Ficha 16

Treaps (Árboles de Búsqueda Aleatorizados)

1.] Introducción: Algoritmos Aleatorizados.

Sabemos que un árbol de búsqueda posibilitará búsquedas muy eficientes, con tiempos de ejecución *O(log(n))* pero siempre y cuando el árbol se mantenga equilibrado, y que la noción o estrategia de equilibrado que se emplee llevará a implementaciones sofisticadas y complejas para los algoritmos de inserción y borrado de claves en el árbol. Un ejemplo conocido es el equilibrio AVL, pero existen otras nociones de equilibrio que también llevan a implementaciones complejas (árboles Rojo-Negro, árboles 2-3-4, etc.)

La razón de la complejidad de implementación de los algoritmos para mantener el equilibrio en los diversos tipos de árboles citados está basada en el hecho de que el árbol crece e incorpora nuevas claves en diversos subárboles, y ese crecimiento puede desbalancear el árbol en distintas formas. Cualquier algoritmo determinista (es decir, un algoritmo que dadas las mismas entradas, producirá siempre las mismas salidas) que tenga por objetivo mantener el equilibrio en el árbol, deberá prever todas las variantes en cuanto a desequilibrios posibles, y la forma de resolverlos (normalmente en base a algún proceso de rotación).

Se han propuesto muchas formas de estructuras de datos que permitan búsquedas de tiempo logarítmico, pero sin tanta complejidad de implementación. Un ejemplo conocido son las Skip Lists (o listas de salto) que esencialmente son listas ordenadas (y no árboles…) en las cuales cada nodo incorpora punteros para acceder a nodos "más lejos" que el inmediato siguiente. Pero la novedad es que en las *Skip Lists* el algoritmo de inserción incorpora la *randomización* (o *aleatorización*) para fijar el nivel en el cual será agregado un nuevo nodo, con el cual el algoritmo pasa a ser un *algoritmo aleatorizado*. En las *Skips Lists*, se confía (y se prueba) que si se trabaja con un número *n* elevado o muy elevado de valores en la lista, las probabilidades actuarán en promedio en forma favorable en cuanto a la estructura de la lista, manteniendo aceptable el porcentaje de nodos en cada nivel.

Desde un punto de vista práctico, lo anterior implica que la implementación del algoritmo de inserción no necesariamente deberá tener en cuenta cada posible caso que pudiera presentarse, sino simplemente calcular en forma aleatoria el nivel del nodo a insertar, y agregarlo en ese nivel haciendo ajustes locales. El algoritmo resultante es notablemente menos complejo que el algoritmo de inserción en un árbol AVL (por ejemplo), y mantiene niveles de eficiencia en tiempo de ejecución del mismo orden para *n* grande o muy grande.

En esta ficha se propone otra estructura de datos para búsquedas de tiempo logarítmico, designada como *Treap*, partiendo de la misma idea de un árbol de búsqueda (insertando los menores a la izquierda y los mayores a la derecha en forma normal) pero agregando elementos de aleatorización en el proceso de inserción, con un objetivo similar al buscado en las *Skip Lists*:

volcar las probabilidades a favor del programador y reducir en forma notable la complejidad de implementación.

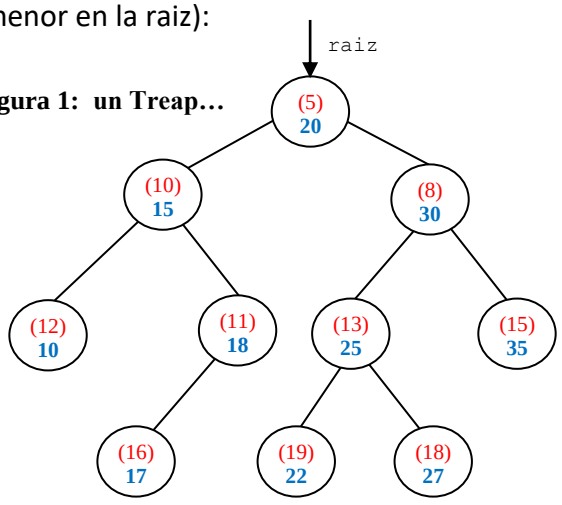
2.] Treaps: Conceptos básicos.

