APUNTES AED

1. INTRODUCCIÓN

> Puede ocurrir **shadowing**: Si hay dos variables que se llaman igual, una puede ocultar a la otra

> Hay que inicializar el tamaño de un array, si no se hace **NullPointerException**

> Si accedemos a una posición negativa o mayor que el tamaño del array → **ArrayIndexOutOfBoundsException**

```
> int [] a;                    int [] a2 = null;
  a[0] = 4;                    a[2] = 10;

  int [] b = new int[2];       String [] d = new String[10];
  b[2] = 7;                    d[0] = new String("Hola");

  int [] f = { };
  int [] g = { 7, 9, 8 };
```

> Cuando se declara una variable de tipo clase o interfaz se **inicializa a null**. Si se intenta usar ese objeto → NullPointerException

> La **identidad de un objeto** es la dirección de memoria en la que está el objeto

> El **estado de un objeto** son los valores que tienen los atributos de un objeto

> Las **comparaciones** son diferentes. Para comparar la identidad comparamos con == . Para comparar el estado utilizamos equals.

> Si hago *Vehiculo v2 = v1*, hago que v2 **apunte** a v1, pero no se crea uno nuevo

> Hay unas **clases envoltorio** que se comportan como clases y como tipos básicos: Integer, Double, Float… Siempre comparar con equals

> public class Hijo **extends** Padre (con nuevos atributos y métodos o sobreescribiendo)

> **Upcasting**: *Vehiculo v = new Automovil ()* --> La variable v es de tipo vehículo y el objeto se ha creado de tipo Automovil (v es variable polimorfica)

> **Downcasting**: debe hacerse de forma explícita --> *Automovil a2 = (Automovil) v;*

> hijo **instanceof** Padre --> Para saber si un objeto es de esa clase (o de la de su padre), devuelve booleano

> **Sobreescritura:** cuando el hijo hereda un método y lo "adaptas". Si una clase no sobrescribe un método y es invocado, se subirá la jerarquía hasta la primera clase que lo implemente.

> **Sobrecarga:** permite nombrar con el mismo identificador diferentes métodos o variables. La elección de método la marca el tipo de variable usada para invocarlo, han de ser !=

> **Interfaz:** conjunto de cabeceras de métodos, no tiene código ni constructores. Una clase que implementa una interfaz ha de implementar TODOS sus métodos.
Public class C implements I {

> Un interfaz puede **extender** a otros interfaces

> Una misma clase puede **implementar muchos interfaces**, pero solo puede extender una clase. El tipo con el que se declara una variable indica qué métodos podrán ser invocados

> Las **variables de tipo genéricas** se usan para generalizar el tipo de los elementos de los TADs contenedores

> Java permite **métodos genéricos**, en vez del static. EJ: *public <E> E identidad(E elem) {*

> No pueden crearse **arrays de tipo genérico**, pero sí declararse.

2. TADS

> Conjunto de **valores y de operaciones** definidos mediante una especificación independiente de cualquier representación

> En Java un TAD consiste en una **jerarquía de clases e interfaces**. Los interfaces especifican el *qué*. Las clases implementan el *cómo*.

> **Abstracción Funcional**: Atender a la función ignorando la estructura/representación interna. La representación interna se detalla en la/s implementación.

> **Encapsulación** (ocultación de información): sólo es necesario conocer la función, la estructura puede quedar oculta. Debe ser autocontenido y no depender de cambios en otros módulos

3. TADS LIFO Y FIFO

- **INDEXEDLIST**

> Es una colección de elementos en un cierto **orden**

> El orden no depende de los elementos en sí, sino de la **posición** que ocupan en la lista

> A nivel teórico puede tener **infinitos elementos**.

> Decimos que una lista está indexada cuando se usa un **índice** para acceder a los elementos

> La clase ArrayIndexedList<E> implementa el interfaz IndexedList<E>, usa internamente un array para el almacenamiento de los elementos

> public interface IndexedList <E> extends Iterable <E> {

> public int size ();

> public void add (int index, E e) throws IndexOutOfBoundsException;

> public E get (int index) throws IndexOutOfBoundsException;

> public boolean isEmpty ();

> public E set (int index, E e) throws IndexOutOfBoundsException;

> public int indexOf (E search);

> public E removeElementAt (int i) throws IndexOutOfBoundsException;

> public boolean remove (E element);

> public Object [] toArray ();

- **PILA LIFO**

> Es una secuencia lineal de elementos donde: El último elemento apilado (push) es el primer elemento en ser desapilado (pop).

> No existe operación de búsqueda sobre los elementos. Únicamente se puede consultar la **cima de la pila**

> No tiene límite de tamaño.

