Estructuras de datos no lineales - Parte II

Luisa Micó

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante
Curso 2017-18

Index

1. Objetivos
2. Grafos
3. Conjuntos
   - Tablas hash
   - Conjuntos en `java.util`
4. Mapping

Index

1. Objetivos
2. Grafos
3. Conjuntos
   - Tablas hash
   - Conjuntos en `java.util`
4. Mapping

Objetivos

- describir los tipos de datos no lineales más comunes y las estructuras de datos que los implementan, describiendo sus operaciones básicas y tiempos de ejecución
  - grafos
  - conjuntos
- conocer las interfaces y clases en las librerías de Java
- conocer algunas aplicaciones de estructuras de datos no lineales

Visualizaciones:
`https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`

Index

1 Objetivos

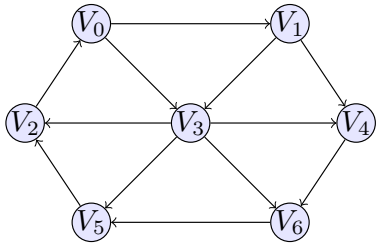
2 Grafos

3 Conjuntos
- Tablas hash
- Conjuntos en `java.util`

4 Mapping

Grafos

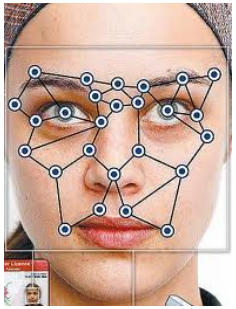
Definición. Un grafo es una **estructura de datos no lineal** que sirve para representar relaciones arbitrarias (no necesariamente jerárquicas) entre objetos.

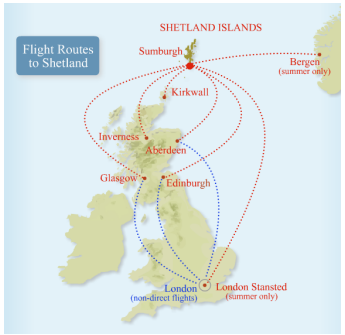


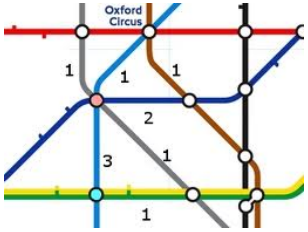
El grafo puede ser **dirigido** $(v, w) \neq (w, v)$ o no.

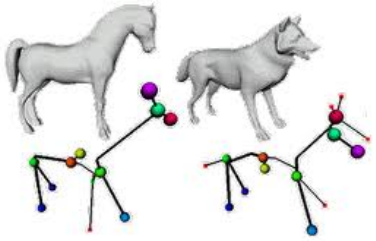
El grafo puede ser **ponderado** (con un peso a la arista entre dos vértices) o no.

Ejemplos

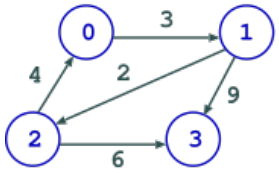




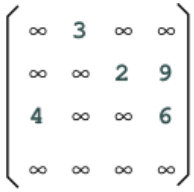




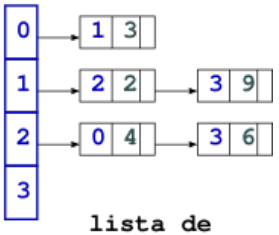
Representaciones de grafos



grafo dirigido ponderado



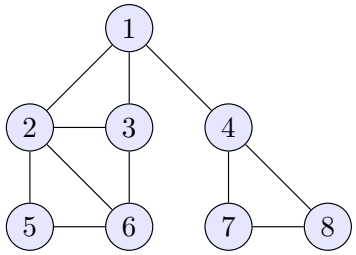
matriz de adyacencia



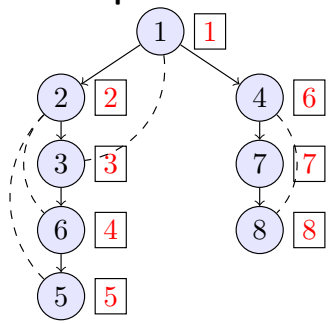
lista de adyacencia

Recorrido en profundidad (DFS) [1]

- Se selecciona un nodo cualquiera como punto de partida (PP) (y se marca).
- Se elige un nodo adyacente a PP como nuevo PP, y se invoca recursivamente al procedimiento.
- En la vuelta atrás, si queda algún nodo adyacente no marcado, se toma como nuevo PP, y así sucesivamente.
- Si queda algún nodo que no haya sido marcado (grafo no conexo), tomamos cualquiera de ellos como nuevo PP y volvemos a empezar.