Los *Treaps* (o *Arboles de Búsqueda Aleatorizados*) fueron propuestos por primera vez en 1996 por *Raimund Seidel* y *Cecilia Aragón*[1]. El desarrollo de esta ficha de estudio y el modelo de implementación en Java que la acompaña, está basado en el escrito y modelo planteado en lenguaje C por *Leopoldo Silva Bijit*[2].

Un *Treap* es un árbol binario de búsqueda en el cual cada nodo incorpora un *factor de prioridad* calculado en forma aleatoria, además del dato (o *info*) que almacena en forma normal. La idea esencial es que al insertar un nuevo nodo, se sigue con el valor en el *info* la misma regla de menores a la izquierda y mayores a la derecha de los *árboles de búsqueda* comunes, pero luego se usa el factor de prioridad aleatoria para rebalancear el árbol. En ese rebalanceo, el árbol debe quedar organizado de forma que siga siendo un árbol de búsqueda para el conjunto de sus *info*, pero adicionalmente debe quedar estructurado en forma de *heap ascendente* para el conjunto de sus *factores de prioridad*. De esta forma, se combinan en forma elegante dos estructuras de datos muy conocidas y muy usadas: los árboles de búsqueda (normalmente llamados *Trees* en inglés) y los montículos binarios (conocidos como *Heaps*). La abreviatura y contracción de ambos nombres en inglés, lleva a la designación de *Treap* con la cual se conoce a la nueva estructura.

En la siguiente figura, cada nodo contiene dos valores: el inferior es el ***info*** en base al cual se ordena el árbol de búsqueda (menores a la izquierda, mayores a la derecha) y el superior (escrito entre paréntesis) es el ***factor de prioridad*** mediante el cual el árbol de equilibra y forma un heap ascendente (el menor en la raiz):

Figura 1: un Treap…

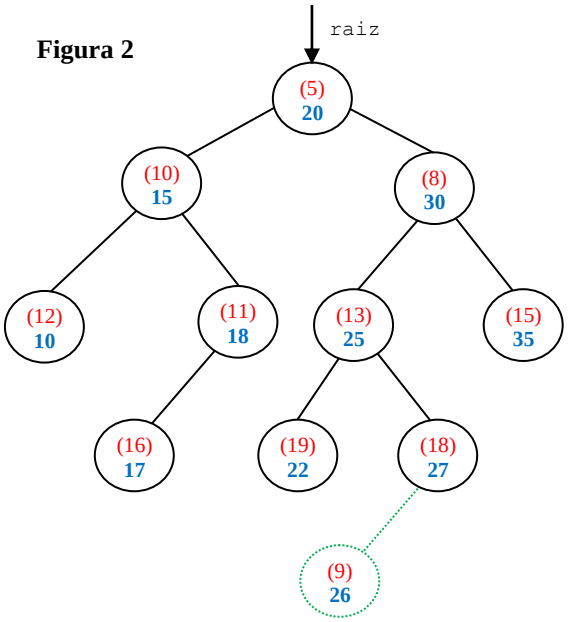


En cada nodo, el *factor de prioridad* se calcula en forma aleatoria (en lugar de calcularse en base a la estructura del árbol, como en el equilibrio AVL por ejemplo). Cuando un nuevo elemento *x* debe insertarse en el árbol, se crea un nodo con *info* igual a *x* y en ese momento se asigna al nodo un *factor de prioridad* aleatorio. En un primer paso, el nodo se inserta en forma normal como en todo árbol de búsqueda, comenzando desde la raiz y descendiendo por las ramas izquierda o derecha según sea *x* menor o mayor al valor de cada nodo. Si suponemos que en el árbol de la Figura 1 se inserta un nodo con *info* igual a 26, y que para ese nodo se calculó un *factor de prioridad* igual a 9,

[1] Seidel, R; Aragón, C. (1996). *Randomized Search Trees*. Algorithmica 16 (4/5): 464–497.
[2] Silva Bijit, L. (2010). *Estructuras de Datos y Algoritmos*. Valparaiso: Universidad Técnica Federico Santa María.

