Resumen para Algoritmos y Estructuras de Datos II

Tomás Spognardi

18 de julio de 2022

Índice

1. Tipos Abstractos de Datos

Un *Tipo Abstracto de Datos* (TAD) está definido por un **tipo**, es decir, un conjunto de valores y operaciones entre los mismos, y la **semántica** de esas operaciones (en este caso, se describe a través de **axiomas**).

Los TADs se diseñan a partir de la especificación informal de un problema. Un paso importante es el de **abstracción**, que consiste en identificar los aspectos relevantes del problema e ignorar el resto de los detalles.

Por otro lado, un TAD no especifica ninguna implementación del comportamiento que describe: delinea el qué, y no el cómo.

Formalmente, cada TAD define una teoría de primer orden con igualdad. Esto se logra especificando la **signatura** de las operaciones, junto con la cual se definen **restricciones** sobre el dominio de cada una, y los **axiomas** que cumplen.

1.1. Secciones de un TAD

Las siguientes son las posibles secciones incluidas en la definición de un TAD.

- **Parámetros formales:** Son los tipos y operaciones requeridas por los TADs paramétricos.
- **Igualdad Observacional:** Define el criterio bajo el cual las instancias de un TAD son indistinguibles.
- **Género:** Es el nombre que recibe el conjunto de valores del tipo.
- **Usa:** Incluye los géneros y operaciones externas con las que interactúa el TAD.
- **Exporta:** Las operaciones y géneros que quedan a disposición de usuarios del tipo.
- **Generadores:** Son las operaciones que permiten construir valores del tipo. Para ser adecuadas, debe ser posible construir cualquier instancia posible a través de ellas. Además, es ideal que sean *minimales*, es decir, el conjunto más chico posible que cumple la condición anterior.
- **Observadores Básicos:** Son las operaciones que permiten obtener información acerca de las instancias de un tipo. También es conveniente que sean minimales.
- **Otras Operaciones:** Las operaciones que no son generadores ni observadores.
- **Axiomas:** Son las reglas que definen el comportamiento de las funciones.

1.2. Errores de especificación

- **Sobreespecificación:** La sobreespecificación se da cuando varios axiomas aplican a la misma instancia, lo cual causa inconsistencias lógicas. Para evitarla, es importante recordar que los TADs no admiten "pattern matching": el orden de los axiomas no tiene ningún efecto sobre su aplicación.
- **Subespecificación:** Implica no definir axiomas sobre el comportamiento de una operación para ciertas instancias del tipo. No siempre es un problema, como en el caso de *dameUno* del TAD CONJUNTO.

- **Casos Restringidos:** Los axiomas no deben especificar sobre casos restringidos, ya que están fuera del dominio.
- **Incongruencia:** Se da cuando una de las operaciones de un TAD no es *congruente*[1] con respecto a la igualdad observacional, es decir, se vuelve posible diferenciar elementos dentro de la misma clase de equivalencia de esta relación.

Una buena práctica para evitar errores es axiomatizar los observadores básicos sobre todos los generadores no restringidos, y definir las otras operaciones utilizándolos. De esta forma se puede garantizar la congruencia con respecto a la igualdad observacional.

1.3. Igualdad observacional

La **igualdad observacional** es un predicado binario entre las instancias de un tipo que indica cuándo son iguales. Es parte del metalenguaje, y como tal no puede ser utilizado dentro de los axiomas. Generalmente, se define en términos de los observadores básicos.

Cuando dos instancias son observacionalmente iguales, se los considera indistinguibles, así que cualquier operación aplicada a ambos debe tener el mismo resultado. Esto puede servir como criterio para definir los observadores básicos: se busca el conjunto minimal de operaciones que permita diferenciar entre todas las instancias de un tipo.

2. Complejidad Computacional

La complejidad computacional es una medida de la eficiencia de un algoritmo en términos de su consumo de recursos, y permite comparar algoritmos entre sí. Los recursos considerados incluyen:

- Tiempo de ejecución.
- Espacio en memoria.
- Cantidad de proceasadores.
- Utilización de la red de comunicaciones.

El análisis de complejidad se puede realizar de dos formas distintas:

- **Empírica:** Es el método más primitivo, y consiste en medir los recursos utilizados por una computadora particular en un conjunto de entradas concreto (idealmente, representativo de las entradas sobre las cuales se va a implementar). Tiene la desventajas de no ser portable y potencialmente consumir mucho tiempo.
- **Teórica:** Se basa en un análisis estático del algoritmo dado un modelo de cómputo, sin necesidad de implementarlo ni correrlo. Es independiente de la máquina y lenguaje de programación, y vale para todas las instancias del problema. Es el método que utilizamos en la cursada.

El análisis teórico de la complejidad computacional se basa en un modelo de cómputo consensuado, y en general se hace en función del tamaño del input. Si hay características de las entradas que causan variaciones importantes en el consumo de recursos, entonces se pueden analizar como distintos tipos de input. Por último, se realiza un análisis del comportamiento asintótico, es decir, el crecimiento de la función a medida que se toman instancias de mayor tamaño.

[1] Una función f es congruente con respecto a una relación de equivalencia $\sim$ cuando $\forall x, y,\ \ x \sim y \iff f(x) \sim f(y)$.

2.1. Modelo de cómputo

Formalmente, un modelo de cómputo define la forma en la que una función matemática es computada dada una entrada. En este caso se utiliza para definir qué operaciones pueden ser realizadas en un tiempo acotado por una constante c, llamadas *Operaciones Elementales* (OE).

Consideramos OEs a las operaciones aritméticas básicas, las operaciones lógicas, las transferencias de control, y asignaciones de variables a tipos básicos. Para contar la cantidad de OEs que realizan las estructuras de control de flujo, se pueden utilizar las siguientes reglas generales:

- Condiciones:

$$T(\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_1 \texttt{ end}) = T(C) + \max\{T(S_1), T(S_2)\}$$

- Loops:

$$T(\texttt{while } C \texttt{ do } S \texttt{ end}) = T(C) + \#\text{iteraciones} \cdot (T(S) + T(C))$$

- Llamadas:

$$T(F(P_1, P_2, P_n)) = 1 + \sum_{i=1}^{n} T(P_i) + T(F)$$

El resto de las estructuras (`for`, `switch`, etc.) pueden ser reescritas en términos de las anteriores. Se debe tener en cuenta que esto da una cota superior de la cantidad de operaciones realizadas, y a veces se requiere un análisis más profundo (en especial en el caso de `if`).

2.2. Cantidad de operaciones elementales

Luego, para cada algoritmo A, se define la función $t_A : I_A \Rightarrow \mathbb{N}$, donde $t_A(I)$ indica la cantidad de operaciones elementales que realiza el algoritmo A para la entrada $I \in I_A$. Luego, si se toma $T_A(I)$ como el tiempo que tarda A en ser ejecutado para la instancia I, se tiene que:

$$T_A(I) \leq c \cdot t_A(I)$$

Esta definición permite comparar el tiempo de ejecución entre distintos algoritmos al contar la cantidad de operaciones elementales que realizan. Más aún, vamos a tomar como unidad de tiempo a la constante c, por lo cual cada OE tomará a lo sumo una unidad de tiempo (y por conveniencia, definimos $T(OE) = 1$, ya que es indistinto a la hora de hacer análisis asintótico).

2.3. Tamaño de la entrada

Considerar la complejidad en función del tamaño de una entrada, en cambio de la entrada en sí, simplifica el análisis. Sin embargo, es posible que distintas instancias del mismo tamaño consuman distintas cantidades de recursos. Para remediar esto, suelen estudiarse tres casos para cada algoritmo:

- **Caso peor:** Es la medida que más se usa, porque asegura garantías certeras (aunque pesimistas) sobre el tiempo de ejecución de un algoritmos. Se define como:

$$T_{\text{peor}}(n) = \max\{t(i) \mid I \in I_A, |I| = n\}$$

- **Caso mejor:** Es dual al caso peor, dado que se define como:

$$T_{\text{mejor}}(n) = \text{mín}\,\{t(i) \mid I \in I_A, |I| = n\}$$

- **Caso promedio:** Intuitivamente $T_{\text{prom}}(n)$ se corresponde al "tiempo esperado" sobre instancias típicas. Definirla requiere conocer la distribución estadística del input, que en muchos casos no es posible. Si $P(I)$ es la probabilidad de que el input sea la instancia I, entonces está definida por:

$$T_{\text{prom}}(n) = \sum_{I \in I_A, |I| = n} P(I) \cdot t(I)$$

2.4. Análisis asintótico

El *orden* de la función $T(n)$ es el que expresa el comportamiento dominante cuando el tamaño de la entrada es grande. Existen distintas medidas para analizarlo:

- $\mathcal{O}$, la cota superior.
- Ω, la cota inferior.
- Θ, el orden exacto.

2.4.1. Cota superior

Se define como:

$$\mathcal{O}(f) = \{g \mid \exists n_0, k > 0 \text{ tal que } \forall n \geq n_0,\ g(n) \leq k \cdot f(n)\}$$

Es decir, $g \in \mathcal{O}(f)$ significa que, asintóticamente, g no crece más que f.

