**Estructuras de datos fundamentales**

Las estructuras de datos (DSs) son uno de los fundamentos de la programación y se han introducido progresivamente desde los inicios de la informática.

En esta lectura, vamos a explorar las formas más básicas de organizar elementos de datos en la memoria de modo que esos elementos puedan ser recuperados posteriormente de acuerdo con criterios específicos. La naturaleza de esos criterios, junto con la forma en que se utiliza el almacenamiento y el rendimiento de las operaciones básicas (añadir, eliminar y buscar elementos) son los que determinan las características de una estructura de datos.

Estas DSs fundamentales son los bloques de construcción para las implementaciones de innumerables DSs avanzadas.

**Array**

Esta es una de las estructuras de datos más simples y, sin embargo, de las más utilizadas. Es proporcionada de forma nativa por la mayoría de los lenguajes de programación: un array es una colección de datos homogéneos. A bajo nivel, es, en resumen, un bloque de memoria donde los elementos se almacenan de forma contigua. Muchos lenguajes de programación solo proveen arrays estáticos. Sus tamaños no pueden cambiar y el número de elementos que almacenan debe decidirse cuando se crean (o al menos en la inicialización). Sin embargo, los arrays dinámicos pueden crecer cuando se añaden nuevos elementos y encogerse cuando se eliminan elementos.

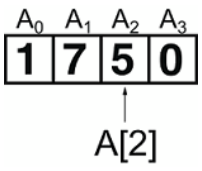
**Nota:** Se puede demostrar que es posible implementar arrays dinámicos de tal manera que, en conjunto la inserción y eliminación son tan rápidas como en los arrays estáticos.

Los elementos en los arrays deben tener todos el mismo tipo y requerir el mismo espacio para ser almacenados (al menos en teoría, la implementación a bajo nivel puede variar entre diferentes lenguajes de programación). Cuando se cumple esta especificación, se facilita pasar de un elemento al siguiente simplemente añadiendo, a la dirección de memoria del primero, el tamaño de los elementos del array.

**Nota:** Debido a que se asignan en un solo bloque de memoria, es más probable que los arrays muestren la llamada localidad de referencia. Al recorrer un array, por ejemplo, es probable que se acceda a datos en la misma página de memoria en una ventana de tiempo corta. Esto puede llevar a varias optimizaciones (el principio de localidad).

La principal ventaja de los arrays es que tienen acceso en tiempo constante para todos los elementos. Sería más preciso decir que cada posición en un array puede accederse en tiempo constante. Es posible recuperar o almacenar el primer elemento, o el último, o cualquier elemento intermedio, siempre y cuando se conozca su posición.

Véase la figura como referencia.



Si bien el acceso aleatorio es una de las fortalezas de los arrays, otras operaciones son más lentas para ellos.  No se puede cambiar el tamaño de un array simplemente añadiendo o eliminando un elemento en su extremo final. Cada vez que se requiere tal operación, se debe reasignar el array, a menos que se esté utilizando un array dinámico. En ese caso, se debe asignar inicialmente un bloque de memoria mayor, llevar un seguimiento del número de elementos y amortiguar la sobrecarga por redimensionamiento en un mayor número de operaciones.

Las listas pueden ser óptimas para inserciones y eliminaciones, aunque son más lentas para el acceso aleatorio.

**Lista enlazada**

Una lista enlazada almacena una secuencia de elementos envolviendo cada elemento en un objeto, llamado nodo.

Cada nodo contiene un valor y uno o dos enlaces (referencias) a otros nodos.

Un valor puede ser un tipo simple, como un número, o un tipo complejo, como una cadena o un objeto. El orden de los elementos está determinado exclusivamente por los enlaces de la lista. Los nodos no necesitan estar asignados de forma contigua y, por lo tanto, las listas son dinámicas por naturaleza. Así, pueden crecer o encogerse según sea necesario.

