Diccionarios

Estructuras de datos que permiten almacenar elementos y gestionarlos mediante operaciones basadas en claves.
Es interesante recalcar, que a diferencia de los mapas, los diccionarios permiten múltiples entradas con la misma clave; es decir, permite claves duplicadas.

1. **Operaciones básicas**:
    - **Search(D, k)**: Busca un elemento por su clave kk, devolviendo un puntero al elemento si existe. Complejidad variable según la implementación.
    - **Insert(D, x)**: Añade un elemento xx al diccionario.
    - **Delete(D, x)**: Elimina un elemento xx dado un puntero a él.
2. **Operaciones extendidas** (dependen de la implementación):
    - **Max(D)/Min(D)**: Devuelve el elemento con la clave máxima/mínima. Útil para colas de prioridad.
    - **Predecessor(D, k)/Successor(D, k)**: Encuentran el elemento inmediatamente anterior/posterior a kk en orden lógico. Facilitan recorridos ordenados.
3. **Propiedades clave**:
    - **Acceso por contenido**: No se requiere conocer la posición física del elemento, solo su clave.
    - **Flexibilidad**: La eficiencia de las operaciones varía según la estructura subyacente (arreglos, listas, árboles, tablas hash, etc.).
    - **Ordenación**: Algunas implementaciones mantienen los elementos ordenados, lo que acelera operaciones como *Successor* o *Min*.

Bajemos a tierra, tiremos conceptos relevantes

¿Un diccionario es un mapa?
No, la principal diferencia es que el diccionario permite que haya claves repetidas, mientras que en un mapa no, cada clave es única.

Aspecto	Mapa	Diccionario	HashMap
Claves duplicadas	No permitidas	Permitidas	No permitidas
Orden	Depende de la implementación	Depende de la implementación	No ordenado
Implementación en Java	HashMap, TreeMap, etc.	No estándar (se emula)	Tipo concreto de mapa
Eficiencia	Alta (depende de la impl.)	Variable (según emulación)	O(1)O(1) promedio
Uso típico	Búsquedas rápidas	Claves repetidas + valores	Almacenamiento clave-valor

---

¿Qué onda con los HashMaps?
Son mapas, pero una implementación concreta que utiliza tablas hash para almacenar claves-valor.
Garantizan operaciones en tiempo constante promedio de O(1), para las operaciones de get() y put(), suponiendo una buena función hash.
¿Qué son las colisiones?
Cuando coexisten dos elementos o más, que tengan la misma clave, aun esta haya sido procesada por la función hash. Estos casos son manejados utilizando listas enlazadas o árboles, para almacenar todos aquellos elementos con clave repetida en el HashMap.

A este espacio dedicado a almacenar elementos repetidos, se le conoce como "Bucket".

Asumiendo que utilizamos **arreglos**, el orden varía dependiendo si estos están ordenados o no. Hablando de diccionarios.

Dictionary operation	Unsorted array	Sorted array
Search(L, k)	$O(n)$	$O(\log n)$
Insert(L, x)	$O(1)$	$O(n)$
Delete(L, x)	$O(1)^*$	$O(n)$
Successor(L, x)	$O(n)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(1)$
Minimum(L)	$O(n)$	$O(1)$
Maximum(L)	$O(n)$	$O(1)$

Aludiendo a las operaciones espaciales de sucesor y predecesor, es lógico pensar que las mismas no son eficientes si el array está desordenado, pues en dicho caso, siendo el sucesor una función que devuelve el valor inmediatamente superior a otro valor dado, si el array está ordenado, es simplemente hacer el elemento posterior a A. Si A es Array[8], entonces su sucesor es Array[8+1]. Si se tratase de un array desordenado, había que recorrerlo por completo para garantizar que encontremos a su predecesor y sucesor inmediato.
Pasa algo similar con las operaciones de minimo y maximo, pues si contamos con un array ordenado, simplemente referenciamos al primer y último índice del infame array.

Por que razon es mas eficiente insertar y eliminar en un array desordenado?
Esto se debe simplemente a que, en caso de insertar, si su estructura no sigue un orden, es simplemente asignarle al nuevo elemento el último índice del array. Sin embargo, si queremos insertar ese mismo elemento en un array ordenado, debemos verificar exactamente en qué posición se debería ubicar el nuevo elemento. Por ejemplo, si tenemos un caso: 3-9-10-2-15-4-1. Si quisiéramos insertar el "7",

naturalmente al mismo le corresponde el índice [6], pero para eso hay que mover todos aquellos elementos que se encuentren antes de dichoso e infame índice, garantizando u n tiempo de búsqueda en el peor caso de O(n). De hecho, buscar en qué posición debe estar nuestro nuevo elemento y es de Log(n), pero además suma el hecho de tener que empujar al resto de elementos hacia un lado, lo que es en el peor caso O(n)
En caso de eliminar, si el array está desordenado, podemos simplemente eliminar el elemento que más nos plazca, y reemplazarlo por el último (el cual está referenciado) del arrey; efectuando, de ese modo, solo dos operaciones (quedando en O(2) = O(1)). Si el infame arrey estuviese ordenado, debemos hacer lo propio con el método de insertar, pues eliminar un elemento altera el orden de nuestra pérfida estructura del dato.

En un diccionario, cada elemento tiene un getKey() y un getValue(). “Insertar” en un diccionario, es el equivalente a “poner” o put() en un mapa. Además, estos últimos tienen una función “recuperar(k)”. => Siendo “k” la clave.

Hashing

Principal dificultad: el conjunto de posibles valores resulta ser mucho mayor que el conjunto de direcciones posibles.

- Ejemplo: conjunto de nombres formados por hasta 16 letras, que identifican a los individuos de un grupo de 1000 personas.
- Habrá $26^{16} = 4.36 * 10^{22}$ claves posibles, que deben mapearse en $1 * 10^{3}$ índices posibles.

El buen amigo Allan, en la edición de su libro de la cual no se arrepiente de haber escrito, explica el concepto de tabla hash como una estructura que permite acceso rápido a datos mediante una función hash, que mapea claves a posiciones en un arreglo.
Su gran virtud, es garantizar operaciones de inserción, búsqueda y eliminación en un tiempo promedio de O(1).

¿Qué es el hashCode()?
Es una función, la cual siempre retorna un valor entero, generado mediante un algoritmo de hashing. Aquellos objetos que son iguales, definido por equals(), deben retornar el mismo hashCode. Objetos que sean diferentes, no necesariamente deben retornar hashCodes diferentes entre sí. De todos modos, es importante reconocer, que el que produzcan distintos valores hash si son distintos, es mejor, pues mejora el performance de las HashTables.

Desventajas del Hashing:
Relacionado al tamaño fijo de la tabla. Es necesaria una buena estimación “a priori” del número de elementos a clasificar.

En caso de que se conozca el tamaño del conjunto, normalmente se dimensiona la tabla un 10% más grande de lo necesario.

No se llevan bien con la eliminación. De hecho son estructuras muy ineficientes en este aspecto.