La cota superior permite obtener una garantía sobre el comportamiento asintótico de una función, y cumple las siguientes propiedades:

1. $f \in \mathcal{O}(f)$
2. $f \in \mathcal{O}(g) \implies \mathcal{O}(f) \subseteq \mathcal{O}(g)$
3. $\mathcal{O}(f) = \mathcal{O}(g) \iff f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f)$
4. $f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \implies f \in \mathcal{O}(h)$
5. $f \in \mathcal{O}(g) \wedge f \in \mathcal{O}(h) \implies f \in \mathcal{O}(\text{mín}\,\{g, h\})$
6. $f_1 \in \mathcal{O}(g) \wedge f_2 \in \mathcal{O}(h) \implies f_1 + f_2 \in \mathcal{O}(\text{máx}\,\{g, h\})$
7. $f_1 \in \mathcal{O}(g) \wedge f_2 \in \mathcal{O}(h) \implies f_1 \cdot f_2 \in \mathcal{O}(g \cdot h)$
8. Si existe el límite:

$$\lim_{n \implies \infty} \frac{f(n)}{g(n)} = k$$

Luego:

$$k \neq 0 \wedge k < \infty \implies \mathcal{O}(f) = \mathcal{O}(g)$$
$$k = 0 \implies \mathcal{O}(f) \subsetneq \mathcal{O}(g)$$

2.4.2. Cota inferior

Se define como:

$$\Omega(f) = \{g \mid \exists n_0, k > 0 \text{ tal que } \forall n \geq n_0,\ g(n) \leq k \cdot f(n)\}$$

Esta notación suele usarse para dar cotas inferiores para problemas: a veces es posible demostrar que, dado un problema Π, todo algoritmo A que lo resuelve cumple $T_{\text{peor}}(A) \in \Omega(g)$.

La cota inferior cumple las siguiente propiedades:

1. $f \in \Omega(f)$
2. $f \in \Omega(g) \implies \Omega(f) \subseteq \Omega(g)$
3. $\Omega(f) = \Omega(g) \iff f \in \Omega(g) \wedge g \in \Omega(f)$
4. $f \in \Omega(g) \wedge g \in \Omega(h) \implies f \in \Omega(h)$
5. $f \in \Omega(g) \wedge f \in \Omega(h) \implies f \in \Omega(\text{máx}\{g, h\})$
6. $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \implies f_1 + f_2 \in \Omega(g + h)$
7. $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \implies f_1 \cdot f_2 \in \Omega(g \cdot h)$
8. Si existe el límite:

$$\lim_{n \implies \infty} \frac{f(n)}{g(n)} = k$$

Luego:

$$\begin{aligned} k \neq 0 \wedge k < \infty &\implies \Omega(f) = \Omega(g) \\ k = 0 &\implies \Omega(g) \subsetneq \Omega(f) \end{aligned}$$

2.4.3. Orden exacto

Se define como:

$$\Theta(f) = \{g \mid \exists n_0, k_1 > 0, k_2 > 0 \text{ tal que } \forall n \geq n_0,\ k_1 f(n) \leq g(n) \leq k_2 f(n)\}$$

El orden exacto cumple las siguientes propiedades:

1. $f \in \Theta(f)$
2. $f \in \Theta(g) \implies \Theta(f) = \Theta(g)$
3. $\Theta(f) = \Theta(g) \iff f \in \Theta(g) \wedge g \in \Theta(f)$
4. $f \in \Theta(g) \wedge g \in \Theta(h) \implies f \in \Theta(h)$
5. $f \in \Theta(g) \wedge f \in \Theta(h) \implies f \in \Theta(\text{máx}\{g, h\})$
6. $f_1 \in \Theta(g) \wedge f_2 \in \Theta(h) \implies f_1 + f_2 \in \Theta(\text{máx}\{g, h\})$
7. $f_1 \in \Theta(g) \wedge f_2 \in \Theta(h) \implies f_1 \cdot f_2 \in \Theta(g \cdot h)$

8. Si existe el límite:

$$\lim_{n \Longrightarrow \infty} \frac{f(n)}{g(n)} = k$$

Luego:

$$k \neq 0 \wedge k < \infty \implies \Theta(f) = \Theta(g)$$
$$k = 0 \implies \Theta(g) \neq \Theta(f)$$

9. $f \in \mathcal{O}(g) \wedge f \in \Omega(g) \implies f \in \Theta(g)$

2.5. Complejidades comunes

- $\mathcal{O}(1)$ – **Complejidad constante:** Para operaciones independientes del tamaño de la entrada.
- $\mathcal{O}(\log n)$ – **Complejidad logarítmica:** Suele aparecer en determinados algoritmos recursivos, como la búsqueda binaria. Dado que los logaritmos de distintas bases son proporcionales entre sí, no especificamos ninguna.
- $\mathcal{O}(n)$ – **Complejidad lineal:** Suele aparecer en algoritmos de bucles simples y algunos algoritmos con recursión.
- $\mathcal{O}(n \log n)$ – **Complejidad linearítmica:** Suele aparecer en algunos algoritmos de *Divide & Conquer*, como `mergesort`.
- $\mathcal{O}(n^2)$ – **Complejidad cuadrática:** Suele aparecer en bucles o recursiones doblemente anidados.
- $\mathcal{O}(n^3)$ – **Complejidad cúbica:** Suele aparecer en bucles o recursiones triples.
- $\mathcal{O}(n^k)$ $(k \geq 1)$ – **Complejidad polinómica**.
- $\mathcal{O}(k^n)$ $(k > 1)$ – **Complejidad exponencial:** Suele aparecer en algoritmos de backtracking.

3. Diseño de TADs

3.1. Introducción

Esta sección cubre el pasaje entre descripciones abstractas del comportamiento de los tipos de datos y la implementación de dichos comportamientos. Además de respetar la semántica establecida, se deben considerar aspectos no funcionales, tales como la eficiencia en tiempo y espacio, así como buenas prácticas, como el encapsulamiento. Este pasaje implica un cambio de paradigma: se pasa del funcional, utilizado en la especificación de TADs, al imperativo, más cercano a los lenguajes de programación tradicionales. Sin embargo, esta no es la etapa donde se lleva a cabo la escritura del programa en sí, sino las siguientes tareas:

- Proveer una *representación* para los valores del tipo.
- Definir las *funciones* del tipo.
- **Demostrar** que estas definiciones son correctas.

3.1.1. Contexto de uso

A la hora de decidir entre distintas implementaciones de un mismo TAD, es importante tener en cuenta el contexto de uso: dichas implementaciones pueden diferir en el consumo de recursos para sus operaciones, y se debe examinar cuáles son las importantes según las restricciones que impone cada situación.

3.1.2. Diseño jerárquico

Llevar a cabo un proceso de diseño jerárquico significa definir representaciones de los tipos en funciones de otras representaciones más simples, separando las responsabilidades en la construcción de la solución.

3.2. Metodología de diseño

Habiendo establecido los principios generales que va a seguir el diseño, la metodología es la siguiente:

- Elección del tipo a diseñar (empezando por el más abarcativo).
- Introducció de elementos *no funcionales*.
- Vinculación entre la *representación* y su *abstracción*.
- Iteración sobre los tipos restantes (filosofía *top-down*).

Dentro de este proceso se tienen en cuenta 2 aspectos claves:

- Los *aspectos de la interfaz* de un tipo son aquellos relacionados con el uso del mismo, toda cuestión que resulta visible por sus usuarios. Incluyen
  - **Servicios exportados:** describe para cada operación su complejidad, sus aspectos de *aliasing*, los efectos colaterales sobre sus argumentos, etc.
  - **Interfaz:** define en el paradigma imperativo las operaciones exportadas junto con sus precondiciones y postcondiciones, lo cual establece la correspondencia entre las operaciones de un TAD y sus implementaciones.
- Las *pautas de implementación* son las cuestiones vincualadas a los medios a través de los cuales el tipo garantiza los aspectos de uso, y deben ser tenidas en cuenta por quienes implementen concretamente el diseño.

3.2.1. Transparencia referencial

Una función es *referencialmente transparente* si su resultado se puede determinar únicamente a partir de sus parámetros explícitos. El término es equivalente a "función pura", y las funciones que tienen esta característica son equivalentes a funciones matemáticas. El uso de variables globales o comportamiento no determinístico puede resultar en funciones no referencialmente transparentes.

3.2.2. Aliasing

El *aliasing* se refiere a la posibilidad de tener más de un nombre/referencia/puntero para el mismo objeto. Los aspectos de aliasing que producen las operaciones de un tipo deben ser comunicados a los usuarios para que puedan usarlos eficientemente (evitando copias innecesarias) y sin errores (evitando modificar resultados incorrectamente).

3.2.3. Valores imperativos y lógicos

Como el lenguaje de implementación es distinto al de especificación, es necesario realizar un pasaje entre sus términos. Con este objetivo se define la función $\hat{\bullet} : G_I \to G_T$, apodada *sombrerito*, que para cada valor imperativo devuelve el término lógico correspondiente. La definición de este pasaje queda a cargo de los diseñadores, a través de su *función de abstracción*. Se permite, como abuso de notación, omitir el uso del sombrerito en las interfaces.