Más formalmente, una lista se puede definir de forma recursiva . Puede ser:

- Una lista vacía
- Un nodo que contiene un valor y una referencia a una lista enlazada

Otra característica clave de las listas que pueden ser:

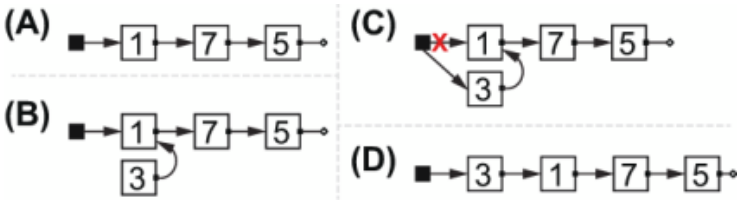
- Enlazadas de manera simple—Cada nodo tiene solo un enlace (también llamado puntero o referencia) al siguiente elemento.
- Enlazadas de manera doble—Se almacenan dos enlaces por nodo: un enlace al siguiente elemento y otro al elemento anterior.

Como se mencionó anteriormente, la elección entre listas enlazadas simples y dobles es un compromiso. La primera requiere menos espacio por nodo, y la segunda permite algoritmos más rápidos para eliminar un elemento, pero necesita una pequeña sobrecarga para mantener los punteros actualizados.

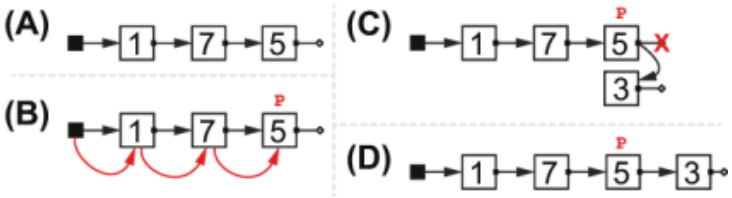
Para llevar un seguimiento de una lista enlazada, la cabecera de la lista debe almacenarse en una variable. Al agregar o eliminar elementos de la lista, se debe tener especial cuidado de actualizar esa referencia a la cabecera de la lista también: si no se realiza la actualización, podría terminar apuntando a un nodo interno o, peor aún, a una ubicación inválida.

La inserción en listas puede ocurrir:

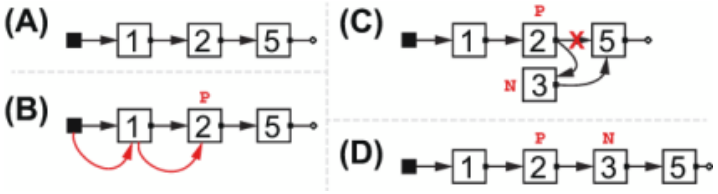
**Al principio:** Ese es el lugar más fácil para añadir un nodo en tiempo constante. La figura  muestra, de hecho, que solo es necesario actualizar la referencia de la cabecera con un nuevo nodo y actualizar los enlaces para este nuevo nodo y la antigua cabecera (a menos que la lista estuviera vacía).




**Al final:** Esta no es una solución muy práctica, ya que se requeriría recorrer toda la lista para encontrar el último nodo; consulta la figura  para ver el motivo. Se podría pensar en mantener un puntero extra para la cola de la lista, pero esto causaría una sobrecarga, ya que cada método que modifique la lista tendría que verificar si esa referencia necesita ser actualizada.




**En cualquier otra posición:** Esto es útil cuando se mantiene la lista ordenada. Sin embargo, también es costoso, requiriendo tiempo lineal en el peor de los casos, como se muestra en la figura




De manera similar, se puede aplicar el mismo razonamiento para la eliminación de nodos:

- Desde el principio:Se puede hacer en tiempo constante actualizando la referencia de la cabecera al sucesor de la antigua cabecera (después de verificar que la lista no esté vacía). Además, se debe asegurar que el nodo eliminado sea desalojado o puesto a disposición para la recolección de basura.
- Desde el final:Es necesario encontrar el penúltimo nodo y actualizar su puntero al siguiente nodo. Para una lista enlazada doble, simplemente se llega al final de la lista y se toma su predecesor. Para listas enlazadas simples, se requiere mantener una referencia tanto al nodo actual como a su predecesor mientras se recorre la lista. Esto es una operación de tiempo lineal.
- Desde cualquier otra posición: Se aplican las mismas consideraciones que para las inserciones, además de la preocupación por mantener una referencia al penúltimo elemento de la lista.