> La clase LIFOList<E> implementa el interfaz LIFO<E>, utiliza internamente una lista para el almacenamiento de los elementos.

> La clase LIFOArray<E> también implementa el interfaz LIFO<E>, utiliza internamente un array para el almacenamiento de los elementos.

> public interface LIFO <E> extends Iterable <E> {

> public int size ();

> public boolean isEmpty ();

> public E top () throws EmptyStackException;

> public E pop ();

> public void push (E elem);

> public Object [] toArray();

- **COLA FIFO**

> Es una secuencia lineal de elementos donde el primer elemento encolado (enqueue) es el primer elemento en ser desencolado (dequeue)

> No existe una operación de búsqueda sobre los elementos. únicamente se puede consultar el **primer elemento de la cola**

> No tiene límite de tamaño

> La clase FIFOList<E> implementa el interfaz FIFO<E>, utiliza internamente una lista para el almacenamiento de los elementos

> La clase FIFOArray<E> también implementa el interfaz FIFO<E>, utiliza internamente un array para el almacenamiento de los elementos

> public interface FIFO <E> extends Iterable <E> {

> public int size ();

> public boolean isEmpty ();

> public E first () throws EmptyFIFOException;

> public void enqueue (E elem);

> public E dequeue () throws EmptyFIFOException;

> public Object [] toArray ();

4. COMPLEJIDAD

> La complejidad de un programa es una medida de su **tiempo de ejecución** o de su **uso de memoria.**

> Hablamos de 2 tipos de complejidad:

- Complejidad temporal: tiempo de ejecución
- Complejidad espacial: espacio de memoria utilizado

> La complejidad en tiempo y en espacio suelen estar reñidas. Se ahorra tiempo a costa de usar más espacio y viceversa.

> La complejidad de un programa se calcula **en función de los datos de entrada**. Al crecer el tamaño de la entrada, la complejidad puede crecer o permanecer constante.

> EJ: un metodo busca un elem que está en un array. El “tamaño de la entrada” es el tamaño del array en el que se hace la búsqueda. El tamaño del elemento a buscar no suele ser relevante.

> La complejidad de un método es su complejidad * la de los métodos a los que llame.

> No sólo el tamaño de los datos es relevante, también la distribución de los datos puede ayudar (o perjudicar)

> Podemos encontrarnos diferentes escenarios:

> El elemento a buscar es siempre el primero en ser accedido
> El elemento a buscar puede estar en cualquier sitio (todos los casos son equiprobables)
> El elemento a buscar está siempre el último o no está
> El elemento a buscar estás más veces el primero que en otra posición

> Podemos estudiar la complejidad:

> En el caso mejor (lower-bound)
> En el caso (pro-)medio
> En el caso peor (upper-bound) → da resultados más precisos y fiables
> En el caso amortizado

> La complejidad asintótica implica calcular la complejidad cuando el tamaño de los datos de entrada tiende a infinito.

> Los interfaces no implican ninguna medida de complejidad. La complejidad va asociada a la implementación de los métodos del TAD

- Función constante f (n) = c
- Función polinomial f (n) = c1 ∗ne1 + ... + cm ∗neml → El grado del polinomio lo marca el exponente de mayor valor.
- Función logarítmica f (n) = log2(n)
- Función exponencial f (n) = cn

> Decimos que un método tiene un orden de complejidad O(n) cuando O(n) acota asintóticamente la función de complejidad del método.

Una escala de complejidad de menor a mayor:

Constante	$O(1)$
Logarítmica	$O(log(n))$
Lineal	$O(n)$
N-Log-N	$O(n * log(n))$
Cuadrática	$O(n^2)$
Cúbica	$O(n^3)$
Polinomial de orden k	$O(n^k)$
Exponencial	$O(2^n)..O(m^n)$

Función	n = 32	n = 64	n = 128
$log_2(n)$	5	6	7
n	32	64	128
$n * log_2(n)$	160	384	896
n^2	1.024	4.096	16.384
n^3	32.768	262.144	2.097.152
2^n	4.294.967.296	18.446.744.073.709.551.616	(no cabe)

5. ORDENACIÓN

> Se busca tener **conjuntos de objetos ordenados**, la comparación de punteros no sirve

> El orden puede depender de los valores de un atributo, o de varios, o de una combinación de sus valores

> Mediante los siguientes interfaces podemos definir ordenes totales entre objetos

→ Interfaz java.lang.Comparable:

```
public interface Comparable <T> {
    public int compareTo (T t); }
```

→ Interfaz java.util.Comparator:

```
public interface Comparator <T> {
    public int compare (T t1, T t2); }
```