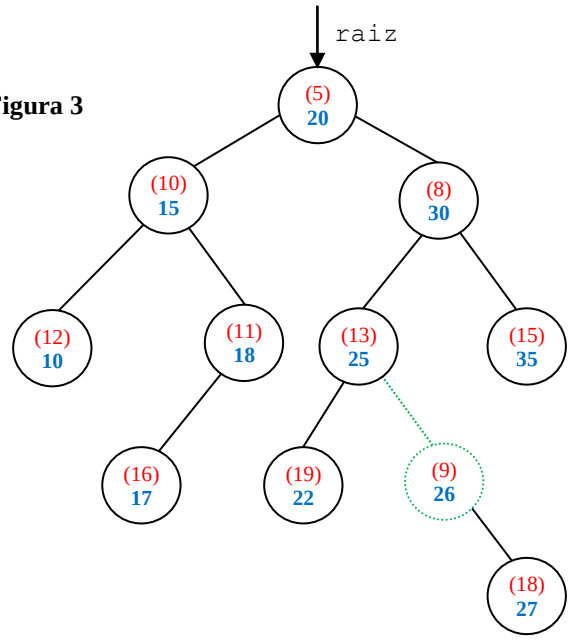
entonces en un primer momento ese nodo quedaría como se ve en la siguiente figura (dibujado en línea de puntos):

Figura 2



Luego de ser insertado el nodo comienza el proceso de eventual rebalanceo, pero ahora considerando el valor del *factor de prioridad* y la estructura de heap ascendente, en la cual cada nodo debería tener un *factor de prioridad* menor que el de sus hijos. En este caso, el nodo que se acaba de insertar tiene un *factor de prioridad* de 9 que es menor que el 18 de su padre. Por lo tanto ambos nodos deben rotarse, pero al hacerlo debe mantenerse la estructura de búsqueda en cuanto a los valores de los *info*. Está claro que en este caso, el nodo 27 debe quedar a la derecha del nodo 26 y este último como hijo del 25:

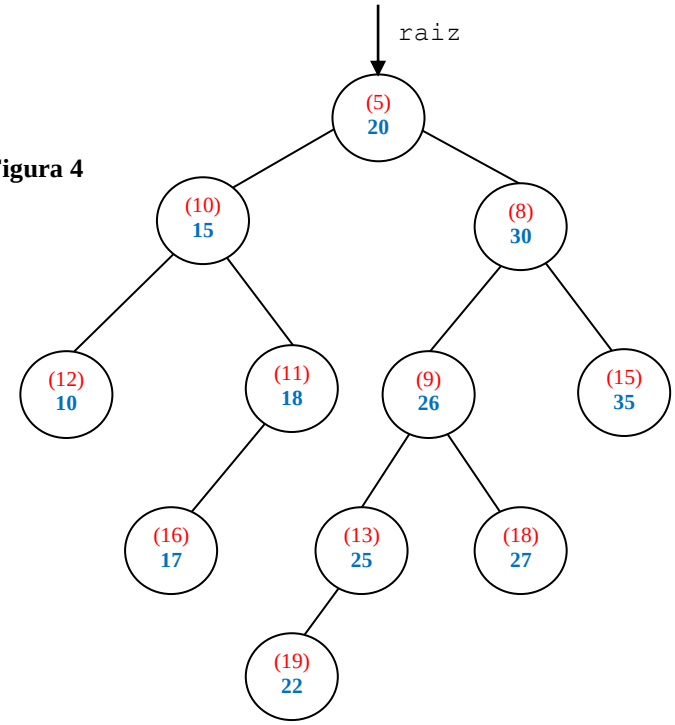
Figura 3



Sin embargo, puede verse que la estructura de heap del árbol todavía es incorrecta, ya que el nodo 26 aún tiene un *factor de prioridad* (9) menor que el de su nuevo padre (13). Por

consiguiente, debe hacerse una nueva rotación esta vez entre el 25 y el 26. El nodo 26 queda como hijo izquierdo del 30, y el 25 queda como hijo izquierdo del 26. El resultado (ahora final) se ve en la figura siguiente:

Figura 4



Puede notarse que en el proceso de rebalanceo, se siguen las mismas reglas básicas que en el proceso de inserción en un heap común, haciendo que el nuevo nodo suba por el árbol siempre por una misma rama, hasta dar con un padre con factor de prioridad menor o llegar a la raiz. Es claro que el árbol así obtenido, además, mantiene dos conjuntos ordenados de valores, aunque con diferentes criterios: el conjunto de los *info* ordenado con regla de "menores a la izquierda y mayores a la derecha", y el conjunto de los *factores de prioridad* con regla de "valor del padre menor al valor de los hijos".

El algoritmo de borrado o remoción de un nodo es esencialmente el proceso inverso de la inserción: Dado el valor *x* que se desea eliminar del árbol, se lo busca en forma normal con el proceso típico de búsqueda en árbol ordenado, comparando contra los valores de los *info* de cada nodo. Pero luego, al encontrar el nodo que contiene a *x*, se procede a "hundir" ese nodo en el árbol hasta que se convierta en una hoja (un nodo sin hijos), con lo cual es simple terminar de removerlo ya que sólo se debe cortar (poner en *null*) el enlace con su padre. Para hundir el nodo, debemos rotarlo con el hijo cuyo *factor de prioridad* sea el menor entre los dos posibles hijos. Por ejemplo, si en el *treap* de la Figura 4 se desea ahora remover el nodo con *info* = 26, debemos rotarlo con el nodo cuyo *info* es 25, ya que ese nodo contiene un *factor de prioridad* de 13 que es menor al 18 del nodo con *info* 27. Al rotar los nodos 26 y 25, el árbol debe seguir siendo de búsqueda, por lo que el 26 quedará a la derecha del 25 (tal como lo veíamos en la Figura 3). Una vez que el nodo 26 quedó como hijo del 25, debemos seguir hundiéndolo hasta dejarlo convertido en una hoja. En este caso, el 26 debe rotar con su único hijo que es el nodo 27 y se logra así el árbol que vimos en la Figura 2. En este momento, el 26 es ya una hoja, y podemos eliminarlo sin problemas, dejando el *treap* tal como lo vimos en la Figura 1.

En cuanto a elementos de eficiencia y análisis de tiempo de ejecución, los propios autores del paper original (*Seidel y Aragón*) probaron que la altura esperada de un *treap* será *O(log(n))*, siendo *n* el número de nodos del árbol, y adicionalmente demostraron también que el número *promedio* de rotaciones que deberán hacerse para terminar de insertar un nodo será menor a 2, lo cual en definitiva es un factor constante. Por lo tanto, los procesos de inserción y remoción que acabamos de describir tendrán un tiempo promedio de ejecución de *O(log(n))*. Con esto se logran cotas de eficiencia del mismo orden que otros árboles equilibrados, pero con el detalle de que la implementación de la estructura de *treap* es notablemente más sencilla.

3.] Consideraciones de implementación en Java.

En el modelo *TSBTreaps* que acompaña a esta ficha, mostramos una implementación en Java de la estructura de *TSBTreap*. Para ello hemos reusado los diseños que ya teníamos para las clases *TSBSearchTree* y *TreeNode* de fichas anteriores, aplicando herencia: al fin y al cabo, un *TSBTreap* es un *TSBSearchTree* en el cual el equilibrio se logra aplicando reglas específicas (cosa que ya ocurría con el *TSBAVLSearchTree*), y un nodo para un *TSBTreap* es un nodo común para un *TSBSearchTree* al cual se requiere agregarle el factor de equilibrio. Con estas ideas, la clase *TreapNode* (interna de la clase *TSBTreap*) que representa un nodo para un *TSBTreap*, puede quedar esquemáticamente así:

```
private class TreapNode <E> extends TSBSearchTree.TreeNode<E>
{
        private float priority = (float) Math.random();  // el valor aleatorio de prioridad...

        public TreapNode(E x, TSBSearchTree.TreeNode<E> iz, TSBSearchTree.TreeNode<E> de)
        {
           super(x, iz, de);
        }

        public float getPriority()
        {
           return priority;
        }

        public void setPriority(float priority)
        {
           this.priority = priority;
        }
}
```