Las listas enlazadas son estructuras de datos recursivas. Esto se deriva de su definición recursiva, lo que significa que los problemas sobre listas pueden abordarse por inducción. Se puede intentar proporcionar una solución para el caso base (la lista vacía) y, a la vez, una forma de combinar algunas acciones sobre la cabecera de la lista con soluciones para su cola (que es una lista más pequeña).

Por ejemplo, si se debe desarrollar un algoritmo para buscar el máximo de una lista de números, se puede hacer lo siguiente:

- Si la lista está vacía, devolver null.
- Si la lista tiene al menos un elemento, tomar la cabecera de la lista (llamarla x) y aplicar el algoritmo a la lista con los N - 1 elementos restantes, obteniéndose un valor y. Entonces, el resultado es:

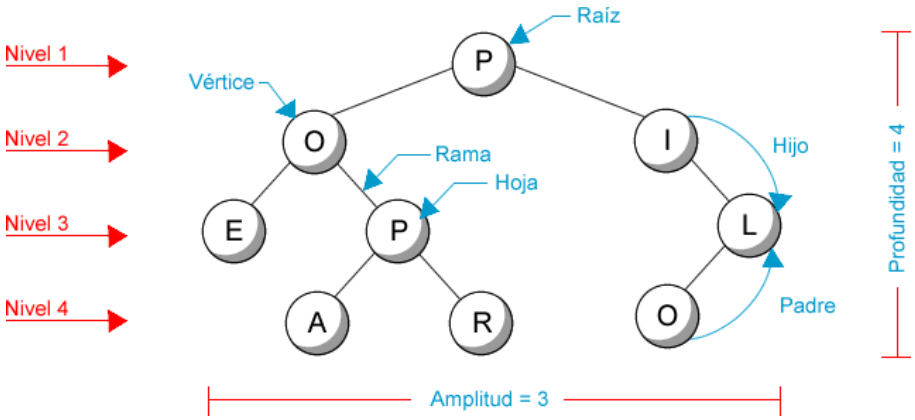
- x, si y es null o x ≥ y
- y, en caso contrario.

**Árbol**

Los árboles son otro tipo de estructura de datos abstracta ampliamente utilizada que proporciona una estructura jerárquica, similar a las listas, pero en forma ramificada. De hecho, pueden considerarse como una generalización de las listas enlazadas: cada nodo aún tiene un único predecesor, llamado su padre, pero puede tener más de un sucesor, aquí denominado sus hijos. Cada hijo de un nodo del árbol es, a su vez, un subárbol (ya sea vacío o que incluya una raíz y sus subárboles).

Un árbol genérico se ilustra en la figura, pero un árbol también se puede definir formalmente como:

- Un árbol vacío.
- Un nodo con una o más referencias a sus hijos y (opcionalmente) una referencia a su padre.




Cada nodo del árbol, al igual que los nodos de una lista, contiene un valor. Además, un nodo posee una lista de referencias a otros nodos, sus hijos. Existe una restricción: en un árbol, cada nodo es hijo de solo un nodo, excepto la raíz, que no tiene padre (por lo que ningún nodo en un árbol apunta a su raíz). Por ello, será necesario mantener una referencia (usando una variable) a la raíz del árbol. Esto es lo mismo que en una lista enlazada simple, donde la cabecera de la lista no está enlazada por otros nodos. Además, los árboles definen una jerarquía “vertical”, una relación padre-hijos, mientras que no puede existir relación entre nodos hermanos o entre nodos en diferentes subárboles.

