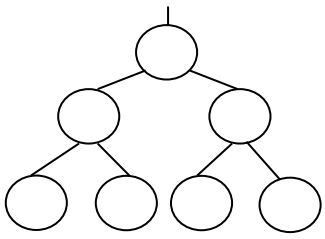
Ficha 00 (Repaso)

Heaps (Montículos Binarios)

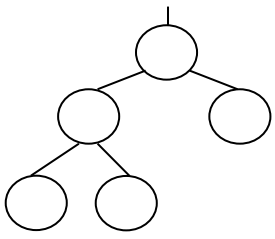
1.] Introducción.

Existen muchos problemas en los cuales se requiere poder obtener en forma rápida el valor menor (o el mayor) de un conjunto de *n* datos. Obviamente, se puede aplicar un recorrido de todos los datos y por comparación exhaustiva obtener el menor o el mayor, pero eso llevaría un tiempo de ejecución lineal [*O(n)*] en el peor caso. Los *Heaps* o *Montículos Binarios* (también llamados *Grupos de Ordenamiento* en algunos textos) permiten organizar el conjunto de *n* datos de forma de poder insertar nuevos valores y/o obtener el menor o el mayor en tiempo logarítmico [*O(log(n))*].

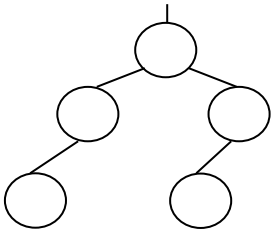
La implementación de un *Heap* requiere el uso de *árboles binarios completos* o *casi completos*. *Un árbol binario completo de altura h* es aquel en el que todo nodo hasta el nivel *h – 1* tiene dos hijos no nulos, y en consecuencia el nivel *h* está "completo". Y un *árbol binario casi completo de altura h* puede entenderse como un completo de altura *h* al cual se le van quitando hojas siempre del nivel *h*, de derecha a izquierda, sin saltear ninguna hoja. Las siguientes figuras ilustran ambos conceptos, además de mostrar un ejemplo de un árbol que no cumple con ninguna de ambas definiciones:



Un árbol binario completo de altura h = 3

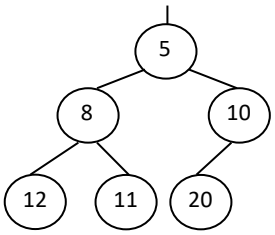


Un árbol binario casi completo de altura h = 3

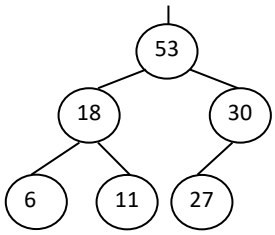


Un árbol binario (altura h = 3) que no es completo ni casi completo

Intuitivamente un *Heap* para facilitar la obtención *del menor* de un conjunto de *n* datos, puede verse como un árbol binario completo o casi completo, en el cual además el valor contenido en cada nodo es siempre menor que los valores contenidos en sus hijos y los descendientes de sus hijos. De esta forma, resulta obvio que el valor contenido en la raíz del árbol será siempre el valor menor de los *n* valores contenidos en el árbol, y suele designarse como un *Heap Ascendente*. En forma similar, un *Heap Descendente* se organiza de forma que la raíz contenga *al mayor* del conjunto, y por lo tanto, cada nodo contiene un valor siempre mayor que el de sus hijos y los descendientes de sus hijos:



Un Heap Ascendente…



Un Heap Descendente…

2.] Implementación de Heaps.

Si un Heap puede entenderse como un árbol binario completo o casi completo, entonces puede representarse e implementarse con relativa sencillez usando un arreglo, en base a las siguientes reglas directas:

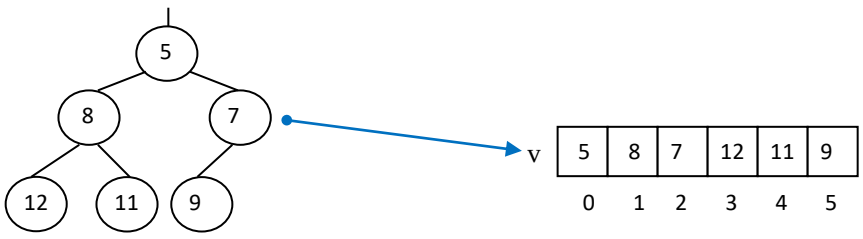
i. Cree un arreglo *v* cuyo tamaño *n* coincida con la cantidad de elementos del heap que quiere representar.

ii. Asigne el elemento raíz del heap, en v[0].

iii. Dado el elemento del heap almacenado en v[i], entonces su hijo izquierdo debe almacenarse v[2i + 1] y su hijo derecho en v[2i + 2].

iv. Dado el elemento del heap almacenado en v[i], con i != 0, entonces su padre estará almacenado en el casillero v[(i - 1) / 2].