La clase deriva de *TSBSearchTree.TreeNode*, desde la cual adquiere los atributos *info*, *izq* y *der*, y luego simplemente agrega el atributo *priority* para contener el factor de prioridad. Ese valor se calcula en forma aleatoria (como un *float* entre 0 y 1) y se asigna en forma directa al declarar el atributo, con lo cual su valor queda asignado antes de invocar a cualquier constructor, cuya tarea pasa simplemente a ser la de inicializar el resto de los atributos heredados.

La propia clase *TSBTreap* deriva de la clase *TSBSearchTree*, desde la cual hereda el atributo *root* para contener la dirección del primer nodo, el atributo *count* para llevar la cuenta de la cantidad de nodos que tiene el árbol, el atributo modCount para implementar la estrategia *fail-fast iterator* y el atributo *comparator* para contener la dirección del objeto que proveerá el método de comparación para la relación de orden (*compareTo()* o *compare()*). La clase *TSBTreap* declara

algunos atributos adicionales (*son* y *father*), que sólo están allí para ser usados a modo de variables y constantes de uso global en el método *remove()* simplificando así su implementación.

En cuanto a los métodos, la clase *TSBTreap* sólo redefine los métodos *add()* y *remove()* que adquiere desde la clase *TSBSearchTree*, pero hereda y usa sin más el resto de los métodos de la clase *TSBSearchTree*, tales como *contains()*, *size()*, *toString()*, *toStringPreOrder()* y *toStringPostOrder()* (entre otros). Esto tiene sentido ya que como se dijo, un *TSBTreap* sigue siendo un *TSBSearchTree* y el atributo *priority* sólo impacta en los procesos de inserción y borrado: las búsquedas, recorridos, conversiones a *String* y otros procesos se mantienen exactamente igual que en un *TSBSearchTree*. La clase *TSBTreap* puede verse a continuación:

```
public class TSBTreap <E> extends TSBSearchTree<E>
{
   // constantes auxiliares para remove()...
   private enum Direction {ROOT, RIGHT, LEFT};

   // auxiliar global para remove: direccion de descenso...
   private Direction dir;

   // auxiliares globales para remove(): punteros de persecucion...
   private TreapNode<E> son, father;

   public TSBTreap()
   {
   }

   public TSBTreap(Comparator<? super E> comparator)
   {
      super(comparator);
   }

   public TSBTreap(Collection<? extends E> c)
   {
      super(c);
   }

   public TSBTreap(SortedSet<E> s)
   {
      super(s);
   }

   @Override
   public boolean add(E x)
   {
     int ca = this.size();
     this.root = insTreap((TreapNode<E>)this.root, x);
     return (this.size() != ca);
   }

   @Override
   public boolean remove(Object x)
   {
     this.son = (TreapNode<E>) this.root;
     this.dir = Direction.ROOT;
```

```
if(this.son != null)
{
   this.father = null;
   while(son != null && this.compare(x, this.son.getInfo()) != 0)
   {
      this.father = this.son;
      if(this.compare(x, this.son.getInfo()) < 0)
      {
         dir = Direction.LEFT;
         this.son = (TreapNode<E>) this.son.getLeft();
      }
      else
      {
         dir = Direction.RIGHT;
         this.son = (TreapNode<E>) this.son.getRight();
      }
   }

   // si lo encontró, proceder a eliminarlo...
   if(this.son != null)
   {
      // mientras no sea una hoja... hundirlo...
      while(this.son.getLeft() != null || this.son.getRight() != null)
      {
         // si el derecho es null subir el de la izquierda...
         if(this.son.getRight() == null) { left_up(); }
         else
         {
            // si el izquierdo es null subir el de la derecha...
            if(this.son.getLeft() == null) { right_up();}
            else
            {
               // ambos hijos son distintos de null... subir el de menor prioridad...
               if(((TreapNode<E>)this.son.getLeft()).getPriority()<((TreapNode<E>)this.son.getRight()).getPriority())
