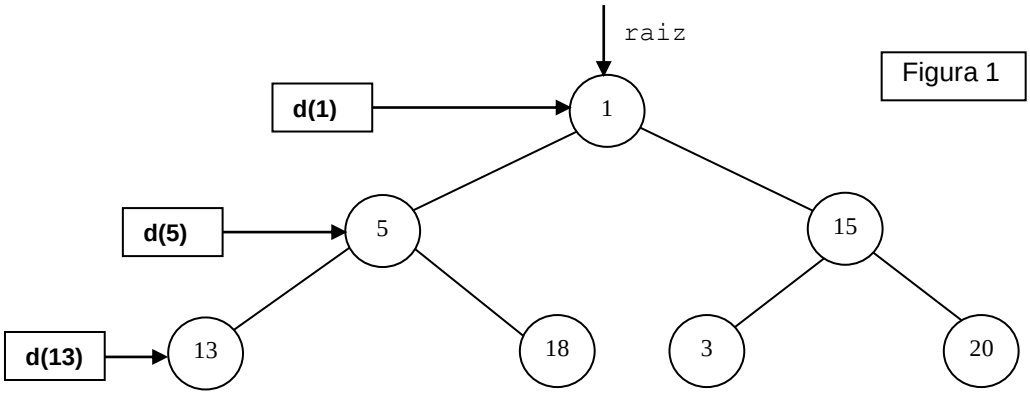
Una vez que un árbol binario ha sido creado (sea o no de búsqueda), tarde o temprano surge el problema de tener que recorrerlo en forma completa. Por ejemplo, si se pide obtener la conversión a *String* del árbol completo, el método *toString()* debería hacer un recorrido nodo a nodo de todo el árbol (y no sólo el camino de búsqueda de una clave).

El proceso de recorrido de un árbol binario no es tan sencillo como el de recorrer una lista o un arreglo unidimensional, simplemente porque un árbol *no es una estructura lineal*: cada nodo puede tener más de un sucesor, y por lo tanto a partir de un nodo hay varios caminos posibles por los que puede seguir el recorrido. En general, los problemas a los que se enfrenta un programador a la hora de desarrollar un método para recorrer un árbol binario pueden resumirse en los siguientes:

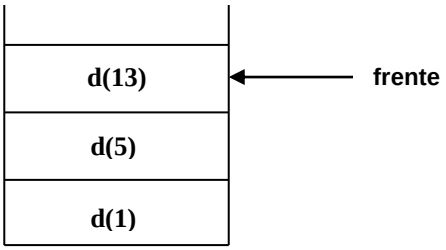
- ✓ En cada nodo del árbol pueden abrirse dos caminos diferentes, y se debe decidir por cuál seguir.
- ✓ Sin embargo, una vez decidido el camino a seguir (a la izquierda o a la derecha), en alguna parte debe guardarse la dirección del nodo que se abandona, para luego poder volver a él y recorrer el otro camino. Por ejemplo, si estando en un nodo ***x*** se decide seguir por el subárbol izquierdo de ***x***, debemos guardar la dirección de ***x*** en algún lugar, pues de otro modo al terminar de recorrer el subárbol izquierdo no podríamos volver a ***x*** para seguir con el derecho. Este problema se repite con cada nodo del árbol, y no sólo con la raiz.
- ✓ Finalmente, queda por determinar en qué momento se detiene el recorrido. En el recorrido de una lista el mismo se detenía al encontrar la dirección *null*, pero en un árbol binario cada rama termina en *null* y por ello encontrar en un árbol una dirección nula no es señal de haber terminado el proceso.

Analizando con cuidado los tres problemas recién planteados, vemos que en realidad es el segundo de ellos el más complicado: ¿dónde (y cómo) guardar las direcciones de los nodos que se abandonan al decidir un recorrido por la izquierda o por la derecha? Como son muchos los nodos en los cuales ese problema se presentará, es obvio que debemos usar una estructura de datos para ir almacenando esas direcciones. En principio cualquiera podría ser útil: una lista, un arreglo, una cola, una pila... Pero la estructura de datos elegida debe facilitar al máximo el proceso de guardado y recuperación de una dirección, para no complicar inútilmente el algoritmo de recorrido del árbol.
Veamos en un gráfico un proceso simplificado de un recorrido de árbol binario (en este caso, el árbol dibujado no es de búsqueda, pero eso no es importante), para tratar de deducir en qué estructura de datos debemos apoyarnos para guardar las direcciones. Supongamos que el proceso de recorrido comienza con el nodo raiz, cuyo valor es 1. Llamemos *d(1)* a la dirección del nodo que contiene al valor 1 (en la práctica, esa dirección será almacenada en un puntero usado para recorrer el árbol). Luego de procesar el nodo *d(1)*, supongamos que se decide seguir el recorrido por la izquierda. Por lo tanto, antes de saltar al nodo *d(5)*, la

dirección *d(1)* debe guardarse en alguna parte. Del mismo modo, al terminar con *d(5)* debe guardarse esa misma dirección para poder saltar a *d(13)*:



Una buena idea es que las direcciones de los nodos "abandonados" se vayan almacenando en una *pila*, pues de esta forma la última dirección guardada en la pila queda encima de la misma, y es la primera que será recuperada, pemitiendo volver al último nodo que se procesó antes de bajar por esa rama. Para nuestro ejemplo, en esa pila se almacenaría primero la direccion *d(1)*, luego la dirección *d(5)* y finalmente *d(13)*, quedando como se muestra:



¿Por qué es útil una pila? Pues bien: al terminar de procesar el nodo *d(13)* ya no es posible seguir "bajando" por el árbol, pues *d(13)* no tiene descendientes. Por lo tanto, se debe volver al nodo padre de *d(13)*, que es *d(5)*, y desde allí seguir el recorrido hacia la derecha del padre. El acceso a la dirección del padre de *d(13)* debería ser inmediato y simple: no deberíamos tener que "buscar" esa dirección, sino que la misma tendría que estar disponible en orden constante [*O(1)*]. Y eso es justamente lo que garantiza una pila al invertir la secuencia de entrada: antes de bajar a *d(13)*, la última dirección guardada en pila será *d(5)*. Por lo tanto, al terminar con *d(13)* nos quedará disponible *d(5)* en forma inmediata.

En este punto, quizás el lector estará comenzando a sentirse incómodo al pensar que deberá programar una pila para guardar direcciones, y luego usar esa pila en un algoritmo que sea capaz de recorrer un árbol binario. Si bien es cierto que semejante algoritmo es una muy buena práctica de programación, veremos que no será necesario tomarse tanto trabajo para gestionar la pila de direcciones, pues conocemos una técnica que puede manejar automáticamente una pila para facilitar la tarea del programador, sin siquiera tener que declarar esa pila: esa técnica es la *recursividad*.

Si un problema puede plantearse de forma *recursiva*, entonces también puede escribirse una función o un método recursivo que lo resuelva. En el caso que nos ocupa, el problema es el *recorrido de un árbol binario*, y existen al menos tres formas clásicas de plantear ese problema en forma recursiva. Los tres métodos son:

- Recorrido en Preorden (u *orden previo*)
- Recorrido en Entreorden (u *orden central*)
- Recorrido en Postorden (u *orden posterior*)

En general, cuando se recorre una estructura de datos cualquiera, el proceso que se aplica a cada nodo o componente de la misma suele designarse como *visitar* el nodo. Con esta aclaración, el recorrido en *Preorden* se define de la siguiente forma recursiva:

Recorrido en Preorden

1.) Visitar el nodo raiz.
2.) Recorrer en ***Preorden*** el subárbol izquierdo.
3.) Recorrer en ***Preorden*** el subárbol derecho.

Puede verse claramente que la definición anterior es *recursiva*: el concepto que se está definiendo (*recorrido en preorden*) vuelve a nombrarse en la propia definición (en los pasos 2 y 3). Si bien parece un poco concisa, la definición dice simplemente que si se quiere recorrer en *preorden* un árbol binario, comience procesando el nodo raiz, y luego aplique el *mismo* método de recorrido para el árbol de la izquierda y para el árbol de la derecha. Esto es: en el árbol de la izquierda, aplique los tres pasos desde el principio: procesar la raiz, recorrer en *preorden* el árbol izquierdo y luego el derecho, y así con cada nodo del árbol.