3.2.4. Ocultamiento de información

El diseño jerárquico está íntimamente relacionado con el ocultamiento de información, también conocido como encapsulamiento o modularización. Tiene las siguientes ventajas:

- Puede cambiar la implementación y no es necesario modificar el uso.
- Ayuda a modularizar.
- Facilita la comprensión.
- Alienta el reuso.

3.3. Representación

Definir la representación de un módulo implica satisfacer todos los requerimientos declarados en la interfaz. Esto incluye:

- Definir la *estructura de representación*.
- Definir la *función de abstracción*.
- Definir los *algoritmos*.
- Declarar los *servicios* usados que definen otros módulos.

3.3.1. Estructura de representación

La *estructura de representación* describe los valores sobre los cuales se representa el género implementado, y por lo tanto debe poder representar todos los elementos de dicho género. Además, debe ser posible implementar todas las operaciones de acuerdo a las exigencias del contexto de uso.

El *invariante de representación* es un predicado que determina qué instancias de la estructura representan un elemento válido del género. Esto sirve como documentación, y además es una condición necesaria para establecer la relación de abstracción. También es una postcondición implícita que se agrega a todas las operaciones, garantizando que los algoritmos no rompan la estructura, así

como una precondición, asegurando la validez de la misma[2]. El invariante de representación sobre una estructura *estr* se denota como $Rep : \hat{estr} \to bool$.

3.3.2. Función de abstracción

La *función de abstracción* es la herramienta que permite vincular una estructura con algún valor abstracto al que representa. Si T es un género abstracto que se representa con la estructura *estr*, la función de abstracción se denota como $Abs : \hat{estre} \to \hat{T}\{Rep(e)\}$.

- Su dominio está restringido a la instacias de la estructura que cumplen el invariante de representación, pero es total sobre dicho dominio.
- No necesita ser inyectiva: dos instancias válidas pueden representar al mismo término de un TAD.
- Debe ser suryectiva sobre las clases de equivalencia de la igualdad observacional.

3.3.3. Algoritmos

Parte del proceso de diseño implica implementar los algoritmos de la interfaz en pseudocódigo. Para demostrar que son correctos, la función *Abs* debe ser un homomorfismo repescto de la estructura del TAD, es decir, para toda operación $\bullet$ y su implementación $\bullet_i$, $Abs(\bullet_i(p_1, ..., p_n)) =_{obs} \bullet(Abs(p_1), ..., Abs(p_n))$

4. Diseño de conjuntos y diccionarios

4.1. TAD Diccionario

Un *diccionario* es un tipo de datos abstracto que representa una colección de pares de *claves* y *significados*, de forma que cada clave posible aparece a lo sumo una vez en la misma. Es el equivalente computacional a una función matemática con dominio finito, y debe incluir las operaciones:

- **Búsqueda**: también conocida como **lookup**, es la operación que devuelve el significado asociado a una clave dada.
- **Inserción**: también conocida como **definición**, es la operación que asigna un significado a una clave.
- **Borrado**: es la operación que borra el significado asociado a una llave.

En este caso vamos a partir la búsqueda en las operaciones $def?$ y *obtener*, donde $def?$ indica si una clave tiene un significado asociado, y *obtener* devuelve ese significado (tiene como restricción $\{def?(d, clave)\}$). Esto resalta la relación entre los diccionarios y los conjuntos: se puede implementar un conjunto como un caso particular de un diccionario, donde solo es posible usar la primera operación (es decir, donde las claves solo existen o no, no tienen significado).

Este es uno de los TADs más importantes de la computación. La forma más simple de implementarlo es una secuencia de pares, donde el primer elemento representa una clave y el segundo

[2]Su validez como precondición y postcondición es lo que le da el carácter de invariante.

un significado. Esto resulta en complejidades bastante altas (excepto para la inserción en el caso de un arreglo sin orden), así que no se usa mucho. Otra posibilidad sería usar un arreglo con una posición para cada clave posible (o un *bitset* en el caso del conjunto). La desventaja de esto es que la inicialización y el consumo de espacio es de complejidad $\mathcal{O}(\#(\text{claves posibles}))$, lo cual en muchos casos no es factible.

En cambio, los diccionarios suelen implementarse de 2 maneras distintas: *tablas de hash* (que veremos más adelante) y *árboles de búsqueda*.

4.2. Árbol binario de búsqueda

4.2.1. Definición

Un *árbol binario* es un árbol donde cada nodo tiene a lo sumo 2 hijos, mientras que un *árbol binario de búsqueda* cumple que, para todo nodo, los valores de los elementos de su subárbol izquierdo son menores que el valor de dicho nodo, mientras que los de su subárbol derecho son mayores. Esto es equivalente a la definición recursiva:

- Los valores de los elementos del subárbol izquierdo son menores al de la raíz.
- Los valores de los elementos del subárbol derecho son mayores al de la raíz.
- Tanto el subárbol izquierdo como el derecho son ABB.

4.2.2. Algoritmos

- **Búsqueda:** La búsqueda de una clave es recursiva: si el valor es menor al de la raíz, se busca en el subárbol izquierdo, mientras que si es mayor se busca en el subárbol derecho. Esto continúa hasta que el valor es igual a la raíz, o se llega a un árbol vacío (en cuyo caso el elemento no está presente en el árbol).

Buscar en ABB

```
function BUSCAR(clave c, árbol A)
    if A = nil then
        return false
    else (A = bin(L, (c_r, s_r), R))
        if c = c_r then
            return s_r
        else
            if c < c_r then
                return BUSCAR(c, L)
            else
                return BUSCAR(c, R)
            end if
        end if
    end if
end function
```

La cantidad máxima de llamadas que hace este algoritmo es la altura del árbol. Suponiendo que las claves están distribuidas uniformemente, la complejidad de caso promedio resulta ser $\mathcal{O}(\log n)$, pero la de peor caso es siempre $\mathcal{O}(n)$ (el ejemplo es el árbol degenerado en el que las claves se agregan en orden).

- **Definir**: La inserción de un elemento es análoga a la búsqueda: si el valor del elemento es menor al de la raíz, se inserta en el subárbol izquierdo, mientras que si es mayor se inserta en el subárbol derecho. El proceso continúa hasta llegar a un árbol vacío, donde se inserta el elemento como raíz.

Definir en ABB

```
function DEFINIR(clave c, significado s, árbol A)
    if A = nil then
        return bin(nil, (c, s), nil)
    else (A = bin(L, (c_r, s_r), R))
        if c < c_r then
            return bin(DEFINIR(c, s, L), (c_r, s_r), R)
        else
            return bin(L, (c_r, s_r), DEFINIR(c, s, R))
        end if
    end if
end function
```

- **Borrar**: El algortimo tiene 3 casos distintos.
  - Si el nodo a borrar es una hoja, solo es necesario buscarlo y eliminarlo.
  - Si el nodo tiene un único hijo, se lo busca, elimina, y reemplaza la conexión con el padre por ese hijo.
  - Si el nodo tiene 2 hijos, se debe encontrar su predecesor inmediato o su sucesor inmediato. Por un lado, el predecesor inmediato es el nodo más a la derecha del subárbol izquierdo, y no puede tener hijo derecho (porque es el nodo que más a la derecha está), mientras que el sucesor inmediato es el nodo más a la izquierda del subárbol derecho, y, análogamente, no puede tener hijo izquierdo. Por lo tanto, se pueden intercambiar las posiciones de cualquiera de los dos, y borrar el nodo, que necesariamente tendrá a lo sumo un hijo, por lo cual se pueden usar los procedimientos anteriores.

  La complejidad de este algoritmo es similar a la de **Definir**, porque en todos los casos implica buscar el nodo y realizar una operación constante.

Los problemas de complejidad del ABB podrían resolverse si el árbol tuviera una altura menor. Esto se podría lograr si estuviera *balanceado*, ya que los árboles balanceados de n elementos tienen a lo sumo $\log_2(n)$ niveles.

4.3. Árbol AVL

Mantener un árbol completamente balanceado implicaría que sea completo, lo cual no es factible. Otra posibilidad que aseguraría una altura logarítmica sería requerir que todas las ramas del árbol

difieran en a lo sumo 1, pero eso también es difícil de mantener. En cambio, un *árbol AVL* respeta una propiedad un poco más débil: el *balanceo en altura*. Un árbol está balanceado en altura cuando las alturas de los subárboles izquierdo y derecho de cada nodo difieren en a lo sumo una unidad. Es decir, T cumple:

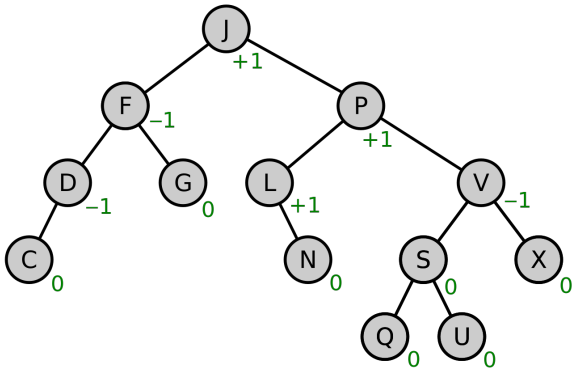
- T_i es AVL.
- T_d es AVL.
- $|H(T_d) - H(T_i)| <= 1$

4.3.1. Factor de balanceo

El *factor de balanceo* de un árbol T se define como la diferencia entre la altura de su subárbol derecho y la de su subárbol izquierdo. Es decir:

$$FDB(T) = H(T_d) - H(T_i)$$

Esta definición puede extenderse a los nodos internos de T, considerando sus respectivos subárboles.