- **COMPARABLE<T>**

> Implementa el método compareTo para comparar dos objetos de tipo T

> La implementación del método fija el orden de los objetos de tipo T

> La llamada t1.compareTo(t2) donde t1 y t2 son de tipo T, compareTo devuelve un entero res tal que:

- res < 0 si t1 es menor que t2
- res == 0 si t1 es igual que t2
- res > 0 si t1 es mayor que t2

> Debe ser consistente con equals, es decir → t1.equals(t2) <--> t1.compareTo(t2) == 0

- **COMPARATOR<T>**

> Implementan un método de comparación compare para objetos de tipo T

> La llamada c.compare(t1,t2) donde t1 y t2 son de tipo T, y c es de tipo Comparator<T>, compare devuelve un entero res tal que:

- res < 0 si t1 es menor que t2
- res == 0 si t1 es igual que t2
- res > 0 si t1 es mayor que t2

> Debe ser consistente con equals, es decir → t1.equals(t2) <--> c.compare(t1,t2) == 0

6. POSITIONLIST

> Una lista es un TAD contenedor que consiste en una **secuencia lineal de elementos**

> La **posición de los elementos cambia** al insertar y borrar, y puede crecer de acuerdo con las necesidades del programa y de la capacidad del computador

> Las operaciones de PositionList son

- Interrogadores: size, isEmpty
- Acceso: first, last, next, prev
- Inserción: addFirst, addLast, addBefore, addAfter
- Modificación: set
- Borrado: remove
- Conversión: toArray

> PositionList utiliza la interfaz Position<E> para abstraer cada nodo de la lista. Position significa nodo abstracto. Sólo tiene la operación .element() que devuelve el elemento contenido en el nodo

> La excepción IllegalArgumentException será lanzada cuando no se recibe una posición correcta

> Se recorren usando bucles y nodos cursor de tipo Position<E>

> La inicialización suele consistir hacer que el cursor apunte al primer nodo de la lista usando l.first() (o al último, l.last())

> Para avanzar moveremos el cursor a la siguiente posición con l.next(cursor) o a la anterior l.prev(cursor)

> La condición de parada depende del problema. Suele incluir la condición de rango (cursor != null). NOTA: Ojo con los posibles elementos null.

> Todos sus métodos tienen complejidad 1, menos toArray, que es de complejidad n.

7. ITERADORES

> Se puede **abstraer el cursor** para tener **código reutilizable** para otros TADs (siempre que sean de colecciones de elementos) convirtiendo el cursor en un TAD llamado Iterator.

> **No permite el acceso al nodo**, solo al elemento (con .next)

> La iteración se realiza utilizando **métodos del Iterator**, no se usan métodos del TAD

> El método iterator devuelve un iterador inicializado en el primer elemento de la estructura de datos → Iterator <E> it = list.iterator();

> hasNext devuelve true mientras haya algún elemento pendiente de recorrer (¿es el cursor distinto de null?)

> next guarda el elemento al que apunta el cursor, avanza el cursor y devuelve el elemento guardado (puede dejar el cursor a null si es el último)

- Si al crearse el iterador el TAD está **vacío**
  - `hasNext` devuelve `false`
  - `next` lanza `NoSuchElementException`
- Si el TAD **no** está **vacío**
  - `hasNext` devuelve `true`
  - `next` avanza el cursor devolviendo el elemento actual
- Si el cursor se sale del rango de la estructura (p.e. apunta a `null` en una lista)
  - `hasNext` devuelve `false`
  - `next` lanza `NoSuchElementException`

> remove borra del TAD el elemento devuelto por el ultimo next. Tiene que haber un next antes, si no → IllegalArgumentException

> Sólo se puede borrar el último elemento devuelto por el iterador

> Iteración con bucle **for-each** → permite recorrer cualquier estructura de datos iterable de forma homogénea. Recorre el TAD iterable por completo.

"Para todo elemento 'e' de tipo 'E' en el iterable 'list' ejecuta 'System.out.println(e)"'

```
for (E e : list)
    System.out.println(e);
```

8. RECURSIÓN

> Un método recursivo es un método que **se invoca a sí mismo**

> Cualquier programa iterativo tiene uno equivalente recursivo y viceversa

> **Las llamadas recursivas se van apilando**, hasta que una de ellas llega al caso base y ya no se hacen más llamadas recursivas.

> La recursión se suele implementar utilizando la **pila de la memoria** (donde se almacenan los registros de activación de los métodos)

> Pasamos un **parámetro extra** sobre el que vamos acumulando el resultado

> La función debe tener uno o más **casos base** (no recursivos)

> La mayoría de las veces es necesario un **método auxiliar** recursivo

> Las listas de posiciones se pueden definir de forma recursiva como una secuencia de nodos que, o bien es vacía, o está formada por un nodo seguido por una secuencia de nodos.