Antes de continuar, repasemos la terminología utilizada al tratar con árboles:

- Si un nodo x tiene dos hijos, y y z, entonces x es el padre de estos nodos. Así, lo siguiente es cierto: parent(y) == parent(z) == x.
- Siendo x, y y z como se definen aquí, y y z se llaman hermanos (siblings).
- La raíz de un árbol es el único nodo en un árbol que no tiene padre.
- Un ancestro de un nodo x es cualquier nodo en el camino desde la raíz del árbol hasta x. En otras palabras, puede ser parent(x), parent(parent(x)) y así sucesivamente.
- Una hoja es cualquier nodo que no tiene hijos. Otra forma de expresarlo es que, para una hoja, todos los hijos del nodo son subárboles vacíos.
- Una característica fundamental de un árbol es su altura, definida como la longitud del camino más largo desde la raíz hasta una hoja. Puedes identificar la altura en la figura anterior,

**Árboles de búsqueda binaria**

Un árbol binario es un árbol en el que el número de hijos de cualquier nodo es como máximo 2, lo que significa que cada nodo puede tener 0, 1 o 2 hijos.

Un árbol de búsqueda binaria (BST) es un árbol binario en el que cada nodo tiene una clave asociada y satisface dos condiciones: si key(x) es la clave asociada a un nodo x, entonces

- key(x) > key(y) para cada nodo y en el subárbol izquierdo de x.
- key(x) < key(z) para cada nodo z en el subárbol derecho de x.

La clave de un nodo también puede ser su valor, pero en general, los nodos de un BST pueden almacenar una clave y un valor, o cualquier dato adicional, de forma independiente.

Continuemos con algunas definiciones más:

- Un árbol balanceado es un árbol en el que, para cada nodo, la altura de sus subárboles izquierdo y derecho difiere en lo sumo en 1, y ambos subárboles están balanceados
- Un árbol es un árbol completo si tiene altura H y cada nodo hoja se encuentra ya sea en el nivel H o en H-1
- Un árbol perfectamente balanceado es un árbol balanceado en el que, para cada nodo interno, las alturas de sus subárboles izquierdo y derecho son iguales.
- Un árbol perfectamente balanceado también es completo.

**Definiciones:** Existen dos definiciones para el balanceo de un árbol: el balanceo por altura, que es el utilizado en las definiciones proporcionadas, y el balanceo por peso. Son características independientes de un árbol, ya que ninguna implica la otra. Ambas podrían conducir a resultados similares, pero la primera es la que normalmente se utiliza.

Los árboles de búsqueda binaria son estructuras recursivas. De hecho, cada BST puede ser:

- Un árbol vacío
- Un nodo con una clave y un subárbol izquierdo y derecho

Esta naturaleza recursiva permite algoritmos recursivos intuitivos para todas las operaciones básicas en los BST.

Los árboles de búsqueda binaria ofrecen un compromiso entre la flexibilidad y el rendimiento de las inserciones en una lista enlazada y la eficiencia de la búsqueda en un array ordenado. Todas las operaciones básicas (insertar, eliminar, buscar, mínimo y máximo, sucesor y predecesor) requieren examinar un número de nodos proporcional a la altura del árbol.

Por lo tanto, cuanto más corta logremos mantener la altura de un árbol, mejor será el rendimiento en dichas operaciones.

**¿Cuál es la altura mínima posible de un árbol?**

Para un árbol binario con n nodos, log(n) es la altura mínima posible.