DFS: Depth First Search



[1] Depth First Search

Algoritmo del recorrido en profundidad

procedure DFS($G = (V, A)$: GRAFODP)
  for every vértice $v \in V$ **do** marca(v) = no visitado; ▷ inicialización
  end for
  for every vértice $v \in V$ **do** ▷ decidir orden
    if marca(v) $\neq$ visitado **then** RP(v);
    end if
  end for
end procedure

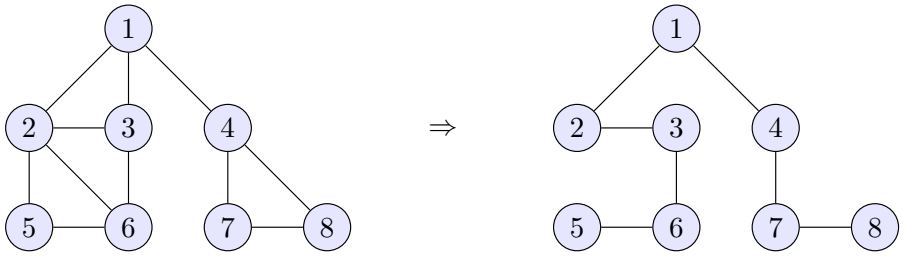
procedure RP(INT v) ▷ versión recursiva
  {El nodo v no ha sido visitado anteriormente}
  marca(v) = visitado;
  for every vértice w adyacente a v **do** ▷ decidir orden
    if marca(w) $\neq$ visitado **then**
      RP(w);
    end if
  end for
end procedure

Características y coste

- El algoritmo se llama de recorrido en profundidad porque inicia tantas llamadas recursivas como sea posible (llegar a una hoja) antes de devolver una llamada.
- Es una **generalización del recorrido en preorden** de un árbol;
- ¿Cuanto tiempo se necesita para explorar un grafo con n nodos y a aristas? dependerá de la implementación:
  - con listas de adyacencia $\mathrm{O}(\max(n,a))$
  - con matriz de adyacencia $\mathrm{O}(n^2)$
- ¿Cuando interesa la representación con listas?

Aristas de cubrimiento mínimo

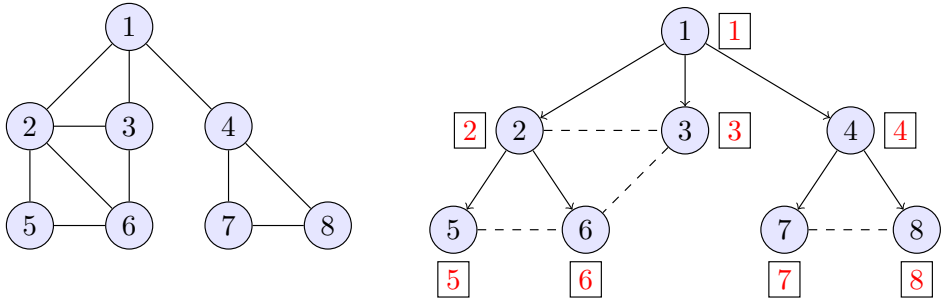
Los nodos solo se visitan una vez. Es decir, si se guardan en alguna estructura las aristas que se van recorriendo se obtiene un **conjunto de aristas de cubrimiento mínimo** del grafo, estructura que se utiliza frecuentemente para reducir la complejidad del grafo. A este conjunto se le conoce como árbol DFS (**DFS Tree**).



Recorrido en anchura (BFS)

Después de visitar un nodo, se visitan sus sucesores, y así sucesivamente. Si queda algún nodo sin visitar (grafo no conexo), repetimos el proceso.