> Las llamadas recursivas se ejecutan en orden inverso al que se las llama

> Una lista de posiciones puede definirse de forma recursiva como una secuencia de nodos que es, o bien vacía (caso base) o bien está formada por un nodo seguido de una secuencia de nodos

> Para crear una lista nueva de forma recursiva:

- Opción 1: Crear la lista en el caso base e ir pasándola "hacia arriba" en el return del método. Ojo, que esto implica que la lista se construye empezando en la última llamada recursiva

- Opción 2: Recibir una lista ya creada por parámetro e ir añadiendo los elementos en cada llamada recursiva

9. ÁRBOLES

> El objetivo de los arboles es **organizar los datos** de forma jerárquica

> Se suelen utilizar en las implementaciones de **otros TADs** (colas con prioridad, maps ordenados, ...)

> Un árbol general es, o bien vacío, o bien tiene dos componentes: (1) un nodo raíz que contiene un elemento, y (2) un conjunto de cero o más (sub)árboles hijos.

> Un árbol está formado por nodos. Un nodo tiene un elemento y un conjunto de nodos que son la raíz de los subárboles hijos

> Terminología

- **Raíz** ("root"): nodo sin padre
- **Nodo interno** ("internal node"): nodo con al menos un hijo
- **Nodo externo** ("external node"): nodo sin hijos
- **Nodo hoja** ("leaf node"): sinónimo de nodo externo, usaremos estos dos nombres indistintamente
- **Subárbol** ("subtree"): árbol formado por el nodo considerado como raíz junto con todos sus descendientes
- **Ancestro** de un nodo: un nodo 'w' es ancestro de 'v' si y sólo si 'w' es 'v' o 'w' es el padre de 'v' o 'w' es ancestro del padre de 'v'
- **Descendiente** de un nodo (la inversa de ancestro): 'v' es descendiente de 'w' si y sólo si 'w' es ancestro de 'v'

- **Hermano** ("sibling") de un nodo: nodo con el mismo padre
- **Arista** ("edge") de un árbol: par de nodos en relación padre-hijo o hijo-padre
- **Grado** ("degree") de un *nodo*: el número de hijos del nodo
- **Grado** ("degree") de un *árbol*: el máximo de los grados de todos los nodos
- **Camino** ("path") de un árbol: secuencia de nodos tal que cada nodo consecutivo forma una arista. La **longitud del camino** es el número de aristas
- **Árbol ordenado** ("ordered tree"): existe un orden lineal (total) definido para los hijos de cada nodo: primer hijo, segundo hijo, etc...

- La **profundidad de un nodo** ("depth") es la longitud del camino que va desde ese nodo hasta la raíz (o viceversa)
- La **altura de un nodo** ("height") es la longitud del mayor de todos los caminos que van desde el nodo hasta una hoja
- **Altura de un árbol no vacío**: la altura de la raíz
- **Nivel** ('level'): conjunto de nodos con la misma profundidad. Así, tenemos desde el nivel 0 hasta el nivel 'h' donde 'h' es la altura del árbol

> public interface Tree <E> extends Iterable <E> {

> public int size();

> public boolean isEmpty();

> public Position <E> addRoot(E e) throws NonEmptyTreeException;

> public Position <E> root();

> public Position <E> parent(Position <E> p) throws IllegalArgumentException

> public boolean isInternal(Position <E> p);

> public boolean isExternal(Position <E> p);

> public boolean isRoot(Position <E> p);

> public E set(Position <E> p, E e) throws IllegalArgumentException;

> public Iterable <Position <E>> children(Position <E> p); → devuelve un Iterable para recorrer los hijos de un nodo

> Fundamentalmente se dispone de métodos observadores. Los métodos addRoot y set son los únicos métodos modificadores, pero no son suficientes para construir el ´árbol

Ejemplo

Método que indica si un nodo es hermano de otro

```
boolean isSibling(Tree<E> t,
                  Position<E> w,
                  Position<E> v) {
 if (w == v || t.isRoot(w) || t.isRoot(v))  return false;
 return t.parent(w) == t.parent(v);
}
```

Ejemplo

Método que indica si un nodo es ancestro de otro

```
boolean ancestro(Tree<E> tree,
                 Position<E> w,
                 Position<E> v) {

  if (w == v) return true;
  if (tree.isRoot(v)) return false;
  return esAncestro (tree,w,tree.parent(v));
}
```