Veamos cómo sería recorrido el árbol de la *Figura 1* aplicando esta técnica, suponiendo que la visita del nodo consiste en mostrar en pantalla su valor contenido:

- El recorrido comienza con el nodo raiz del árbol, y muestra el valor del mismo: muestra el valor 1.
- Luego, se aplica el paso 2 que indica "*recorrer en preorden el subárbol izquierdo*" del 1. Por lo tanto, el proceso se "traslada" al subárbol cuya raiz es 5, y en ese nodo se aplica un *preorden*. Es decir, se vuelven a aplicar sobre ese nodo los tres pasos de la definición:
  - Visitar raiz: muestra el valor 5.
  - *Preorden en el subárbol izquierdo* del 5: trasladar el proceso al nodo 13, y volver a empezar todo:
    - Visitar raiz: muestra el valor 13.
    - *Preorden en el subárbol izquierdo* del 13: aquí volvería a empezar todo, pero este subárbol está vacío, por lo tanto, termina sin efectuar ninguna operación, y vuelve al nodo 13.
    - *Preorden en el subárbol derecho* del 13: también está vacío, y el proceso termina sin efectuar acción alguna.
  - *Preorden en el subárbol derecho* del 5: trasladar el proceso al nodo 18 y volver a empezar todo:
    - Visitar raiz: muestra el valor 18.
    - *Preorden en el subárbol izquierdo* del 18: aquí volvería a empezar todo, pero este subárbol está vacío, por lo tanto, termina sin efectuar ninguna operación, y vuelve al nodo 18.
    - *Preorden en el subárbol derecho* del 18: también está vacío, y el proceso termina sin efectuar acción alguna.

☞ Ahora estamos de nuevo en el nodo 1, habiendo recorrido y mostrado el contenido de todo el subárbol izquierdo. El mismo proceso se repite ahora, haciendo un *preorden en el subárbol derecho* (dejamos el seguimiento del mismo para el lector).

Ahora bien: un análisis detallado de la secuencia de recorrido en *preorden*, revela que cada nodo del árbol será *tocado* tres veces durante el recorrido: la primera vez, cuando se llega al nodo desde *arriba*, la segunda vez cuando se vuelve *desde el subárbol izquierdo*, y la tercera cuando se vuelve *desde el subárbol derecho*. Pero será procesado (*visitado*) en sólo una de esas ocasiones. Y allí está la diferencia entre los tres tipos de recorrido:

- ✓ En el *preorden*, cada nodo se visita cuando se llega a él por *primera vez*.
- ✓ En el *entreorden*, cada nodo se visita cuando se vuelve a él *desde la izquierda*.
- ✓ En el *postorden*, cada nodo se visita cuando se vuelve a él *desde la derecha*.

Por lo tanto, las definiciones de los recorridos que faltan pueden ser las siguientes:

Recorrido en Entreorden

1.) Recorrer en ***Entreorden*** el subárbol izquierdo.
2.) Visitar el nodo raiz.
3.) Recorrer en ***Entreorden*** el subárbol derecho.

Recorrido en Postorden

1.) Recorrer en ***Postorden*** el subárbol izquierdo.
2.) Recorrer en ***Postorden*** el subárbol derecho.
3.) Visitar el nodo raiz.

Podemos ver que la secuencia que saldría en pantalla con cada uno de los tres métodos, para el árbol de la Figura 1, sería la siguiente (tenga cuidado el lector de hacer un seguimiento estricto de cada algoritmo para asegurarse que logra los resultados que se muestran):

Recorrido en Preorden: { 1, 5, 13, 18, 15, 3, 20 }
Recorrido en Entreorden: { 13, 5, 18, 1, 3, 15, 20 }
Recorrido en Postorden: { 13, 18, 5, 3, 20, 15, 1 }

Finalmente, mostramos a nivel de esquema un método recursivo que recorre un árbol binario y muestra el contenido del mismo en preorden (prescindiendo de detalles de interfaz):

```
private void mostrarPreorden (TreeNode<E> p)
{
  if ( p != null )
  {
    System.out.print ( p.getInfo() );
    mostrarPreorden(p.getIzquierdo());
```

```
      mostrarPreorden(p.getDerecho());
    }
  }
```

El método anterior toma como parámetro una referencia a la raiz del árbol que se quiere recorrer, verifica si esa referencia está apuntando a algún nodo (pues en caso contrario el árbol está vacío y no hay recorrido posible) y en caso afirmativo lanza los tres pasos del recorrido en preorden, tal como los indica la definición. Los dos últimos pasos, son llamadas recursivas al propio método *mostrarPreorden()*. El método se declaró privado, pues debería ser invocado desde algún otro método de la clase del árbol (necesita que se le pase como parámetro la raiz del árbol, que a su vez es privada)

Uno de los temas que más dificultad suele presentar cuando se escribe un algoritmo recursivo es encontrar condición de corte de ese algoritmo. Una buena estrategia es identificar en qué casos el problema que se quiere resolver es trivial, y preguntar por ese caso. Pues bien: ¿en qué caso es trivial el recorrido de un árbol binario? Respuesta: cuando el árbol está vacío, pues en ese caso no hay acción alguna que realizar. Por ello, el métod comienza preguntando si la referencia de entrada es distinta de null y sólo en ese caso procede al recorrido.