Ejemplo de un árbol AVL, con el factor de balanceo de cada nodo.

4.3.2. Cota sobre altura

Para demostrar que, para todo AVL T de n nodos, $H(T) <= \log_k n$, demostramos que $k^{H(T)} <= n$. Si consideramos los AVLs con una altura h de mínima cantidad de nodos, llamados árboles de Fibonacci de orden h, se tiene que sus subárboles también son árboles de Fibonacci, porque deben ser AVLs (por propiedad de AVL) y de cantidad de nodos mínima (en otro caso, el árbol no sería mínimo). Además, uno debe ser de orden $h-1$ y el otro de $h-2$, debido a que todos los árboles de altura h tienen algún subárbol (derecho o izquierdo) de altura $h-1$, y en este caso el otro puede tener a lo sumo 1 unidad de diferencia (podría ser de la misma altura, pero en ese caso habría árboles con menos nodos). Esto implica que, si AF_h es un árbol Fibonacci de orden h:

$$|AF_{h+2}| = |AF_{h+1}| + |AF_h| + 1$$

Esta recurrencia es muy similar a la definición de la secuencia de Fibonacci, $F_{i+2} = F_{i+1} + F_i$. De hecho:

$$|AF_i| = F_{i+2} - 1$$

Como la secuencia de Fibonacci crece exponencialmente (lo cual implica que $|AF_h| \in \Omega(k^h)$ para algún k) y, por defínción, $|T| > |AF_h|$ para todo árbol AVL T de altura h, la cantidad de nodos de cualquier árbol crece exponencialmente con respecto a su altura, lo cual implica que la altura crece logarítmicamente con respecto a la cantidad de nodos ($H(T) \in \log n$).

4.3.3. Algoritmos

La estructura AVL mantiene el FDB de cada nodo interno, para facilitar las operaciones de rebalanceo.

- **Inserción**: La inserción de un AVL consta de 2 pasos. Primero se debe insertar el nodo de la misma forma que para un ABB, y luego realizar *rotaciones* en caso de que el árbol haya quedado desbalanceado (es decir, queda con un factor de balanceo de ± 2). Las rotaciones son operaciones que preservan la propiedad de ABB, y se realizan de la siguiente manera:

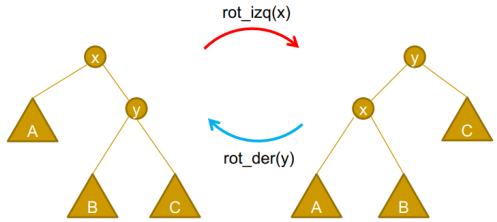


Ilustración de una rotación y su inversa.

Las rotaciones aplicadas se dividen en cuatro casos, donde se toma como referencia el nodo desbalanceado:

- **RR**: Inserción en el subárbol derecho de un hijo derecho.
- **LR**: Inserción en el subárbol izquierdo de un hijo derecho.
- **RL**: Inserción en el subárbol derecho de un hijo izquierdo.
- **LL**: Inserción en el subárbol izquierdo de un hijo izquierdo.

Los casos LL y RR se pueden resolver con una única rotación: se rota a la derecha/izquierda el hijo del nodo desbalanceado. Por otro lado, los casos LR y RL requieren 2 rotaciones: una para el hijo, de forma que quede el caso RR o LL, y otra en el sentido contrario.

La complejidad de este algoritmo está dada por la complejidad de sus pasos:

1. Buscar la posición apropiada del nodo a insertar – $\Theta(\log n)$.
2. Recalcular los FDB – $\Theta(\log n)$.
3. Aplicar la/s rotación/es correspondiente/s – $\Theta(1)$.

- **Borrar**: La operación de borrado se puede definir facilmente en términos de las anteriores:
  1. Borrar el nodo como en un ABB tradicional.
  2. Recalcular los FDB en la rama en que ocurrió el borrado (a lo sumo son $\log n$ nodos).
  3. Para cada nodo con factor de balanceo ± 2, hacer una rotación simple o doble (a lo sumo $\log n$ rotaciones).

Las operaciones de inserción, búsqueda y borrado terminan siendo de complejidad $\Theta(\log n)$ gracias a la cota que asegura el balanceo en altura de los AVLs.

5. Tries

La motivación para esta estructura de datos es obtener una implementación de los conjuntos/diccionarios con operaciones menos dependientes de la cantidad de claves, mantiniendo un rendimiento razonable en función del tamaño. Los *tries* se basan en comparar claves parciales, en lugar de completas, lo cual los hace ideales para situaciones donde se usan strings como tipo de clave. La desventaja es que algunas implementaciones pueden requerir mucha memoria.

5.1. Árboles de búsqueda digital

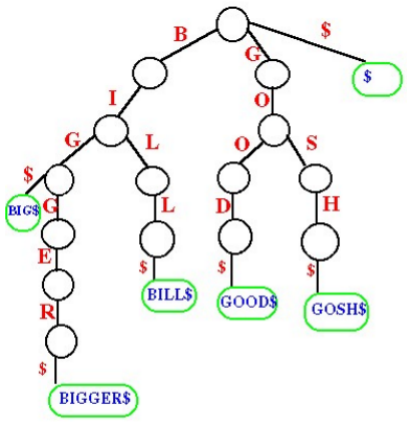
Los *árboles de búsqueda digital* (ABD) son similares a los ABB, en el sentido que para buscar una clave, se realiza una serie de comparaciones a lo largo del árbol. Sin embargo, algunas de estas comparaciones no son entre claves en sí, sino sus símbolos (bits, dígitos, caracteres, etc.), y son estas las que determinan las posiciones de las claves. La regla es que cada nodo se encuentra en una posición determinada por un prefijo de cierta longitud, y sus hijos son aquellas claves que comparten ese prefijo.

La inserción de una nueva clave se logra de la siguiente manera: se recorre el árbol utilizando los símbolos de la clave, hasta llegar a un nodo vacío, donde se coloca la clave. Por otro lado, la búsqueda recorre el árbol de la siguiente manera, comparando cada nodo con la clave buscada. Estas operaciones, dada una clave de b símbolos, realizan $\log n$ comparaciones entre claves en promedio[3] y b en el peor caso.

5.2. Tries

Un *trie*, dado un alfabeto de k elementos, es un árbol $k+1$-ario. En este caso, los ejes toman interés: representan los símbolos de las claves. Por lo tanto, cada subárbol representa al conjunto de claves que comienza con los símbolos de la rama entre la raíz y él. Los nodos internos no contienen claves: todas están en hojas, y se utiliza un símbolo especial para demarcar el fin de una clave (en este caso, \$).

[3] Siendo n la cantidad de claves en el diccionario.



Representación de un trie con claves BIG, BIGGER, BILL, GOOD, GOSH.

A diferencia de los ABD, la estructura resultante de un trie es independiente del orden en el que se agregan sus claves (es decir, hay un único trie para un conjunto de claves).

La búsqueda/inserción de una clave c se realiza siguiendo los ejes de la clave. Esto implica realizar a lo sumo $|c|$ comparaciones de símbolos. Suponiendo una distribución uniforme de las claves, las operaciones requieren $\mathcal{O}(\log n)$ comparaciones entre símbolos en promedio y $|c|$ en el peor caso.

5.2.1. Comparación con AVL

Para comparar a los tries con los árboles AVL, se tiene que tener en cuenta el tamaño máximo de una clave, denotado L. En ambos casos las operaciones están acotadas por la altura de los árboles: para los tries esto es L, mientras que para un AVL esto es $\log n$. Por lo tanto, la eficiencia de los algoritmos dependerá de la relación entre estos dos valores.

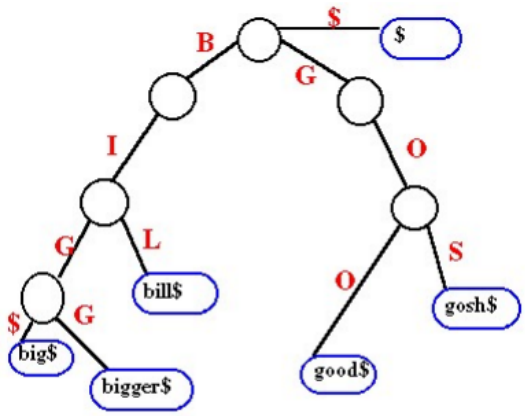
5.2.2. Representación

Las dos posibles representaciones de un trie se corresponden con las dos maneras de representar secuencias: memoria estática (arreglos) y memoria dinámica (listas). Esto se debe a que se está representando un árbol $k+1$-ario, y por lo tanto cada nodo debe guardar una referencia a todos sus hijos.