               {
                  left_up();
               }
               else
               {
                  right_up();
               }
            }
         }
      }

      //this.son ahora es una hoja...
      if(this.father == null) { this.root = null; }
      else
      {
         if(this.compare(x, father.getInfo()) < 0) { father.setLeft(null); }
         else { father.setRight(null); }
      }
      this.count--;
      this.modCount++;
      return true;
```

```
      }
    }
    return false;
}

private TreapNode<E> insTreap(TreapNode<E> p, E x)
{
   if(p == null)
   {
      p = new TreapNode<>(x, null, null);
      this.count++;
      this.modCount++;
   }
   else
   {
      if(this.compare(x, p.getInfo()) < 0)
      {
         p.setLeft(insTreap((TreapNode<E>)p.getLeft(),x));
         float pl = ((TreapNode<E>)p.getLeft()).getPriority();
         if(pl < p.getPriority()) { p = with_left(p); }
      }
      else
      {
         if(this.compare(x, p.getInfo()) > 0)
         {
            p.setRight(insTreap((TreapNode<E>)p.getRight(),x));
            float pr = ((TreapNode<E>)p.getRight()).getPriority();
            if(pr < p.getPriority()) { p = with_right(p); }
         }
      }
   }
   return p;
}

private TreapNode<E> with_left(TreapNode<E> p)
{
   TreapNode<E> aux = p;
   p = (TreapNode<E>) p.getLeft();
   aux.setLeft(p.getRight());
   p.setRight(aux);
   return p;
}

private TreapNode<E> with_right(TreapNode<E> p)
{
   TreapNode<E> aux = p;
   p = (TreapNode<E>) p.getRight();
   aux.setRight(p.getLeft());
   p.setLeft(aux);
   return p;
}

private void right_up()
{
   this.son = with_right(this.son);
```

```
      adjust_father(this.dir, this.son, this.father);
      this.father = this.son;
      this.dir = Direction.LEFT;
      this.son = (TreapNode<E>) this.son.getLeft();
   }

   private void left_up()
   {
      this.son = with_left(this.son);
      adjust_father(this.dir, this.son, this.father);
      this.father = this.son;
      this.dir = Direction.RIGHT;
      this.son = (TreapNode<E>) this.son.getRight();
   }

   private void adjust_father(Direction d, TreapNode<E> p, TreapNode<E> q)
   {
      if(d == Direction.RIGHT && q != null) { q.setRight(p); }
      else
      {
         if(d == Direction.LEFT && q != null) { q.setLeft(p); }
         else
         {
            if(d == Direction.ROOT) { this.root = p; }
         }
      }
   }

   private class TreapNode <E> extends TSBSearchTree.TreeNode<E>
   {
      private float priority = (float) Math.random();  // el valor aleatorio de prioridad...

      public TreapNode(E x, TSBSearchTree.TreeNode<E> iz, TSBSearchTree.TreeNode<E> de)
      {
         super(x, iz, de);
      }

      public float getPriority()
      {
         return priority;
      }

      public void setPriority(float priority)
      {
         this.priority = priority;
      }
   }
}
```

Observe que el método *add()* está implementado en forma recursiva, ya que eso facilita el retorno *hacia arriba* en el árbol para ir reacomodando los nodos a medida que rotan con su padre. El método *remove()*, por el contrario, está implementado en forma no recursiva (y con ello es algo más complejo de lo que podría esperarse…) ya que el proceso de borrado requiere *descender* en el árbol, recordando en cada momento la dirección del padre del nodo que va bajando y la dirección

por la cual se bajó. Los métodos privados *with_rigth()* y *with_left()* implementan las rotaciones padre-hijo requeridas tanto por *add()* como por *remove()*. Dejamos el análisis general del código fuente como tarea para el alumno, pero insistimos en hacer notar la relativa sencillez de la implementación completa (comparada con, por ejemplo, la implementación de un árbol AVL).