En este caso, se supone que la visita que debe realizarse a cada nodo consiste en mostrar su info por pantalla. En el recorrido en preorden, la visita se hace antes de los recorridos de los subárboles. Esos recorridos son realizados por el mismo método *mostrarPreorden()*, que se invoca a sí mismo dos veces luego de la visita al nodo *p*: en la primera invocación toma como parámetro el puntero al hijo izquierdo de *p* (con lo cual recorrerá ese árbol aplicando de nuevo todo el algoritmo), y en la segunda toma como parámetro hijo derecho de *p*, con el mismo efecto.

Introduciendo pequeños cambios se pueden obtener los recorridos en entreorden y postorden, simplemente moviendo la instrucción que hace la visita del nodo. Aquí van los planteos de cada método:

```
private void mostrarEntreorden (NodeTree p)
{
  if ( p != null )
  {
    mostrarEntreorden( p.getIzquierdo() );
    System.out.println( p.getInfo() );
          mostrarEntreorden( p.getDerecho() );
  }
}

private void mostrarPostorden (NodeTree p)
{
  if ( p != null )
  {
    mostrarPostorden( p.getIzquierdo() );
    mostrarPostorden( p.getDerecho() );
   System.out.println ( p.getInfo() );
  }
}
```

Como ya dijimos, la *recursividad* se encarga de gestionar la pila necesaria para almacenar las direcciones de los nodos que se abandonan durante el recorrido. La idea es que al llamar al método, cada vez que se invoca a sí mismo se almacena en el *stack segment* el valor del parámetro *p*. Como el *stack segment* es en los hechos un segmento de memoria que se accede en modo *LIFO*, se comporta como una *pila*. Cada vez que el método termina su ejecución actual, se retiran del *stack segment* los valores colocados por ese método, y quedan disponibles los que se encontraban inmediatamente debajo, que resultan ser los valores correspondientes al nodo padre del que se está procesando. Todos los pasos que se mostraron para explicar el recorrido del árbol de la *Figura 1 (página 15)*, son realizados también ahora pero de forma implícita por la *recursividad*.

Cabe una observación más, por demás interesante: si el árbol binario que se está recorriendo *es de búsqueda* (la función de inserción graba con la regla *menores a la izquierda y mayores a la derecha*), entonces el recorrido en *entreorden* propuesto para el método *mostrarEntreorden()* provocará que la secuencia de valores que se muestra en pantalla salga *ordenada*. Esto es un resultado que no depende de la suerte de quien cargue los datos: siempre que el árbol sea de búsqueda, el recorrido en *entreorden* visitará los nodos en secuencia ordenada. ¿Por qué? Es lógico si se piensa en qué hace el recorrido en *entreorden*: primero procesa todos los nodos del subárbol izquierdo (que son menores que el valor de la raiz), luego procesa la raiz, y finalmente procesa los nodos del subárbol derecho (que son mayores que el valor de la raiz). Como en cada subárbol procede recursivamente igual, entonces cada subárbol es también recorrido *de menor a mayor*. El resultado general es una secuencia ordenada de visitas.

Todos los métodos de recorrido aquí planteados, tienen *precedencia izquierda*: primero se recorre el subárbol izquierdo, y luego se recorre el derecho. Sin embargo, esto no es obligatorio. Los mismos algoritmos pueden plantearse con *precedencia derecha* con mucha facilidad, simplemente invirtiendo el orden de las llamadas recursivas:

```
// Preorden con precedencia derecha
private void mostrarPreorden (NodeTree p)
{
  if ( p != null )
  {
    System.out.print ( p.getInfo() );
    mostrarPreorden(p.getDerecho());
    mostrarPreorden(p.getIzquierdo());
  }
}

// Entreorden con precedencia derecha
private void mostrarEntreorden (NodeTree p)
{
  if ( p != null )
  {
    mostrarEntreorden( p.getDerecho() );
    System.out.println( p.getInfo() );
          mostrarEntreorden( p.getIzquierdo () );
  }
```

```
}

//Postorden con precedencia derecha
private void mostrarPostorden (NodeTree p)
{
  if ( p != null )
  {
     mostrarPostorden( p.getDerecho() );
     mostrarPostorden( p.getIzquierdo () );
    System.out.println ( p.getInfo() );
  }
}
```

Como es obvio, si el árbol es de búsqueda con regla *menores a la izquierda, mayores a la derecha*, entonces si se invierte la precedencia y se recorre el árbol en entreorden el resultado será una secuencia también ordenada, pero ahora de mayor a menor... Por supuesto, el mismo resultado podría lograrse cambiando la regla de inserción, haciendo: *mayores a la izquierda, menores a la derecha* y luego recorriendo el árbol en entreorden con precedencia izquierda.