La siguiente figura ilustra lo dicho (tómese un momento para analizar la gráfica y asegurarse que entiende la forma en que se ha procedido para armar el arreglo a partir del árbol):



Un Heap Ascendente… … mapeado en un Arreglo.

La idea de usar un arreglo tiene ventajas y desventajas. Entre las ventajas, podemos ver que la implementación resulta sencilla y que por añadidura, se puede navegar tanto hacia arriba como hacia abajo en el árbol simplemente haciendo multiplicaciones y divisiones de índices. La navegación en ambos sentidos es esencial: cuando se inserte un nuevo elemento deberemos ser capaces de ir hacia arriba (hacia la raíz) y al eliminar un elemento deberemos poder descender en el árbol desde la raíz hacia las hojas para volver a armar el *Heap*.

Sin embargo, también es obvia la desventaja principal: para usar un arreglo, se debe estimar el tamaño a priori, antes de tener todos los datos que luego serán insertados en el *Heap*. El tema es que esto no es siempre posible, y por ende el arreglo puede quedar con cierta cantidad de espacio sin usar. Además, si en el momento de hacer una inserción se diera el caso de no tener ya espacio libre en el arreglo, se deberá reajustar la situación creando un

nuevo arreglo y copiando los elementos del "viejo" vector hacia el "nuevo", lo cual agrega algo de pérdida de tiempo (aunque esto se minimiza por el uso del método *System.arraycopy()* nativo de Java, que permite realizar esa copia con mucha eficiencia).

Es cierto que podríamos pensar en que en lugar de un arreglo común, se use algún objeto de la clase *java.util.Vector* o de la clase *java.util.ArrayList* para representar al árbol y de esta forma expandir o retraer el tamaño del vector a medida que se requiere. Pero estas estructuras (*Vector* y *ArrayList*) están implementadas a su vez como listas, lo cual implica que una inserción o una eliminación no será tan eficiente en tiempo de ejecución como hacerlo en un arreglo.

En el modelo *TSBHeaps* que acompaña a esta ficha de estudio se provee una clase *TSBHeap* que implementa estas ideas usando (como de costumbre) el mecanismo generics para realizar el control homogeneidad en tiempo de compilación.

3.] Detalles de implementación y operaciones básicas.

La clase *TSBHeap* del modelo citado declara como atributo y luego crea un arreglo *heap* de elementos de tipo *Slot*:

```
public class TSBHeap<E extends Comparable>
{
    private Slot<E> heap[];          // el arreglo para representar al árbol.
    int initial_capacity;            // la capacidad inicial del arreglo.
    private int count;               // la cantidad de elementos que tiene el Heap.
    private boolean ascendent;  // true: ascendente  -  false: descendente.
```

La clase *Slot* es una clase interna y privada dentro de la clase *TSBHeap*. Cada objeto *Slot* representa una de las entradas o casilleros del arreglo *heap* que se usa para implementar el montículo:

```
    private class Slot <E extends Comparable>
    {
       private E data;

       public Slot(E d)
       {
          data = d;
       }

       public E getData()
       {
          return data;
       }

       public void setData(E d)
       {
          if(d != null) { data = d; }
       }
       public String toString()
       {
```

```
            return data.toString();
        }
    }
```

El atributo *initial_capacity* se usa para almacenar el tamaño con el que fue originalmente creado el arreglo, lo cual será oportunamente útil cuando se quiera reducir el tamaño del arreglo (en los métodos *remove()* y *clear()*) El atributo *count* se usa para llevar la cuenta de la cantidad de elementos que efectivamente contiene el heap, y también para indicar el índice del primer casillero desocupado en el arreglo (comenzando de izquierda a derecha). El atributo booleano *ascendent* se usa para saber si el heap que se está usando es ascendente (valor *true*) o descendente (valor *false*). Recordemos que, si el heap es ascendente, entonces la raíz del árbol contendrá siempre al menor valor del conjunto, y si es descendente la raíz contendrá al mayor.

La clase tiene varios *constructores*, pero finalmente todos ellos terminan invocando al que se muestra a continuación:

```
public TSBHeap(int n, boolean t)
{
   int th = n;
   if(th <= 0) { th = 100; }
   heap = new Slot[th];
   initial_capacity = th;
   count = 0;
   ascendent = t;
}
```

Este constructor crea un heap vacío, con capacidad inicial para *n* componentes de tipo *Slot* (siendo *n* el primero de sus parámetros). La clase *Slot* es necesaria justamente para poder crear el arreglo: si se usa *new* para crear directamente un arreglo de elementos de tipo *E*, el compilador lanzará un error al no poder determinar exactamente qué tipo en concreto representa *E* (y ese tipo es necesario al crear un arreglo para que el compilador sepa exactamente el tamaño en bytes de cada casilla). Para evitar este problema, se hace que el arreglo contenga referencias a los objetos de la clase *Slot* (que actúa como un envoltorio para los objetos del tipo *E*): todas las referencias son punteros del mismo tamaño en bytes, y el compilador no necesita saber exactamente cuál es el tipo o clase *E* de los objetos apuntados.