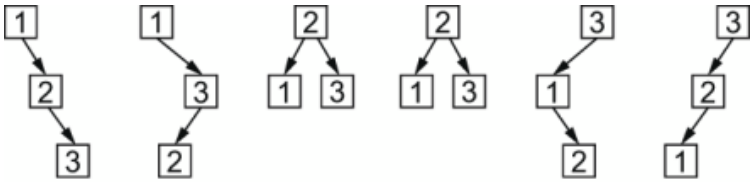
Consideremos un árbol binario. Obviamente, solo puede haber una raíz. Puede tener como máximo 2 hijos, por lo que puede haber como máximo 2 nodos con altura 1. Cada uno de ellos puede tener 2 hijos, por lo que puede haber como máximo $4 = 2^2$ nodos con altura 2. ¿Cuántos nodos con altura 3 puede tener un BST? Como probablemente adivinaste, son $2^3 = 8$. Al descender en el árbol, en cada nivel se incrementa la altura en uno y se duplica el número de nodos que el nivel actual puede contener.

Entonces, a la altura h, tenemos 2^h nodos. Pero también sabemos que el número total de nodos para un árbol completo con altura h es 2^0 + 2^1 + 2^2 + 2^3 + ... + 2^h = 2^(h+1) - 1.

Para mantener la altura lo más pequeña posible, es fácil ver que se deben llenar todos los niveles (excepto, posiblemente, el último), ya que de lo contrario se podrían mover los nodos del último nivel hacia arriba en el árbol hasta que no haya vacantes en los niveles superiores.

Por lo tanto, si tenemos n nodos, 2^(h+1) - 1 ≤ n, y al tomar el logaritmo de ambos lados, se obtiene h ≥ log(n+1) - 1.

Los BST no están balanceados por naturaleza: al contrario, para el mismo conjunto de elementos, su forma y altura pueden variar enormemente dependiendo de la secuencia de inserciones de esos elementos (mira la figura):




En promedio, después de un gran número de inserciones, la probabilidad de tener un árbol sesgado es bastante baja. Cabe mencionar, sin embargo, que el algoritmo más simple para eliminar un nodo tiende a producir árboles sesgados. Se han propuesto varias soluciones alternativas para este problema pero, hasta la fecha, no existe una prueba de que siempre produzcan un mejor resultado que la versión ingenua.

La luz de esperanza es que existen varias soluciones que nos permiten mantener los BST balanceados sin degradar el rendimiento de las operaciones de inserción o eliminación. Por ejemplo:

- Árboles de búsqueda 2-3
- Árboles rojo-negros
- B-trees
- Árboles AVL

**Tabla hash**

El hashing es probablemente la forma más común de representar tablas de símbolos. Si necesitamos asociar cada una de un conjunto de claves a un valor, entonces nos enfrentaremos a algunos problemas.

**Almacenamiento de pares clave-valor**

Se asume que las claves y los valores pueden seleccionarse de diferentes dominios. Entonces, necesitamos decidir si se permiten claves duplicadas. Para simplificar, consideraremos únicamente conjuntos de claves únicas: siempre es posible hacer que un grupo estático de claves sea único.

El caso más sencillo es cuando las claves son enteros no negativos. En teoría, se puede usar un array para almacenar el valor asociado con la clave k en el k-ésimo elemento del array. Esto, sin embargo, solo es posible cuando el rango de las claves posibles es limitado. Si, por ejemplo, se permitiera cualquier entero positivo de 32 bits, necesitaríamos un array con más de 3 mil millones de elementos — posiblemente mayor que la RAM de la máquina más moderna. Se complica aún más si para los valores posibles consideramos “longs”, es decir, enteros de 8 bytes. Estaríamos hablando de 18 mil millones de miles de millones de elementos. No, no es un error tipográfico: estamos hablando de miles de millones de miles de millones.

Lo peor es que, al usar arrays, incluso si sabemos que almacenaremos solo unos pocos miles de claves enteras a la vez, aún necesitaremos un array con $2^{32}$ elementos si se permite cualquier valor entero para una clave.