BFS: Breath First Search



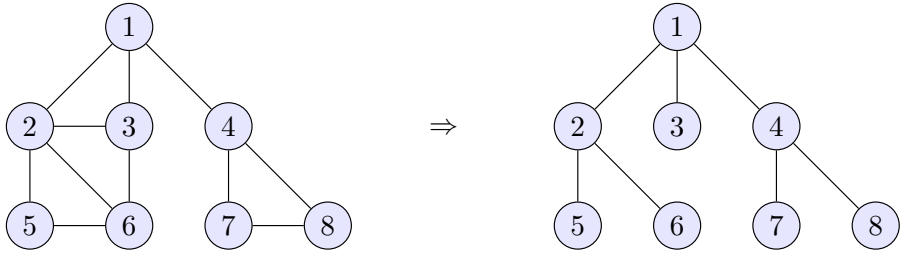
Algoritmo de recorrido en anchura

procedure BFS($G = (V, A)$: GrafoDP)
  for every vértice $v \in V$ **do** ▷ inicialización marca(v) = no visitado;
  end for
  for every vértice $v \in V$ **do**
    if marca(v) $\neq$ visitado **then** RA(v);
    end if
  end for
end procedure

procedure RA(int p)
  Q: cola; marca(p) = visitado; encola(p, Q);
  while Q no está vacía **do**
    v = desencola(Q);
    for every vértice w adyacente a v **do**
      if marca(w) $\neq$ visitado **then**
        marca(w) = visitado; encola(w, Q);
      end if
    end for
  end while

Características y coste

- Es una **generalización del recorrido en niveles** del árbol;
- El coste es equivalente al del recorrido en profundidad.
- Con este recorrido también se obtiene un **conjunto de aristas de cubrimiento mínimo** del grafo, conocido como árbol BFS (**BFS Tree**).

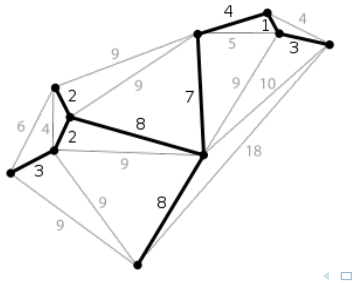


Árboles de expansión mínima (Minimum spanning tree)

Se obtienen cuando lo que queremos es encontrar, sobre un grafo conexo y no dirigido, la **manera menos costosa de conectar todos los vértices del grafo**.

Propiedades:

- si el grafo está ponderado, el árbol de expansión mínima es el árbol cuyo peso (suma de los pesos de todas sus aristas) no es mayor al de ningún otro árbol de expansión;
- el subgrafo que se obtiene no tiene ciclos (es árbol);
- si cada arista tiene un peso distinto, sólo se obtiene un árbol de expansión mínima;
- si todas las aristas tienen el mismo peso, todo árbol de expansión es mínimo;



Algoritmo de Prim (NO)

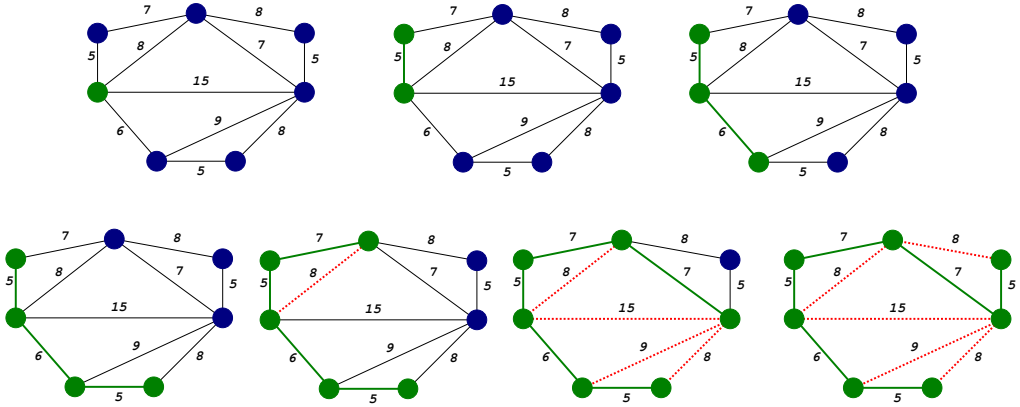
El algoritmo de Prim es un algoritmo de teoría de grafos que encuentra un árbol de expansión mínima en un grafo conexo, no dirigido y ponderado.