- Arreglos: Cada nodo interno cuenta con un arreglo de punteros a sus subárboles, donde las posiciones de ese arreglo representan las aristas de los símbolos. Transitar el árbol se vuelve simple: solo hace falta acceder a las posiciones correspondientes de los arreglos para cada símbolo de la clave buscada/insertada. Sin embargo, esta implementación puede ser extremadamente ineficiente en términos de memoria, en especial cuando el alfabeto es grande y la cantidad de claves chica. Esto se podría mitigar agrupando los caracteres poco frecuentes.
- Listas: Los hijos de cada nodo se encuentran en una lista enlazada (opcionalmente ordenada), lo cual hace que recorrer el árbol requiera recorrer cada una de las listas de una rama de nodos. Esto solo es eficiente en términos de tiempo cuando hay pocas claves, pero es mucho más eficiente en términos de espacio que la opción anterior.

5.2.3. Tries compactos

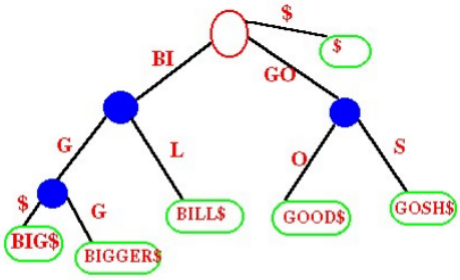
Los *tries compactos* son similares a los tries, pero compactan las cadenas que llevan hacia las hojas. Esto significa que, para cada clave, solo hay nodos internos para el prefijo más largo que comparte con alguna otra clave.



Representación de un trie compacto equivalente al de la figura anterior.

5.2.4. Tries PATRICIA

Los *tries PATRICIA* (Practical Algorithm To Retrieve Information Coded In Alphanumeric) son una versión más compacta aún de los tries compactos. En este caso, solo hay nodos en las posiciones donde 2 o más claves difieren, así que se minimiza la cantidad de nodos internos.



Representación de un trie PATRICIA equivalente al de la figura anterior.

La altura de un árbol PATRICIA está acotada por el número de claves además de la cota anterior (la longitud de la clave más larga).

6. Tablas de hash

Las *tablas de hash* son una implementación de conjunto/diccionario utilizada extensamente en la computación moderna. Resultan particularmente útiles en aplicaciones con acceso a memoria secundaria, donde representan el costo predominante en tiempo.

Son una especie de generalización del concepto de arreglo, donde se pueden utilizar índices distintos de los números entre 0 y $n-1$ (como strings).

6.1. Hashing

Para lograr indexar la estructura utilizando otros tipos de datos, se aplica una función de correspondencia entre el tipo y un entero. Esta transformación depende de la implementación, pero debe ser una función ($x = y \implies f(x) = f(y)$), y es ideal que sea inyectiva ($x = y \impliedby f(x) = f(y)$).

Esta función se denomina *hash*, y tiene la forma $h : K \to \{0, ..., n-1\}$ donde K es el género que contiene a las posibles claves del diccionario y n es el tamaño de la tabla de hash.

6.2. Colisiones

Una función de hash perfecta sería inyectiva, lo cual implicaría que $|K| \leq n$ (la tabla de hash es al menos tan grande como el conjunto de claves posibles). Esto no resulta factible en la práctica, así que toda tabla de hash tiene que tener una forma de manejar *colisiones* entre distintas claves con el mismo hash. Estos métodos se denominan *resolución de colisiones*.

Más allá de la resolución de colisiones, una buena función de hash debería cumplir la propiedad de *uniformidad simple*, que implica que todos los elementos de su imagen tienen probabilidades similares de ser resultado de alguna clave. Formalmente:

$$\forall i \in \{0, ..., n-1\}, \sum_{k \in K | h(k) = i} P(k) \approx \frac{1}{n}$$

Debido a la dificultad (o imposibilidad) de estimar la distribución de las claves, esta propiedad no suele cumplirse en la práctica, así que se intenta encontrar una función independiente de la distribución.

6.2.1. Direccionamiento por concatenación

El *direccionamiento cerrado* o *direccionamiento por concatenación* es simple: a cada posición i del arreglo se le asigna una lista que contiene las claves k (y significados asociados) tales que $h(k) = i$. La búsqueda e inserción de una clave k requiere recorrer la lista asociada a $h(k)$ para encontrarla (o verificar que no está), así que su complejidad es proporcional a la longitud de dicha lista. Si definimos el *factor de carga* $\alpha = \frac{n}{|T|}$, donde n es la cantidad de llaves en la tabla T y $|T|$ su tamaño máximo, se puede demostrar lo siguiente[4]:

- Una búsqueda fallida requiere en promedio tiempo $\Theta(1 + \alpha)$.
- Una búsqueda exitosa requiere en promedio tiempo $\Theta(1 + \frac{\alpha}{2})$.

[4] Asumiendo la hipótesis de simplicidad uniforme de h.

Por ende, esta implementación tiene operaciones de complejidad $\mathcal{O}(1)$ en promedio, siempre que n sea cercano a $|T|$.

6.2.2. Direccionamiento abierto

En los métodos de *direccionamiento abierto*, todos los elementos se almacenan directamente en la carga, y las colisiones se resuelven dentro de la misma. En este caso, redefinimos la función de hashing a $h(k, i)$, que representa la dirección del i-ésimo intento de guardado. Cuando se intenta insertar una clave en una posición ocupada, se sigue intentando hasta encontrar una posición libre. Un esquema del algoritmo de inserción sería:

Insertar en tabla de hash con direccionamiento abierto

```
function INSERTAR(significado s, clave k, diccionario T)
    i ← 0
    while OCUPADO?(T[h(k, i)]) ∧ i < |T| do
        i ← i + 1
    end while
    if i < |T| then
        T[h(k, i)] ← (k, s)
    else
        OVERFLOW
    end if
end function
```

La búsqueda se realiza análogamente, realizando intentos sucesivos hasta que se encuentra la clave, o se llega a una posición vacía (que implica que la clave nunca fue agregada).

Buscar en tabla de hash con direccionamiento abierto

```
function BUSCAR(clave k, diccionario T)
    i ← 0
    while OCUPADO?(T[h(k, i)]) ∧ T[h(k, i)] ≠⊥ ∧ i < |T| do
        i ← i + 1
    end while
    if i < |T| ∧ T[h(k, i)] ≠⊥ then
        return T[h(k, i)]
    else
        return ⊥
    end if
end function
```

Esto presenta un problema: si se inserta una clave, luego se insertan otras que colisionan con ella, y se elimina la primera, las otras claves quedarían inaccesibles (porque el algoritmo retornaría falso al encontrar $\perp$ en el primer intento). Esto se puede solucionar cambiando el método de borrado: en vez de usar $\perp$, se puede usar un valor especial que indica que fue borrado.

La forma en la que se relaciona $h(k, i)$ con $h(k, i+1)$ se denomina *método de barrido*, y algunos son:

- **Barrido linear**: $h(k, i+1) = h(k, i) + 1$[5] (o, equivalentemente, $h(k, i) = h(k) + i$). Es simple, pero es propensa a la aglomeración primaria: si dos secuencias de barrido tienen una colisión, siguen colisionando, y se producen largos tramos de aglomeración ($h(k_1, i) = h(k_2, j) \implies h(k_1, i + a) = h(k_2, j + a)\ \forall a \in \mathbb{N}$).
- **Barrido cuadrático**: En este caso, $h(k, i) = h(k) + c_1 i + c_2 i^2$, con c_1, c_2 constantes. Esto evita el problema anterior, pero sufre de aglomeración secundaria, donde si dos claves colisionan en el primer intento, colisionan siempre ($h(k_1) = h(k_2) \implies h(k_1, i) = h(k_2, i)$).
- **Hashing doble**: Se define $h(k, i) = h_1(k) + ih_2(k)$, donde h_1, h_2 son dos funciones de hashing distintas. Esto reduce la aglomeración secundaria (y no tiene aglomeración primaria). El problema con este esquema es que es ineficiente en su uso del caché de la CPU (debido a que las claves colisionadas podrían quedar en posiciones distantes).

6.3. Funciones de hash

6.3.1. Prehashing

El *prehashing* es la etapa del hashing en la que se pasa del tipo utilizado como clave a un entero. El método debe definirse para cada tipo deseado y, como siempre, es deseable que la función tenga pocas colisiones (o sea inyectiva).

6.3.2. Métodos de hash

Una vez que se tiene un número, existen distintos métodos de hashing:

- **División**: $h(k) = k \mod |T|$. Este método es simple de implementar y rápido de calcular. Para evitar colisiones, es ideal que el tamaño de la tabla sea un número primo no muy cercano a una potencia de 2.
- **Partición**: Se particiona la clave k en $k_1, k_2, ..., k_n$, y luego se define $h(k) = f(k_1, k_2, ..., k_n)$ para alguna función f. La idea es romper los patrones que tenían los datos para lograr una distribución uniforme.
- **Extracción**: Se usa una sola parte de la clave para calcular el hash, la que más varíe dentro del conjunto.

7. Colas de prioridad

El tipo abstracto *cola de prioridad* es similar a una cola o pila donde, en lugar de decidir el siguiente elemento a desencolar a través del orden de inserción, se utiliza una relación de prioridad $<_\alpha$. Esto se puede representar utilizando un AVL, donde la búsqueda y el desencolado se pueden implementar en tiempo logarítmico. En esta sección, exploraremos una implementación más elegante y sencilla (y que posibilita algunas operaciones adicionales).

[5] En todos los casos la función h trabaja con aritmética $\mod |T|$, para evitar salir del rango del arreglo.