Aunque hay poco que podamos hacer para mejorar cuando tenemos que almacenar muchos elementos, cuando sabemos que quizás almacenaremos solo un puñado de elementos (digamos, unos pocos

cientos o miles), es otra historia. Incluso si aún podemos elegir estos elementos de un conjunto grande (por ejemplo, todos los enteros que se pueden representar con 32 bits, para un total de alrededor de 4 mil millones de elementos), pero solo almacenaremos un puñado de ellos al mismo tiempo, entonces podemos hacerlo mejor.

Y es precisamente ahí donde el hashing viene al rescate.

**Hashing**

El hashing proporciona un compromiso entre los arrays indexados por claves y los arrays sin ordenar en combinación con la búsqueda secuencial. La primera solución ofrece búsqueda en tiempo constante, pero necesita espacio proporcional al conjunto de claves posibles. La segunda requiere tiempo lineal para la búsqueda, pero espacio proporcional al número de claves reales.

Usando tablas hash, fijamos el tamaño del array en, digamos, M elementos. Como veremos, podríamos almacenar más de M elementos, dependiendo de cómo solucionemos las colisiones. Entonces, transformamos cada clave en un índice entre 0 y M - 1, utilizando una función hash.

Vale la pena notar que introducir tal transformación relaja la restricción de tener solo enteros no negativos como claves. Podemos "serializar" cualquier objeto en una cadena y transformar cualquier cadena en un entero módulo M como parte del hashing. En el resto de la discusión, asumiremos que las claves son enteros por brevedad.

La función hash exacta que necesitamos usar depende del tipo de claves, y está correlacionada con el tamaño del array. Los ejemplos más notables incluyen:

**El método de la división:** Dada una clave entera k, definimos su hash h(k) como h(k) = k % M donde % representa el operador módulo.

Para este método, M, el tamaño de la tabla, debería ser un número primo que no esté demasiado cerca de una potencia de 2.

Por ejemplo, si M = 13, tendríamos h(0) = 0, h(1) = 1, h(2) = 2,... h(13) = 0, h(14) = 1 y así sucesivamente.

**El método de la multiplicación:** $h(k) = \lfloor M \cdot (k \cdot A \% 1) \rfloor$

donde 0 < A < 1 es una constante real, y (k**A % 1) es la parte fraccionaria de k**A. En este caso, M usualmente se elige como una potencia de 2, pero A debe elegirse cuidadosamente, dependiendo de M.

Por ejemplo, digamos que M = 16 y A = 0.25; entonces

k = 0 => h(k) = 0

k = 1 => k * A = 0.25, k**A % 1 = 0.25, h(k) = 4*

*k = 2 => k * A = 0.5, k**A % 1 = 0.5, h(k) = 8

k = 3 => k * A = 0.75, k**A % 1 = 0.75, h(k) = 12*

*k = 4 => k * A = 1, k**A % 1 = 0, h(k) = 0

k = 5 => k * A = 1.25, k*A % 1 = 0.25, h(k) = 4

y así sucesivamente.

Como se puede ver, 0.25 no fue una gran elección para A, porque h(k) solo asumirá cinco valores diferentes. A este respecto, sin embargo, fue una gran elección para ilustrar tanto el método en sí como por qué se debe tener cuidado al elegir sus parámetros.

También existen métodos más avanzados para mejorar la calidad de la función hash de modo que se acerque cada vez más a una distribución uniforme.

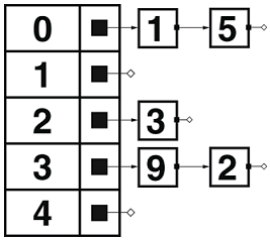
**Resolución de conflictos en el hashing**

No importa cuán buena o uniforme sea la función hash que creemos, el número de claves, m, puede crecer hasta volverse mayor que el tamaño de la tabla, n. En ese punto, el principio del casillero (pigeonhole principle) entra en acción para arruinar nuestra fiesta.