Ejemplo

Método que calcula la profundidad de un nodo de un árbol dado (recursivo)

```
int depth(Tree<E> tree, Position<E> v)
                    throws InvalidPositionException {
  if (tree.isRoot(v))
    return 0;
  else
    return 1 + depth(tree,tree.parent(v));
}
```

> Las estructuras de datos lineales (p.e. listas, arrays) presentan un único recorrido lógico de todos sus elementos. En el caso de arboles, es posible recorrer todos sus elementos en diferente orden

- Recorrido en **profundidad**
  - Se visitan todos los nodos de una rama (subárbol) antes de pasar a la siguiente rama
  - En **pre-orden**: se visita cada nodo antes de visitar los subárboles hijos. En el ejemplo anterior: 1,2,4,5,6,3,7,8,9
  - En **post-orden**: se visita cada nodo después de visitar los subárboles hijos. En el ejemplo anterior: 4,5,6,2,7,9,8,3,1
- Recorrido en **anchura** (por niveles)
  - Se visitan todos los nodos de un nivel antes de visitar los nodos del siguiente nivel. En el ejemplo anterior: 1,2,3,4,5,6,7,8,9

Método que recorre un árbol en **pre-orden**

```
void preorder(Tree<E> tree)
  if (tree.isEmpty()) {
    return;
  }
  preorder(tree,tree.root());
}

void preorder(Tree<E> tree, Position<E> v)
          throws InvalidPositionException {

  // AQUI "visitamos" el nodo "v"
  for (Position<E> w : tree.children(v)) {
    preorder(tree, w);
  }
}
```

Método que recorre un árbol en **post-orden**

```
void postorder(Tree<E> tree) {
  if (tree.isEmpty()) {
    return;
  }
  postorder(tree,tree.root());
}

void postorder(Tree<E> tree, Position<E> v)
          throws InvalidPositionException {

  for (Position<E> w : tree.children(v)) {
    postorder(tree, w);
  }
  // AQUI "visitamos" el nodo "v"
}
```

Método que recorre un árbol en **anchura**

```
void breadth(Tree<E> tree) {
FIFO<Position<E>> fifo = new FIFOList<Position<E>>();
  fifo.enqueue(tree.root());
  while (!fifo.isEmpty()) {
    Position<E> v = fifo.dequeue();

    System.out.println(v.element());

    for (Position<E> w : tree.children(v)) {
      fifo.enqueue(w);
    }
  }
}
```

> Este interfaz extiende el interfaz Tree<E> con los métodos modificadores necesarios para construir arboles

public interface GeneralTree<E> extends Tree<E> {

> Position <E> addChildFirst (Position <E> parentPos , E e) ;

> Position <E> addChildLast (Position <E> parentPos , E e) ;

> Position <E> insertSiblingBefore (Position <E> siblingPos , E e) ;

> Position <E> insertSiblingAfter (Position <E> siblingPos , E e) ;

> void removeSubtree (Position <E> p) ;

> **Arboles binarios** → tipo especial de árbol en el que todo nodo tiene como mucho 2 hijos, el izquierdo y el derecho

> Un árbol binario es o bien vacío o bien consiste en (1) un nodo raíz, (2) un (sub)árbol izquierdo, y (3) un (sub)árbol derecho.

public interface BinaryTree <E> extends Tree <E> {

> public boolean hasLeft(Position <E> p);

> public boolean hasRight(Position <E> p);

> public Position <E> left(Position <E> p);

> public Position <E> right(Position <E> p);

> public Position <E> insertLeft(Position <E> parentPos , E e)
throws NodeAlreadyExistsException;

> public Position <E> insertRight(Position <E> parentPos , E e)
throws NodeAlreadyExistsException;

> public void removeSubtree (Position <E> pos);

Ejemplo

Método que devuelve la altura de un árbol binario

```
int height(BinaryTree<E> tree, Position<E> v)
      throws InvalidPositionException {
  if (tree.isExternal(v)) return 0;

  int hi = 0, hd = 0;

  if (tree.hasLeft(v))
    hi = height(tree, tree.left(v));
  if (tree.hasRight(v))
    hd = height(tree, tree.right(v));
  return 1 + Math.max(hi,hd);
}
```

▪ **Ejemplo 9.13** Implementación del método `member` que busca un si un elemento está en el árbol.

```
public static <E> boolean member (BinaryTree<E> tree, E e) {
  return member(tree,tree.root(),e);
}

private static <E> boolean member (BinaryTree<E> tree, Position <E> n, E e) {
  if (n.element().equals(e)) return true;
  if (tree.isExternal(n)) return false;

  boolean found = false;

  if (tree.hasLeft(n)) {
    found = member(tree, tree.left(n),e);
  }
  if (!found && tree.hasRight(n)) {
    found = member(tree, tree.right(n),e);
  }
  return found;
}
```