1. se marca un nodo cualquiera de salida;
2. seleccionamos la arista de menor valor incidente en el nodo marcado anteriormente, y marcamos el otro nodo en el que incide;
3. se repite el paso 2 si la arista elegida enlaza un nodo marcado y otro no marcado;
4. el proceso termina cuando tenemos todos los nodos del grafo marcados.

Coste: $\mathrm{O}(n^2)$, siendo n el número de vértices.

El algoritmo se basa en un **esquema voraz**.

Ejemplo de aplicación del algoritmo de Prim



Algoritmo de Kruskal (NO)

1. se crea un bosque B (un conjunto de árboles), donde cada vértice del grafo es un árbol separado;
2. se crea un conjunto C que contenga a todas las aristas del grafo*;
3. mientras C es no vacío
   - eliminar una arista de peso mínimo de C;
   - si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol;
   - en caso contrario, se desecha la arista.

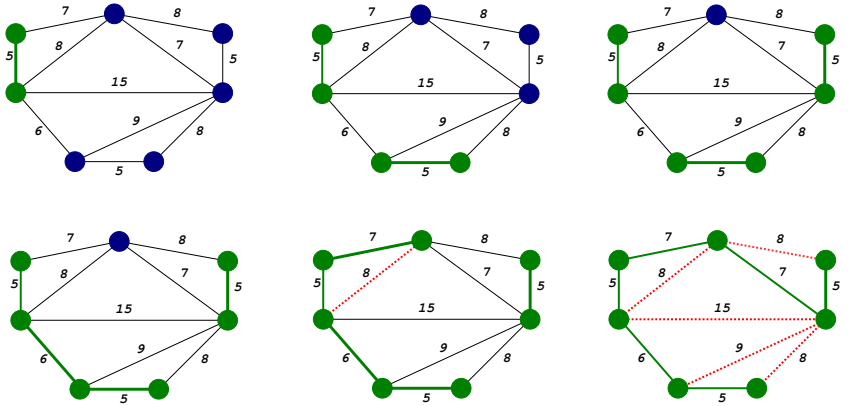
Al acabar el algoritmo, el bosque B tiene un solo componente, el cual forma un árbol de expansión mínima del grafo.

***Coste:** $\mathrm{O}(a \log(n))$, siendo n el número de vértices y a el número de aristas.

El algoritmo se basa en un **esquema voraz**.

Si el grafo no es conexo, entonces busca un bosque expandido mínimo

Ejemplo de aplicación del algoritmo de Kruskal



Comparación de costes (NO)

	Prim $\Theta(n^2)$	**Kruskal** $\Theta(a\log(n))$
grafo denso: $a \to n(n-1)/2$	$\Theta(n^2)$	$\Theta(n^2\log(n))$
grafo disperso: $a \to n$	$\Theta(n^2)$	$\Theta(n\log(n))$

Aplicación: implementación del cableado para el servicio de televisión por cable en ciertos puntos de una ciudad, con el objeto de ahorrar la mayor cantidad de cable (recursos) en los puntos estratégicos (torres de distribución) para llegar a todos los destinos deseados.

`https://www.powtoon.com/online-presentation/bNCG5b6wHsg/algoritmo-de-kruskal-y-aplicacion-en-la-vida-real-parte-1/?mode=movie`

Index

1. Objetivos
2. Grafos
3. Conjuntos
   - Tablas hash
   - Conjuntos en `java.util`
4. Mapping

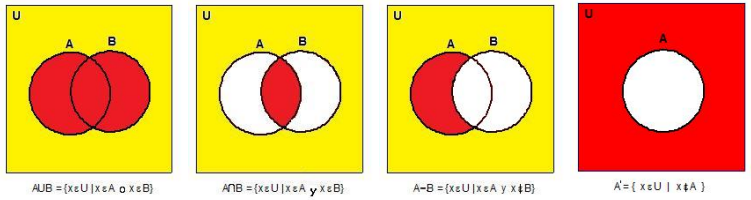
Definición de conjunto

Un conjunto se define como una colección de elementos, cada uno de los cuales puede ser a su vez un conjunto, o un elemento primitivo.