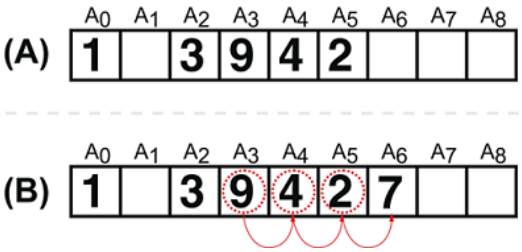
**Definición**: El principio del casillero establece que si el número de valores posibles de clave a almacenar es mayor que los espacios disponibles, en algún punto, es inevitable que dos claves diferentes se asignen al mismo espacio. ¿Qué sucede si intentamos agregar ambas claves a la tabla? En ese caso, tenemos un conflicto. Por lo tanto, necesitamos una forma de poder resolver el conflicto y asegurarnos de que sea posible distinguir entre las dos claves diferentes y encontrarlas ambas en una búsqueda.

Existen dos formas principales de resolver los conflictos debido a que diferentes claves se asignan al mismo espacio:

**Encadenamiento (chaining) :** Cada elemento del array almacena un enlace a otra estructura de datos, que contiene todas las claves asignadas a ese elemento (véase la figura siguiente).
La estructura de datos secundaria puede ser una lista enlazada (usualmente), un árbol o incluso otra tabla hash (como en el hashing perfecto, una técnica que permite el mejor rendimiento posible de hashing en un conjunto estático de claves, conocido de antemano). En este caso, no hay límite en el número de elementos que se pueden almacenar en la tabla hash, pero el rendimiento se degrada a medida que se añaden más y más elementos, ya que al menos algunas listas se alargan y requieren más pasos para ser recorridas a fin de encontrar un elemento.




**Direccionamiento abierto (open addressing) :** Se almacenan los elementos directamente en el array, y en caso de conflicto se genera (de forma determinista) otro valor hash para la siguiente posición a intentar. La figura siguiente muestra un ejemplo de una tabla hash que utiliza direccionamiento abierto para la resolución de conflictos.




Las funciones hash para direccionamiento abierto tienen la forma: h(k, i) = (h'(k) + f(i, k)) % M donde i cuenta el número de posiciones ya examinadas, y f es una función del número de intentos, y posiblemente también de la clave.

El direccionamiento abierto nos permite ahorrar memoria, ya que no hay estructuras de datos secundarias, pero tiene algunos problemas que lo hacen rara vez la mejor opción. Primero y ante todo, eliminar elementos se vuelve demasiado complicado, y debido a que el tamaño de la tabla hash es limitado y se decide en la creación, esto significa que estas tablas se llenarán rápidamente y necesitarán ser reasignadas. Peor aún, los elementos tienden a agruparse en clústeres y, cuando la tabla se llena, se requieren muchos intentos para encontrar un espacio libre — básicamente, hay una alta probabilidad de que se necesite un número lineal de intentos cuando la tabla esté casi medio llena.

**Problemas con la escalabilidad de las tablas hash**

Incluso con el hashing por encadenamiento, el número de espacios en la tabla es usualmente estático, ya que cambiar el tamaño de la tabla cambiaría la función hash y, por lo tanto, el espacio objetivo para potencialmente todos los elementos ya almacenados. Esto, a su vez, nos obligaría a eliminar cada elemento de la tabla antigua para luego insertarlo en la nueva. Vale la pena notar que este tipo de situación suele surgir con caches distribuidas (como Cassandra o Memcached) cuando se necesita agregar un nuevo nodo y a menos que se hayan implementado soluciones adecuadas en la arquitectura de un sitio web— esto puede causar cuellos de botella o incluso hacer que se caiga todo el sitio.

**Rendimiento**

Como se mencionó, el encadenamiento es usualmente el método de elección para resolver colisiones, porque ofrece varias ventajas en términos de tiempo de ejecución y asignación de memoria. Para una tabla hash que utiliza encadenamiento y listas enlazadas, donde la tabla tiene tamaño m y contiene n elementos, todas las operaciones requieren en promedio O(n/m) tiempo.