7.1. Heap

Un *heap* es un árbol binario perfectamente balanceado que cumple el siguiente invariante: la prioridad de cada nodo es mayor/menor a la de sus hijos. Esto implica que todo subárbol del heap es un heap. Opcionalmente (y siempre en nuestro caso) puede ser *izquierdista*, donde el último nivel se llena de izquierda a derecha. Cuando las prioridades son mayores a las de los hijos, se tiene un máx-heap, mientras que cuando son menores se tiene un mín-heap.

7.1.1. Representaciones

El heap admite cualquier representación de árboles binarios. Una opción podría ser la tradicional, donde cada nodo contiene su valor junto con punteros a sus hijos, pero en este caso la representación con arrays es particularmente eficiente. Consiste en almacenar los valores de los nodos en un array, y relacionar a los padres con sus hijos a través de las posiciones: si v es un nodo almacenado en la posición $p(v)$, su hijo izquierdo se almacena en $2p(v) + 1$, y el derecho en $2p(v) + 2$.

Los arrays tienen la ventaja de ser muy eficientes en términos de espacio y de utilización de caché de CPU, pero su tamaño estático hace necesaria el uso de técnicas adicionales para agregar elementos (como la implementada en los arreglos dimensionables).

7.1.2. Algoritmos

- **Próximo**: El elemento de prioridad máxima siempre se encuentra en la posición 0 del arreglo, y la complejidad de acceder a él es siempre $\Theta(1)$.
- **Encolar**: El algoritmo de inserción consiste en insertar el elemento en la siguiente posición vacía (dada por el orden izquierdista) y *percolar* hacia arriba hasta que se restaure el invariante de heap. Esto implica intercambiar a un nodo con su padre en caso de que este tenga prioridad mayor.

Insertar en heap

```
function INSERTAR(elemento e, heap H)
    Insertar e en la última posición de H.
    while e no está en la raíz ∧_L e > PADRE(e, H) do
        Intercambiar e con PADRE(e, H).
    end while
end function
```

  Este algoritmo corre en tiempo proporcional a la altura del árbol que, por ser perfectamente balanceado, está acotada por $\log_2 n$ (y por ende se tiene una complejidad de $\Theta(\log n)$).

- **Desencolar**: El algoritmo de desencolado es parecido al de inserción: consiste en intercambiar el elemento de la raíz con el último, eliminarlo, y percolar hacia abajo la nueva raíz de éste hasta restaurar el invariante de heap.

Desencolar en heap

```
function DESENCOLAR(heap H)
    result ← RAÍZ(H)
    Intercambiar RAÍZ(H) con su último elemento.
    Eliminar el último elemento.
    e ← RAÍZ(H)
    while e tiene algún hijo ∧_L e < máx{HIJOIZQ(e), HIJODER(e)} do
        Intercambiar e con máx{HIJOIZQ(e), HIJODER(e)}.
    end while
    return result
end function
```

Al igual que en el algoritmo de inserción, el de desencolamiento tiene complejidad $\Theta(\log n)$.

7.1.3. Array a Heap

La operación de transformar un array a un heap se puede implementar simplemente: basta con encolar uno por uno cada elemento del array. De esta forma se logra una complejidad temporal de $\Theta(n \log n)$. Esto se puede mejorar a través del algoritmo de Floyd, que está basado en *heapificar* los subárboles del árbol representado por el arreglo, empezando desde el anteúltimo nivel hacia arriba. Esto se logra a través de la operación BAJAR, que es equivalente a la de percolar hacia abajo en el algoritmo de desencolado. Si se aplica BAJAR a todos los elementos del arreglo de atrás para adelante, se consigue un heap, y la complejidad es de $\Theta(n)$.

8. Ordenamiento

Ahora detallamos varios métodos de ordenamiento de secuencias.

8.1. Selection Sort

Selection sort es un algoritmo simple, que se basa en tomar para cada posición el mínimo del subarreglo desordenado. El ciclo exterior se caracteriza por el siguiente invariante para cada iteración i:

1. Los elementos de $arr[0...i]$ son los $i+1$ más pequeños de todo el arreglo (y menores a todos los demás).
2. Los elementos de $arr[0...i]$ están ordenados.
3. arr es una permutación del arreglo original lo cual, junto con 1., implica que $arr[i+1...n)$ contiene a los $n-i-1$ elementos más grandes del arreglo original.

Selection Sort

for $i \in \{0, ..., n-2\}$ **do**
  $m \leftarrow \arg\min arr[i, n)$
  SWAP(arr, i, m)
end for

Como encontrar la posición del elemento mínimo de un arreglo es $\Theta(n)$ (y swappear es constante), este algoritmo tiene complejidad $\sum_{i=1}^{n-1} \Theta(i) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$. El algoritmo realiza la misma cantidad de operaciones para todos los arreglos, así que este es el costo peor, promedio y mejor.

8.2. Insertion Sort

Al igual que selection sort, *insertion sort* mantiene un subarreglo ordenado, solo que en este caso este es una permutación del subarreglo correspondiente original. Su invariante es el siguiente:

1. Los elementos de $arr[0, i]$ son una permutación de los del arreglo original.
2. Los elementos de $arr[0, i]$ están ordenados.
3. El arreglo arr es una permutación del arreglo original.

Insertion Sort

for $i \in \{0, ..., n-2\}$ **do**
  while $i > 0 \wedge_L arr[i] < arr[i-1]$ **do**
    SWAP(arr, i, $i-1$)
    $i \leftarrow i - 1$
  end while
end for

Análogamente al caso de selection sort, al realizar a lo sumo i comparaciones en cada iteración, el algoritmo tiene complejidad de peor caso $\Theta(n^2)$. Sin embargo, el ciclo interno puede terminar antes, y el algoritmo puede hacer $\Theta(n)$ operaciones en el mejor caso (cuando el arreglo ya se encuentra ordenado).

Este algoritmo es el primero que analizamos que tiene la propiedad de ser *estable*: se mantiene el orden previo de elementos con igual clave. Esto es útil a la hora de realizar varios ordenamientos con distintas claves, ya que se preserva cualquier orden previo que podía existir. En realidad, selection sort puede volverse estable con una modificación: en vez de swappear el elemento mínimo con la posición actual, se lo inserta, corriendo los elementos posteriores.

8.3. Heap Sort

El algoritmo *heap sort* se basa en aprovechar la estructura de heap. Podría pensarse como una versión del selection sort, donde primero se heapifica el arreglo (a través del algoritmo de Floyd) y luego se obtiene el elemento mínimo en cada iteración desencolando en el heap. Esto se puede realizar en tiempo $\Theta(n \log n)$ (son n desencolados).

El heap sort puede implementarse de forma *in-place*, es decir, usando una cantidad constante de memoria adicional. Para ello, se debe convertir el arreglo original en un máx-heap, e ir colocando los elementos desencolados al final del arreglo. De esta manera, las operaciones de heap se realizan sobre el subarreglo desordenado, y luego de n iteraciones el array queda ordenado de menor a mayor. Sin embargo, este algoritmo no es estable.

8.4. Merge Sort

El algoritmo *merge sort* es un ejemplo de la metodología "Divide & Conquer", y sigue el siguiente esquema recursivo:

- **Si** $n < 2$: Es el caso base, y el arreglo está ordenado (tiene 1 o 0 elementos).
- **Si no**:
  1. Dividir el arreglo en dos mitades de igual tamaño.
  2. Aplicar merge sort en cada mitad.
  3. *Mergear* las mitades ordenadas (se puede implementar en tiempo lineal).

El costo de este algoritmo está dado por la recurrencia $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, con el caso base $T(1) = \Theta(1)$. Esto está acotado por $\Theta(n \log n)$.

8.5. Quick Sort

El *quick sort* es otro algoritmo Divide & Conquer. A diferencia de merge sort, la división en cada paso no se realiza a la mitad, sino que se elige un *pivot*, y se particiona el arreglo entre menores y mayores a ese elemento. Luego, el esquema sería el siguiente:

- **Si** $n < 2$: Caso base, el arreglo está ordenado.
- **Si no**:
  1. Se elije el pivot p.
  2. Se calculan las particiones $P_{<p}$ y $P_{\geq p}$.
  3. Se ordenan utilizando quick sort.

  Notar que en el último paso no hace falta mergear las particiones porque, al estar definidas en relación a p, basta con concatenarlas (los elementos mayores a p son mayores que los menores a p por transitividad).

La elección del pivot tiene un gran impacto en la complejidad temporal. Por ejemplo, si siempre se toma el primer elemento del arreglo como pivot, quicksort correrá en tiempo $\Theta(n^2)$ cuando la entrada ya está ordenada, ya que en cada paso de partición solo se reduce en 1 el tamaño del subarreglo a ordenar (la ejecución sería equivalente a selection sort). Lo ideal sería tomar el elemento mediano, pero en la práctica es difícil encontrar ese elemento en tiempo lineal. En cambio, elegir un elemento al azar permite una complejidad promedio de $\Theta(n \log n)$.