Características:

- todos los elementos de un conjunto son **distintos**
- el orden de los elementos no es importante

Operaciones de conjuntos: inserción, eliminación, búsqueda, vacío, cardinalidad, unión, intersección y diferencia.



Implementación de conjuntos

Implementación: cualquier estructura de datos, analizando el coste de cada operación según su frecuencia de uso y las características de los conjuntos.

- listas
- vectores
- árboles binarios de búsqueda (y versiones)
- **tablas hash**

Muchas aplicaciones requieren búsqueda dinámica **basada sólo en un nombre**. Las tablas hash permiten asociar llaves (o claves) con valores.

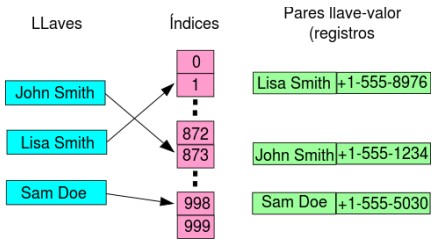
Con las tablas hash se puede conseguir el acceso en **tiempo constante**.

Introducción a las tablas hash

Una **tabla hash** viene definida por:

- un **vector** para almacenar los datos, V;
- una **función de dispersión**, que convierte un elemento en un entero, $h : U \to 0 \ldots |V| - 1$;
- un **método de resolución de colisiones**.

Ejemplo: listín telefónico



Funciones hash (o de dispersión)

Función hash (o de dispersión): función h que pueda transformar una llave particular K (cadena, número, registro ...) en un índice de la tabla.

Función hash perfecta: función inyectiva, que corresponde a que la función transforma diferentes datos de entrada a valores hash diferentes:

$$k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$$

Si la función hash no es inyectiva, se producirán **colisiones**.

Para crear una función hash perfecta, la tabla debe contener al menos el mismo número de posiciones que el número de elementos que se está transformando (pero a veces no se conoce este número).

¿Cómo definir una función h que permita acceder de inmediato a una posición?

Tipos de funciones hash (de localización)

Una función hash debe

- garantizar que el número que devuelve es un índice válido a una de las celdas de la tabla;
- calcularse de forma sencilla y eficiente;
- distribuir uniformemente las claves por toda la tabla.

Además debemos evitar el uso de vectores absurdamente grandes.

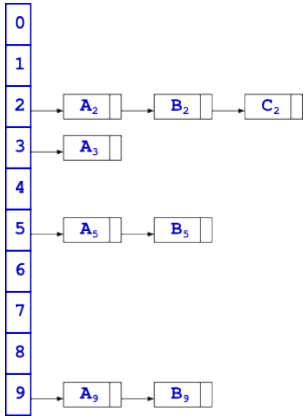
- **División.** Lo más simple es usar el módulo del tamaño de la tabla:
  - si x es un entero (no negativo) arbitrario, entonces (x%`tam_Tabla`) genera un número entre 0 y (`tam_Tabla -1`) [2];
  - pueden aparecer **colisiones** por haber más elementos potenciales que posiciones;
- este método es el preferido para la función hash si se sabe muy poco acerca de las llaves.

[2] se recomienda que `tam_Tabla` sea un número primo

Hashing abierto

La solución consiste en tener una lista de todos los elementos que se dispersan en un mismo valor.

- al buscar, usamos la función de dispersión para decidir qué lista recorrer;
- al insertar un nuevo elemento, lo hacemos en la lista dada por la función de dispersión;
- la tabla nunca puede desbordarse;
- es un método muy rápido para listas cortas;



Hashing cerrado

En un sistema de dispersión cerrada, si ocurre una colisión, se buscan celdas alternativas hasta encontrar una vacía.