**Nota:** Aunque la mayor parte del tiempo es correcto considerar que las operaciones en tablas hash son O(1), se debe tener en cuenta que el tiempo en el peor de los casos es O(n). Esto sucede si todos los elementos se asignan a la misma ranura (es decir, la misma cadena) dentro de la tabla. En tales casos, el tiempo necesario para eliminar o buscar un elemento es O(n). Sin embargo, este es un evento muy poco probable, al menos cuando las funciones hash utilizadas están diseñadas adecuadamente.

La buena noticia es que si el conjunto de claves posibles es estático y se conoce de antemano, entonces es posible usar hashing perfecto y tener un tiempo O(1) en el peor de los casos para todas las operaciones.

**Análisis comparativo de las estructuras de datos fundamentales**

Ahora que hemos descrito todas las estructuras de datos fundamentales, intentaremos resumir sus características listando sus propiedades y rendimiento en la tabla siguiente.

Las propiedades que incluiremos son:

- Orden :Si se puede mantener un orden determinista para los elementos. Podría ser un orden natural para los elementos o el orden de inserción.
- Único: Si se prohíben elementos/claves duplicadas.
- Asociativo:Si los elementos pueden ser indexados por una clave.
- Dinámico :Si el contenedor puede redimensionarse al insertar/eliminar, o si su tamaño máximo debe decidirse de antemano.
- Localidad : Localidad de referencia. Si los elementos están almacenados todos en un único bloque ininterrumpido de memoria.

| Estructura | Orden | Único | Asociativo | Dinámico | Localidad |
|---|---|---|---|---|---|
| Array | yes | no | no | no[4] | yes |
| Lista enlazada simple | yes | no | no | yes | no |
| Lista enlazada doble | yes | no | no | yes | no |
| Árbol balanceado (por ejemplo, BST) | yes | no | no | yes | no |
| Heap | no[5] | yes | key-priority | no | yes |
| Tabla hash | no | yes | key-value | yes[6] | no |

[4] (Los arrays son nativamente estáticos en la mayoría de los lenguajes, pero los arrays dinámicos pueden construirse a partir de los estáticos con poca sobrecarga de rendimiento.)

[5] (Los heaps solo definen un orden parcial entre sus claves. Permiten ordenar claves basadas en su prioridad, pero no mantienen ninguna información sobre el orden de inserción.)

[6] (Las tablas hash son dinámicas en tamaño cuando la resolución de conflictos se resuelve mediante encadenamiento.)

Como parte de la comparación, también debemos tener en cuenta su rendimiento relativo. Pero, ¿qué significa realmente el rendimiento para una estructura de datos completa, si solo hemos discutido el tiempo de ejecución de sus métodos individuales?

Usualmente, el rendimiento de una estructura de datos, e incluso más de su implementación, es un compromiso entre los rendimientos individuales de sus métodos. Esto es válido para todas las estructuras de datos: no existe una estructura de datos ideal que tenga un rendimiento óptimo para todas las operaciones que ofrece. Por ejemplo, los arrays son más rápidos para el acceso aleatorio basado en la posición. Pero son lentos cuando se necesita cambiar su forma y muy lentos[7] cuando se necesita buscar un elemento por su valor. Las listas permiten una rápida inserción de nuevos elementos, pero la búsqueda y el acceso por posición son lentos. Las tablas hash tienen una búsqueda rápida basada en la clave, pero encontrar el sucesor de un elemento, o el máximo y el mínimo, es muy lento.

En la práctica, elegir la mejor estructura de datos debe basarse en un análisis cuidadoso del problema y del rendimiento de la estructura de datos, pero es más importante evitar usar una mala estructura de datos (que podría causar un cuello de botella) que encontrar la única mejor opción disponible, solo para obtener un pequeño ahorro en el promedio (y una opción más fácil de implementar).