10. COLAS CON PRIORIDAD

> En una cola FIFO el primer elemento en entrar es el primero en salir, en las colas con prioridad el orden de salida viene determinado por la **prioridad** del elemento

> Lo importante es que la cola devuelva **primero** el elemento con **más prioridad**

> Las **entradas** de una cola con prioridad tienen:
- Una **clave** que indica la prioridad del elemento (key)
- Un **valor** que indica el elemento a insertar (value)
- Al par clave-valor lo llamamos entrada (entry)

> En el interfaz **Entry<K,V>** tenemos:
- **K** es la clave (key) de la entrada que vamos a insertar
- **V** es el valor (value) que vamos a insertar

Por convención, la clave establece la **prioridad inversamente**: cuanto menor es la clave mayor es la prioridad. Se utilizará el **orden total** entre las claves, los objetos que se usen para la clave deben ser Comparable o disponer de un Comparator.

Nos podemos encontrar con:
- Dos o más entradas con la **misma clave**, pero distintos valores
- Dos o más entradas con el **mismo valor**, pero distintas claves
- Puede **modificarse la clave o el valor** de una entrada. En el primer caso estamos diciendo que la prioridad de un valor puede cambiarse para esa entrada.

> public interface PriorityQueue <K,V> {

> public int size();

> public boolean isEmpty();

> public void enqueue(K key, V value) throws InvalidKeyException;

> public Entry <K,V> first() throws EmptyPriorityQueueException; → método para consultar el elem con mayor prioridad (con la clave con menor valor)

> public Entry <K,V> dequeue() throws EmptyPriorityQueueException → devuelve el elem con mayor prioridad (con la clave con menor valor) y lo borra de la cola

> **EmptyPriorityQueueException** se lanza cuando se intenta acceder la entrada de clave mínima en una cola vacía

> **InvalidKeyException** se lanza cuando la clave es null o no tiene definido un orden para los elementos de su clase

> En la implementación de colas con prioridad mediante una **lista** usamos una lista ordenada usando el orden total de claves. Se deben tener al menos dos atributos: la lista de entradas y el comparador de claves

> En una implementación con un **montículo** se utiliza un **árbol binario (casi)completo** (puede estar completo o faltarle únicamente nodos “a la derecha” del último nivel). Todos los niveles (menos el último) deben estar completos. El último nivel se va llenando de izquierda a derecha. Para todo nodo distinto de la raíz, su entrada (clave) es mayor o igual que la entrada almacenada en el nodo padre, es decir, todos los caminos de la raíz a las hojas están ordenados de forma ascendiente.

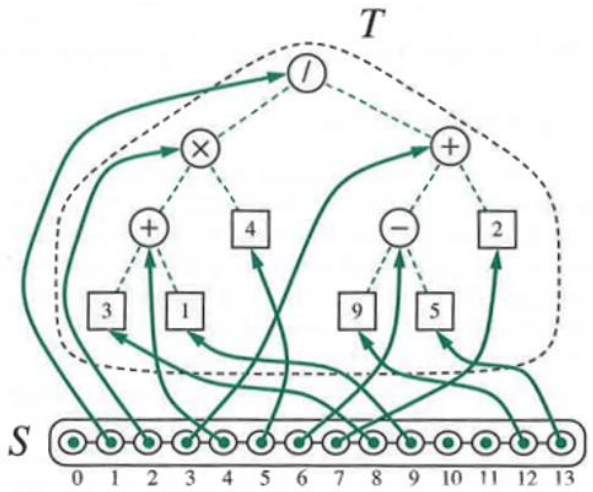
La idea fundamental es conseguir que la posición de una entrada en la cola con prioridad no dependa de la posición absoluta de su clave en el orden total de claves, como ocurre en una implementación de colas con prioridad mediante listas de posiciones.

Al igual que en la implementación con listas, necesitamos un Comparator o bien que las claves sean Comparable

> Operación **enqueue** → La inserción de un nuevo nodo se hace incluyendo el nuevo nodo como el **último nodo del árbol**. Puede ser necesario reajustar los nodos del árbol.
Se comprueba la ”heap-order property” entre el nuevo nodo y su padre, si se cumple hemos acabado, si no se cumple entonces se intercambian las entradas entre el nodo nuevo y el padre. Se repite la operación con el nodo intercambiado y el padre correspondiente hasta que se cumpla la ”heap-order property” o hasta llegar a la raíz.