Si valor inicial de *n* fuera inválido (cero o negativo) entonces el heap se crea con capacidad inicial igual a 100 (que de ahora en adelante se toma como valor por default). El tipo de heap a crear depende del parámetro *t*: si *t = true*, el heap será ascendente (orientado a la obtención rápida del menor), y si *t = false* entonces será descendente (orientado a la obtención rápida del mayor).

Las operaciones elementales para realizar con un heap son dos. La primera de ellas es la *inserción* de un nuevo elemento en el heap, lo cual se realiza con el método *add()*:

```
public void add(E data)
{
```

```
    int n = heap.length;
    if(count == n) { adjustCapacity(2*n); }

    int s = count;
    heap[s] = new Slot<>(data);
    while(s != 0)
    {
       int f = (s - 1) / 2;

       if( ! valid_change(s, f) ) { break; }

       Slot <E> aux = heap[s];
       heap[s] = heap[f];
       heap[f] = aux;
       s = f;
    }

    count++;
}

 private void adjustCapacity( int nc )
 {
    int n = heap.length;

    // cuantos elementos debo trasladar?
    int cant = n;
    if(nc < n) { cant = count; }

    // hacer el traslado...
    Slot <E> newheap[] = new Slot[nc];
    System.arraycopy(heap, 0, newheap, 0, cant);
    heap = newheap;
 }
```

El método controla que el espacio que aún queda disponible en el arreglo *heap* alcance para agregar un nuevo elemento: si *count* es igual al tamaño del arreglo, entonces el arreglo está lleno, y se invoca a otro método privado (*adjustCapacity()*) para crear un nuevo arreglo, copiar los elementos del viejo arreglo al nuevo, y cambiar las referencias. La capacidad del nuevo arreglo se ajusta al doble de la capacidad anterior (*2*n*). Puede ver el código fuente del método *adjustCapacity()* a continuación del método *add()*.

El método *add()* finalmente agrega el nuevo elemento *data* al final del vector, en la primera casilla libre comenzando desde la izquierda (la casilla cuyo índice coincide con *count*). Desde allí, se usa un ciclo para "subir" en el árbol, comparando a *data* con el contenido de su nodo padre. Cada vez que se detecte que el nodo padre contiene un objeto menor que *data* (en un heap ascendente) o mayor que *data* (en un heap descendente) se intercambian los objetos padre – hijo para restaurar la estructura del heap en ese nivel y se continúa el ciclo con el padre en el siguiente nivel hacia arriba. Así, en el peor caso, el objeto *data* subirá hasta la raíz y en ese momento el ciclo cortará. Si al comparar a *data* con su padre actual, se

detecta que el orden es correcto, se corta el ciclo con un *break*, sin tener que llegar al nodo raíz en este caso. Al salir del ciclo, el objeto *data* está ya en su lugar correcto, y se incrementa en uno el valor de *count*.

Note el uso del método privado *valid_change(s, f)*: este método se usa para saber si se deben intercambiar el objeto hijo (que viene subiendo por el árbol y está en la casilla *heap[s]*, siendo *s* el primer parámetro del método) con el objeto padre (que es el que está en la casilla *heap[f]*, siendo *f* el segundo parámetro del método). El método retorna *true* si el intercambio debe hacerse, y *false* en caso contrario. Pero lo interesante es que el método hace la comparación teniendo en cuenta si el heap es ascendente o descendente, consultando el valor del atributo *ascendent* de la clase. La lógica es simple, y puede deducirse directamente del código fuente y los comentarios incluidos en el método:

```
private boolean valid_change(int s, int f)
{
   // comparar el hijo (heap[s]) con el padre (heap[f])...
   int r = heap[s].getData().compareTo(heap[f].getData());

   // ...si el menor debiera estar arriba, pero esta abajo, retorne true...
   if(ascendent && r < 0) { return true; }

   // ...si el mayor debiera estar arriba, pero esta abajo, retorne true...
   if( ! ascendent && r > 0) { return true; }

   // en cualquier otro caso, esta todo en orden... y no hay que hacer cambios...
   return false;
}
```

Podemos ver que el método *add()* recorre el árbol hacia arriba, comenzando desde la hoja que contiene al objeto que se acaba de agregar. Como el árbol es completo o casi completo, entonces los *n* nodos están repartidos de forma de llenar progresivamente (de izquierda a derecha) una cantidad de *log(n)* niveles (o sea, la altura del árbol es *h = log(n)*). Como al subir por el árbol el nuevo elemento se compara solo una vez en cada nivel, se deduce que el tiempo de ejecución de una inserción en el peor caso es *O(log(n))*.