La operación más compleja de este algoritmo es realizar la partición. Se puede implementar en tiempo lineal utilizando un algoritmo de 2 punteros:

Particionar un arreglo dada una posición de pivot

```
function PARTICIONAR(arreglo arr, posición p)
    pivot ← arr[p]
    SWAP(arr, p, 0)
    SWAP(arr, arg máx arr, |arr| − 1)
    lower ← 1, upper ← |arr| − 2
    while lower ≤ upper do
        while arr[lower] < pivot do
            lower ← lower + 1
        end while
        while arr[upper] ≥ pivot do
            upper ← upper − 1
        end while
        if lower ≤ upper then
            SWAP(arr, lower, upper)
        end if
    end while
end function
```

8.6. Optimalidad

Para el problema de ordenamiento, se puede demostrar un *lower bound* de $\Omega(n \log n)$, es decir, la complejidad mínima que tiene que cumplir un algoritmo que lo resuelve. En realidad, la cota vale para algoritmos que están basados en comparaciones: si se cuenta con alguna hipótesis adicional (como elementos acotados por un rango), se pueden implementar algoritmos más rápidos (como counting sort).

Para demostrar eso, vamos a asumir que todas las comparaciones tienen la forma de $a_i < a_j$, y que el resultado es siempre *verdadero* o $falso$. Utilizamos la herramienta de *árboles de decisión*, cuyas ramas representan todos los posibles caminos de cómputo que podría tomar un algoritmo. Los nodos internos representan las distintas comparaciones que se realizan, mientras que cada hoja se corresponde con una posible output del algoritmo. La altura del árbol representa la cantidad máxima de ejecuciones posibles.

En el caso del ordenamiento, hay una hoja por cada permutación posible, así que tiene $n!$ nodos hoja. Esto implica que la altura mínima es $\Omega(\log n!) = \Omega(n \log n)$. Por ende, los algoritmos como merge sort y heap sort tiene una complejidad asintóticamente óptima.

9. Dividir y Conquistar

9.1. Defínción

Dividir y Conquistar (DyC) es una técnica de diseño de algoritmos con el siguiente esquema general:

1. **Divide**: Dividir el problema en *subproblemas* más chicos, resolviéndolos recursivamente.
2. **Conquer**: Combinar las soluciones de los subproblemas para obtener la solución del problema original.

Esta metodología tiene sentido siempre y cuando la división y combinación sean implementables de forma eficiente.

Esquema general de DyC

```
function DyC(instancia X)
    if X es suficientemente chico then                    ▷ Caso Base
        return AdHoc(X)
    else                                                  ▷ Caso Recursivo
        X_1, X_2, ..., X_k ← Descomponer(X)
        for i ∈ {1, 2, ..., k} do
            Y_i ← DC(X_i)
        end for
        return Combinar(Y_1, Y_2, ..., Y_k)
    end if
end function
```

9.2. Análisis de complejidad

En un algoritmo típico de DyC, se divide el problema en a subproblemas de tamaño $\frac{n}{c}$, y el costo de la división en sí junto con la combinación está dado por una función $f(n)$. Si $f(n) = bn^d$, entonces se tiene:

$$T(n) = aT\left(\frac{n}{c}\right) + bn^d$$

$$T(n) = bn^d \left(\sum_{i=0}^{\log_c n} \left(\frac{a}{c^d}\right)^i \right)$$

Esta recurrencia puede ser resuelta a través de un análisis de casos:

- Cuando $a = 1, d = 0$ (un solo subproblema y combinación de costo constante, como en la búsqueda binaria):
$$T(n) \in \mathcal{O}(\log n)$$
- Cuando $d = 1$ y:
  - $a < c$ ("pocos subproblemas"):
  $$T(n) \in \mathcal{O}(n)$$
  - $a = c$ ("conquer lineal", como en merge sort):
  $$T(n) \in \mathcal{O}(n \log n)$$
  - $a > c$ ("muchos subproblemas"):
  $$T(n) \in \mathcal{O}(n^{\log_c a})$$

Para otros costos de conquer $f(n)$, se puede aplicar el **Teorema Maestro**:

Teorema. *Si se tiene una relación de recurrencia de la siguiente forma:*

$$T(n) = \begin{cases} aT\left(\frac{n}{c}\right) + f(n) & si\ n > 1 \\ 1 & si\ n = 1 \end{cases}$$

Su fórmula cerrada es:

1. *Cuando* $f(n) \in \mathcal{O}(n^{\log_c a - \epsilon})$ *para algún* $\epsilon > 0$,

$$T(n) \in \Theta(n^{\log_c a})$$

2. *Cuando* $f(n) \in \Theta(n^{\log_c a})$,

$$T(n) \in \Theta(n^{\log_c a} \log n)$$

3. *Cuando* $f(n) \in \Omega(n^{\log_c a + \epsilon})$ *y* $af\left(\frac{n}{c}\right) \leq kf(n)$ *para* $k < 1$ *y* n *suficientemente grande,*

$$T(n) \in \Theta(f(n))$$

En los casos 1. y 3., se cumple $T(n) \in \Theta(f(n) + n^{\log_c a})$.

10. Algoritmos en memoria secundaria

En esta sección vamos a considerar algoritmos para realizar búsqueda y ordenamiento en ambientes donde se cuenta con mucho espacio de almacenamiento, pero el acceso a ese espacio es lento. En particular, para el caso de discos duros, el tiempo de acceso está dominado por la operación de encontrar el dato a leer (el "posicionamiento" del cabezal), así que acceder a un byte tiene un costo similar a acceder a un bloque contiguo.

Si se cuenta con un método con buena *localidad de referencia*, se puede implementar directamente y delegar el acceso a memoria al mecanismo de *memoria virtual* del sistema operativo.

10.1. Ordenamiento–fusión

Una estrategia para ordenar un archivo presente en una memoria de acceso secuencial es seguir el siguiente esquema:

- Dividir el archivo en bloques que entren en memoria.
- Ordenar esos bloques.
- Fusionarlos/Mergearlos.

Vemos 3 ejemplos de algoritmos de ordenamiento-fusión. Su complejidad se mide en *pasadas* por memoria externa, porque ese es el costo que predomina debido al tiempo de búsqueda mencionado.

10.1.1. Fusión múltiple equilibrada

Para el algoritmo de *fusión múltiple equilibrada*, se supone que se tiene que ordenar un archivo de registros presente en una cinta, que sólo puede ser accedido secuencialmente. También se dispone de una memoria con espacio para un número constante de registros (denotado m), y una cantidad arbitraria de cintas auxiliares. El procedimiento es el siguiente:

1. En la primera pasada se realiza el siguiente ciclo: Mientras que queden elementos en la cinta, leer bloques de m elementos, ordenarlos, y escribirlos en la cinta auxiliar i para cada $i \in \{1, ..., p\}$.
2. En la segunda pasada se realiza este otro ciclo: Mientras haya bloques de p en las p cintas auxiliares, fusionarlos a p vías, armando bloques de p^2 en nuevas cintas auxiliares.
3. El paso 2. se repite hasta que todos los bloques queden en una sola cinta, que representan al la secuencia ordenada.

Este algoritmo realiza $\log_p \frac{n}{m}$ pasadas cuando se ordenan n registros con memoria interna de tamaño m y fusiones a p vías.

10.1.2. Selección por sustitución

El algoritmo de *selección por sustitución* es una versión modificada de la fusión múltiple equilibrada. La idea principal es utilizar una cola de prioridad de tamaño m para armar los bloques iniciales, "pasando" los elementos a través de ella al desencolar el menor y sustituirlo por el siguiente. Pero, si el elemento nuevo es menor al que acaba de salir, se lo marca como si fuera mayor, hasta que alguno de los nuevos elementos llegue al primer lugar de la cola, en cuyo caso se inicia un nuevo bloque (que se escribe en otra cinta).

Esta modificación permite construir bloques más largos que los de m registros de la fusión múltiple equilibrada original: para secuencias aleatorias, las secuencias armadas son de largo promedio $2m$. Esto permite reducir la cantidad de pasadas en 1.

10.1.3. Fusión Polifásica

La *fusión polifásica* es otra versión de la fusión múltiple equilibrada, que se enfoca en reducir la cantidad de cintas auxiliares necesarias. Si no se cuenta con las $p + 1$ necesarias para la fusión de p vías, se puede hacer lo siguiente:

- Distribuir los datos en todas las cintas menos una, aplicando la estrategia de "fusión hasta el vaciado".
- Cuando alguna de las cintas se vacía, rebobinarla y usarla como nueva cinta de salida

10.2. Búsqueda externa

La implementación de diccionarios en memoria externa es un problema común en aplicaciones como bases de datos o sistemas de archivos. Se toma el modelo de memoria de acceso a *páginas*, bloques contiguos de datos.

10.2.1. ISAM

El método de *acceso secuencial indexado* (ISAM) se basa en almacenar las claves del diccionario ordenadas en cada *disco*[6] disponible, y además para todos construir un *índice* que se almacena en la primera página de estos. Los índices indican, para cada página, cuál es la última clave de la página anterior. Para tamaños de página suficientemente grandes (donde no puede haber una página llena de un mismo índice), la búsqueda de una clave se puede realizar en $\log_2 \#discos$ accesos a índices y un único acceso a la página . Esto se puede mejorar aún más a través de un *índice maestro*, que indica qué claves están en cada disco y puede ser guardado en memoria principal.

Mantener los índices presenta un problema a la hora de insertar claves, por lo cual esto no suele usarse en la práctica.