> Operación **dequeue** → La entrada de menor clave siempre es la raíz, pero la raíz no se puede borrar directamente. Intercambiamos la raíz con el último nodo del árbol y borramos el último nodo. Como esto puede violar la heap-order property puede ser necesario reajustar los nodos del árbol.
Se comprueba la heap-order property entre la raíz y sus hijos. Si se cumple hemos acabado, si no se cumple entonces se intercambian las entradas entre la raíz y el nodo hijo de menor clave. Se repite la operación con el hijo intercambiado y sus hijos hasta que se cumpla la ”heap-order property” o hasta llegar a una hoja.

> Un árbol binario (casi)completo se puede **implementar mediante un array**. Como el árbol es perfecto en todos los niveles menos el último. En el último nivel se meten de izquierda a derecha, es decir, consecutivamente en el array.



11. MAPS

> Los **maps** se usan para organizar la información usando una **clave** para acceder a esta

> Las claves tendrán las propiedades que permitan organizar la información de forma eficiente. La inserción, búsqueda y borrado se hará a través de las claves

> Son conjuntos finitos de **entradas (clave-valor)** donde todas las entradas tienen distinta clave. Si algunas entradas tienen claves iguales, los valores asociados también lo son.

> El requisito para poder implementar un **Map<K,V>** es poder establecer una **relación de igualdad** entre dos elementos de tipo K

- Si K es un objeto "normal" se utilizará el método equals
- Si K implementa Comparable<K> se podrá usar compareTo
- Si se dispone de un Comparator<K> se podrá usar el método compare del comparador

> public interface Map<K,V> {

> public int size();

> public boolean isEmpty();

> public V put(K key, V value) throws InvalidKeyException; → En caso de que la entrada ya exista, reemplaza el valor almacenado para la entrada *key* por *value*

> public V get(K key) throws InvalidKeyException; → si la entrada no existe, devuelve null

> public boolean containsKey(K key) throws InvalidKeyException;

> public V remove(K key) throws InvalidKeyException; → Devuelve el valor almacenado en key o null si no existía

> public Iterator <K> keys();

> public Iterator <Entry <K,V>> entries();

> Notad que si **map.get(k) devuelve null**, puede ser porque:

- no hay ninguna entrada con la clave k
- el valor asociado con la clave k es null. Es decir, hemos ejecutado map.put(k,null) antes

```
// Recorremos Claves
System.out.println("*** Claves");
Iterator<Character> itk = map.keys();
while(itk.hasNext()) {
    System.out.println(k + " " + map.get(itk.next()));
 }

// Recorremos entradas
System.out.println("*** Entradas");
Iterator<Character> ite = map.entries();
while(ite.hasNext()) {
    Entry<Character,Integer> e = it.next();
    System.out.println(e.getKey() + "—" + e.getValue());
 }
```

> **Tablas Hash** → las claves tienen que ser dispersables. Utilizaremos una **función de codificación o función hash**. El objetivo de la función hash es, dada una clave, devolver un valor numérico dentro de un determinado rango [0…K −1]
Se utilizará una **función de compresión/dispersión** cuyo objetivo es comprimir el código de tamaño K obtenido en el punto anterior a tamaño N para que todos los posibles hash codes se compriman y distribuyan al rango [0...N−1]
Usaremos un array de tamaño N como **tabla de dispersión**.

- Dada una función hash que codifica claves en un rango [0...K −1] y un objeto key como clave
- Dada una función de compresión/dispersión que trabaja en un rango [0...N −1]
- Usaremos un array arr de tamaño N para almacenar la tabla
- La secuencia de operaciones sería:

  - Dado un objeto o obtenemos su hash code (p.e. método hashCode de Object)
  - Aplicamos la compresión/dispersión (comprimir) sobre el hash code obtenido
  - La entrada arr[comprimir(o.hashCode)] almacenará la información referente al objeto o

- Si obtenemos la misma posición del array para dos objetos distintos, se ha producido una **colisión.**
Esta se puede producir si el hashCode de dos objetos es el mismo o si la función de compresión devuelve la misma dirección, aunque tengamos dos hash code distintos.