La otra operación elemental para realizar con un heap es la extracción del menor (o mayor) elemento (designado como el *óptimo* de aquí en más), que siempre estará en la raíz. Puede pensarse que entonces la extracción es de orden constante (*O(1)*), pero no es así: el detalle es que luego de extraer el valor de la raíz debe reorganizarse el heap para volver colocar en la raíz al óptimo de entre los elementos que quedan.

La idea es que, una vez extraído el valor de la raíz, se tome el objeto ubicado en el último casillero ocupado del arreglo (el valor en *heap[count - 1]*), se lo ubique como nueva raíz temporal, y desde allí se obtenga el óptimo entre sus dos hijos para que se coloque como nuevo valor raíz. Esto traslada el problema al nodo hijo que contenía al óptimo: en ese nivel se repite la idea, tomando el óptimo entre los dos nuevos hijos, y comparando contra el valor que está "bajando" en el árbol, hasta llegar al último nivel o hasta dar con un hijo que ya esté ordenado con relación al que está bajando. Igual que antes, esta operación completa

requiere una comparación por cada nivel del árbol en el peor caso, con lo cual se tiene se tiene también un tiempo de ejecución *O(log(n))*.

El método *remove()* provisto en la clase *TSBHeap* implementa estas ideas. Note que este método también valida la cantidad de espacio disponible, pero a la inversa que en la inserción y con algún control adicional: la reducción de espacio sólo se hará si la mitad de la capacidad actual es mayor o igual que la capacidad inicial del arreglo (*initial_capacity*) y la cantidad actual *count* de elementos es mayor a *initial_capacity*, para evitar que el arreglo recién creado sea inmediatamente reducido si se invoca a *remove()* en momentos cercanos a la creación del *Heap*. Si todo eso se cumple y si al eliminar el objeto raíz se detecta que el espacio ocupado en el vector es menor que el 90% de la mitad del arreglo, entonces se ajusta el tamaño del arreglo invocando nuevamente al método privado *adjustCapacity()*, pero ahora para *contraer* el arreglo a la mitad de su tamaño actual.

Cuando se comparan los objetos contenidos en los hijos del nodo que está bajando por el árbol, se usa el método privado *optimal_left(sl, sr)*. Este método retorna *true* el objeto ubicado en la casilla *heap[sl]* es el óptimo comparado contra el objeto en *heap[sr]*. El método supone que *sl* es el índice del hijo izquierdo del objeto que está bajando, y *sr* es el índice del hijo derecho. Si el heap es ascendente, el método retornará *true* si *heap[sl] < heap[sr]*. Si el heap es descendente, el método retornará *true* cuando *heap[sl] > heap[sr]*. El análisis de los detalles de implementación del método *remove()* y del método *optimal_left()* se dejan para el lector, así como el resto de los métodos de la clase *TSBHeap*, que son triviales:

```
public E remove()
{
   if(isEmpty()) { return null; }

   E x = heap[0].getData();

   heap[0] = heap[count - 1];
   count--;

   // si sobra demasiado lugar, achicar el vector...
   int n = heap.length;
   int ic = initial_capacity;
   if(n/2 >= ic && count > ic && count < (n/2) * 0.9) { adjustCapacity(n/2); }

   int f = 0;
   while(f < count)
   {
      // calcular direcciones del hijo izquierdo (sl) y el derecho (sr)...
      int sl = f * 2 + 1;
      int sr = sl + 1;

      // si no hay hijos, cortar y terminar...
      if(sl >= count) { break; }

      // ...guardar en s el indice del hijo por el que se debe bajar...
```

```
        int s = sl;
        if(sr < count && !optimal_left(sl, sr) ) { s = sr; }

        // si ese hijo no debe cambiar de lugar con el padre, cortar y terminar...
        if( ! valid_change(s, f) ) { break; }

        // ... caso contrario, intercambiar padre (heap[f]) con el hijo correcto (heap[s])...
          Slot <E> aux = heap[f];
          heap[f] = heap[s];
          heap[s] = aux;

        // ...seguir desde el hijo...
        f = s;
      }

      // ... y devolver el elemento que estaba originalmente en la cima...
      return x;
    }

    private boolean optimal_left(int sl, int sr)
    {
      int r = heap[sl].getData().compareTo(heap[sr].getData());

      // si el heap es ascendente y el izquierdo es menor que el derecho, salir con true...
      if( ascendent && r < 0 ) { return true; }

      // si el heap es descendente y el izquierdo es mayor que el derecho, salir con true...
      if( ! ascendent && r > 0) { return true; }

      // el optimo no es el izquierdo...
      return false;
    }
```