10.2.2. Árboles B

Los *Árboles B* son árboles completamente balanceados: todas las hojas están a la misma distancia de la raíz. No son binarios, es decir, cada nodo puede tener más de 2 hijos. En el caso de la memoria secundaria, cada nivel se corresponde con un nivel en la jerarquía.

Un caso particular de los B-Trees son los *Árboles 2-3-4*, donde cada nodo puede tener 2, 3 o 4 hijos. Por ejemplo, para los 3-nodos, la raíz contiene 2 valores (c_1 y c_2) y se cumple el siguiente invariante:

- El hijo izquierdo contiene todas las claves menores a c_1.
- El hijo medio contiene todas las claves entre c_1 y c_2.
- El hijo derecho contiene todas las claves mayores a c_2.

Para insertar un nuevo valor, se encuentra el nodo correspondiente. Si está saturado (es un 4-nodo), se *parte* en 2 nodos y una de las claves se inserta en el nodo de arriba, partiéndolo si es necesario, y así sucesivamente. Otra posible solución es partir preventivamente todos los 4-nodos que se encuentren en el camino de descenso de la clave, asegurando que en la inserción final habrá lugar en el nodo padre.

Volviendo a los árboles B, la funcionalidad es igual a de los 2-3-4, pero cada nodo puede tener hasta M hijos, y cuenta con un arreglo de $M-1$ claves. Para encontrar una clave, se debe realizar una búsqueda binaria para cada uno de los nodos. Luego, para utilizarlos en memoria secundaria, la idea es que cada nodo corresponda a una página de disco (incluyendo claves, significados y enlaces). En términos de complejidad, esta estructura puede realizar las operaciones de búsqueda, inserción y borrado en $\mathcal{O}(\log n)$ accesos de nodos, pero los algoritmos tienen cierta dificultad para implementarse.

10.2.3. Hashing dinámico

El *hashing dinámico* o *extensible* es un híbrido de los métodos anteriores: al igual que en los árboles B, los registros se almancenan en páginas que, al llenarse, se dividen en dos partes, mientras que también se mantiene un índice que se utiliza para encontrar la página que contiene los registros correspondientes a una clave, como en ISAM.

[6] Disco es el antecesor de las página en la jerarquía de memoria, no necesariamente en disco duro en sí.

Este método requiere contar con una serie de funciones de hash distintas $h_1, h_2, ...$ que tienen como imagen el conjunto $\{0, 1\}$. Inicialmente, se cuenta con dos páginas, y cada registro se inserta según el valor de h_1. Cuando alguna página se llena, esta se parte en 2, donde los elementos se dividen según la siguiente función de hash. Siempre y cuando el árbol de índices entre en memoria, la búsqueda solo requiere un solo acceso a página y la inserción 3 (la página original y las 2 nuevas).

Para evitar borrar y crear páginas frecuentemente, la partición se puede realizar cuando una página llega a cierto umbral de su capacidad (por ejemplo, 80 %), así como la fusión de páginas (cuando ambas tengan un largo combinado de 20 % de una página). Otra consideración son las fusiones de hash: para mantener un árbol medianamente balanceado, es ideal que cada hash tenga buena uniformidad sobre las claves.

11. Algoritmos probabilísticos

11.1. Definición

Un algoritmo *probabilístico* es uno que incorpora algún elemento de azar a su ejecución. Esto implica que no es determinístico: múltiples ejecuciones del algoritmo con la misma entrada pueden producir distintas salidas o tiempos de ejecución. En este caso, analizaremos algoritmos *Las Vegas*, que son aquellos que siempre dan un resultado correcto, pero cuyo tiempo de ejecución es una variable aleatoria (un ejemplo sería quicksort con elección de pivot aleatorio). Los *Montecarlo*, por otro lado, son aquellos que tienen alguna probabilidad de producir un resultado incorrecto.

11.2. Skip lists

Las *skip lists* son una representación de diccionarios basada en listas enlazadas ordenadas. Una implementación directa tendría una complejidad de búsqueda de peor caso de $\mathcal{O}(n)$: la lista solo puede ser recorrida secuencialmente, y si el elemento se encuentra al final, hay que recorrerla toda. Esto puede ser acelerado "salteando" elementos, que se logra agregando referencias a nodos posteriores. Por ejemplo, si todos los nodos en posiciones pares tienen referencias a la siguiente posición par, la búsqueda se puede implementar en $\left\lceil \frac{n}{2} \right\rceil + 1$ lookups.

Las skip lists *perfectas* extienden esta idea: cada 2^i-ésimo nodo posee una referencia al nodo 2^i posiciones más adelante en la lista. Luego, la búsqueda de una clave c se realiza análogamente a la búsqueda binaria: para cada nivel i, existen 2 posibles rangos en los que podría encontrarse la clave, y se determina en cuál comparándola con el elemento en el medio. Por ende, se examinan a lo sumo $\log_2(n)$, y el número total de referencias es solo el doble de una lista enlazada tradicional.

Sin embargo, estas estructuras son demasiado rígidas, ya que es muy costoso mantener el invariante a la hora de insertar o borrar claves. Las skip lists normales *randomizan* el requerimiento: al insertar un nodo, este es promocionado al nivel superior con una probabilidad de $\frac{1}{2}$ iterativamente (se frena cuando deja de ser promocionado o cuando llega al último nivel). Esto permite que la distribución de nodos en cada nivel sea equivalente a la de las skip lists perfectas, lo cual produce un costo de búsqueda promedio de $\mathcal{O}(\log n)$.

11.3. Splay trees

Si se conocen de antemano las *frecuencias de acceso* a cada clave de un diccionario, se puede construir un *ABB óptimo*, donde la altura de cada elemento (que es proporcional a su tiempo de

acceso) depende de su frecuencia esperada. Esto se puede realizar en tiempo $\mathcal{O}(n^2)$, pero resulta en una estructura demasiado rígida.

Por otro lado, los *splay trees* son ABBs auto-ajustantes que mueven cada clave accedida a la raíz (asumiendo que será accedida en el futuro cercano). Esto se logra a través de rotaciones parecidas a las de AVL, llamadas *splaying*:

- Si se accede a la raíz del árbol, no se hace nada.
- Si se accede a un hijo de la raíz, se realiza una rotación simple para ponerlo en la raíz.
- Si se accede a k, y su padre no es la raíz, hay 2 casos:

Rotación Zig-Zag

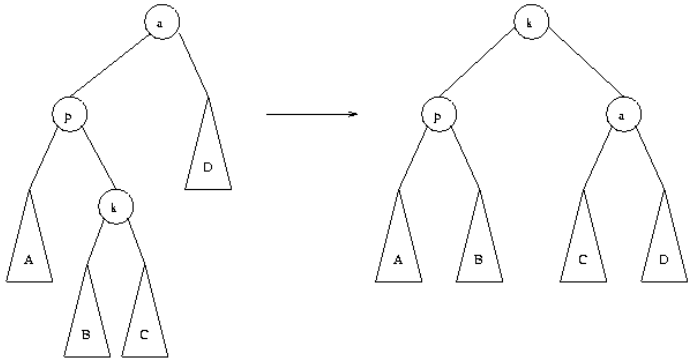


Diagrama de una rotación zig-zag.

Rotación Zig-Zig

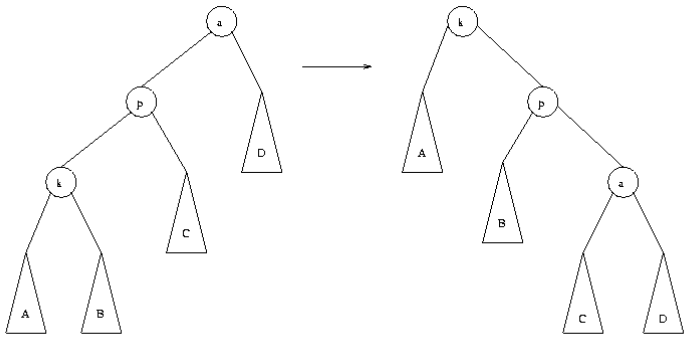


Diagrama de una rotación zig-zig.

Estas rotaciones se aplican hasta mover el elemento hasta la raíz.

La inserción se realiza como en cualquier otro ABB, solo que se realizan las operaciones de rotación para que el elemento quede en la raíz. Para borrar un elemento, primero se lo busca, lo cual lo deja en la raíz, luego se busca el mayor elemento del subárbol izquierdo, que pasa a ser la nueva raíz, como ese elemento era un antecesor inmediato, no tiene subárbol derecho, así que el subárbol derecho original queda como nuevo subárbol derecho.

Los splay trees resultan más simples de implementar que los árboles balanceados, y no tienen requerimientos de memoria adicionales. Si bien no se puede garantizar $\mathcal{O}(\log n)$ por operación, una secuencia de m operaciones tendrá tiempo $\mathcal{O}(\log m \log n)$ (tiene tiempo $\mathcal{O}(\log n)$ amortizado).

11.4. Análisis amortizado

El análisis *amortizado* permite asegurar garantías de la complejidad de secuencias de operaciones, en lugar de una única ejecución. Existen 2 mecanismos para demostrar complejidades amortizadas: el método del banquero y el método de potenciales.

A. Apéndice – Tablas