> Es necesario que haya **coherencia entre equals y hashCode.** Si el objeto implementa Comparable, también se debe conservar la coherencia con compareTo.
Dados o1 y o2 usados como claves puede ocurrir:

  - **`o1.hashCode()!=o2.hashCode() && !o1.equals(o2)`**
    - OK. Son dos objetos distintos, dos entradas distintas en el map
  - **`o1.hashCode()==o2.hashCode() && o1.equals(o2)`**
    - OK. Una única entrada en el map
  - **`o1.hashCode()==o2.hashCode() && !o1.equals(o2)`**
    - Colisión. No hay problema, dos entradas distintas en el map
  - **`o1.hashCode()!=o2.hashCode() && o1.equals(o2)`**
    - Inconsistente. Si los objetos son iguales deberían tener el mismo hashCode
    - Se producen dos entradas distintas en el map, teniendo dos objetos iguales (de acuerdo con el resultado de equals)

12. GRAFOS

> Un grafo es una forma de **representar las relaciones** que existen entre pares de objetos

> Un grafo es un conjunto de objetos, llamados **vértices** (vertex), y una colección de **aristas** (edges), donde cada arista conecta dos vértices.

> Las aristas pueden ser **dirigidas** (el par (u,v) está ordenado) o **no dirigidas**.

> Si hay alguna arista dirigida, el grafo es un **grafo dirigido**

> Definiciones:

- Dos vértices son **adyacentes (adjacent)** si hay una arista que los conecta
- El **origen** y **destino** son los vértices inicial y final de una arista dirigida
- Un nodo puede tener **aristas salientes (outgoing edges)** que tienen como origen el nodo y **aristas entrantes (incoming edges)**, que tienen el nodo como destino
- El **grado** de un nodo es el número de aristas que entran y salen del nodo
  - Podemos distinguir entre el grado *entrante* y el grado *saliente*
- Un **camino (path)** es una sencuencia de vértices y aristas que empieza en y acaba en un vértice, de forma que cada arista del camino es adyacente con su vértice anterior y su vértice siguiente del camino
- Un **ciclo (cycle)** es un camino cuyo primer y último nodo son el mismo
- Un **camino simple** es un camino que no repite vértices (no contiene ciclos)
- Un **bosque** es un grafo sin ciclos

> public interface Graph <V, E> {

> public int size();

> public boolean isEmpty();

> public int numVertices();

> public int numEdges();

> public int degree(Vertex <V> v) throws IAE;

> public Iterable <Vertex <V>> vertices();

> public Iterable <Edge <E>> edges();

> public V set(Vertex <V> p, V o) throws IAE;

> public E set(Edge <E> p, E o) throws IAE;

> public Vertex <V> insertVertex(V o);

> public V removeVertex(Vertex <V> v) throws IAE;

> public E removeEdge(Edge <E> e) throws IAE;

> Un grafo dispone de dos genéricos <V,E> para almacenar información en los vértices (V) y en las aristas (E)

> **UndirectedGraph<V,E>** define el interfaz de un grafo no dirigido
public interface UndirectedGraph <V,E> extends Graph <V,E> {

> public Iterable <Vertex <V>> endVertices(Edge <E> e) throws IAE;

> public Edge <E> insertUndirectedEdge(Vertex <V> u, Vertex <V> v, E o) throws IAE;

> public Vertex <V> opposite(Vertex <V> v, Edge <E> e) throws IAE; → devuelve el nodo que está "al otro lado de la arista"

> public boolean areAdjacent(Vertex <V> u, Vertex <V> v) throws IAE; → permite saber si dos nodos están conectados mediante alguna arista

> public Iterable <Edge <E>> edges(Vertex <V> v) throws IAE;

> **DirectedGraph<V,E>** define el interfaz de un grafo dirigido
public interface DirectedGraph <V,E> extends Graph <V,E> {

> public Vertex <V> startVertex(Edge <E> e) throws IAE;

> public Vertex <V> endVertex(Edge <E> e) throws IAE;

> public Edge <E> insertDirectedEdge(Vertex <V> from , Vertex <V> to, E o) throws IAE;

> public Iterable <Edge <E>> outgoingEdges(Vertex <V> v) throws IAE;

> public Iterable <Edge <E>> incomingEdges(Vertex <V> v) throws IAE;

> public int inDegree(Vertex <V> v) throws IAE;

> public int outDegree(Vertex <V> v) throws IAE;

▪ **Ejemplo 12.6** Método que devuelve un conjunto con los vértices visitables desde el vértice `startVertex`.

```
public static <V,E> Set<Vertex<V>> reachableVertices (DirectedGraph<V,E> g,
Vertex<V> startVertex) {

  Set<Vertex<V>> vertices = new HashTableMapSet<Vertex<V>>();
  visit(g, startVertex, vertices);
  return vertices;
}

private static <V,E> void visit(DirectedGraph<V,E> g,
Vertex<V> vertex,
Set<Vertex<V>> vertices) {
  if (vertices.contains(vertex)) {
    return;
  }
  vertices.add(vertex);
  for (Edge<E> edge : g.outgoingEdges(vertex)) {
    visit(g, g.endVertex(edge), vertices);

  }
}
```