En nuestra clase *TSBSearchTree*, el método *toString()* retorna una cadena con el contenido del árbol recorrido en entre orden, por lo que los elementos aparecerán en orden de menor a mayor. El recorrido recursivo es realizado por el método privado *make_center_order()*:

```
public String toString()
{
   StringBuffer cad = new StringBuffer("");
   make_center_order(this.root, cad);
   return cad.toString();
}

// crea una cadena con el contenido del árbol en entre orden...
private void make_center_order(TreeNode<E> p, StringBuffer cad)
{
   if (p != null)
   {
      make_center_order (p.getLeft(), cad);
      cad = cad.append(p.getInfo().toString()).append(" ");
      make_center_order (p.getRight(), cad);
   }
}
```

La clase incluye otros dos métodos públicos (no estándar ni emulados de la clase TreeSet) que permiten obtener cadenas en base a recorridos en pre-orden y en post-orden, y también se usan dos métodos privados para hacer los recorridos en forma recursiva. Dejamos su análisis para el estudiante:

```
public String toStringPreOrder()
{
   StringBuffer cad = new StringBuffer("");
   make_pre_order(this.root, cad);
```

```
    return cad.toString();
}

private void make_pre_order(TreeNode<E> p, StringBuffer cad)
{
    if (p != null)
    {
        cad = cad.append(p.getInfo().toString()).append(" ");
        make_pre_order (p.getLeft(), cad);
        make_pre_order (p.getRight(), cad);
    }
}

public String toStringPostOrder()
{
    StringBuffer cad = new StringBuffer("");
    make_post_order(this.root, cad);
    return cad.toString();
}

private void make_post_order(TreeNode<E> p, StringBuffer cad)
{
    if (p != null)
    {
        make_post_order (p.getLeft(), cad);
        make_post_order (p.getRight(), cad);
        cad = cad.append(p.getInfo().toString()).append(" ");
    }
}
```

5.] Borrado en Arboles de Búsqueda.