- se busca en sucesión en las celdas: $d_0(x)$, $d_1(x)$, $d_2(x)$ ... donde

$$d_i(x) = (h(x) + p(i)) \ \texttt{mod} \ tam\_tabla$$

  y:
  - p es una función de sondeo [3] e i es un sondeo;
  - $p(0) = 0$;
- Como todos los datos se guardan en la tabla, ésta tiene que ser más grande para la dispersión cerrada que para la abierta.

  Exploración lineal: $(p(i) = i) \Rightarrow (h(x) + i) \ \texttt{mod} \ tam\_tabla$

[3] estrategia de resolución de colisiones: lineal, cuadrática, etc

Resolución de colisiones (hashing abierto y cerrado)

Hashing abierto: se genera una lista de colisiones

Hashing cerrado: sobre la tabla las colisiones se resuelven con
- exploración lineal $(p(i) = i)$
- exploración cuadrática $(p(i) = i^2)$

Es evidente que el aumento de tamaño de una tabla hash y la elección de una buena función hash puede reducir las colisiones, pero no eliminarlas.

Factor de carga de una tabla de dispersión (λ): es la relación entre el número de elementos de la tabla (n) y su tamaño (tam_tabla).

$$\lambda = \frac{n}{tam\_tabla}$$

Analizaremos cómo debe ser el factor de carga en el caso de hashing abierto y cerrado.

Ejemplo de hash cerrado con exploración lineal

Insertar A_5, A_2, A_3

0	1	2	3	4	5	6	7	8	9
		A_2	A_3		A_5				

Insertar B_5, A_9, B_2

0	1	2	3	4	5	6	7	8	9
		A_2	A_3	B_2	A_5	B_5			A_9

Insertar B_9, C_2

0	1	2	3	4	5	6	7	8	9
B_9		A_2	A_3	B_2	A_5	B_5	C_2		A_9

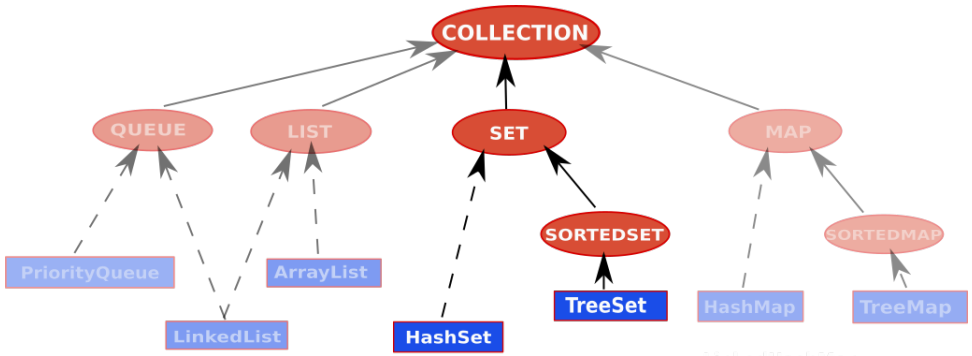
Resumen

Ventajas de tablas de dispersión. Cuando se cumplen todos los requisitos, son una alternativa mejor que los árboles equilibrados para los Tipos Conjunto y Mapa.

Desventajas de tablas de dispersión. No se comportan bien para Tipos con orden interno: Lista ordenada, Cola de Prioridad o Diccionario. λ = factor de carga

- el hash abierto (listas) es más eficiente y con menos degradación
- para mantener el tiempo constante por operación, se recomienda duplicar el tamaño de las tablas
  - si $n > 0,9 \cdot$ `tam_Tabla` en un hash cerrado
  - si $n > 2 \cdot$ `tam_Tabla` en un hash abierto

Implementación de conjuntos (colecciones) en `java.util`



Conjuntos en `java.util`

El interface **Set** se diferencia respecto al interface **Collection** en que no permite etiquetas repetidas (modeliza el "conjunto matemático").