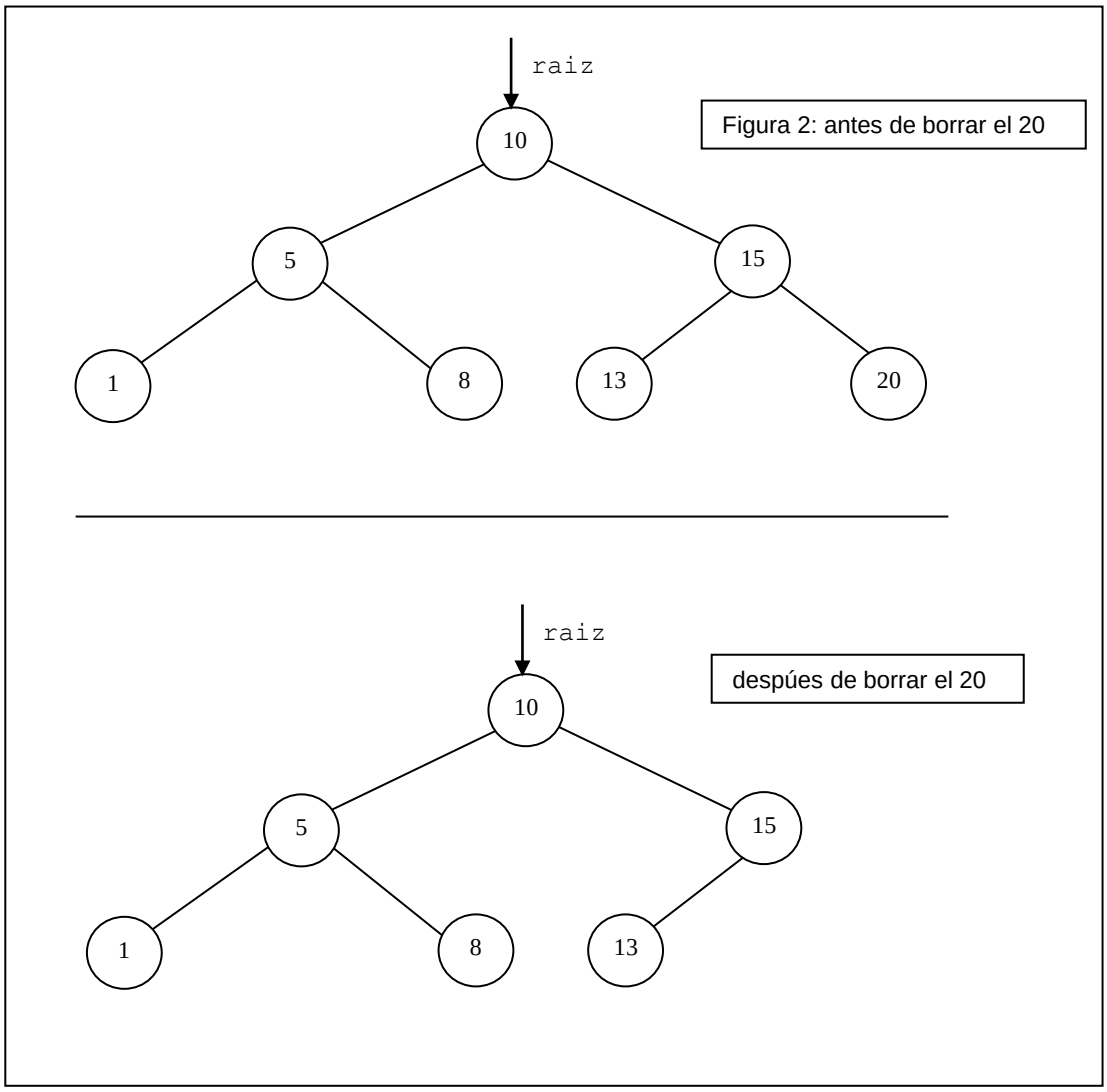
La tarea de gestionar un árbol de búsqueda (o cualquier otra estructura de datos) no estaría completa si no se incluyen métodos para permitir la eliminación de elementos del árbol. Nos proponemos desarrollar un método que tome como parámetro una clave *x*, la busque en el árbol, y en caso de encontrarla elimine al nodo que la contiene, haciendo que el árbol (obviamente) siga siendo de búsqueda. Un rápido análisis del problema sugiere que hay que prestar atención a varios casos, y que no todos ellos son triviales:

a. El nodo a borrar podría ser una hoja (es decir, un nodo sin hijos).

b. El nodo a borrar podría tener un (y sólo un) hijo, sin importar de qué lado.

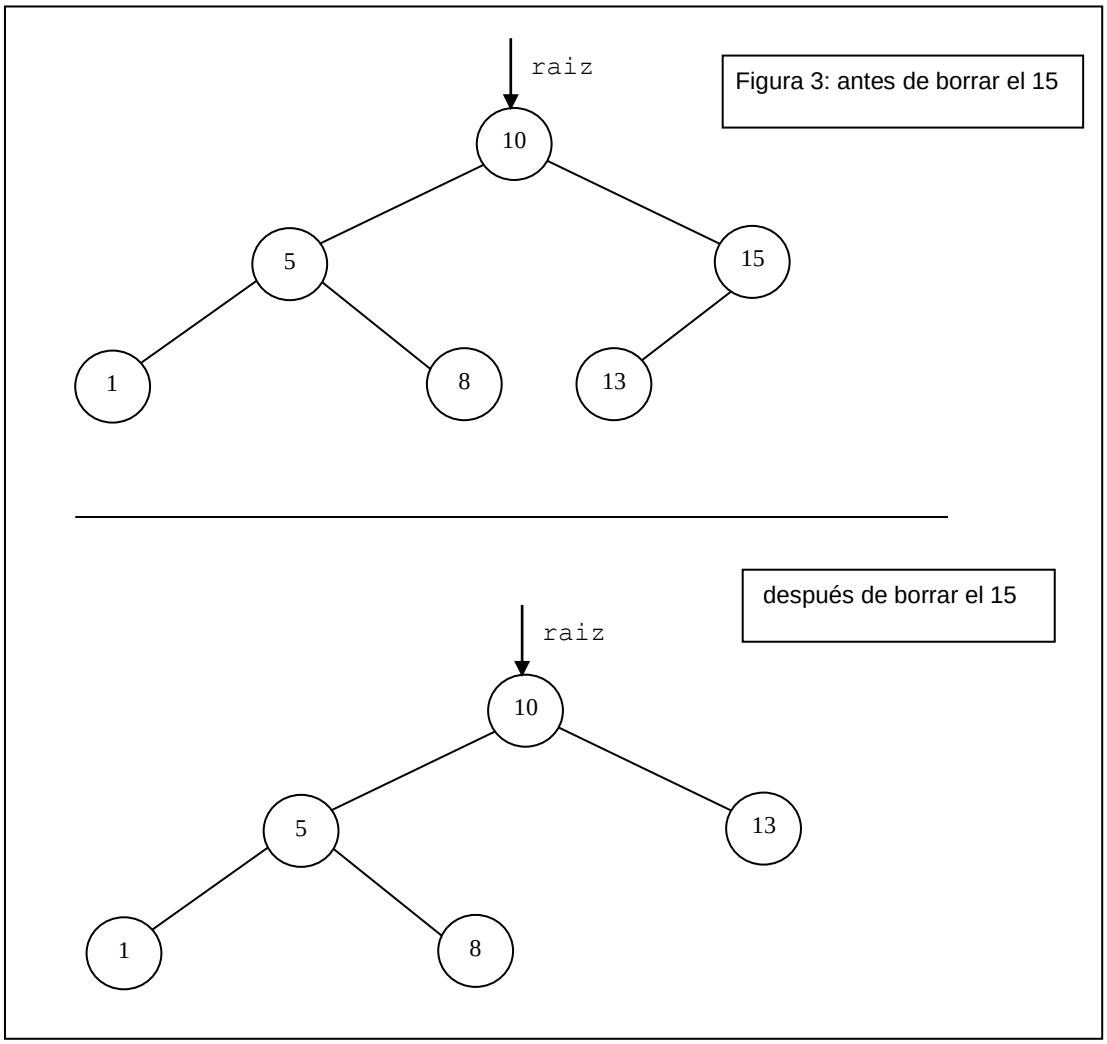
c. El nodo a borrar podría tener exactamente dos hijos.