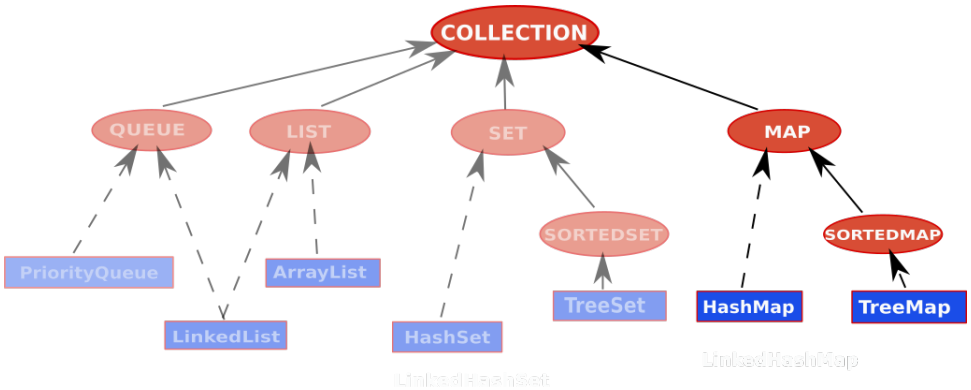
- **HashSet: coste constante** de operaciones básicas (add, remove, contains, size), con la función hash distribuyendo los elementos entre los "buckets"; el constructor por defecto crea una tabla vacía con capacidad inicial 16 y factor de carga igual a 0.75
- **TreeSet: coste logarítmico** de operaciones básicas (add, remove, contains), implementación basada en un **TreeMap** (árbol rojo-negro)

```
LinkedList<String> L = new LinkedList<String>();
L.add("oh"); L.add("as"); L.add("si"); L.add("as"); L.add("oh");
System.out.println("Lista = "+L); //Lista = [oh, as, si, as, oh]

Set<String> H = new HashSet<String>();
H.add("oh"); H.add("as"); H.add("si"); H.add("as"); H.add("oh");
System.out.println("Hash = "+H); //Hash = [as, si, oh]

Set<String> T = new TreeSet<String>();
T.add("oh"); T.add("as"); T.add("si"); T.add("as"); T.add("oh");
System.out.println("Tree = "+T); //Tree = [as, oh, si]
```


Index

1. Objetivos
2. Grafos
3. Conjuntos
   - Tablas hash
   - Conjuntos en `java.util`
4. Mapping

Implementación de mapas en `java.util`



Mapas en java.util

Map<K,V> es un interface que representa colecciones con elementos compuestos de parejas **[clave, valor]**.

- Las claves de los elementos son únicas (son conjuntos);
- cada clave puede tener asignado como mucho un valor;
- el orden del mapa se establece en base a las implementaciones concretas del interface: **HashMap, TreeMap**.

Las operaciones más importantes del interface son:

- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `V get(Object key)`
- `V put(K key, V value)`
- `V remove(Object key)`

Clases:

- **HashMap:** colección que no mantiene un orden específico. Utiliza el código hash sobre las claves para determinar el orden de sus elementos. Coste constante con las operaciones `get` y `put`.
- **TreeMap:** Es un mapa ordenado de forma natural. Coste logarítmico en las operaciones `containsKey`, `get`, `put` y `remove`.

Ejemplos de uso de TreeMap y HashMap

```
Map<String, Integer> L = new TreeMap<String, Integer>();

L.put("o", 2); L.put("c", 1); L.put("o", 7); L.put("M", 6);
L.put("A", 2); L.put("P", 3); L.put("r", 3);
System.out.println(L); // A=2, M=6, P=3, c=1, o=7, r=3

for (Map.Entry<String, Integer> entry : L.entrySet()){
   System.out.println("("+entry.getKey()+","+entry.getValue()+")
       ");}

Map<String, Integer> H = new HashMap<String, Integer>();

H.put("o", 2); H.put("c", 1); H.put("o", 7); H.put("M", 6);
H.put("A", 2); H.put("P", 3); H.put("r", 3);
System.out.println(H); // P=3, A=2, r=3, c=1, M=6, o=7

for (Map.Entry<String, Integer> entry2 : H.entrySet()){
   System.out.println("("+entry2.getKey()+","+entry2.getValue()+
       ")");}
```

Ojo, si hay una clave repetida, pero el valor asociado es distinto, se actualiza el valor asociado a la clave.

Más información sobre HashMap y TreeMap
`http://coding-geek.com/how-does-a-hashmap-work-in-java/`