El primer caso es simple: sólo debemos ubicar el nodo a borrar y poner en *null* el enlace con su padre. Como el nodo borrado no tiene hijos, no debemos preocuparnos por re-enganchar sus sucesores. La *Figura 2* muestra un árbol de búsqueda antes y después de borrar el nodo 20, que es una hoja. El nodo 15 queda con su hijo derecho nulo:

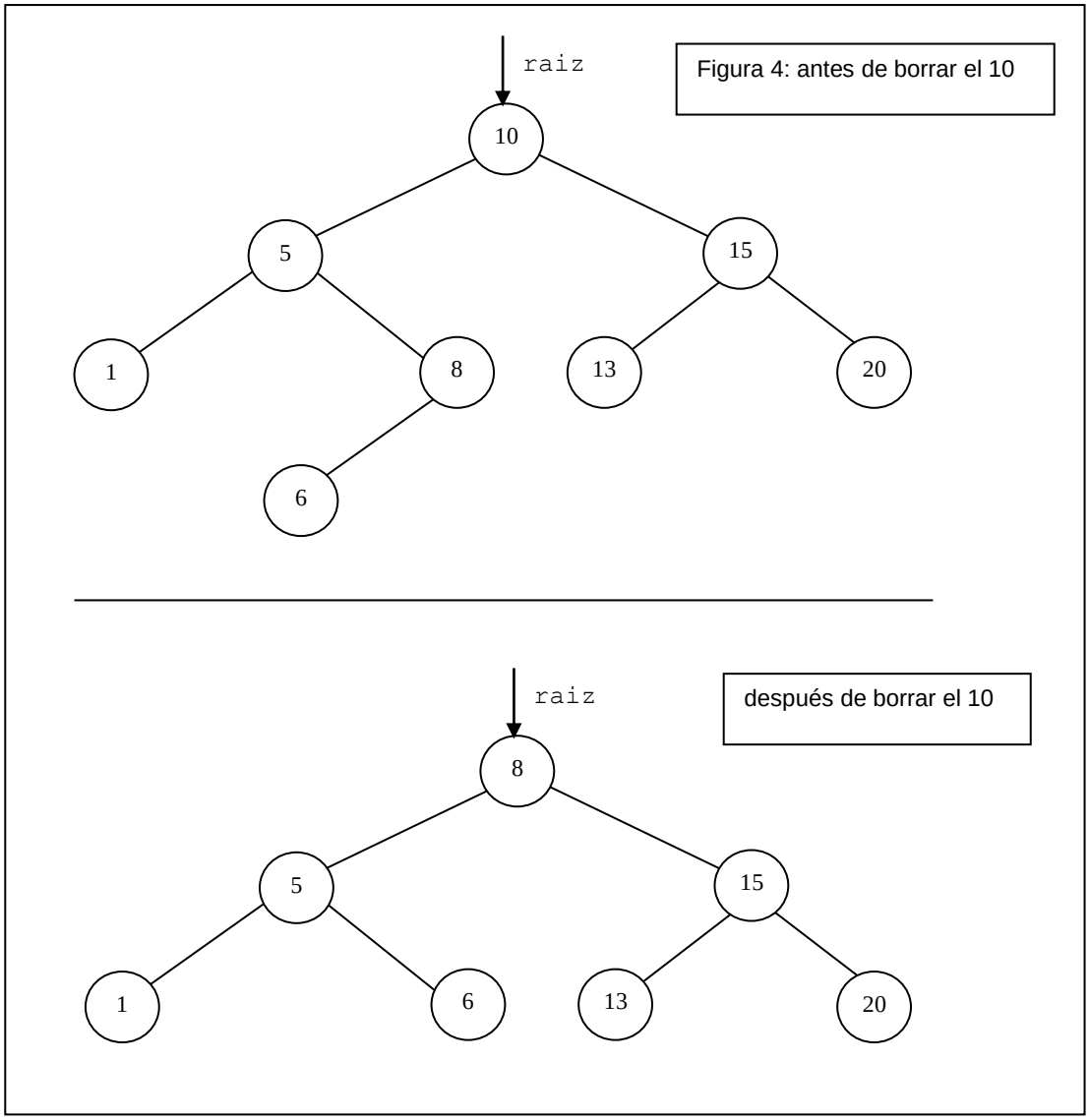


El segundo caso es también simple, pues en los hechos equivale a borrar un nodo en una lista: se ubica el nodo a borrar y se ajusta el enlace del padre de forma que ahora apunte al único hijo del nodo que se quiere borrar. La idea es directa: si borramos un nodo con un único hijo, hacemos que el hijo ocupe el lugar del padre eliminado (ver *Figura 3 en página siguiente*).

El tercer caso ya no es tan simple: el nodo a borrar tiene dos hijos y no es posible que ambos sean apuntados desde el padre del que queremos borrar. En este caso, la idea es reemplazar al nodo que se quiere borrar por otro, tal que este otro sea fácilmente suprimible de su lugar original, pero que oblique a la menor cantidad posible de operaciones en el árbol.



Si observamos bien, ese nodo puede ser el *mayor descendiente izquierdo* del que queremos borrar (o el *menor descendiente derecho*). En el árbol de la *figura 2*, si queremos borrar el 10 podemos reemplazarlo por el 8 (que es el mayor de todos los que son menores que el 10). Eliminar del árbol a ese mayor descendiente izquierdo es simple, pues o bien tendrá un solo hijo a la izquierda, o no tendrá hijos (no puede tener un hijo a la derecha, pues si lo tuviera ese hijo sería mayor que él y no podría entonces ser elegido como el *mayor* descendiente isquierdo). En el caso de la *Figura 2*, el 8 no tiene hijos, y por lo tanto bastaría con poner en *null* el enlace derecho del 5. Si el 8 tuviera un hijo a la izquierda, ese hijo quedaría a la derecha del 5, reemplazando al 8. La figura que sigue ilustra lo dicho:



En el modelo *TSBSearchTree* que acompaña esta ficha, se incluye un método para borrar un nodo del árbol aplicando las ideas vistas. En la clase *TSBSearchTree*, el método *remove()* elimina un objeto *x* del árbol, y usa dos métodos auxiliares, privados. El primero (*delete_node()*) busca recursivamente el nodo que contiene a *x*, y determina si tiene uno o ningún hijo. En cualquiera de los dos casos, procede a eliminarlo. Pero si el nodo tiene dos hijos, invoca a otro método (*delete_with_two_children()*) que es el que finalmente aplica el proceso de reemplazar a ese nodo por su mayor descendiente izquierdo, recursivamente. Se deja su análisis para el alumno.

La clase *TSBSearchTree* del modelo sigue las pautas de implementación esperables en Java para el diseño de una estructura que represente un árbol de búsqueda (en este caso, derivando de *AbstractSet* e implementando *NavigableSet*). La clase incluye un iterador que permite recorrer el árbol en entre orden (y por lo tanto, recuperando sus elementos enforma ordenada). Pero por razones de simplificación, no hemos incluido la implementación de las *vistas stateless* previstas por Java para *TreeSet*. Esas vistas (y sus métodos asociados) pueden ser incluidas por el estudiante, a modo de ejercitación.